

Formal Aids for the Growth of Software Systems

Mathai Joseph

*Tata Research Development & Design Centre
Pune, India*

FM 2005, Newcastle, 20 July 2005

Automating the Software Process

Progress in engineering industries usually comes from automation of its processes.

Little of the software engineering process has been automated:

- **Requirements** collection - largely manual,
- **Software design** - manual,
- **Coding** - manual, and
- **Test generation** manual, test execution automated.

Mainly book-keeping of the process has been automated.

Challenges

Major deficiencies in software engineering:

- **Incomplete requirements**
- **Manual coding** takes too long
- **Platform dependence of the solution**
- **Inadequate testing**

Note also:

- ~30% of all software work is new development, but ...
- ~70% is 'maintenance'.

Can formal techniques play a role?

Formal Techniques – Then and Now

Areas listed in 1993 study* on the application of formal techniques (FTs):

reverse engineering, system certification, device software, nuclear plant shutdown systems, train protection systems, aircraft control systems, engine monitoring, processor design, medical applications, etc.

A similar study today would find few changes to this list.

Yet FTs have since been greatly extended, new FTs are being developed.

What is the goal?

* Craigen, Gerhart, Ralston

“Our ultimate aim, however, is not to perform small pilot projects that show that verification is sometimes feasible in an industrial setting ...

... our aim must be to integrate verification techniques into the software design cycle as a non-negotiable part of quality control.”

Gerard Holtzmann, FORTE94, October 1994.

Progress ... or Persistence?

Growth in problem complexity and improvements in FTs.

- e.g. for processor design, Moore's Law seems to apply to FT capability (e.g. model-checking 10^{20} – 10^{30} states).

Success is highest close to the hardware-software boundary.

Other applications are relatively untouched by FTs, e.g.

- System-level software
- Small, medium, large application development
- ...

Yet, use and understanding of FTs in chosen areas are growing.

Spread of FT

Many more conferences on FTs but little change in the distribution of papers; typically

Verification	~35%
New techniques	~35%
Reactive/real-time	~15%
Testing	~10%
Other	~ 5%

Few papers from industry, either stating problems or solving them.

Very few applications or realistic case studies.

Keep the Faith ... or Fall from Grace?

Concentrating on a few application areas limits the use of FTs.

- *Hesitation in **taking** FTs to software development*

matched by

- *Resistance from software developers to **using** FTs.*

It does not need many software crises to accept that this gulf is too wide.

FTs could be used to solve major problems in software development.

But it will need effort from both sides.

Directions

Choices:

- Develop new FTs to *illustrate* a prototype solution to a class of problems;

or

- Adapt FTs to enable industry-scale use.

Can FTs be used to **improve** today's software engineering methods, or only to **replace** them?

Which problems will benefit from use of FTs?

How can more software engineers start to use FTs?

Problems

There are problems to be solved throughout the software lifecycle. For example,

- **Requirements** collection - largely manual,
Use FTs for modelling requirements?
- **Software design** - manual,
Can component architectures, models be used?
- **Coding** - manual,
Can code generation be automated?
- **Test generation** - manual, test execution - automated.
Can specification-based and code-based test generation be used?

FTs in Industrial Practice?

Five rules for industrial use of a new technique:

1. It must **reduce time or effort** for a task,
2. It must **add measurable quality**,
3. It must **scale up** to project needs,
4. It must become part of a software engineering **method**,
5. It must be **usable reliably** and **repeatedly**.

*Few FTs meet these criteria; no FT meets even **most**.*

Adding Value

Requirements analysis

System design

Coding

Testing

I M P R O V E M E N T S

Action	Check consistency, missing details	Use models/ specification	Automate where possible	Automate <ul style="list-style-type: none"> • code review • test generation • test execution
Desired result	Improve accuracy save later re-work	Flexible, reliable architecture	<ul style="list-style-type: none"> • Re-use • Platform independence 	Improve coverage, save time
Formal technique	Formal specn, Model-checking		Transformations, correctness	Program analysis

In this talk I will focus on:

- the design-code-test part of the development cycle;
- problems in software maintenance.

The two topics are closely related.

Practical Software Development

In typical large-scale software development :

- Requirements *evolve* over time;
- *New requirements* keep appearing.

... sometimes described as the '*spiral method*' of software development.

After an *acceptance test*, the software is delivered to the client.

