Model-Driven Development: Its Essence and Opportunities

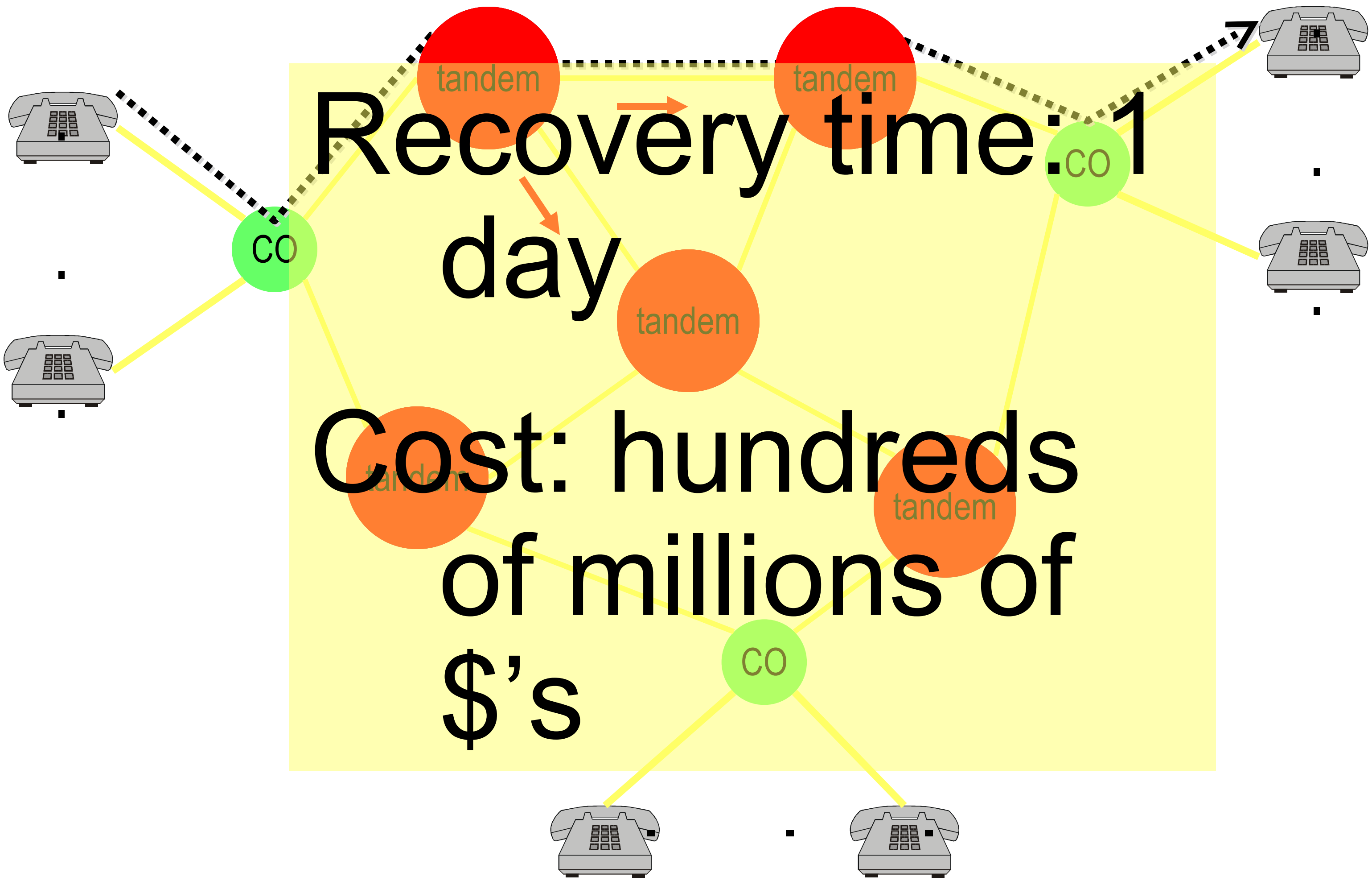
-- A Melodrama in 3 Parts --

Bran Selic
President, Malina Software Corp. and Carleton U.
selic@acm.org

PART I: ON PRESENT-DAY SOFTWARE DEVELOPMENT TECHNOLOGIES

The Anatomy of an Engineering Disaster

- 1990: AT&T Long Distance Network (Northeastern US)



Recovery time: 1 day

Cost: hundreds of millions of $'s

The Root Cause

- Missing "break" statement in a software module
  - one (missing) line among millions

```
. . .;
switch ( . . .
    case a : . . .;
         break;
    case b : . . .;
         break;
         . . .
    case m : . . .;
    case n : . . .;
         . . .
};
```

Recovery time: 1 day

Cost: hundreds of millions of $'s

Execution "fell through" unintentionally into the next case

Our Enemy: Complexity

- Many modern software systems are reaching levels of complexity encountered in biological systems
  - Systems of systems each of which may include tens of millions of lines of code
  - …any one of which might be the culprit that brings down the entire system
- Furthermore, we can only see an increase in this complexity due:
  - Growing demand for greater and more sophisticated functionality
  - Increasing interaction with the implacable complexity of the real world
- Given our current track record, how will we cope with this rise in complexity?

Fred Brooks on Complexity

- [From: F. Brooks, "*The Mythical Man-Month*", Addison Wesley, 1995]
- *Essential* complexity
  - inherent to the problem
  - cannot be sidestepped or eliminated by technology or method
  - e.g., the computational complexity of the "traveling salesman" problem
- *Accidental* complexity
  - due to the use of inappropriate technologies or methods
  - e.g., building a skyscraper using only hand tools

A Bit of Modern Software

```
SC_MODULE(producer)
{
sc_outmaster<int> out1;
sc_in<bool> start; // kick-start
void generate_data ()
{
for(int i =0; i <10; i++) {
out1 =i ; //to invoke slave;}
}
SC_CTOR(producer)
{
SC_METHOD(generate_data);
sensitive << start;}};
SC_MODULE(consumer)
{
sc_inslave<int> in1;
int sum; // state variable
void accumulate (){
sum += in1;
cout << "Sum = " << sum << endl;}
SC_CTOR(consumer)
{
SC_SLAVE(accumulate, in1);
sum = 0; // initialize
};
SC_MODULE(top) // container
{
producer *A1;
consumer *B1;
sc_link_mp<int> link1;
SC_CTOR(top)
{
A1 = new producer("A1");
A1.out1(link1);
B1 = new consumer("B1");
B1.in1(link1);}};
```

Can you see what this program does?

On Mainstream Programming Languages

- *Most mainstream programming languages abound in accidental complexity*
  - Syntactic overload, goto statements, misaligned pointers, uninitialized variables, etc.
- These languages are:
  - Difficult to understand
    - Require significant intellectual effort to master
  - Defect intolerant, with a chaotic quality:
    - The effects of seemingly minute and almost undetectable defects cannot be predicted, but could be catastrophic
- (The embarrassing bit) *Yet, we have persistently held on to these outdated technological paradigms, investing enormous financial and intellectual resources in improving it*
  - ..at the cost of overlooking many new and better approaches

