Automatic Generation of Program Families by Model Restrictions

Andrzej Wąsowski
Department of Innovation
IT University of Copenhagen

wasowski@itu.dk
http://www.mini.pw.edu.pl/˜wasowski/scope

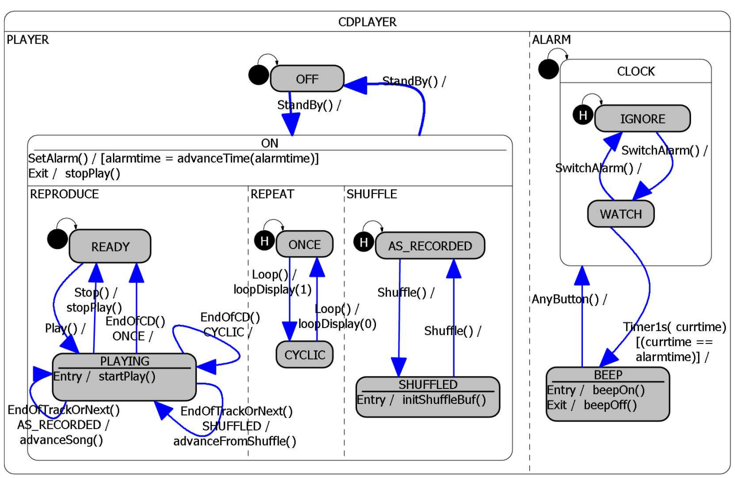
1 September, SPLC 2004, Boston, MA

IT University of Copenhagen

Smaller Means Cheaper

- Specialized variant can often be fit into cheaper hardware.
- Memory is cheap? Industry accounting does not confirm this.
- Save 1$ per item replacing an 8K with 16K RAM chip.
- With production goal being 400 000 items a year for about 8 years this money can't be ignored.
- Will memory become cheaper? Probably yes
  - need for small memory software will not disappear
  - everybody wants smaller and more portable devices with lower power consumption
  - which yet can do more than today's state of the art.

Erik's CD Player



Outline

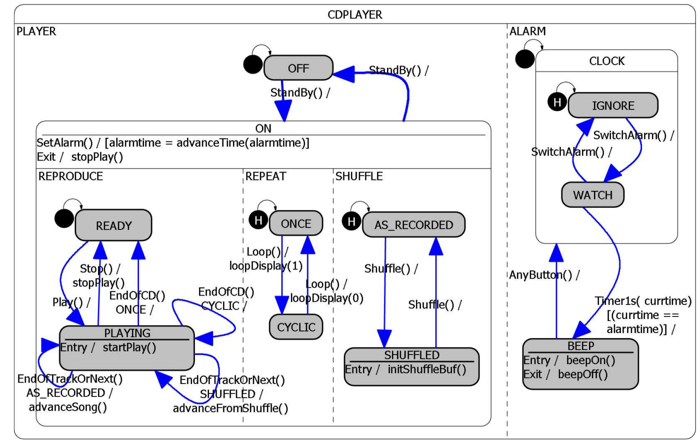
- **Model restriction and environment specifications**
- **Syntax and semantics of Restriction Language**
- **Implementation relation, behavioral guarantees**
- **Current work**
- **Summary**

Outline

- **Model restriction and environment specifications**
- Syntax and semantics of Restriction Language
- Implementation relation, behavioral guarantees
- Current work
- Summary

A CD Player without Alarm Clock

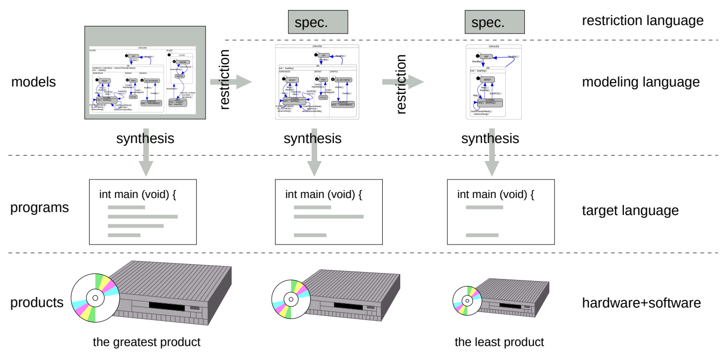
Execution environment determines a variant of the CD player



