The "Theory" and Practice of Modeling Language Design – "Теорија" и пракса пројектовања језика за моделирање софтверских система

Бранислав Селић

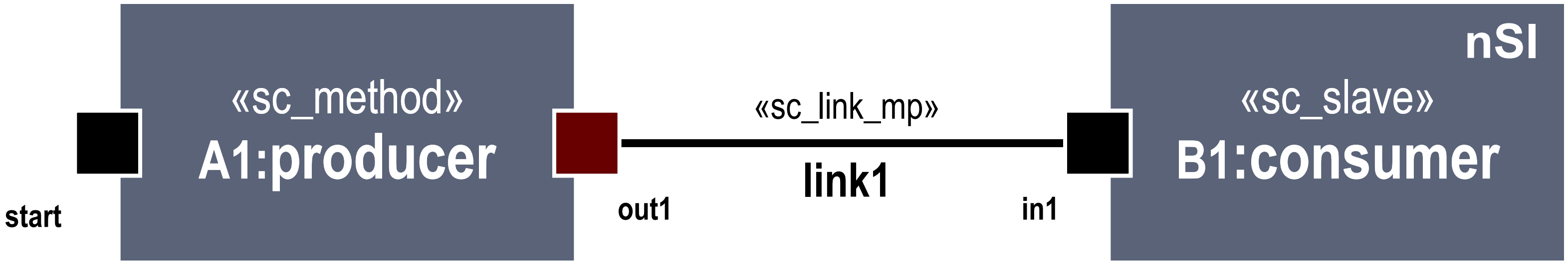
Malina Software Corp., Canada
Zeligsoft (2009) Ltd., Canada
Simula Research Labs, Norway
University of Toronto, Canada
Carleton University, Canada

selic@acm.org

A Programming Language Sample

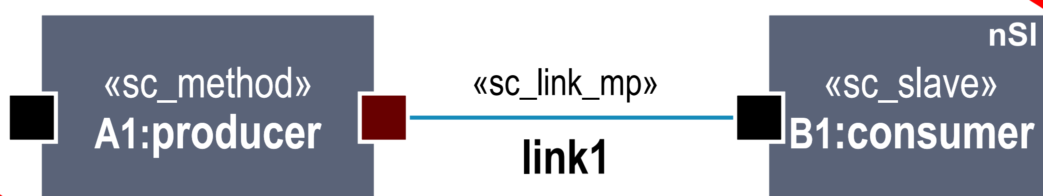
```
SC_MODULE(producer)
{
sc_outmaster<int> out1;
sc_in<bool> start; // kick-start
void generate_data ()
{
for(int i =0; i <nSl; i++) {
out1 =i ; //to invoke slave;}
}
SC_CTOR(producer)
{
SC_METHOD(generate_data);
sensitive << start;}};
SC_MODULE(consumer)
{
sc_inslave<int> in1;
int sum; // state variable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

…and its Model



```
SC_MODULE(producer)
{
sc_outmaster<int> out1;
sc_in<bool> start; // kick-start
void generate_data ()
{
for(int i =0; i <nSl; 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;
}
```

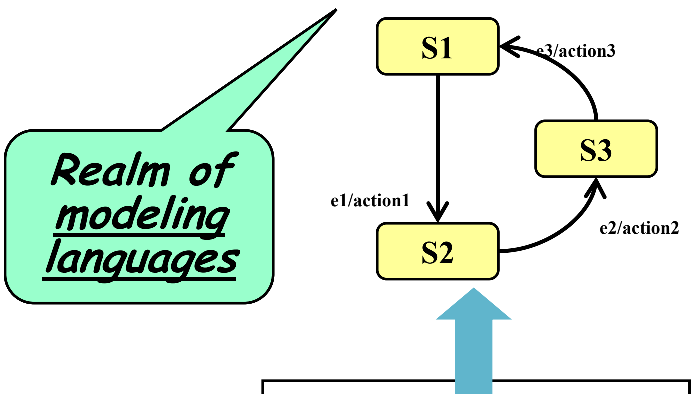
```
SC_CTOR(consumer)
{
SC_SLAVE(accumulate, in1);
sum = 0; // initialize
};
SC_MODULE(top) // container
{
producer *A1;
consumer *B1;
sc_link_mp<int> link1;
SC_CTOR(top)
{
A1 = new producer("A1");
A1.out1(link1);
for(int i =0; i <nSl; i++) {
  B1[i] = new consumer("B1");
  B1[i].in1(link1);}
}}
```



Model-Based (Software) Engineering (MBE)

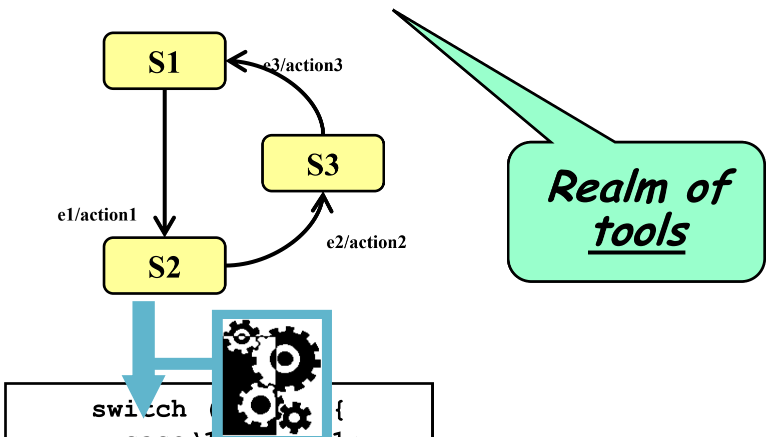
- An approach to system and software development in which software models play an <u>indispensable</u> role
- Based on two time-proven ideas:

(1) ABSTRACTION




```
switch (state) {
   case‘1:action1;
         newState(‘2’);
               break;
   case‘2:action2;
         newState(‘3’);
               break;
   case’3:action3;
         newState(‘1’);
              break;}
```

(2) AUTOMATION



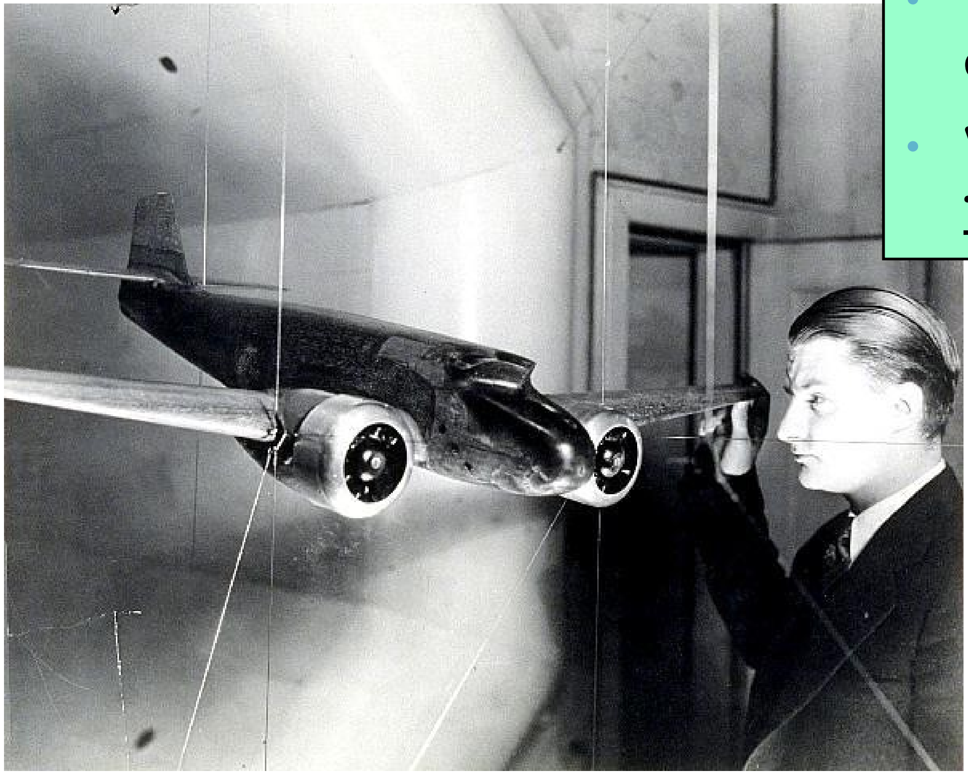

```
switch (          {
   case‘1        1;
         newState(‘2’);
               break;
   case‘2:action2;
         newState(‘3’);
               break;
   case’3:action3;
         newState(‘1’);
              break;}
```

Talk Outline

- Models: What and Why
- Modeling Language Design
- Modeling Language Specification
- Summary

Engineering Models

- **ENGINEERING MODEL:** *A selective representation of some system that specifies, accurately and concisely, all of its essential properties of interest for a given set of concerns**



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

* Selektivni prikaz nekog sistema, koji predstavlja, precizno i koncizno, suštinske odlike tog sistema sa odredjene tačke gledišta

Why Do Engineers Build Models?

- To understand
  - ...problems and solutions
  - Knowledge acquisition
- To communicate
  - ...understanding and design intent
  - Knowledge transfer
- To predict
  - ...the interesting characteristics of system under study
  - Models as surrogates
- To specify
  - ...the implementation of the system
  - Models as "blueprints"

Types of Engineering Models

- <u>Descriptive</u>: models for understanding, communicating, and predicting
  - E.g., scale models, mathematical models, qualitative models, documents, etc.
  - Tend to be highly abstract (detail removed)
- <u>Prescriptive</u>: models as specifications
  - E.g., architectural blueprints, circuit schematics, state machines, pseudocode, etc.
  - Tend to be detailed so that the specification can be implemented

Q: Is it useful to have models that can serve both kinds of purposes?

Characteristics of Useful Engineering Models

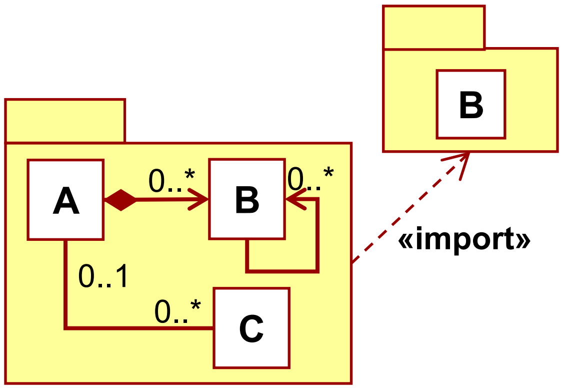
- **Purpose oriented:**
  - Constructed to address a specific set of concerns/audience
- **Abstract**
  - Emphasize important characteristics while obscuring irrelevant ones
- **Understandable**
  - Expressed in a form that is readily understood by intended audience
- **Accurate**
  - Faithfully represents the modeled system
- **Predictive**
  - Can be used to answer questions about the modeled system
- **Cost effective**
  - Should be much cheaper and faster to construct than actual system

A useful engineering model must satisfy at least these core characteristics.

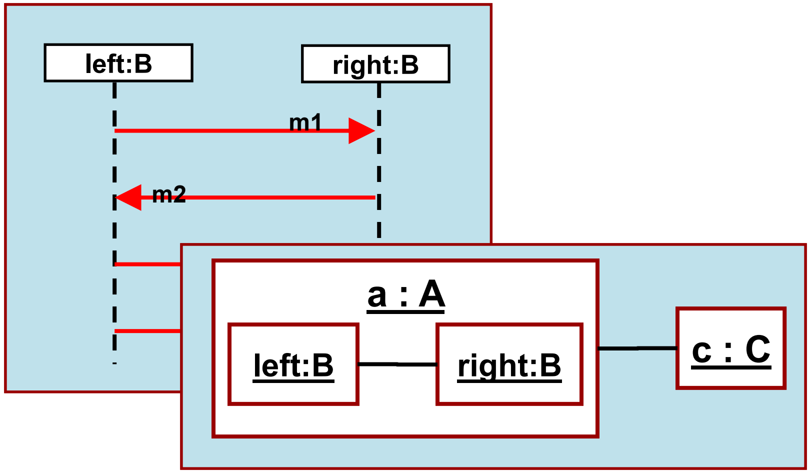
Modeling Software

What's a Software Model?

- **SOFTWARE MODEL:** An engineering model of a software system from *one or more viewpoints* specified using one or more *modeling languages*
  - E.g.:



Structural (design-time) view



Execution (run-time) view

What's a Modeling Language?

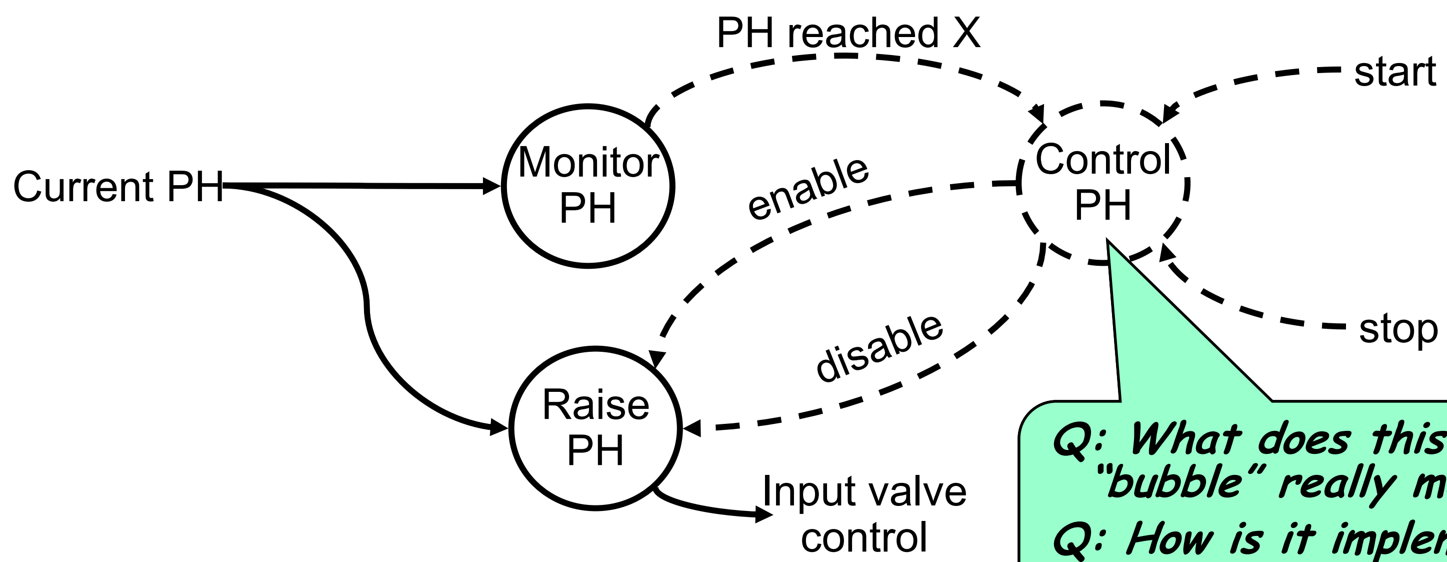
- MODELING LANGUAGE: A computer language intended for constructing models of systems and the contexts in which these systems operate
- Examples:
  - AADL, Matlab/Simulink, Modelica, SDL, SysML, UML, etc.

