Technische Universität Braunschweig

ISF

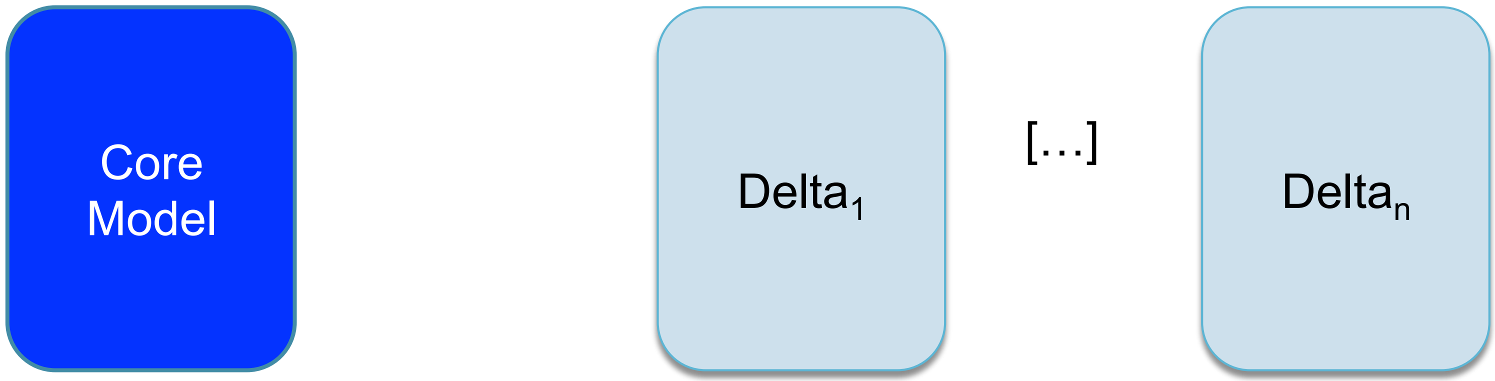
Engineering Delta Modeling Languages

Ina Schaefer

(joint work with Bernhard Rumpe, Arne Haber, Karsten Kolassa, Markus Look, Klaus Müller, Kathrin Hölldobler)

WG 2.11, Minneapolis, June 2013

Delta Modeling



Core Model:

- A model for a complete variant
- Developed with standard techniques or an existing legacy application

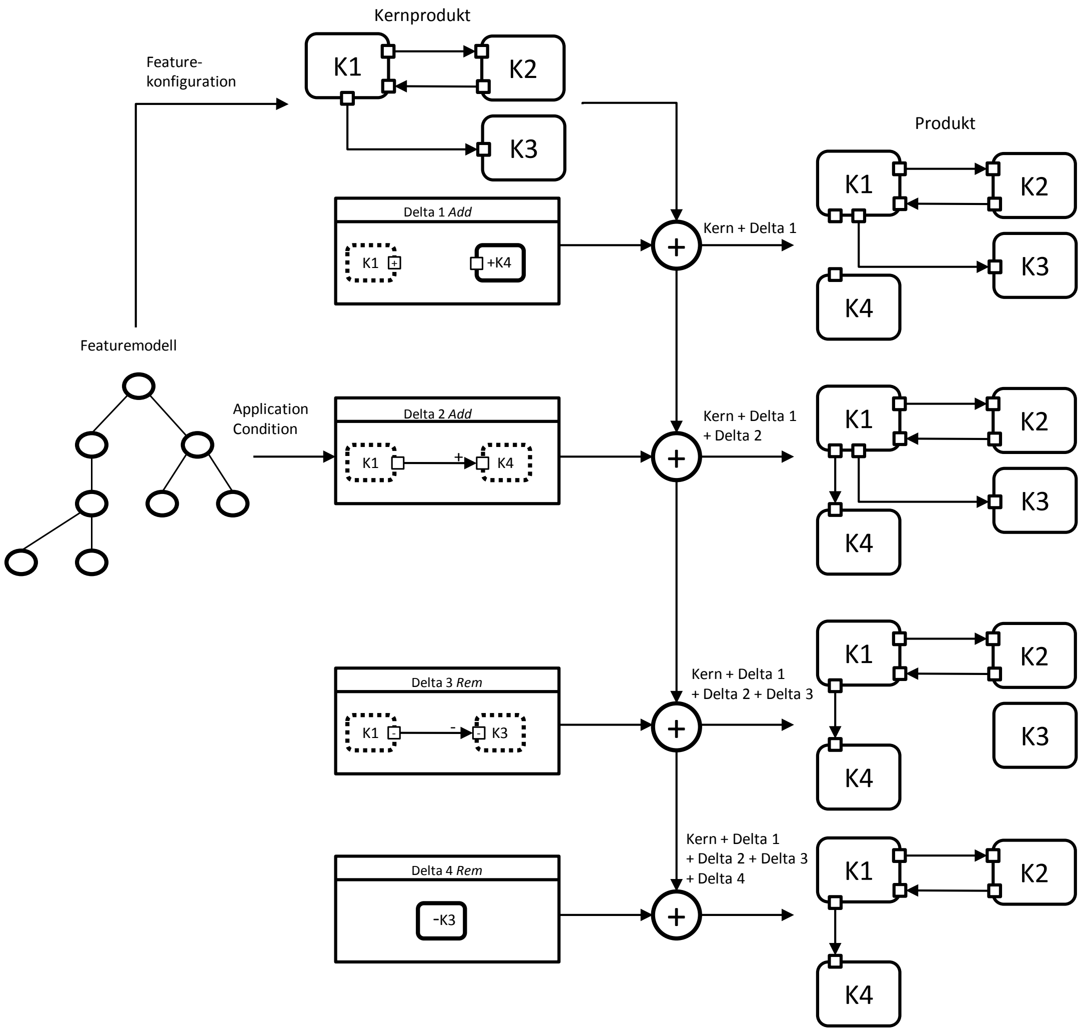
Deltas:

- Modifications of core model
- Selection of deltas for each model variant
- Ordering for conflict resolution

