Modeling, Simulation, Verification & Code Generation with IAR visualSTATE

Kim G Larsen
Jens Frederik D. Nielsen
Henrik Schiøler
Arne Skou
Andrzej Wąsowski

http://www.mini.pw.edu.pl/~wasowski/

21 November 2003

Outline

- Introducing the modeling language (air conditioner example).
- Tool demo (modeling, simulation, verification and code generation).
- Discussion of generated code.
- Usage contexts (specialization, validation, monitored execution).

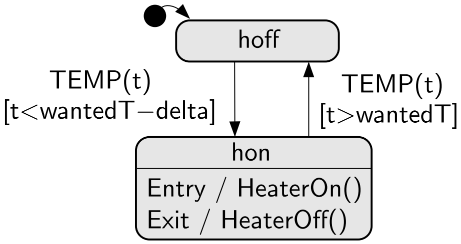
Outline

- **Introducing the modeling language** (air conditioner example).
- Tool demo (modeling, simulation, verification and code generation).
- Discussion of generated code.
- Usage contexts (specialization, validation, monitored execution).

Trivialized Air Conditioning System

- Components:
  - heater
  - cooler
  - fan
  - user interface
  - goal temperature display
  - current fan speed display
- User can set a goal temperature in the room.
- System uses the heater and cooler to achieve the temperature.
- Efficiency of conditioning is controlled by the speed of fan.
  - manual mode: user controls the speed
  - automatic: a fixed, factory predefined speed
  - fast: fan at maximum speed.

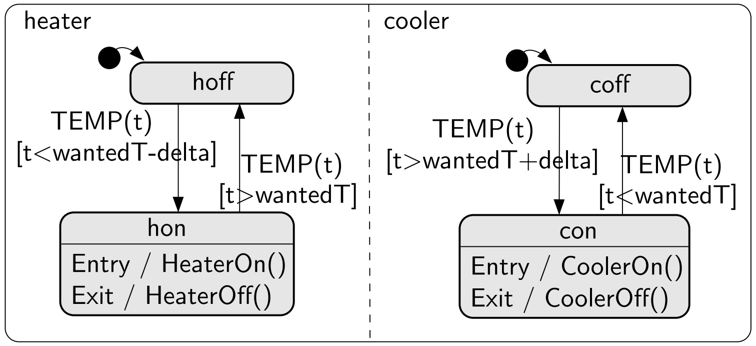
Heater



- Heater is initially in *hoff*.
- A periodic process supplies TEMP(t) regularly.
- Value t is the current temperature returned by the sensor.
- Variable int *wantedT* stores current goal temperature.
- Constant int *delta* gives the acceptable error.
- Heater is activated (*HeaterOn*), whenever *hon* is entered.
- Heater is deactivated (*HeaterOff*) on exit from *hon*.

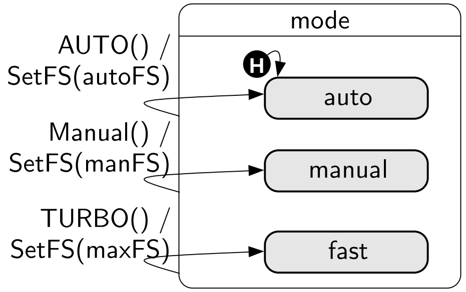
Heater & Cooler

Cooler and heater are analogous and independent.



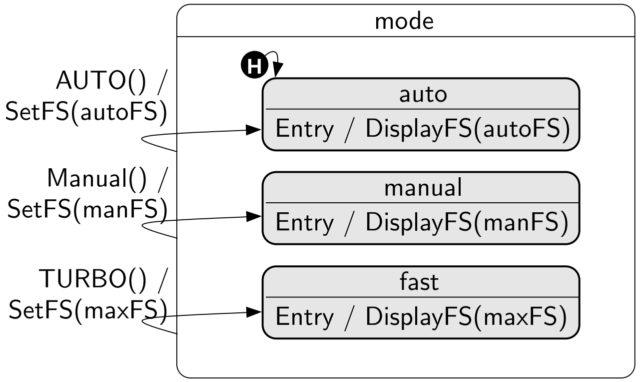
Compose them together.

