

Developing Embedded Software with Eclipse, State Machines, and Block Diagrams

about us ...



Andreas Unger
Software Engineer
andreas.unger@itemis.de



Axel Terfloth
Head R&D Embedded Systems
axel.terfloth@itemis.de

- work at itemis AG, Germany
- work on model driven development of embedded systems
- work on YAKINDU open source project

about itemis AG ...

Open Source

- Eclipse Strategic Member
- Eclipse Committer – Eclipse Modeling Project (Xpand, Xtext)
- others: YAKINDU, openArchitectureWare, CM3, formax, jmove



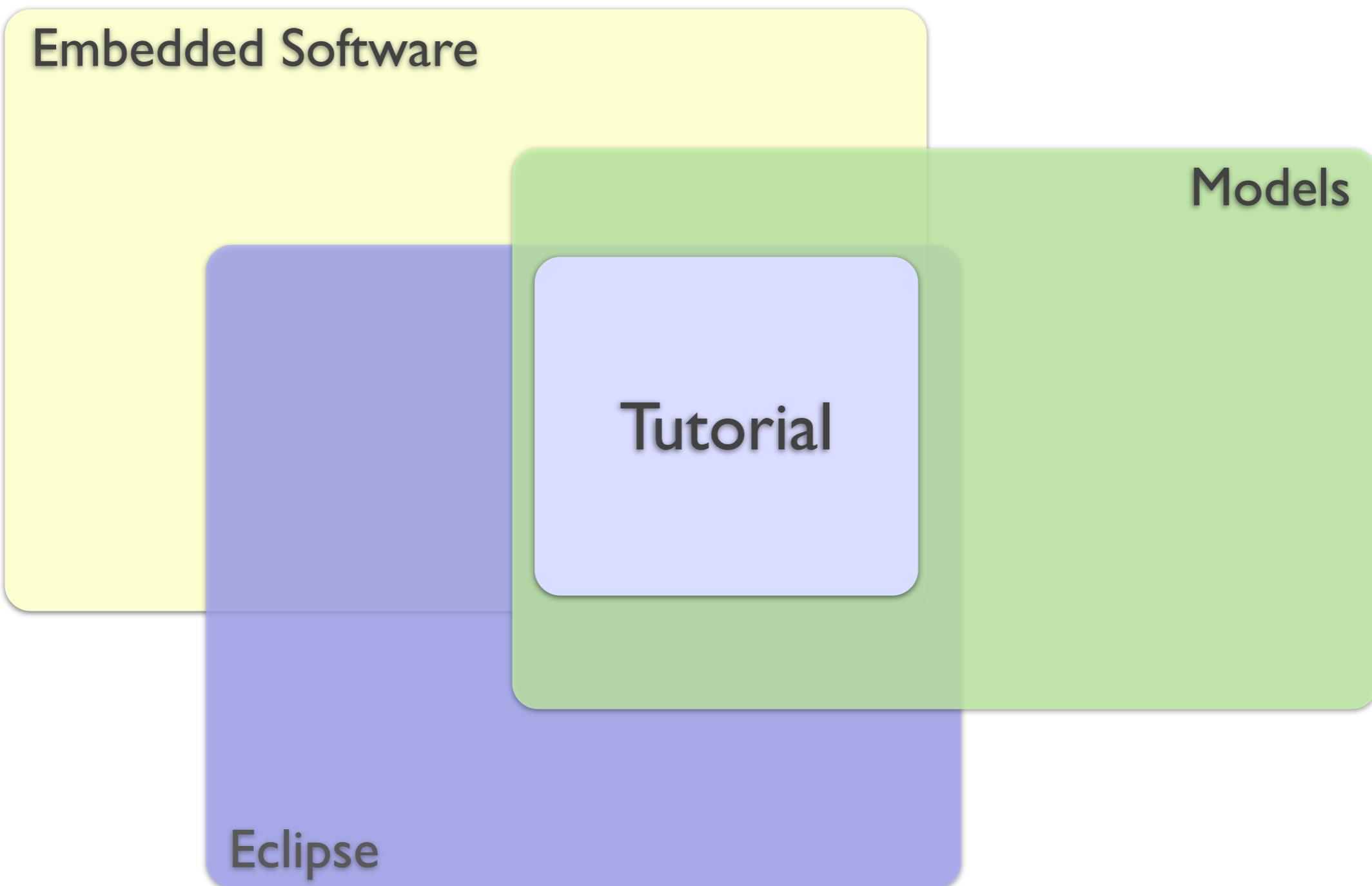
founded 2003, ± 140 people

Training, Coaching, Consulting

- Model Driven Software Development (MDSD)
- Processes, Methodology, & Tools
- Embedded Systems & Enterprise Systems



Tutorial Context



Goals...

- See how Eclipse can support Embedded Software Development beyond coding
- See how models can help dealing with increasing software complexity
- Understand how state machines work
- Understand how block diagrams work
- Understand the role of Eclipse technologies

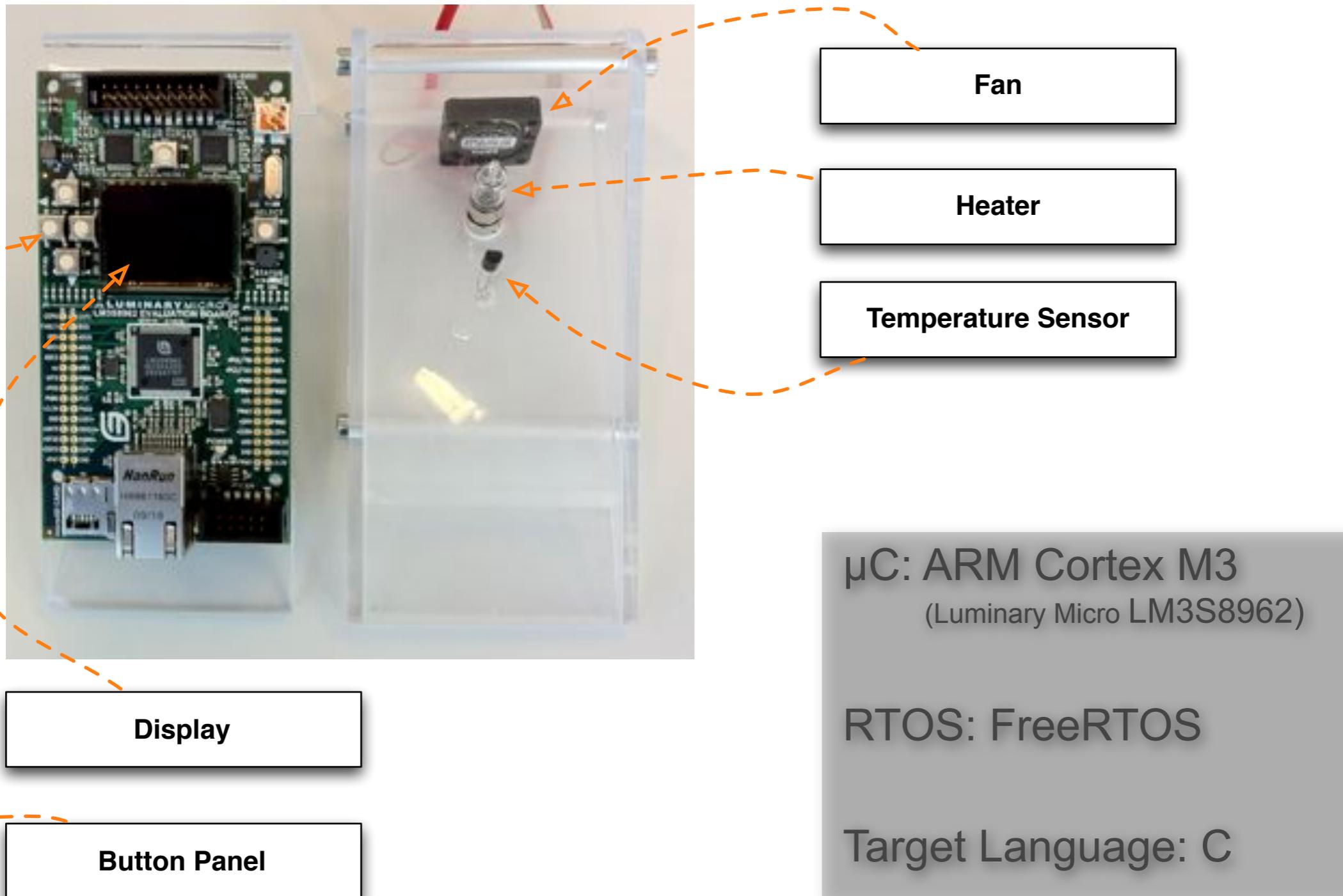


Agenda

- Anatomy of the Demo Application
- Model Driven Development
- The Toolchain
- Statecharts
- Blockdiagrams
- Component Model
- Generating Code

Anatomy of the Demo Application

... a Heating Control



Heating Control - Requirements



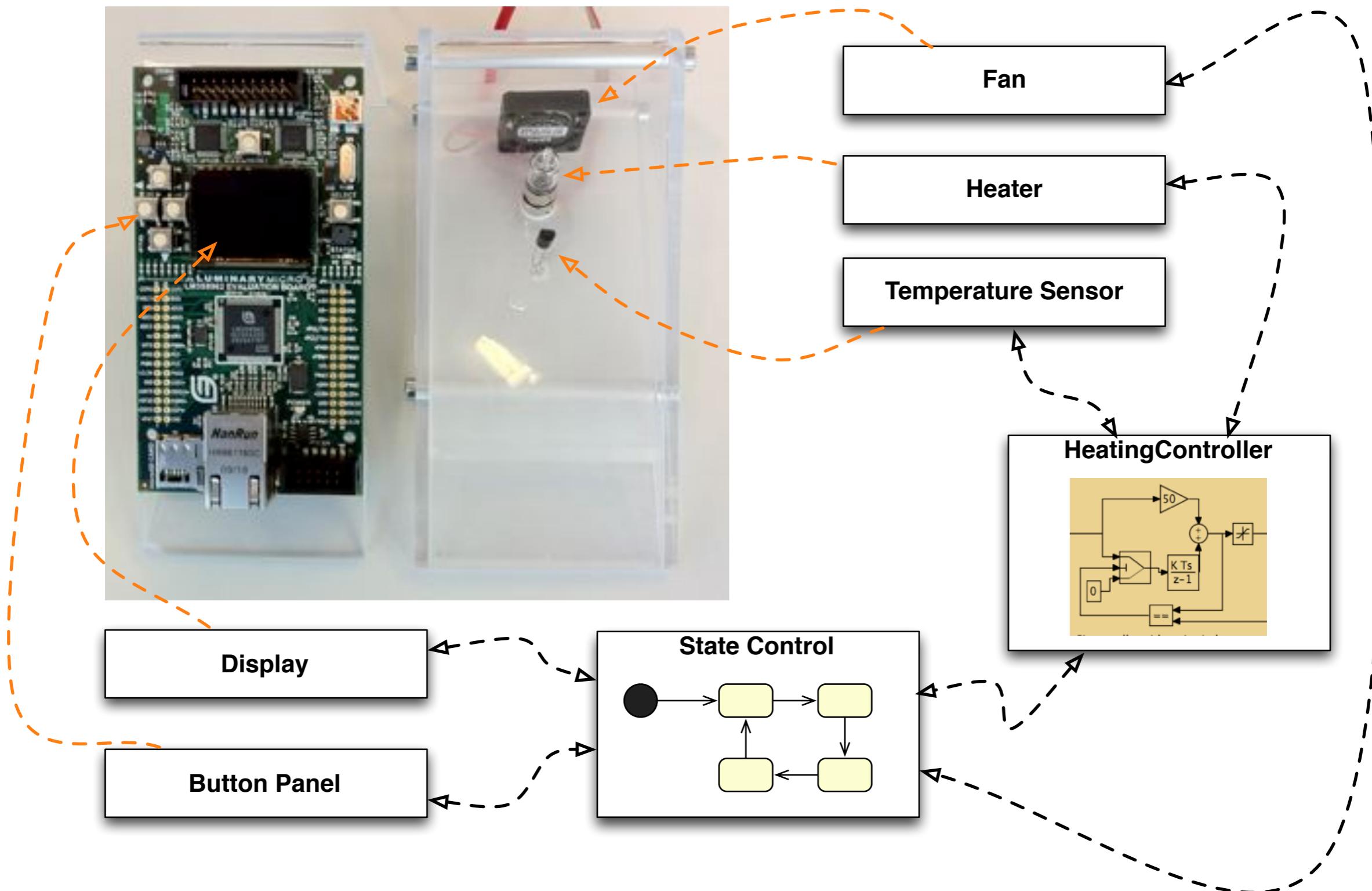
- The user can switch on and off the heating control using a button
- The user can define the temperature setpoint using the buttons.
- The heating control controls the temperature of the environment continuously
- In the case of a temperature drop the control must switch off the heater
- The system displays the current temperature, the temperature setpoint, the current heating power, and the current status of the system

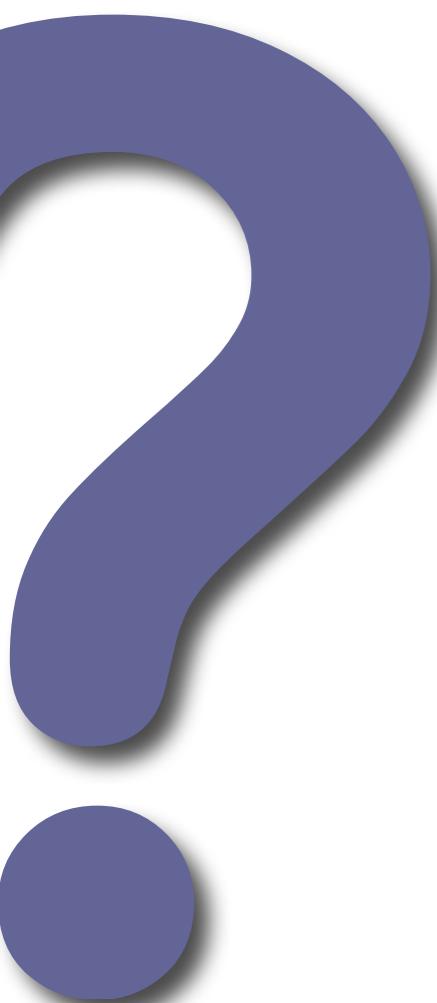
Heating Control - System Properties



- **Reactive:** system continuously interacts with the environment
- **System is event-driven (asynchronous inputs)**
- **System continuously processes data from the environment (isochronous inputs)**
- **Non deterministic environment**
- **State dependent:** the state of the system evolves over time depending on previous inputs and time
- **Real Time**

Heating Control - Software Structure

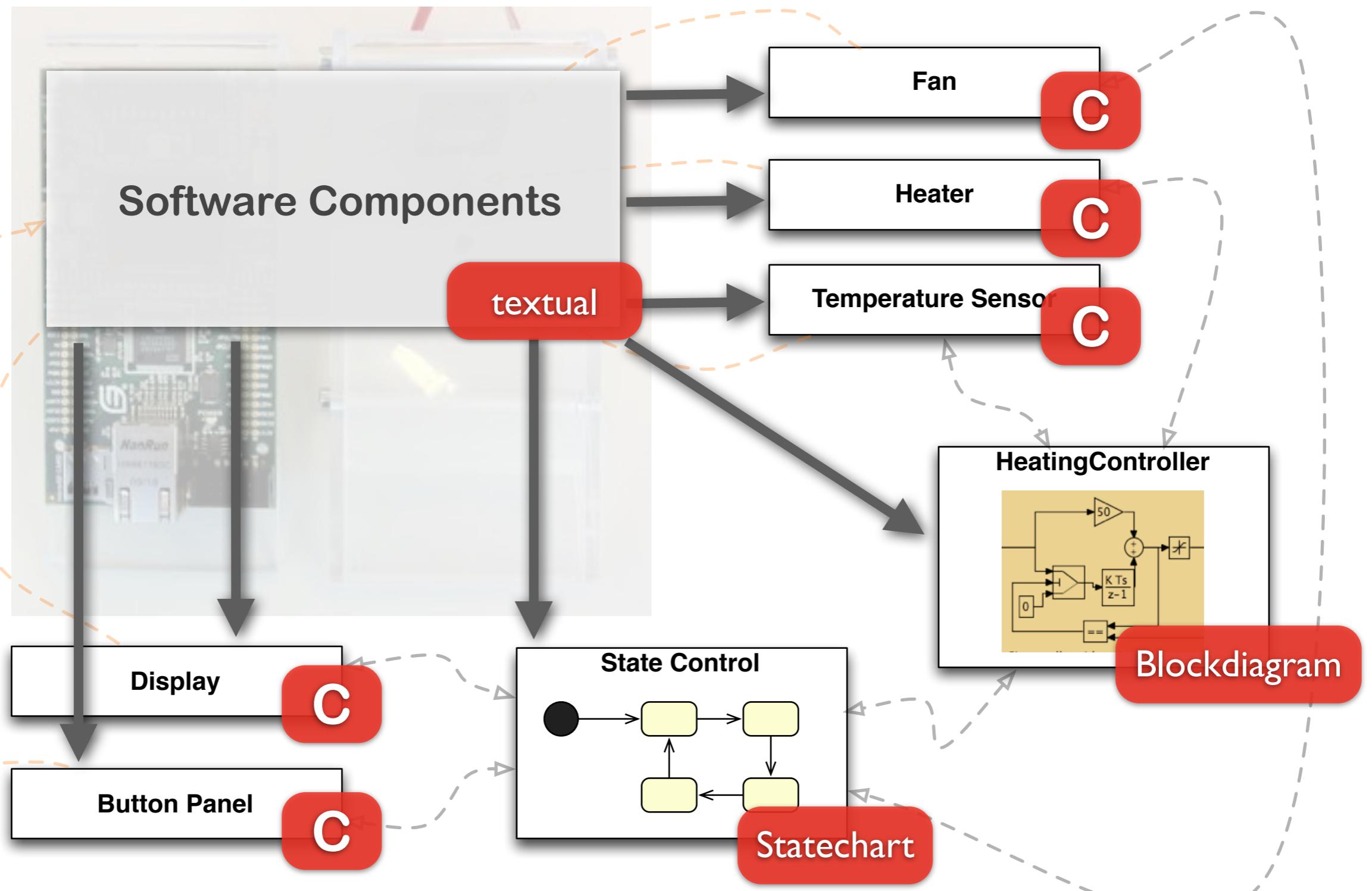




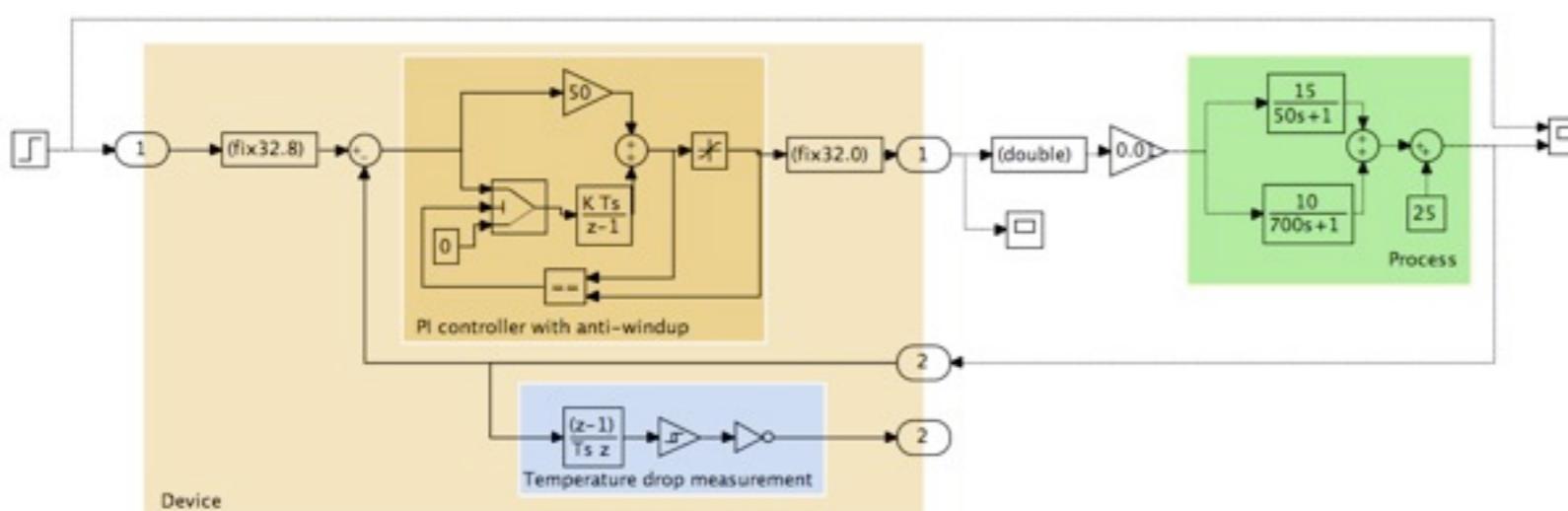
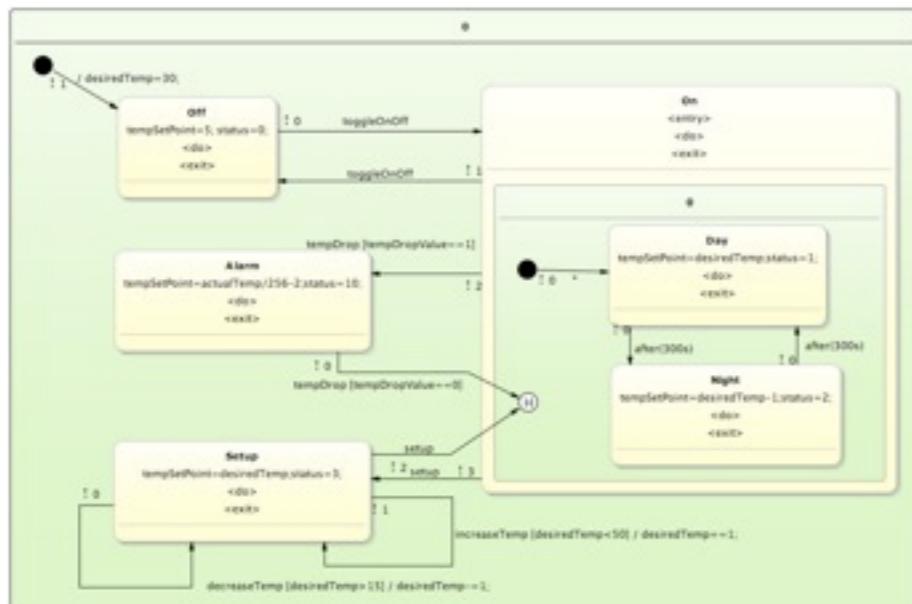
Statecharts vs. Blockdiagrams

What is the difference?