"Classical" Software Modeling Languages

- Flow charts, SA/SD, 90's OO notations (Booch, OMT, OOSE, UML 1)
- Most of them were intended exclusively for constructing descriptive models
  - Informal "sketching" [M. Fowler]*
  - No perceived need for high-degrees of precision
  - Languages are ambiguous and open to interpretation ⇒ source of undetected miscommunication

*http://martinfowler.com/bliki/UmlAsSketch.html

Classical SW Modeling: SA/SD



Q: What does this "bubble" really mean?

Q: How is it implemented in code?

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

Modeling languages have yet to recover from this "debacle"

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

New Generation of Modeling Languages

- Formal languages designed for modeling
  - ⇒ Support for both descriptive and prescriptive models
  - ...sometimes in the same language
- Key objectives:
  - Well-understood and precise semantic foundations
  - Can be formally (i.e., mathematically) analyzed (qualitative and quantitative analyses)
  - And yet, can still be used informally ("sketching") if desired

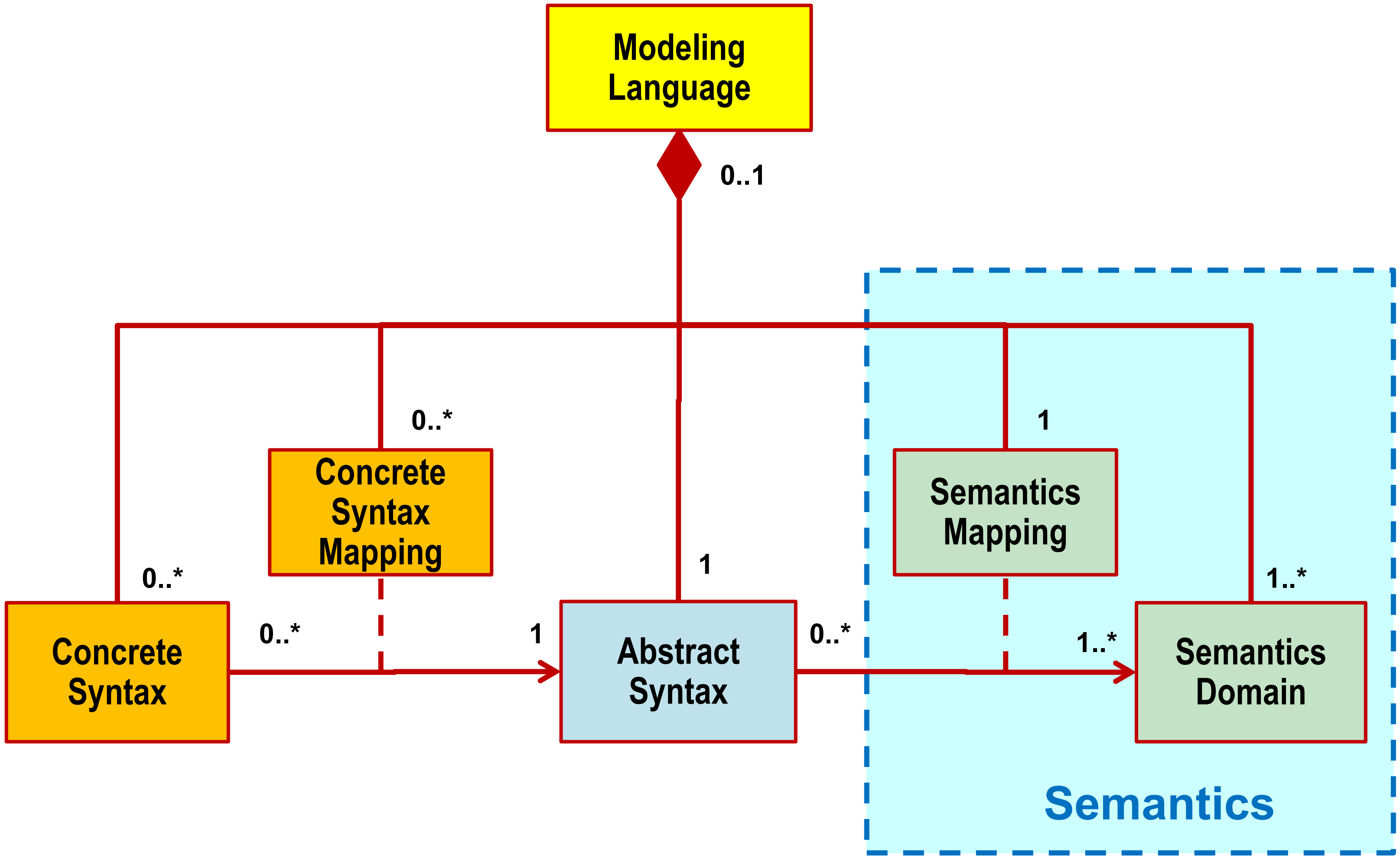
Modeling vs Programming Languages

- **The primary purpose and focus of programming languages is <u>implementation</u>**
  - The ultimate form of <u>prescription</u>
  - ⇒ Implementation requires total precision and "full" detail
  - ⇒ Takes precedence over description requirements
- **To be useful, a modeling language must support <u>description</u>**
  - I.e., <u>communication</u>, <u>prediction</u>, and <u>understanding</u>
  - These generally require omission of "irrelevant" detail such as details of the underlying computing technology used to implement the software

Components of a Modeling Language

- **The definition of a modeling language consists of:**
  - ABSTRACT SYNTAX
    - Set of language concepts/constructs ("ontology")
      - e.g., Account, Customer, Class, Association, Attribute, Package
    - Rules for combining language concepts (well-formedness rules)
      - e.g., "each end of an association must be connected to a class"
  - CONCRETE SYNTAX (notation/representation)
    - e.g., keywords, graphical symbols for concepts
    - Mapping to abstract syntax concepts
  - SEMANTICS: the *meaning* of the language concepts
    - Comprises: Semantic Domain and Semantic Mapping (concepts to domain)

Elements of a Modeling Language



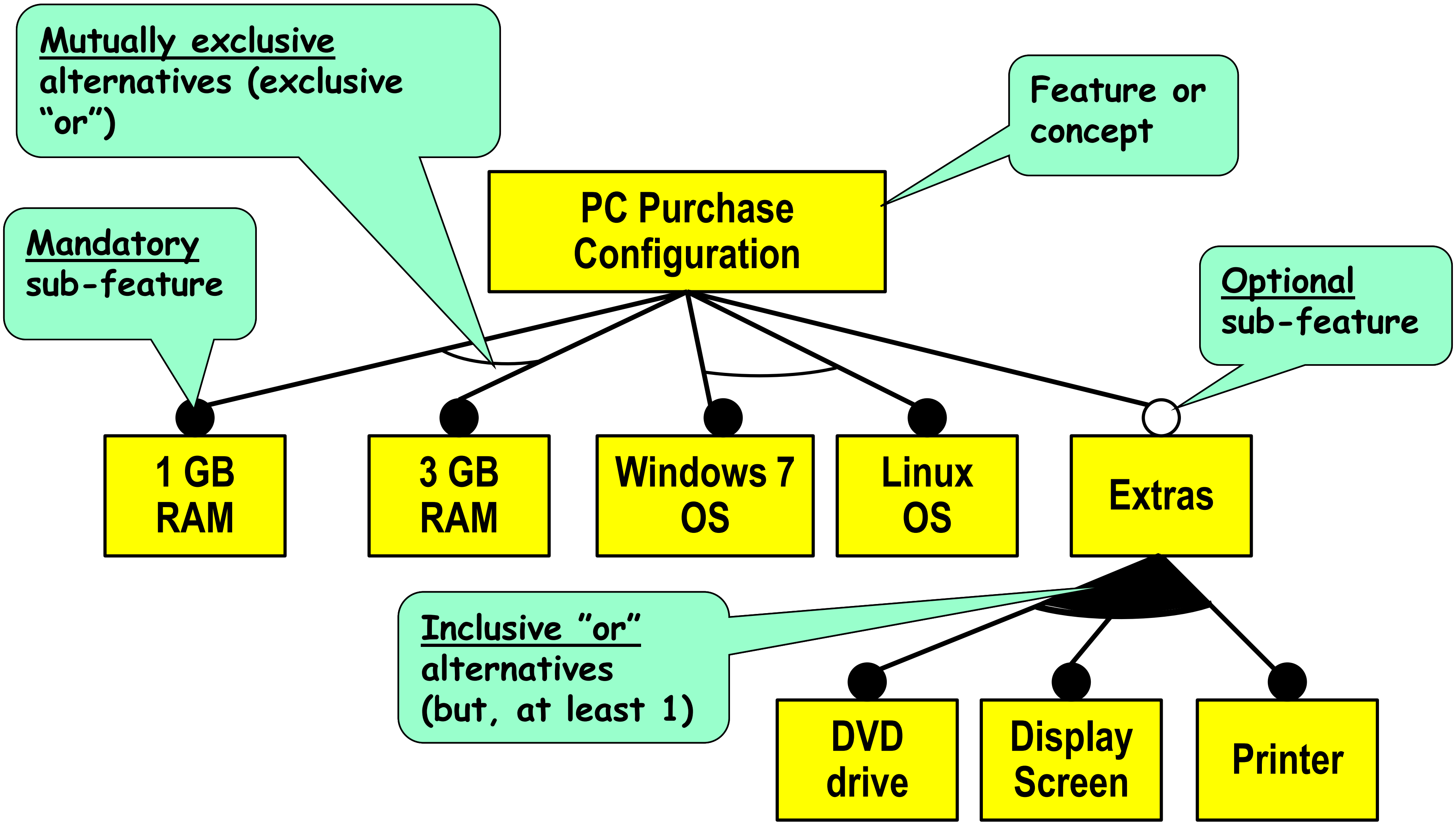
Talk Outline

- Models: What and Why
- Modeling Language Design
- Modeling Language Specification
- Summary

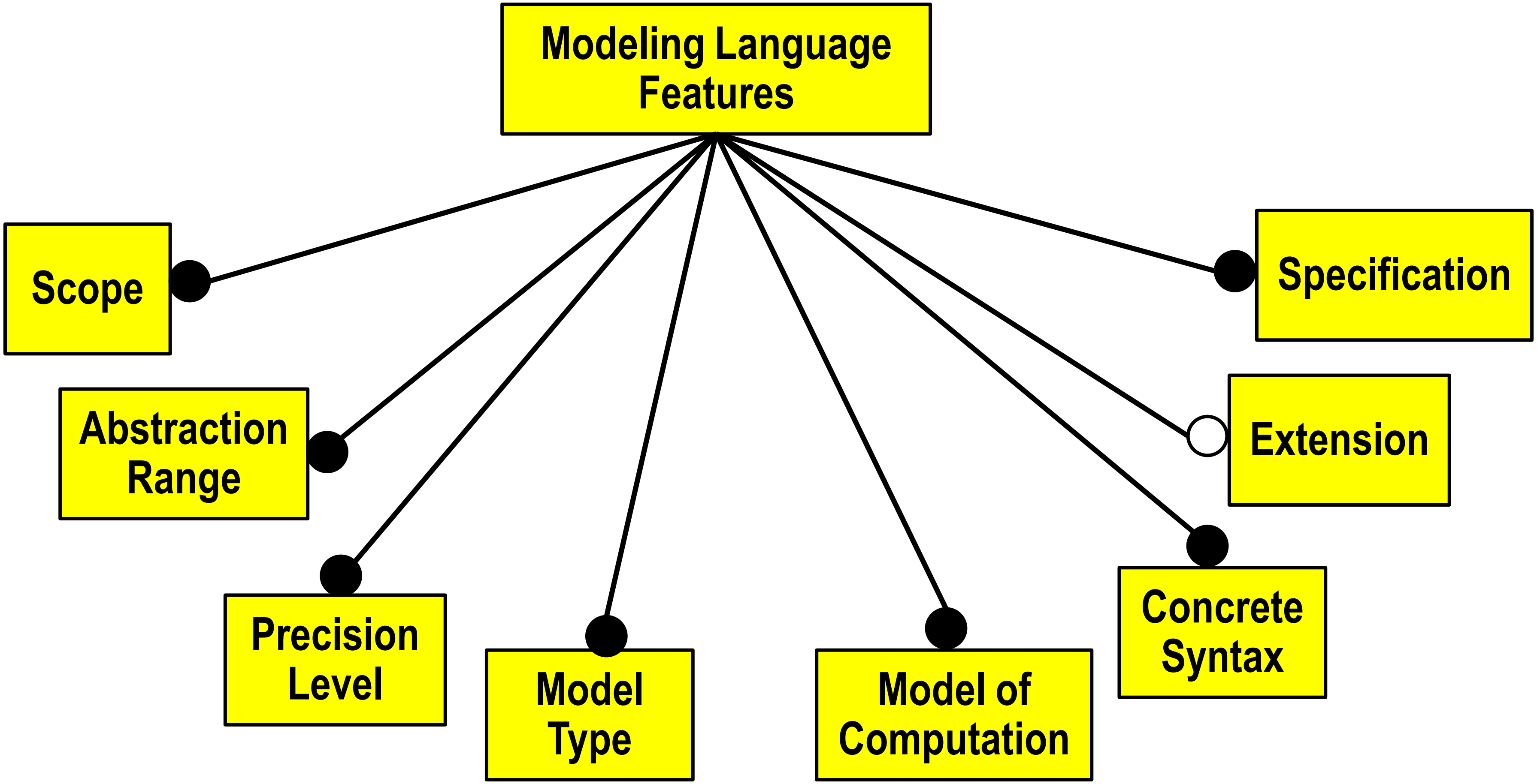
Primary Language Design Concerns

- Who are the primary end users?
  - Authors / readers? (i.e., primary use cases)
- What kind of models do they want?
  - Descriptive, prescriptive, or both?
- What is the domain?
  - What is the application domain and what are its salient technical characteristics?

Sidebar: Feature Diagram Essentials



Key Language Design Choices



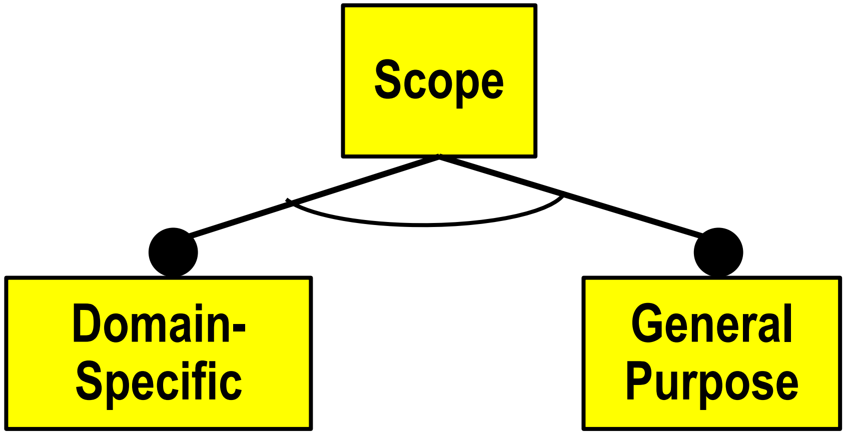
Some choices are inter-dependent

Selecting Language Scope

- A common opinion:

  "Surely it is better to design a small language that is highly expressive, because it focuses on a specific narrow domain, as opposed to a large and cumbersome language that is not well-suited to any domain?"