Characteristics of Delta Modeling

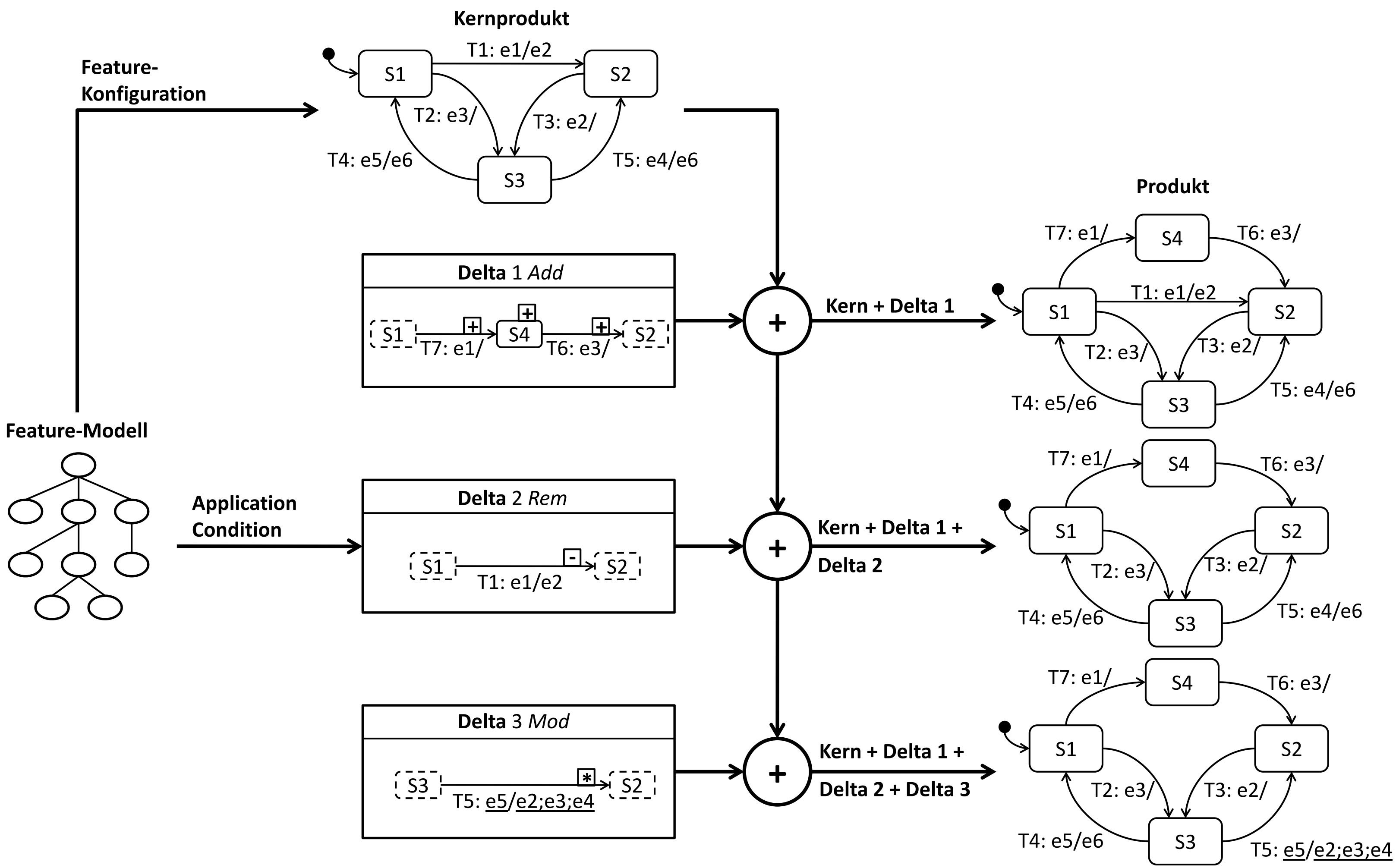
- Modular and flexible description of change
- Automatic variant generation
- Easily integrates variability and evolution
- Paves way for efficient testing, analysis and verification
- Support for proactive, reactive and extractive SPLE

Deltas on Software Architectures



Technische Universität Braunschweig

Deltas on State Machines



Deltas in Java

Core Modul: Java-Programs

```
core Interval, Vehicle_Speed {

class IntervalCmdProcessor {
  int intervalSelection;
  int vehicleSpeed;
  int wipeCmd;

  void computeWipeCmd() {
    [...]
  }
}
```