User Interface



- Buttons: AUTO, TURBO, TINC, TDEC, FSINC, FSDEC.
- Button group: Manual = { FSINC, FSDEC }.
- Variable manFS stores fan speed manually adjusted by user.
- Constant autoFS specifies factory-set automatic mode speed.
- Constant maxFS specifies maximum possible fan speed.
- History state – remember value across executions.

User Interface (II)



Interface has got 2 displays: current fan speed and goal temperature. *DisplayFS* updates the value shown for fan speed. Generally, state actions help to guarantee state invariants.

User Interface (III)

- Internal rules for adjusting the goal temperature and fan speed:

TINC [wantedT<maxT] / [wantedT=wantedT+1] DisplayT(wantedT)

TDEC [wantedT>minT] / [wantedT=wantedT-1] DisplayT(wantedT)

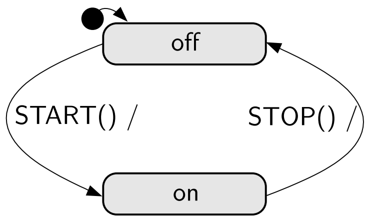
FSINC [manFS<maxFS] / [manFS=manFS+1]

FSDEC [manFS<minFS] / [manFS=manFS-1]

- These rules use variables instead of states.
- As with normal transitions they fire, whenever guards are satisfied.
- They should be active whenever the user interface is active.
- Variable *wantedT* stores the goal temperature set by user.
- Variable *manFS* stores the desired fan speed.
- Maximum&minimum goal temperature: constants *maxT*,*minT*.
- Maximum&minimum fan speed: constants *maxFS*,*minFS*.

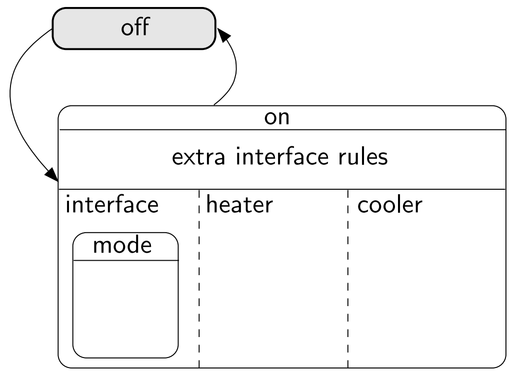
Top Level

START and STOP buttons turn the machinery on and off.



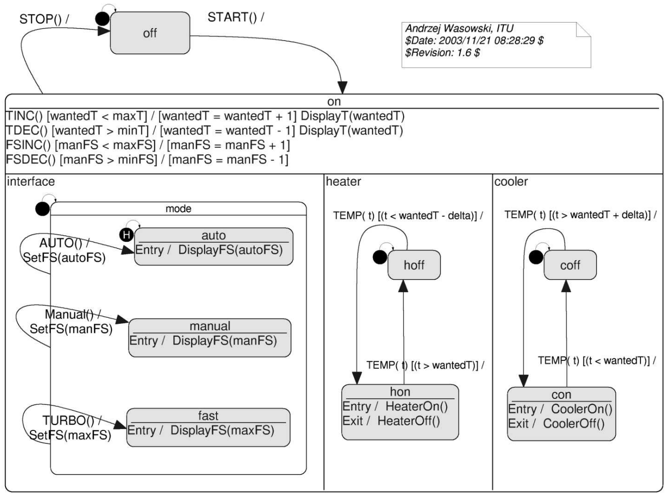
Some entry and exit actions should be added.

System Overview



Complete Model of Air Conditioner

The complete visualSTATE model (tool printout):



Outline

- Introducing the modeling language (air conditioner example).
- **Tool demo** (modeling, simulation, verification, code generation).
- Discussion of generated code.
- Usage contexts (specialization, validation, monitored execution).