- Which suggests:



But, this may be an oversimplification

Scope: How General/Specific?

- **Generality often comes at the expense of expressiveness**
  - Expressiveness: the ability to specify *concisely* yet *accurately* a desired system or property
  - Example:
    - UML does not have a concept that specifies mutual exclusion devices (e.g. semaphore) ⇒ to represent such a concept in our model, we would need to combine a number of general UML concepts in a particular way (e.g., classes, constraints, interactions)
  - ...which may(?) be precise, but not very concise
- **It also comes at the cost of detail that is necessary to:**
  - Execute models
  - Generate complete implementations

Specialization: Inevitable Trend

- Constant branching of application domains into ever-more specialized sub-domains
  - As our knowledge and experience increase, domain concepts become more and more refined
    - E.g., simple concept of computer memory → ROM, RAM, DRAM, cache, virtual memory, persistent memory, etc.
- One of the core principles of MBE is raising the level of abstraction of specifications to move them closer to the problem domain

- This seems to imply that domain-specific languages are invariably the preferred solution
- But, there are some serious hurdles here...

The Case of Programming Languages

- **Literally hundreds of domain-specific programming languages have been defined over the past 50 years**
  - Fortran: for scientific applications
  - COBOL for "data processing" applications
  - Lisp for AI applications
  - etc.
- **Some relevant trends**
  - Many of the original languages are still around
  - More often than not, highly-specialized domains still tend to use general-purpose languages with specialized domain-specific program libraries and frameworks instead of domain-specific programming languages
  - In fact, the trend towards defining new domain-specific <u>programming</u> languages seems to be diminishing
- **Why is this happening?**

Success* Criteria for a Language (1)

- Technical validity: absence of major design flaws and constraints
  - Ease of writing correct programs
- Expressiveness
- Simplicity: absence of gratuitous/accidental complexity
  - Ease of learning
- Efficiency: speed and (memory) space
- Familiarity: proximity to widely-available skills
  - E.g., syntax

* "Success" ⇒ language is adopted by a substantive development community and used with good effect for practical applications

Success Criteria for a Language (2)

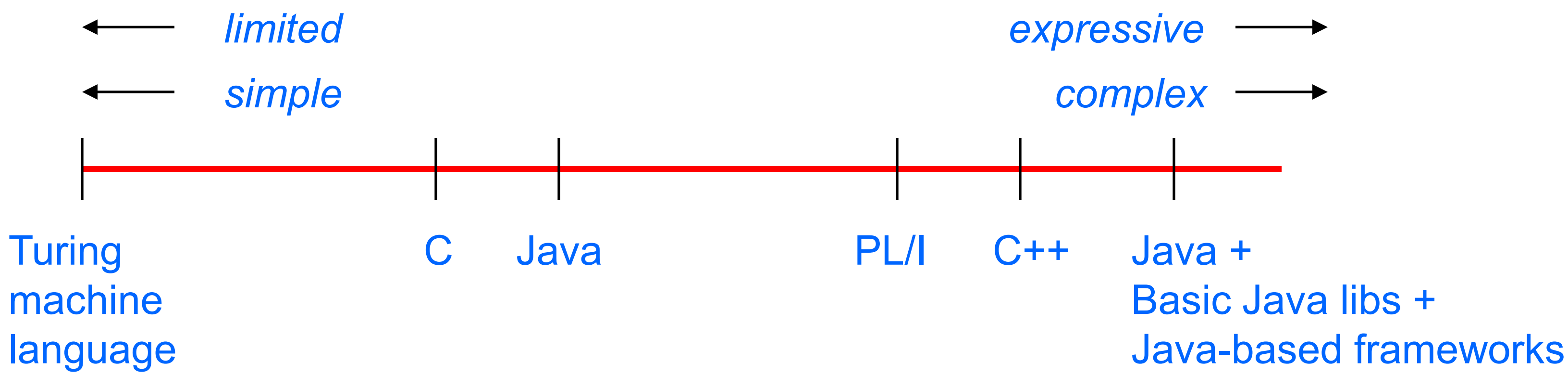
- **Language Support & Infrastructure:**
  - Availability of necessary tooling
  - Effectiveness of tools (reliability, quality, usability, customizability, interworking ability)
  - Availability of skilled practitioners
  - Availability of teaching material and training courses
  - Availability of program libraries
  - Capacity for evolution and maintenance (e.g., standardization)

Sidebar: Basic Tooling Capabilities

- **Essential**
  - Model Authoring
  - Model validation (syntax, semantics)
  - Model export/import
  - Document generation
  - Version management
  - Model compare/merge
- **Useful (to Essential)**
  - Code generation
  - Model simulation/debug/trace
  - Model transformation
  - Model review/inspection
  - Collaborative development support
  - Language customization support
  - Test generation
  - Test execution
  - Traceability

Language Size

- ***How complex (simple) should a language be to make it effective?***



