GRNSPG Working Seminar
Methodologies for I&C Systems Specification and Design
Pisa, 2 September, 2009

Andrea Domenici

DIIEIT, Università di Pisa

Andrea.Domenici@iet.unipi.it

Outline

- DEVELOPMENT OF SAFE SOFTWARE
  - Issues and Requirements
- Part 1: THE OBJECT-ORIENTED APPROACH
  - Structural Modeling
  - Behavioral Modeling
  - Component-oriented Modeling
  - Development and Verification
- THE FORMAL APPROACH
  - Process Algebras and Temporal Logics
  - Model Checking
  - Example
- THE AUTOMATED DEVELOPMENT APPROACH
  - The TXS SPACE Development Environment

DEVELOPMENT OF SAFE SOFTWARE

Requirements (1)

The three main requirements for I&C software in NPP's are:

1. Safety,
2. safety, and
3. safety.

. . . but let us take a closer look.

Requirements (2)

- **Fault avoidance**: *Software quality* is the first line of defense.
  - Rigorous development process.
  - Testing, verification, and validation.
  - Competence and experience.
- **Fault tolerance**: Specific provisions designed into the software are the second line of defense, against undetected design (or coding) errors and unforeseen accidents.
  - Compliance with system-level safety requirements and design.
  - System-level exception handling.
  - Choice of fault-tolerant mechanisms (redundancy, diversity, voting schemes. . .).
  - Issues specific to computer systems (interrupts, memory management, clocks. . .).

HW and SW issues affecting reliability

- Operating system and processes.
- Interrupt handling, clocks, and scheduling.
- Memory management.
- Loops and other programming issues.

Operating system and processes

The *operating system* (OS) has direct control on the physical resources (CPU, memory, peripherals) of a computer system.

User applications are physically prohibited from accessing the computer resources directly: they must go through the OS by issuing *system calls* (except in *very* simple systems).

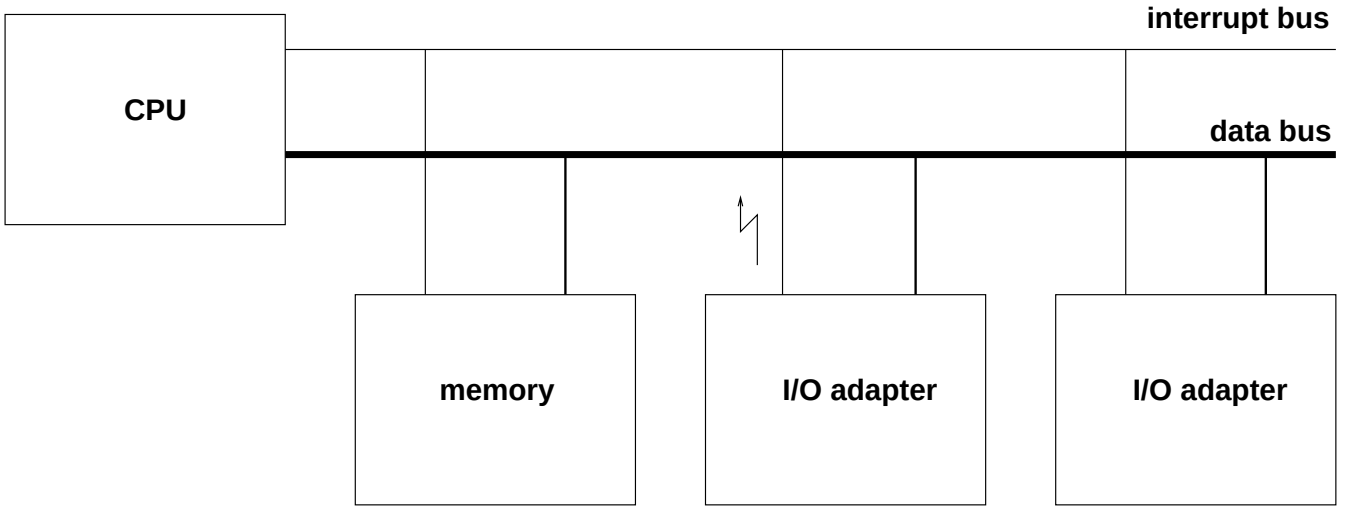
A *process* is a program in execution. Processes are controlled by the OS, that starts their execution and *allocates* (assigns) resources (including execution time-slots).

OS's support *multitasking*, i.e. concurrent execution of multiple proceses.

General-purpose OS's (e.g., Linux, Mac OSX, MS Windows) handle multitasking with the goal of maximizing throughput and average response time.

Real-time OS's must ensure respect of execution deadlines and priorities among processes.

External Interrupts



External interrupts are signals that peripheral devices send asynchronously to the CPU on the occurrence of some events, such as completion of a HW operation, availability of input data, HW faults. . .

Interrupt handling, clocks, and scheduling

When the CPU receives an interrupt, it suspends its current activity, executes an *interrupt handler* routine associated with the specific signal, then returns to the previous activity.

Internal interrupts may be generated by program errors (e.g., division by zero or access to forbidden memory areas) or system calls.

Clock interrupts are used for various timing purposes, including the demarcation of time-slices allotted to processes.

Interrupts are a fundamental mechanism in the OS and are very convenient in general-purpose time-sharing environments, but they may introduce *nondeterminism*, i.e., unpredictable behavior.

Real-time and *control* systems often reduce usage of interrupts to a minimum and resort to *cyclic execution*: all operations are executed in a fixed sequence within a main operation cycle, and no operation can be interrupted by another task. The CPU repeats the cycle indefinitely.

Memory management

A process memory space can be managed in three ways:

- *Static*: allocated at the beginning of process execution, released at the end. The space size is known in advance (computed by the compiler) and fixed.
- *Automatic*: allocated for each subroutine of the process when the subroutine is called, released when it exits. The space size is known in advance and fixed.
- *Dynamic*: allocated on demand at any time during execution, with a size known only at execution time. The space may be released by the process explicitly, or (depending on the language) reclaimed by an underlying *garbage collection* mechanism.

Dynamic memory management allows for a very flexible and efficient use of memory, but it may introduce nondeterminism due to “out of memory” failures or untimely intervention of the garbage collector. Programming errors are possible with explicit memory release.

As a result, some development methods or tools avoid dynamic memory management.

Example: Memory Management in C++

```
int n = 0;                  // a static variable
                            // (declared outside
                            // any function)
void some_function()
{
    float x;                 // an automatic variable
    float* p =               // a pointer to an array
                             // of 4 numbers...
       new float[4];         // ... created in
                             // dynamic memory
    p[0] = 3.14;
    // ...
    delete[] p;              // release memory
}
```

Example: Loops

```
for (int i = 0; i++; i < 100 {
    // this loop will be executed
    // no more than 100 times
};

float x = 100.0;
while (x != 0) {   // not equal
    // we do not know how many times
    // this loop will be executed...
    // maybe for ever!
    x = ... ;      // change x
}
```

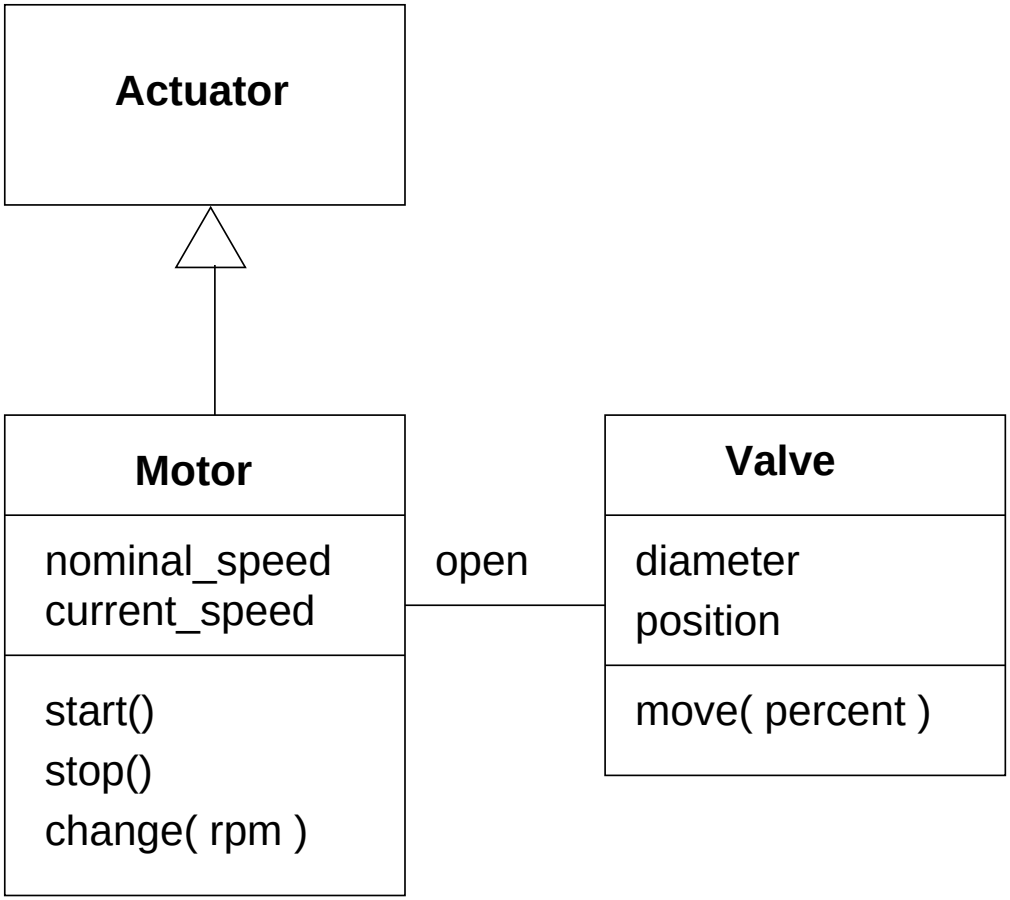
THE OBJECT-ORIENTED APPROACH

Simulating Real-world Objects

O-O methods and languages (e.g., UML [1]) are based on *simulation*.

- An *object* represents a real-world entity, concrete (e.g., a motor) or abstract (e.g., a system of equations).
- An object is defined by its *identity*, by the values of the entity *attributes* (e.g., a motor's nameplate data, current speed and torque. . .) and by the *operations* the entity can execute on request by other entities (e.g., changing the speed. . .).
- *Links* between objects represent logical relationships between the corresponding entities. E.g. a motor *opens* a valve, a vector *satisfies* a set of equations.
- A *class* is a template description for a set of objects having the same attributes (possibly with different values) and operations.
- An *association* between two classes is a template for the links between the respective objects.