The Impact

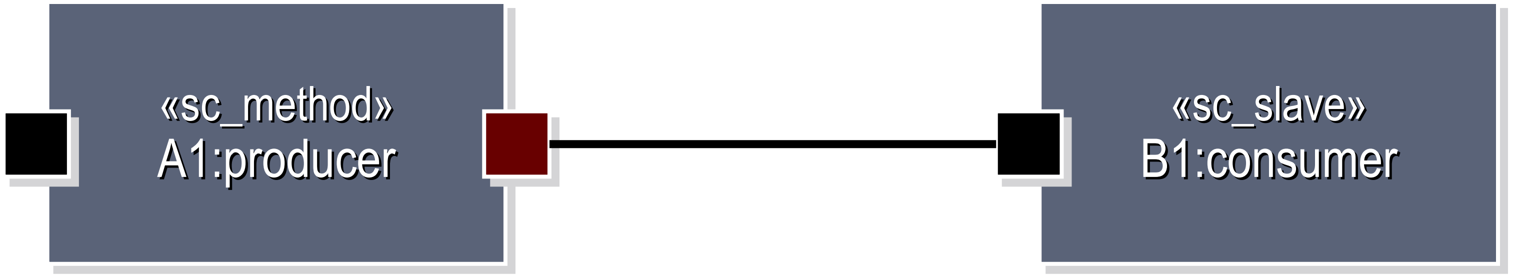
- Abstraction (modeling) of programs is difficult and risky
  - Any detail can be critical!
  - Eliminates our most effective means for managing complexity
- Our ability to exploit formal mathematical methods is severely impeded
  - Mathematics is at the core of all successful modern engineering
  - Mathematical methods depend on abstraction to avoid state explosion problems
- We are also seriously underutilizing the automation potential provided by computing technology

A Bit of Modern Software

```
SC_MODULE(producer)
{
sc_outmaster<int> out1;
sc_in<bool> start; // kick-start
void generate_data ()
{
for(int i =0; i <10; i++) {
out1 =i ; //to invoke slave;}
}
SC_CTOR(producer)
{
SC_METHOD(generate_data);
sensitive << start;}};
SC_MODULE(consumer)
{
sc_inslave<int> in1;
int sum; // state variable
void accumulate (){
sum += in1;
cout << "Sum = " << sum << endl;}
SC_CTOR(consumer)
{
SC_SLAVE(accumulate, in1);
sum = 0; // initialize
};
SC_MODULE(top) // container
{
producer *A1;
consumer *B1;
sc_link_mp<int> link1;
SC_CTOR(top)
{
A1 = new producer("A1");
A1.out1(link1);
B1 = new consumer("B1");
B1.in1(link1);}};
```

Can you see what this program does?

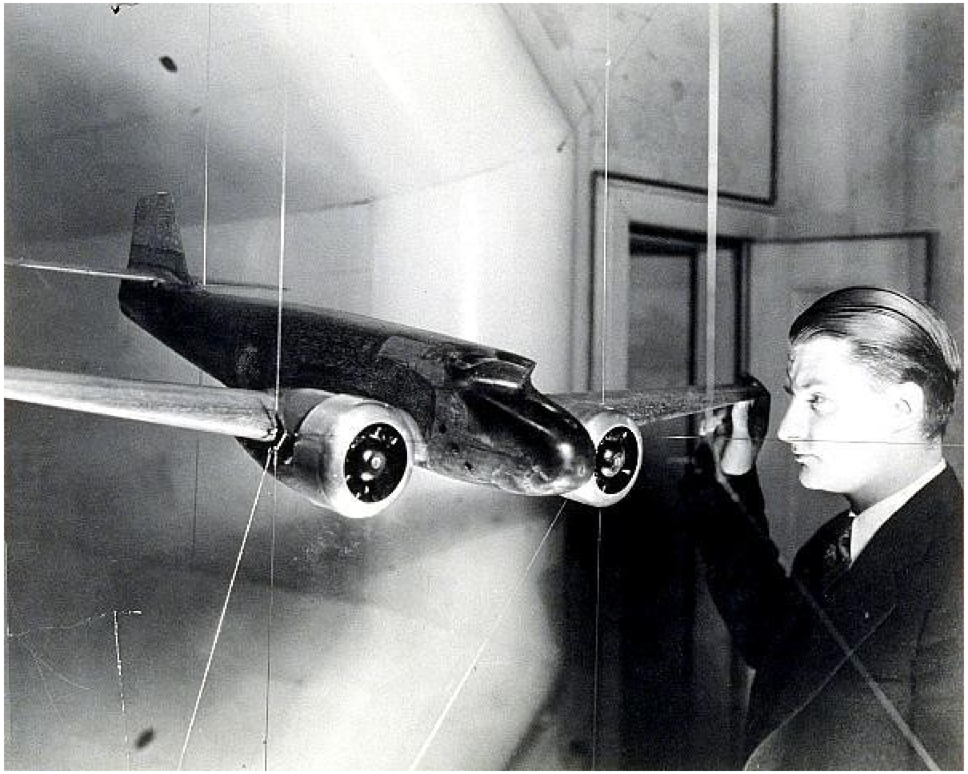
...and its (UML) Model



Can you see it now?

Use of Models in Engineering

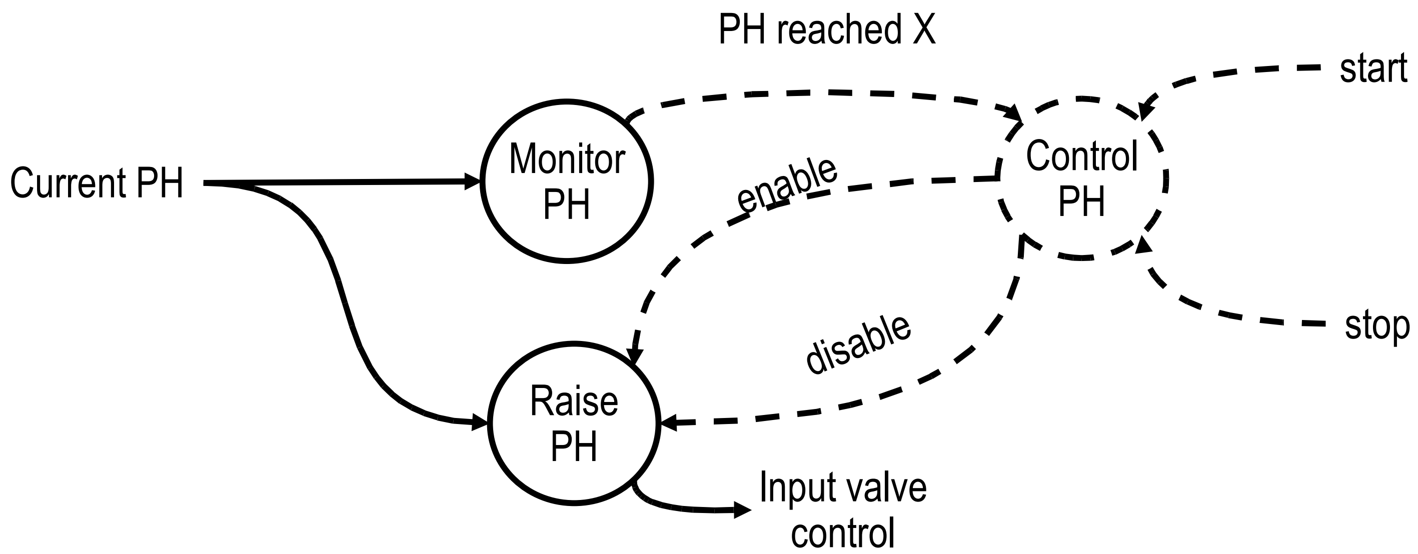
- Probably as old as engineering (c.f., Vitruvius)
- Engineering model:
  - A reduced representation of some system that highlights its properties of interest from a given viewpoint



- We don't <u>see</u> everything at once
- What we <u>do</u> see is adjusted to human understanding

What about modeling software?

