Extensible Programming and Modeling Languages

Ted Kaminski, Yogesh Mali, August Schwerdfeger
and *Eric Van Wyk*

University of Minnesota

September 20, 2012, Lund, Sweden

- Languages are not monolithic.
- But most language tools primarily support monolithic design and implementation.

Extensible Language Frameworks — ableP

- add features to a “host” language — Promela
- new language constructs - their syntax and semantics
  - `select (altitude: 1000 .. 10000);`
  - `select (altitude: 1000 .. 10000 step 100);`
  - `select (altQuality: High, Med, Low);`
  - DTSpin constructs: `timer t; t = 1; expire(t);`
- new semantics of existing constructs
  semantic analysis, translations to new target languages, ...
  - type checking
  - advanced ETCH-style type inference and checking

Various means for extending Promela

- `select (v:  1 ..  10)` added in SPIN version 6.
- DTSPIN features
  - as CPP macros — lightweight
  - or modifying the SPIN implementation — heavyweight
- ETCH, enhanced type checking
  - built their own scanner and parser using SableCC
- ableP — middleweight approach

An example

An altitude switch model that uses

- enhanced `select` statements
- DTSPIN-like constructs
- tabular Boolean expressions (*à la* RSML and SCR)

An instance of ableP parses and analyzes the model, then generates its translation to pure Promela.

```
% java -jar ableP.aviation.jar AltSwitch.xpml

% spin -a AltSwitch.pml
```

Our approach:

- Users choose (independently developed) extensions.
- Tools compose the extensions and Promela host language.
- Distinguish
  1. extension user
     - has no knowledge of language design or implementations
  2. extension developer
     - must know about language design and implementation

1. Tools and formalisms support automatic composition.
2. Modular analyses ensure the composition results in a working translator.

- Value easy composition over expressivity, accept some restrictions
  - on syntax
  - new constructs are translated to "pure" Promela
- ableP "instances" are smart pre-processors

Extending ableP with independently developed extensions

- Extension *user* directs underlying tools to
    - compose chosen extensions and the host language
    - and then create a custom translator for the extended language
- Silver grammar modules define sets of specifications
    - composition is set union, order does not matter
- Consider the Silver specification for this composition.

Developing language extensions

Two primary challenges:

1. composable syntax — enables building a scanner and parser
    - context-aware scanning [GPCE07]
    - modular determinism analysis [PLDI09]
    - Copper

2. composable semantics — analysis and translations
    - attribute grammars with forwarding, collections and higher-order attributes
    - set union of specification components
        - sets of productions, non-terminals, attributes
        - sets of attribute defining equations, on a production
        - sets of equations contributing values to a single attribute
    - modular well-definedness analysis [SLE12]
    - monolithic termination analysis [SLE12]
    - Silver

Context aware scanning

- Scanner recognizes only tokens valid for current "context"
- keeps embedded sub-languages, in a sense, separate
- Consider:

  -
    ```
    chan in, out;
    for i in a { a[i] = i*i ; }
    ```
- Two terminal symbols that match "`in`".
  - `terminal IN 'in' ;`

  -
    ```
    terminal ID /[a-zA-Z_][a-zA-Z_0-9]*/
                submits to {promela_kwd };
    ```
  - `terminal FOR 'for' lexer classes {promela_kwd };`

Allows parsing of embedded C code

```
c_decl {
  typedef struct Coord {
    int x, y; } Coord;      }

c_state "Coord pt" "Global"  /* goes in state vector */
int z = 3;                   /* standard global decl */

active proctype example()
{ c_code { now.pt.x = now.pt.y = 0; };

  do :: c_expr { now.pt.x == now.pt.y }
          -> c_code { now.pt.y++; }
     :: else -> break
  od;

  c_code { printf("values %d: %d, %d,%d\n",
              Pexample->_pid, now.z, now.pt.x, now.pt.y);
  };
}
```