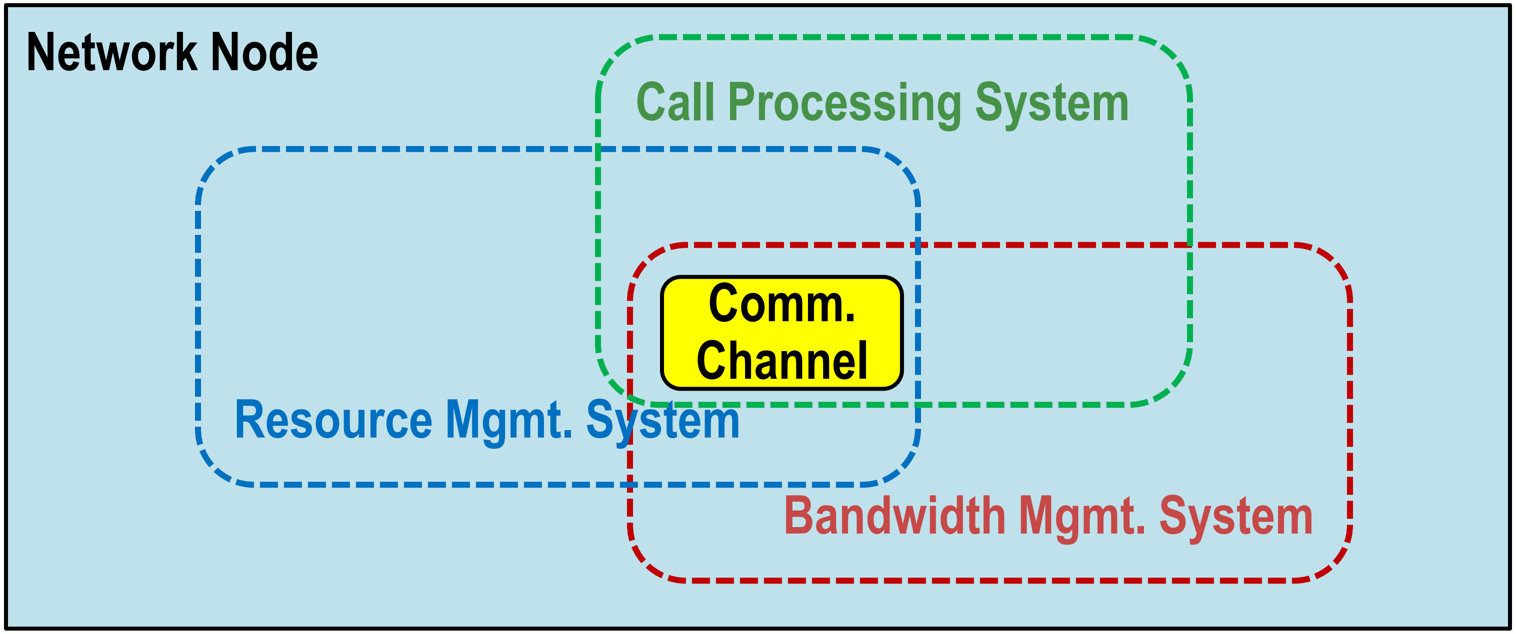
- **The art of computer language design lies in finding the right balance between expressiveness and simplicity**
  - Need to minimize accidental complexity while recognizing and respecting essential complexity
  - Small languages solve small problems
  - No successful language has ever gotten smaller

Practical Issues of Scope

- Practical systems often involve multiple heterogeneous domains
  - Each with its own ontology and semantic and dedicated specialists
- Example: a telecom network node system
  - Basic bandwidth management
  - Equipment and resource management
  - Routing
  - Operations, administration, and systems management
  - Accounting (customer resource usage)
  - Computing platform (OS, supporting services)

The Fragmentation Problem

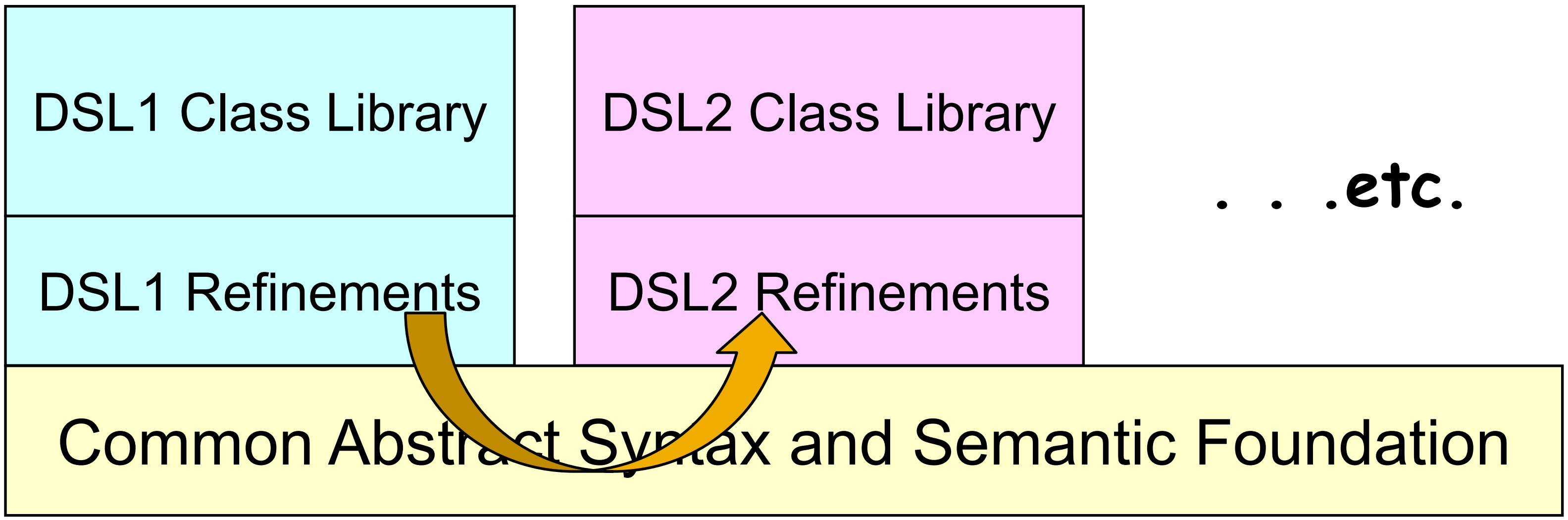
- **FRAGMENTATION PROBLEM:** combining overlapping <u>independently specified</u> domain-specific subsystems, specified using <u>different</u> DSLs, into a coherent and consistent whole (i.e., single implementation)



Sadly, there are no generic composition (weaving) rules – each case has to be handled individually

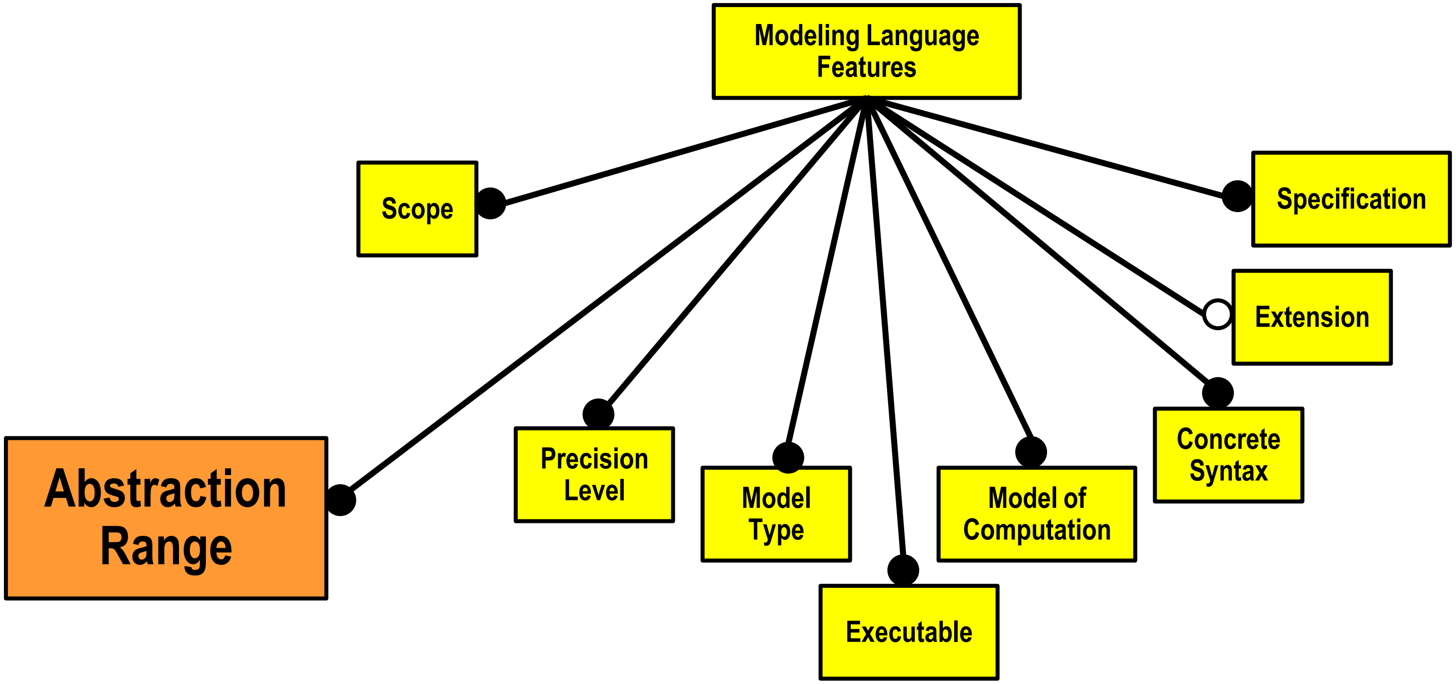
Approach to Dealing with Fragmentation

- Having a common syntactic and semantic foundations for the different DSLs seems as if it should facilitate specifying the formal interdependencies between different DSMLs



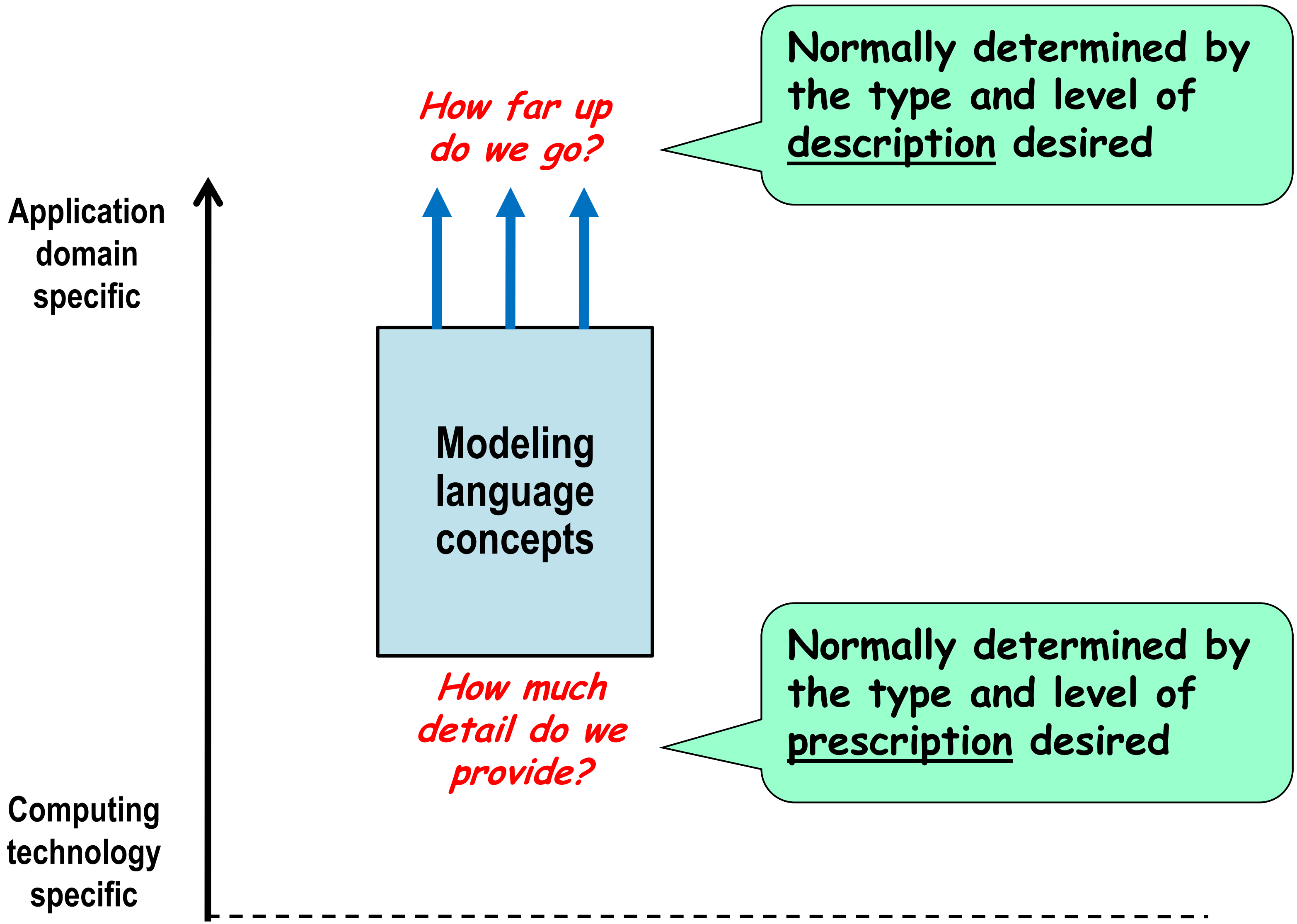
- NB: Same divide and conquer approach can be used to modularize complex languages
  - Core language base + independent sub-languages (e.g., UML)

Selecting An Abstraction Range

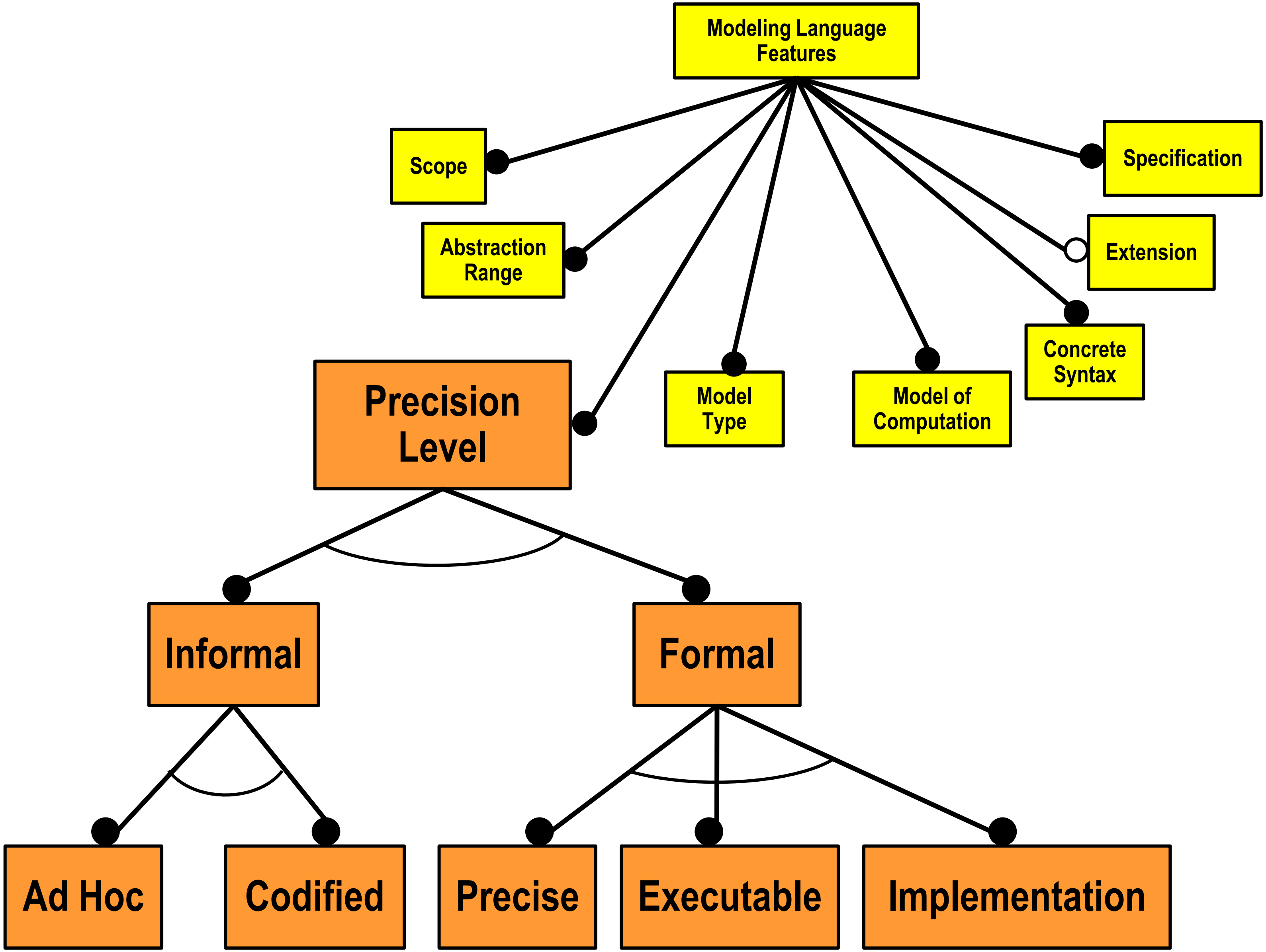


- This decomposes into two separate questions:
  - What is a suitable level of abstraction of the language?
  - How much (implementation-level) detail should the language concepts include?
- The answers depend on other design choices

Abstraction Range of Computer Languages



Selecting a Precision Level



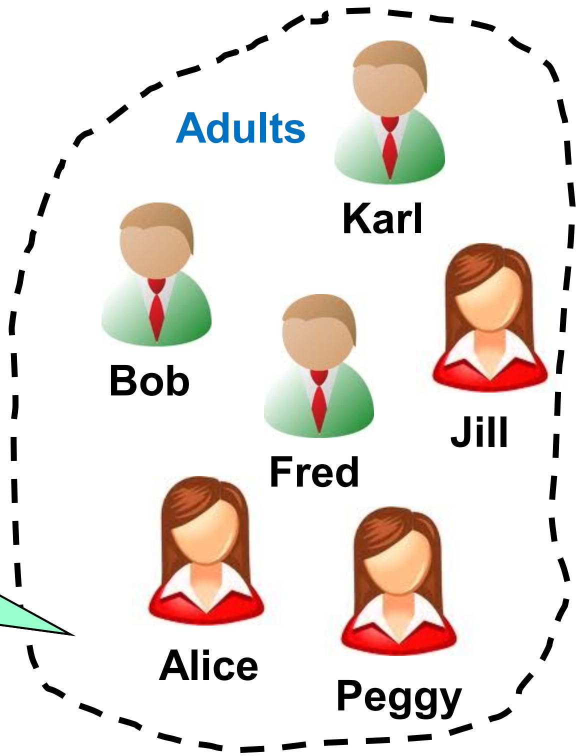
Formality

- **Based on a well understood mathematical theory with existing analysis tools**
  - E.g., automata theory, abstract state machines, Petri nets, temporal logic, process calculi, queueing theory, Horne clause logic
  - NB: precise does not necessarily mean detailed
- **Formality provides a foundation for automated validation of models**
  - Model checking (symbolic execution)
  - Theorem proving
  - However, the value of these is constrained due to scalability issues ("the curse of dimensionality")
- **It can also help validate the language definition**
- **But, it often comes at the expense of expressiveness**
  - Only phenomena recognized by the formalism can be expressed accurately

Precision vs. Detail

- **A specification can be precise but still leave out detail:**
  - E.g., we can identify a set very precisely without necessarily specifying the details associated with its members

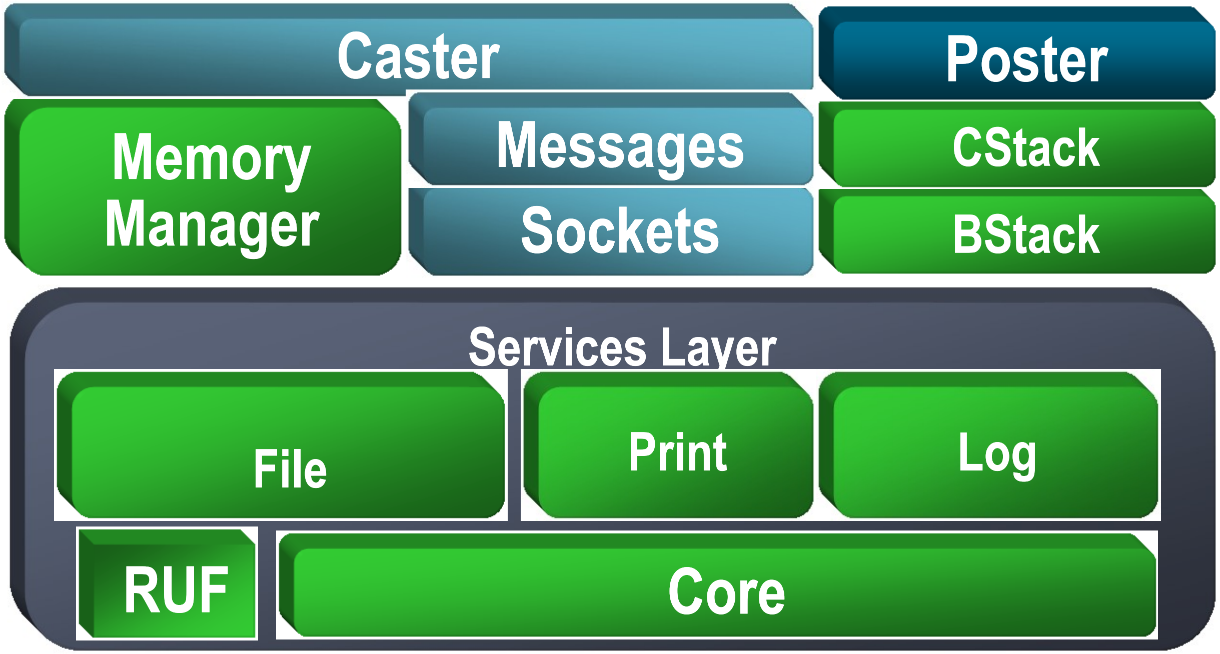
We state very precisely as to what constitutes <u>the set of Adults</u> of some population (age $\geq$ 21) without being specific about details such as names or genders of its members



back

Ad Hoc “Languages”

- Mostly notations created for specific cases (not intended for reuse)
- Used exclusively for descriptive purposes
- No systematic and comprehensive specification of syntax or semantics
  - *Appeal to intuition*



Codified (Informal) Languages

- Example: UML, OMT, SysML, ...