IAR visualSTATE Demo

- Designer
- Simulator
- Verifier
- Code Generator

visualSTATE model checker

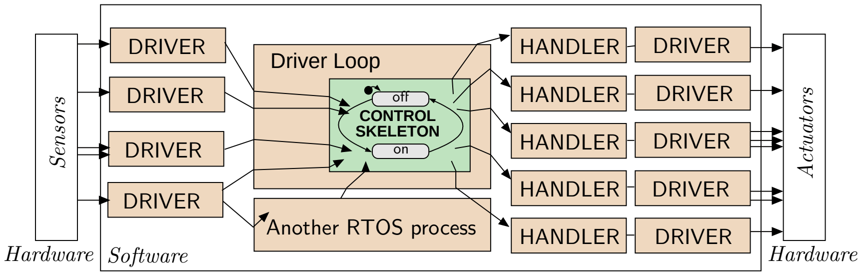
Model checker automatically verifies if following hold in the model:

- No unused components [states, variables]
- No unreachable guards. It must be possible to enable all of the guards in the system. This means that there must exist a reachable state for each guard g that enables this guard. Unreachable guards mean dead code (dead transitions).
- No conflicting transitions.
- No deadlocks.
- No illegal operations. Arithmetic operations should be checked for overflow and illegal operations such as division by zero.
- No divergent behavior. If the signal queue is used then the macrostep should always be finite.
- No overflow of the signal queue.

Outline

- Introducing the modeling language (air conditioner example).
- Tool demo (modeling, simulation, verification, code generation).
- **Discussion of generated code.**
- Usage contexts (specialization, validation, monitored execution).

Application Components



- Generated control skeleton (green).
- Brown parts hand coded, but fortunately small and easier.
- Sometimes multiple processes are avoided in favour of the loop.
- In some cases it is even possible to give up the RTOS entirely.
- Discuss the cost of automatic generation of the skeleton.

Executable Size [Control Algorithm]

The table presents executable sizes for the air conditioner model (only control code, no RTOS, no action functions, interfaces to sensors, etc):

platform	compiler	optimizations	cod. gen.	size [b]
i386/Linux	gcc 3.2	-Os + strip	IAR VS 4.3	4 428
i386/Linux	gcc 3.2	-Os + strip	SCOPE	3 732
h8300	gcc 2.95	-O2 + strip	IAR VS 4.3	8 528
h8300	gcc 2.95	-O2 + strip	SCOPE	7 922
h8300	gcc 3.3	-O2 + strip	IAR VS 4.3	**2 388**
h8300	gcc 3.3	-O2 + strip	SCOPE	**1 822**

The gcc 3.3 reported is an experimental version and executables were not tested. You know better if commercial compilers can be expected to generate more efficiently.

Memory Consumption ctd.

- Following executable sizes given for gcc 3.3 on H8/300:
- The visualSTATE kernel (compiled with dummy model) takes 1.5k
- Complex model of coffee machine (200 transitions) is below 7k
- RAM usage in SCOPE [quick generous estimate, assuming 8bit word, 32bit addressing]

	[bytes]
current event (global)	3
state representation (global)	8
stack	30
model variables (global)	4
TOTAL [bytes]	41+4

- More expensive if signal communication is used.
- VisualSTATE has similar performance.
- If this is not sufficient we can try targeting assembler directly.

Code Excerpts [SCOPE]

Most of the code take up read-only tables:

```
/* and-state projection of hierarchy */
const anatomycell anatomy[10] =  {
    /* 0 */ STMRK, MCHN 3, MCHN 3, MCHN 0, MCHN 2,
    /* 5 */ MCHN 2, MCHN 1, MCHN 1, MCHN 1, MCHN 0,
};

/* transitions array */
const transcell trans[TRANS_MAX] =  {
    /*  0 */ PCNC(2) 1, 0, STATE 3, ACGD(2) 4, 1,
    /*  5 */ STATE 8, STMRK, PCNC(2) 1, 0, STATE 3,
    /* 10 */ ACGD(2) 6, 1, STATE 7, STMRK, PCNC(2) 1,
    /* 15 */ 0, STATE 3, ACGD(2) 6, 1, STATE 7,
    /* 20 */ STMRK, PCNC(2) 1, 0, STATE 9, ACGD(2) 1,
    /* 25 */ 1, STATE 3, STATE 2, STATE 5, STMRK,
    /* 30 */ PCNC(2) 2, 0, STATE 3, STATE 1, ACGD(2) 3,
    ...
```