Heating Control - Software



The Models ...



```
//  
// define components  
//  
component Heater {  
    in power (queued);  
}  
  
component Fan {  
    in power (queued);  
}  
  
component TemperatureSensor {  
    out actualTemperature mapped to isr;  
}  
  
component ButtonPanel {  
    out select mapped to isr;  
    out up mapped to isr;  
    out down mapped to isr;  
    out left mapped to isr;  
    out right mapped to isr;  
}
```

Model Driven Development

Model Based Software Engineering

makes systematic use of (formal)
engineering models as primary
engineering artifacts throughout the
overall engineering life-cycle

Source: Nyß09

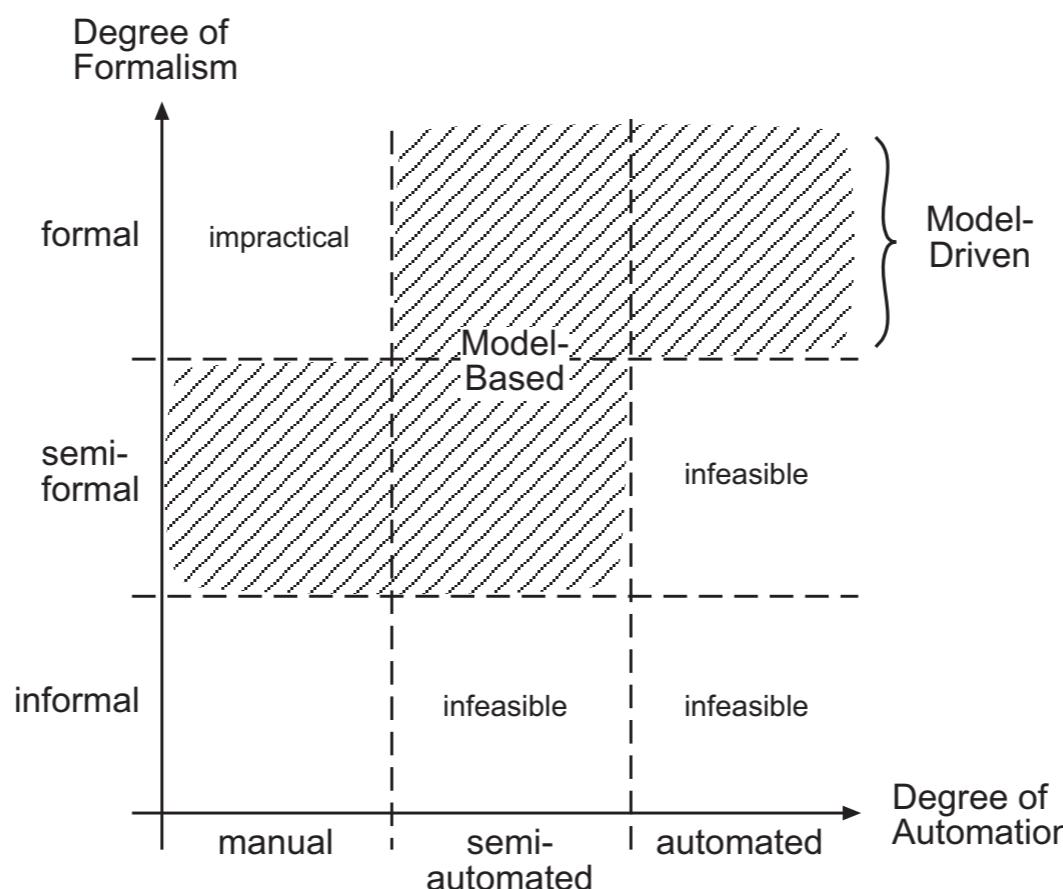
Model Driven Software Development (MDSD)

**is a generic term for techniques, that
transform formal models into
executable software.**

Source: Stahl/Völter

Model Based vs. Model Driven

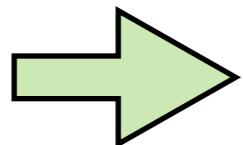
In contrast to model based approaches model driven approaches have a higher degree of automation.



The term 'model based' is often associated with the models in the domain of control engineering.

Modell Driven (Software) Development

- In the domain of software development mostly immaterial models are used
- Important property of software:



Software is immaterial !

- Idea:
 1. Create a descriptive Model of the problem domain,
 2. transform it into a prescriptive model of the solution domain
 3. and finally into the product itself.

Model Driven Development (MDD)

Simplification by abstraction

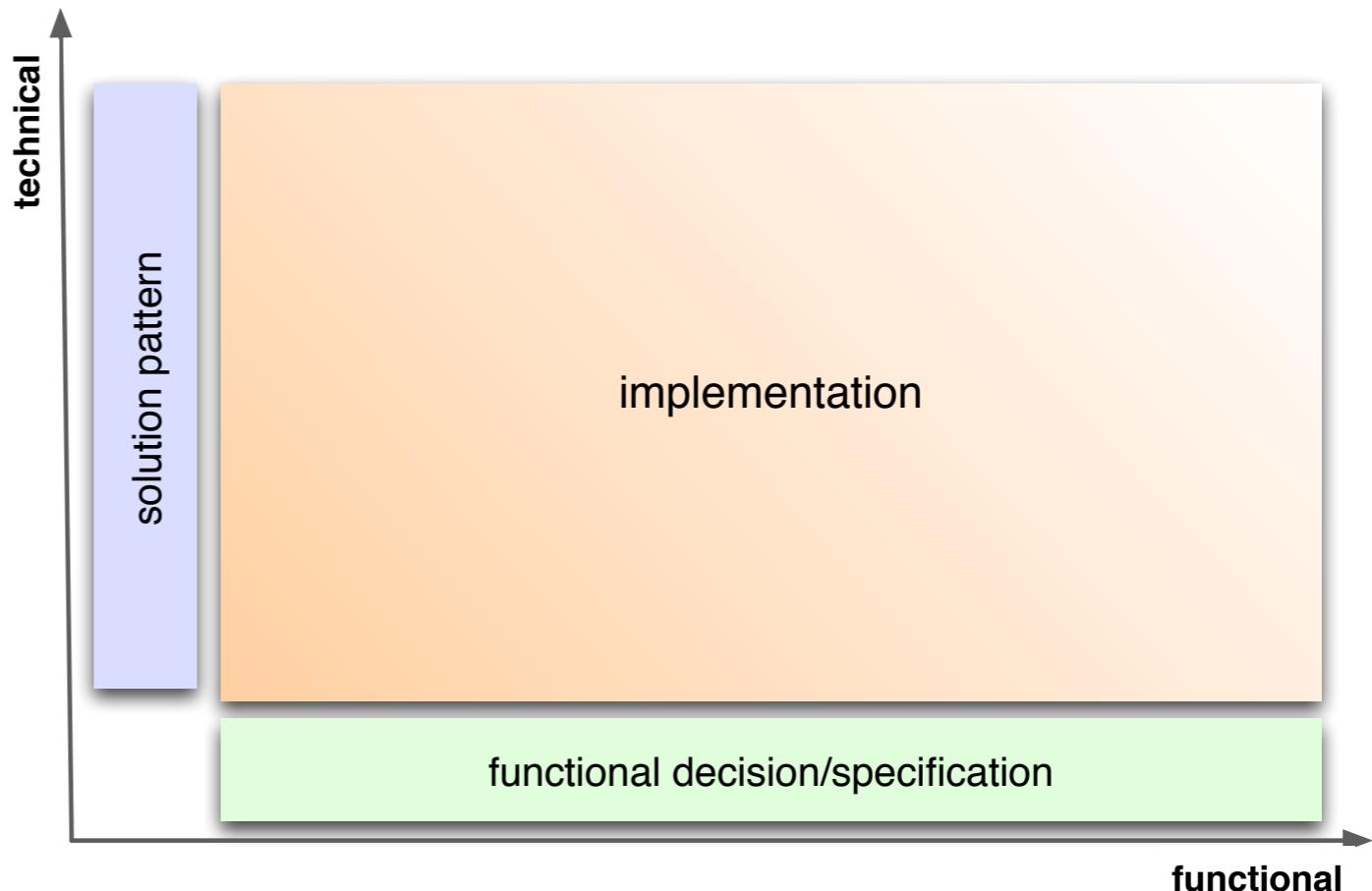
- hide complexity
- focus on domain relevant aspects
- Generator encapsulates technical details

Transformation to target platform using generators (or interpreters)

- avoid redundancy (DRY: Don't Repeat Yourself)
- enforce architecture and design guidelines
- improve quality / reduce error rate / fix errors once
- improve productivity
- more flexibility for variants

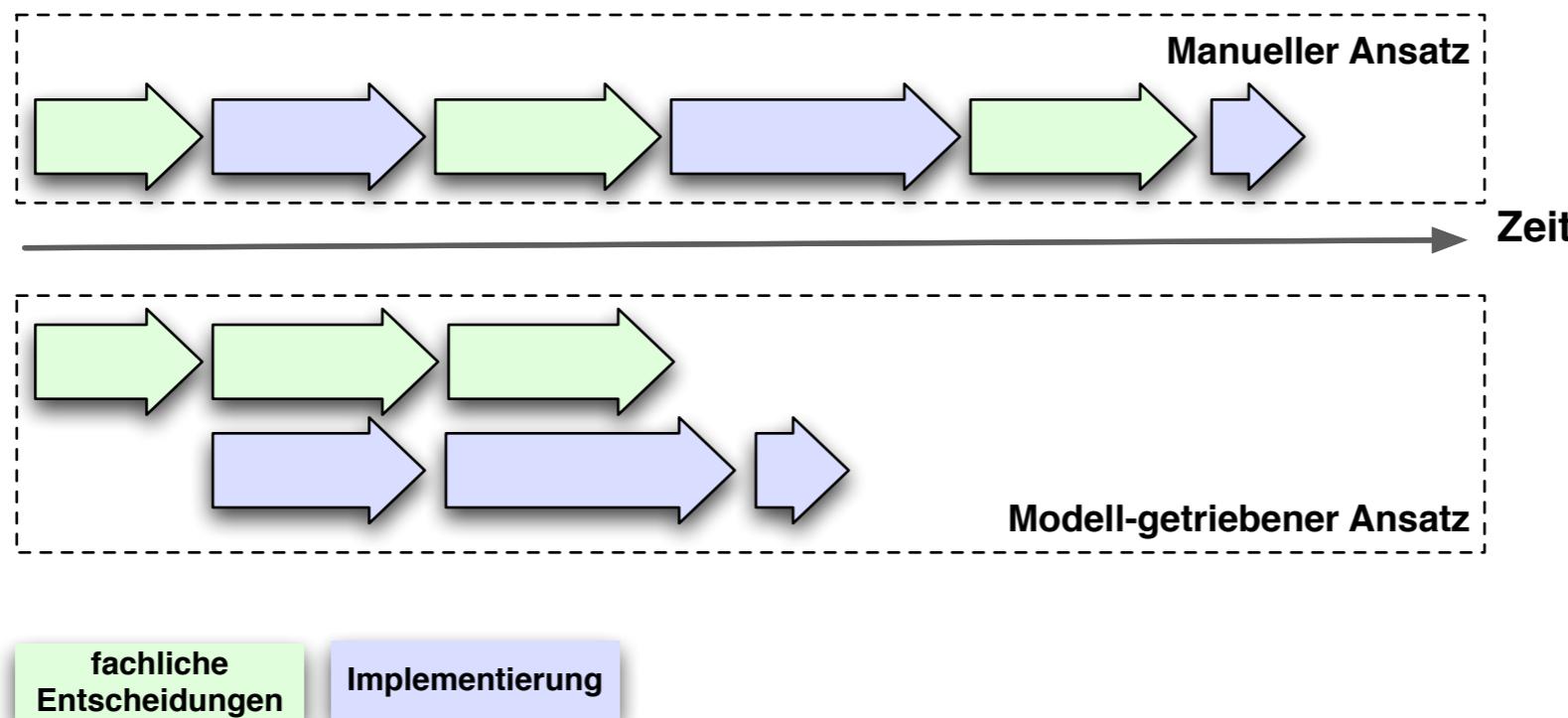
Separation of Concerns

- strict separation of functional and technical implementation aspects
- functional and technical aspects have different life-cycle
- concepts are more stable than technologies
- generator weaves functional and technical aspects into an implementation

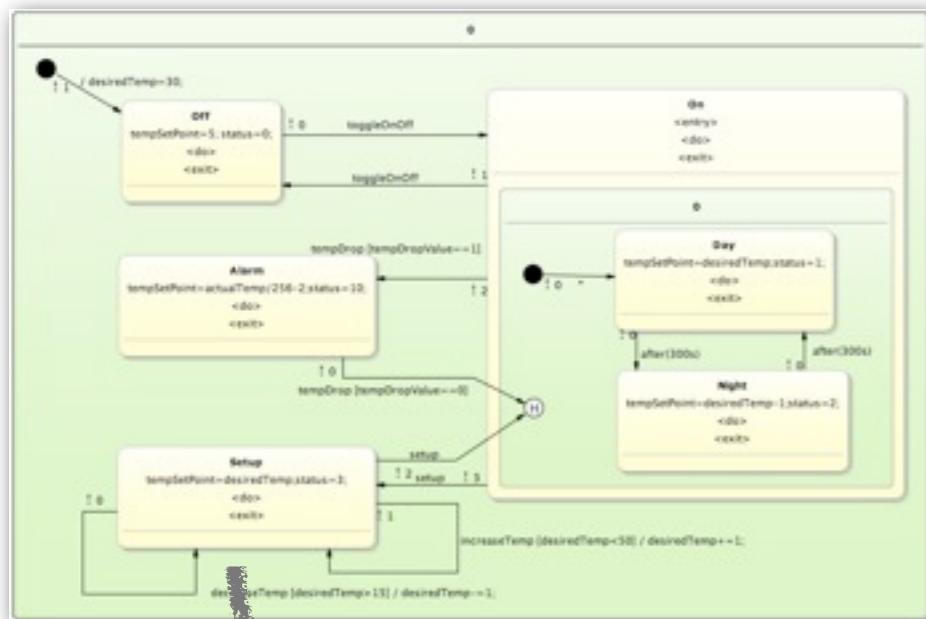


Separation of Concerns

- allows higher degree of parallelization



Abstraction Levels



model of

```
handler(void) {
    average = 0;
    counter = ADC_COUNTER;
    ...
    ADC, 0);
    ADC, 0, &value);
}
```

```
average += value;
if (--adcCounter == 0) {
    average = convertADCtoTemperature(
        FIX_DIV(FIX_FROM_INT(average),
            FIX_FROM_INT(ADC_COUNTER)));
    sensor_actualTemperature_send(&average);
    average = 0;
    adcCounter = ADC_COUNTER;
}
}
```

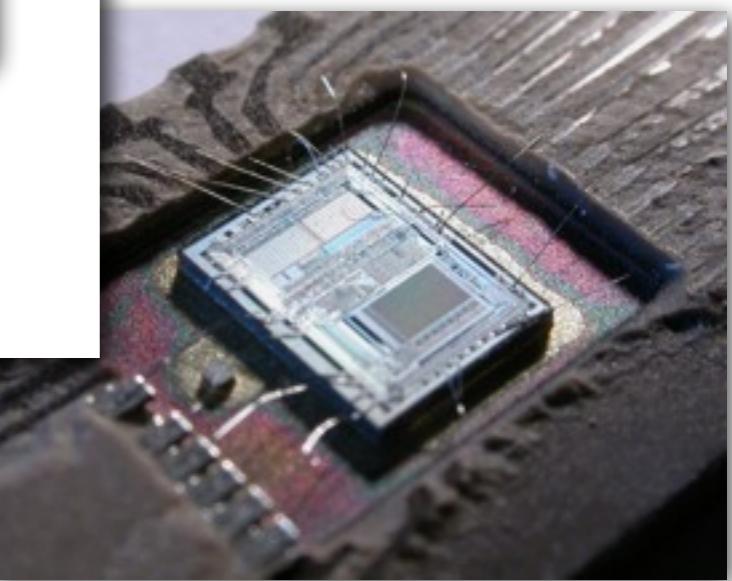
model of

```
lsl    r7, r4,
movt  r0, #:upper16:274877907
smull r7, r5, r0, ip
subs  r6, r2, r6
mov   r7, #2048000
mul   r6, r7, r6
asr   ip, ip, #31
lsls  r7, r2, #10
```

model of

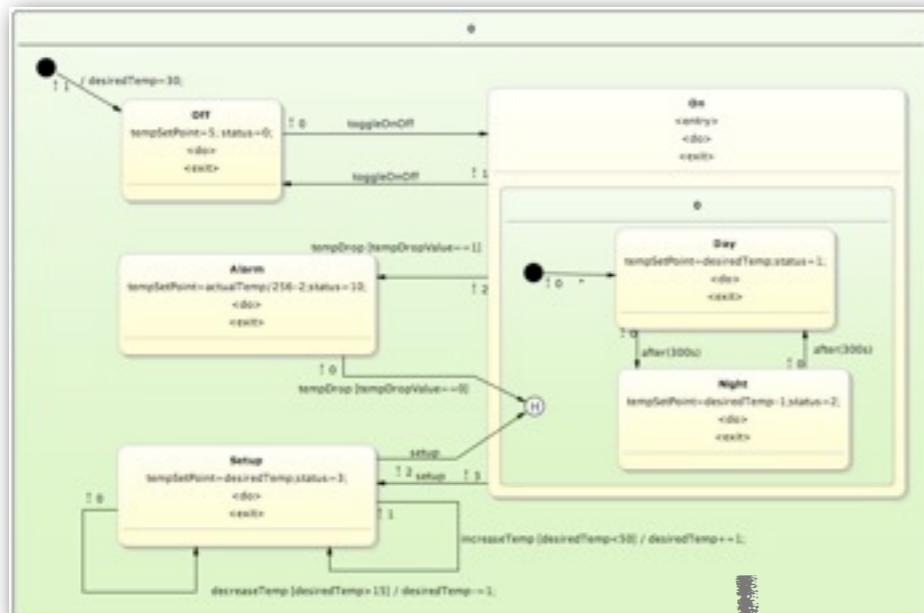
C code as an intermediate artifact

ASM



Abstraction Levels

model and c code for disjoint aspects



C

```
void adcSequence0Handler(void) {
    static int32_t average = 0;
    static int adcCounter = ADC_COUNTER;

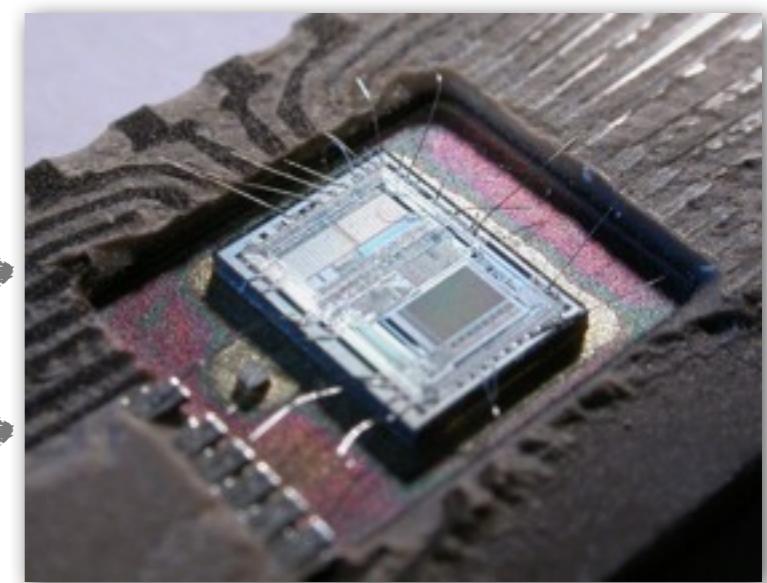
    unsigned long value;

    ADCIntClear(ADC0_BASE, 0);
    ADCSequenceData(ADC0_SEQUENCE, 0, &value);

    average += value;

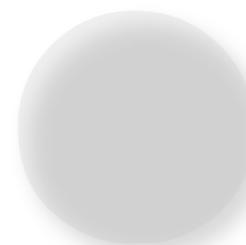
    if (--adcCounter == 0) {
        average = convertADCtoTemperature(
            FIX_DIV(FIX_FROM_INT(average),
                    FIX_FROM_INT(ADC_COUNTER)));
        sensor_actualTemperature_send(&average);
        average = 0;
        adcCounter = ADC_COUNTER;
```

model of





a model is code
code is a model

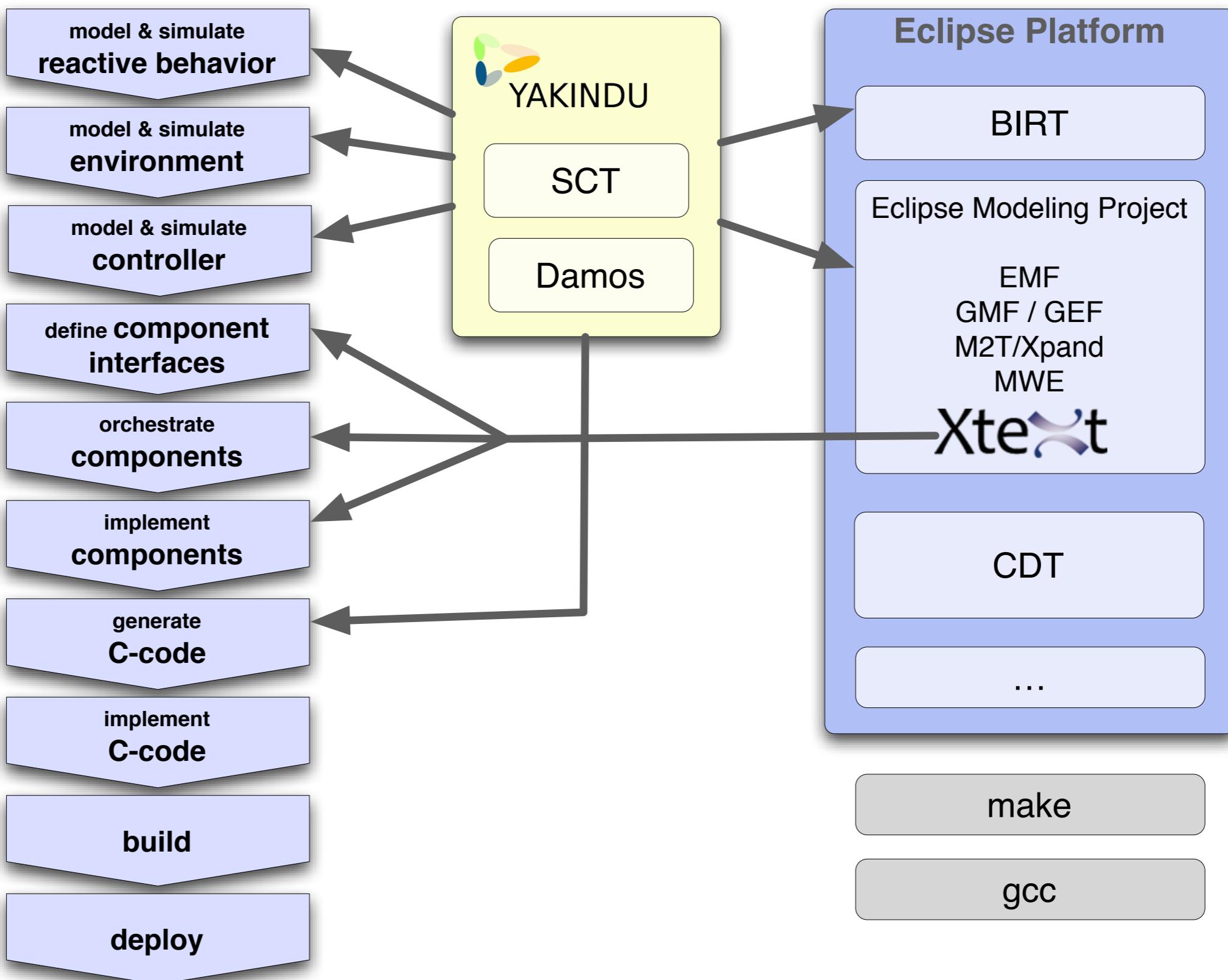


When is MDD applicable?