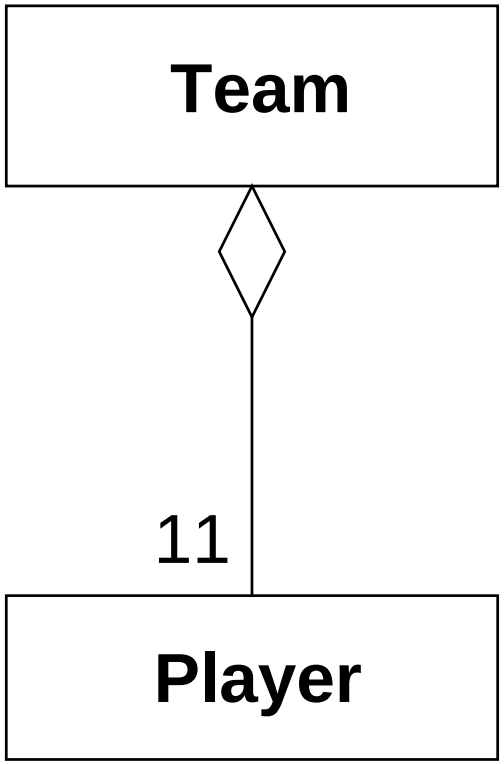
Example: A Very Simple Class Diagram



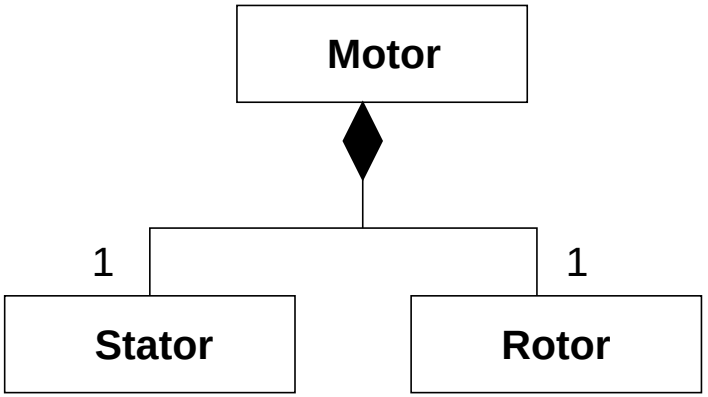
Special Associations and Relationships

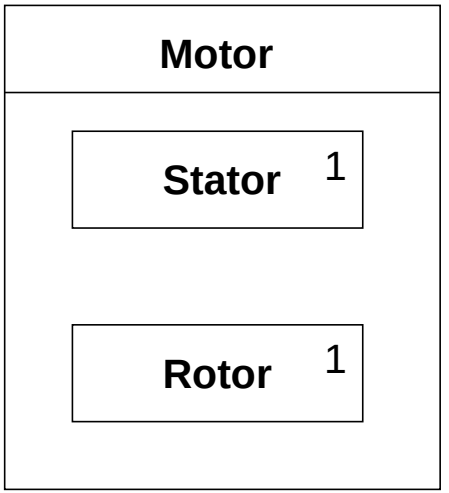
- *Aggregation*: The (weak) association of a compound entity with its components, when the components may exist outside the compound entity (e.g. a team and its players, a library and its books).
- *Composition*: The (strong) association of a compound entity with its components, when the components may not exist outside the compound entity (e.g. a motor and its parts).
- *Generalization*: A *relationship* (not an association) specifying that a class is a subset of another one (e.g., motors are a subclass of actuators).

Example: Aggregation



Example: Composition



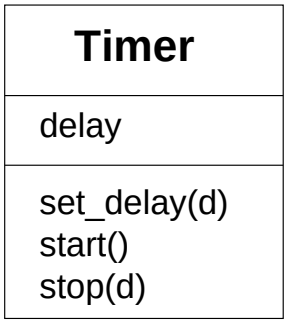


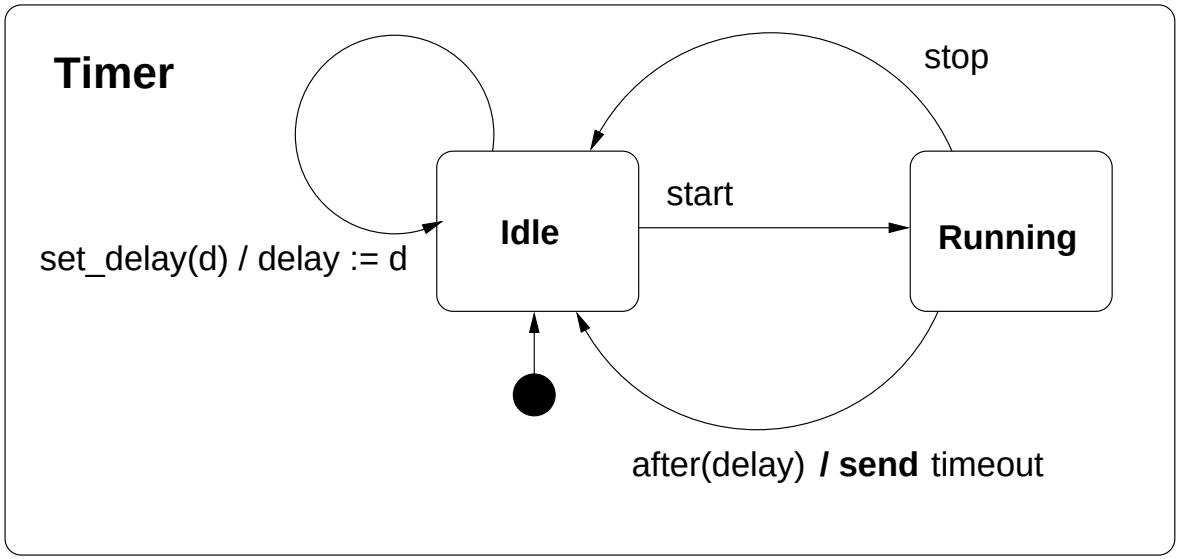
Behavioral Modeling

Modeling languages such as the UML complement the basic OO structural concepts with behavioral concepts drawn from other modeling approaches:

- *State Machines* describe how an object or (sub)system responds to events. The UML uses a complex state machine language derived from the *Statecharts* formalism [2].
- *Interactions* describe how objects or (sub)systems interact by exchanging messages.
- *Activities* describe the flow of control and data involved in carrying out a task.

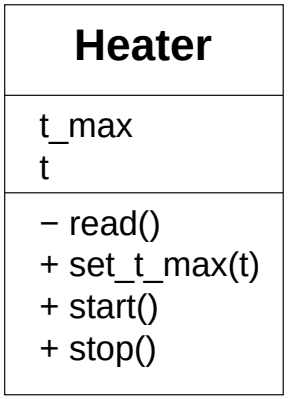
Example: State Machines (1)

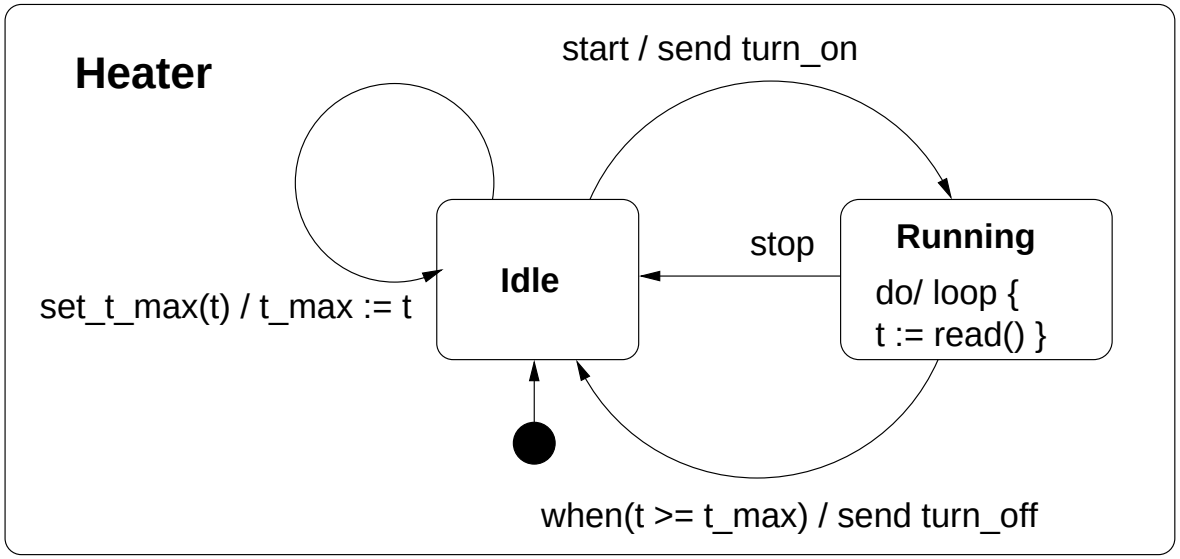




after(. . .): fire transition after *delay* seconds (*time event*)
send: generate a timeout *signal*

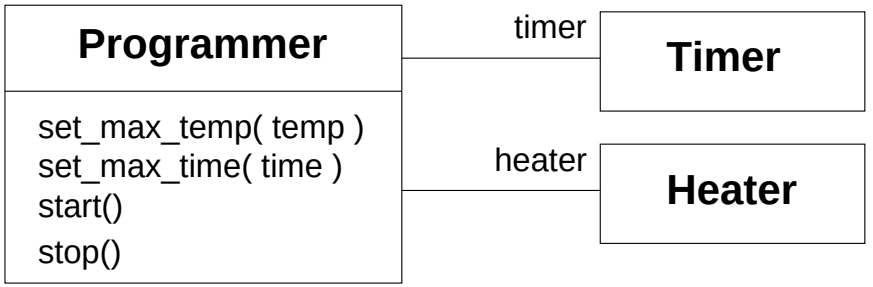
Example: State Machines (2)

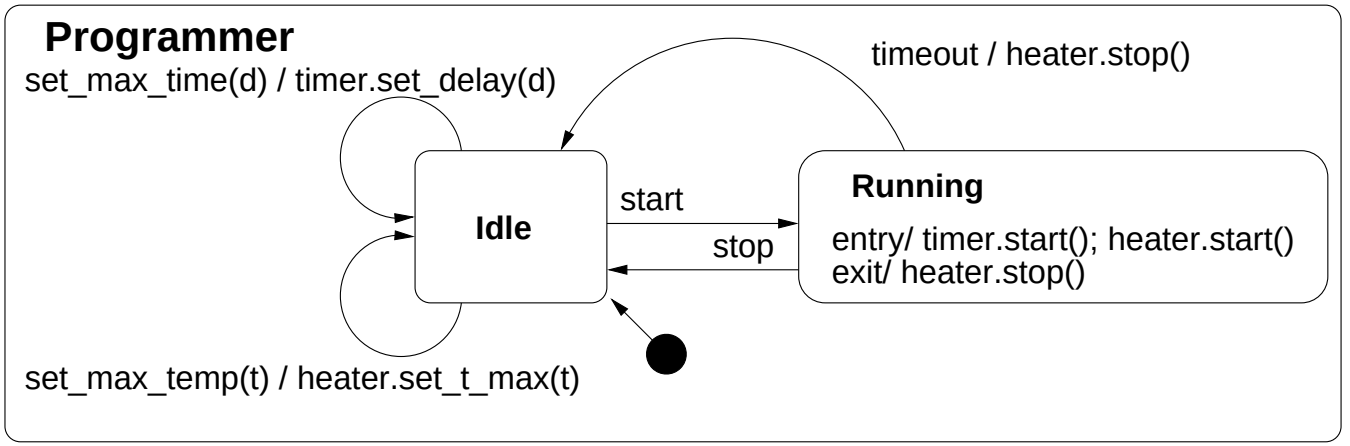




when(. . .): fire transition when a condition becomes true (*change event*)
do/: perform an *activity* while in a given state.