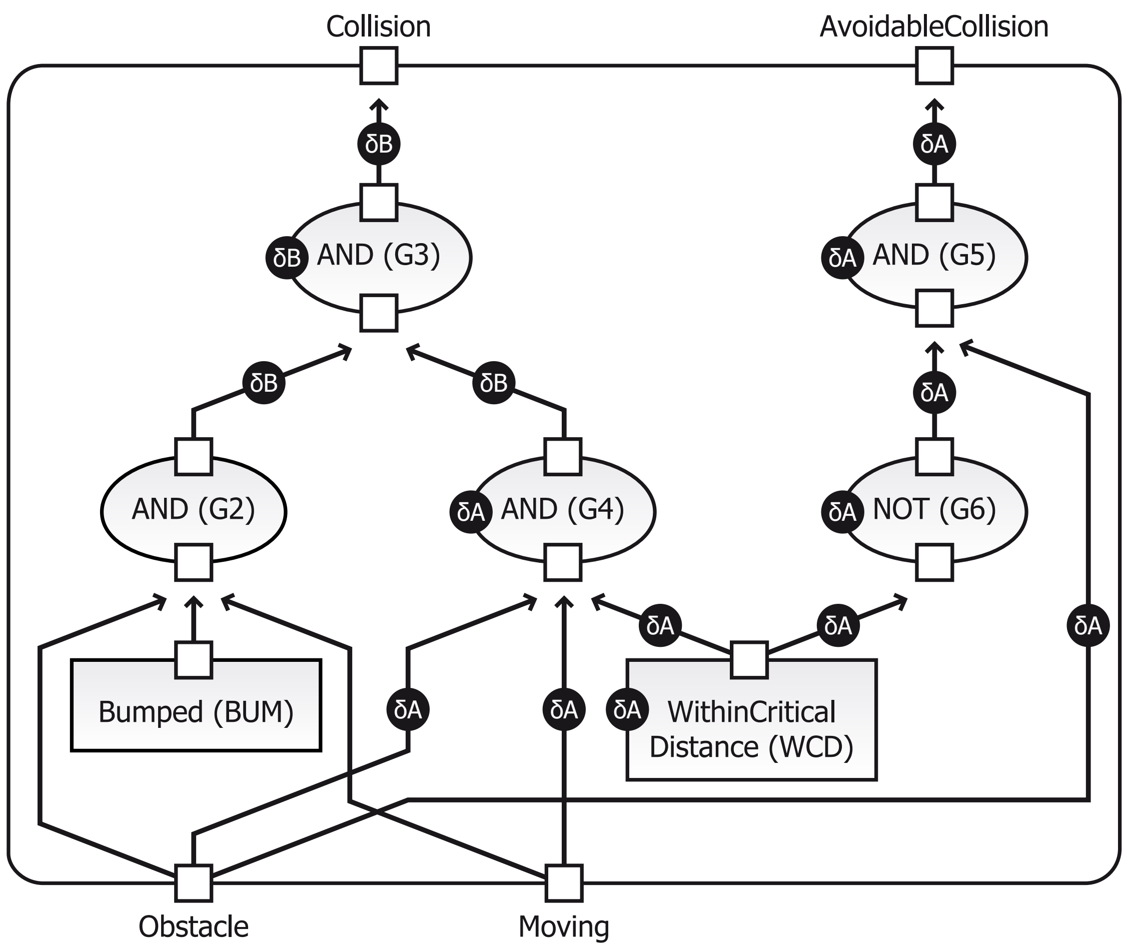
Delta-Modul: Changes to Programs

```
delta DRainEval when Rain_Sensor {

 modifies class IntervalCmdProcessor {

  adds int rainIntensity;

  modifies computeWipeCmd() {
    [...]
  }
 }
}
```

Type system guarantees syntactic correctness of generated variants:

$$\frac{\forall i \in 1..n, \quad \vdash \mathtt{DC}_i : dcc_i}{\vdash \mathtt{delta}\ \delta\ \cdots\ \{\mathtt{DC}_1 \ldots \mathtt{DC}_n\} : \{dcc_1, \ldots, dcc_n\}} \quad (\text{CT-DELTA})$$

Deltas on Component Fault Diagrams



Agenda

- Language Workbench MontiCore
- Engineering Delta Modeling Languages with MontiCore
- Discussion

MontiCore

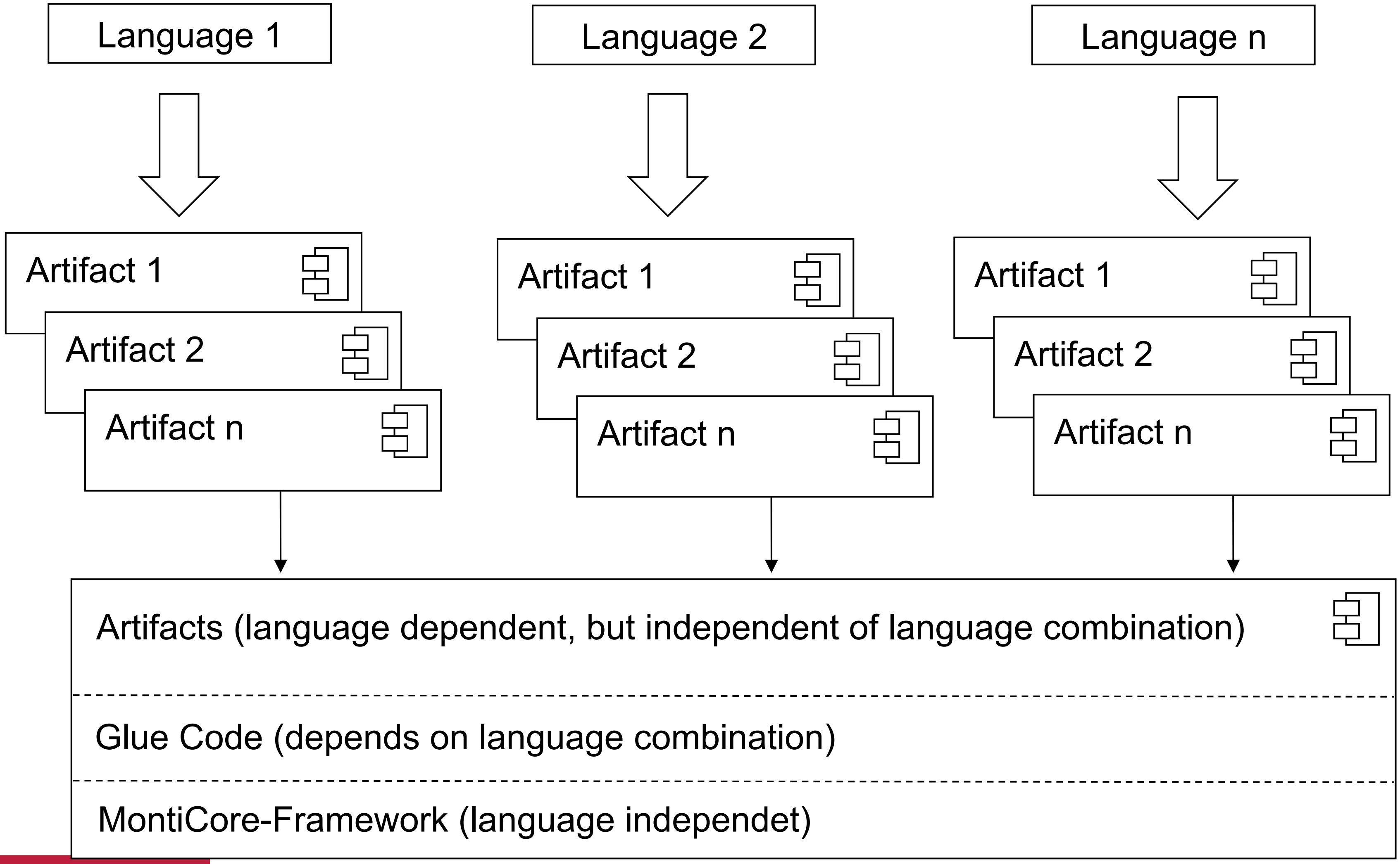
Framework for development and processing of domain-specific languages