- same solution for the same problem
- schematic technical implementation
- differences can be specified in the model
- less effort for specification than for implementation
- critical mass will be reached
 - extent of models
 - frequency of change
 - number of target systems
 - many variants

The Toolchain

Activities and Tools



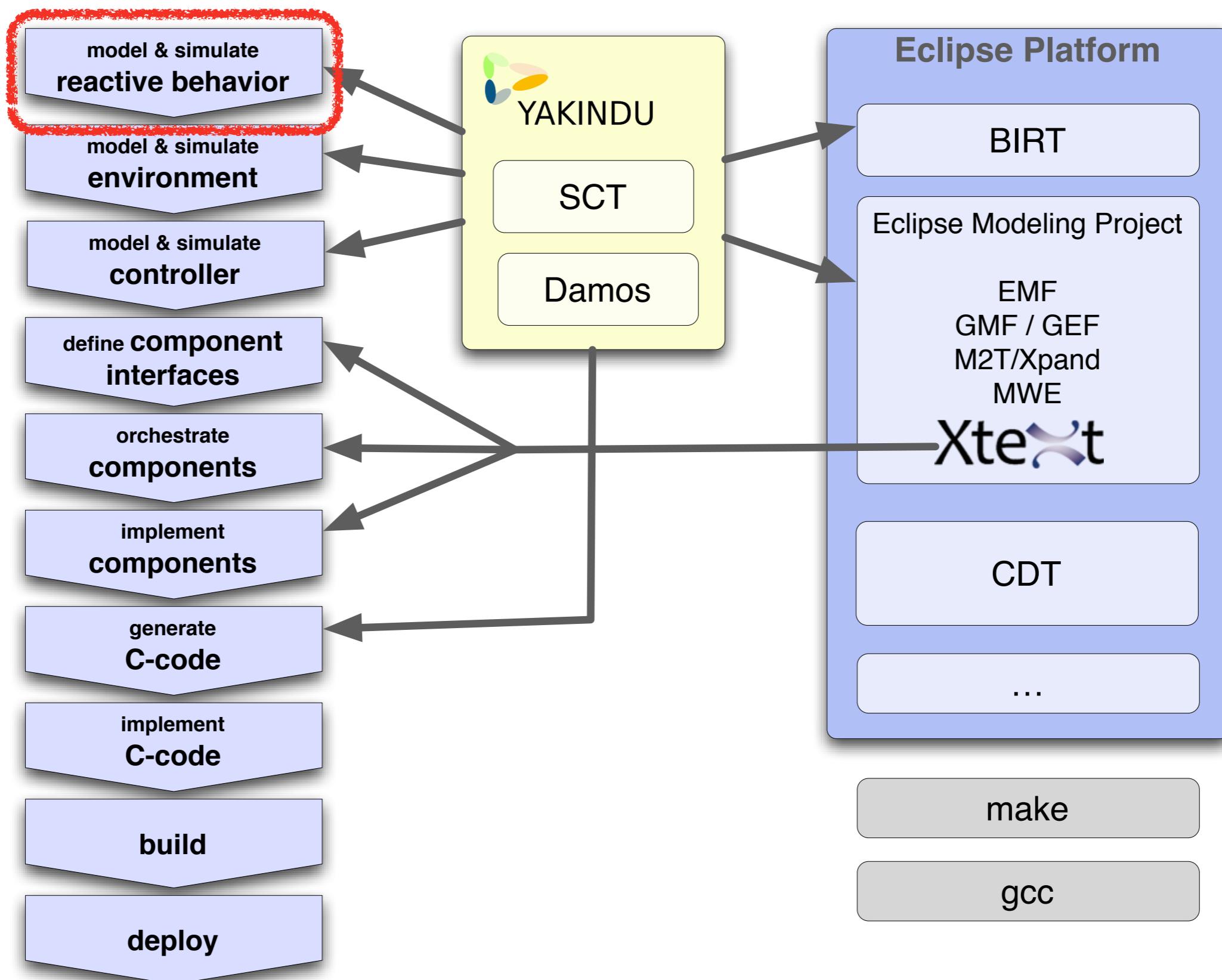
Yakindu is ...



- toolkit for model driven development of embedded systems
- provides set of language modules - currently:
 - SCT - Statechart Tools
 - Damos - Dynamical Systems Modeler
- each language module supports
 - Editor
 - Simulator & Validation
 - Code Generator
- open source
- available at Eclipse Labs (<http://eclipselabs.org/p/yakindu>)

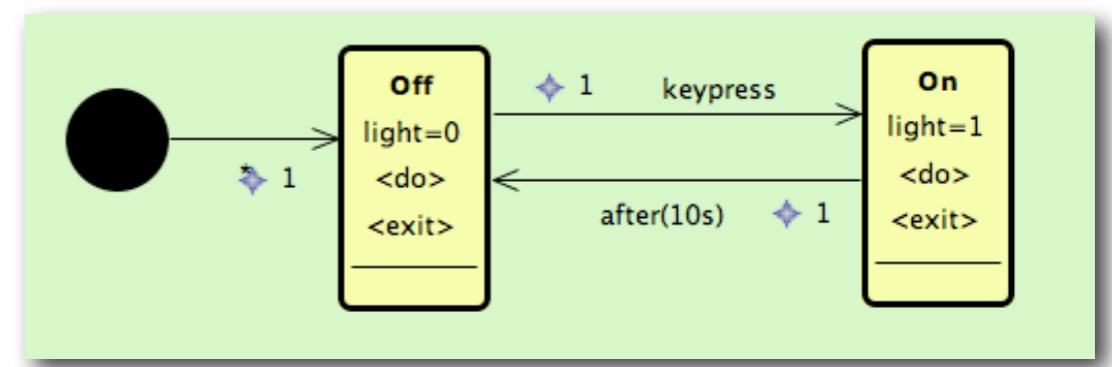
Statecharts

Activities and Tools



State Charts at a glance

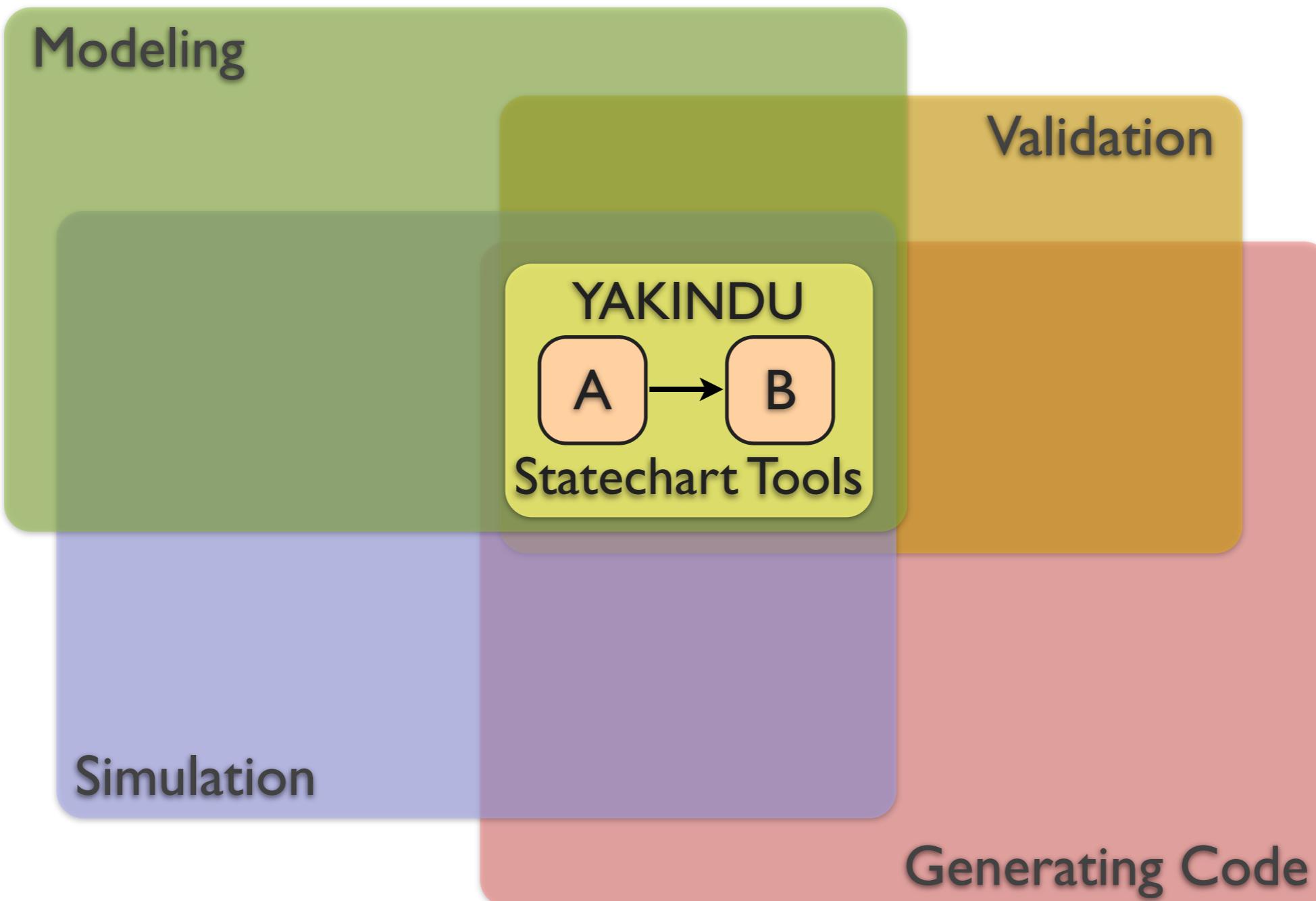
- YAKINDU is based on Statecharts as defined by David Harel (like UML)
- well defined formalism
- extension on deterministic finite state machines
- executable
- describes behaviour based on
 - states
 - transitions
 - actions
- Are used for specifying & implementing event-driven/reactive systems



Finite State Machines

- A system is defined by a finite number of states
 - The behaviour of the system depends on the current state
 - behaves differently to events depending on the state
 - The current state is determined by the history of the state machine
-
- Mealy machine (1955)
 - Actions are associated with transitions
 - Moore machine (1956)
 - Actions are associated with states (entries)

What are YAKINDU Statechart Tools ?



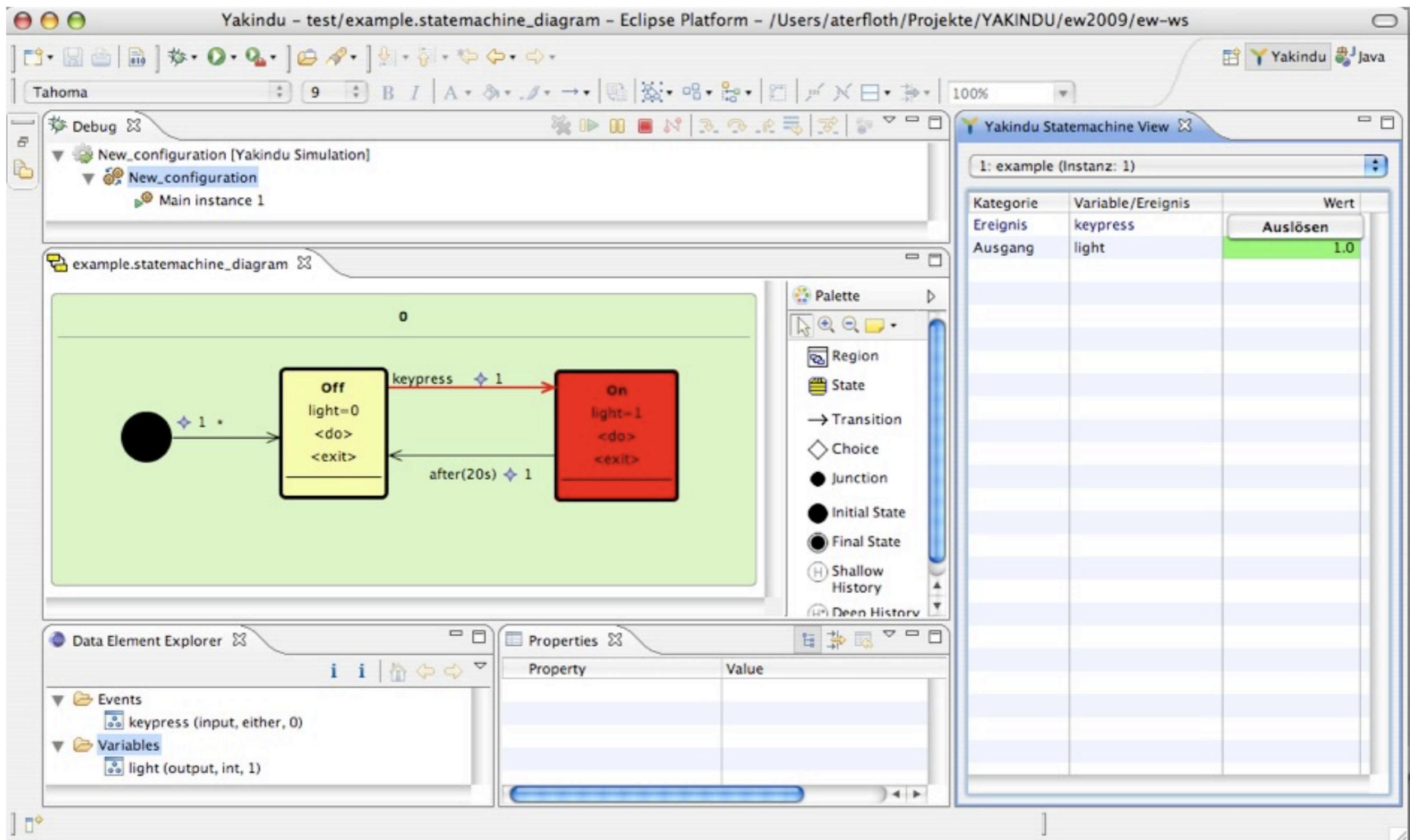
Technology

- completely based on Eclipse technologies
- is Open-Source (EPL)

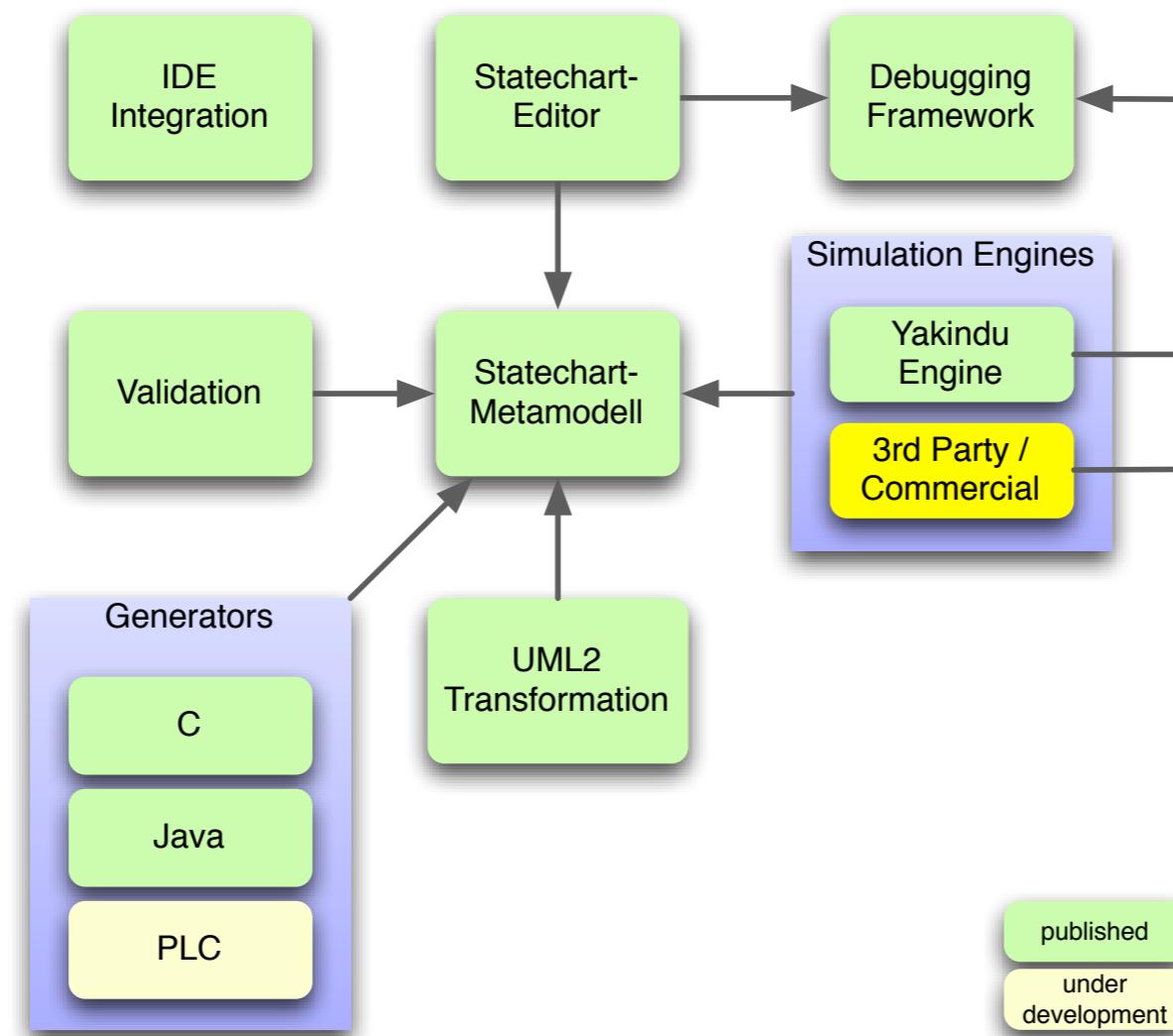


- GMF/GEF
- M2T
 - Xpand
 - Xtend
 - Check
- TMF / Xtext
- further Eclipse stuff ...

YAKINDU Statechart Tools



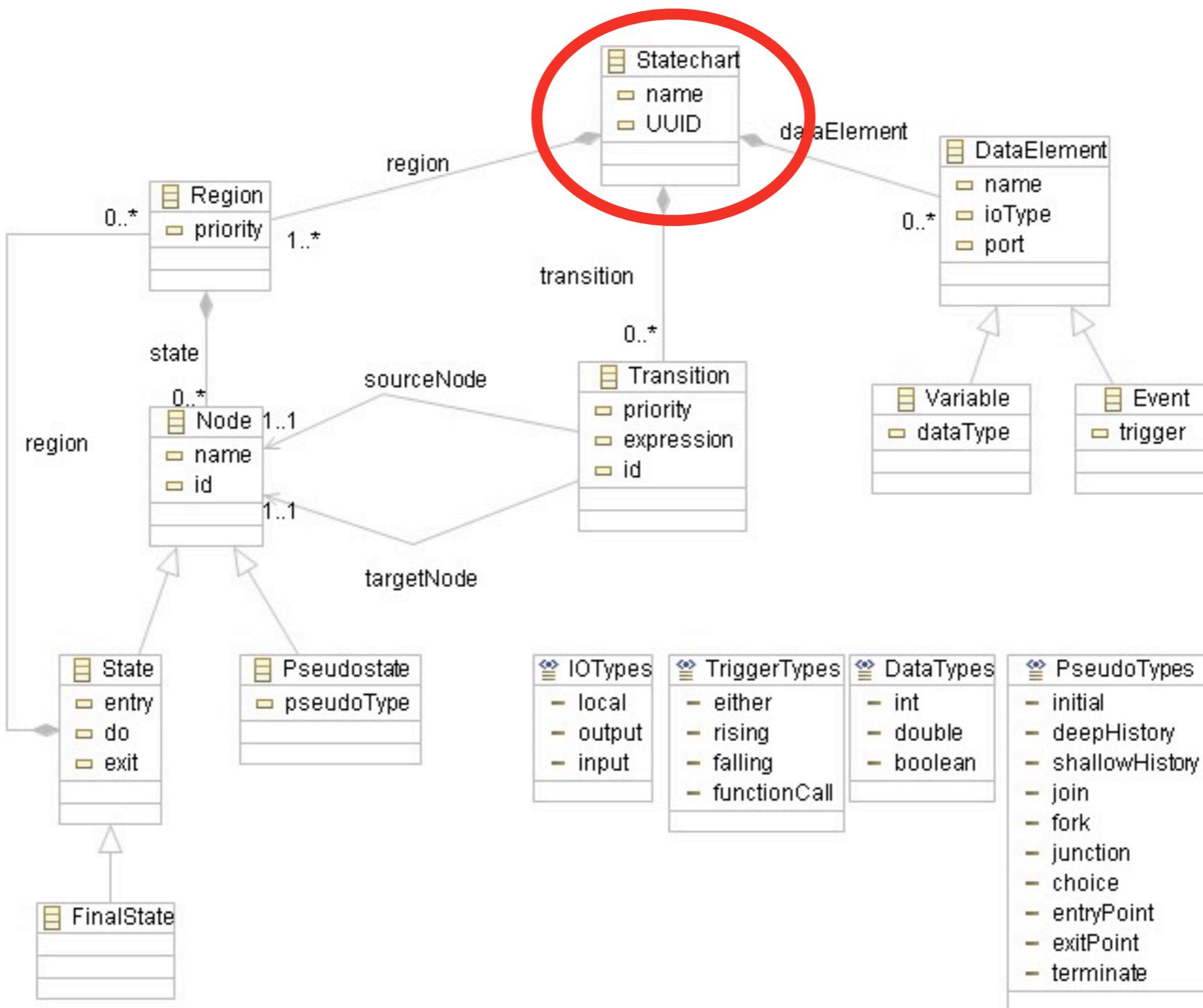
Tools Architecture



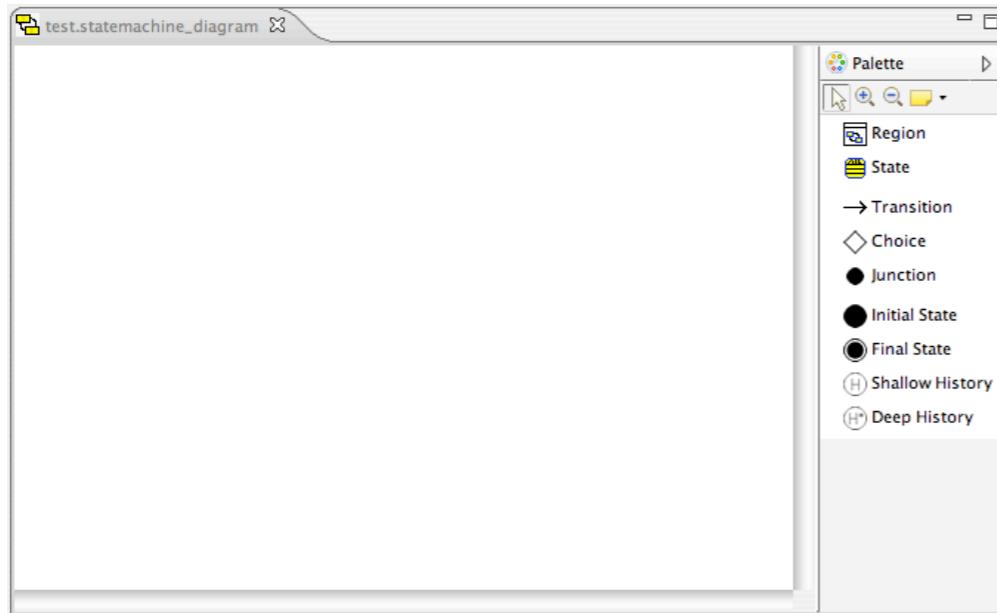
Statechart Semantics

- using an own, simple meta model
- close to UML state machines
- but:
 - YSCs are self contained with an interface well defined by events and variables
 - core execution semantics are cycle-driven and not event-driven
 - allows processing concurrent events
 - event driven behaviour can be defined on top
 - time is an abstract concept for statecharts
 - time control is delegated to the environment
- model interpreter and different flavours of generated code follow the same core semantics

