FMCO 2002

Playing games with UML tools

Perdita Stevens

incorporating early joint work with

Jennifer Tenzer

Division of informatics

University of Edinburgh

http://www.dcs.ed.ac.uk/home/pxs

Grand Challenge

Put a computer on the software design team within 20 years.

How?

Why?

Manifesto

This talk is about some recent developments in

- how design is seen in the modern development process,
- how tools support it, and
- how future tools could support software designers better.

Design of software

(For the purposes of this talk)

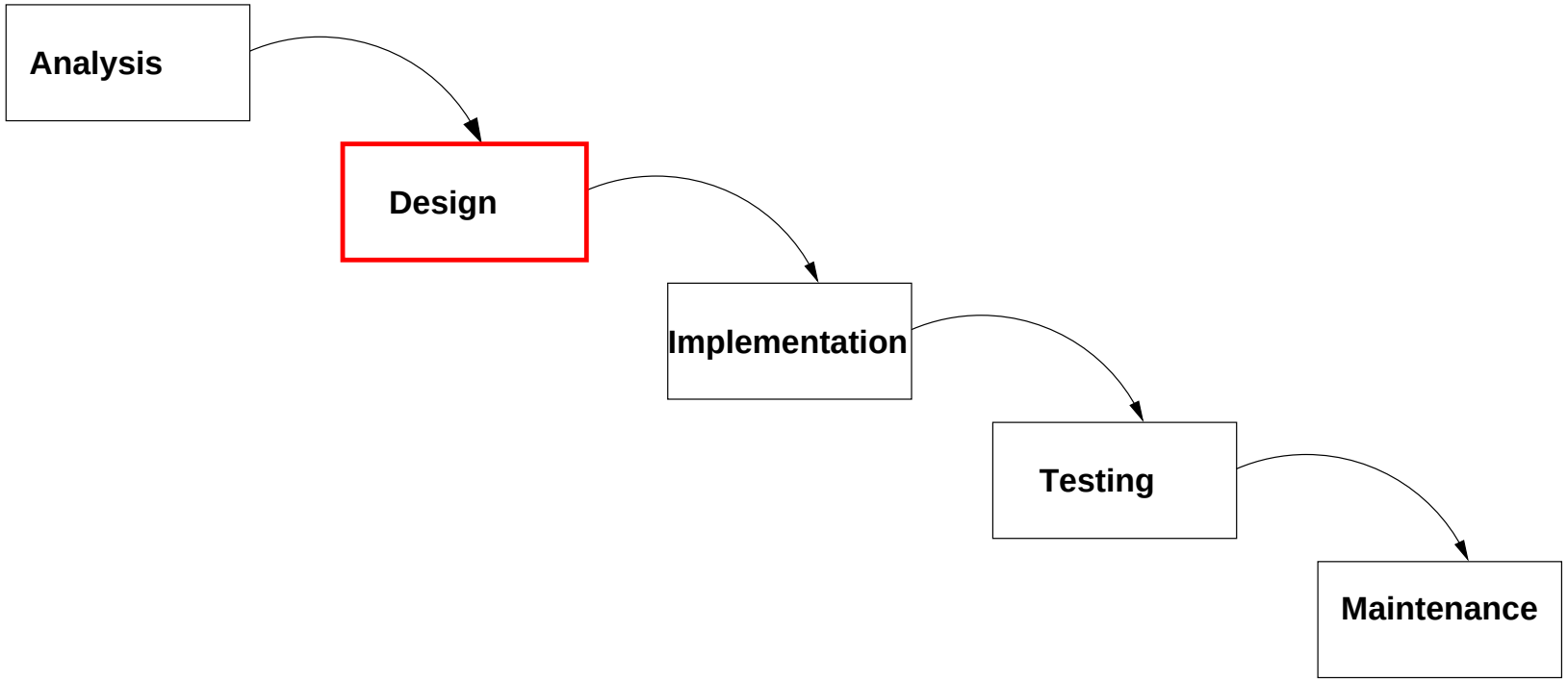
A problem is a *design problem* if

- it abstracts away from the code level
- and it can rely on the requirements being understood

NB I'm mostly interested in (and experienced in) mainstream business software.

In the olden days

we had the waterfall process



In modern times

we understand a lot more about what design is *not*

- not started after analysis is complete
- not done in one fell swoop

But what is it? And how is it now seen within development processes?