Textual, grammar-based definition of languages and tools

- AST classes
- lexer/parser
- pretty printer
- editor

Modular and compositional language development

Language Composition



MontiCore Grammar Format

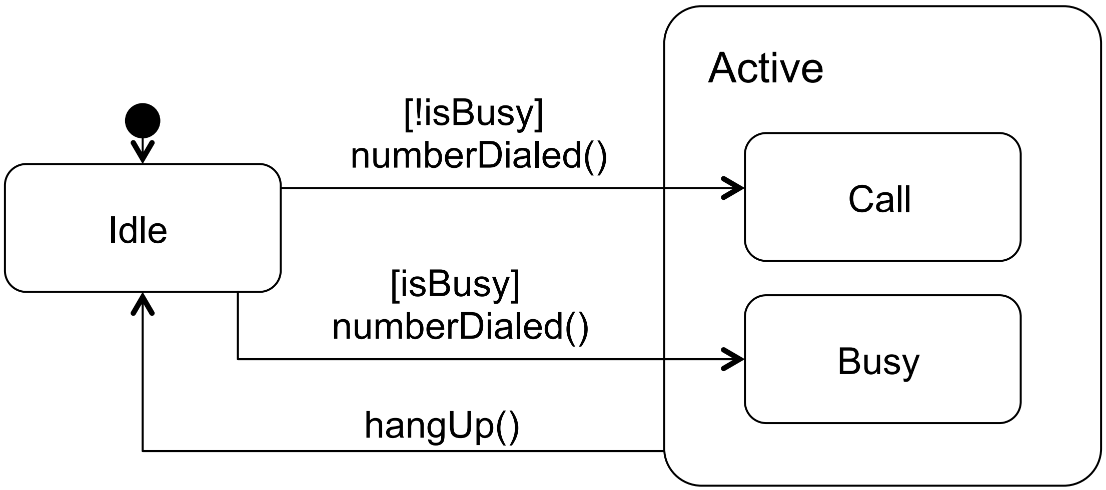
Integrated definition of lexer and parser productions, abstract and concrete syntax.

```
grammar Statechart extends Common {
  SCDefinition =
    "statechart" Name
    "{" Element* "}";

  interface Element;

  Transition implements Element =
    source:Name "->" target:Name
    ( (":" TransitionBody) | ";" );

  State implements Element = "state" Name ...;
   ...
}
```

Interface non-terminal: Set of Alternatives

Statechart Example

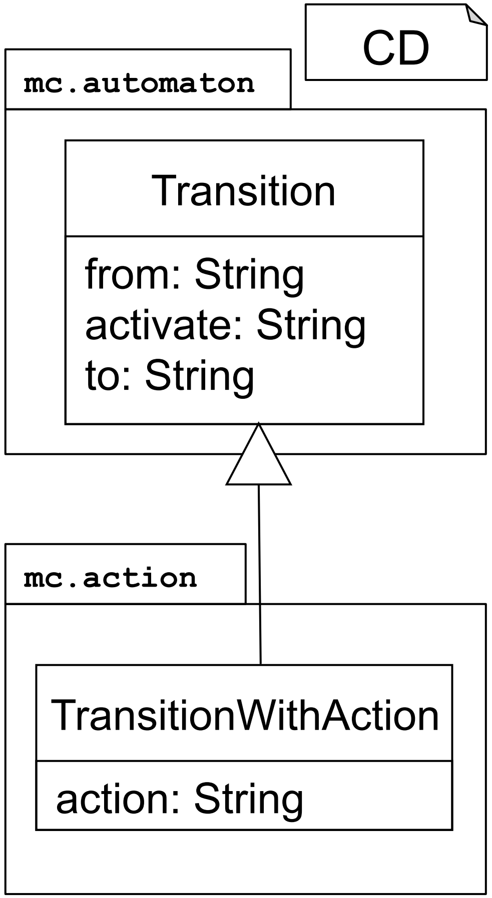
```
statechart Telephone {
   initial state Idle;
   state Active {
      state Busy;
      state Call;
   }
   Idle -> Call :  [!isBusy] numberDialed() ;
   Idle -> Busy : [ isBusy] numberDialed() ;
   Active -> Idle : hangUp();
}
```



Modularity and Reusability: Inheritance

```
package mc.automaton;

grammar Automaton {

  Transition =
    From:IDENT "-" Activate:IDENT ">" To:IDENT ";" ;

}
```

```
package mc.action;

grammar ActionAutomaton extends Automaton {

  TransitionWithAction extends Transition =
    From:IDENT "-" Activate:IDENT
    "[" Action:IDENT "]" ">" To:IDENT ";" ;

}
```