Statechart

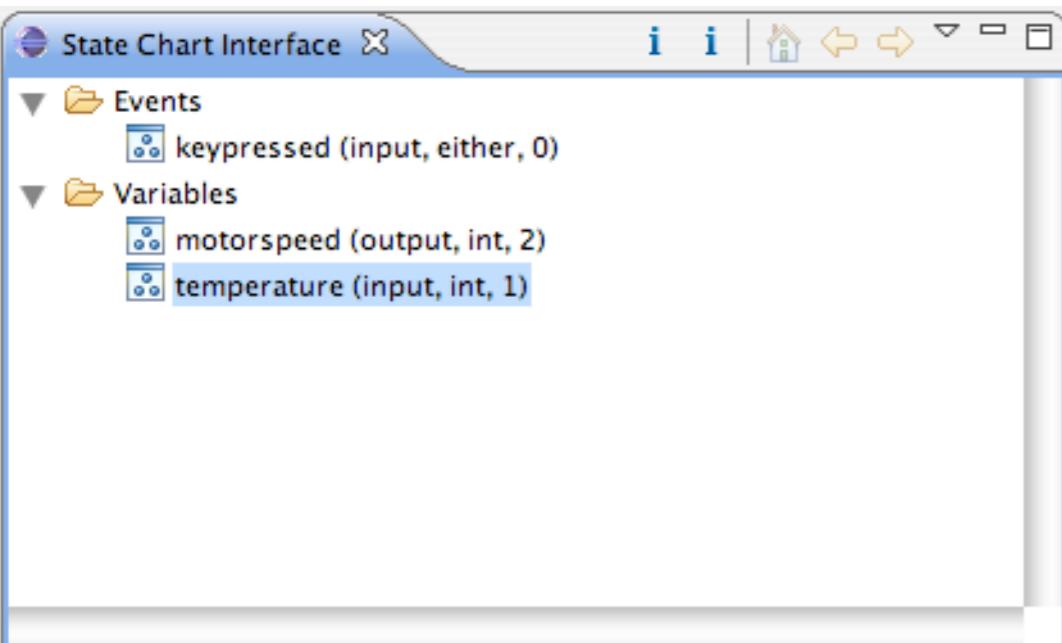


Statechart



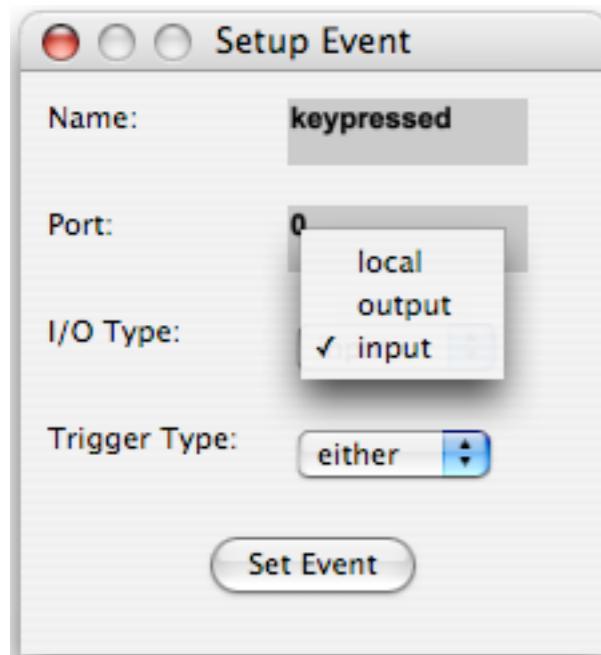
- root model element
- contains everything that belongs to a statechart definition
 - events and variable definitions (interface)
 - one or more regions (implementation)
- represented by the drawing canvas

Statechart Interface



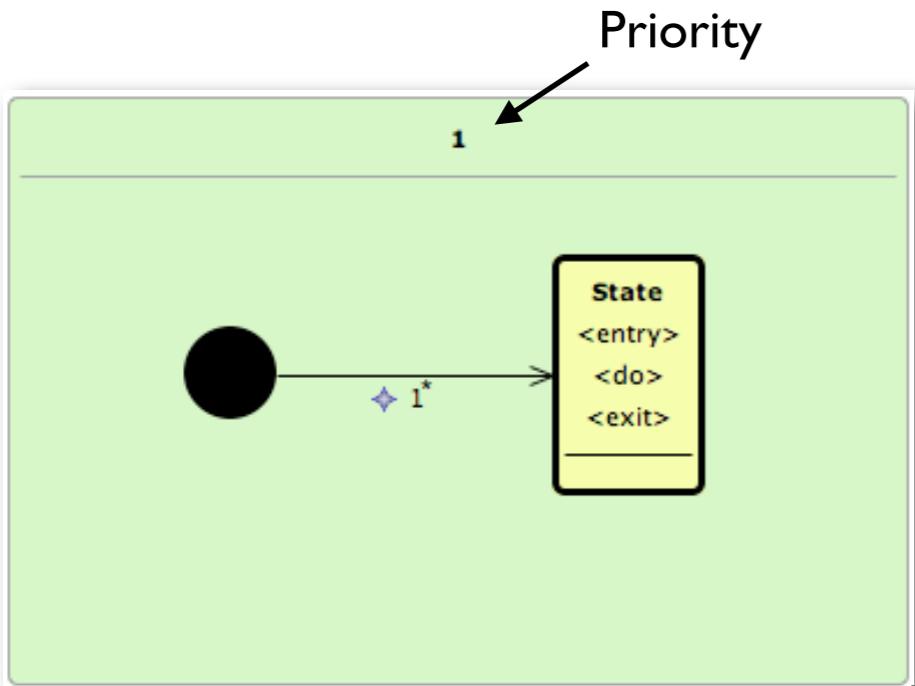
- A statechart interacts with its environment through its interface
- the statechart interface is defined by
 - events
 - variables
 - procedures
- Defines input- and output-interfaces
- Accessible through the ,State Chart

Events



- state charts can
 - consume events (input)
 - produce events (output)
- events trigger behaviour
- event types are:
 - signal
 - time
- YAKINDU events have no values

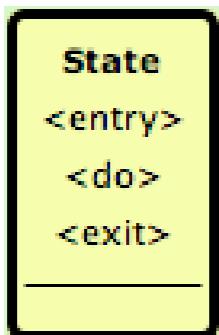
Region



- regions are container for states
 - regions are structural elements
 - statecharts may contain several regions
 - regions have priorities
-
- only one state can be active within a region
 - no behaviour is directly associated with regions

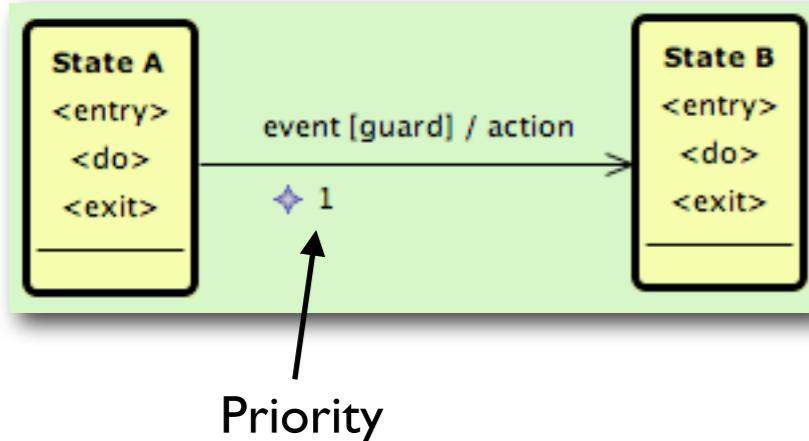
State

„A state models a situation during which some invariant condition holds“ (UML 2 Specification)



- defines the behaviour of a system in these 'situations'
 - ▶ behavioural equivalence classes
- states process events through outgoing transitions
- execute actions
 - on state entry
 - on state exit
 - continuously (do)
- states always 'live' within a region

Transition

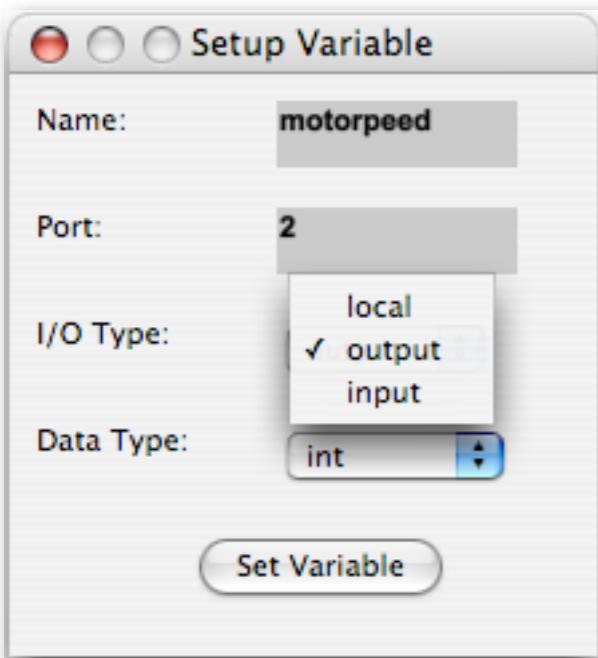


execution sequence:

1. 'State A' exit action
2. transition action
3. 'State B' entry action

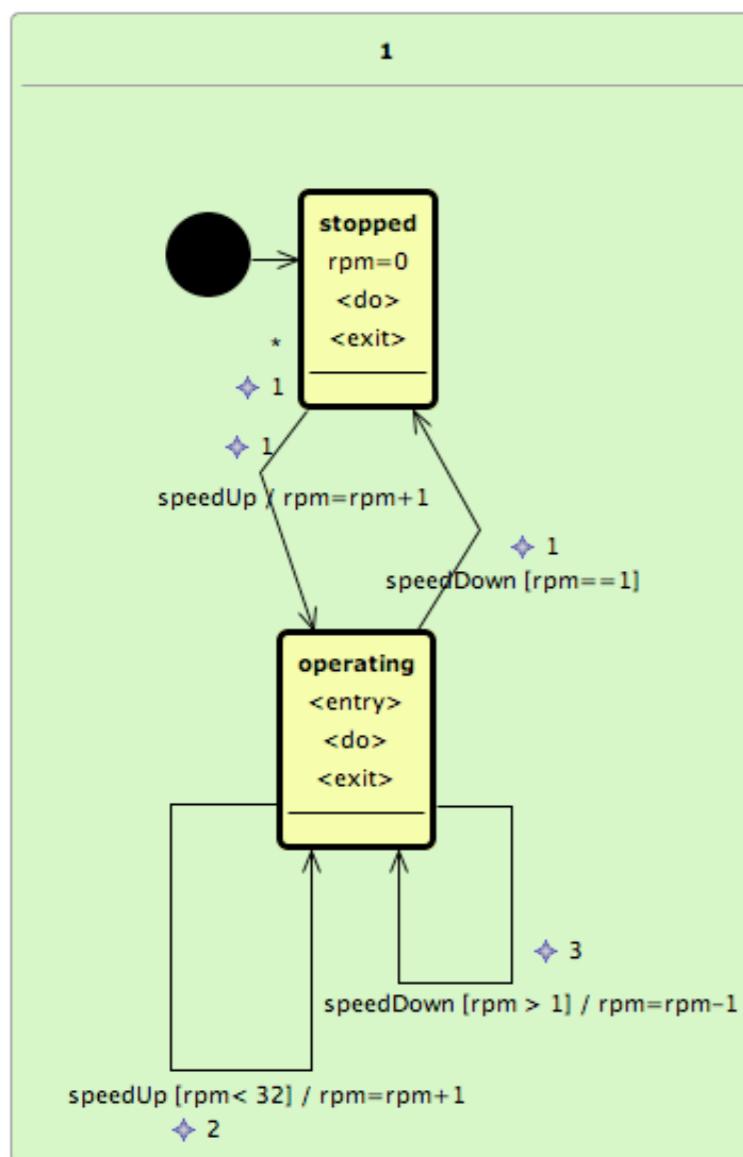
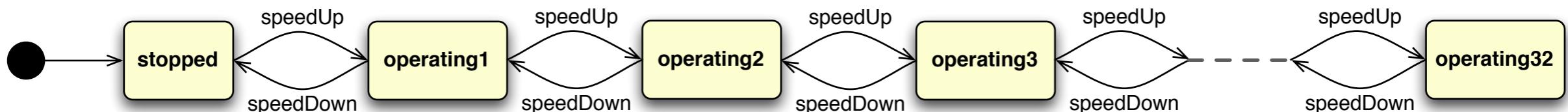
- switching from one state to another is called 'state transition'
- transitions are triggered by an event
- the transition is taken if the guard expression is true
- if the transition is taken then the action is taken
- the transition priority defines the evaluation order of competing transitions (same triggers)

Variables



- used
 - for input & output values, parameter etc.
 - as extended state variables
- hold quantitative aspects while states represent qualitative aspects
- can be input, output or local
- can be evaluated and manipulated within guards and actions

Use of Extended State Variables



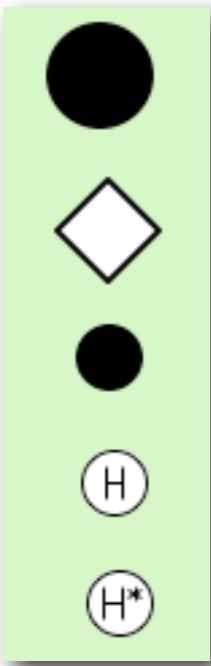
- 33 states without extended state variable
- 2 states with variable ,rpm'
- what happens if we need
 - 256 speed values
 - an ,emergency stop' event

Final State



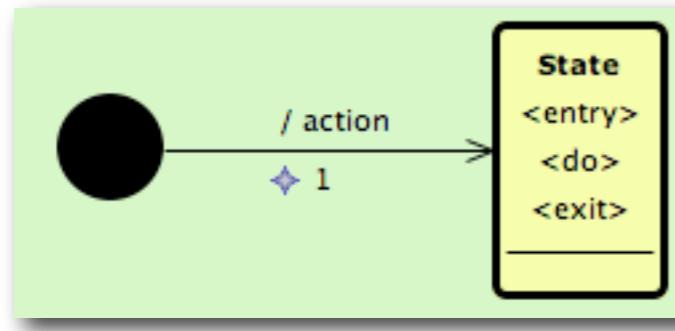
- is a special (real) state
 - ▶ the system can stay in a final state
- indicates 'completion' of enclosing region
- nothing happens anymore
- must not have any outgoing transitions
- has no associated behaviour

Pseudostates



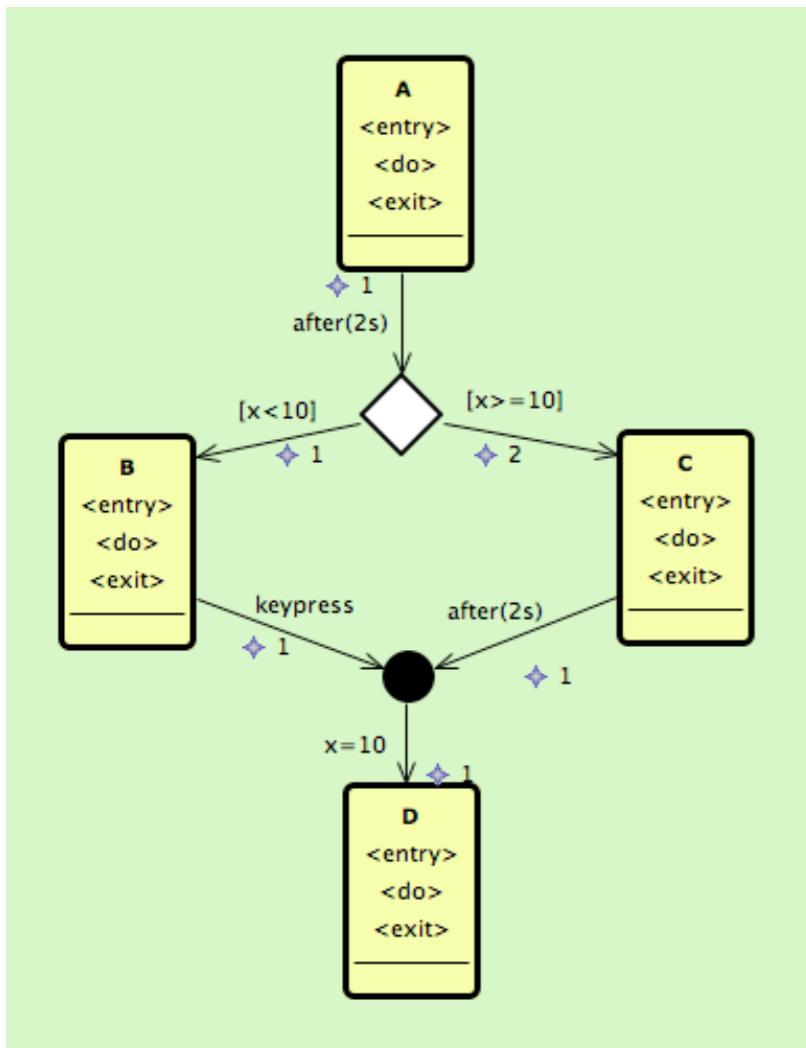
- no real states
 - ▶ the system can NOT stay in pseudostates
- used to increase expressiveness
- supported pseudostates:
 - initial state
 - choice
 - junction
 - shallow history

Initial State

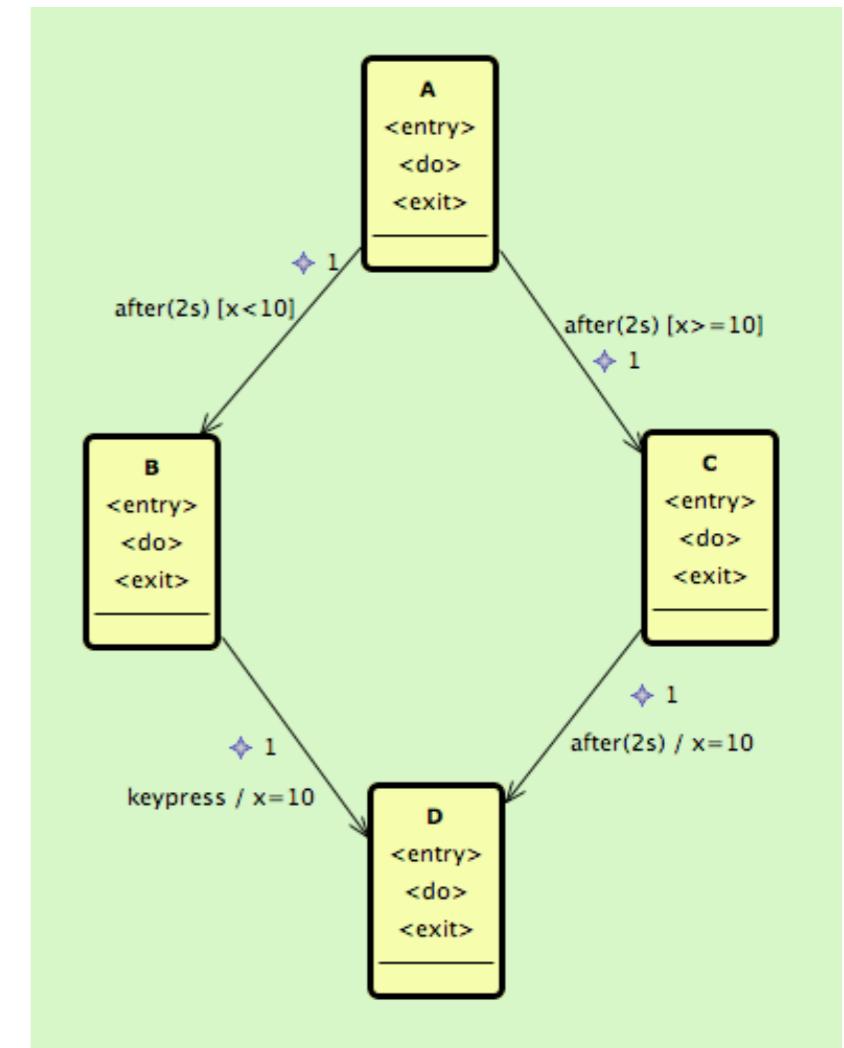


- marks up the initial **real state** of a region
- every region must contain an initial state
- an initial state must have exactly one outgoing transition to a real state
- this transition must not have trigger or guard
- this transition may define an action
- This transition will implicitly