Software Development

Requirements analysis



Software specification



Software construction



Software testing



Software Development

Requirements analysis



Software specification



Software construction



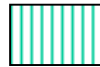
Software testing



What is a good acceptance point?

Software Development

Requirements analysis



Software specification



...

Software construction



...

Software testing



...

Development & Testing

Maintenance

Acceptance

Acceptance could be at many different points

Development Projects

Problems

- Manual programming can be slow (60% of total time).
- Coding errors take time to remedy:
 - manual review (6%) + testing (25%).
- Very little re-use: developers like to write new code.
- Code has some or many platform dependencies.
- Re-targeting code is hard.

Possible solution?

Automate software development, code generation.

Target 1

Automate code generation from UML models

- Generate bug-free code
- Ensure complete platform independence of application
- High execution efficiency needed
- Coding productivity to increase substantially

Automated Development from Models

TCS has used model-driven code generation for large projects for ~8 years: major improvements in productivity and reduction in errors.

Example: Team of 60 people in 6 months produced:

- 180K lines of operation description
- 1200 UML classes

MasterCraft automatically generated

- 5M lines of C++ for (IBM) mainframe
- 1.5M lines of C++ for middle (Sun) and front-end (Windows)
- average of 250 lines produced, tested per programmer day

Almost no rework of requirements during coding; few errors.

60-70 systems built with *MasterCraft*, in production use.

Assessment of Target 1

Automated code generation:

- Generates bug-free code of uniform quality
- Ensures platform independence
- High execution efficiency
- Coding productivity up 10-20 times

- **But**
 - *Difficult learning curve*
 - *Rigid process: no code produced until modeling completed*
 - *Model-based code size may be larger*
 - *Changes made in models, not in code – can be frustrating*
 - *Strong resistance from programmers – lack of choice & control*

Target 2

Keep advantages of model-driven code generation, but with programmer-driven flexibility.

Keep

- Automated code generation
- High productivity
- Platform independence, etc.

Bring in

- *Graded learning curve*
- *Customizability of toolset*
- *Automated test generation*
- *Application maintenance at program level*

Decisions

- Allow manual **design**,
- **Hide detail** by abstracting common operations into generic code stereotypes,
- **Reduce** amount of **hand code**.

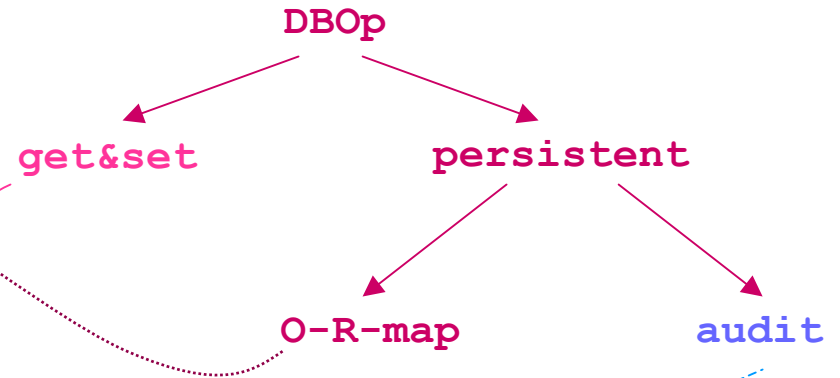
Generic Code Patterns

Programmer builds code using tags, e.g:

```

/@O-R-Map, @audit
/@get&set
Class DBUpdate;
{ /@primarykey
  int a;
  float b;
  . . .
}
    
```