Modularity and Reusability: Embedding

- Production may contain symbols defined in a different grammar

```
grammar... {

  Value = STRING
     | Number
     | IDENT
     | "(" Expression ")"
     | ...

}
```

```
grammar OD extends mc.umlp.common.Common {

   external Value;

   ODAttribute =
         Modifier
         Type?
         name:IDENT
         ("=" Value)? ";";
...
```

import of external nonterminal

usage in production

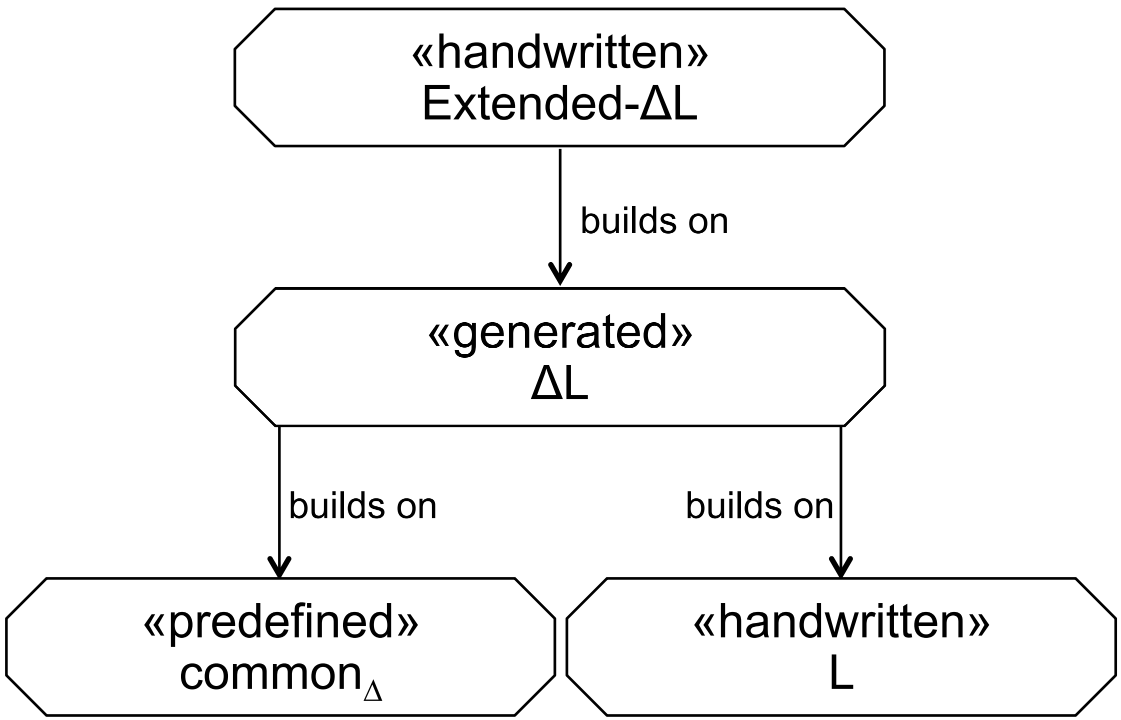
A Delta Language for Statecharts

```
statechart Telephone {
   initial state Idle;
   state Active {
      state Busy;
      state Call;
    }
    Idle -> Call :  [!isBusy] numberDialed() ;
    Idle -> Busy : [ isBusy] numberDialed() ;
   Active -> Idle : hangUp();
}
```

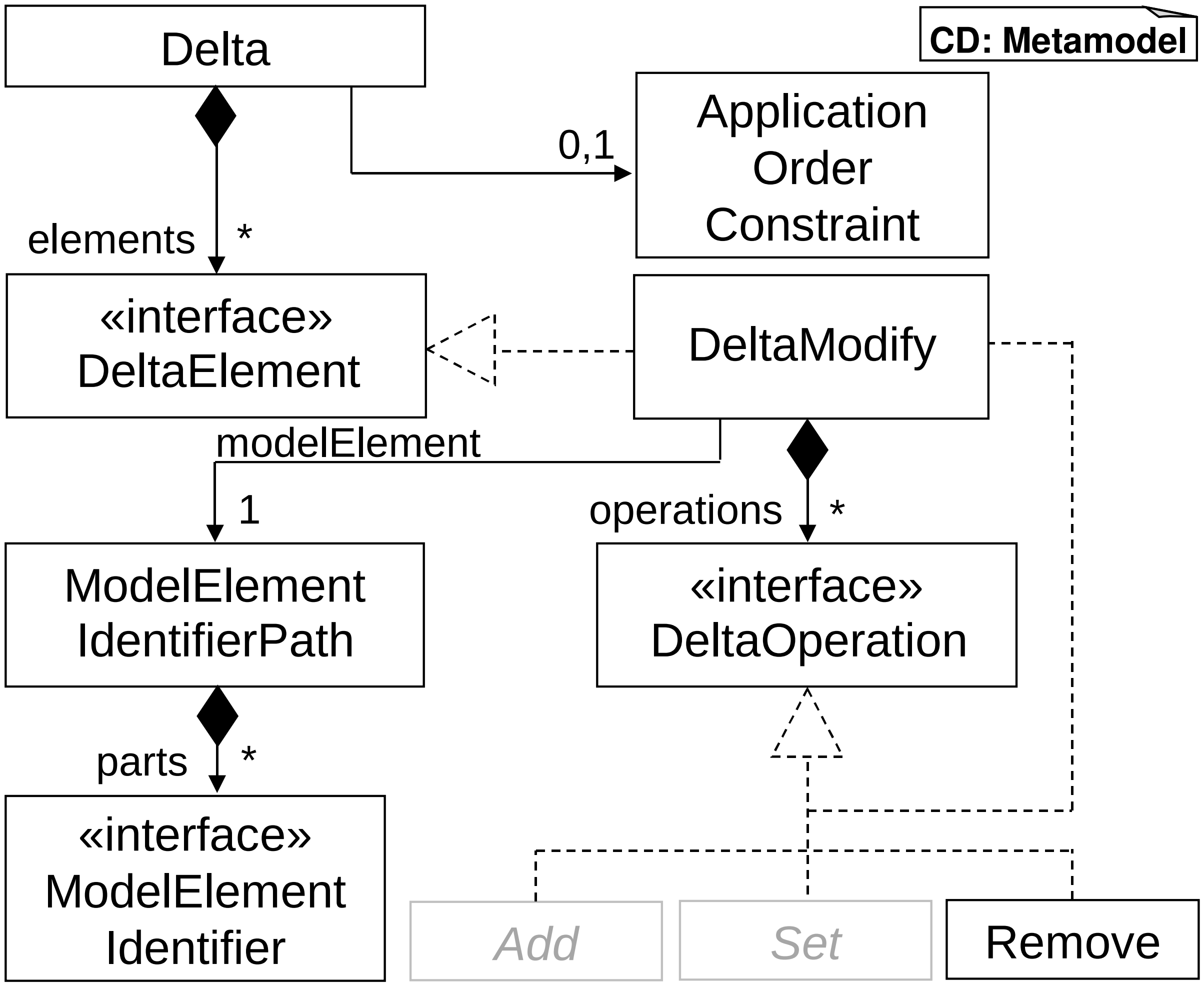
```
statechart Telephone {
   initial state Idle;
   state Active {
      state Voicemail;
      state Call;
   }
   state Dialing;
   Dialing -> Call : [!isBusy] numberDialed();
   Dialing -> Voicemail : [isBusy]
   numberDialed();
   Dialing -> Voicemail : [waited5seconds]
   numberDialed();
   Idle -> Dialing: openLine();
   Active -> Idle : hangUp();}
```

```
delta Voicemail {
  modify statechart Telephone {
    add state Dialing;

    add transition Idle -> Dialing: openLine();

    modify transition [Idle -> Call;]{
      set source Dialing;
    }

    modify transition [Idle -> Busy;]{
      set source Dialing;
    }

    modify state Active.Busy {
      set name Voicemail;
    }

    add transition Dialing -> Voicemail:
      [waited5seconds] numberDialed();
  }
}
```