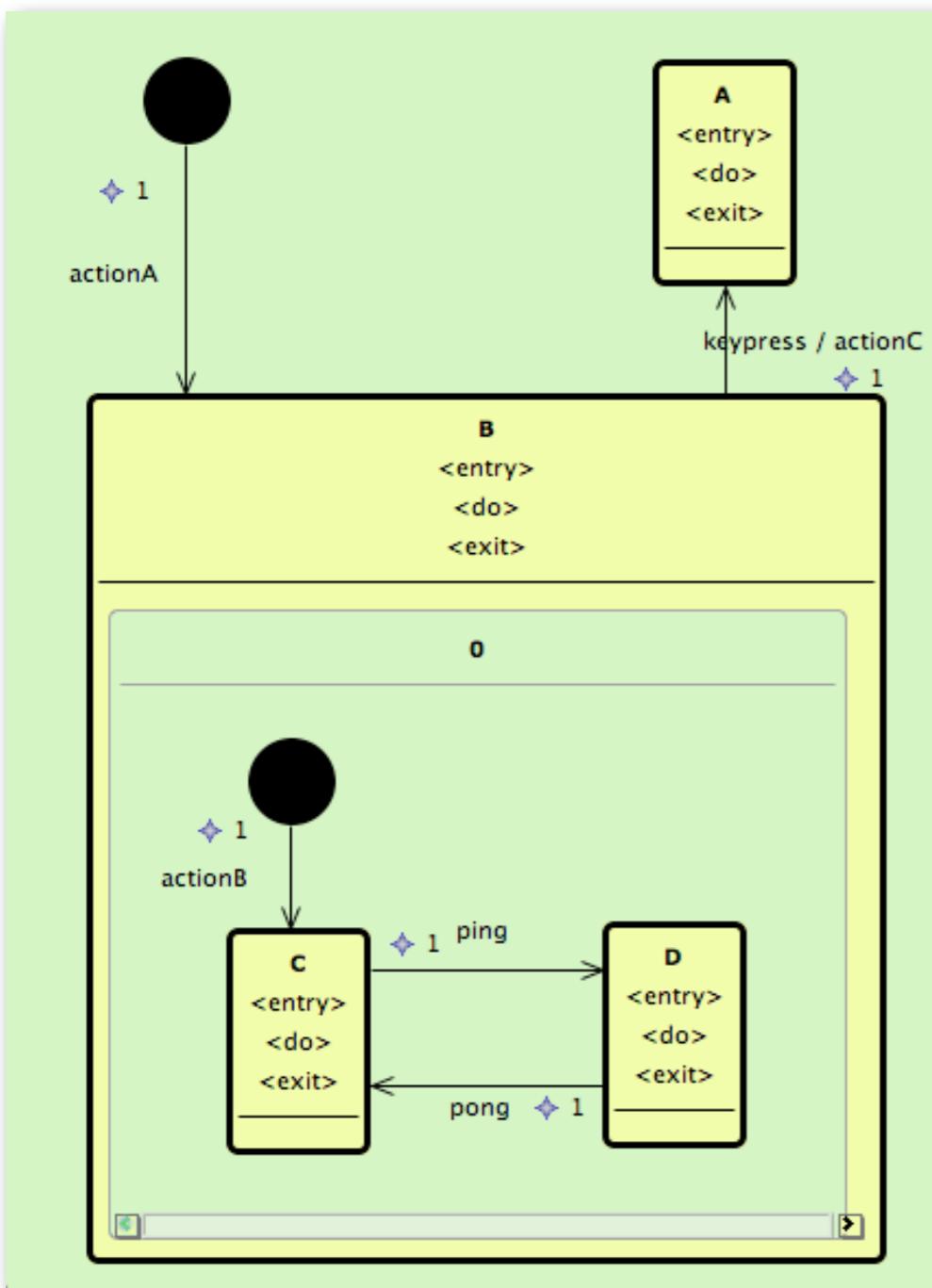
Choice and Junction



- used to structure transitions
- choices evaluates guards on outgoing transitions
- junction are used for 'merges'

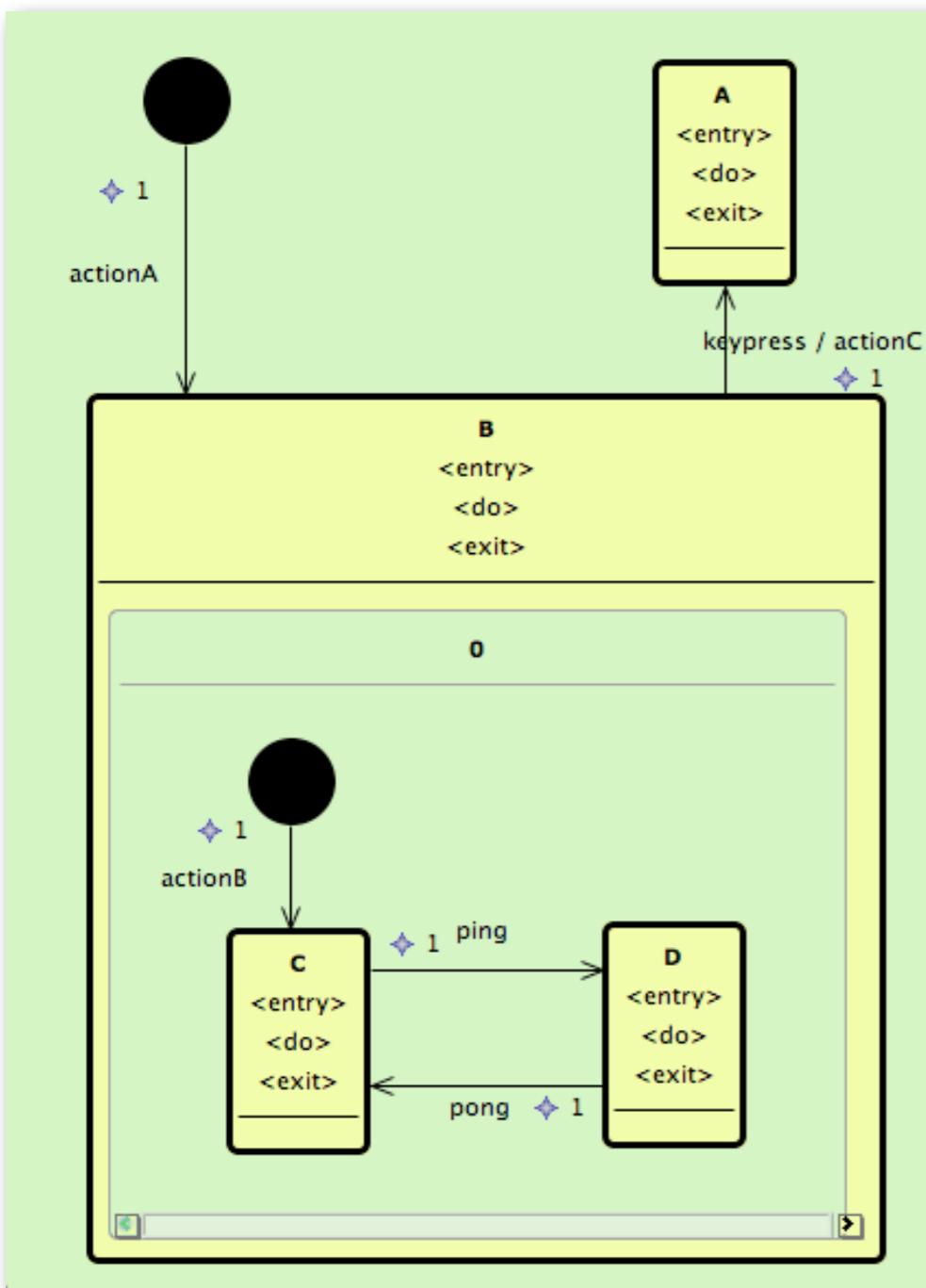


Hierarchical Statecharts



- a state can be refined by substates
- substates are defined within sub regions
- state hierarchies can be used for behavioural abstraction
- reducing number of states and transitions in complex scenarios
- if B is active then either C or D is active
- if C or D is active then also B is active

Hierarchical Statecharts



initial execution sequence

- actionA; B entry; actionB; C entry
→ (B,C) is active state configuration

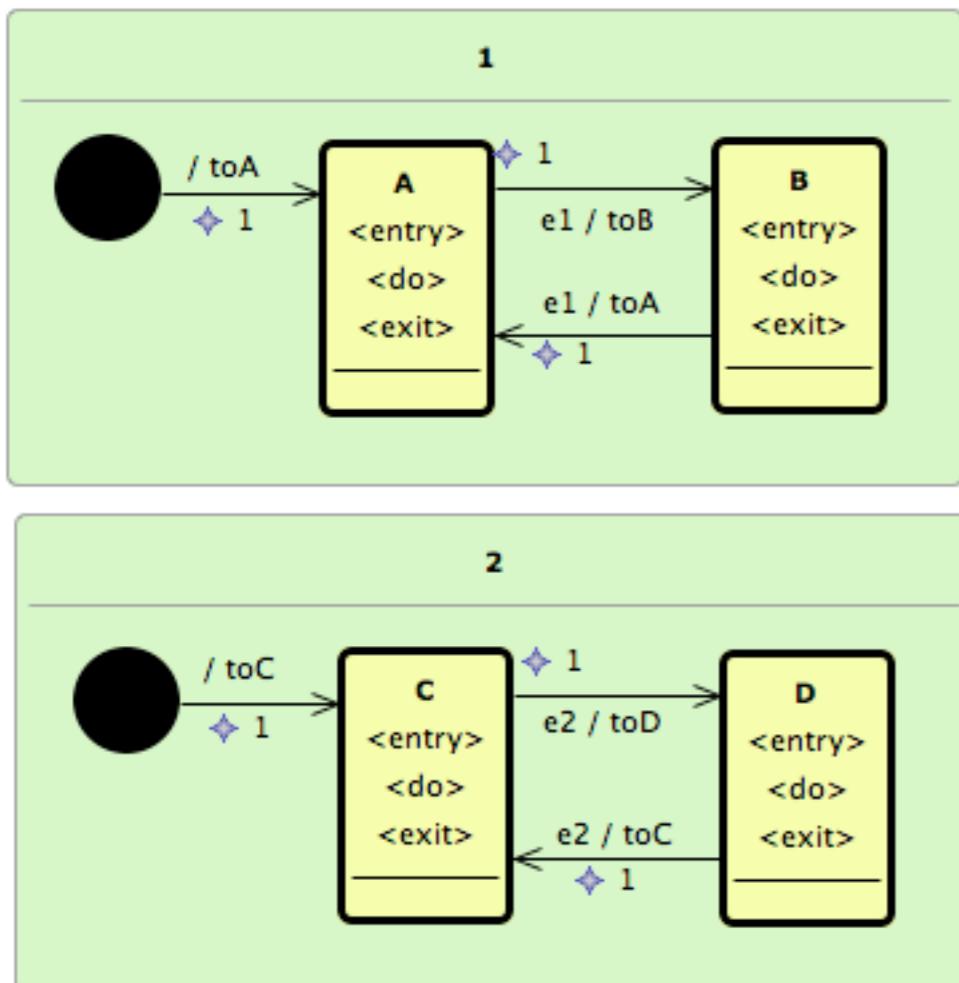
execution sequence on ,ping'

- C exit; D entry
→ (B,D) is active state configuration

execution sequence on ,keypress'

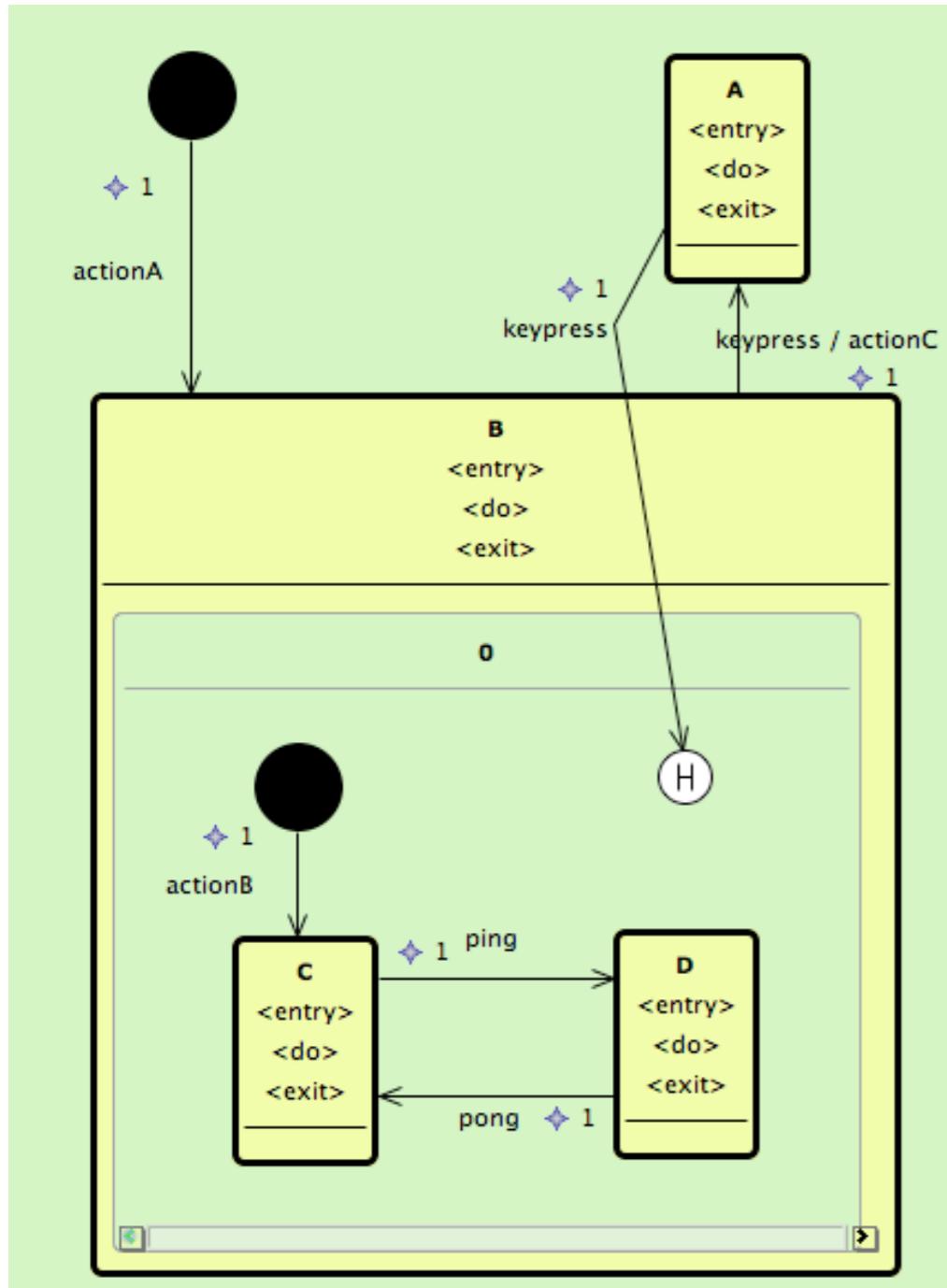
- D exit; B exit; actionC; A entry
→ (A) is active state configuration

Parallel Regions



- statechart and states may contain multiple regions
- each region may contain an active state configuration
- this allows parallel active states
- used to model independent behavioural aspects
- possible state configurations are (A,C), (A,D), (B,C) and (B,D)
- priority of region determines evaluation order

Shallow and Deep History



- shallow history remembers the last active state of a region when the region is left
- deep history remembers the last active state configuration (including substates recursively) of a region when the region is left
- a transition to a history is a transition to stored state (configuration)
- execution sequence is the same as for normal transitions
- histories do not define behaviour

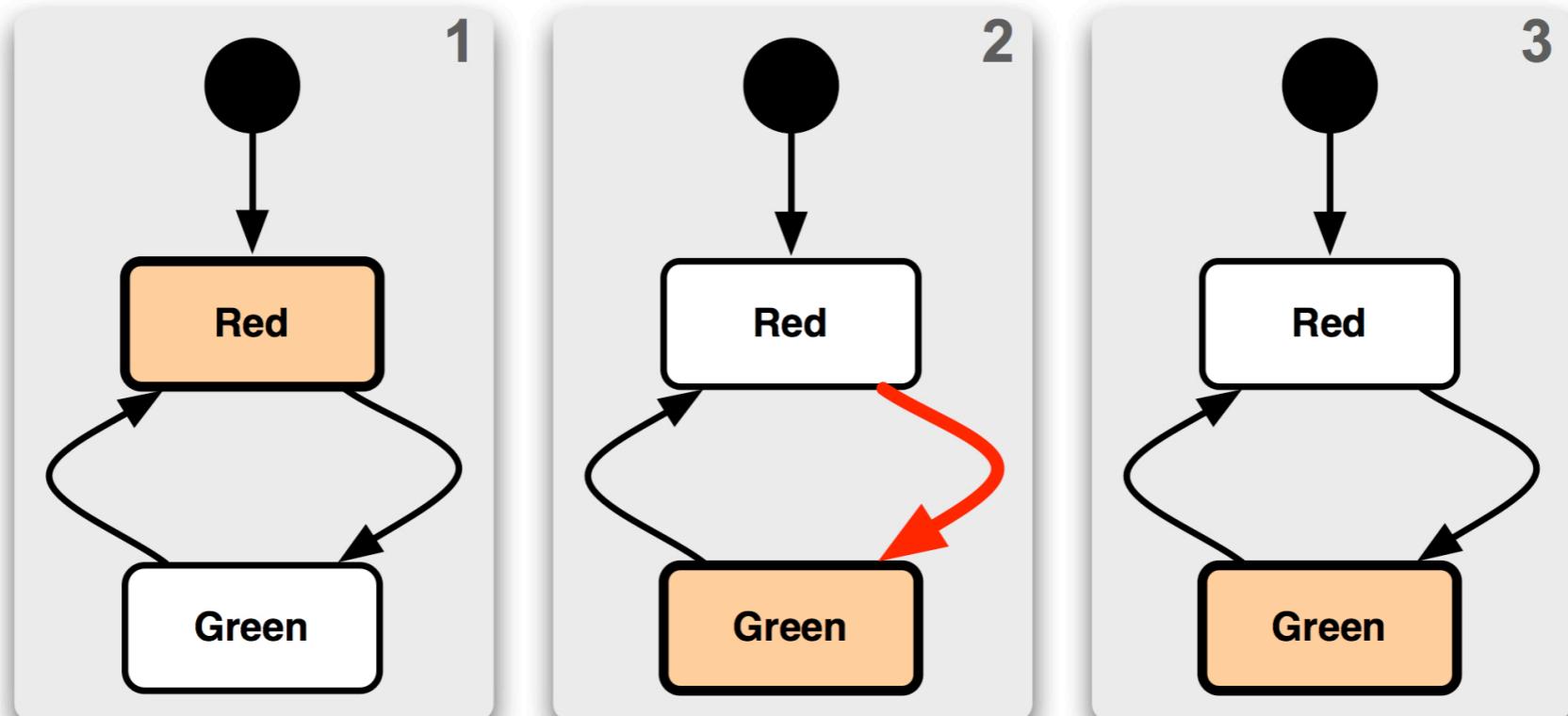
Expressions

- expression language is independent from target language (C, Java etc.)
- is used for trigger, guard and action expressions
- transition: *trigger ([guard])?* (*/ actions*)?
- trigger expressions are a list of signal or time events:
e.g.: *keypress* or *after(100ms)* or *keypress, after(2s)*
- guards are boolean expressions
- actions are statements
 - variable assignment: *x=2*
 - raise event: *raise(stop)*
 - procedure call: *shutdown()*

Expressions

- operators
 - assignment: `=, +=, -=, *=, /=, %=, <<=, >>=, &=, ^=, |=`
 - boolean: `&&, ||, !`
 - compare: `==, !=`
 - arithmetic: `+, -, *, /, %`
 - bit: `&, |, ^, >>, <<`
- literals
 - decimal & hex integer
 - floating point
 - boolean (true, false)
 - string

Simulation



Simulation

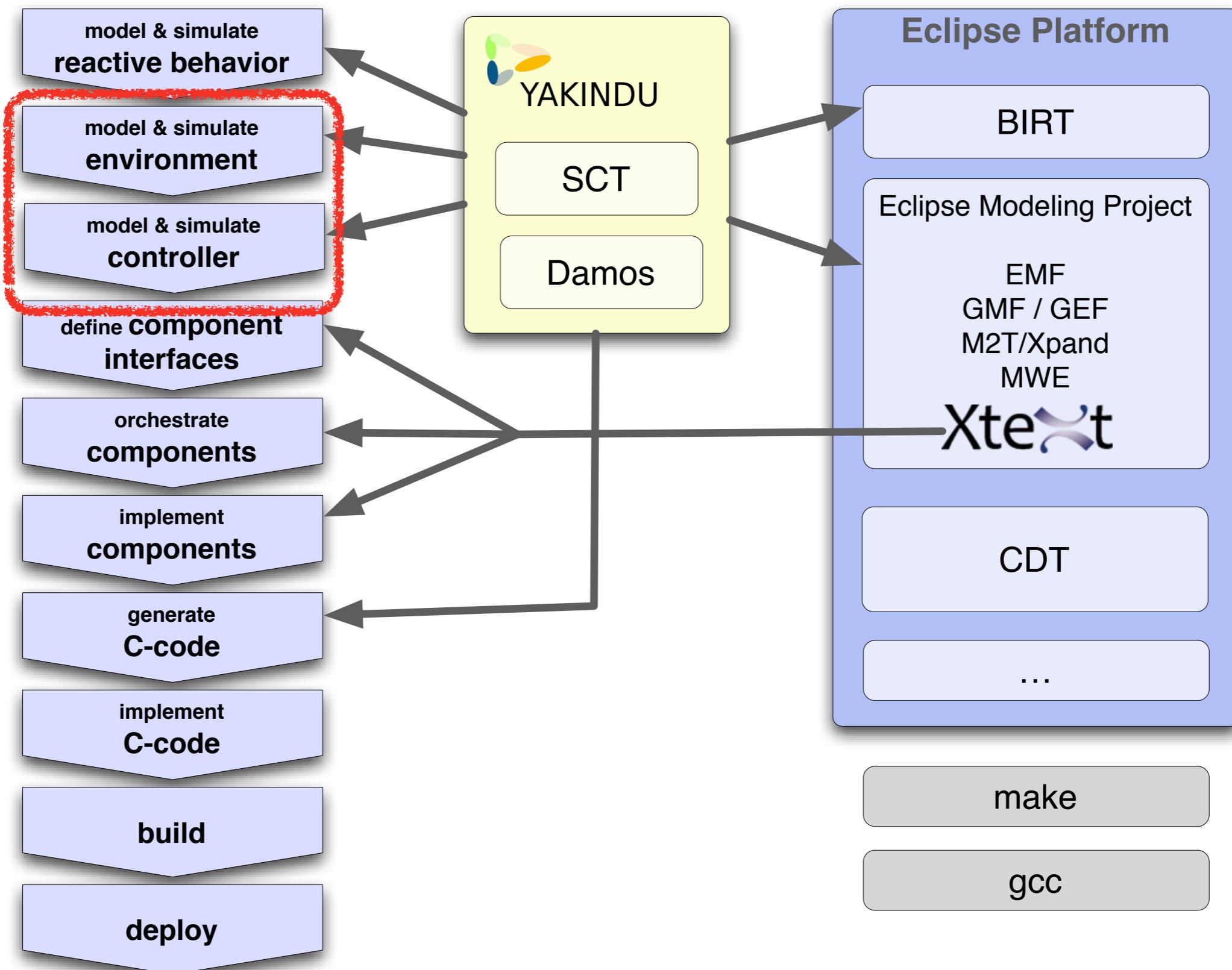
- Appropriate for models that describe behavior
- Tool integration needed
 - Modeling tools
 - Model transformations to simulation models
 - Simulation engines
 - Communication API needed
- Model visualisation
 - May require extension of notation

A close-up photograph of a person's hands typing on a light-colored computer keyboard. The hands are positioned over the keys, with fingers moving rapidly. A bright, semi-transparent white rectangular box containing the word "Demo..." is overlaid on the upper left portion of the image. The background is slightly blurred, showing what appears to be a wooden desk or surface.

Demo...