- Characteristics:
  - Defined: *An application-independent language specification exists*
  - Some aspects of the language are fully defined (usually: concrete syntax, semantics)
  - Semantics usually based on natural language and other informal specification methods
  - Designed primarily for descriptive modeling
  - But, may also be used partly for specification (e.g., partial code generation/code skeletons)

Precise Languages

- Examples: Object Constraint Language (OCL), Layered Queueing Networks (LQN)
- Fully defined semantics (domain and mapping)
- High level of abstraction but typically cover relatively small range
  - I.e., lacking detail for execution or implementation
  - Often declarative
- Mostly designed for prescription (e.g., prediction and analysis), but may also be used for specification

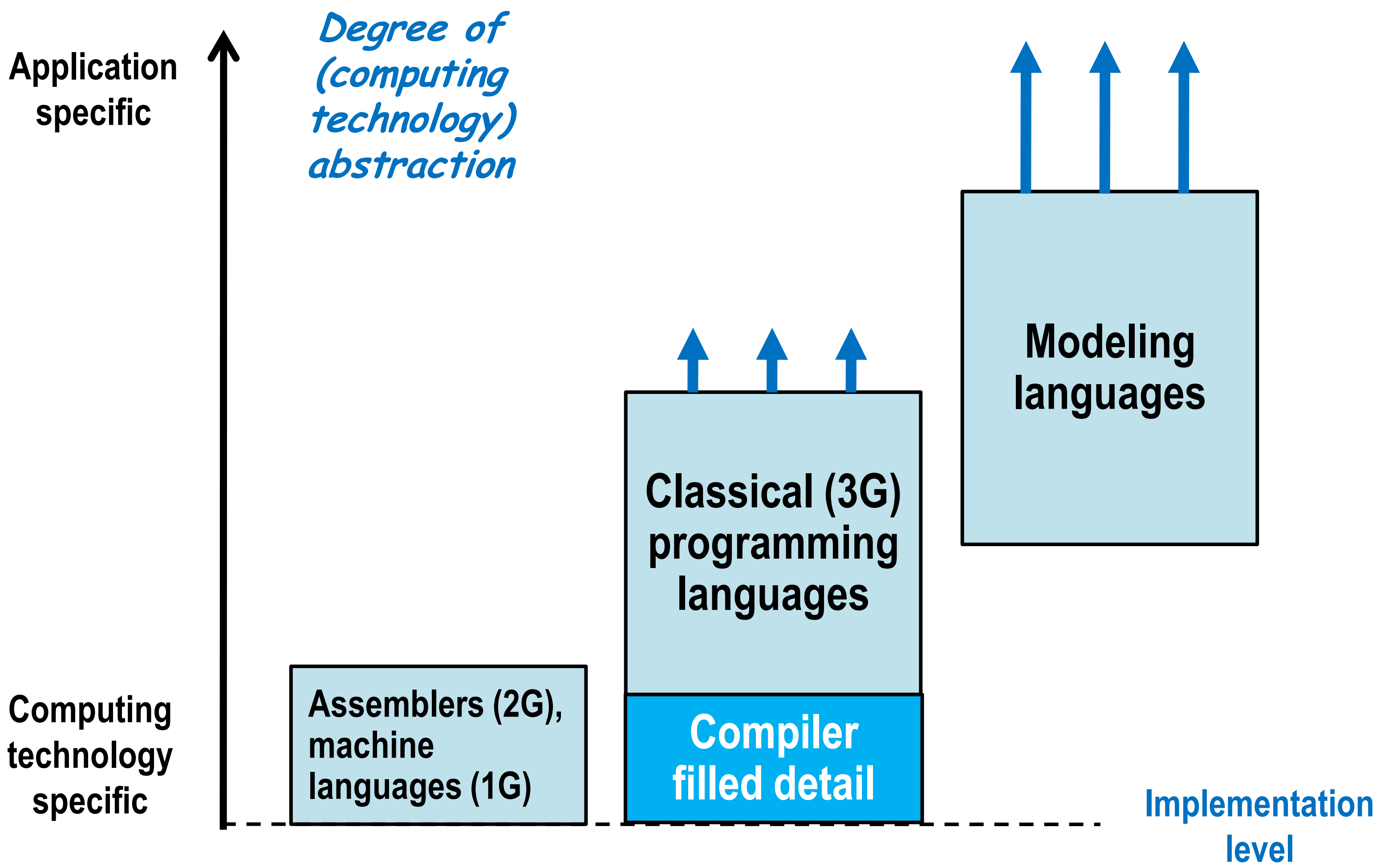
Executable Languages

- "Models that are not executable are like cars without engines", [D. Harel]
- Examples: Modelica, Matlab
- A subcategory of precise languages covering a range that includes sufficient detail for creating executable models
  - But, may be missing detail required for automatic generation of implementations
  - Often based on operational semantics that may not be easily analyzed by formal methods (due to scalability issues)
- Rationale:
  - Enables early detection of design flaws
  - Helps develop engineering intuition and confidence

Implementation (Modeling) Languages

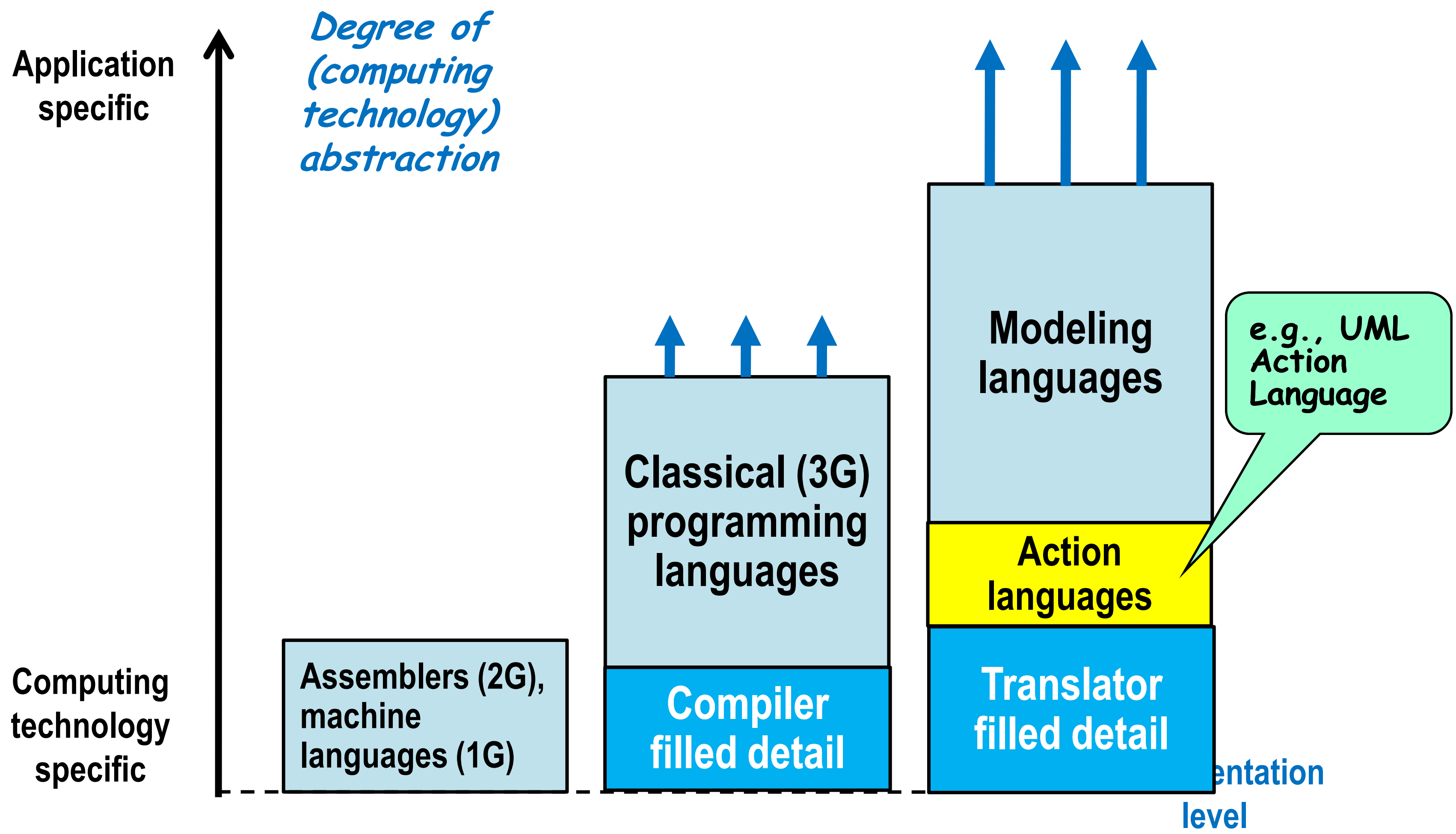
- **Computer languages that:**
  - Provide concepts at high levels of abstraction suitable for descriptive purposes, and also
  - Include detailed-level concepts such that the models can provide efficient implementations through either automatic code generation or interpretation
- **Examples: UML-RT, Rhapsody UML, SDL-2000, Matlab/Simulink, etc.**

Language Abstraction Levels



Full Range Modeling Languages

- A number of "descriptive" modeling languages have evolved into fully-fledged implementation languages



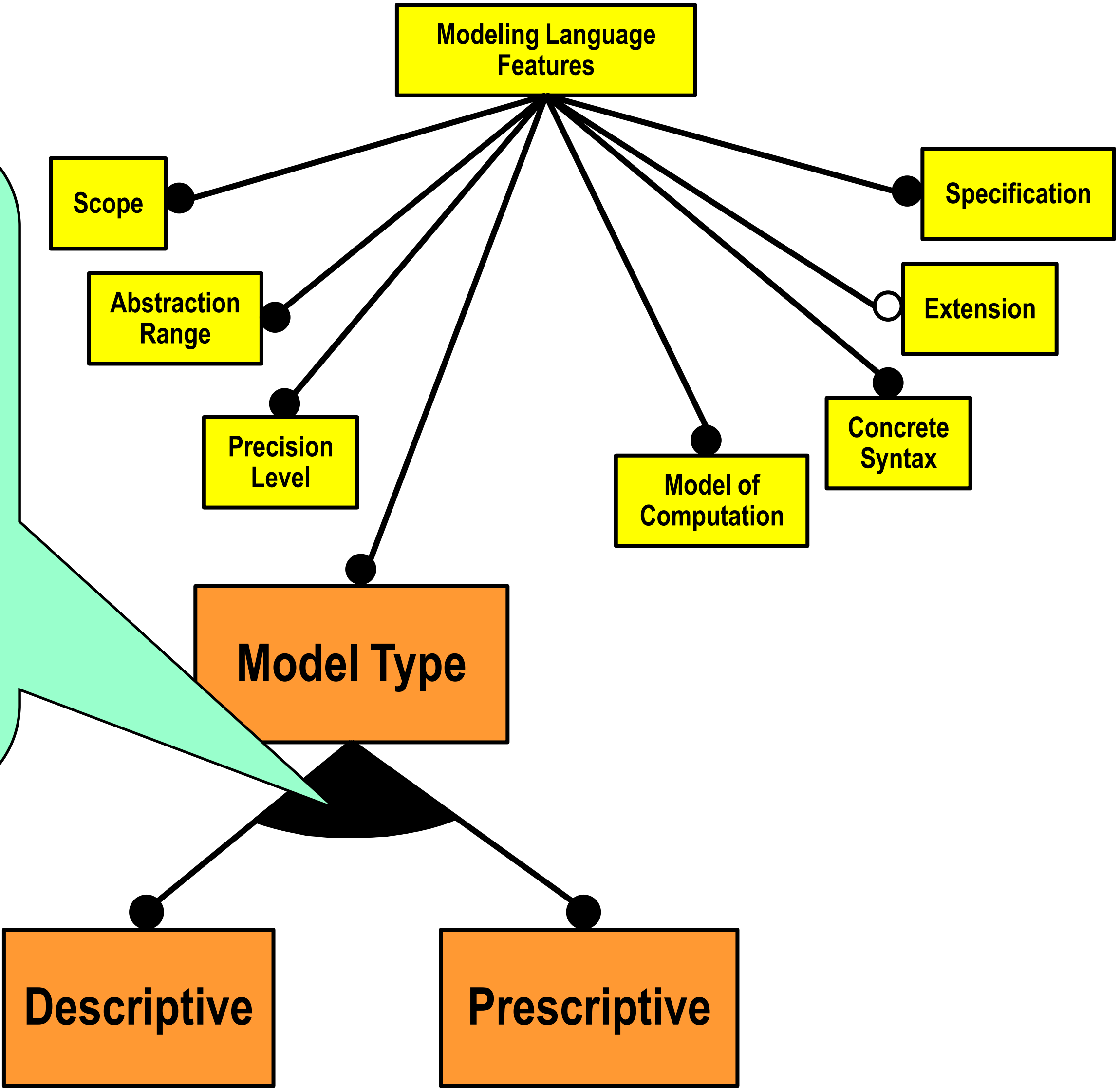
Precision Level Categories

- A more refined categorization based on degree of "formality"
  - Precision of definition, internal consistency, completeness, level of detail covered