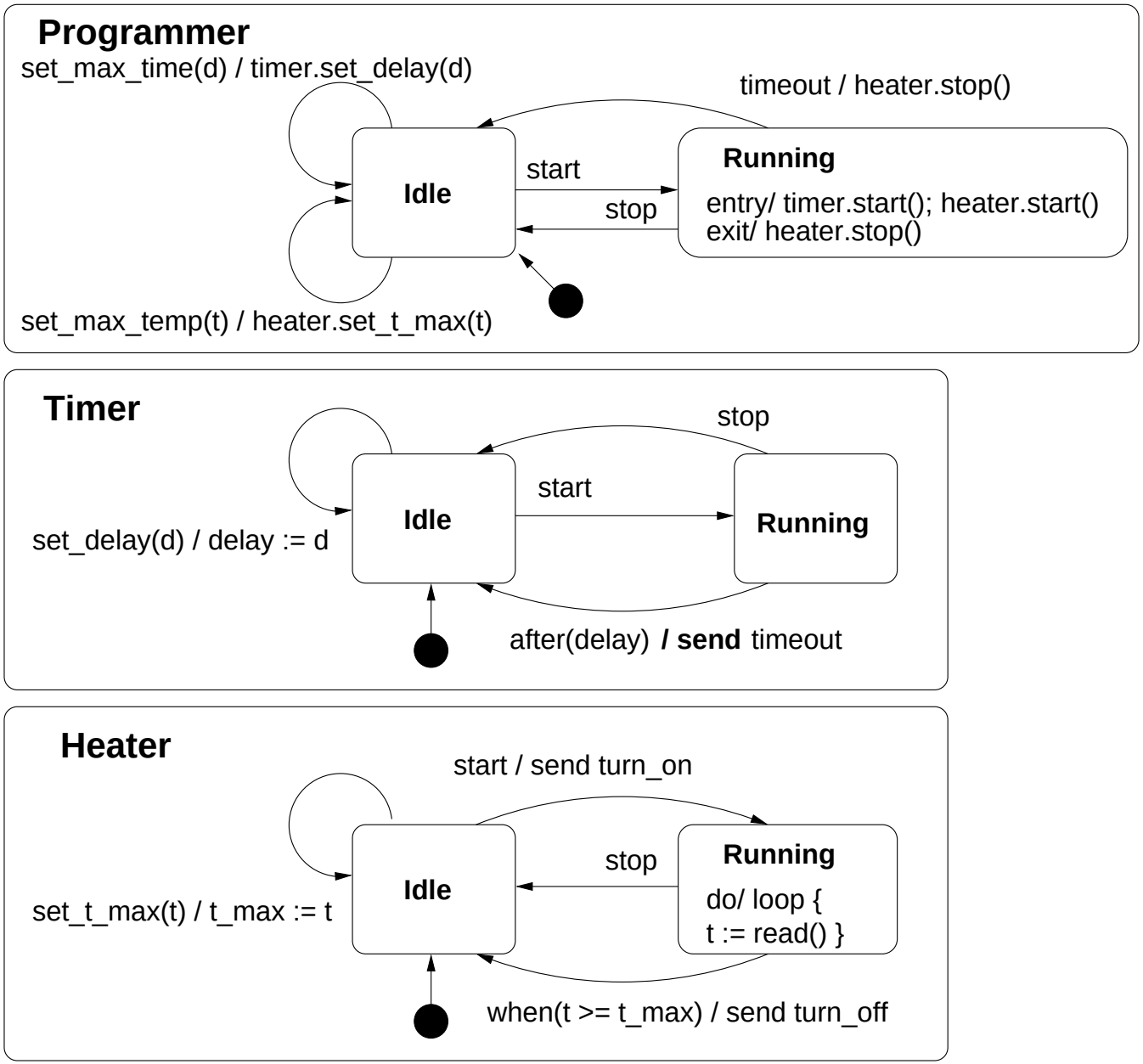
Example: State Machines (3)



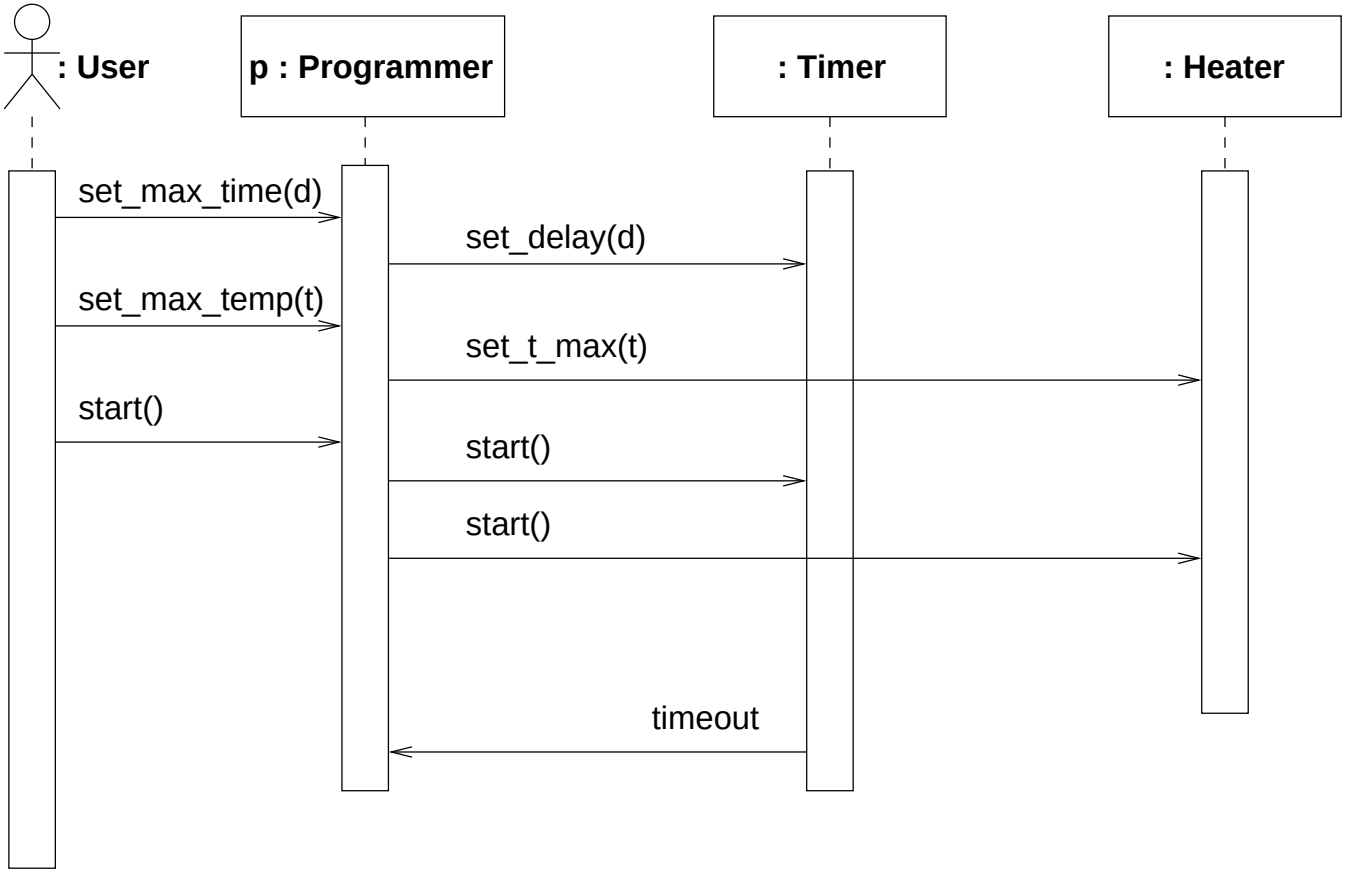


heater, timer: *rolenames* to identify participants in an association
entry/: activity performed when entering a state
exit/: activity performed when leaving a state

Example: State Machines (4)