A Common Perception of the Value of Software Models



"…bubbles and arrows, as opposed to programs, …never crash"

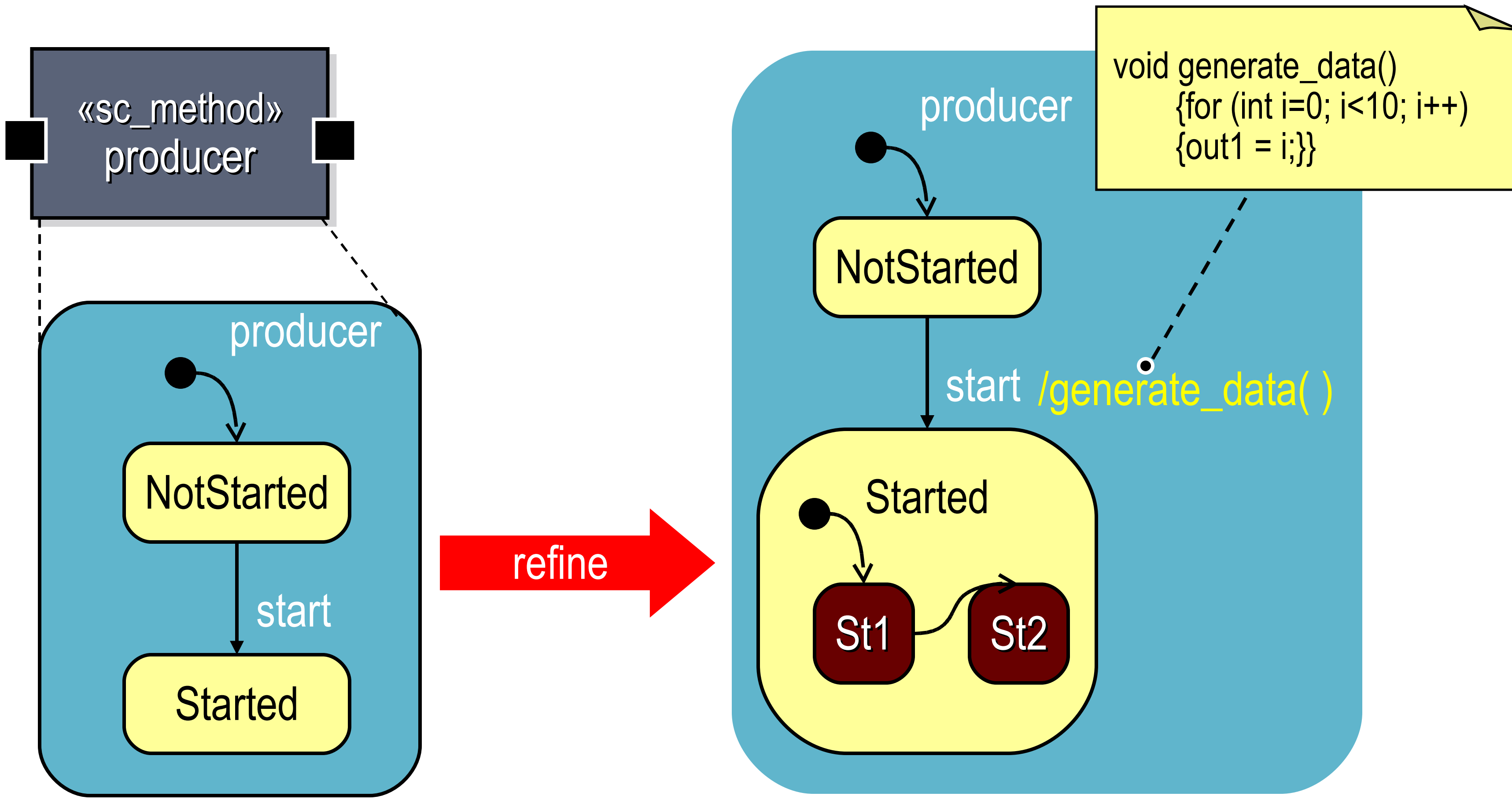
-- B. Meyer
"UML: The Positive Spin"
American Programmer, 1997

Characteristics of Useful Engineering Models

- Abstract
  - Emphasize important aspects while removing irrelevant ones
- Understandable
  - Expressed in a form that is readily understood by observers
- Accurate
  - Faithfully represents the modeled system
- Predictive
  - Can be used to answer questions about the modeled system
- Efficient
  - Should be much cheaper and faster to construct than actual system

To be useful, engineering models must satisfy all of these characteristics!

Modern Software Modeling Practices

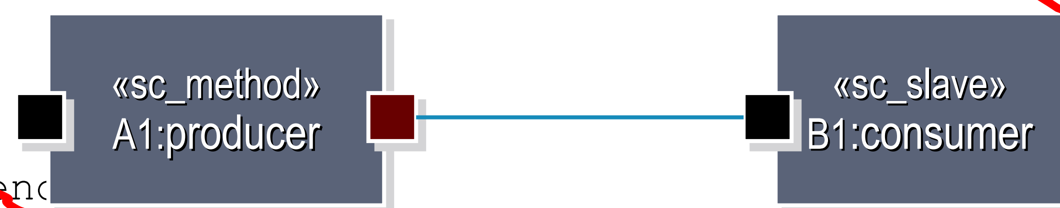


- Models can be refined continuously until the application is fully specified ⇒ *the model becomes the system that it was modeling!*

Exploiting Automation: Code Generation from Models

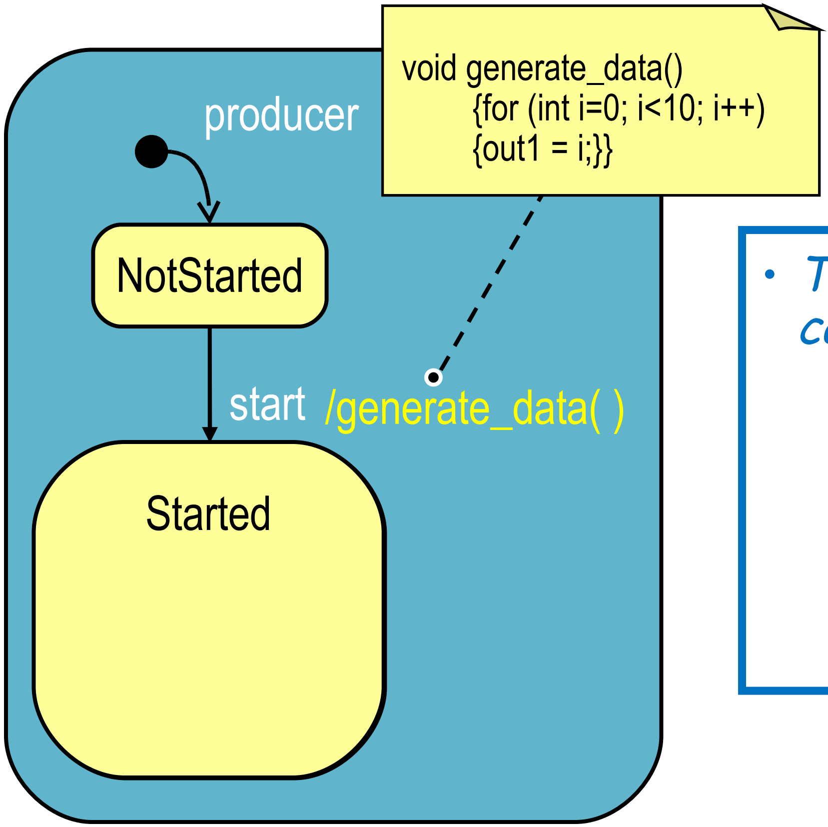
```
SC_MODULE(producer)
{
sc_outmaster<int> out1;
sc_in<bool> start; // kick-start
void generate_data ()
{
for(int i =0; i <10; i++) {
out1 =i ; //to invoke slave;}
}
SC_CTOR(producer)
{
SC_METHOD(generate_data);
sensitive << start;}};
SC_MODULE(consumer)
{
sc_inslave<int> in1;
int sum; // state variable
void accumulate (){
sum += in1;
cout << "Sum = " << sum << enc
```

```
SC_CTOR(consumer)
{
SC_SLAVE(accumulate, in1);
sum = 0; // initialize
};
SC_MODULE(top) // container
{
producer *A1;
consumer *B1;
sc_link_mp<int> link1;
SC_CTOR(top)
{
A1 = new producer("A1");
A1.out1(link1);
B1 = new consumer("B1");
B1.in1(link1);}};
```