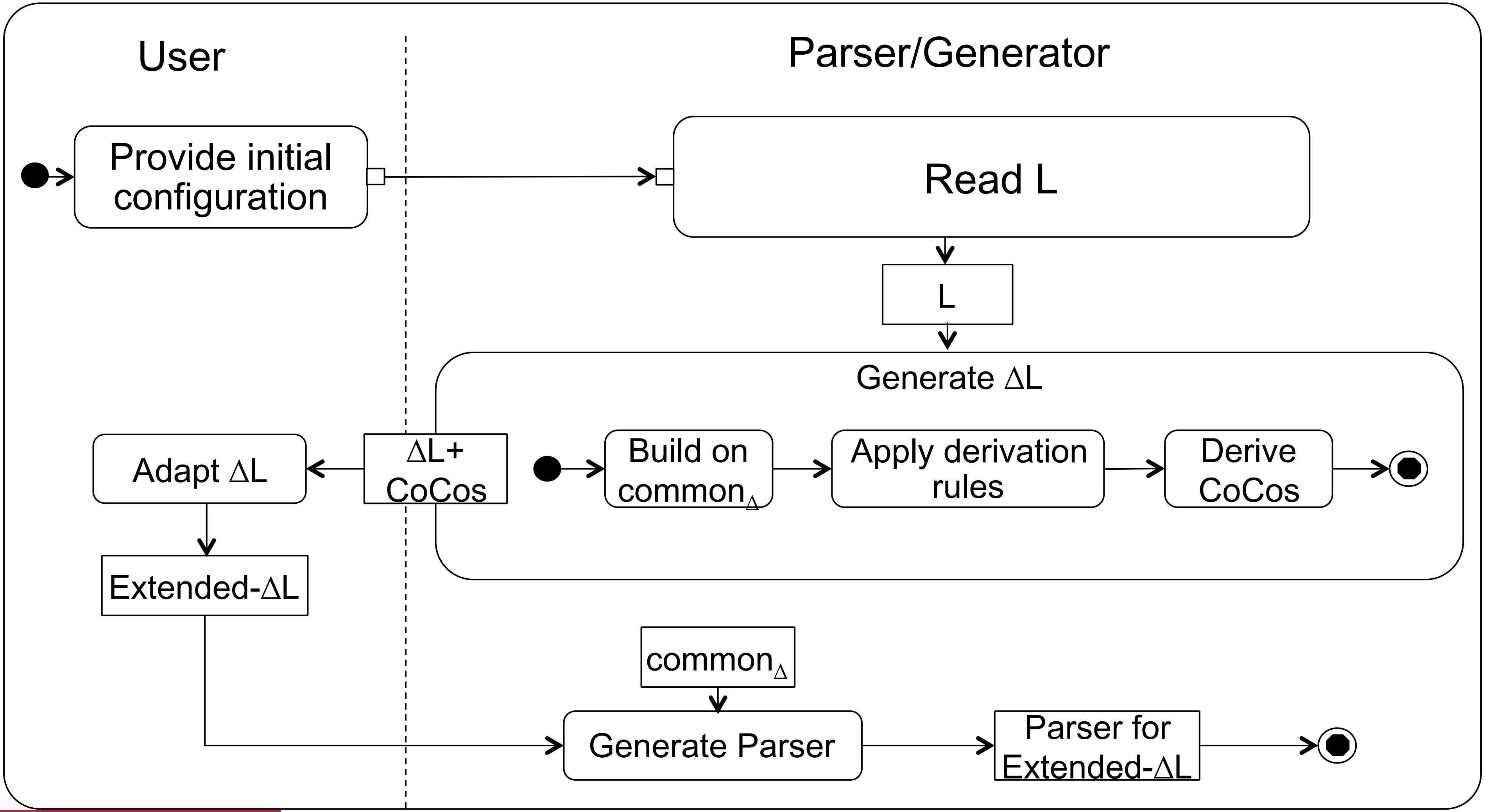
Language Hierarchy



Common-Delta Constructs



Delta Modeling Procedure



Common-Delta as MontiCore Grammar

```
// Elements that may be used directly within a
// delta model.
interface DeltaElement;

// Adds concrete syntax to modifies.
interface ScopeIdentifier;

Delta =
  "delta" Name
  ("after" ApplicationOrderConstraint)?
  "{"
    elements:DeltaElement*
  "}";

DeltaModify implements
      DeltaOperation, DeltaElement =
  "modify" ScopeIdentifier
  modelElement:ModelElementIdentifierPath "{"
  DeltaOperation*
  "}";

// To identify model elements.
interface ModelElementIdentifier;

// Hierarchical path of MEIs.
ModelElementIdentifierPath =
  parts:ModelElementIdentifier
  ("." parts:ModelElementIdentifier)*;