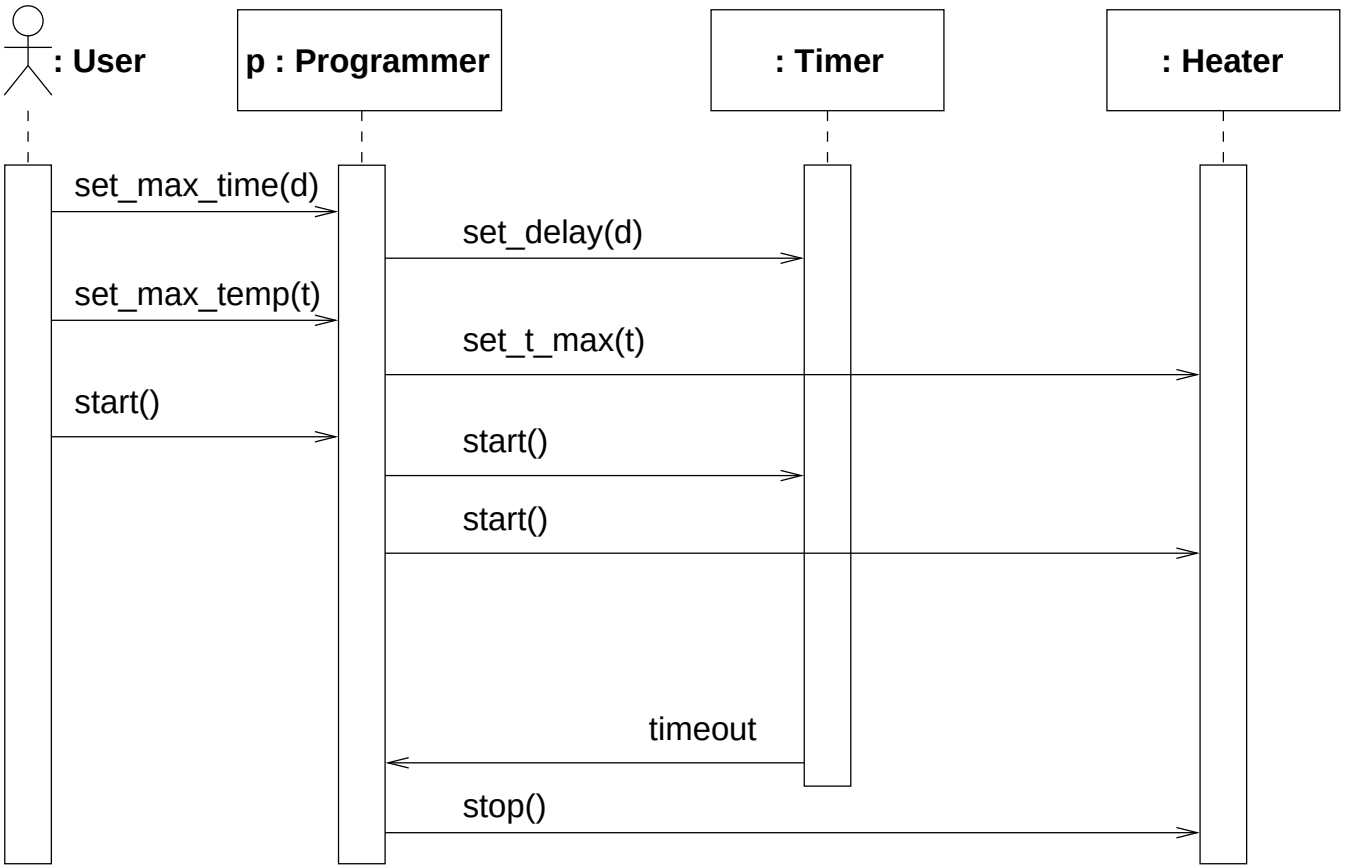


Example: Sequence Diagram



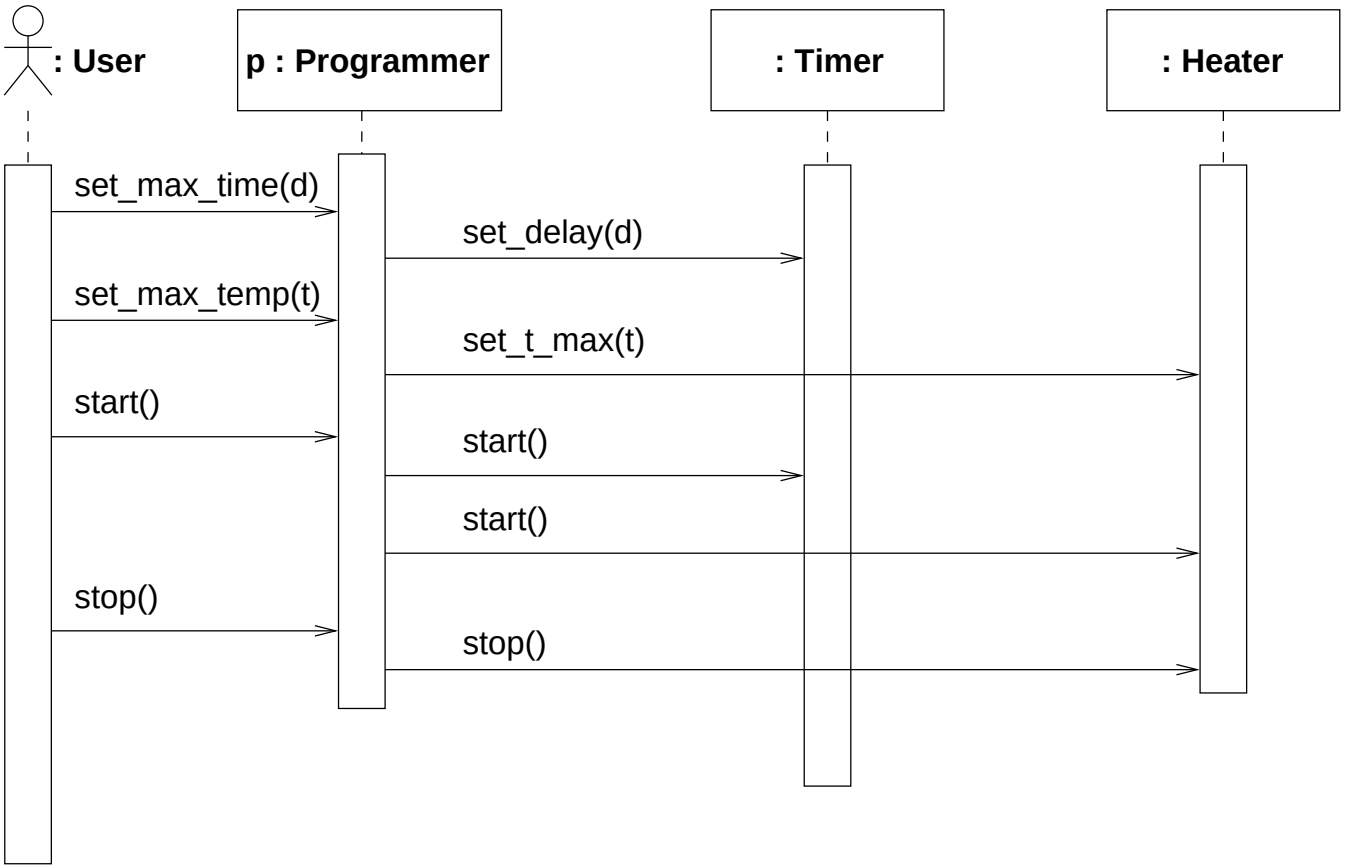
Scenario 1: the heater reaches the limit temperature before timeout or manual stop.

Example: Sequence Diagram



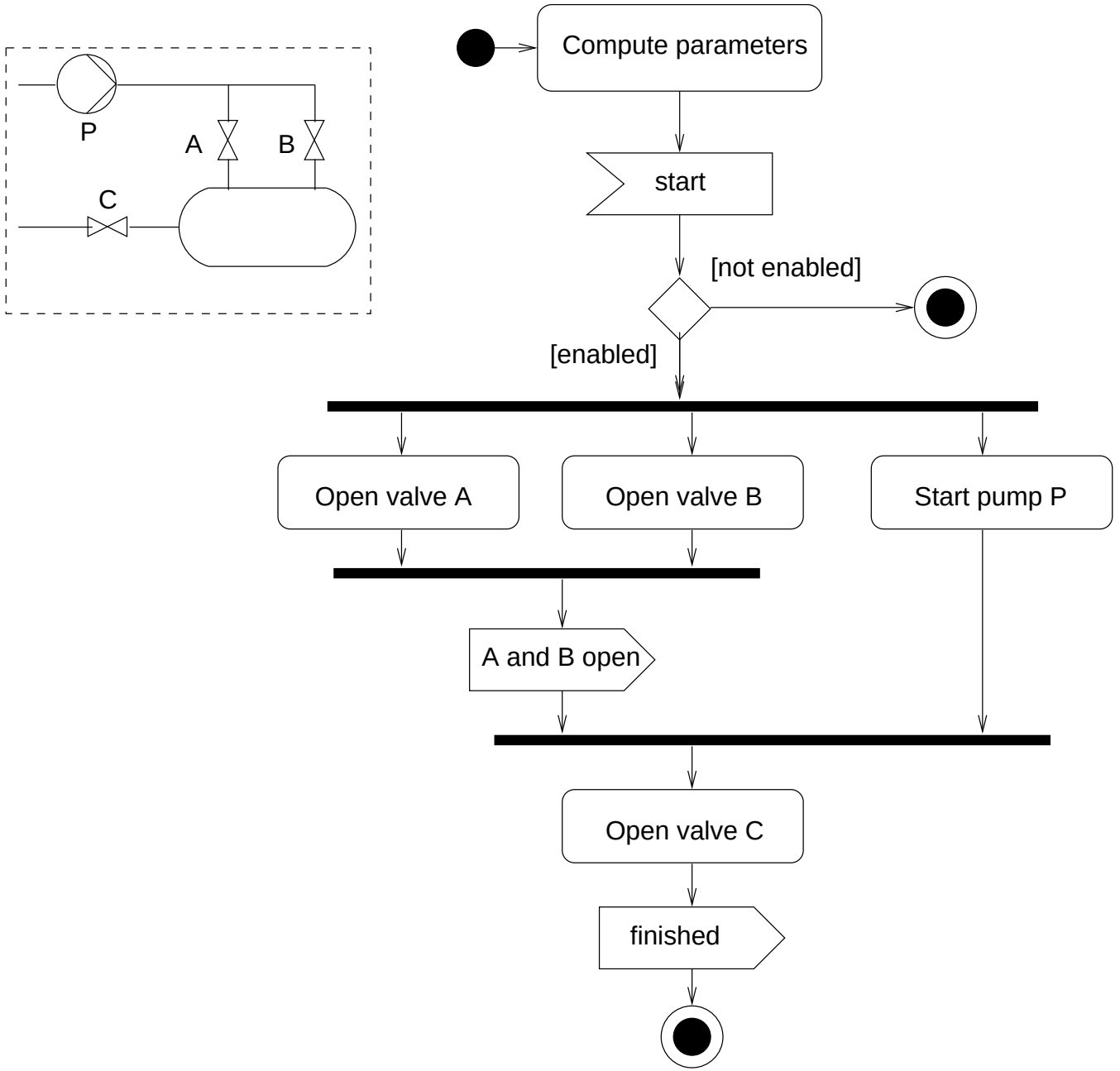
Scenario 2: timeout occurs before the heater reaches the limit temperature.

Example: Sequence Diagram



Scenario 3: manual stop.

Example: Activity Diagram

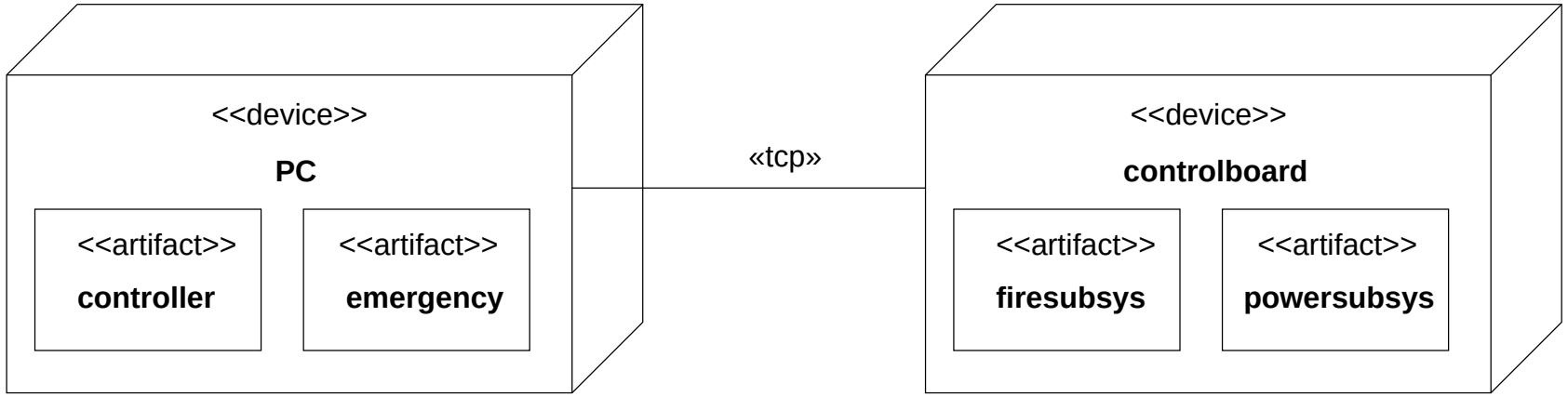


Physical Modeling

Deployment diagrams describe the physical structure of a system:

- *Nodes* represent hardware devices performing computations (typically, whole computer, or lower-level parts if necessary).
- *Artifacts* represent files (containing programs or data).
- *Associations* represent communication paths between nodes.