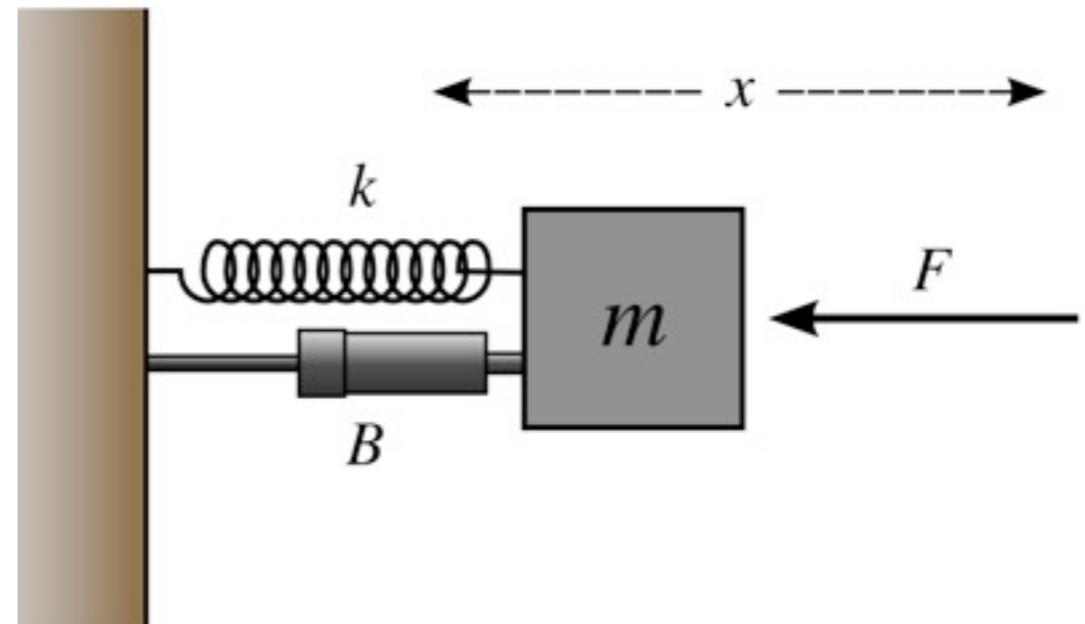
Blockdiagrams

Activities and Tools



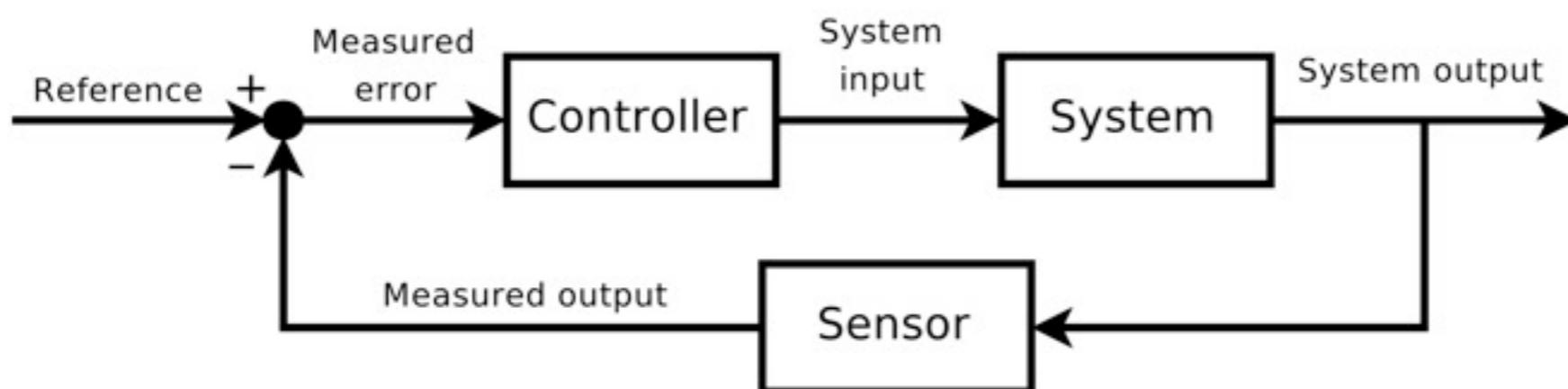
Dynamical Systems

- Mathematical formalization of time-dependent processes
- Examples include:
 - Electronic circuits
 - Mass-spring-damper systems
- Applications include:
 - Control systems
 - Signal processing



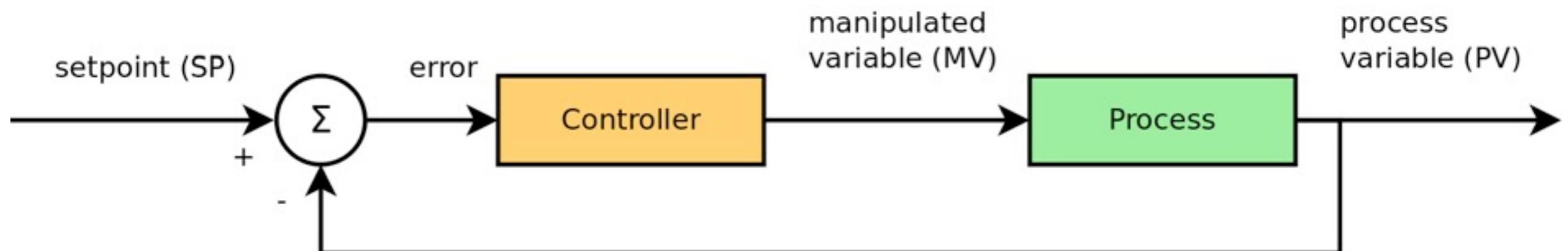
Dynamical Systems Modeling

- Prevalent graphical notation: Block diagrams
- Block: System component's transfer function
- Connection: Signal flow

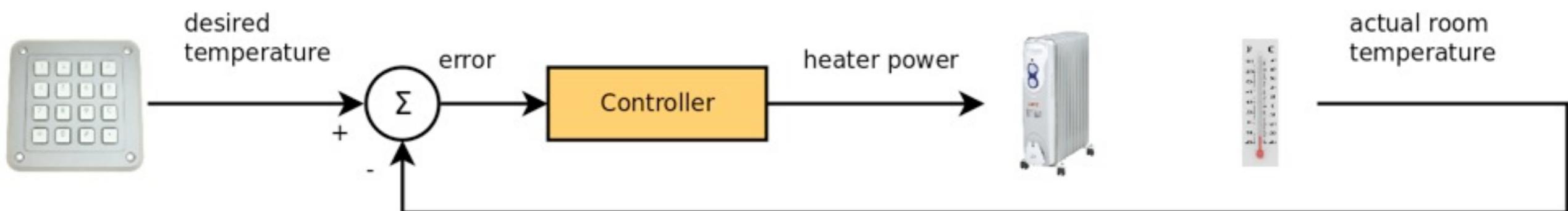


Negative-feedback Control System

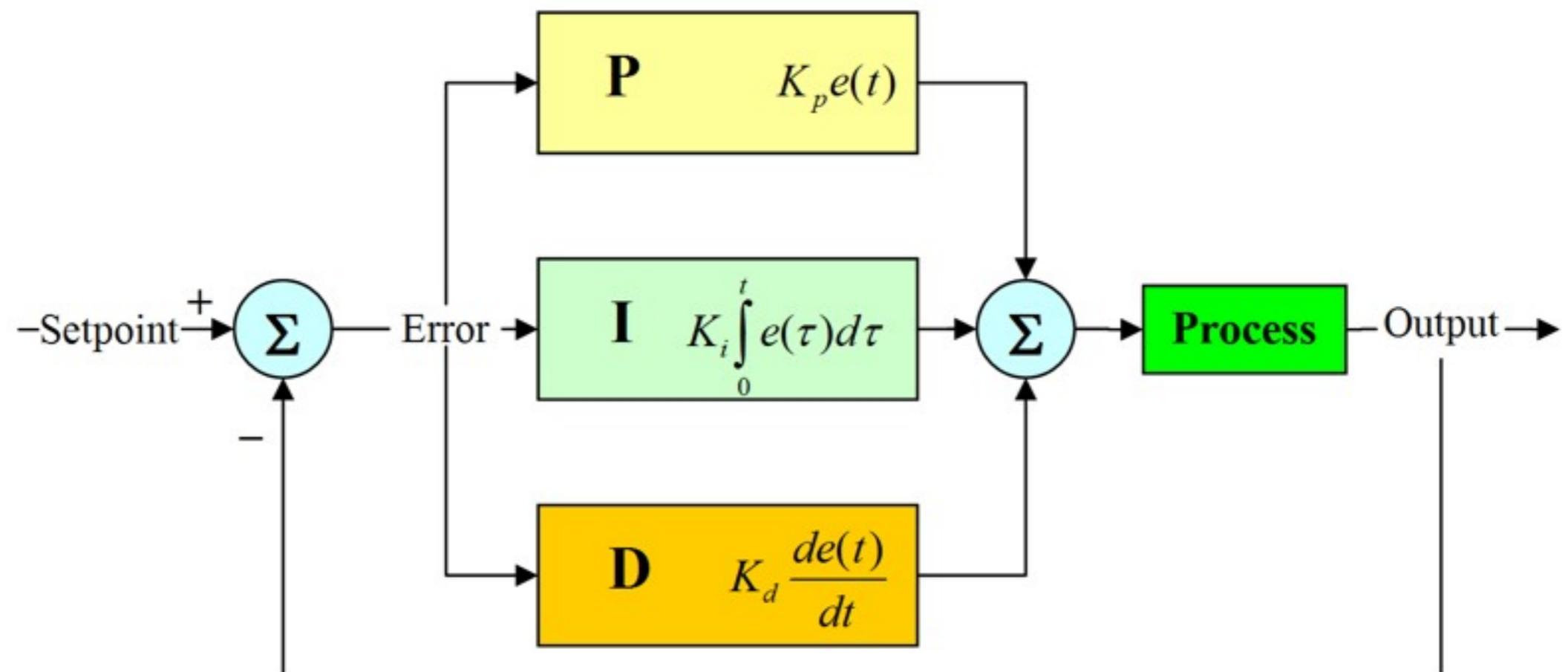
- Controller's goal: $\text{error} = \text{setpoint} - \text{process variable} = 0$



Control System Example: Thermostat



PID Controller



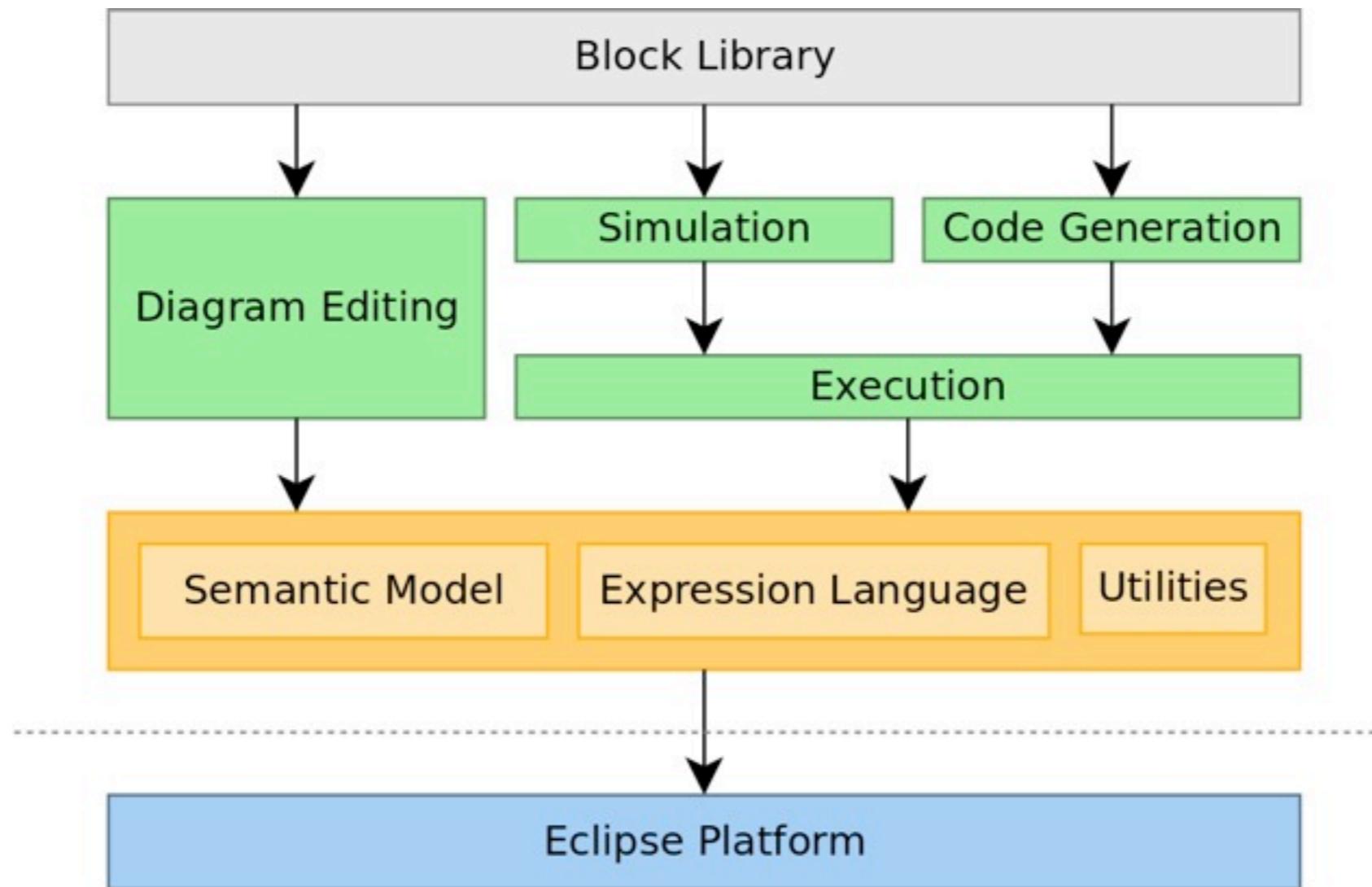
PID Controller Elements

- Proportional Term
 - Higher gain → faster response
 - Excessively large gain → process instability
 - Pure proportional control retain a steady state error
- Integral Term
 - Eliminates steady-state error of proportional-only controllers
 - Can cause overshoot
- Derivative Term
 - Reduces the overshoot produced by the integral term
 - Highly sensitive to noise

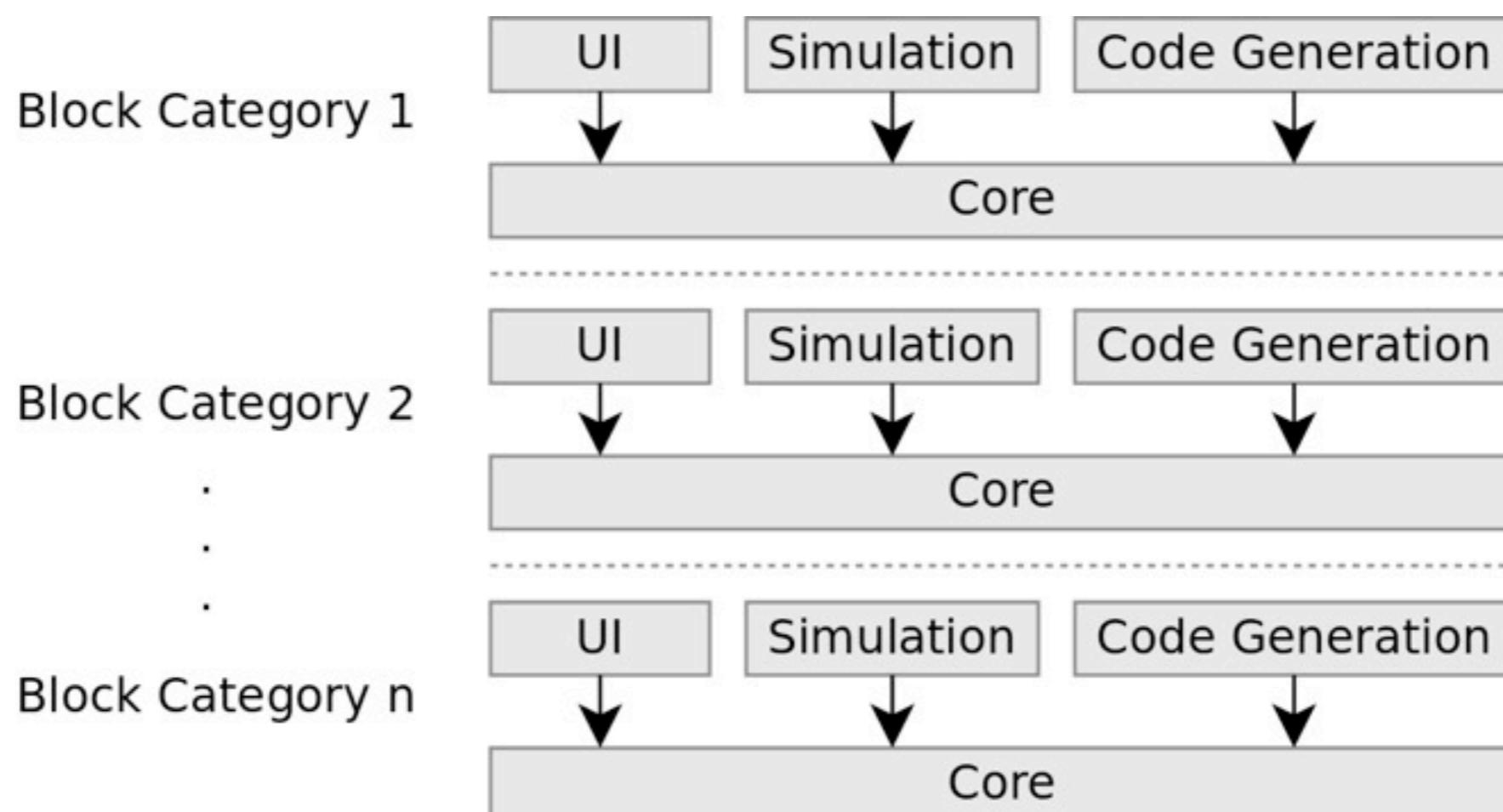
Damos

- Open source Eclipse-based modeling tool
- Deployed as plug-ins
- Consists of
 - Block Diagram Editor
 - Simulator
 - Code generator
 - Standard block library
- Uses standard Eclipse modeling technologies (EMF, GMF, Xpand, etc.)

Architecture

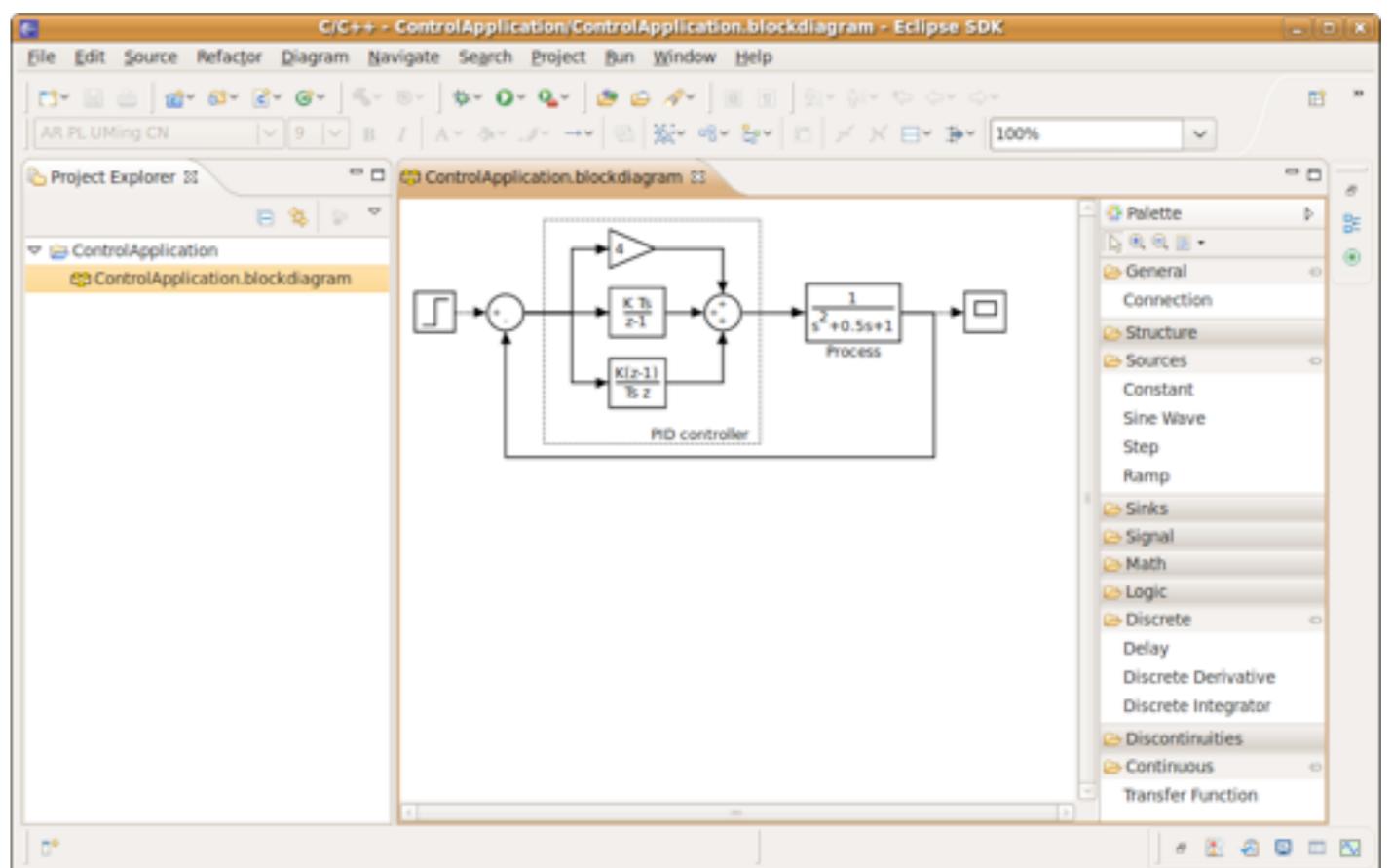


Block Library Architecture



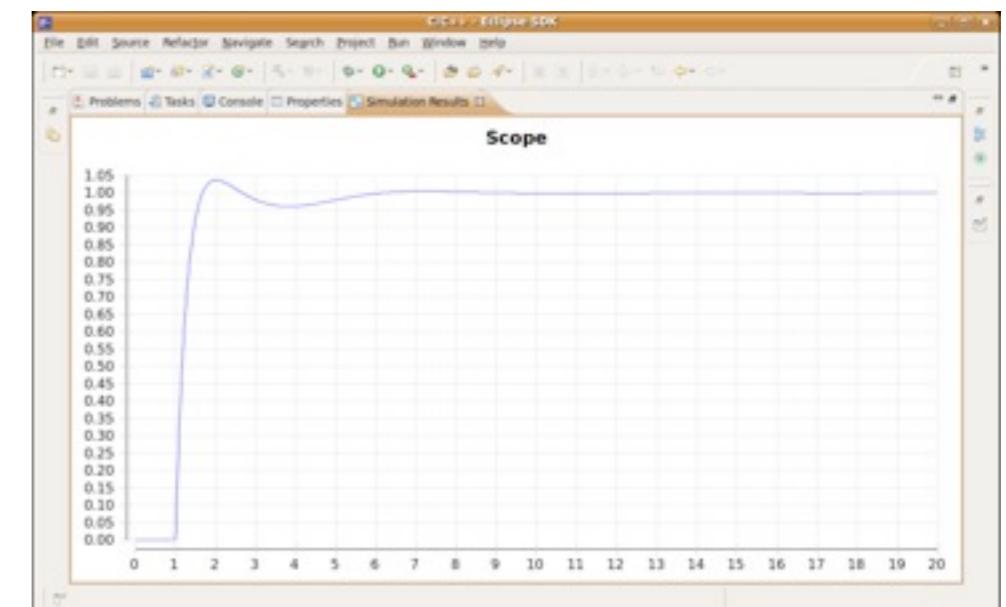
Block Diagram Editor

- Canvas-like drawing area
- Tool palette
 - Selection
 - Zooming
 - Connection tool
 - All available blocks from the block library
- Properties view