```

```
// Default identifier: qualified name.
DefaultModelElementIdentifier implements
  ModelElementIdentifier =
  QualifiedModelElementName;

interface DeltaOperation;

// Operand of a delta operation.
interface DeltaOperand;

DeltaAdd implements DeltaOperand = "add";
DeltaSet implements DeltaOperand = "set";
DeltaRemove implements DeltaOperand = "remove";

// Default remove operation.
DeltaRemoveOperation implements
    DeltaOperation =
  DeltaRemove target:ModelElementIdentifierPath
  ";";
```

Delta Language Derivation I

1. We need to address every element to define a delta.
   - For every nonterminal `N` that can be identified by a qualifiied name, use the default implementation to address it.
   - For every other nonterminal `N`, square brackets are used to address it. We introduce a new nonterminal:

     Δ`N`$_{MEI}$ `implements ModelElementIdentifier= "[" N "]„`

2. We need to define the scope.
   - For every nonterminal `N` $\in$ L, we introduce a new nonterminal `N`$_{SI}$ and generate a production of the form:

     Δ `N`$_{SI}$ `implements ScopeIdentifier = "N"`

Delta Language Derivation II

3. We need rules to specify the delta operation we want to apply.
   - For every nonterminal $N \in L$, we introduce a new nonterminal N_{DO} and generate an operation production of the form:

     Δ `N`$_{DO}$ `implements DeltaOperation = DeltaOperand N`

4. We need to consider that nonterminals can be used more than once on the RHS of a production.
   - For every nonterminal $N \in L$ and for each identier $\mathbf{n}_i$ of **N**, we introduce a new nonterminal **n**DOi and generate a production of the form:

     Δ $\mathbf{n}_{DOi}$ `implements DeltaOperation = DeltaOperand "`$\mathbf{n}_i$`" N`

5. We need to add delimiters.
   - For every nonterminal $N \in L$ that is neither a block statement nor a single line statement with a line delimiter, we modify the operation production and append a delimiter.

Context Conditions

In addition to the derivation rules to create ΔL, we generate context conditions that provide semantic.

The context conditions can check that,

1. the `ModelElementIdentifier` is referencing an existing element.
2. the `ModelElementIdentifier` references a model element that corresponds to its type.
3. a `ModelElementIdentifierPath` is valid in terms of its single concatenated elements.
4. the `DeltaOperation` is applicable within the scope of its surrounding modify statement.
5. a `DeltaOperand` is applicable for its element.
6. an element which should be added does not exist yet.
7. an element which should be remove exists.

Case Study

```
grammar Statechart extends Common {
    SCDefinition =
        "statechart" Name
        "{" Element* "}";

    interface Element;

    Transition implements Element =
        source:Name "->" target:Name
        ( (":" TransitionBody) | ";" );

        State implements Element = state
Name ... ;
...
```

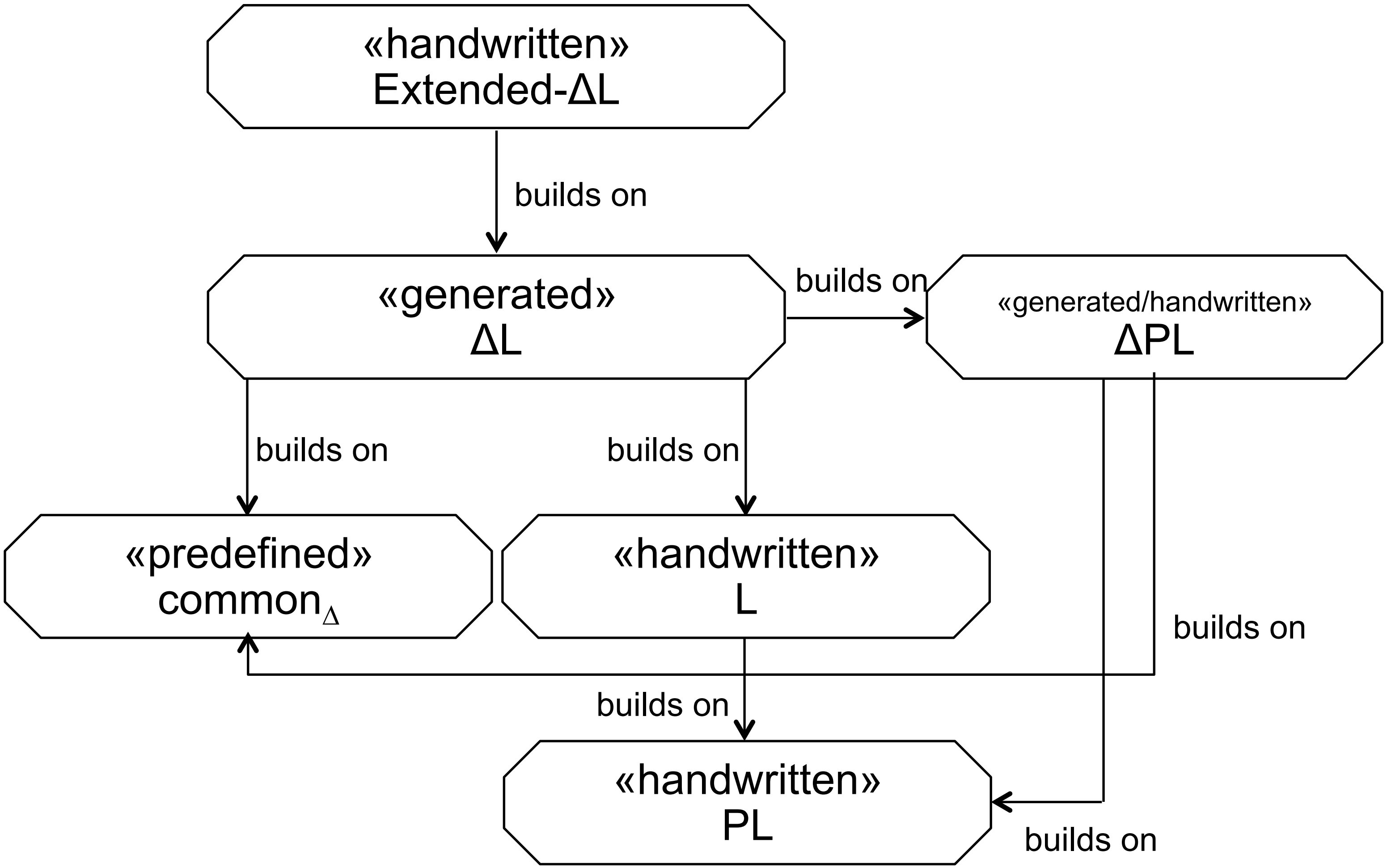
```
grammar DeltaStatechart extends CommonDelta,
Statechart {
  DeltaSCDefinition = Delta;

SCTransitionIdentifier implements
("[" SCTransition "]")=>
ModelElementIdentifier = "[" SCTransition "]";

DeltaSCStateScopeIdentifier implements
ScopeIdentifier = "state";
}
```