Example: Deployment Diagram



Design Modeling

An object-oriented design evolves from the analysis model. The analysis model describes the system in terms of its externally visible properties and behavior, while the design model describes its internal structure.

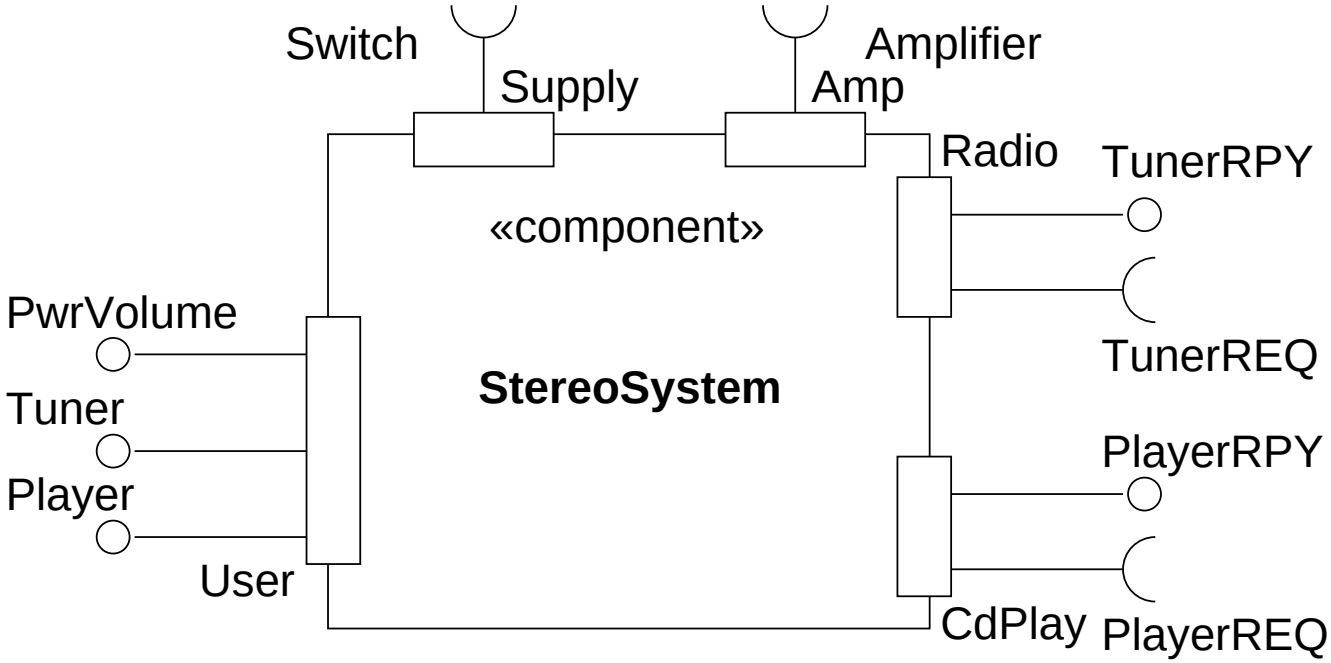
In the analysis model we find classes that represent real-world entities; in the design model we have the same classes (or similar classes related to them) that *simulate* (i.e., define a software model of) the real-world entities or interface them with the system.

Classes and associations in the design model add detailed information to those in the analysis model.

The *component* diagram element models a replaceable part of a system, defined by its provided and required interfaces. A component may have more than one provided or required interface: this enables designers to specify precisely the dependencies between components. A group of related interfaces of a component is a *port*.

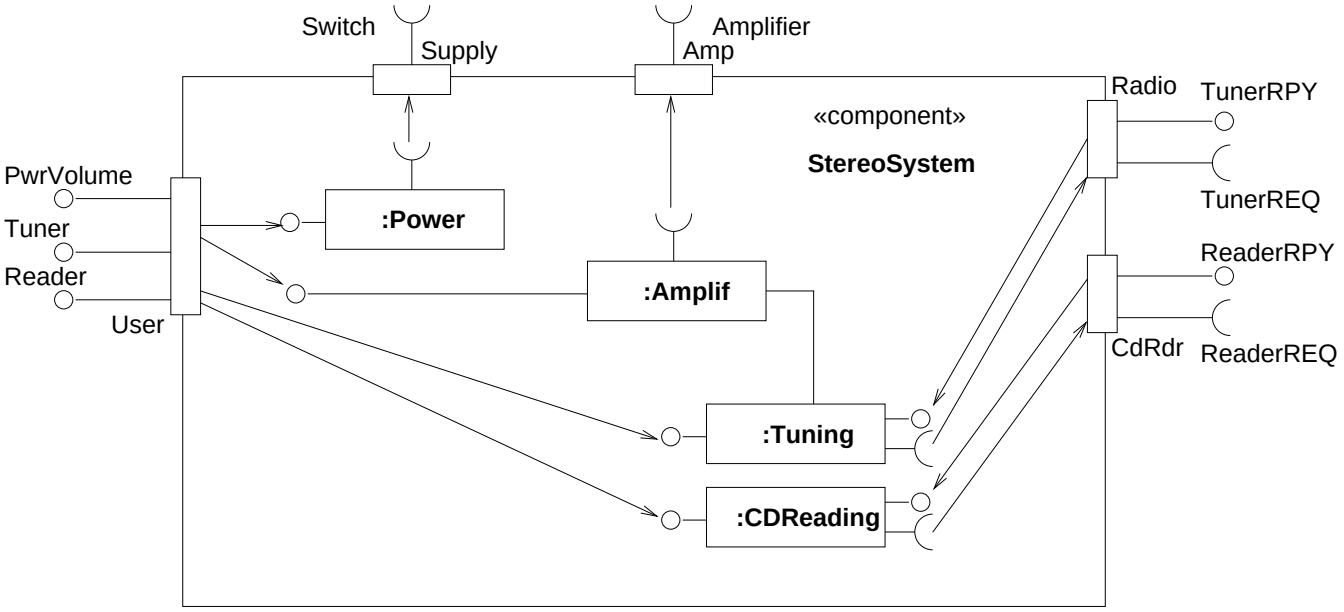
A component usually contains other components or classes. Components enable designers to build hierarchical models.

Example: Component Diagrams (1)



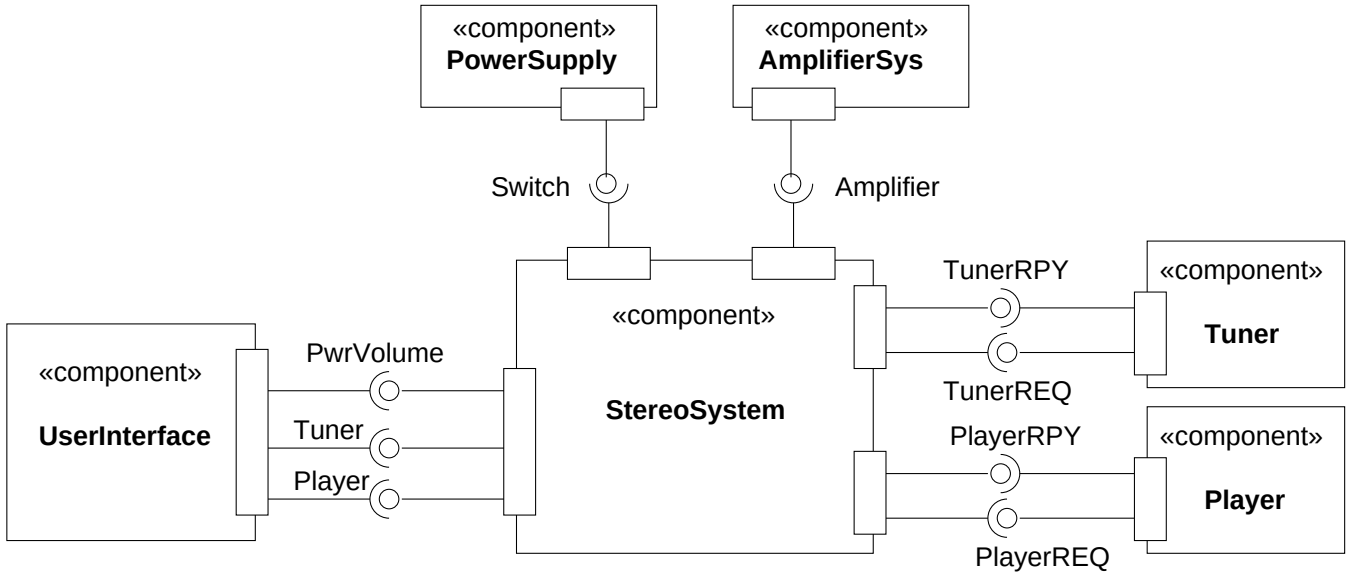
Switch, Amplifier, TunerREQ, PlayerREQ: *required* interfaces (sets of operations called by the component)
PwrVolume, Tuner, Player, TunerRPY, PlayerRPY: *provided* interfaces (sets of operations called by other components)

Example: Component Diagrams (2)



Internal structure of the component.

Example: Component Diagrams (3)



Interconnected components.

Coding

OO languages, such as C++, directly map the concepts of *class* and *generalization* into programming language constructs.

Other concepts (such as associations) are not supported directly and may be implemented in different ways.