Donald Schön writes

“Designers put things together and bring new things into being, dealing in the process with many variables and constraints, some initially known and some discovered through designing. Almost always, designers’ moves have consequences other than those intended for them. Designers juggle variables, reconcile conflicting values, and maneuver around constraints – a process in which, although some design products may be superior to others, there are no unique right answers.”

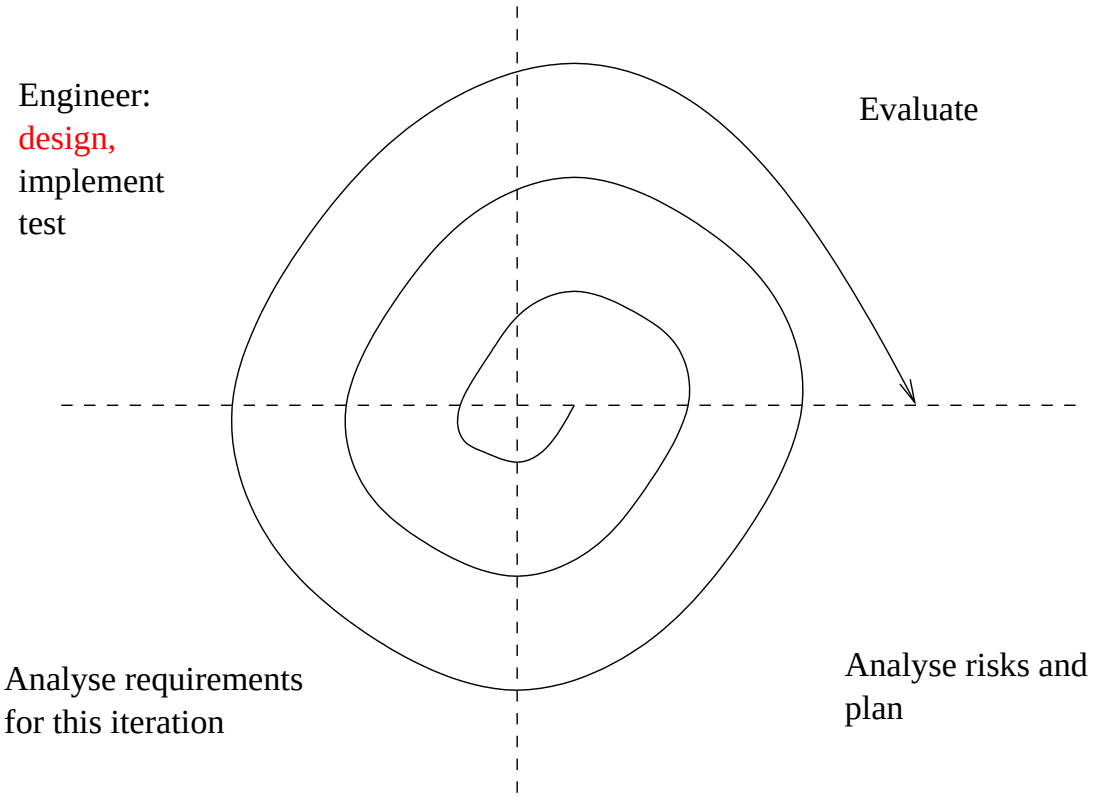
Changes in development process

In software, it was gradually realised that the waterfall process was falsifying real life.

We got variants of the waterfall with backwards arrows...

... but often little idea about what they really meant.

80s Boehm's spiral model



80s/early 90s: OO as silver bullet

Method wars: lots of OODMs, each with its own modelling language.

Improving understanding of [OO]A/[OO]D; insights like

- “gestalt round trip design” (Booch)
- responsibility driven design (Wirfs-Brock)
- design by contract (Meyer)
- use case driven design (Jacobson)

Control of process and meta-process.

late 90s/2000s

emergence of *Unified Modelling Language*

In (mainstream OO) development processes, we got a bifurcation

- (Rational) Unified Process
- “agile methodologies” like XP

Initially seen as opposites, later convergence (at least from one side!)

because it takes both rigour and agility to deliver high quality software consistently.

We know how to do this in easy cases only.

Parallel strand: reuse

Effective reuse has long been the holy grail of software engineering.

We have come

from

naive expectations of a future with “software ICs”

to

a more *mature* understanding of just how hard the problem is

and an appreciation of how important *architecture* is

Architecture and reuse

Most components are just not reusable via simple interfaces in arbitrary contexts – that was a pipe dream.