Semantics for host language assignment constructs

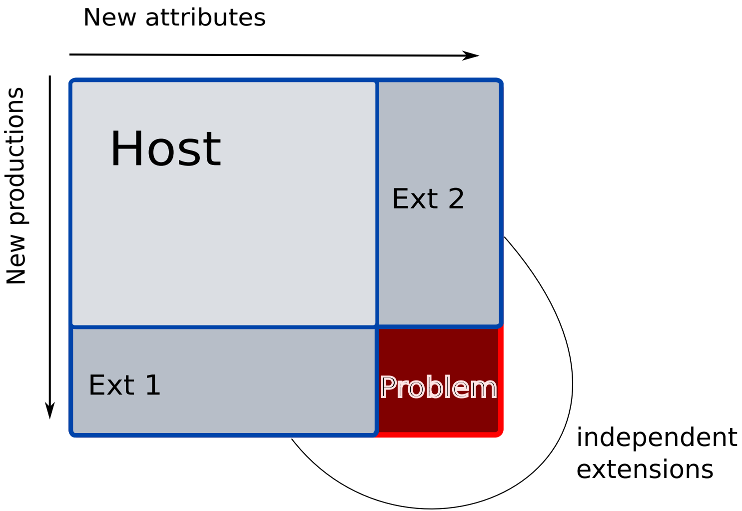
```
grammar edu:umn:cs:melt:ableP:host:core:abstractsyntax;

abstract production defaultAssign
s::Stmt ::= lhs::Expr rhs::Expr
{ s.pp = lhs.pp ++ " = " ++ rhs.pp ++ " ;\n" ;

  lhs.env = s.env;   rhs.env = s.env;
  s.defs = emptyDefs();

  s.errors := lhs.errors ++ rhs.errors ;
}
```

Adding extension constructs involves writing similar productions.

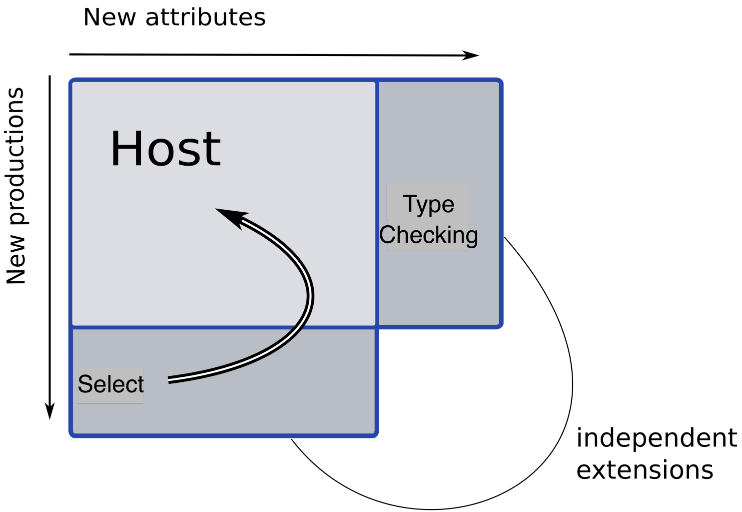
Adding ETCH-like semantic analysis.

```
grammar edu:umn:cs:melt:ableP:extensions:typeChecking ;

synthesized attribute typerep::TypeRep
  occurs on Expr, Decls ;

aspect production varRef
e::Expr ::= id::ID
{ e.typerep = ... retrieve from declaration
                  found in e.env ... ;   }

aspect production defaultAssign
s::Stmt ::= lhs::Expr rhs::Expr
{ s.errors <- if isCompatible(lhs.typerep, rhs.typerep)
              then [ ]
              else [ mkError ("Incompatible types ...") ];
}
```

Extensibility: safe composability



Extensibility: safe composability



Extensions get undefined semantics from host translation.

```
grammar edu:umn:cs:melt:ableP:extensions:enhancedSelect ;

abstract production selectFrom
s::Stmt ::= sl::'select' v::Expr es::Exprs
{
  s.pp = "select ( " ++ v.pp ++ ":" ++ es.pp ++ " ); \n" ;

  s.errors := v.errors ++ es.errors ++
     if  ... check that all expressions in 'es' have
             same type as 'v' ...
     then [ mkError ("Error: select statement " ++
                     "requires same type ... ") ]
     else [ ] ;

  forwards to ifStmt( mkOptions (v, es) ) ;
}
```

Modular analysis

Ensuring that the composition will be successful.