Category	*Characteristics*	*Primary Purpose*
IMPLEMENTATION	Defined, formal, consistent, complete, detailed	Prediction, Implementation
EXECUTABLE	Defined, formal, consistent, complete	Analysis, Prediction
PRECISE	Defined, formal, consistent	Analysis, Prediction
CODIFIED	Defined, informal	Documentation, Analysis
AD HOC	Undefined, informal	Documentation, Analysis (no reuse)

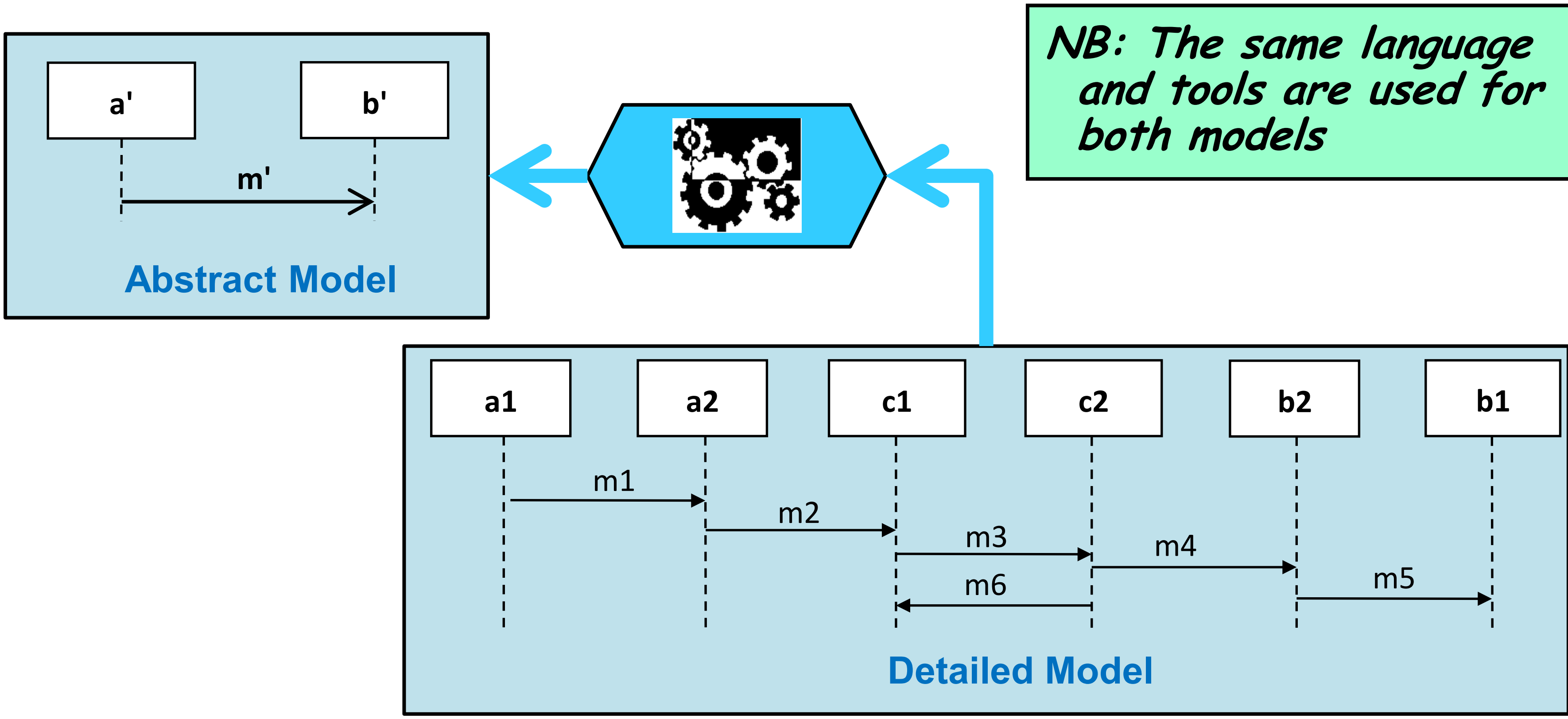
Selecting a Model Type

With the appropriate choice of Abstraction Range and Precision Level in combination with suitable model transforms, it is possible to define languages that support both types of models

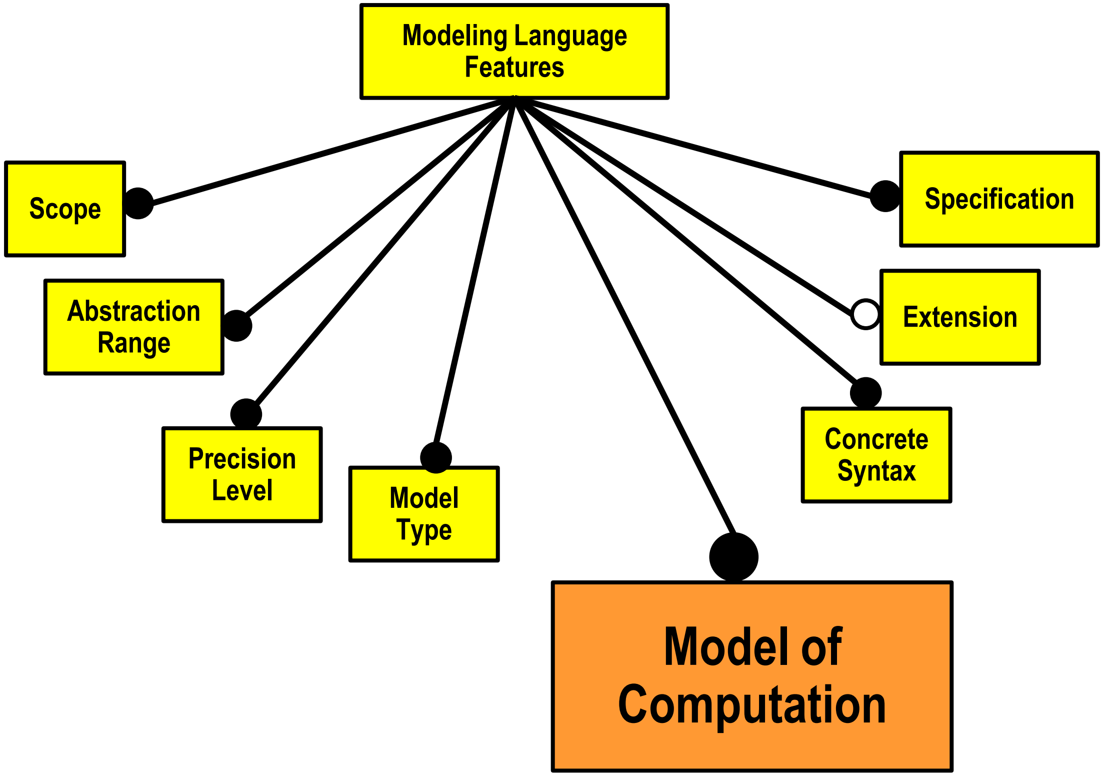


Pragmatics: Multiple Models Needed

- In reality, it is generally not practical to have a single model that covers all possible levels of abstraction
- But, it is possible to formally (i.e., electronically) couple different models via persistent model transforms



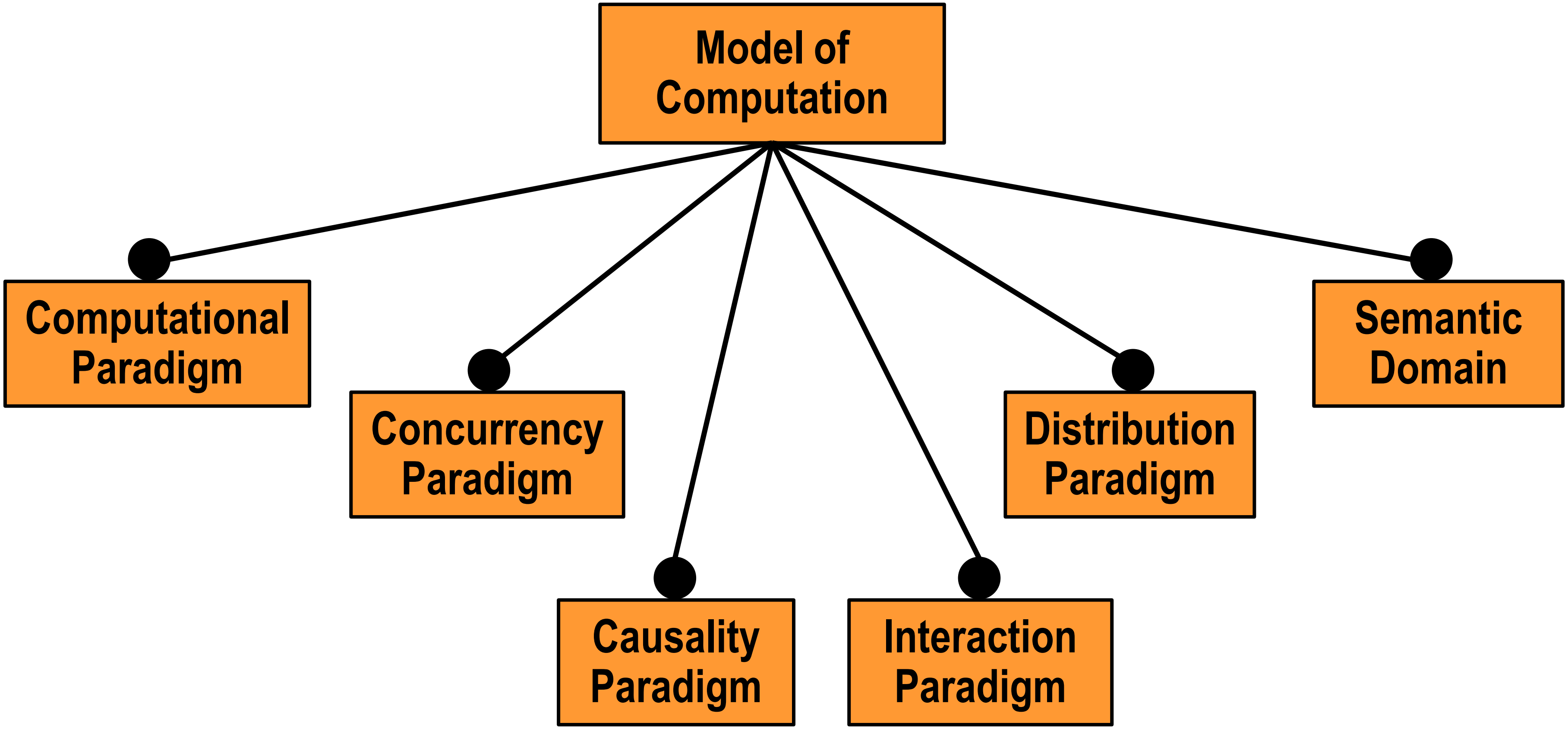
Selecting a Model of Computation



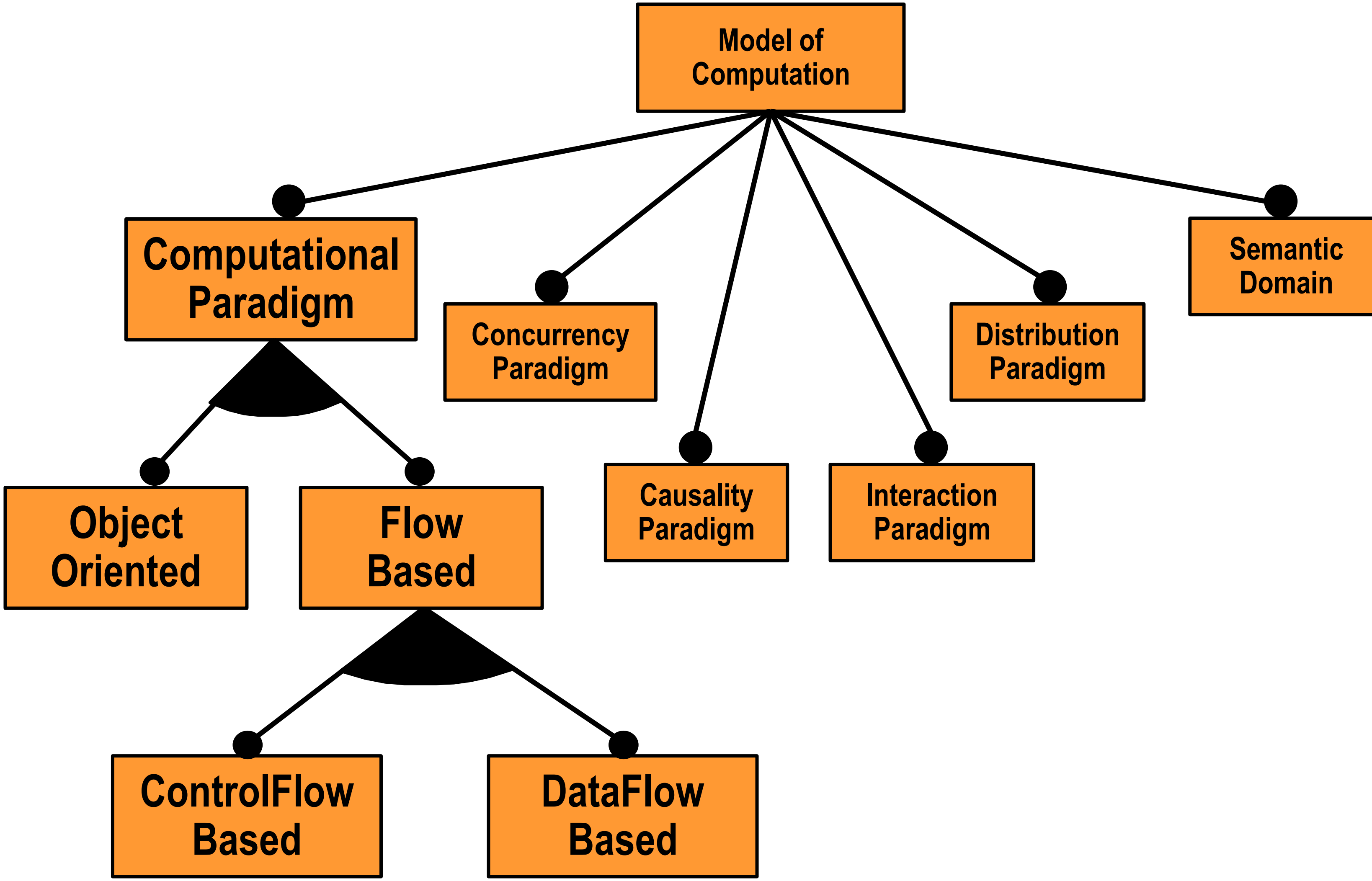
- **Model of Computation**: A conceptual framework (paradigm) used to specify how a (software) system realizes its prescribed functionality
  - Where and how does behavior (i.e., computation) occur
  - Derived usually from domain semantics

Key Dimensions of MoC

- Involves a number of inter-related decisions



Selecting a Computational Paradigm



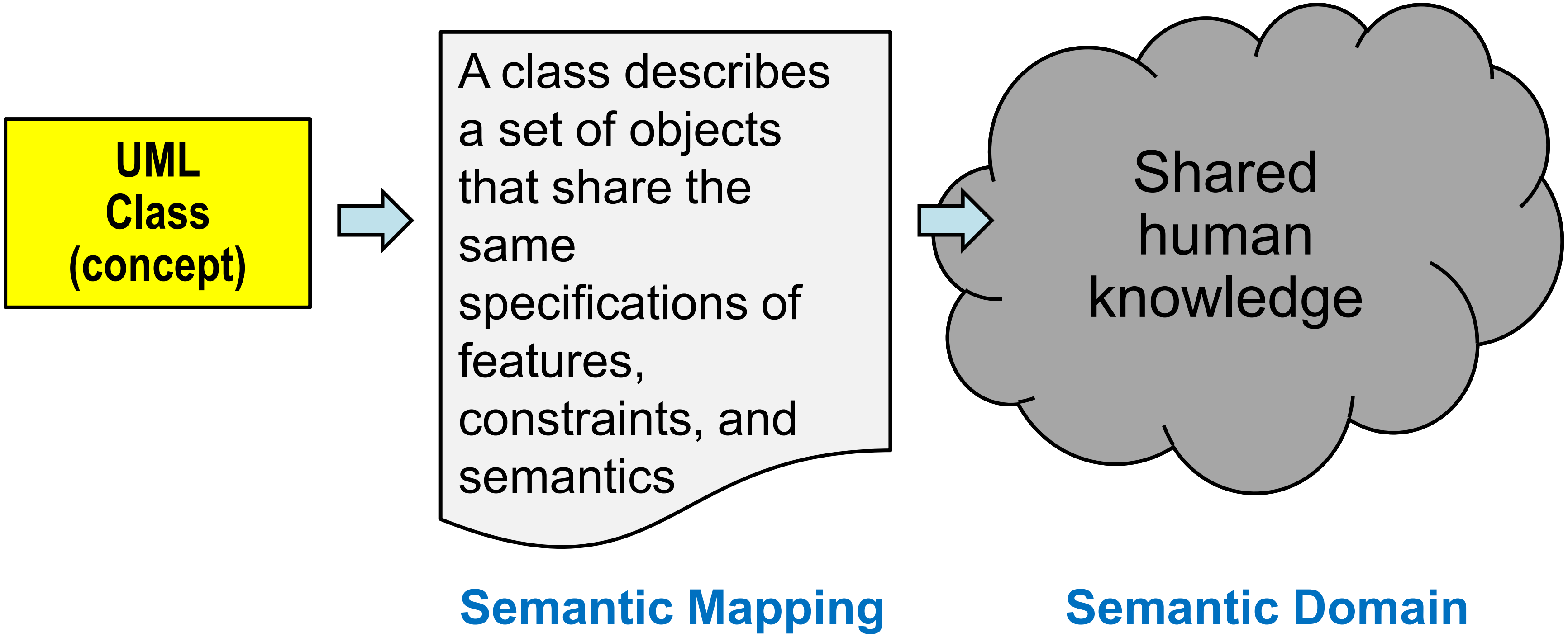
Other MoC Dimensions

- **Concurrency paradigm**: does computation occur sequentially (single thread) or in parallel (multiple threads)?
- **Causality paradigm**: what causes behavior
  - event driven, control driven, data driven (functional), time driven, logic driven, etc.
- **Execution paradigm**: nature of behavior execution
  - Synchronous (discrete), asynchronous, mixed (LSGA)
- **Interaction paradigm**: how do computational entities interact
  - synchronous, asynchronous, mixed
- **Distribution paradigm**: does computation occur in a single site or multiple?
  - Multisite (⇒ concurrent execution) vs. single site
  - If multisite: Coordinated or uncoordinated (e.g., time model, failure model)?