Exploiting Automation: Models and Reality

- In all other engineering disciplines abstractions (models) are artifacts that are necessarily distinct from the systems that they model
  - Models are not always accurate representations of reality
- Uniquely, in software, the model and the modeled system share the same medium and can be formally coupled



- *The computer offers a uniquely capable abstraction device:*

  Software can be represented from any desired viewpoint at any desired level of abstraction

  The abstraction is inside the system and can be extracted automatically

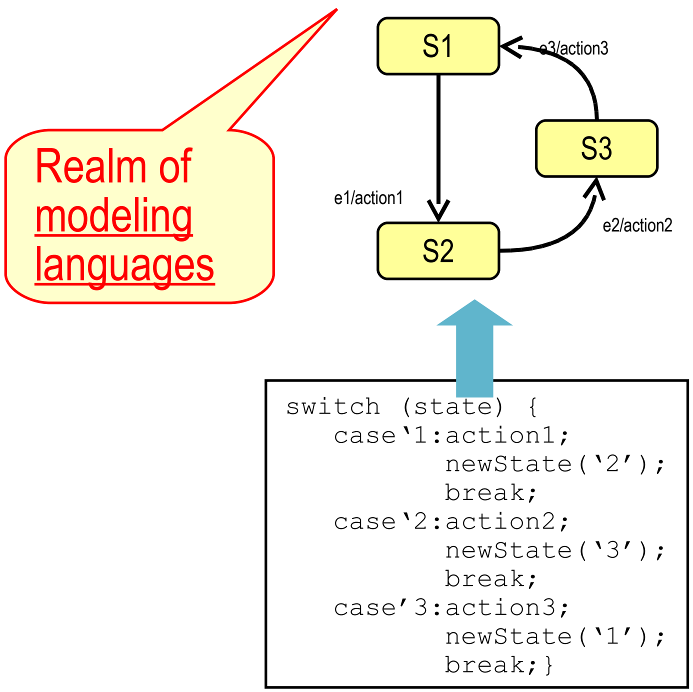
The Remarkable Thing About Software

Software has the unique property that it allows us to directly evolve models into complete implementations without fundamental discontinuities in the expertise, materials, tools, or methods!

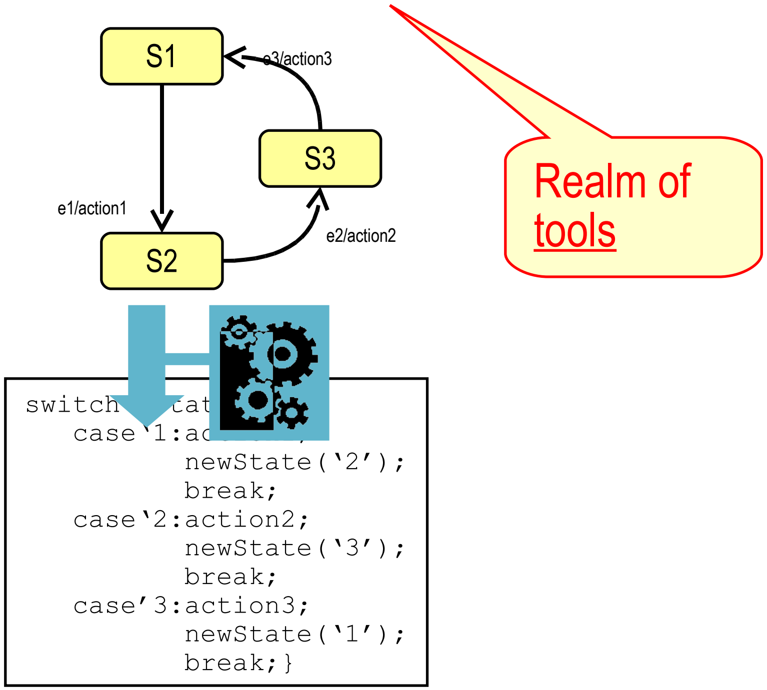
Model-Driven Development (MDD)

- An approach to software development in which models play an indispensable role
- Based on two time-proven methods:

(1) ABSTRACTION

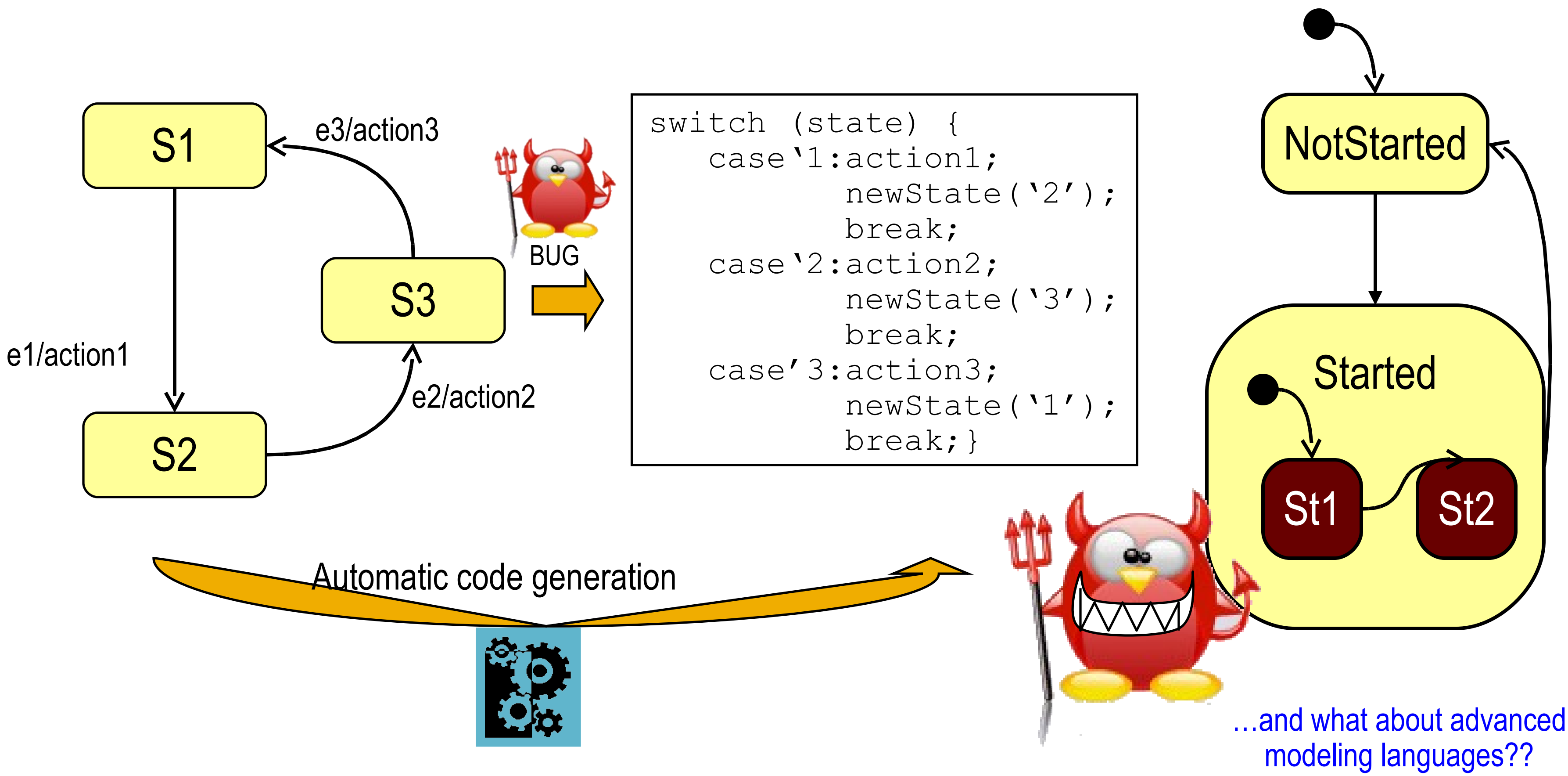


(2) AUTOMATION



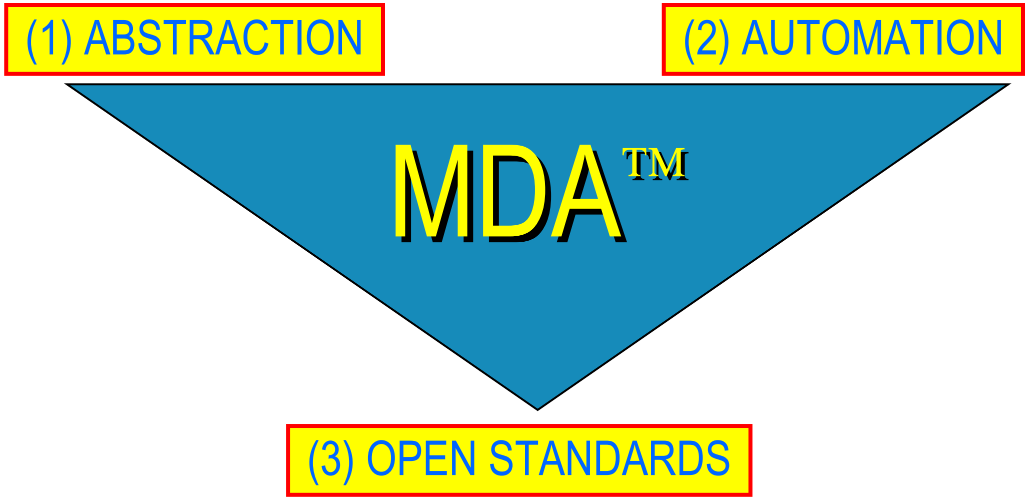
MDD: The Need for Automation

- The accidental complexity of current programming languages can be greatly reduced by the appropriate use of computer-based automation



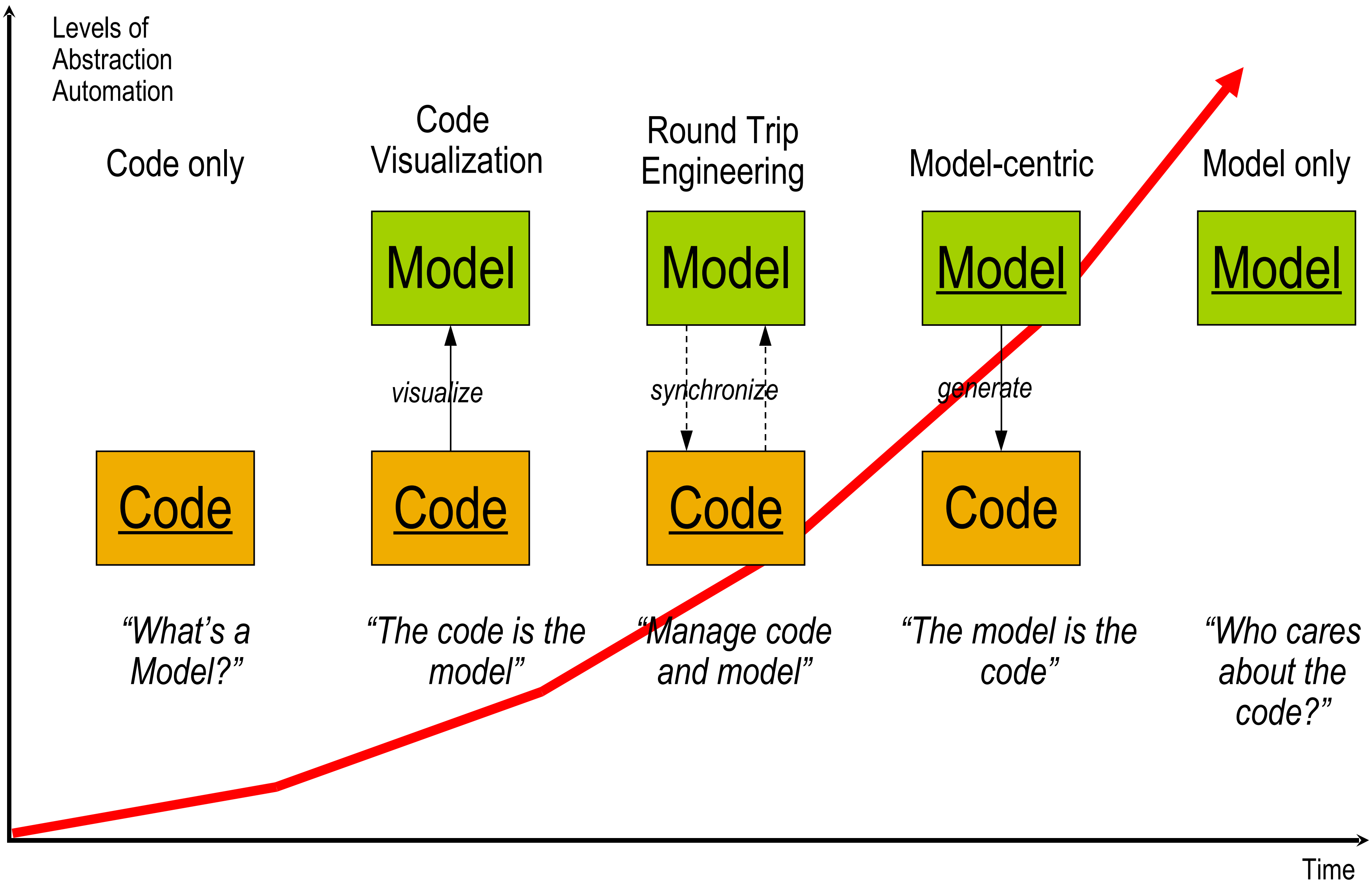
Model-Driven Architecture (MDA)

- An OMG initiative to support model-driven development through a series of open standards



- *Modeling languages*
- *Interchange standards*
- *Model transformations*
- *Software processes*
- *etc.*

Styles of MDD: The MDD Maturity Model



MDD: State of the Practice

- Example: Major Telecom Equipment Vendor
  - Adopted MDD Tooling
  - UsedMDD tools Rose RealTime (fully automated code generation directly from UML models), Test RealTime, RUP
- Product : Network Controller
  - 4.5 Million lines of auto-generated C++ code
  - 200+ developers working on a single model
- Performance (throughput, memory):
  - Within ± 15% of hand-crafted code
- Productivity improvement factors of 200%
  - 80% fewer bugs
  - Estimated productivity improvement = factor of 4
- There are many similar examples…