Context free grammars

$$\boxed{G_H} \cup \boxed{G_E^1} \cup \boxed{G_E^2} \cup \; \ldots \boxed{G_E^i}$$

- $\cup$ of sets of nonterminals, terminals, productions
- Composition of all is an context free grammar.
- Is it non-ambiguous, useful for deterministic (LR) parsing?
- $\mathit{conflictFree}(G_H \cup G_E^1)$ holds
- $\mathit{conflictFree}(G_H \cup G_E^2)$ holds
- $\mathit{conflictFree}(G_H \cup G_E^i)$ holds
- $\mathit{conflictFree}(G_H \cup G_E^1 \cup G_E^2 \cup \; \ldots \; \cup G_E^i)$ may not hold

Attribute grammars

$$\boxed{AG_H} \cup \boxed{AG_E^1} \cup \boxed{AG_E^2} \cup \ \dots \boxed{AG_E^i}$$

- $\cup$ of sets of attributes, attribute equations, occurs-on declarations
- Composition of all is an attribute grammar.
- Completeness: $\forall$ production, $\forall$ attribute, $\exists$ an equation
- $complete(AG_H \cup AG_E^1)$ holds
- $complete(AG_H \cup AG_E^2)$ holds
- $complete(AG_H \cup AG_E^i)$ holds
- $complete(AG_H \cup AG_E^1 \cup AG_E^2 \cup \ \dots \ \cup AG_E^i)$ may not hold
- similarly for non-circularity of the AG

Detecting problems, ensuring composition

When can some analysis of the language specification be applied?
When ...

1. the host language is developed ?
2. a language extensions is developed ?
3. when the host and extensions are composed ?
4. when the resulting language tools are run ?

Libraries, and modular type checking

- Libraries “just work”
- Type checking is done by the library writer, modularly.
- Language extensions should be like libraries, composition of “verified” extensions should “just work.”

Modular determinism analysis for grammars, 2009

$$\boxed{G_H} \cup \boxed{G_E^1} \cup \boxed{G_E^2} \cup \ \ldots \boxed{G_E^i}$$

- $isComposable(G_H, G_E^1) \wedge conflictFree(G_H \cup G_E^1)$ holds
- $isComposable(G_H, G_E^2) \wedge conflictFree(G_H \cup G_E^2)$ holds
- $isComposable(G_H, G_E^i) \wedge conflictFree(G_H \cup G_E^i)$ holds
- these imply $conflictFree(G_H \cup G_E^1 \cup G_E^2 \cup \ \ldots)$ holds
- $(\ \forall i \in [1, n].isComposable(G_H, G_E^i) \ \wedge$ $conflictFree(G_H \cup \{G_E^i)\} \ )$ $\Longrightarrow conflictFree(G_H \cup \{G_E^1, \ldots, G_E^n\})$
- Some restrictions to extension introduced syntax apply, of course.

Modular completeness analysis for attribute grammars

$$\boxed{AG_H} \cup \boxed{AG_E^1} \cup \boxed{AG_E^2} \cup \ \ldots \boxed{AG_E^i}$$

- $modComplete(AG_H \cup AG_E^1)$ holds
- $modComplete(AG_H \cup AG_E^2)$ holds
- $modComplete(AG_H \cup AG_E^i)$ holds
- these imply $complete(AG_H \cup AG_E^1 \cup AG_E^2 \cup \ \ldots)$ holds
- $(\forall i \in [1, n].modComplete(AG_H, AG^i_E))$
  $\Longrightarrow complete(AG_H \cup \{AG_E^1, \ldots, AG_E^n\})$.
- similarly for non-circularity of the AG
- Again, some restrictions on extensions.

So ...

- ableP supports the simple composition of language extensions
- This creates translators and analyzers for customized Promela-based languages.
  - extensions can be verified to (syntactically) compose, with other verified extensions — done by extension developers
  - adding (independently developed) extensions that add new features and new analysis on host features is supported
- Challenge: SPIN verification still occurs on the generated pure Promela specification.
- Future work
  - More extensions: multi-dimensional array, unit/dimension analysis, ...
  - Improve type analaysis
  - Semantic analysis of embedded C code?

Thanks for your attention.

Questions?

`http://melt.cs.umn.edu/`
`evw@cs.umn.edu`