Tag Tree



Generated Code

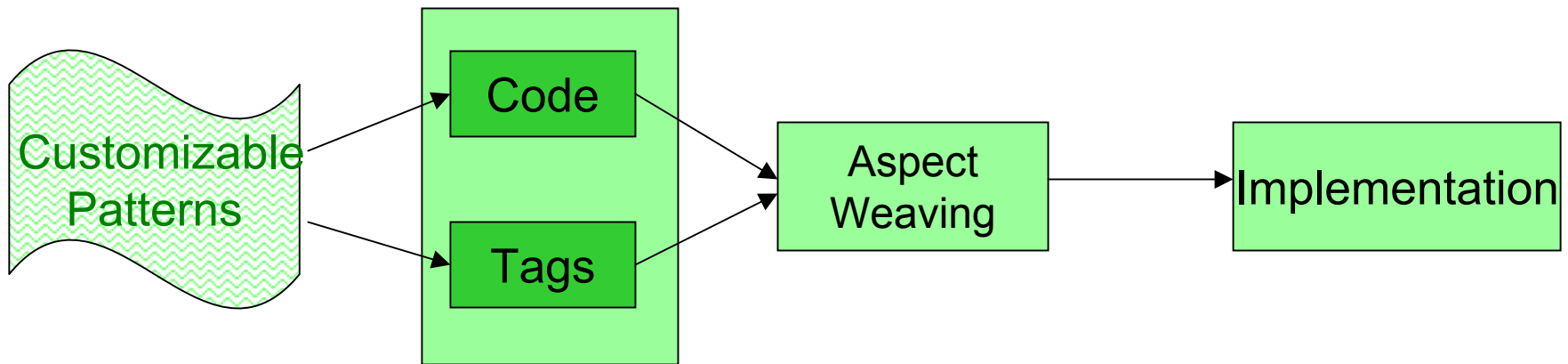
Tags are transformed into code

```

Class DBUpdate
{
    int a; float b;
    Bitvec bitvec;
    int geta(){return a;}
    void seta(int p_a){a=p_a;marka()}
    void marka()
        {bitvec.setbit(bit_a,TRUE);}
    void unmarka()
        {bitvec.setbit(bit_a,FALSE);}
        similarly for b
    Image getImage() {}
    void setPreImage(Image p) {}
    void getPostImage(Image p) {}
    void create(){
        setPreImage(getImage());
        // create code due to O-R-Map
        setPostImage(getImage());
    }
        similarly for update(),delete()
    ...
}

```

`/@O-R-Map, @audit`
`/@get&set`
Class DBUpdate;
{ `/@primarykey`
 `int a; float b;`
 `. . .`
}



MasterCraft Lite
-- similar to Xdoclet
Metadata with Aspects
EJB 3.0, JBOSS

Formal Support

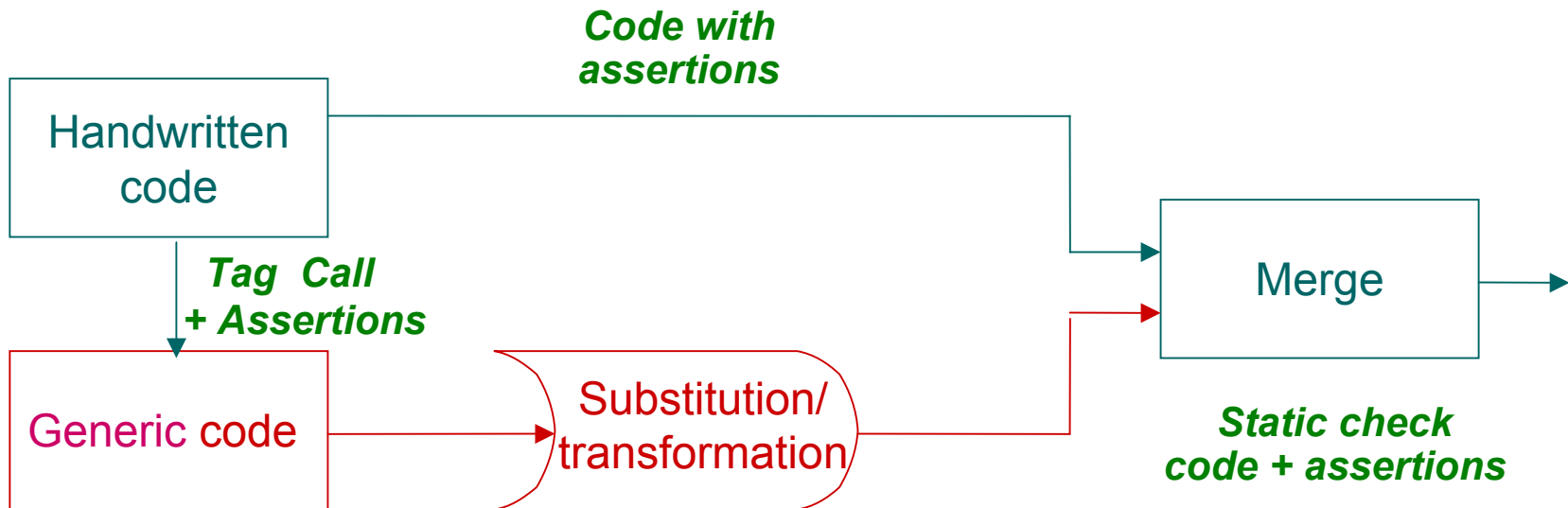
Generic tag code needs:

1. Transformable assertions to preserve properties while weaving with hand code;
2. Pre- and post-conditions over operations.

