Bossa Nova: Introducing Modularity into the Bossa Domain-Specific Language

Julia L. Lawall, DIKU, University of Copenhagen

Hervé Duchesne and Gilles Muller,
OBASCO Group, École des Mines de Nantes-INRIA, LINA

Anne-Françoise Le Meur, Jacquard Group, LIFL/INRIA, Lille

September 29, 2005

Domain-Specific Languages (DSLs)

DSL: A language dedicated to a particular domain.

- Captures a family of programs.
- Provides high-level domain-specific abstractions that
  - Simplify programming.
  - Enable verifications, optimizations.

Useful when:

- Programming within the program family is often needed.
- Programming within the program family requires highly specialized knowledge.

Examples: lex, yacc, SQL, languages for graphics, Web programming, etc.

How to create a DSL?

- Analyze the domain to identify a program family.
- Design the language.
- Implement the language.

Problem: unanticipated program subfamilies

Program subfamily may raise unanticipated needs.

Embedded language approach

- DSL implemented within a fully featured host language.
- Extra features (functions, modules, etc.) available if new needs arise.
- Problem: These features may not match the domain.

Direct approach

- Some work required to implement new features.
- But, new features can be tailored to language design goals.

This talk

- Our example: The Bossa DSL for OS kernel process scheduling.
- Evolution of Bossa to meet unanticipated needs:
  - Modules and aspects.
- Evaluation:
  - Benefits of the new features.
  - Comparison to other approaches.
- Conclusions and future work.

Bossa: a DSL for OS kernel process scheduling

Process scheduling: How an OS selects a process for the CPU.

- Many scheduling policies (round-robin, rate monotonic, etc.).
- No policy is perfect for all applications.
- Policies form a program family.

Implementing a scheduler requires:

- Understanding the scheduling policy.
- Understanding the target OS.
  - Any error can crash the machine.

$\Rightarrow$ An ideal DSL target ...

- Bossa [EW2002, ASE2003, PEPM2004, GPCE04, HASE05, IFM05]

The process scheduling problem

CPU:

Other processes:

The process scheduling problem

CPU:

Other processes: $\boxed{}$

A process arrives.

The process scheduling problem

CPU:

Other processes:

The process is elected.

The process scheduling problem

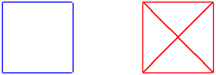
CPU:

Other processes:

Another process arrives.

The process scheduling problem

CPU:

Other processes: 

The red process blocks.

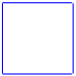
The process scheduling problem

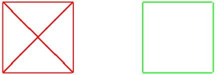
CPU:

Other processes:

The blue process is elected.

The process scheduling problem

CPU: 

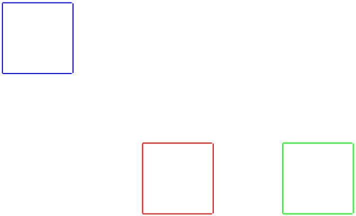
Other processes: 

Another process arrives.

The process scheduling problem

CPU:

Other processes:



The red process unblocks.

The process scheduling problem

CPU:

Other processes:

The blue process blocks.

The process scheduling problem

CPU:

Other processes:

Which process is elected next?

Scheduling concepts

- Process states (running, ready, blocked, etc.).
- Process attributes (quantum, deadline, etc.).
- OS events (blocking, unblocking, etc.).

The Bossa DSL, by example

```
scheduler EDF = {
  states = {
           running
           ready
           blocked
           computation_ended
             terminated
  }
}
```

The Bossa DSL, by example

```
scheduler EDF = {

  states = {
    RUNNING    running             : process;
    READY      ready               : select queue;
    BLOCKED    blocked             : queue;
    BLOCKED    computation_ended   : queue;
    TERMINATED terminated;
  }

}
```

The Bossa DSL, by example

```
scheduler EDF = {
  process = {
    time  period;
    time  wcet;
    time  absolute_deadline;
    timer period_timer;
  }
  states = {
    RUNNING  running            : process;
    READY    ready              : select queue;
    BLOCKED  blocked            : queue;
    BLOCKED  computation_ended  : queue;
    TERMINATED terminated;
  }


}
```

The Bossa DSL, by example

```
scheduler EDF = {
  process = {
    time  period;
    time  wcet;
    time  absolute_deadline;
    timer period_timer;
  }
  states = {
    RUNNING  running            : process;
    READY    ready              : select queue;
    BLOCKED  blocked            : queue;
    BLOCKED  computation_ended  : queue;
    TERMINATED terminated;
  }
  ordering_criteria = { lowest absolute_deadline }

}
```

The Bossa DSL, by example

```
scheduler EDF = {
  process = {
    time  period;
    time  wcet;
    time  absolute_deadline;
    timer period_timer;
  }
  states = {
    RUNNING  running             : process;
    READY    ready               : select queue;
    BLOCKED  blocked             : queue;
    BLOCKED  computation_ended   : queue;
    TERMINATED terminated;
  }
  ordering_criteria = { lowest absolute_deadline }
  handler (event e) { ... }
}
```

Bossa event handlers

```
handler (event e) {
  On block.*

  On unblock.preemptive






  On bossa.schedule



  ...
}
```