Simulator

- Test points are defined by Scope blocks
- Selectable sample rate and simulation time
- Simulation algorithms match algorithms generated by the code generator
- Fixed-point overflow detection



Code Generator

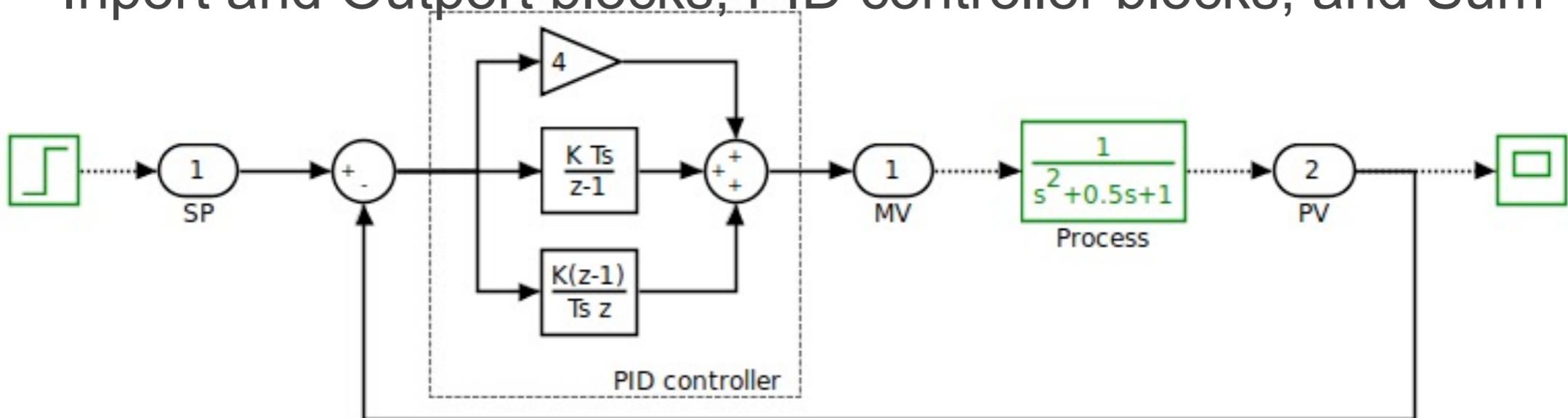
- C99 (ANSI C when providing stdint.h) compliant C source code
- Generated execute function invokes the individual block functions in the order defined by the signal flow
- Imports and Outport are mapped to the input and output parameters of the execute function
- Individual source code files are generated for each block
- Block functions can be implemented as macros

System Components

- Environment components:
 - Solely used for system simulation
 - No code is generated
- Device components:
 - Embedded in the environment
 - Enforce desired behavior on the environment
 - Code is generated

Example Model

- Environment (green):
 - Step function block, Process, and Scope
- Device:
 - Import and Outport blocks, PID controller blocks, and Sum

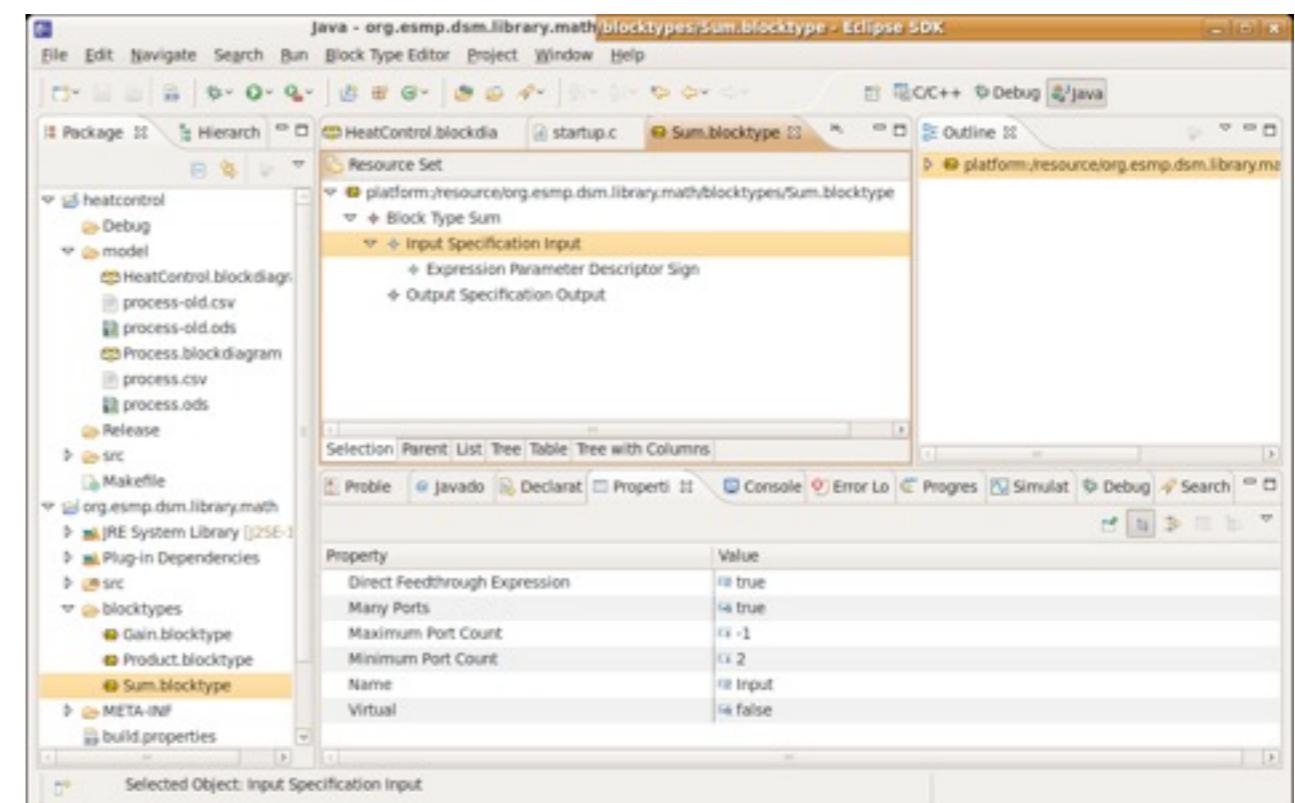


Extending Damos

- Using standard Eclipse extension mechanism
- Four plug-ins must be extended or created:
 - Mandatory core plug-in containing block type model
 - UI plug-in containing UI extensions (e.g. block symbol, properties views)
 - Simulation plug-in containing simulation model
 - Code generation plug-in containing code templates
- Block type editor for the block type model available

Block Library Core Plug-ins

- Block Type definition
 - Inputs and Outputs
 - Block Parameters
- URI mapping extension definition



Block Library UI Plug-ins

- Optional block symbol (i.e. notation) definition
 - Custom figures (view)
 - Custom edit parts (controller)
 - Custom notation model elements
- UI classes are registered using extension mechanism
- Palette entry definition

Execution Engine

- Building execution graph based on
 - signal flow
 - direct-feedthrough inputs
- Computation of block data types
- Used by simulation engine and code generator

Block Library Simulation Plug-ins

- Extending AbstractSimulationModel class, implementing methods
 - initialize()
 - consumeInputValue()
 - computeOutputValues()
 - getOutputValue()
 - update()
- Registering simulation model class using extension mechanism

Block Library Code Generation Plug-ins

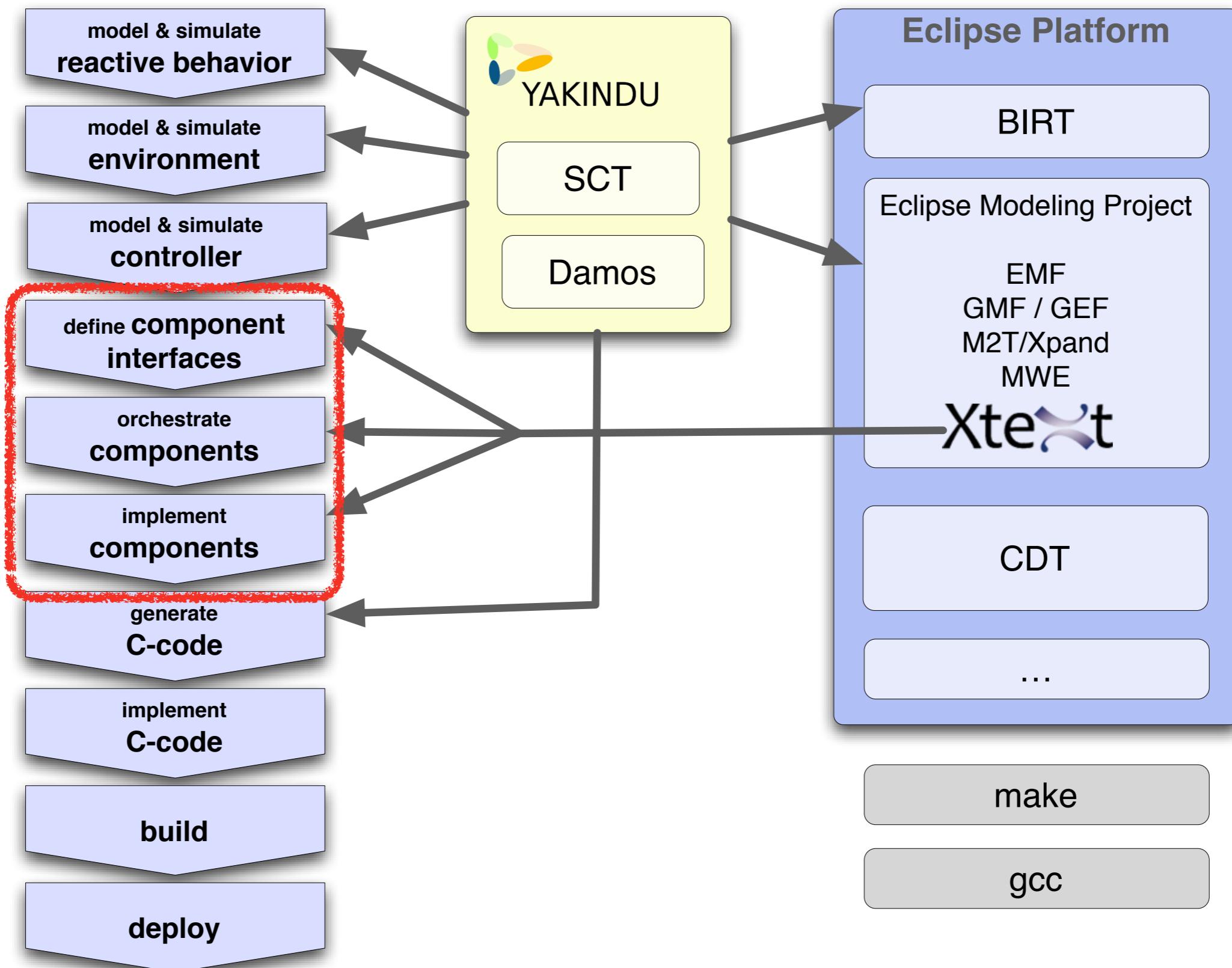
- Individual C code generators can be registered for each block
- Default code generator expands registered Xpand templates
- Block functions are generated:
 - `* __initialize()`
 - `* __consumeInputValue()`
 - `* __computeOutputValues()`
 - `* __getOutputValue()`
 - `* __update()`
- Block functions can be implemented as macros

A close-up photograph of a person's hands typing on a light-colored computer keyboard. The hands are positioned over the keys, with fingers moving rapidly. The background is slightly blurred, showing what appears to be a wooden desk or surface. Overlaid on the upper left portion of the image is the word "Demo..." in a large, white, sans-serif font.

Demo...

Component Model

Activities and Tools



Component Model

- Software architecture is based on components.
- component model captures the essential abstractions
- component model defines
 - component interfaces
 - instances
 - connections
 - mapping to implementations
- code generator enforces architecture conformance

Component Model

- model type is a textual DSL
- individually build
- integrated with statecharts and block diagrams
- implemented using Xtext

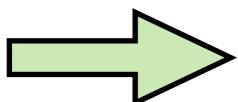
Domain Specific Languages

Domain Specific Languages(DSLs)

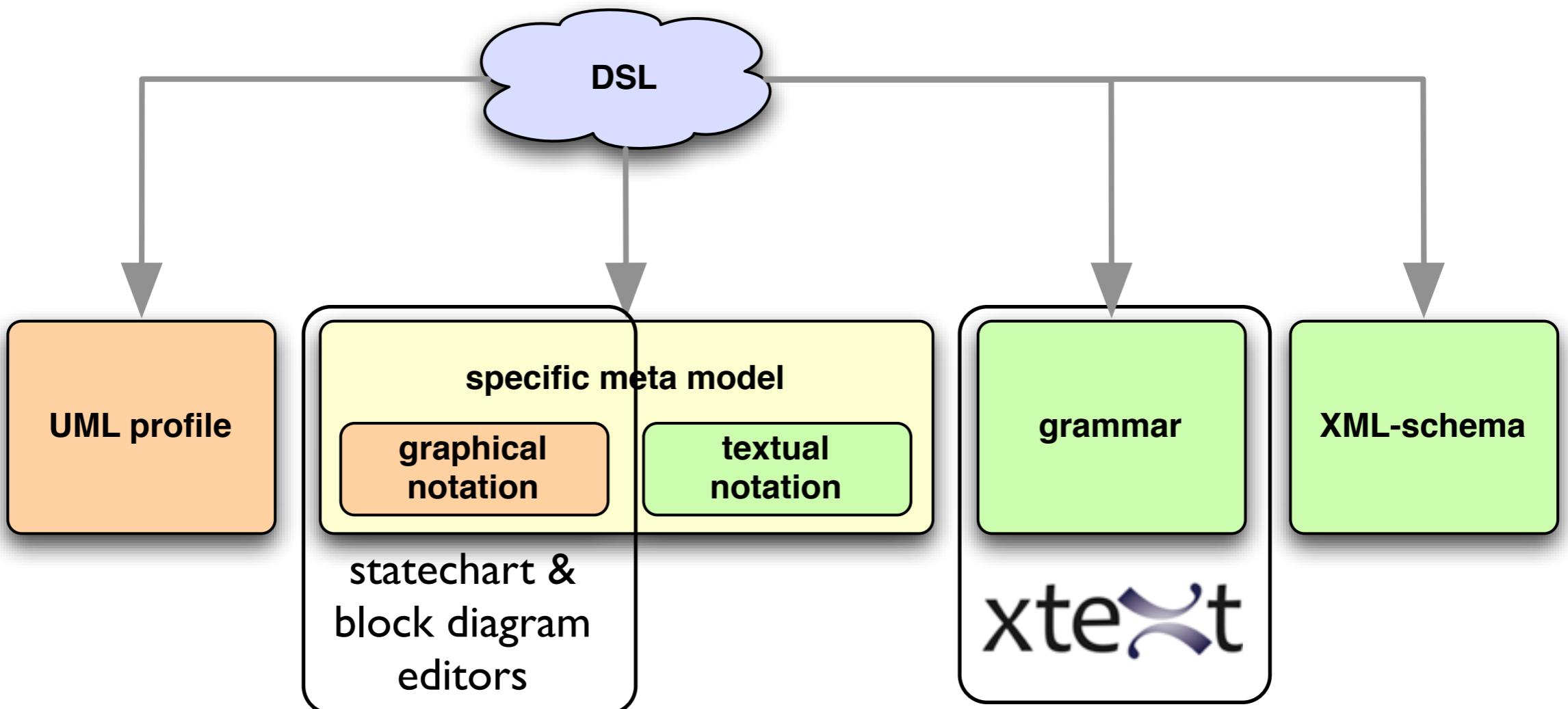
„A domain-specific programming language (domain-specific language, DSL) is a programming language designed to be useful for a specific set of tasks. This is in contrast to general-purpose programming language (general-purpose language, GPL), such as C or Java, or general-purpose modeling languages like UML.“

Quelle: Wikipedia

- Why DSLs?
 - Focus on the relevant aspects of a language
 - Apply the notion of the domain ⇒ more comprehensive for domain experts
 - Optimization by specialization
 - Use of modeling concepts that GPLs (z.B. UML) do not support

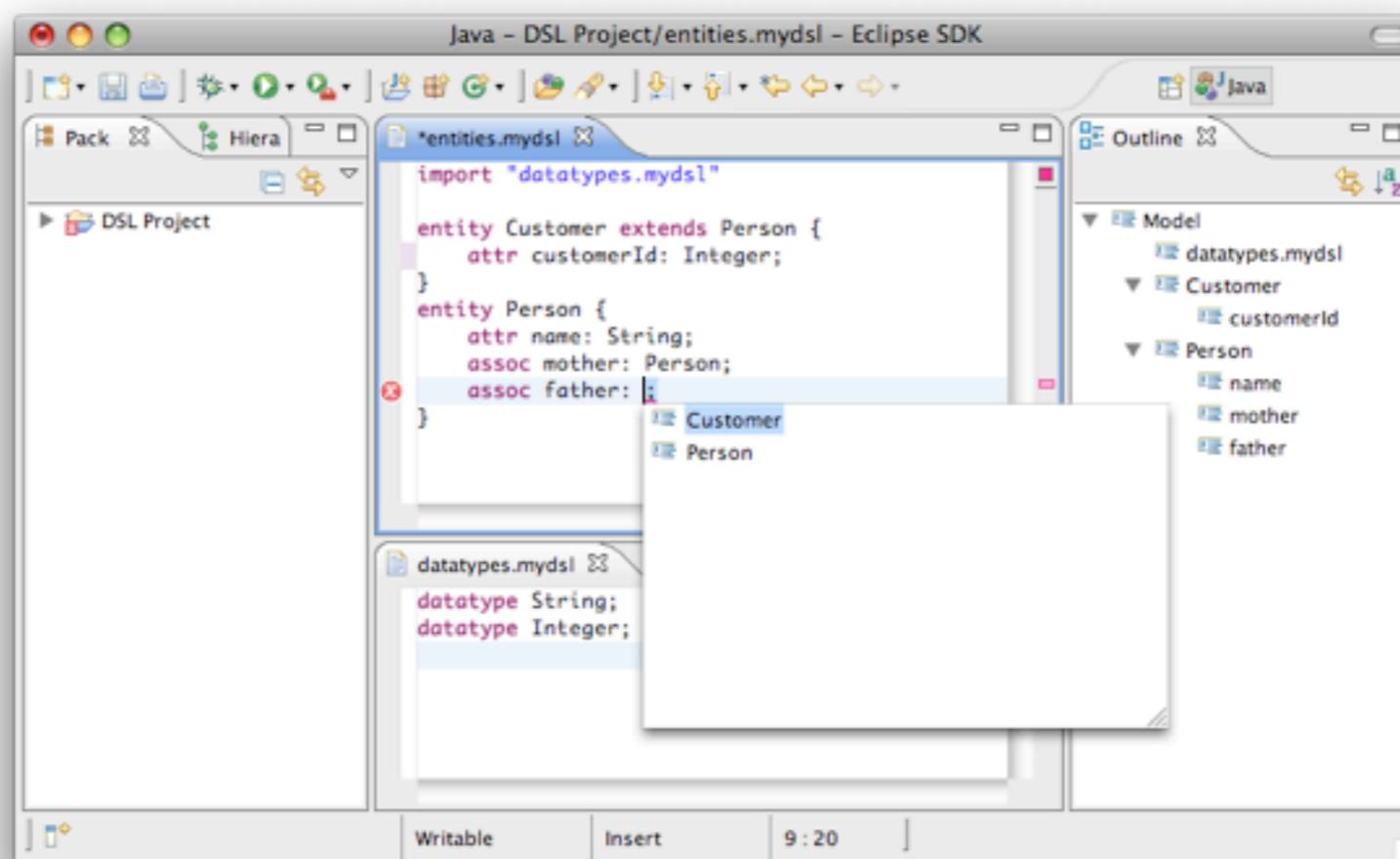
 Important for the success of model driven development

How can DSLs be defined?



Text based DSLs with Xtext

- Textual modeling of DSLs based on EMF
- Definition of abstract and concrete syntax with a EBNF based grammar



Xtext Features

- Generates the following artefacts from the grammar definition:
 - Incremental parser and lexer (ANTLR-3 based) - reads textual models.
 - Serializer - saving textual models.
 - Linker - manages cross document/model references
 - EMF Ecore meta model(s)
 - EMF Resource implementation - integrates the parser into the resource framework.
 - Eclipse IDE integration:
 - Eclipse Workbench UI Editor:
 - Syntax Coloring
 - Model Navigation (F3)
 - Code Completion

A close-up photograph of a person's hands typing on a light-colored computer keyboard. The hands are positioned over the keys, with fingers moving rapidly. A bright, semi-transparent white rectangular box containing the word "Demo..." is overlaid on the upper left portion of the image. The background is slightly blurred, showing what appears to be a wooden desk or surface.

Demo...

Component Model Architecture

- Each component is represented by a **Runnable**, which is executed by an RTOS **task** or an interrupt service routine
- Connections between components are implemented by inter-process communication (**IPC**) techniques
- Components communicate solely through their **interfaces**, i.e. communication paths are hidden from the components
- Component interfaces and IPC code are **generated** by the code generator based on the given component model

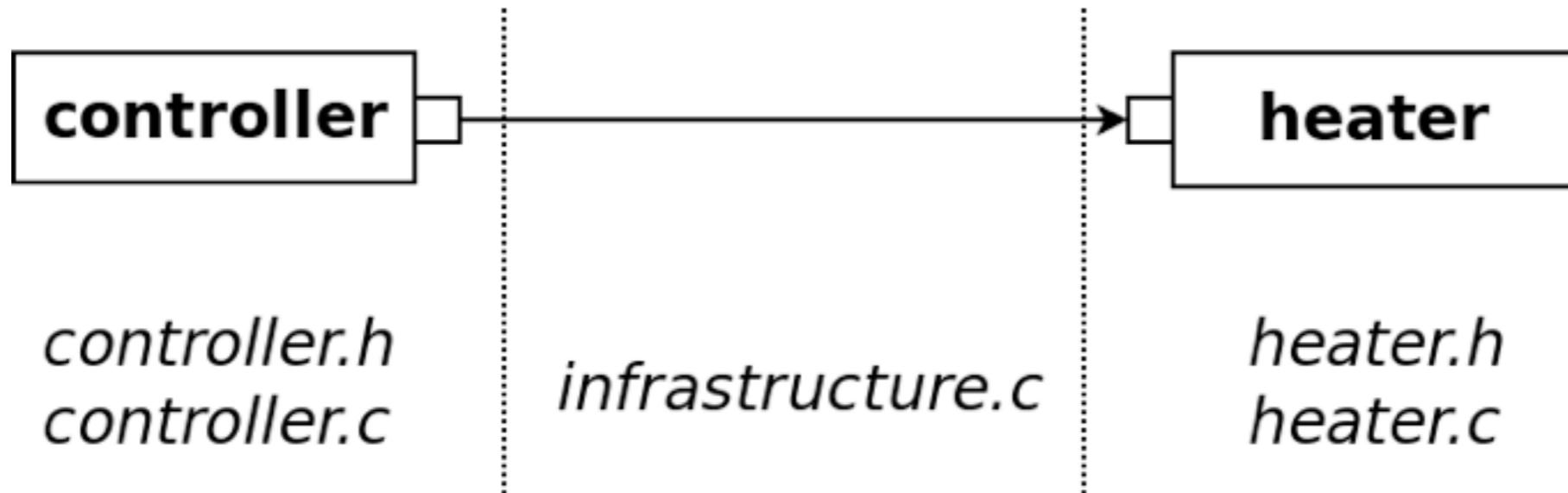
Advantages

- **Exchangeability:** Components and connections can be added, removed or changed without affecting other components or connections
- **Platform-independence:** Components have no knowledge of the underlying RTOS
- **Reusability:** Components can be reused without changing the implementation code
- **Testability:** Components can be tested separately

Generated Artifacts

- Main function
 - Hardware initialization stub
 - RTOS initialization routines
 - Task creation code for each component
 - Scheduler initialization code
- Function prototypes for each Runnable representing the component interface
- Inter-process communication code implementing the "wiring" between the components

Example



Example (cont.)