NB: These choices require a deep understanding of computing technology and cannot be made easily by non-experts

Semantics

- The meaning of language concepts
- Specified by relating them to concepts of a "well-understood" different domain
  - E.g.,



"Formal" Semantics

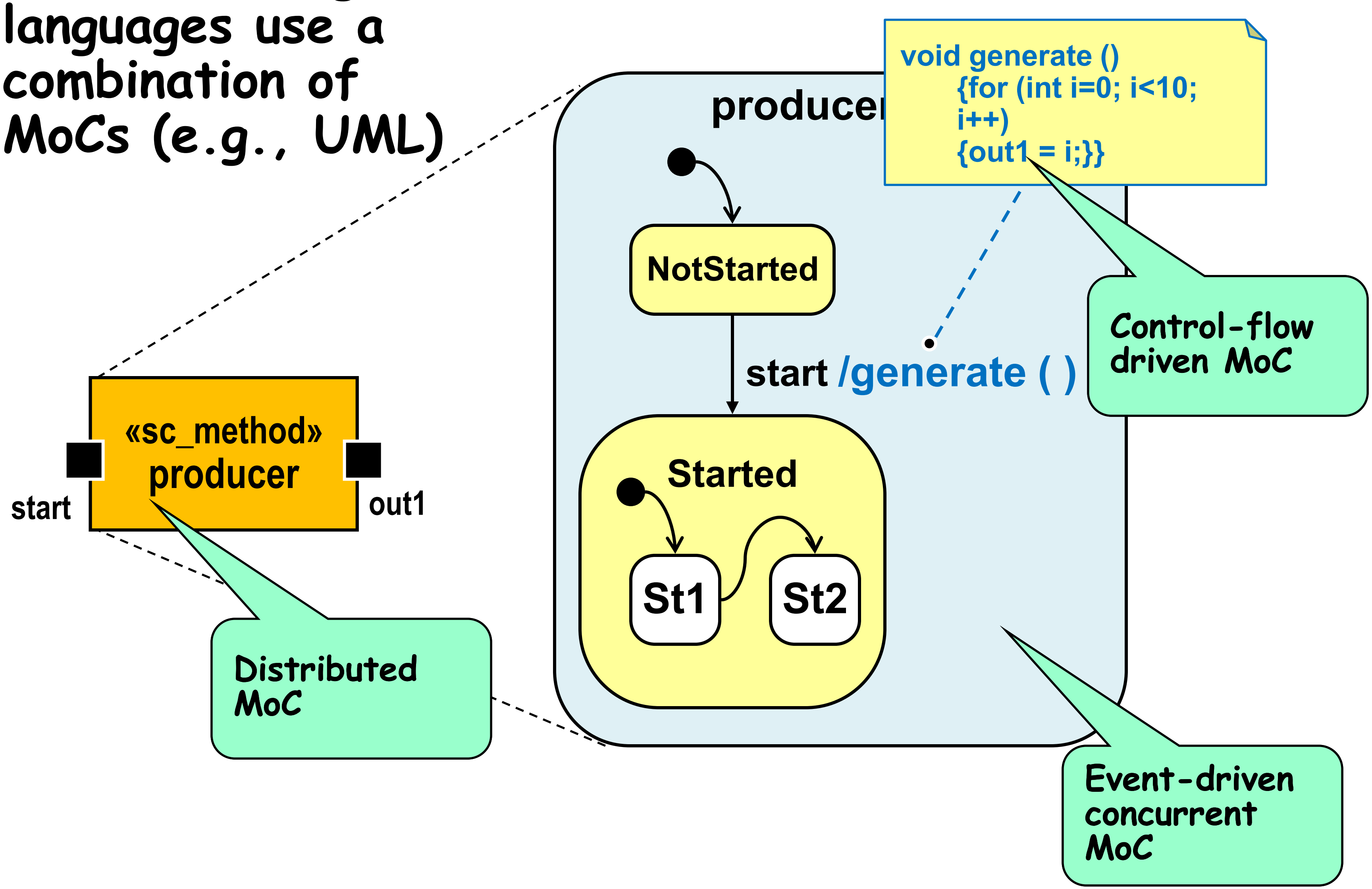
- The mapping and the domain are defined using <u>precisely defined</u> domains and mappings
  - Formal mathematical frameworks (e.g., first-order logic, abstract state machines, process algebras, IO streams, etc.) or
  - Executable computer languages (e.g., Java, Prolog)
    - Which may themselves have a formal semantics definition
- Example:
  - Base UML (bUML) is defined in terms of mappings to the Process Specification Language (PSL), which is itself based on situational calculus and first order logic
  - Foundational UML (fUML) is defined operationally as a bUML program

Selecting a Semantics Domain

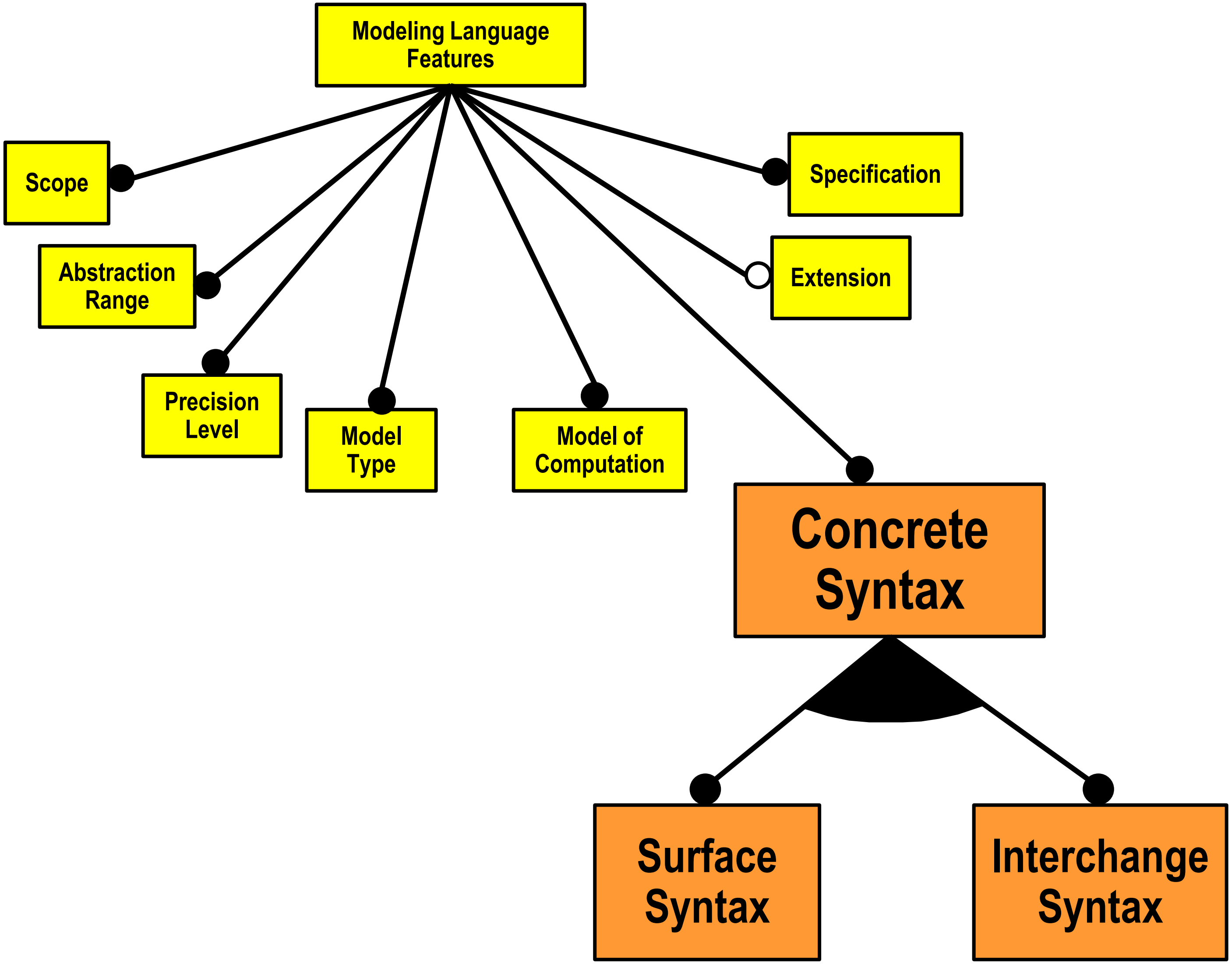
- **Avoid sophisticated mathematical formalisms**
  - Difficult to understand and verify (unless suitable tools are available)
  - Operational methods are generally preferred in practice
- **Choose a domain with existing tool support**
  - Enables verification of the semantics specification itself
  - Enables verification/prediction of model properties
  - Examples:
    - *Abstract state machines*
    - *Temporal Logic of Actions*
    - *Process Specification Language*

Pragmatics: Multiple (Nested) MoCs

- Some modeling languages use a combination of MoCs (e.g., UML)



Selecting a Concrete Syntax



State of the Art

"Very little is documented about why particular graphical conventions are used. Texts generally state what a particular symbol means without giving any rationale for the choice of symbols or saying why the symbol chosen is to be preferred to those already available. The reasons for choosing graphical conventions are generally shrouded in mystery." [S. Hitchman]*

* **S. Hitchman, "The Details of Conceptual Modeling Notations are Important – A Comparison of Relationship Normative Language", Comms. of the AIS, 9, 2002.**

Concrete Syntax Design

- **Two main forms:**
  - For computer-to-computer interchange (e.g., XMI)
  - For human consumption – "surface" syntax
- **Designing a good surface syntax is the area that we understand least**
  - If a primary purpose of models is communication and understanding, what syntactical forms should we use for a given language?
  - D. Moody, "The 'Physics' of Notations: Toward a Scientific Basis for Constructing Visual Notations in Software Engineering", IEE Transactions on Software Engineering, vol. 35, no.6, Nov./Dec. 2009
- **Requires multi-disciplinary skills**
  - Domain knowledge
  - Computer language design
  - Cognitive science
  - Psychology
  - Cultural Anthropology
  - Graphic design
  - Computer graphics

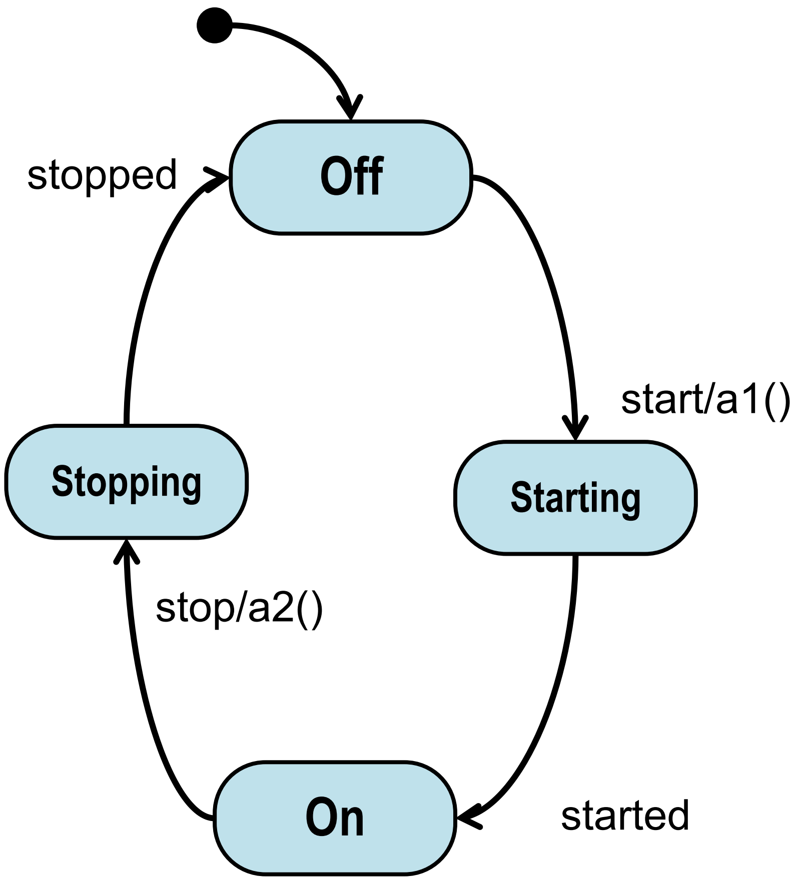
A Couple of Thoughts on Graphics

- "Whenever someone draws a picture to explain a program, it is a sign that something is not understood." – E. Dijkstra*
- "Yes, a picture is what you draw when you are trying to understand something or trying to help someone understand." – W. Bartussek*

* Quoted in D.L. Parnas, "Precisely Annotated Hierarchical Pictures of Programs", McMaster U. Tech Report, 1998.