Bossa event handlers

```
handler (event e) {
  On block.* { e.target => blocked; }

  On unblock.preemptive {
    e.target => ready;
    if (!empty(running) && e.target > running) {
      running => ready;
    }
  }

  On bossa.schedule {
    select() => running;
  }
  ...
}
```

Verified with respect to a model of OS scheduling-related behavior. [GPCE04]

Using Bossa revealed unanticipated needs

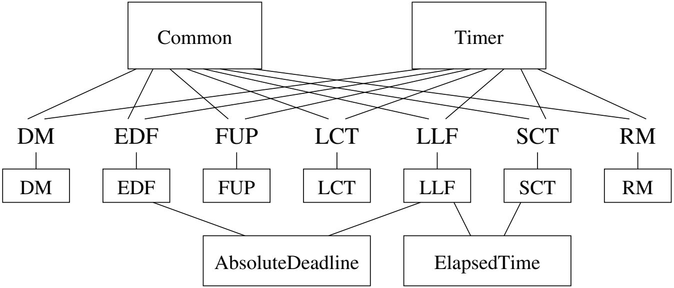
Applications:

- An encyclopedic multi-OS library of scheduling policies.
  - Revealed program sub-families.
- Scheduling policies based on resource usage.
  - Revealed cross-cutting concerns.

Bossa Nova: extending Bossa with modules and aspects.

The need for modules

The subfamily of periodic scheduling policies:



Average size: 123 lines.
100 in common.

Design goals for modules

- Understandability.
- Verifiability.
- Code reuse.

Understandability

Provide a centralized, global view.

```
scheduler EDFSched = {
  states = { RUNNING running : process; READY ready : select queue;
             READY yield : process; BLOCKED period_yield : queue;
             BLOCKED blocked : queue; TERMINATED terminated; }
  modules { EDF(),
            AbsoluteDeadline(),
            Timer (running, ready, period_yield),
            Common (running, ready, yield, blocked, terminated) }
  process { ... }
  ordering_criteria { EDF }
  handler { unblock.timer.period_timer : AbsoluteDeadline, Timer; }
}
```

Verifiability

Enforce fine-grained control over access to module elements.

```
module AbsoluteDeadline() {
  process = { time deadline; time absolute_deadline; ... }
  handler (event e) {
    On unblock.timer.period_timer {
      e.target.absolute_deadline = now() + e.target.deadline; ... } } }

module EDF() {
  process = { requires time absolute_deadline; ... }
  ordering_criteria = { lowest absolute_deadline }
}
```

- Attributes:
  - Defining module can write.
  - Importing modules can only read.
- States can be declared as unshared.

Code reuse

Minimize explicit inter-module references

```
module AbsoluteDeadline() {
  process = { time deadline; time absolute_deadline; ... }
  handler (event e) {
    On unblock.timer.period_timer {
      e.target.absolute_deadline = now() + e.target.deadline;
      next(); } } }

module EDF() {
  process = { requires time absolute_deadline; ... }
  ordering_criteria = { lowest absolute_deadline }
}
```

Connections are made in the scheduler:

```
scheduler EDFSched = {
  process { EDF.absolute_deadline reads AbsoluteDeadline.absolute_deadline, ... }
  handler { unblock.timer.period_timer : AbsoluteDeadline, Timer; }
}
```

Evaluation

Code sharing:

Modules	Common	Timer	AbsoluteDeadline	ElapsedTime
Lines of code	68	47	28	45

		Policy-specific module	Scheduler	Modular	Monolithic
Periodic	DM	23	22	160	109
	EDF	26	34	203	123
	FUP	20	27	162	110
	LCT	9	26	150	106
	LLF	45	39	272	161
	SCT	42	35	237	147
	RM	9	26	150	106
	Family total			503	862

Separation of concerns.

Isolation of OS-specific behavior.

Comparison to other module systems: Understandability

- Our approach: global view in the "scheduler".
- Some systems, eg Units, provide basic blocks and combiners, but combiners can combine combiners.
- Other systems, eg Java, ML, express composition within the basic blocks.

Comparison to other module systems: Verifiability

- Our approach: restrictions on inter-module accesses.
- Const: read-only for everyone.
- Public/private: restricts visibility, not writeability.
- Getter/setter functions: not enforced.

Comparison to other module systems: Code reuse

- Our approach: modules don't name other modules explicitly.
- Some systems address this, eg Units, Mixins, Traits.
- Others use explicit names widely, eg name of a superclass.

Future work

- Modular verification of Bossa Nova code.
- Verification of policy-specific properties.
- Guidelines for DSL design and evolution.
- Further applications and generalizations:
  - Policies for controlling energy usage.
  - Scheduling in OS hierarchies.

Conclusion

- Using a DSL may highlight program subfamilies, which can introduce unanticipated needs.
- Language features to meet these needs should be designed in a domain-specific way, to match language design goals.
- Our examples: modules and aspects in Bossa.
  - Designed according to principles of understandability, verifiability, and code reuse.

More information

- Bossa and Bossa Nova grammars.
- Compiler and verifier for Bossa/Linux.
- Integration of Bossa in Linux 2.4 and 2.6.
- Examples.
- Teaching materials.

http://www.emn.fr/x-info/bossa/