Sampling of Successful MDD Products

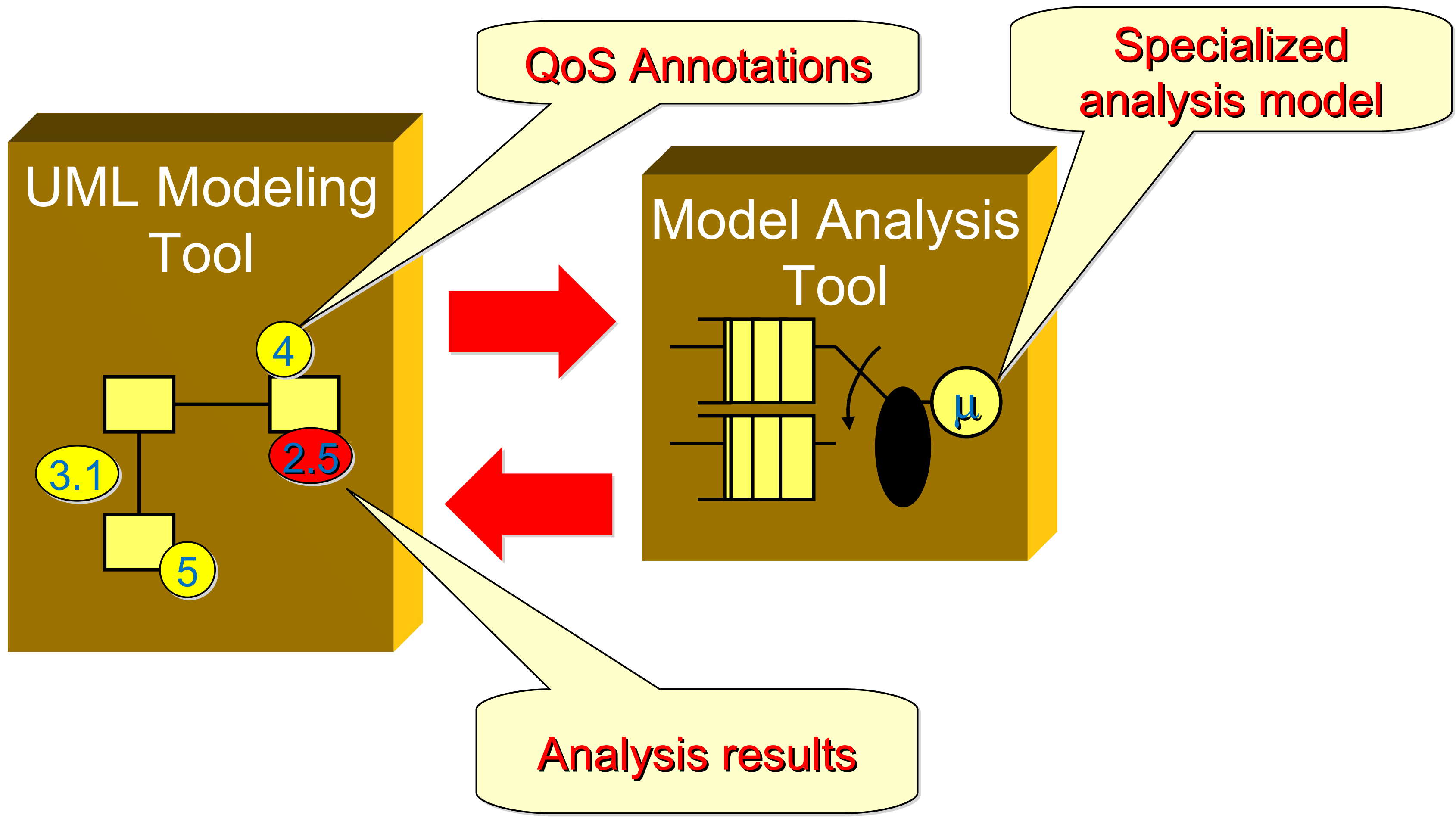
Automated doors, Base Station, Billing (In Telephone Switches), Broadband Access, Gateway, Camera, Car Audio, Convertible roof controller, Control Systems, DSL, Elevators, Embedded Control, GPS, Engine Monitoring, Entertainment, Fault Management, Military Data/Voice Communications, Missile Systems, Executable Architecture (Simulation), DNA Sequencing, Industrial Laser Control, Karaoke, Media Gateway, Modeling Of Software Architectures, Medical Devices, Military And Aerospace, Mobile Phone (GSM/3G), Modem, Automated Concrete Mixing Factory, Private Branch Exchange (PBX), Operations And Maintenance, Optical Switching, Industrial Robot, Phone, Radio Network Controller, Routing, Operational Logic, Security and fire monitoring systems, Surgical Robot, Surveillance Systems, Testing And Instrumentation Equipment, Train Control, Train to Signal box Communications, Voice Over IP, Wafer Processing, Wireless Phone

Automation Opportunity: Formal Methods

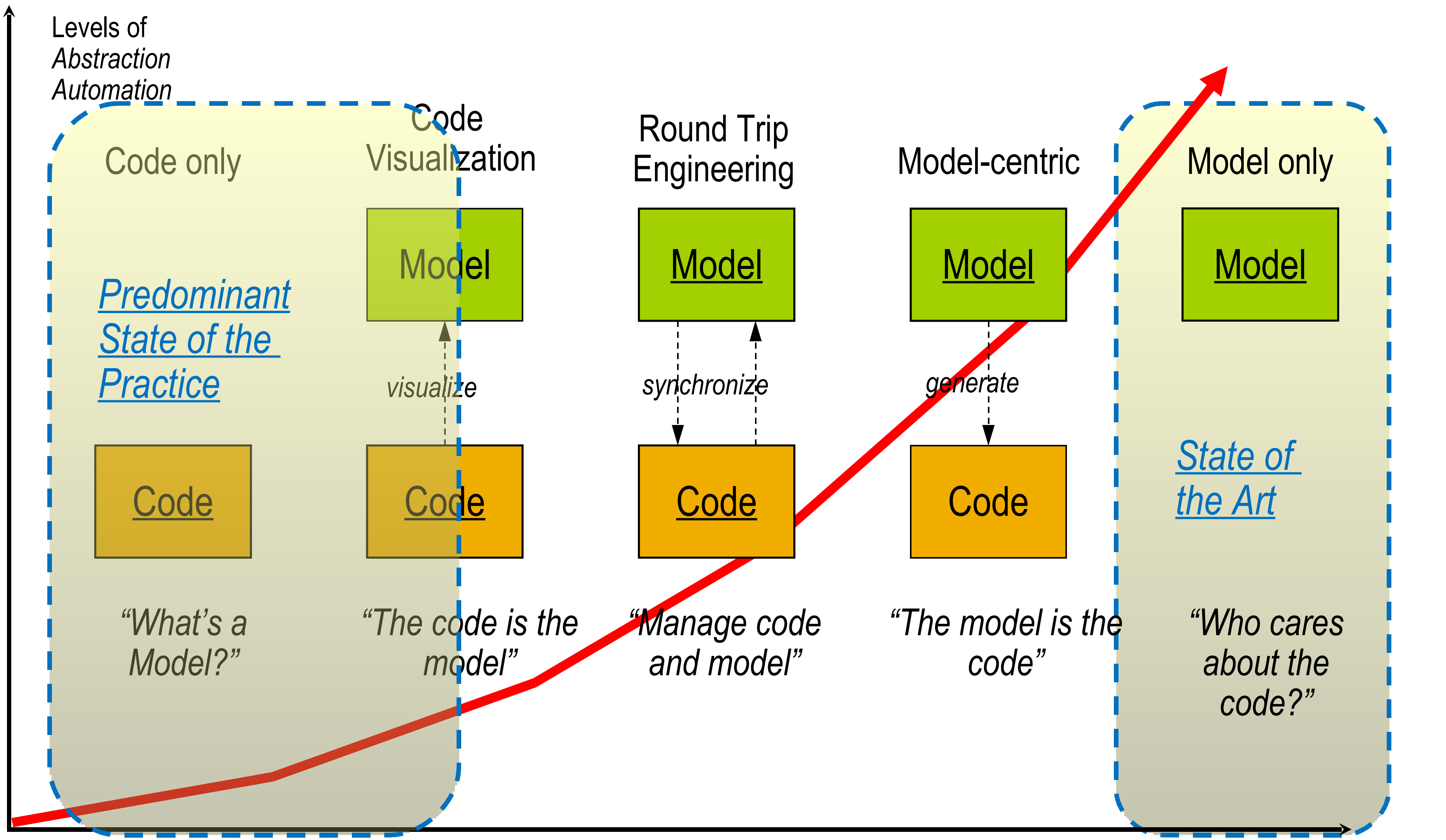
- Given the possibility of making modeling language constructs better behaved than programming language constructs, it is possible to exploit formal methods that could not handle the semantic complexity of programming languages
  - E.g., state machines, Petri nets
  - Model checking, theorem proving
- We need to work on formal semantics of modeling languages

utomation Opportunity: Model Analysis

- Complementary inter-working of specialized tools based on shared standards



The State of the Art and the State of the Practice



If this stuff is so good, what's holding us back?