Instead we can aim for

appropriate libraries of (usually simple) *general purpose components*

complementing

domain specific components, frameworks and *product line architectures*

(And in another context but with similar motivations, *coordination?*)

Frameworks, PLAs and design

Currently, the *only* practical way to design frameworks and PLAs is by abstraction from several successful examples.

Huge pressures to design the framework/PLA first, though. Often tried; seldom works, because

People and paper are not effective at working out the implications of a range of permissible instantiations

Design under uncertainty

More generally, design is hard when you have to abstract over

- future decisions
- environments
- execution orders
- ...

or especially, several of these!

So we need tools: but not just verification tools.

Software design and tools

UML has enabled competition in the design tools market. There are many UML design tools, often costing thousands of dollars per seat.

$64K Q:*Do they help people do design?*

More convincingly, they help with the recording of design.

Real design happens at the whiteboard.

Tool use for recording design

Characteristics of environment:

- usually single person
- working out detail, looking for unforeseen problems
- developing diagrams to go in documents for unknown future readers

Precision and consistency important.

Will revert to whiteboard if hard problems arise.

So far in this talk

Have argued:

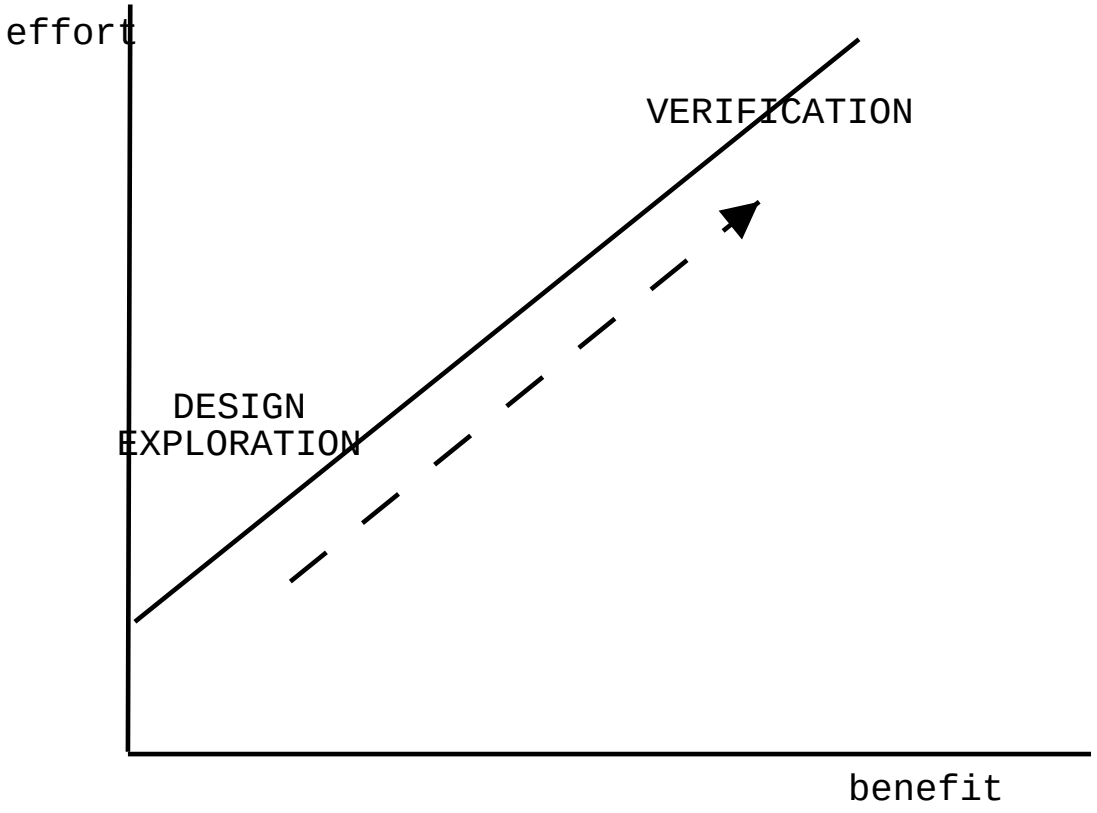
- design has to be iterative and exploratory, for social reasons
- there are also technical problems that humans find hard
- today's tools are not helping (much).

Could tools help more?

- They must be usable by several people at whiteboard e.g. Knight tool (DAIMI, DK)
- They must be well integrated with code (e.g. Together) and refactorings of it (not yet)
- They must actively help designers to make, check and record good design choices

  e.g. using existing verification and debugging techniques, collaborating with modellers to add detail, hypothesise, explore...