No alarm means no *SetAlarm* and no *SwitchAlarm* events

Product Lines Architecture

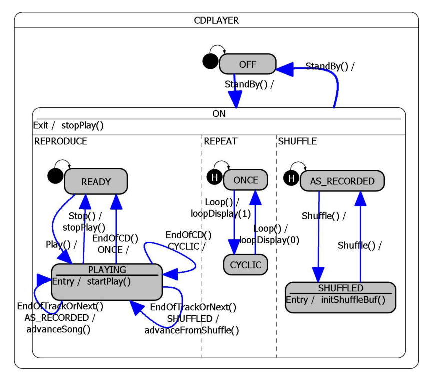
and Specialization



One would like:

- A notion of implementation between products, safety guarantees
- A hierarchy on specifications supporting stepwise development

A CD Player without Alarm Clock (II)

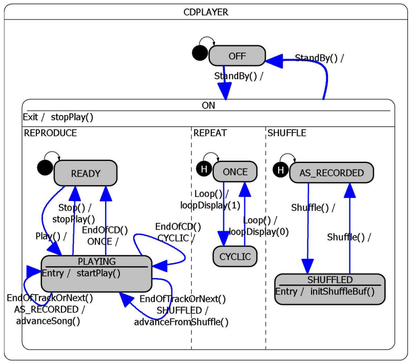



```
restriction WithoutAlarm {
    impossible SetAlarm();
    impossible SwitchAlarm();
};
WithoutAlarm CDPLAYER;
```

- Alarm-related inputs impossible
- Specialized using data-flow and reachability analysis
- Dead code elimination easier due to explicit control flow

CD Player with no Alarm Clock,

no Shuffle and no Continuous Play



Shuffle and *Loop* events are impossible.

A Family of Embedded Systems

- Family of products vs Family of programs
- Structured family $(P, \preceq)$ of n embedded programs, where $\preceq$ is a restriction relation:

$$p_1 \preceq p_2 \triangleq p_1 \text{ may be obtained from } p_2$$
$$\text{by removing some functionality}$$

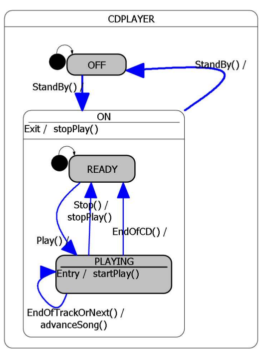
- In linear case (product line):

$$p_1 \preceq p_2 \preceq \ldots \preceq p_n$$

- In general a partial order with a single greatest element, i.e. the program which is able to do ”everything” specific to the family.
- A restriction hierarchy dual to extension hierarchy

CD Player with no Alarm Clock,

no Shuffle and no Continuous Play (II)



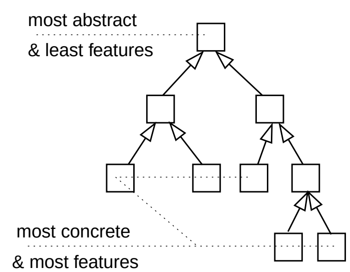

```
restriction Least
              restricts WithoutAlarm {
    impossible Loop();
    impossible Shuffle();
};

Least CDPLAYER;
```

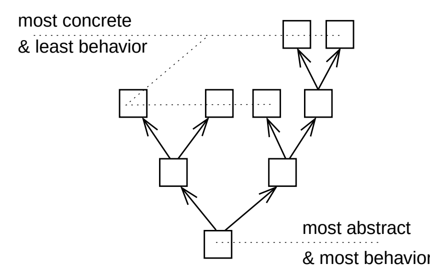
Obtained by further restriction of the version with no alarm clock.

Restriction vs Extension

OO INHERITANCE HIERARCHY



RESTRICTION HIERARCHY



A ⇾ B : A extends B

A → B : B is a restriction of A

Characteristics

- Well suited for families of relatively simple devices like home appliances, Hi-Fi equipment, toys (!), etc.
- Not that useful for highly configurable complex designs (satellite)
- Lightweight – runtime efficiency only depends on quality of specializer. No runtime overhead (contrary to OO-languages).
- One difficult implementation — common model. Numerous small restriction specifications in RL.
- Prototyping is fast. Easily fine tune resources and functionality.
- Enjoy behavioral inheritance.