PART II: WHAT STANDS IN THE WAY

Root Causes of Low Adoption Rate

- Technical
  - Usability issues
  - Technical flaws/immaturity
  - Lack of interoperability and standardization
- Cultural
  - Lack of awareness
  - Psychology of practitioners
  - Short-term ROI business culture
- Economic
  - Investment hurdle
  - Risk

Usability

- The unmitigated complexity of “modern” software technologies is a strong deterrent to their introduction in practice
- Most programs and program interfaces are designed by software practitioners
  - Inadequate understanding of (or sympathy for) end users and their objectives
  - Inadequate understanding of relevant economic factors
  - Inadequate understanding of key human factors
    - e.g., “syntactic sugar” mindset
  - Strong focus on technology, often combined with a penchant for complex (“sophisticated”) solutions
    - The features firehose effect
    - Usability as a late add-on
- Modern software tools are orders of magnitude too complex to be truly effective
  - Require significant investment to master
    - Deflects from core business concerns
    - Limited to a few “keeners” who invest time and effort in mastering tools
    - But, value of such expertise may have a relatively short lifespan

- Technical
  - Usability issues
  - Technical flaws/immaturity
  - Lack of interoperability and standardization
- Cultural
  - Lack of awareness
  - Psychology of practitioners
  - Short-term ROI business culture
- Economic
  - Investment hurdle
  - Risk

Technical Flaws and Immaturity

- Many new technologies harbour serious and often dangerous technical flaws
- Example: Design of MDD technologies such as the UML modeling language
  - Insufficient experience and understanding of the problem and characteristics of potential solutions
- Many (most?) technical innovations in commercial software practice were developed by commercial enterprises
  - Localized and short-term market focus
  - Based on inadequate theoretical understanding $\Rightarrow$ technical flaws
  - No time, resources, or incentive to develop necessary theoretical base
  - Exacerbated by current pressure on research institutions to demonstrate market "relevance"

- Technical
  - Usability issues
  - Technical flaws/immaturity
  - Lack of interoperability and standardization
- Cultural
  - Lack of awareness
  - Psychology of practitioners
  - Short-term ROI business culture
- Economic
  - Investment hurdle
  - Risk

Lack of Interoperability and Standards

- New technologies are often defined with no thought given to compatibility with legacy or other technologies
  - No cost-effective transition path for adoption of new technologies
  - Islands of advanced technology are rarely of major value
  - ⇒ Interoperability is being increasingly more recognized as a fundamental requirement for new software technologies
- More standards, supported by major players, are needed for key technologies to support interoperability
  - E.g., tool interworking standards, metamodeling standards, semantics specification standards,...
  - Requires collaboration of key vendors
  - Standards are, invariably, suboptimal from a technological perspective ⇒ standard excuse for ignoring them

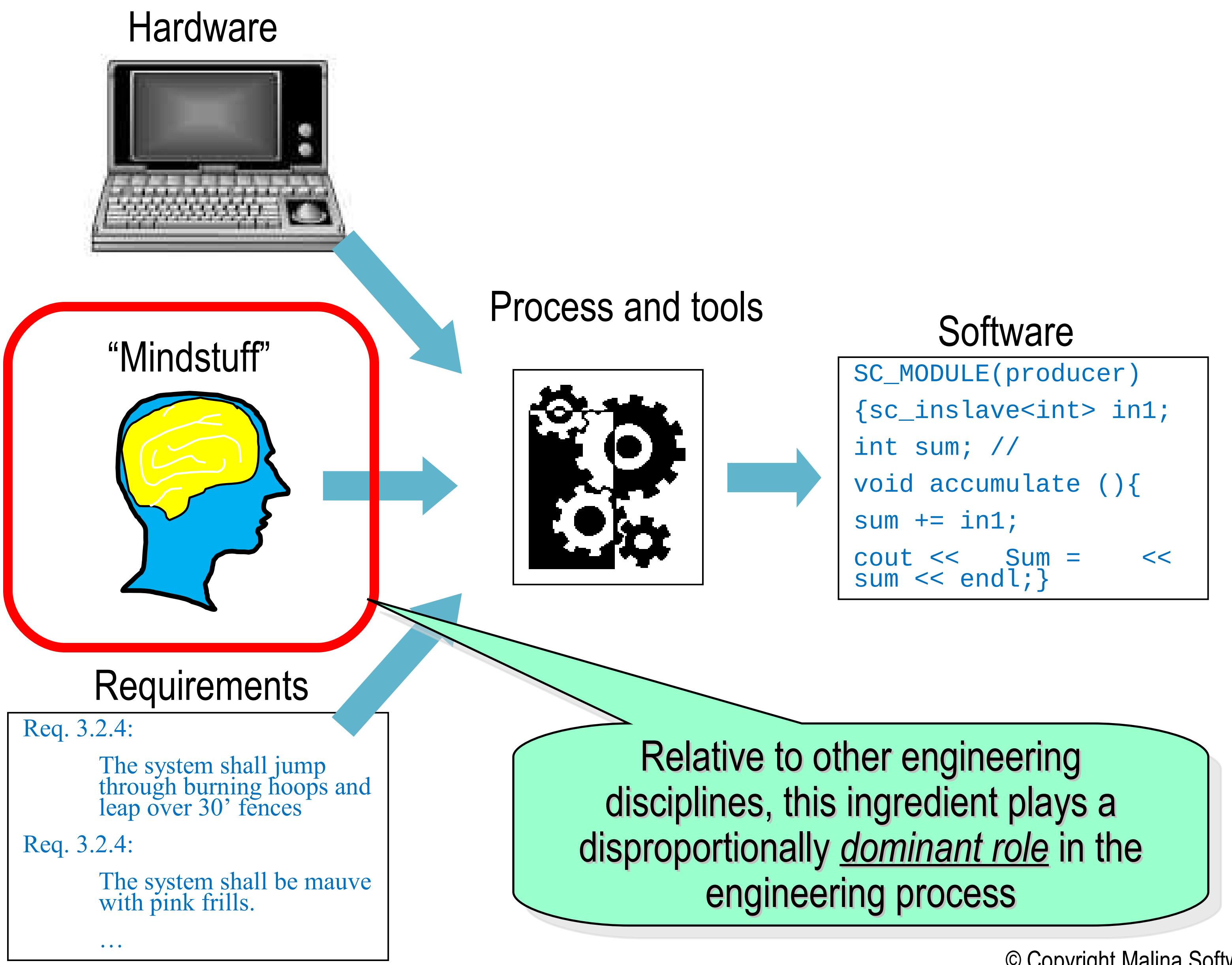
- Technical
  - Usability issues
  - Technical flaws/immaturity
  - Lack of interoperability and standardization
- Cultural
  - Lack of awareness
  - Psychology of practitioners
  - Short-term ROI business culture
- Economic
  - Investment hurdle
  - Risk

Lack of Awareness and Vision

- Despite the glut of information about advances in high-technology, many practitioners remain unaware of the capabilities and achievements of potentially revolutionary technologies such as MDD
- Technological ruts (ratholes?)
  - Practitioners tend to limit their focus on news directly related to the technologies they are already using
  - Even many highly-respected "thought leaders" have fallen into technology ruts
    - E.g., OOPSLA '07 panel on programming languages
- For competitive reasons, enterprises are often unwilling to publicize successful application of new technologies
  - Dearth of published verifiable evidence