Support for Development and Verification

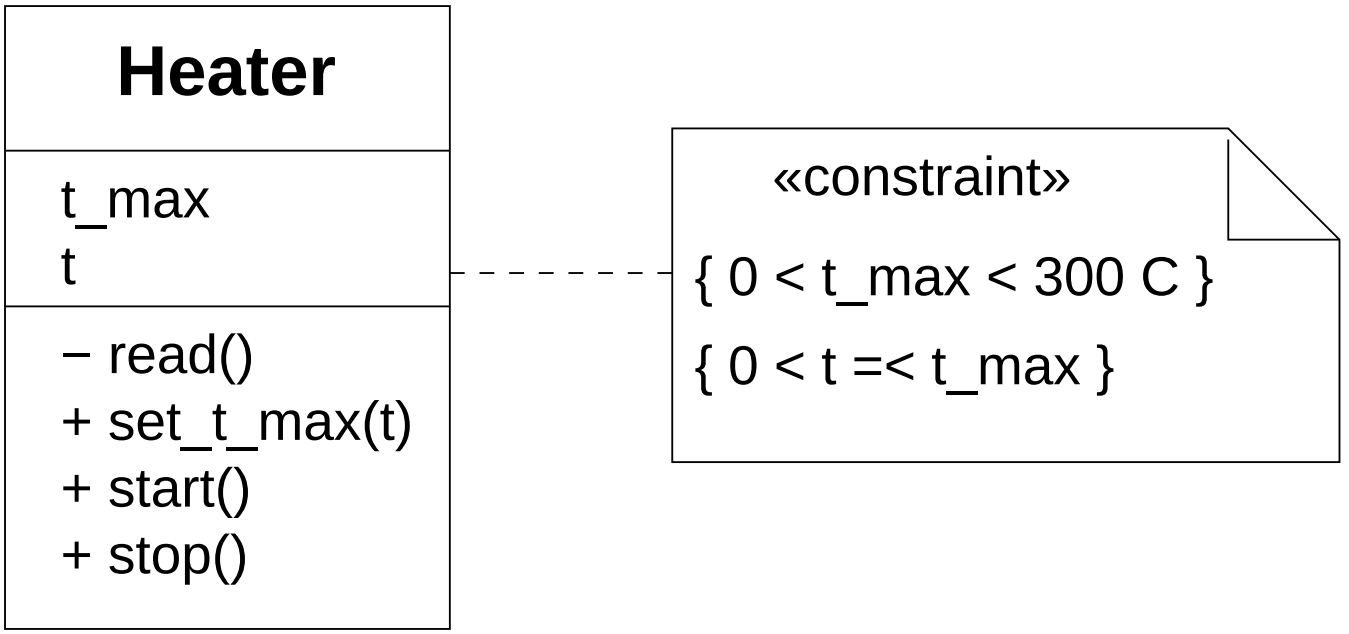
Even if the UML is a semi-formal language, it allows precise and detailed models to be built, thus enabling tools to automatically generate much of the source code.

Tools may also check against errors and inconsistencies.

Any model element may be annotated with *comments* and *constraints*.

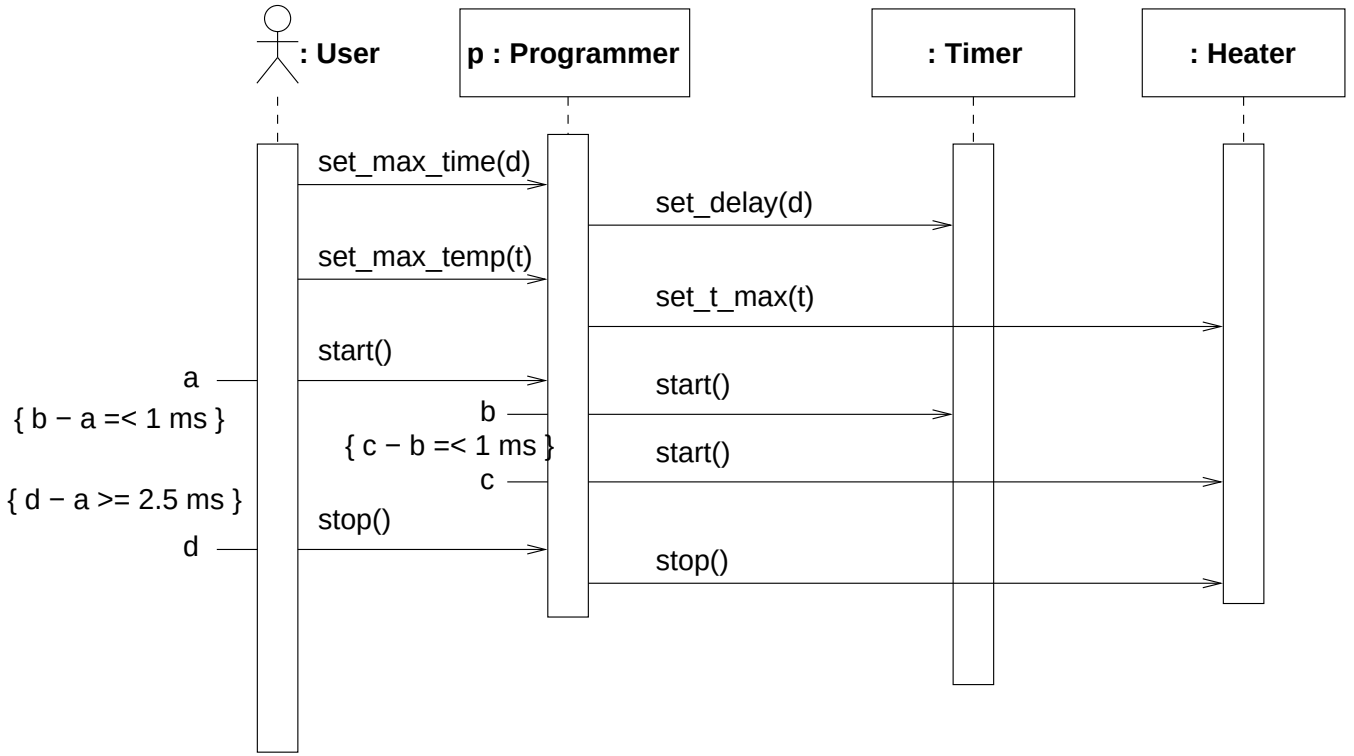
Constraints may be expressed in a formal language, including the UML-specific *Object Constraint Language* (OCL).

Example: Constraints (1)



Constraints on attribute values.

Example: Constraints (2)



a, b, c, and d are *timing marks*
the constraints on the diagram specify that a stop request must not be issued before both the timer and the heater have been started.

THE FORMAL APPROACH

Modeling: Process Algebras and LTS's

- A *process algebra* [3, 4] is a language that describes systems in terms of sets of *states* and sequences of *actions*.
  - More precisely, it describes the actions that a system may execute at each step of its evolution. A state is defined *implicitly* by the set of action that are possible at a given step.
- The algebra defines operations that act on the elements of the domain, such as forming sets and sequences, and combining them in various ways.
- E.g., we may define operations for *parallel* and *sequential* composition to describe the interaction of two processes.
- A system is described by a process algebra formula, that in turn is represented by a *Labelled Transition System* (LTS), a graph whose nodes are states and whose edges are transitions.
- A LTS may also model the behavior of a state machine, or conversely, state machines are an alternative way to specify a LTS.

Equivalence of Labeled Transition Systems

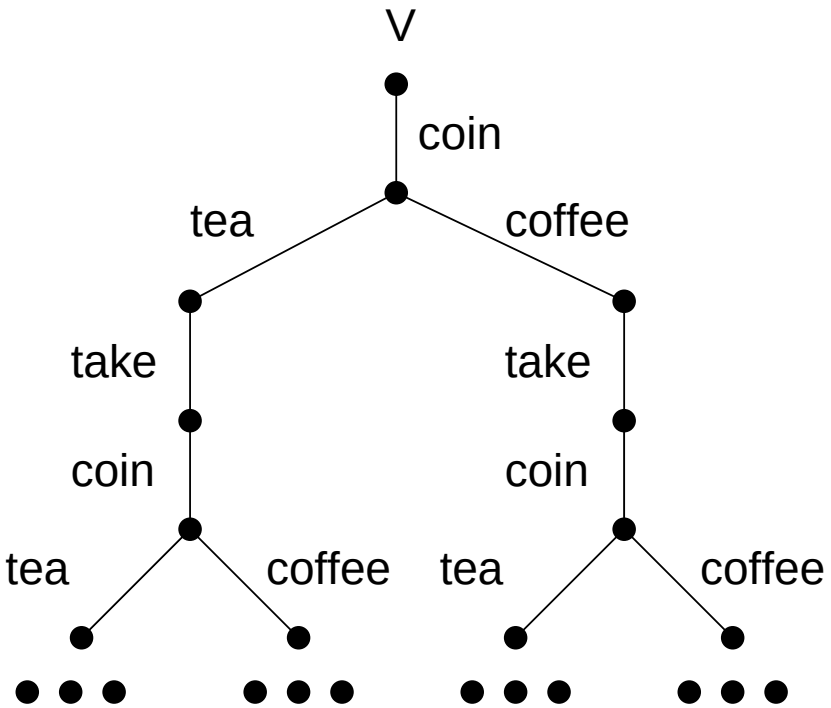
Several *equivalence relations* have been defined for LTS's, along with techniques to prove if such relations are satisfied.

It is then possible to verify if two specifications (by process algebra expressions o state machines) define the same behavior.

In particular, it is possible to prove (at least in principle), that a design model is equivalent to an analysis model.

It is also possible to verify a *refinement* relation, i.e., whether a model satisfies a superset of the requirements of another model.

Example: Vending Machine (1)

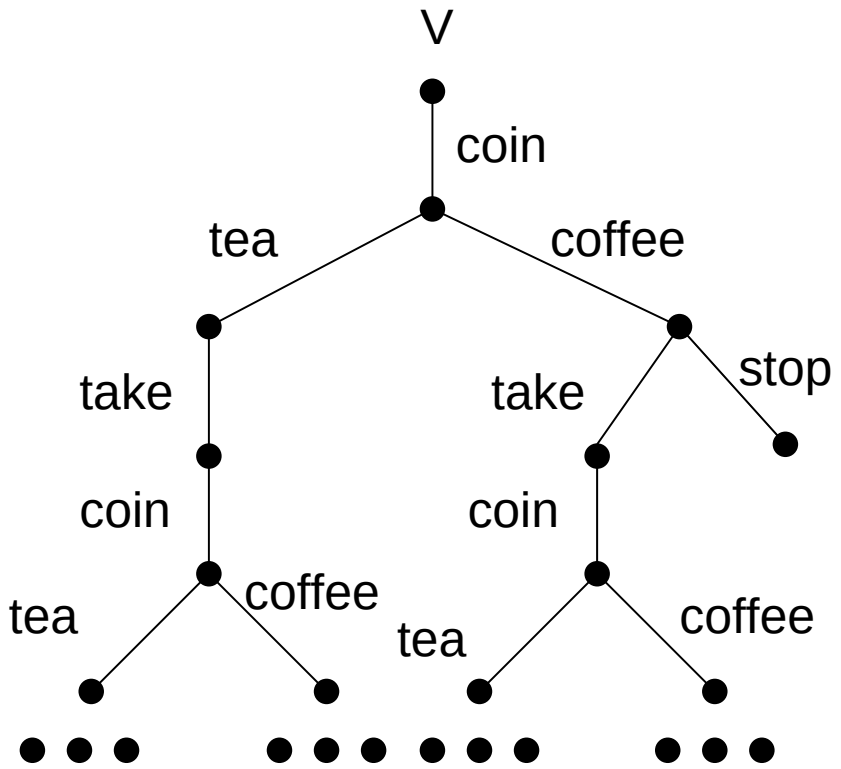


A vending machine V accepts a coin, then it delivers tea or coffee, then, after the drink has been taken by the customer, returns to the initial state.