Abolish verification/exploration gap



Vision: design as collaborative game

You have a partial design in your tool:

- a *class diagram* – rough idea of classes and relationships
- some *dynamic diagrams*, statecharts and interaction diagrams
- some draft *contracts and assumptions*, in OCL or whatever

Tool helps identify dependencies, takes you to inconsistent points, offers alternatives, keeps track of consequences...

Aim to amplify the designers' own skills – not replace them.

So where do games come in?

Informally, design can be seen as a game, or as a (playful) argument.

People stand around, make suggestions, object to one another's suggestions...

What would it mean to turn this view into a framework for tools?

Would it be useful?

Formal games

There’s a long history of using two-player alternating games in computer science, going back to early model theory and beyond.

Let one example suffice...

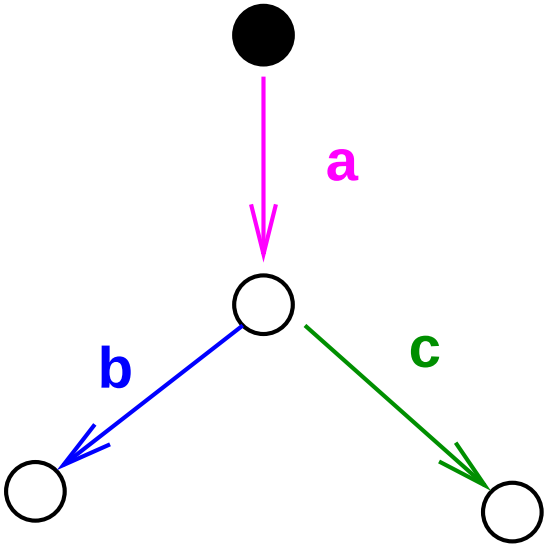
Bisimulation game

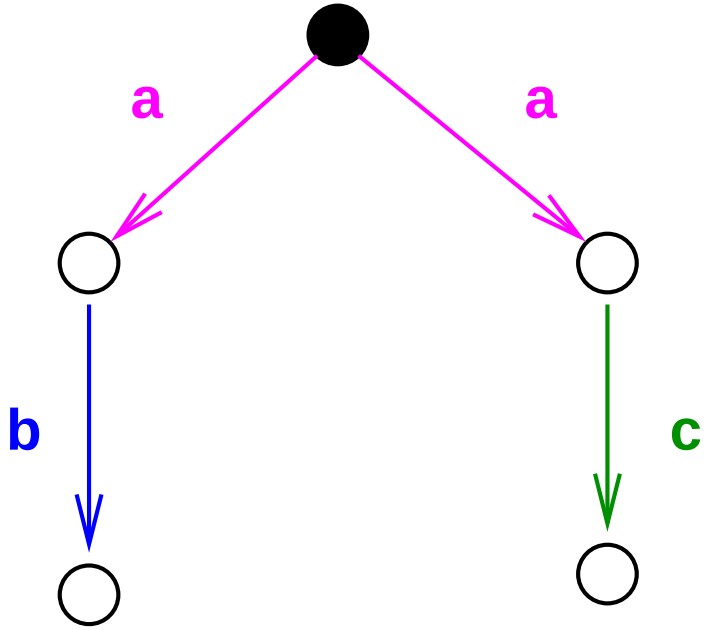
Two processes, two players. Verifier claims they're bisimilar; Refuter, that they are not.

- Refuter challenges: picks a transition from either process
- Verifier has to respond from the other process.
- Refuter challenges again (in either process)...

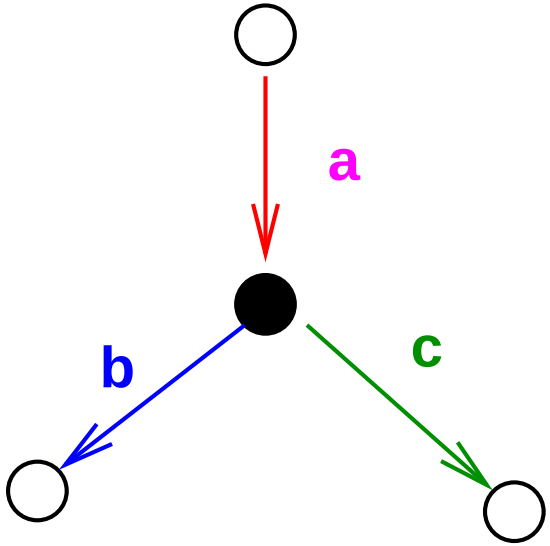
Etc. Each player wins if the other can't go. If play continues for ever, Verifier wins.