- Technical
  - Usability issues
  - Technical flaws/immaturity
  - Lack of interoperability and standardization
- Cultural
  - Lack of awareness
  - Psychology of practitioners
  - Short-term ROI business culture
- Economic
  - Investment hurdle
  - Risk

The Idiosyncrasies of Software Engineering



Some Consequences

- Compared to other engineering disciplines, software development is much less hampered by physical reality
  - …but, not completely free (software systems = software + hardware)
- Low inertia: The path from conception to realization (edit-compile-run cycle) is exceptionally fast
  - Often leads to an impatient state of mind (tinkerer (vs. engineer) mentality)
  - …which leads to unsystematic and hastily conceived solutions (hacking)
  - Also yields a highly seductive and gratifying experience
  - …so that, often, the concern for the product becomes secondary
- The medium becomes the message
  - Focus of many practitioners shifts from the product and the end-user to the development process and the technology

When the Medium Becomes the Message…

- Software engineers often identify themselves not by their domain expertise (e.g., telecom, financial systems, aerospace) but by their technology expertise (e.g., C++, EJB, Linux)
- Consequences:
  - Lack of understanding of and interest in the problem domain and end users
    - Few system architects
  - Resistance towards new/different technologies
    - …and whether they may be better suited to the problem on hand
  - Suboptimal solutions
  - Personal and product obsolescence

Einstein's Message

"Concern for man himself and his fate must always constitute the chief objective of all technological endeavors...in order that the creations or our minds shall be a blessing and not a curse to mankind. Never forget this in the midst of your diagrams and equations."

-- A. Einstein, 1931

And More Consequences

- The unique pliability of software combined with the tinkerer's mentality that it spawns make it very difficult to define stable and widely-adopted engineering standards
  - Key to engineering reuse
  - ⇒ constant re-invention and a false sense of progress
- Also makes it difficult to work out foundational issues
- Culture of impatience with any apparent curtailing of design flexibility
  - Successful engineering typically requires limiting freedom of choice (e.g., software architectures)

The Problem of The Great Inertial Mass

- Numerous generations of software practitioners were raised with this culture
  - ~12-20 million programmers in the world
  - ...most of them holding on to what they know and unwilling to move outside their technological rut comfort zones
- How to overcome this enormous inertial mass?

- Technical
  - Usability issues
  - Technical flaws/immaturity
  - Lack of interoperability and standardization
- Cultural
  - Lack of awareness
  - Psychology of practitioners
  - Short-term ROI business culture
- Economic
  - Investment hurdle
  - Risk

Today's Dominant Business Culture

- Based on short-term return on investment (ROI)
  - Markets today force focus on quarterly results
  - Business and technology development plan horizons are rarely meaningful beyond 12 months
  - Reward structure based on short-term results
- Foundational research and introduction of new technologies requires more distant horizons and long-term investments
  - Today's model of research funding is strongly tied to short-term market relevance
    - Not conducive to research into fundamentals
    - Hampers ground-breaking outside-the-box innovation

- Technical
  - Usability issues
  - Technical flaws/immaturity
  - Lack of interoperability and standardization
- Cultural
  - Lack of awareness
  - Psychology of practitioners
  - Short-term ROI business culture
- Economic
  - Investment hurdle
  - Risk

The Investment Hurdle and Risk

- Switching to new technologies and methods requires a major up-front investment in training and re-tooling.
  - Expense
  - Exacerbated by the short-term ROI syndrome
- There is a high risk of failure even when the new technologies have proven successful in other projects and environments
  - Lack of experience
  - Resistance to change

PART III: WHAT WE CAN DO

Where to Seek Solutions

- Education
  - Foster a user-centred culture
  - Extend SE curriculum beyond mere computing technology
- Research
  - Multidisciplinary effort (not solvable by technologists alone)
  - Investment in theoretical foundations
- Standardization
  - Further push to develop standards

Education

"The [engineer] should be equipped with knowledge of many branches of study and varied kinds of learning, for it is by his judgment that all work done by the other arts is put to test. This knowledge is the child of practice and theory."

- Vitruvius

On Architecture, Book I (1^{st} Century BC)

Education: Getting Closer to the End User

- There is an unfortunate lack of awareness of and lack of respect for end users and their needs
  - Personal gratification should not come solely from having designed and constructed the system, but from seeing it in use
  - The medium is **not** the message
- Implies achieving a deep level of understanding of the value of the system to the customer
  - Implies a scope of skills and knowledge that extends far beyond the technical domain
  - Required at every level (not just system architects)

Education: Understanding the Role of Technology

- More than just finding inspiration for technical solutions in non-technical sources
  - Although, higher levels of general literacy are sorely needed (particularly writing skills)
- Understanding and respect for the greater social, cultural, economic context in which technical inventions function
  - Understand when and how to apply technological solutions
  - Avoid often futile attempts to solve non-technical issues with yet more technology
  - Reduce current glut of confusing and problematic technologies that cause more problems than they solve

Education: Understanding the Business Case

- Software engineers must be trained to understand and appreciate the greater business context
- "Must know" topics
  - Economics fundamentals: how markets work
  - Basics of business management and administration
  - Basics of accounting and key legal aspects (e.g., IP law)
  - Professional ethics
  - Basics of psychology and sociology
  - Project management/work organization
  - The essentials of marketing
  - Presentation skills

Education: Technical Requisites

- Abstraction plays a central role in software
  - More so than any other engineering discipline
- Mathematics is an excellent foundation for developing and honing abstraction skills
  - …and may sometimes even be directly applicable to the technical problems on hand ☺
  - Mathematical logic
  - Probability theory
  - Discrete mathematics
  - Optimization theory
  - History of technology and mathematics
- An understanding of the physics underlying software

Research: Theory and Practice

- *"The difference between theory and practice is much greater in practice than it is in theory"*
- The divide is growing
- Most practitioners disdain theory
  - Unfortunate, since some theory could help them substantially
- Most theoreticians don't understand practice
  - Unfortunate, since they could work on more useful lines of research
- Educational requirements:
  - Instill an appreciation for the value of theory
  - Instill an understanding of the pragmatics of industrial software development

Research: CERAS

- Centre of Excellence for Research in Adaptive Systems (CERAS) established to conduct research in:
  - Virtualization technologies
  - MDD
- A research initiative created by IBM Centre for Advanced Studies (Toronto), the Ontario Centres of Excellence (OCE), a number of major universities in Ontario, North Carolina, and Europe
  - https://www.cs.uwaterloo.ca/twiki/view/CERAS/CerasOverview
- MDD research objectives
  1. Define a systematic and comprehensive conceptual framework (map) for MDD
     - Serves as a foundation for current and future research and development
     - Captures a shared consensus of the technical objectives behind MDD
  2. Initiate a number of major research efforts to fill in crucial parts of the MDD framework

Research: CERAS Research Areas

- Foundations: A General Theory of Engineering Models
- Specifying Models of Software
- Model Transformations
- Model Analysis and Verification
- Model-Driven Development Methods
- MDD Tooling

Summary and Conclusions

- Much of the software engineering community is solidly mired in the 3GL technology rut
  - This technology is unable to cope adequately with the increasing complexity being demanded of modern software systems
- New technologies and methods, such as MDD, have demonstrated time and time again the ability to make a major difference in our battle with complexity
- Unfortunately, due to a variety of diverse and inter-related factors, the adoption of these technologies and methods has been slow
  - Technical, cultural, and economic issues
- Some of these impediments are beyond our influence as technologists and educators
- However, there are still many opportunities for individuals and organizations in the technical community to make a difference and accelerate the pace of adoption
  - Education, research, standardization

QUESTIONS, COMMENTS, ARGUMENTS...