Outline

- Model restriction and environment specifications
- **Syntax and semantics of Restriction Language**
- Implementation relation, behavioral guarantees
- Current work
- Summary

Specifying Restrictions

- Easy for control-oriented systems: restrict inputs and outputs
- Restriction is a description of an environment
- Restriction Language (RL) a custom language for writing restrictions:

```
restriction NoBeep {
     impossible e;         // impossibility constraint
     const int v=1;        // value constraint
     const int f(int)=4;   // (func.) value constraint
     dead int v;           // liveness constraint
     pure void f();        // purity constraint
}

restriction Name restricts ancest1, ..., ancestn
{ ... };

InterfaceName ModelName;
```

Semantics of RL

Assume the semantics of model m is given by CSP-like traces.

$$RL[\![\cdot]\!]_m : \mathscr{P}(\mathit{traces}()) \longrightarrow \mathscr{P}(\mathit{traces}())$$

$$RL[\![\texttt{impossible e}]\!]_m = \lambda S.\, \{ t \mid t \in S \wedge \neg(\langle e \rangle \textbf{ in } t) \}$$

$$RL[\![\texttt{const T v = k}]\!]_m = \lambda S.\, \{ t \mid t \in S \wedge \forall \langle v.l \rangle \textbf{ in } t.\, l = k \}$$

$$RL[\![\texttt{const T f() = k}]\!]_m = \ldots$$

$$RL[\![\texttt{dead T v}]\!]_m = \lambda S.\, \{ t \upharpoonright (\alpha m \setminus \{ v.l \mid \forall l \in \mathit{values}(T) \}) \mid t \in S \}$$

$$RL[\![\texttt{pure T f()}]\!]_m = \ldots$$

$$RL[\![c_1 ; c_2]\!]_m = RL[\![c_2]\!]_m \cdot RL[\![c_1]\!]_m$$

Outline

- Model restriction and environment specifications
- Syntax and semantics of Restriction Language
- **Implementation relation, behavioral guarantees**
- Current work
- Summary

Implementation Relation

Intuitively m_1 implements m_2 if it can be executed with the same trace, perhaps extended by some m_2-specific events.

[Implementation]

Two models m_1 and m_2 such that the set of inputs accepted by m_1 is the subset of the inputs accepted by m_2 ($\alpha m_1 \subseteq \alpha m_2$).

Then m_1 implements m_2, written $m_1 \lesssim m_2$, iff

$$\forall t_1 \in traces(m_1).\ \exists t_2 \in traces(m_2).\ t_1 = t_2 \upharpoonright \alpha m_1.$$

Implementation Relation (II)

An execution trace of CD player without Alarm/Shuffle/Loop:

$$t = \langle \textit{StandBy, Play, startPlay.(), Stop} \rangle$$

It is included in the following trace of the most general CD player:

$$t' = \langle \textit{StandBy, SwitchAlarm, SwitchAlarm, Play, startPlay.(),Stop} \rangle$$

[Soundness of Restriction]

If m_1 has been obtained from m_2 by restriction of sensors and actuators then m_1 implements m_2. More precisely:

$$m_1 \preceq m_2 \Rightarrow m_1 \lesssim m_2.$$

This relies on the fact that the restriction is sound (accurately describes the environment) and only performs semantics preserving optimizations.

Current Work

Dynamic Environments

- Restriction specifications presented above are special cases of state-independent properties of dynamic environments.
- Already small case studies show that static constraints are not sufficient:
  - how to specify a CD player which only has the continues play mode?
  - simply ignoring output is often too strict. One wants to substitute some outputs for others.
- Formulated a theory of dynamic environments with color-blind properties. i.e. anvironments which can produce only some input traces and are tolerant to some mutations in program's outputs.
- Working on implementation of dynamic optimizer, which will be more "creative" than simple restrictions.

Summary

- Execution environments define product variants
- Model restriction can be used to generate variants of control algorithms for embedded systems.
- Improves code reuse and maintainability
- Preserves behavioral inheritance (safety)
- Can be extended to behavioral specifications of environments