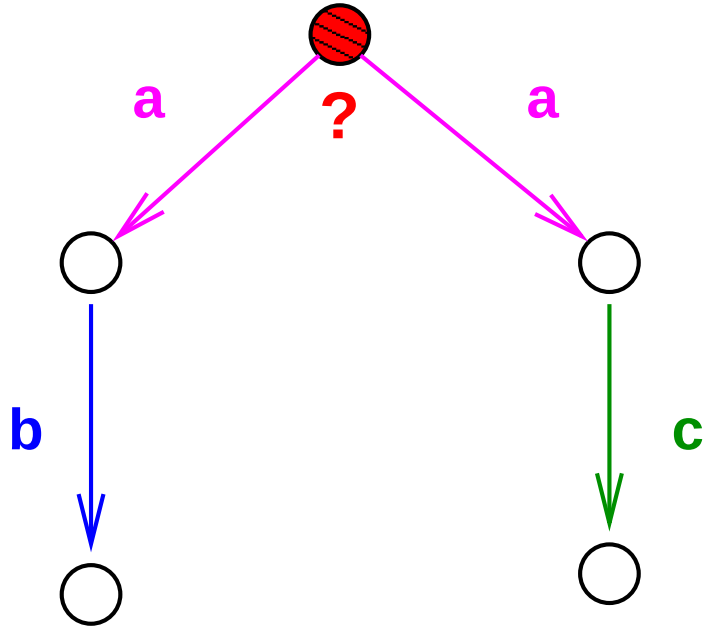
Example: the start



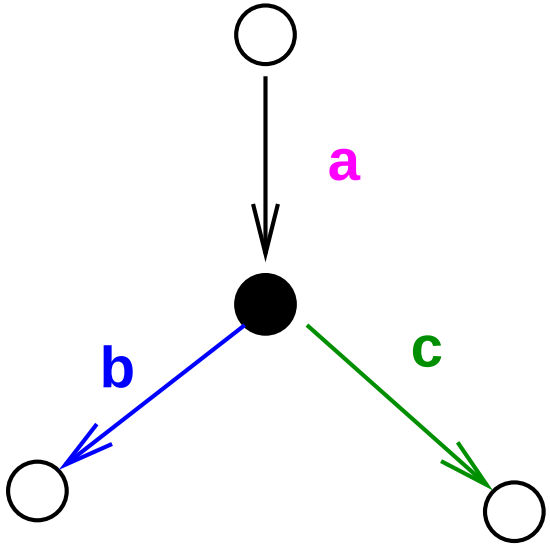


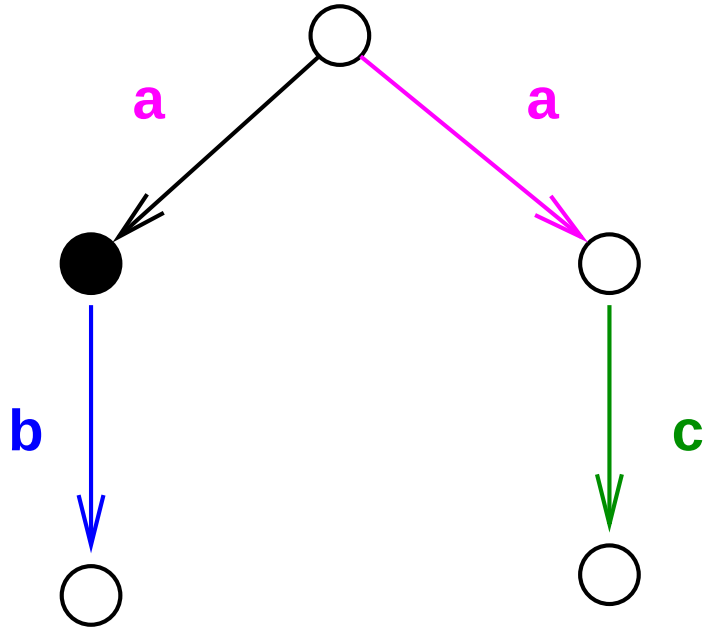
Example: after Refuter's first challenge