In a CCS-like [3] process algebra, this can be described as:

$$V = \mathrm{coin}.(\mathrm{tea}.\mathrm{take}.V + \mathrm{coffee}.\mathrm{take}.V)$$

Example: Vending Machine (2)



A broken vending machine V, that may go out of service if coffee is selected.

$$V = \mathrm{coin}.(\mathrm{tea}.\mathrm{take}.V + \mathrm{coffee}.(\mathrm{take}.V + \mathbf{stop}))$$

Expressing Properties: Temporal Logics

- To each state we may associate *state variables* representing physical quantities or logical conditions.
- Properties of the system *in a given state* are expressed with formulas of ordinary logic (*propositional logic* or *predicate logic*).
- *Temporal logics* express properties related to the evolution of the system in time or, equivalently, to the states it can reach.
- E.g., we may say that "*The coolant temperature will never exceed a given limit*", or "*if signals A and B are activated, signal C will eventually be activated*".
- Temporal logics use operators that relate the truth values of formulas to periods or instants of time, e.g.:
  - $\Box\mathcal{F}$ means that $\mathcal{F}$ will be true from now on.
  - $\Diamond\mathcal{F}$ means that $\mathcal{F}$ will eventually become true.

Proving Properties: Model Checking

Model Checking is a common technique used in the automatic verification of temporal logic properties over systems specified by an LTS.

Model checking consists in exploring the LTS (augmented with state variables) and evaluating temporal logic formulas at each node.

If a formula is not satisfied, the model checker tool can usually produce a *counterexample*, e.g., a path through the LTS leading to a state that violates the formula.

Example: A Stepwise Shutdown Logic (1)

This example is taken from [5] (K. Björkman et al., *Verification of Safety Logic Design by Model Checking*, NPIC&HMIT 2009), with some adaptations.

A *Stepwise Shutdown Logic* (SSL) responds to disturbances in plant variables by applying a corrective action for a limited time interval, and then checking if the disturbed variables have returned within normal limits.

A *manual trip* signal may interrupt the cycle and force the corrective action to a complete shutdown.

In this case study, a SSL design is modeled as a LTS, and then its correctness is verified by means of the Uppaal [6] model checking tool.

The tool is based on a behavior specification language called *Timed Automata* [7], and a temporal logic called *Timed Computational Tree Logic* (TCTL) [8].

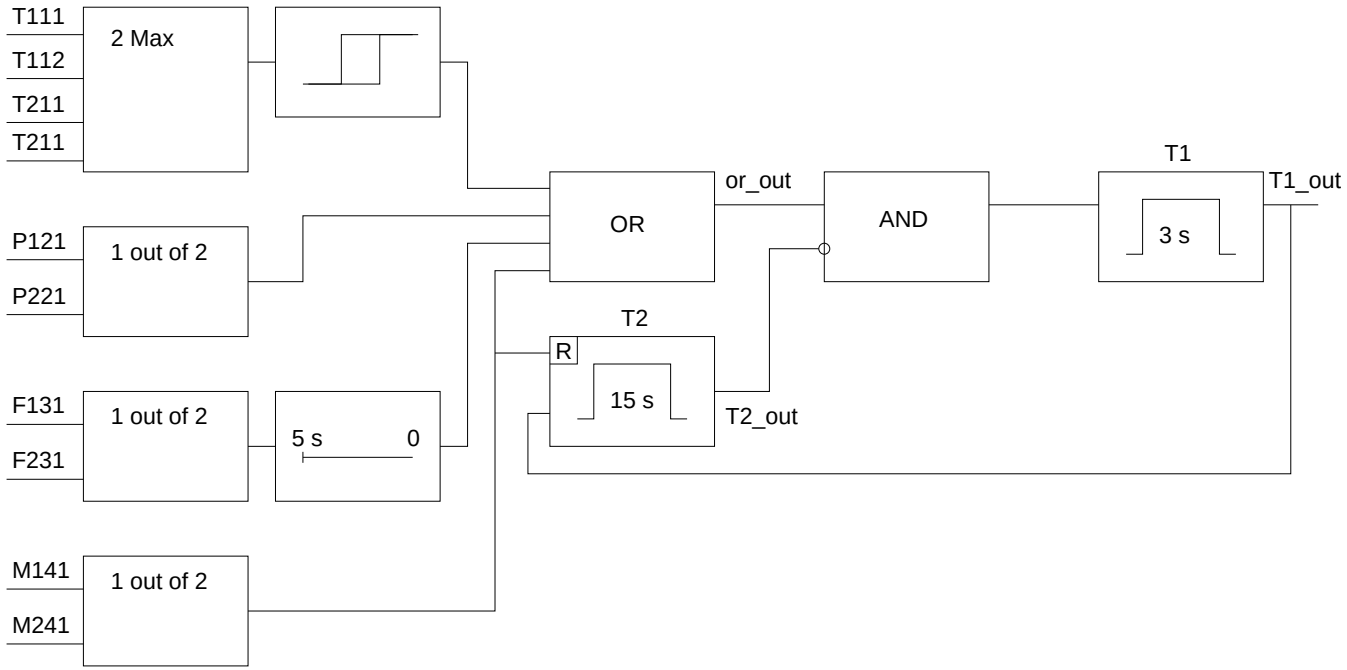
Timed Automata

- *Timed Automata* are finite state machines extended with *clocks* that model the passing of time.
- A transition may be triggered by an *action*, possibly under a constraint (called a *guard*) on one or more clock values.
- A transition may reset clocks.
- The language used in the Uppaal tool introduces several other features, including integer variables and *channels* that connect automata to form networks.

Timed Computational Tree Logic

- *State formulae* are logical conditions on clocks and integer variables associated with single states.
- *Path formulae* are logical conditions on clocks and integer variables, quantified wrt time and paths in the LTS associated with an automaton or a network:
  - $E\Diamond\phi$: there exists a path leading to a state where ϕ holds (*reachability*).
  - $E\Box\phi$: there exists a maximal path whose states all satisfy ϕ (*safety*).
  - $A\Box\phi$: all reachable states satisfy ϕ (*safety*).
  - $A\Diamond\phi$: for all paths, ϕ will eventually hold (*liveness*).
  - $\psi \rightsquigarrow \phi$: shorthand for $A\Box(\psi \Rightarrow A\Diamond\phi)$, i.e., for all paths, if ψ holds, then ϕ will eventually hold (*liveness*).

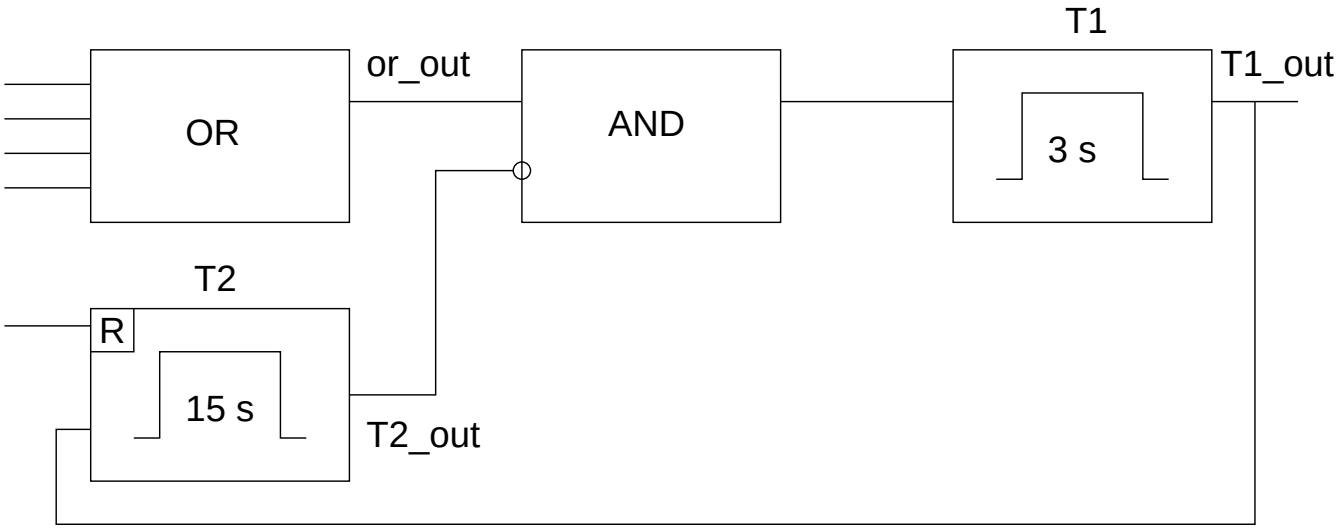
Example: A Stepwise Shutdown Logic (2)



T: temperature
P: pressure
F: water flow
M: manual trip

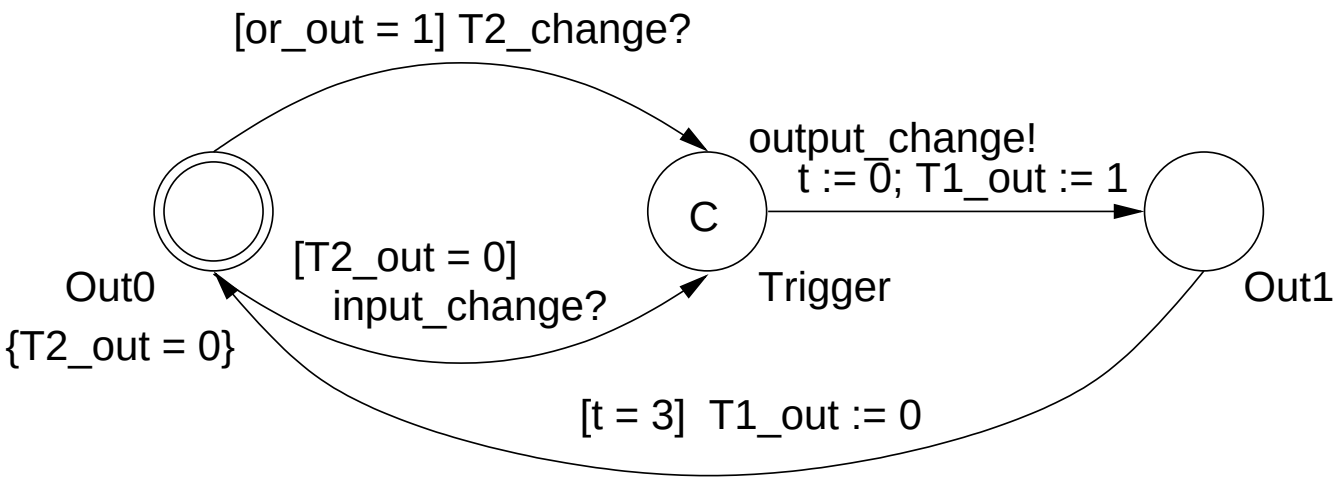
(adapted from K. Björkman et al., *Verification of Safety Logic Design by Model Checking*, NPIC&HMIT 2009)

Example: A Stepwise Shutdown Logic (3)



A close-up view of the control logic.