Code Excerpts [SCOPE] (II)

```
/* guards dispatcher */
int eval ( const guardref g ) {
  switch (g) {
    case 1: return ((CurrEvent.fields._E_TEMP.f0)
                   <(wantedT));
    case 2: return ((CurrEvent.fields._E_TEMP.f0)
                   >((wantedT)+(delta)));
    case 3: return ((CurrEvent.fields._E_TEMP.f0)
                   >(wantedT));
    case 4: return ((CurrEvent.fields._E_TEMP.f0)
                   <((wantedT)-(delta)));
    case 5: return (1);
    ...
  }
  return 1;
}
```

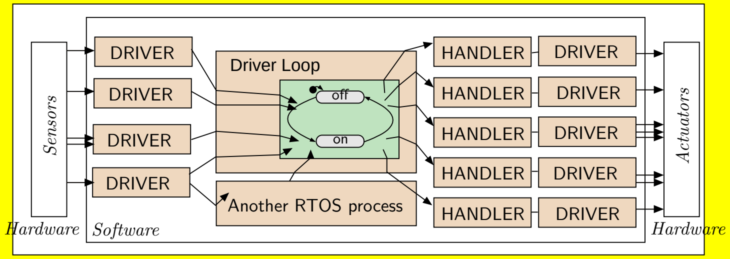

Code Excerpts [SCOPE] (III)

```
/* Main loop */

int main (void)
{
    st_init();       // initialize the system
    while (1) {

        ...          // compute the next event in i

        CurrEvent.tag = i;
        macrostep(); // call the VS kernel
    }
    return 0;
}
```


Outline

- Introducing the modeling language (air conditioner example).
- Tool demo (modeling, simulation, verification, code generation).
- Discussion of generated code.
- **Usage contexts** (specialization, validation, monitored execution).

Usage Contexts [in progress]



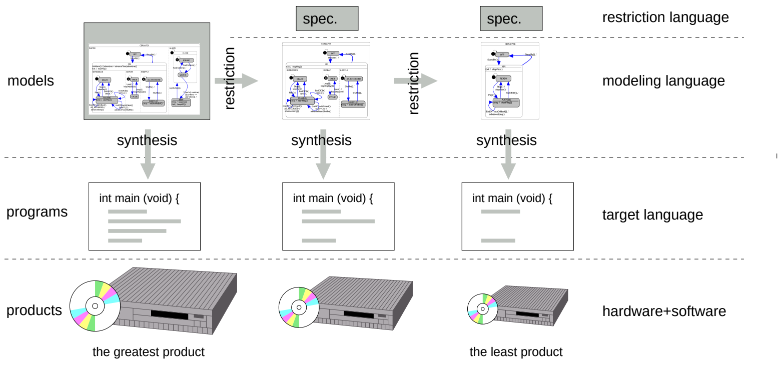
Execution Context (reality...)

- All systems are executed in contexts which are limited in some ways.
- Context limits possible interactions of the embedded systems.
- Modeling contexts adds another level of value to the development process.

Usage Contexts [in progress] (II)

- Test case generation is improved. Resources are not wasted on executing completely unrealistic cases. Number of false negatives is reduced.
- Number of false negatives is reduced for verification too.
- Monitored execution: alarms can be generated at runtime whenever system (or environment) violates the assumed contract.
- Specialization: possibility of generating code specifically optimized for given user context. Supports architecture of product line based on single source code.
- Air conditioner can be specialized to a heater, a cooler, a device with less than 3 modes, a device without display, etc.
- A single model for all those.
- A small context specification for each of those.
- Automatic instantiation of general model in specific context
- We need your input and examples on what contexts are.

Fast Generation of User Variants



Fast Generation of User Variants (II)

```
restriction WithoutAlarm {
    impossible SetAlarm();
    impossible SwitchAlarm();
};
WithoutAlarm CDPLAYER;

restriction Least restricts WithoutAlarm {
      impossible Loop();
      impossible Shuffle();
};

Least CDPLAYER;
```

- Hierarchies of contexts can be build.
- More dynamic context properties can be expressed with automata. Especially useful for test generation and automatic verification.
- Keen to see what kind of properties are needed.

Summary

- Introducing the modeling language (air conditioner example).
- Tool demo (modeling, simulation, verification, code generation).
- Discussion of generated code.
- Usage contexts (specialization, validation, monitored execution).

Thank you for Your attention.