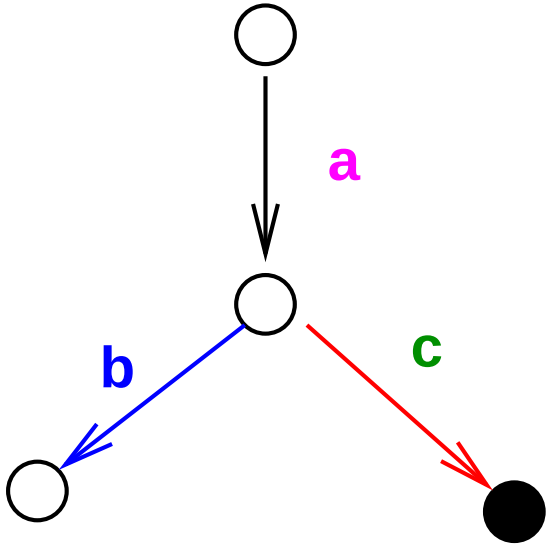


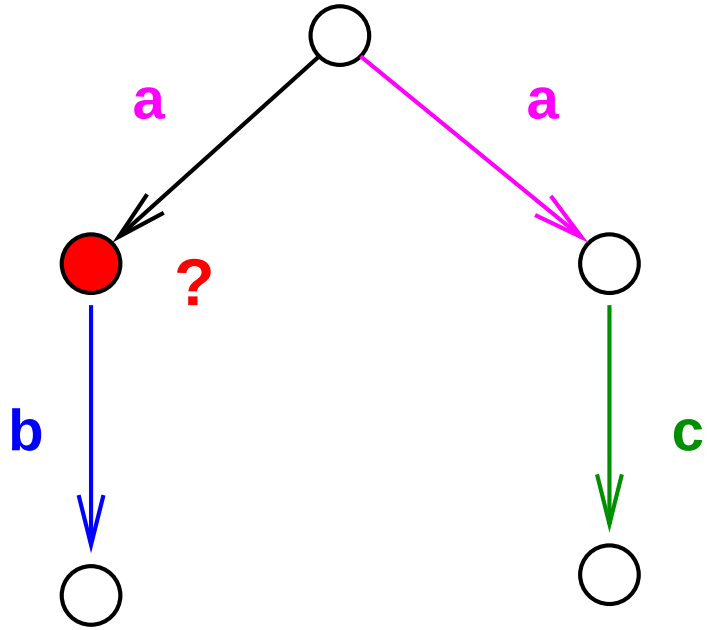
Example: after Verifier's reply





Example: after Refuter's second challenge





The game captures bisimulation

in the sense that it’s an easy theorem that Verifier has a winning strategy if and only if the processes are bisimilar

and in that case her winning strategy “is” a bisimulation

Games are useful in verification tools

(e.g., the Edinburgh Concurrency Workbench) because:

- All verification problems can be coded as games
- there are uniform algorithms for finding winning strategies
- problems with value-passing and some other kinds of infinite state spaces are (often) OK via abstraction techniques
- can give feedback to the user via games

NB “just” a way of thinking: but helps us ignore what’s irrelevant

Beyond verification

Traditional path of virtue:

1. define a specification for a system
2. define a design that meets the specification

But this is never, ever, what's done in real life on any non-trivial problem - *even when the end result is that both specification and design exist*.

This is not because practitioners are negligent.

Different specifications, same idea

In concurrency, we often design a system (e.g. in CCS) and want to verify it, by either:

1. defining a second process that should stand in some relation with our system; or
2. defining a logical formula that our system should satisfy

Both can be expressed as verification games: and it will be *essentially the same game*.

Arguably the game is closer to capturing our intentions than either specification. Can we use this idea?