Example: A Stepwise Shutdown Logic (4)



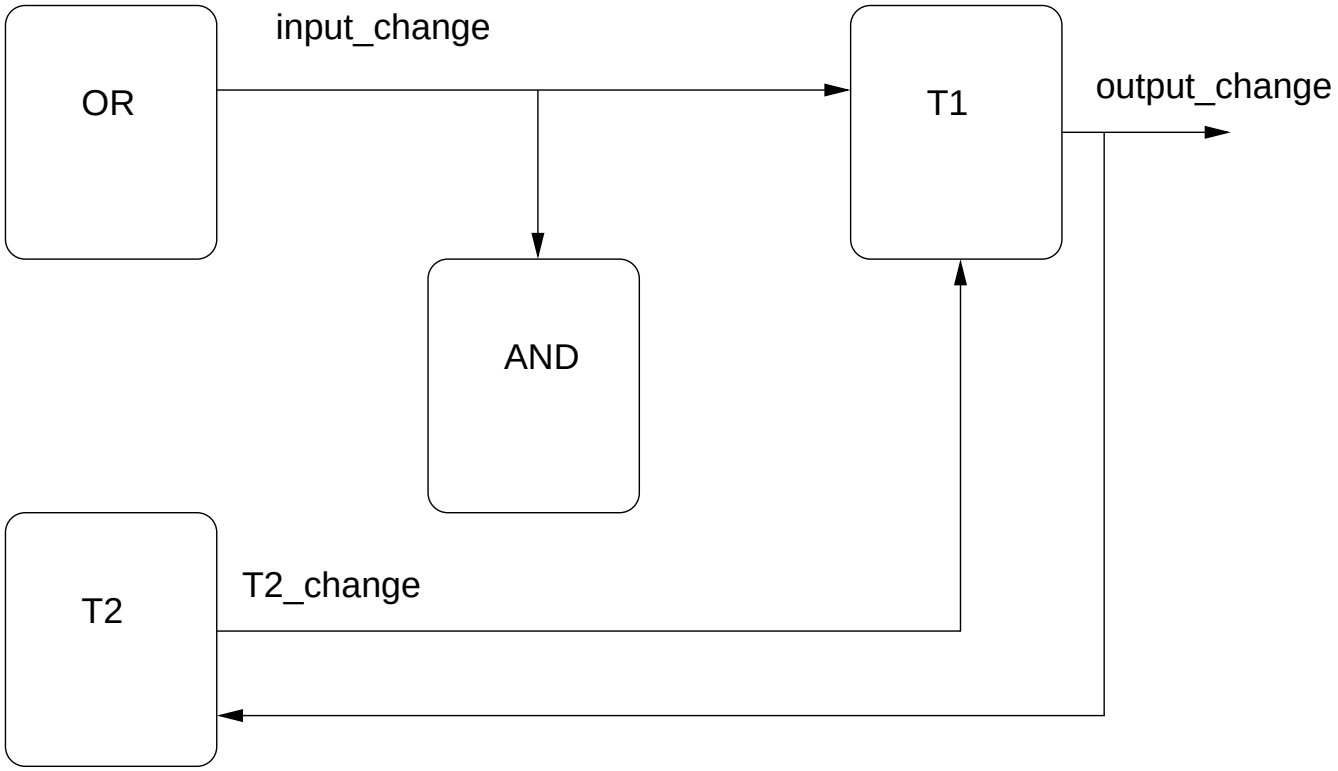
The timed automaton for the T1 timer.

Out0, Trigger, Out1: *states*; or_out, T1_out, T2_out: *state variables*; input_change, T2_change: *input events*; output_change: *output event*. t: *clock variable*;

T1 is triggered if either there is a rising edge on the or_out wire (input_change event) and T2_out is zero, or there is a falling edge on the T2_out wire (T2_change event) and or_out is one. Then T1 enters Out1, T1_out is set to 1 and clock t starts. After 3 s T1 goes back to Out0.

NOTE: Notation and terminology are slightly different from those of the original paper.

Example: A Stepwise Shutdown Logic (5)



The network of timed automata for the SSL.

Automata are connected by *channels* that propagate events.

NOTE that channels do not necessarily correspond to physical connections.

Only the channels related to T1 are shown.

Example: A Stepwise Shutdown Logic (6)

The model was checked against these properties:

1. If at least two of the temperature measurements are over the limit, then *eventually* the output of the system becomes 1.
2. If at least one of the temperature measurements are over the limit, then *eventually* the output of the system becomes 1.
3. If at least one of the two water inflow signals have been on at least five seconds, then *eventually* the output of the system becomes 1.
4. If at least one of the two manual trigger signals is on, then *eventually* the output of the system becomes 1.

E.g., Property 2, can be expressed by this TCTL formula:

$$A\Box(((P121 \vee P221) \wedge \mathit{T2\_out} = 0) \Rightarrow A\Diamond \mathit{T1.Out1})$$

where *T1.Out1* means “T1 is in state Out1”.

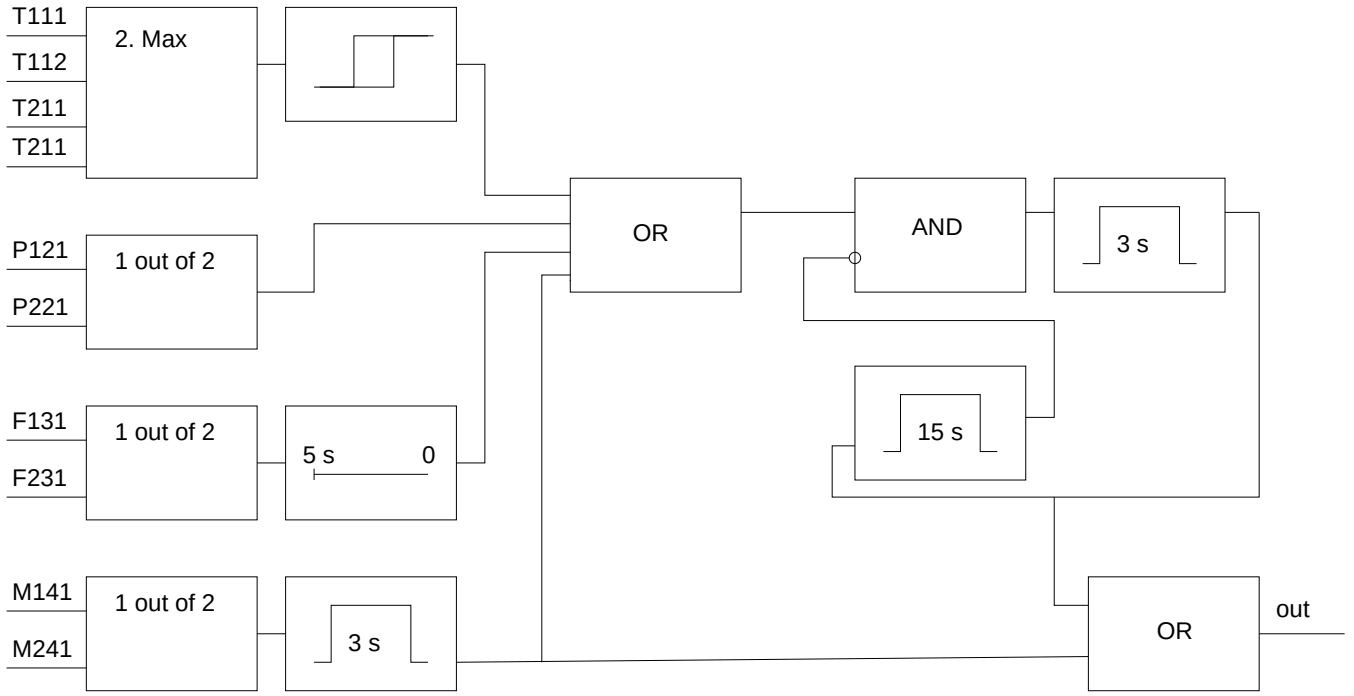
Example: A Stepwise Shutdown Logic (7)

Automatic verification found that all four properties were violated.

The counterexamples provided by the model checker tool enabled designers to find the error: If a manual trip button is pushed during the 3 second time pulse, then the 15 second block is reset and the 3 second block will never receive a rising edge again.

A different design was proved to be correct.

Example: A Stepwise Shutdown Logic (8)



A correct design for the SSL.

THE AUTOMATED DEVELOPMENT APPROACH

Code from Specifications

"The graphic user interface [. . .] permits the process engineer to specify the functional requirements of the I&C system **without any software knowledge**".

Qualification of the Framatome ANP TXS Digital Safety I&C System – Revision to EPRI TR-114017: Compliance with EPRI TR-107330, "Generic Requirements Specification for Qualifying a Commercially Available PLC for Safety-Related Applications in Nuclear Power Plants", EPRI, Palo Alto, CA: 2002. 1003567.

Example: the SPACE Environment (1)

- The *Specification and Coding Environment* (SPACE) [9] is the tool used to develop application software for the *TELEPERM XS Digital Safety I&C System* (TXS).
- Process engineers specify both the software and the hardware architecture of the application.
- The HW architecture is specified by hierarchical structural diagrams and interconnection diagrams.
- The SW (or, more precisely, *functional*) architecture is specified by *Function Block Diagrams*.
- The information contained in the diagrams is stored in a relational database.
- A *code generator* produces the application code and the configuration parameters needed for execution.
- Each function block is implemented by a library module.

Example: the SPACE Environment (2)

- The quality of the development process is based on the reliability of the library modules, of the code generating tool, and of the underlying run-time environment.
- All SW module have a simple structure, they have been developed according to well-established safety guidelines (e.g., no dynamic memory etc.), and have been thoroughly tested.
- The code generator is (probably) rather straightforward, as it must translate FBD's composed out of a small set (ca. 120) of predefined blocks with well defined interfaces and semantics, and strict composition rules that are checked automatically.
- The run-time environment and OS have been developed according to strict safety requirements (e.g., limited use of interrupts, cyclical behavior. . .).
- The SPACE environment provides tools for simulation and automated testing.

References (1)

[1] J. Rumbaugh *et al*., *The Unified Modeling Language Reference Manual – Second Edition*, Addison–Wesley, 2005.

[2] D. Harel, *Statecharts: A Visual Formalism for Complex Systems*, Science of Computer Programming, 8(3):231–274, 1987.

[3] R. Milner, *A Calculus of Communicating Systems*, Lecture Notes in Computer Science, vol. 92, Springer-Verlag, 1980.

[4] L. Aceto *et al*., *Reactive Systems: Modelling, Specification and Verification*, Cambridge University Press, 2007.

[5] K. Björkman *et al*., *Verification of Safety Logic Design by Model Checking*, NPIC&HMIT, 2009.

[6] Uppaal integrated tool environment v. 4.0.6, http://www.uppaal.com (2009).

References (2)

[7] R. Alur and D. L. Dill, *A Theory of timed automata*, Theoretical Computer Science, 126(2):183–235, 1994.

[8] R. Alur *et al*., *Model-checking for real-time systems*, Fifth Annual IEEE Symposium on Logic in Computer Science, pp. 414–425, IEEE, 1990.

[9] K. Waedt and S. Stöcker, SPACE, Specification and Coding Environment. A Toolkit allowing the Graphical Specification of Safety Critical Programs for Automation, ESREL ’93, pp. 825–839, München, 1993.

Thank you

Ačiū

Grazie

Obrigado