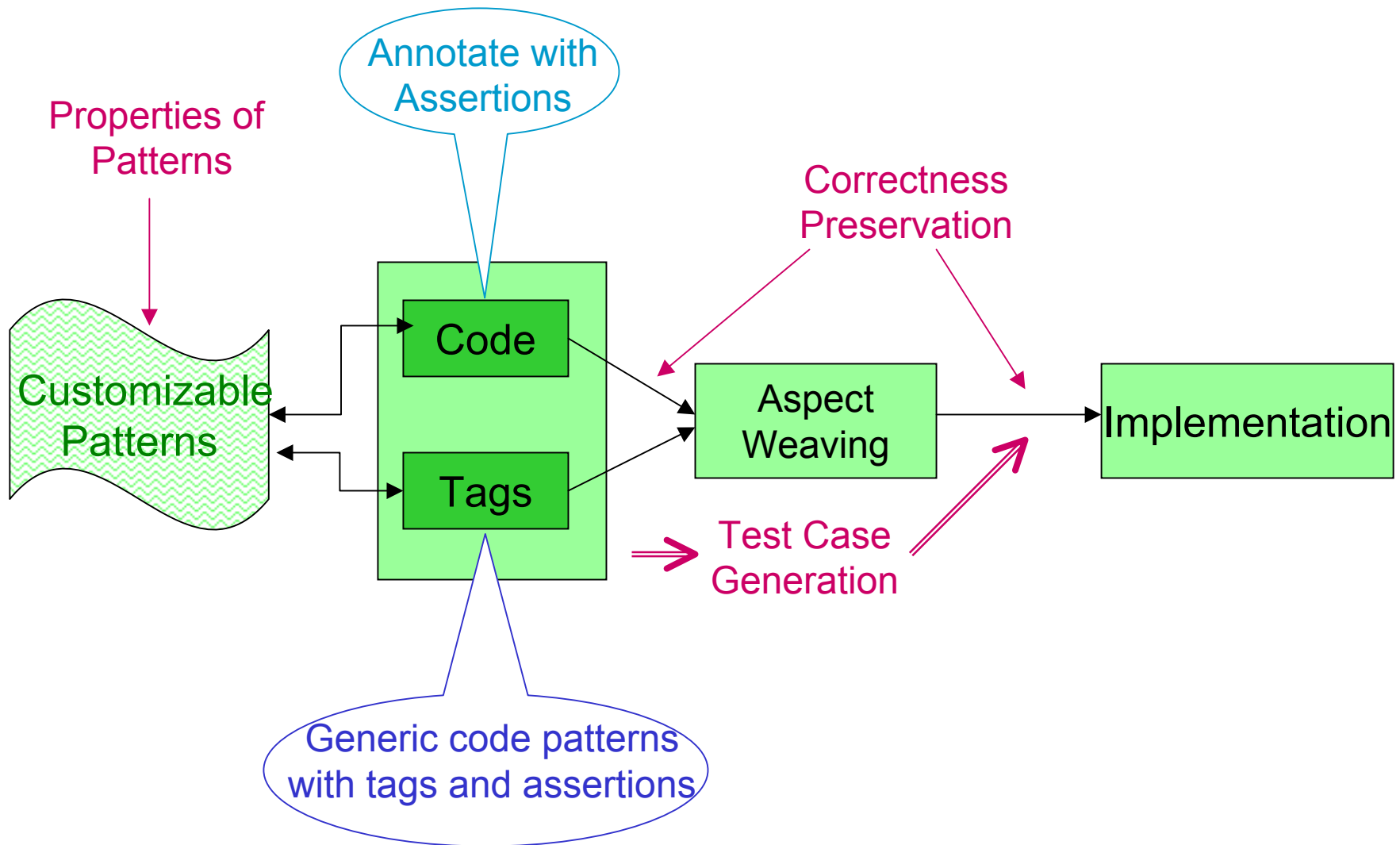
Assertions and tags code in the tag repository -- must be checked when woven with hand code.