“Loose” games

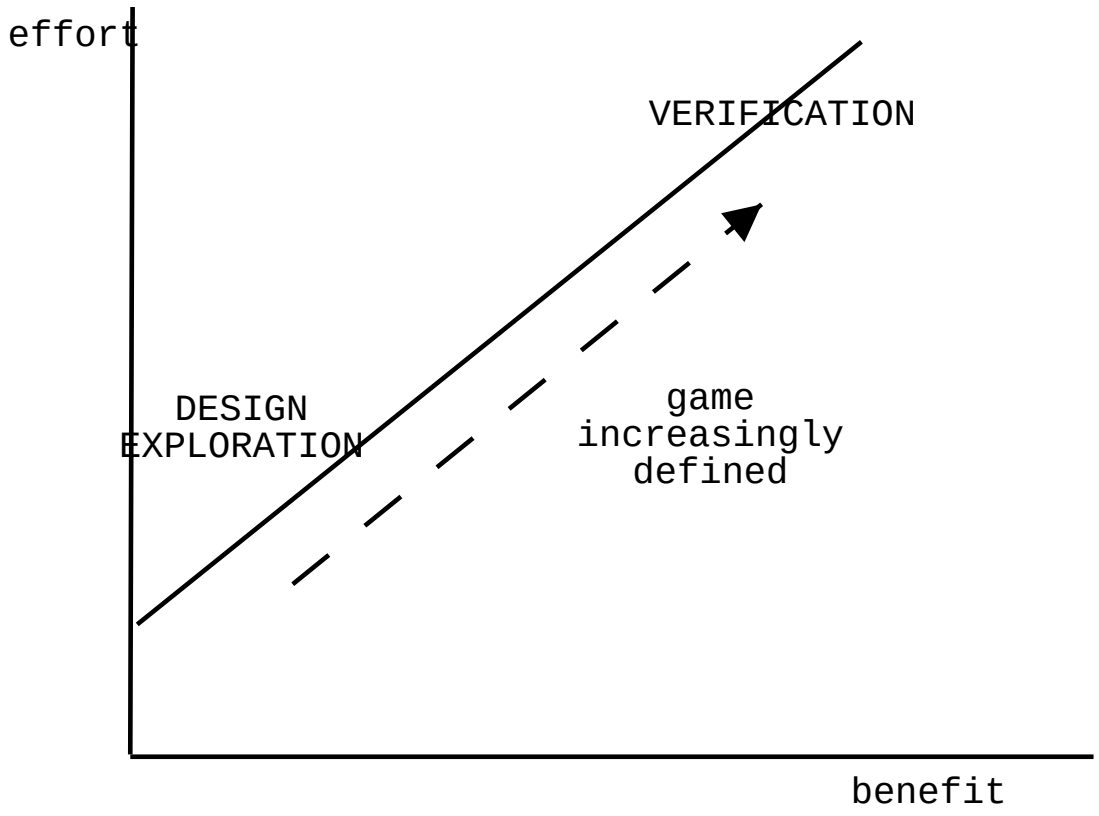
In verification games, the tool is omniscient and the user can only lose! The game is predefined and fixed.

But strategies can be built incrementally. Why not games too?

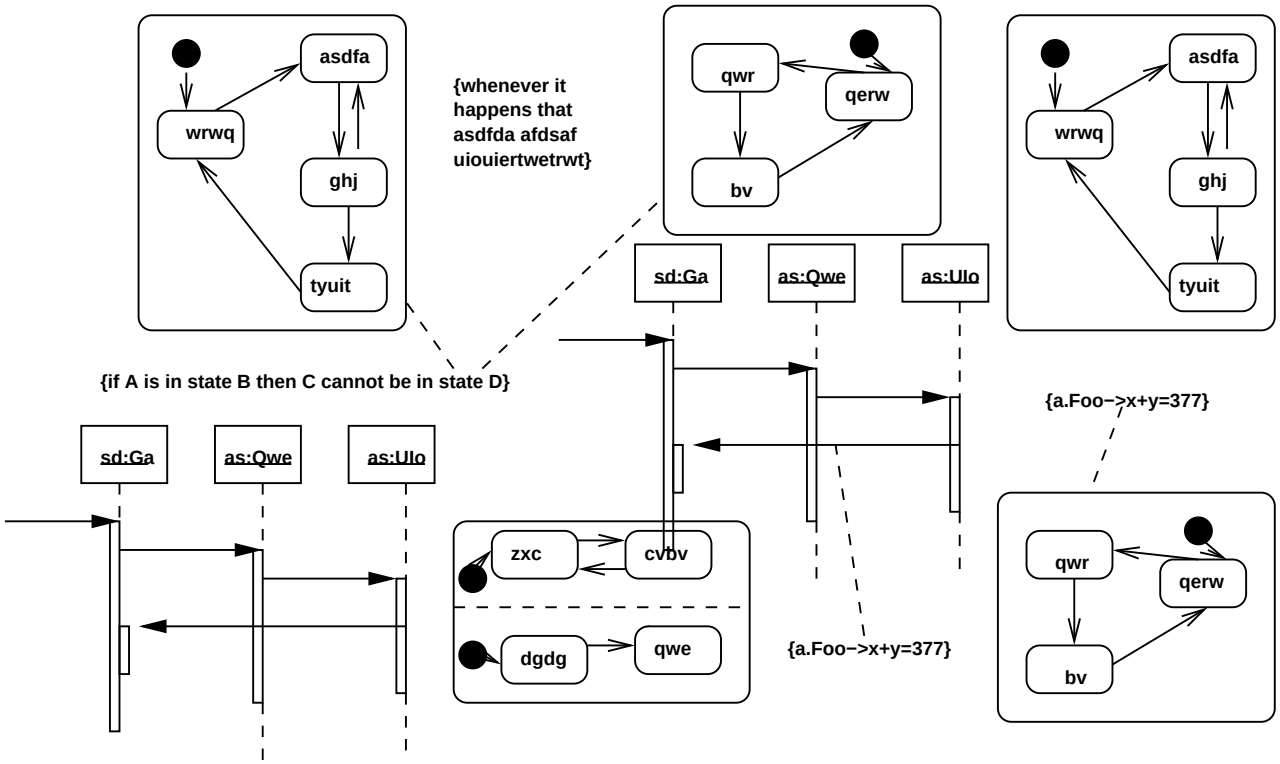
Can we exploit this to use games differently in software design?