Text vs. Graphics: Example

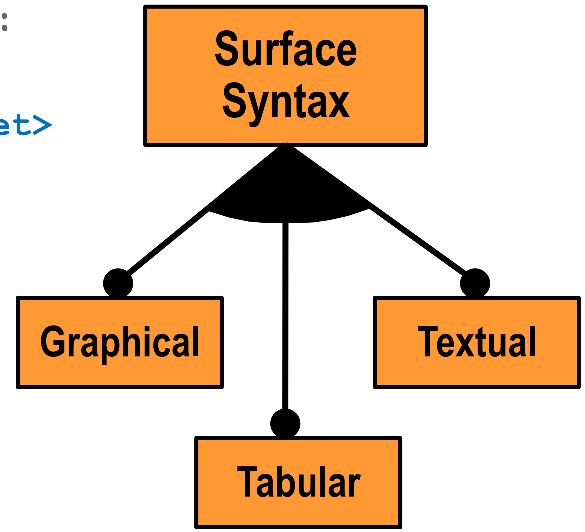
```
State: Off, On, Starting, Stopping;
Initial: Off;

Transition:
    {source: Off;
     target: Starting;
     trigger: start;
     action: a1();}
Transition:
    {source: Starting;
     target: On;
     trigger: started;}
Transition:
    {source: On:
     target: Stopping;
     trigger: stop;
     action: a2();}
Transition:
    {source: Stopping;
     target: Off;
     trigger: stopped;}
```



Surface Syntax

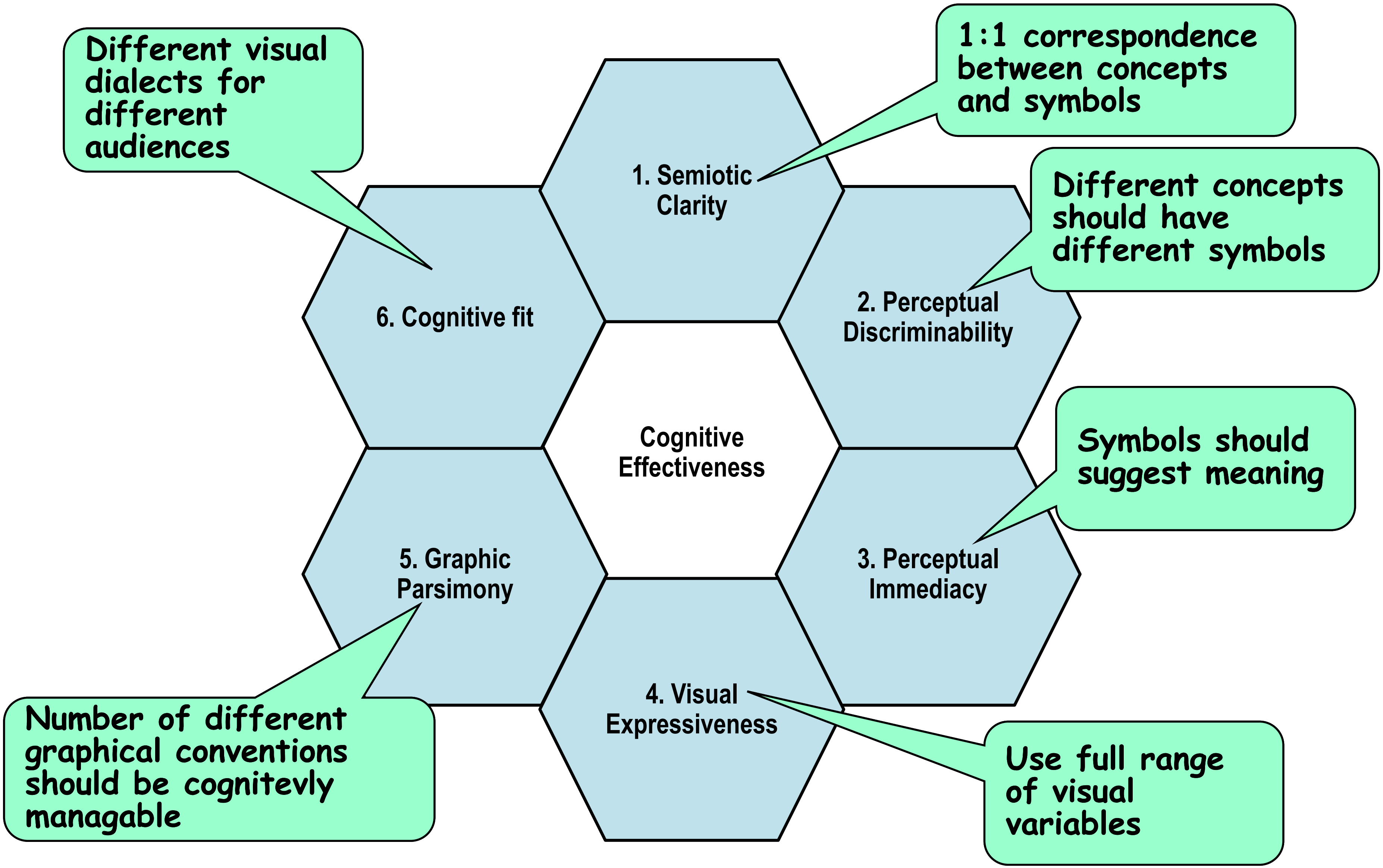
- **Textual forms**
  - Same as for programming languages: linear sequence of symbols
  - Usually specified as a type of BNF with terminals; e.g.:

```
<add-statement> ::= 'ADD' <left-bracket>
                     <[arguments-list]> <right-bracket>
<left-bracket> ::= '('
```



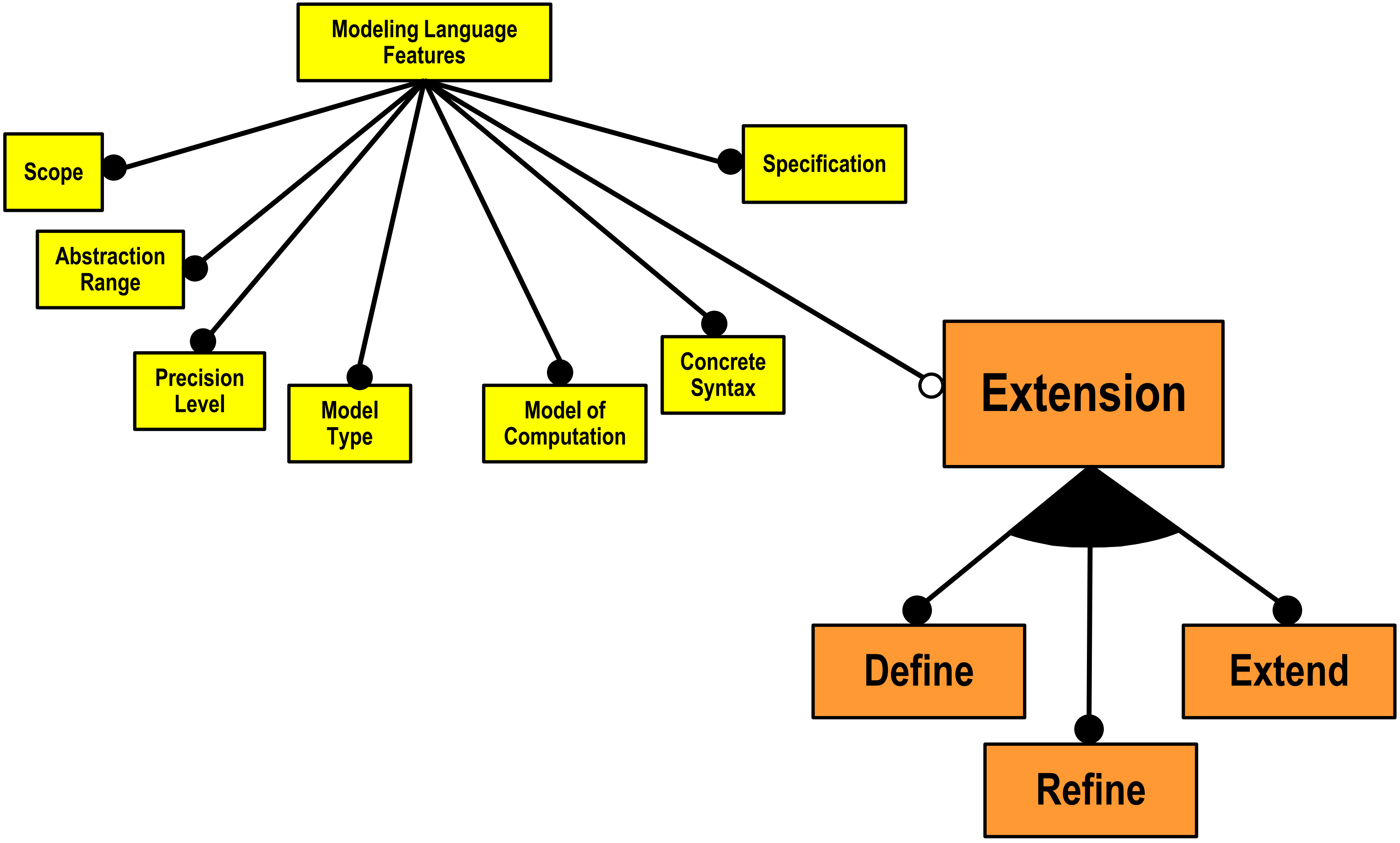
- **Tabular forms**
  - Constrained 2-dimensional
  - E.g., spreadsheets, Parnas tables
- **Graphical forms**
  - More complex: unconstrained 2-dimensional
    - Actually 2.5 dimensional – concept of Z-dimensions (overlapping graphics)
  - More flexible: user can choose which parts of the model to represent and how!
    - E.g., shape, line/fill styles, x-y position, size, font, etc.

Guidelines for Effective Visual Design



* D. Moody and J.v.Hillegersberg, “Evaluating the Visual Syntax of UML: An Analysis of the Cognitive Effectiveness of the UML Suite of Diagrams”,

Extending an Existing Language?

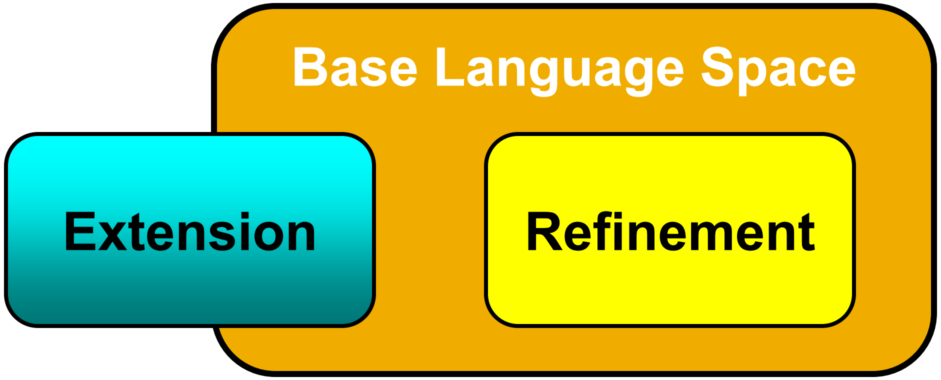


Approaches to DSML Design

1. Define a completely new language from scratch
2. Extend an existing language: add new domain-specific concepts to an existing (base) language
3. Refine an existing language: specialize the concepts of a more general existing (base) language

Refinement vs Extension

- Semantic space = the set of all valid programs that can be specified with a given computer language
- Refinement: subsets the semantic space of the base language (e.g., UML profile mechanism)
  - Enables reuse of base-language infrastructure
- Extension: intersects the semantic space of the base language



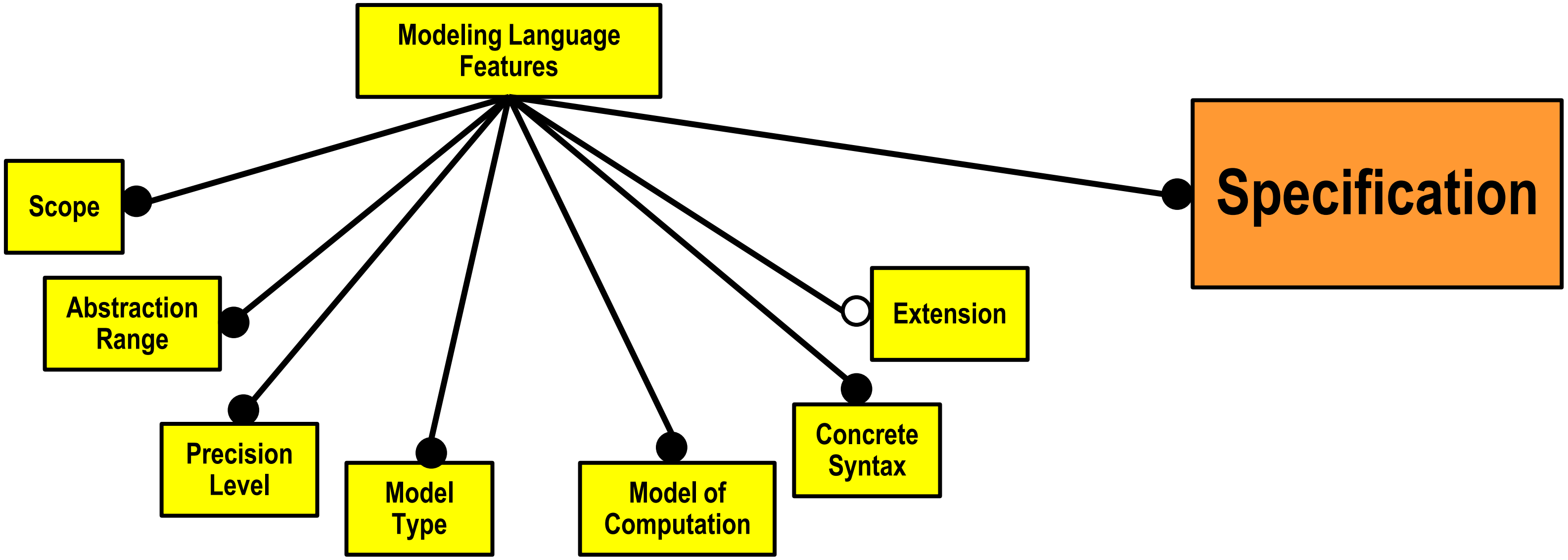
Comparison of Approaches

Approach	Expressive Power	Ease of Lang.Design	Infrastructure Reuse	Multimodel Integration
New Language	*High*	*Low*	*Low*	*Low*
Extension	*Medium*	*Medium*	*Medium*	*Medium*
Refinement	*Low*	*High*	*High*	*High*

Define, Refine, or Extend?

- **Depends on the problem at hand**
  - Is there significant semantic similarity between the base language metamodel and the new language metamodel?
    - Does every new language concept represent a semantic specialization of some base language concept?
    - No semantic or syntactic conflicts?
  - Is language design expertise available?
  - Is domain expertise available?
  - Cost of establishing and maintaining a language infrastructure?
  - Need to integrate models with models based on other DSMLs?
- **The ability to reuse the infrastructure of a language has often led to refinement or extension as the preferred choice**
  - Not necessarily optimal from a purely technical viewpoint
  - E.g., Z.109 (SDL profile of UML), SysML4Modelica (SysML profile), SystemC (UML profile), AADL (UML profile), MoDAF/DoDAF (UML profile)…

Selecting a Language Specification Method



- What methods should be used to specify a modeling language?

Summary: Modeling Language Design

- Modeling language design is still much more of an art than a science
  - Few reference texts; no consensus
- Doing it well requires a rare combination of skills:
  - Understanding of modeling technologies, computer language technologies, domain knowledge, and even non-technical knowledge such as cognitive psychology
  - Many complex technical and non-technical design choices and tradeoffs need to be made
- DSMLs are an important and inevitable trend, but the often advertised notion of "end-user language design" is far from practical reality

Bibliography/References

- A. Kleppe, "Software Language Engineering", Addison-Wesley, 2009
- T. Clark et al., "Applied Metamodeling – A Foundation for Language Driven Development", (2nd Edition), Ceteva, http://www.eis.mdx.ac.uk/staffpages/tonyclark/Papers/
- S. Kelly and J.-P. Tolvanen, "Domain-Specific Modeling: Enabling Full Code Generation," John Wiley & Sons, 2008
- J. Greenfield et al., "Software Factories", John Wiley & Sons, 2004
- D. Harel and B. Rumpe, "Meaningful Modeling: What's the Semantics of 'Semantics'", IEEE Computer, Oct. 2004.
- E. Seidewitz, "What Models Mean", IEEE Software, Sept./Oct. 2003.
- T. Kühne, "Matters of (Meta-)Modeling, Journal of Software and Systems Modeling, vol.5, no.4, December 2006.
- Kermeta Workbench (http://www.kermeta.org/)
- OMG's Executable UML Foundation Spec (http://www.omg.org/spec/FUML/1.0/Beta1)
- UML 2 Semantics project (http://www.cs.queensu.ca/~stl/internal/uml2/index.html)
- ITU-T SDL language standard (Z.100) (http://www.itu.int/ITU-T/studygroups/com10/languages/Z.100_1199.pdf)
- ITU-T UML Profile for SDL (Z.109) (http://www.itu.int/md/T05-SG17-060419-TD-WP3-3171/en)

- THANK YOU -

QUESTIONS,
COMMENTS,
ARGUMENTS...