Assertions in hand code are desirable (but not enforceable!). However, checks in generic code will make it much easier to detect bugs in hand code.

Formal Aids for the Growth of Software Systems



Formal Aids for the Growth of Software Systems



Formal Support

Formal support to a development process aims to:

- be unobtrusive
- be seen as effective
- provide guidance related to program steps

```
e.g. 'Precondition not met for tag @get&set at line 37  
    -- expression a+b<10 not satisfied'
```

Advantage

- checking in step with development, not *post facto*

Weakness

- Only some safety properties can be checked

Checks

Assume a high degree of automated assistance for program construction, weak verification at each step

Testing still needs to be done:

- Automated verification reduces the time taken for testing.
- Verification should lead to fewer & more specialized tests.
- Testing *is* needed before any practical deployment.

Bugs detected in testing typically from limited hand code.

Static Analysis & Testing

Assertions over hand code and patterns support static analysis (verification) of properties.

They also provide the basis for test case generation.

Automating Testing (An Example)

Testify

- Produces test sequences and state specifications
- Supports manual specification
- Generates initial state and inputs for each step
- Integrates into development frameworks
- Interfaces with test execution tools

Sample of improvements:

<i>Size</i>	<i>Tests</i>	<i>New defects detected</i>
Small	100	40
Large	523	116

Software Maintenance

Post-acceptance is the *maintenance* period.

Maintenance activities are:

- Similar to those in development, but
- Emphasis on *supporting* and *growing* an operational software system.

Typically, new code is added during maintenance but code is never removed.

Software Maintenance

Software maintenance consists four main activities:

- **Remedial** – correcting errors 21%
- **Enhancing** – adding new features 50%
- **Adaptive** – changes in the environment 25%
- **Improving** – making software more robust 4%

These activities may be interleaved.

Note: most maintenance work is not bug-fixing.

* <http://fox.wikis.com/wc.dll?Wiki~SoftwareMaintenance~SoftwareEng>

Problems in Maintenance

- Software usually maintained by a new team, not the developers;
- Language and platform may be uncommon.

Problems:

- Limited software documentation;
- Specialized platform choices;
- Regular enhancements needed;
- Bugs to be checked, corrected, tested;
- Long regression test cycles, even for small changes.

Maintenance

Major needs:

- Improve understanding of the program
- Impact analysis of a change;
- Rapid testing of enhancements and corrections.
- Minimize regression testing: test only the impact of a change.

Formal support

- How can a *change* be represented and analyzed?
 - Incremental analysis?
 - Incremental verification?
- Can verification keep in step with proposed changes, with low incremental overhead?

Re-engineering

An application will remain in use until the risks become unacceptable:

- **Language** becomes obsolete, programmers hard to find;
- **Platform support** reduces, is expected to end;
- **Architecture** (e.g. batch processing, client-server) limits application growth.

Abstracting *pre-*, *post-*conditions would allow easier business rule extraction.

Formal approaches to these problems are largely unexplored.

SE *and* FM *or* SE *with* FM ?

Many opportunities for bringing FTs into in large-scale software engineering.

Level of automation in development is increasing, variability is decreasing, adding formal analysis becomes feasible.

Hoare's Verifying Compiler is a challenge that seeks to go far beyond this in a 10-15 year timescale.

Steps in that direction can be taken now.

Science teaches us to:

- study a problem in isolation,
- find a solution,
- experiment if the solution applies more widely.

Practice demands that:

- the solution applies where the problem arises.

We can choose our weapons but not the battlefield.

There is a great opportunity to make formal methods **improve** software development, rather than tackling problems in isolation.

Formal techniques do not make programming difficult:

“the difficulty has always been there, and only by making it more visible can we hope to design programs with a high confidence level”

Dijkstra, 1976.

Claim: Engineering methods can today be used with FTs to:

- reduce level of *detail* seen by the developer,
- automate solutions to many repetitive problems,
- make verification *part* of development.

Acknowledgements

Thanks to

- Sushma Mohanan and her team for the data analysis,
- Vinay Kulkarni for the work on MasterCraft Lite,
- Ashok Sreenivas for the work on Testify,

and many others in TRDDC for their comments and suggestions.

Thank You