- gradually define and explore a game
- at the end, verify the “complete” game

Incremental definition of games



Example situation



Design problem

A correct system model

- will be consistent (i.e. correct UML)
- and will satisfy certain properties

but it's unreasonable to expect user to draw a full set of diagrams and write down all the properties up front!

Can the tool do the hard work?

To define a game we need

1. a starting position

   - e.g. a system configuration we want to consider

2. definition of legal moves

   - for Refuter: what challenges are reasonable?

   - for Verifier: e.g. what does underspecification mean?

3. winning conditions

   - e.g. bad configurations or actions of the system (Refuter wins)

(Contrast model-checking games, where the user must write a formula in a logic.)

Tool as game wizard

The user builds the game, which captures what it is for this system to be correct, in parallel with building the system model itself.

The tool offers *guidance* for building the game, just as it does for building the model.

The tool allows the user to explore the consequences of the system and game decisions so far, offering a spectrum of interaction

fully manual play $\longleftrightarrow$ fully automatic analysis

Preliminary example

Refuter = environment vs

Verifier = team, one per object (CRC cards on steroids)

Work with:

- a given collaboration

- modifiable collection of protocol state machines, one per class

- sequence diagram, maybe incomplete, recording play

- modifiable constraints

Aim is to build a game for which Verifier will have a winning strategy iff the model is consistent and “complete enough” (...)

Legal moves

Refuter:

- chooses initial states of objects

- makes initial challenge

- resolves non-determinism

Verifier (member):

- may be challenged by receiving an event, e.g. operation invocation or return (respecting stack)

- may react by changing state and/or sending an event

- may change model, *but* may not win a play where this is done

Winning

Either player wins if the other can't/won't play.

Refuter wins if a constraint is violated.

Refuter wins if a PSM transition is violated.

Refuter wins all plays in which a Verifier changed the model.

Otherwise, Verifier wins all infinite plays.

Variants (making it harder for Verifier): record each class's full reaction to each message e.g. using MSMs (TS02).

Potential for frameworks/PLAs

The main problem of these parameterised designs is that

Exploration is important and hard where much is intrinsically unknown.

Many hard problems remain: e.g., the need to pin down the limits of permissible instantiations.

Role of the computer

We have fuzzified the role of the computer: it may be at different times

- game-building partner
- referee
- omniscient opponent

“Game” may sometimes be a metaphor.

Role of humans

Variants worth exploring:

- human as verifier and/or refuter (old and powerful!)
- human as responsible for particular component (CRC on steroids?)
- human as responsible for particular characteristic (cf AOD?)

Perhaps playful tools could make it easier for a lone designer to solve problems?

Conclusion

We have a much better understanding of what software design is than we did a few decades ago.

UML enables competition and cooperation between tools and techniques.

Tools should be able to help in design much more than they do.

But to make this happen, we will have to think differently about the interaction between tools and their users.

Advertisements

UML San Francisco Oct 20-24

Submission dates: 8 Apr (abstracts) 15 Apr (full/short papers)

FMOODS Paris Nov 3-7 (Uwe Nestmann & PS)

Submission dates (provisional): 1 Jun (abstracts) 8 Jun (papers)

PITPAT Eindhoven Jun 30 - Jul 4 (Arend Rensink & PS)