// Component Model

```
component Controller {
    out power;
}

component Heater {
    in power;
}

instance controller : Controller;
instance heater : Heater;

connect controller.power -> heater.power;
```

Example (cont.)

```
// controller.h
```

```
int controller_power_send(const int32_t  
*power);
```

```
// controller.c
```

```
void controller_Runnable(void) {
```

```
...
```

```
controller_power_send(&newPower);
```

```
...
```

```
}
```

```
// heater.h
```

```
int heater_power_receive(int32_t *power);
```

```
// heater.c
```

```
int heater_Runnable(void) {
```

```
...
```

```
heater_power_receive(&newPower);
```

```
...
```

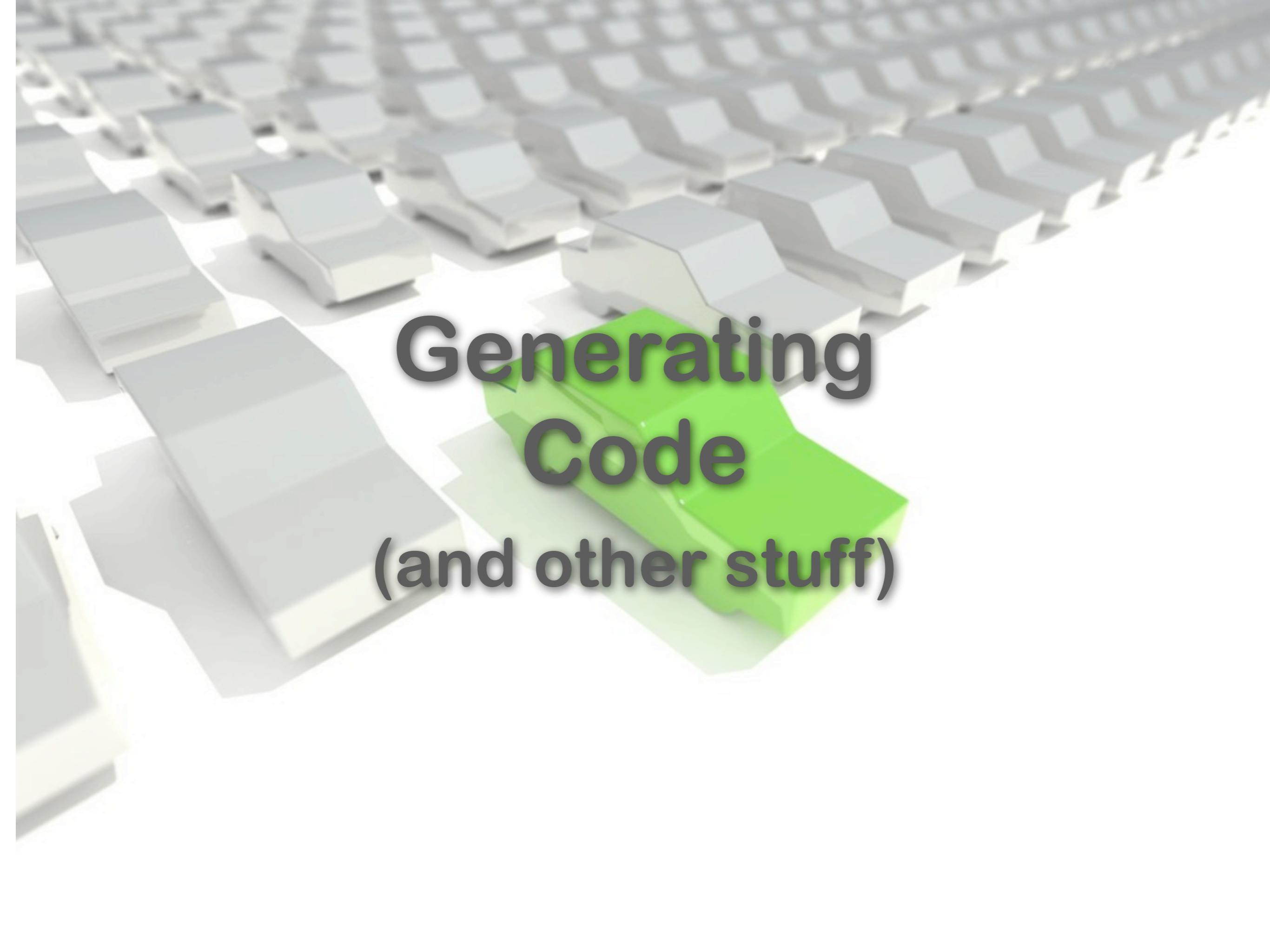
```
}
```

Example (cont.)

// *infrastructure.c*

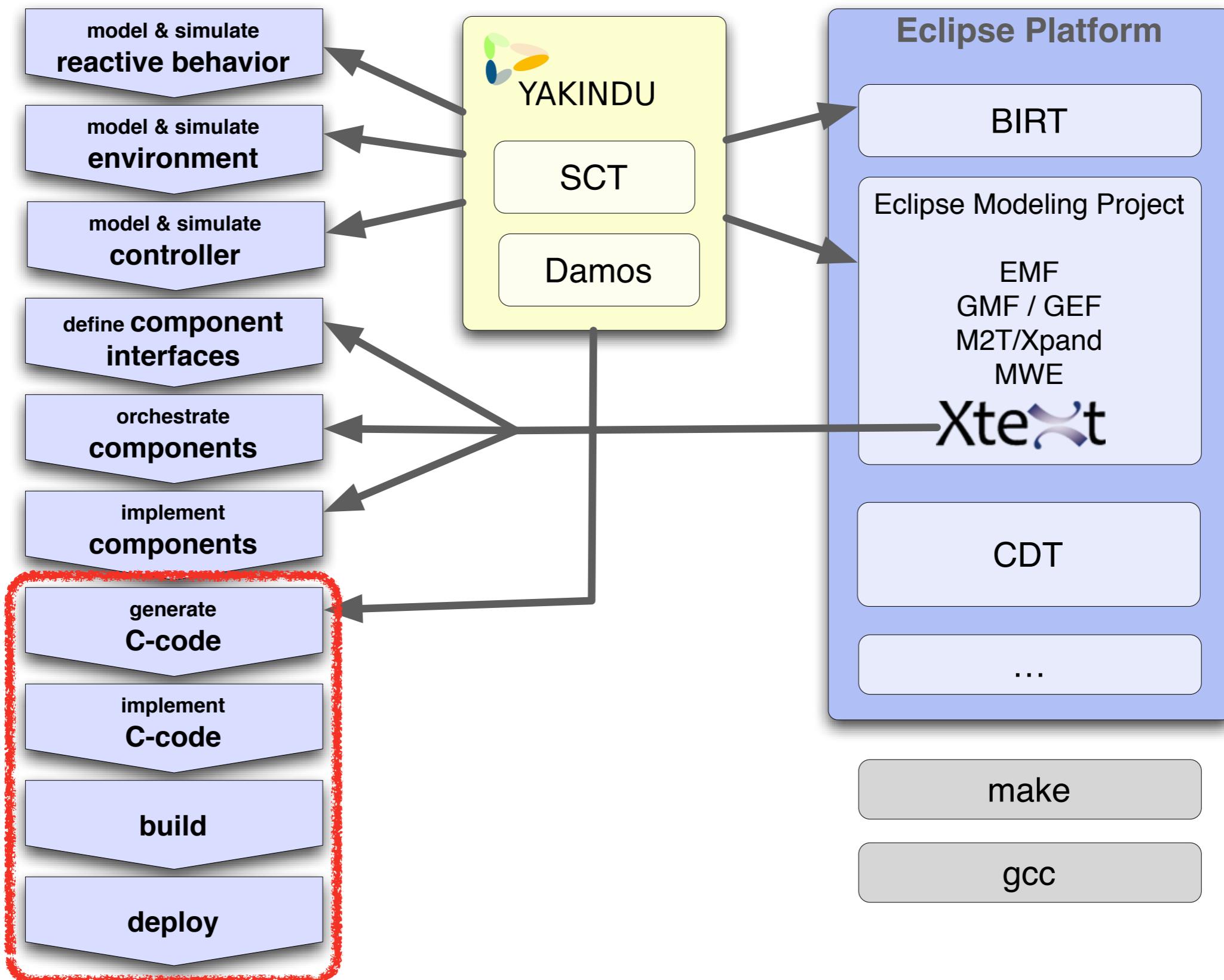
```
int controller_power_send(const int32_t
*power) {
    ...
    xQueueSend(
        heater_power_queue,
        (const void *) power,
        (portTickType) 0);
    ...
    return 1;
}

int heater_power_receive(int32_t *power) {
    return xQueueReceive(
        heater_power_queue,
        (void *) power,
        (portTickType) 0) == pdTRUE;
}
```



Generating Code (and other stuff)

Activities and Tools



Reusing Solutions

- code generators capture solution knowledge
- for each construct in a model the generator knows an implementation pattern
- variants of generators capture variants of solutions

Black Box Of The Shelf Code Generators

- ... are bound to a specific modeling language - mostly a GPL
- ... are bound to a method
- ... are bound to a specific (set of) target platform
- ... have limited capability for customization

If there is a Black Box Generator that fits well, then buy !!

Individual Code Generators

black box code generators may not fit due to

- ... specialized methodology DSLs or UML based
 - ... specific need regarding target platform
- then individual generators must be built

Building Individual Code Generators

Approach

- derive the generator from a reference implementation
- choose a template based approach that makes building the generator itself easy
- defining the reference implementation is the engineering task

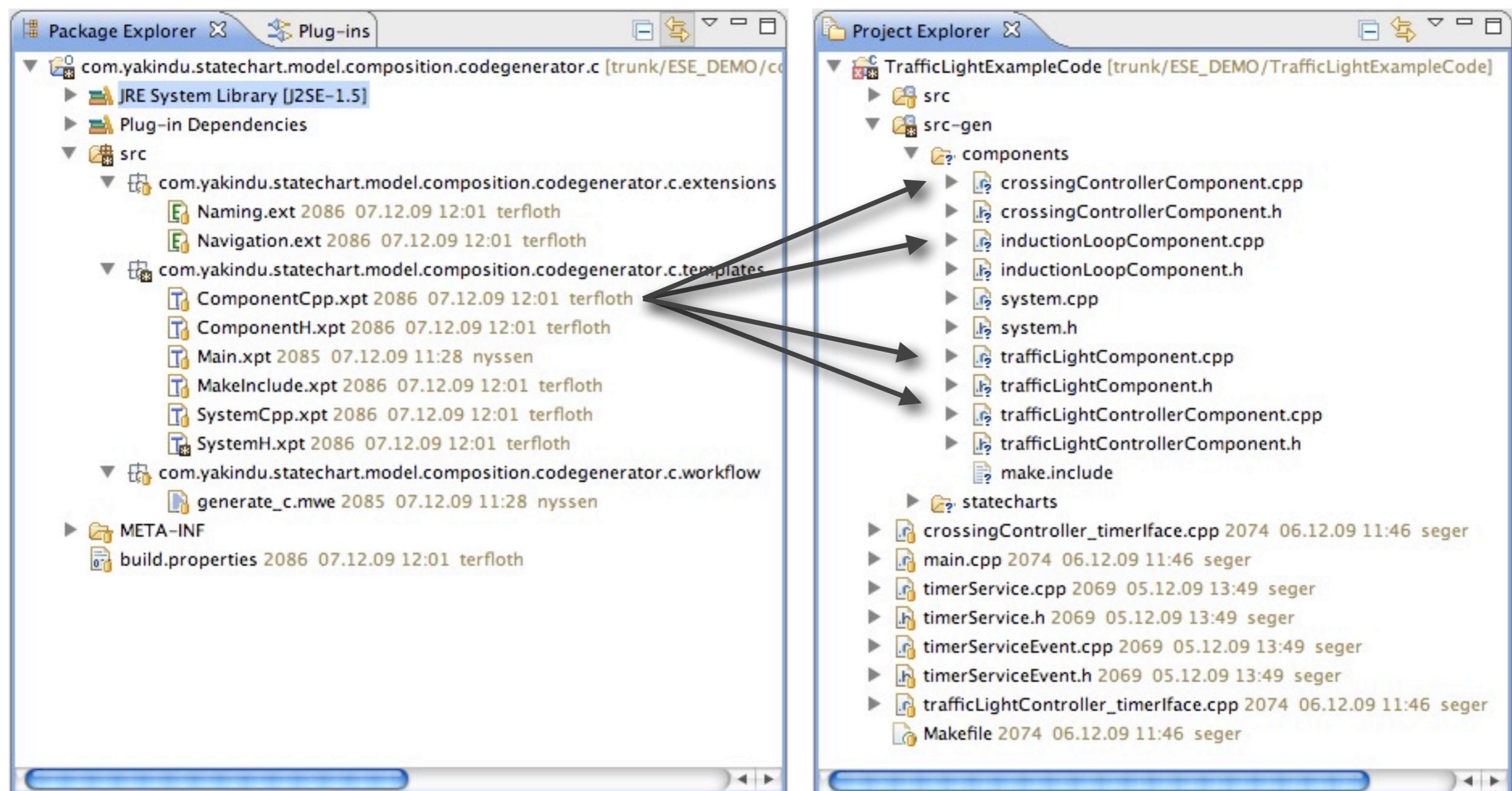
Creating Xpand Templates (2)

```
class System {  
public:  
  
    Scheduler* scheduler;  
  
    InductionLoopComponent* byRoadInduction;  
    TrafficLightComponent* byRoadLight;  
    TrafficLightComponent* mainRoadLight;  
    CrossingControllerComponent* controller;  
    TrafficLightControllerComponent* byRoadLightController;  
    TrafficLightControllerComponent* mainRoadLightController;  
  
    System(Scheduler& scheduler);  
    virtual ~System();  
  
    void connect();  
    void raiseTimerEvent(void* handle);  
};
```

```
«DEFINE Class FOR System»  
class System {  
public:  
  
    Scheduler* scheduler;  
  
    «FOREACH instances AS instance -»  
        «instance.type.typeName()»* «instance.memberName()»;  
    «ENDFOREACH»  
  
    System(Scheduler& scheduler);  
    virtual ~System();  
  
    void connect();  
    void raiseTimerEvent(void* handle);  
};  
«ENDEFINE»
```

variabler Code

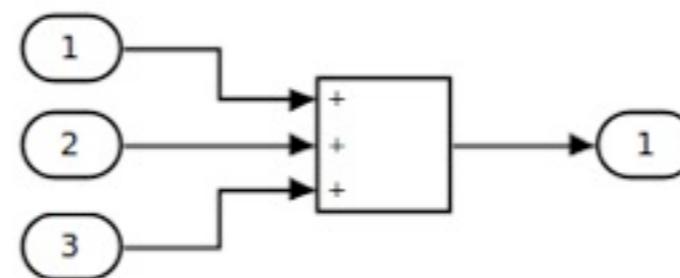
Code Generator



Efficient C Code Generation

- Is generated source code really inefficient?
- Efficient code generation depends on the quality of the code templates
- Well-written code templates produce efficient code for any input model (i.e. reproducible results)
- Different code templates can be used for different requirements
 - Easily readable and debugable code (but not so efficient)
 - Highly optimized code (e.g. tricky algorithms, macros vs function calls)
 - Or both?

Block Functions as Inline Functions or Macros



```
void BlockDiagram_execute(const BlockDiagram_Input *input,
BlockDiagram_Output *output) {
#define BLOCKDIAGRAM_SUM_LOCAL_DECLS
BLOCKDIAGRAM_SUM_LOCAL_DECLS
#endif /* BLOCKDIAGRAM_SUM_LOCAL_DECLS */

#define BLOCKDIAGRAM_SUM_LOCAL_INITS
BLOCKDIAGRAM_SUM_LOCAL_INITS
#endif /* BLOCKDIAGRAM_SUM_LOCAL_INITS */

BlockDiagram_Sum_Input_0_consumeInputValue(input->inport);
BlockDiagram_Sum_Input_1_consumeInputValue(input->inport2);
BlockDiagram_Sum_Input_2_consumeInputValue(input->inport3);
{
    int32_t value;
    BlockDiagram_Sum_computeOutputValues();
    value = BlockDiagram_Sum_Output_getOutputValue();
    output->outport = value;
}
BlockDiagram_Sum_update();
}
```

Block Functions as Inline Functions or Macros (cont.)

Expanded code:

```
void BlockDiagram_execute(
    const BlockDiagram_Input *input,
    BlockDiagram_Output *output) {

    int32_t BlockDiagram_Sum_sum = 0;

    BlockDiagram_Sum_sum += input->inport;
    BlockDiagram_Sum_sum += input->inport2;
    BlockDiagram_Sum_sum += input->inport3;

    {
        int32_t value;
        value = BlockDiagram_Sum_sum;
        output->outport = value;
    }
}
```

Code Optimization

```
BlockDiagram_execute:
push    {r7}
sub    sp, sp, #60
add    r7, sp, #0
str    r0, [r7, #4]
str    r1, [r7, #0]
mov    r3, #0
str    r3, [r7, #12]
mov    r3, #0
str    r3, [r7, #16]
mov    r3, #0
str    r3, [r7, #20]
mov    r3, #0
str    r3, [r7, #28]
mov    r3, #0
str    r3, [r7, #32]
movw   r3,
#:lower16:BlockDiagram_DiscreteIntegrator_sum
movt   r3,
#:upper16:BlockDiagram_DiscreteIntegrator_sum
ldr    r3, [r3, #0]
lsl    r2, r3, #8
movw   r3, #:lower16:274877907
movt   r3, #:upper16:274877907
smull  r1, r3, r3, r2
asr    r1, r3, #17
asr    r3, r2, #31
rsb    r3, r3, r1
str    r3, [r7, #8]
ldr    r3, [r7, #8]
str    r3, [r7, #36]
ldr    r2, [r7, #32]
ldr    r3, [r7, #36]
add    r3, r2, r3
str    r3, [r7, #32]
ldr    r3, [r7, #4]
ldr    r3, [r3, #0]
ldr    r2, [r7, #16]
add    r3, r2, r3
str    r3, [r7, #16]
ldr    r3, [r7, #4]
ldr    r3, [r3, #4]
ldr    r2, [r7, #16]
rsb    r3, r3, r2
str    r3, [r7, #16]
ldr    r3, [r7, #16]
str    r3, [r7, #40]
ldr    r3, [r7, #40]
lsl    r3, r3, #10
asr    r3, r3, #8
str    r3, [r7, #12]
ldr    r3, [r7, #40]
lsl    r3, r3, #10
asr    r3, r3, #8
str    r3, [r7, #12]
ldr    r3, [r7, #40]
lsl    r3, r3, #10
str    r3, [r7, #20]
ldr    r3, [r7, #40]
str    r3, [r7, #28]
ldr    r3, [r7, #20]
str    r3, [r7, #44]
ldr    r2, [r7, #32]
ldr    r3, [r7, #44]
add    r3, r2, r3
str    r3, [r7, #32]
movw   r3,
#:lower16:BlockDiagram_DiscreteDerivative_previousValue
e
movt   r3,
#:upper16:BlockDiagram_DiscreteDerivative_previousValue
ue
ldr    r3, [r3, #0]
ldr    r2, [r7, #28]
rsb    r3, r3, r2
str    r3, [r7, #24]
ldr    r3, [r7, #24]
mov    r2, #2048000
mul    r3, r2, r3
asr    r3, r3, #8
str    r3, [r7, #24]
ldr    r3, [r7, #24]
lsl    r3, r3, #10
asr    r3, r3, #8
str    r3, [r7, #24]
ldr    r3, [r7, #24]
str    r3, [r7, #48]
ldr    r2, [r7, #32]
ldr    r3, [r7, #48]
add    r3, r2, r3
str    r3, [r7, #32]
ldr    r3, [r7, #32]
str    r3, [r7, #52]
ldr    r3, [r7, #0]
ldr    r2, [r7, #52]
str    r2, [r3, #0]
movw   r3,
#:lower16:BlockDiagram_DiscreteIntegrator_sum
movt   r3,
#:upper16:BlockDiagram_DiscreteIntegrator_sum
ldr    r2, [r3, #0]
ldr    r3, [r7, #12]
add    r2, r2, r3
movw   r3,
#:lower16:BlockDiagram_DiscreteIntegrator_sum
movt   r3,
#:upper16:BlockDiagram_DiscreteIntegrator_sum
str    r2, [r3, #0]
movw   r3,
#:lower16:BlockDiagram_DiscreteDerivative_previousValue
e
movt   r3,
#:upper16:BlockDiagram_DiscreteDerivative_previousValue
ue
ldr    r2, [r7, #28]
str    r2, [r3, #0]
add    r7, r7, #60
mov    sp, r7
pop    {r7}
bx    lr
```

Generated PID controller code

CFLAGS: -O0

102 lines of code!

Code Optimization (cont.)

```
BlockDiagram_execute:  
movw r3, #:lower16:.LANCHOR0  
movt r3, #:upper16:.LANCHOR0  
push {r4, r5, r6, r7}  
ldr r2, [r0, #4]  
ldr r5, [r0, #0]  
ldr r4, [r3, #0]  
ldr r6, [r3, #4]  
movw r0, #:lower16:274877907  
subs r2, r5, r2  
lsl ip, r4, #8  
movt r0, #:upper16:274877907  
smull r7, r5, r0, ip  
subs r6, r2, r6  
mov r7, #2048000  
mul r6, r7, r6  
asr ip, ip, #31  
lsls r7, r2, #10  
rsb r5, ip, r5, asr #17  
asr ip, r7, #8  
add r5, ip, r5  
sbfx r0, r6, #6, #24  
adds r0, r5, r0  
add r4, ip, r4  
str r0, [r1, #0]  
str r2, [r3, #4]  
str r4, [r3, #0]  
pop {r4, r5, r6, r7}  
bx lr
```

Generated PID controller code
CFLAGS: -O2
28 lines of code!



**Questions &
Comments**