1. We start using the Statechart grammar
2. We create our new grammar
3. We implement the `ModelElementIdentifier` production for every nonterminal N ∊ L.
4. We derive the productions for the modify statements, which are used to denote which Statechart grammar construct we want to modify.
5. …

```
DeltaSCStateChartScopeIdentifier
implements ScopeIdentifier =
"statechart";

DeltaSCTransitionScopeIdentifier
implements
ScopeIdentifier = "Transition";
```

```
DeltaStateOperation implements
(DeltaOperand Transition)=> DeltaOperation
= operand:DeltaOperand Transition;

DeltaStateOperation implements
(DeltaOperand State)=> DeltaOperation =
operand:DeltaOperand State;
```

Extensions


«handwritten»
Extended-ΔL
builds on
«generated»
ΔL
builds on
«generated/handwritten»
ΔPL
builds on
builds on
«predefined»
common$_\Delta$
«handwritten»
L
builds on
builds on
«handwritten»
PL
builds on

Conclusion

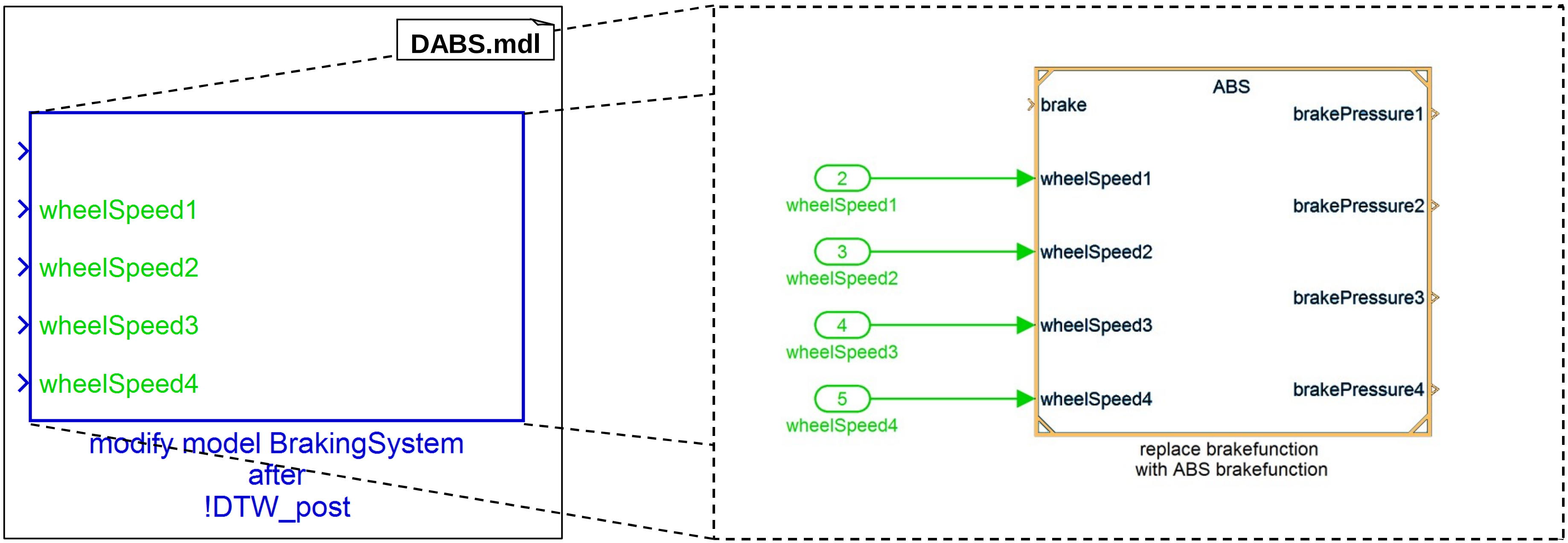
Summary:

- Delta modeling is a flexible approach to represent variability and evolution of models.
- Delta modeling languages can be generated for textual modeling languages.

Future Work:

- Perform larger case studies on various modeling languages
- Generating delta modeling languages for visual modeling languages?
- Development of a pluggable delta type system

Visual Delta Modeling in Simulink



Related Work

Comparison with Model transformation languages:

(D1) Addressing elements to be modified:

- Delta models refer to concrete model.
- Graph transformation rules describe general patterns.

(D2) Negative application conditions:

- Delta languages deliberately don’t offer constructs to define NACs.
- In order to avoid invalid models after transformation, NACs are used in transformative approaches.

(D3) Different scope:

- Delta languages provide a restricted amount of delta operations for model-specific modifications.
- In contrast, graph transformation rules are capable of modeling arbitrary modifications.