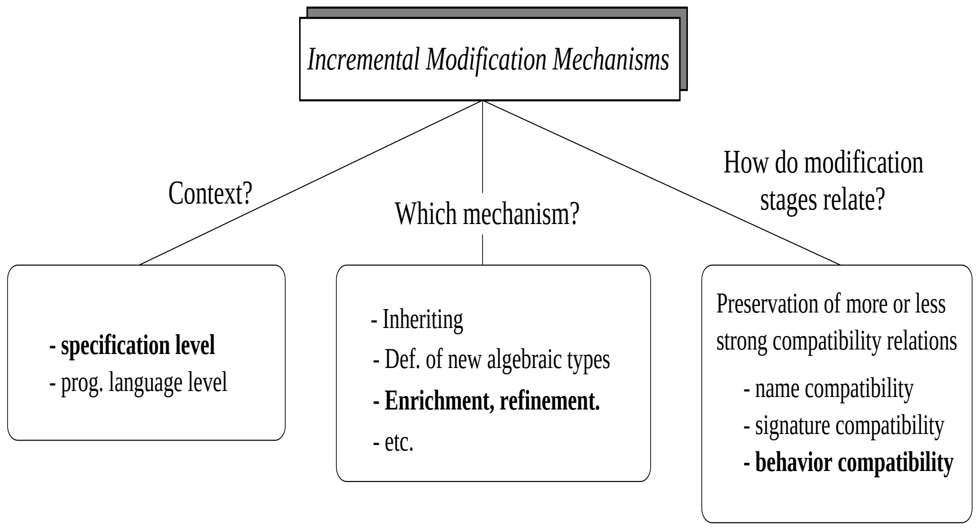
Journée scientifique *ConForM* - Juin 97

Subtyping as a Support for Incremental Development

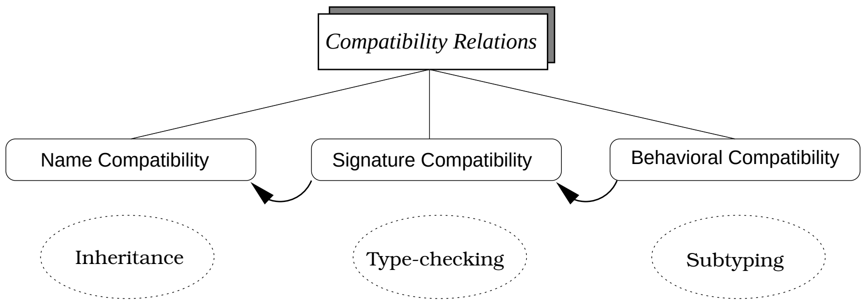
Julie Vachon

Ecole Polytechnique Fédérale de Lausanne - *Laboratoire de Génie Logiciel*,
CH-1015 Lausanne, Switzerland

Aspects of Incremental Modification in the Software Development Life Cycle



Varieties of Compatibility Relations



Context	*Mechanisms inducing compatibility relations*	*Comments*
O.-O. Programming Languages	↪ Inheritance	(@ execution time)
	↪ Type-checking	(@ compile time)
O.-O. Specification Languages	↪ Inheritance	
	↪ Type-checking	(types ≡ domains of values)
	↪ Subtyping	(types ≡ behaviors)

How do inheritance, type-checking and subtyping constraint incremental modification in specification languages?

1. The distinction between notions of class and type.
2. The compatibility relations and their constraining power...

Types and Classes

Classes	*Types in static type-checking*	*Types in behavioral specifications*
Templates.	Predicates defining domains of values.	Predicates defining behaviors.
Instance creation semantics.	Semantics based on inclusion relations of domains.	Semantics based on algebraic and axiomatic models.
Subclasses are obtained by template modification.	Subtypes are obtained by constraining domains.	Subtypes are enriched/refined behavioral specifications obeying to some more or less contraining version of the substitutability principle.
Verification: none.	Verification: type-checking algorithm (co-variant/contra-variant policies, etc.).	Verification: methodological proof of some notion of behavioral equivalence.

Inheritance
An Operational View

$\equiv$ Hierarchical incremental modification mechanism for entities defined by sets of attributes that allow any $R = P + M$ to be further modified to $R1 = R + M1$, for example. [*Wegner & Zdonik*]

$\equiv$ A composition mechanism that internalizes inherited attributes by late (execution-time) binding of self-reference to the inheriting objet. [Cook]

$\Rightarrow$ Implies design decision concerning:

modifiability (*How?*), **granularity** *(Which unit?)*, **multiplicity** (*multiple inheritance?*), **quality** (*What is inherited?*)

Journée scientifique ConForM - Juin 97

Inheritance (con't)

Entities concerned by the modification => **Class Templates.**

Classes - Templates for object generation.

- May be seen as types defining collections of instances sharing a common creation implementation.

In its basic form, inheritance gives way to

*** NAME COMPATIBILITY ***

- ◇ Specified in terms of search algorithm.
- ◇ Weakest form of compatibility relation (no extra-checking at compile or execution time).
- ◇ Based on the language operational (instance creation) semantics.

Example: SmallTalk

Static Type-Checking

≡ Compile-time checking of constraints on signatures.

Entities concerned by the modification => **Signatures**.

Signatures Types are defined by syntactic algebra specifications with some associated semantics.

Static type-checking gives way to

*** SIGNATURE COMPATIBILITY ***

- ◇ Subtypes depend on inclusion relations of domains.
- ◇ More constraining than name compatibility: requires compile-time verification.
- ◇ Sig. define types which denote partially ordered domains of values.

Example: Typed object-oriented programming languages.

Subtyping

≡ Proving the behavioral equivalence (weather complete, partial, observational, etc.) of an entity when substituted to another.

Entities concerned by the modification => **Formal specifications of objects behaviors**.

Formal Specification. Composed of a signature and of a precise semantics defining objects behavior

(Many-sorted algebras, process algebras, transition automata, etc.)

Subtyping gives way to

*** BEHAVIORAL COMPATIBILITY ***

Subtyping (con't)

*** BEHAVIORAL COMPATIBILITY ***

◇ Specified by a notion of behavioral equivalence (strong, weak, observational...)

◇ The most constraining compatibility relation:

(+) implies both name and signature compatibilities;

(+) possibiliy of defining successively stronger notions of behavioral compatibility (i.e. possible contextual relaxation of constraints)

(−) may require fastidious proofs (undecidable problems...)

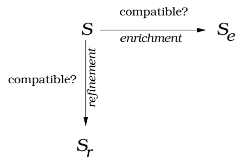
◇ Based on some more or less strict satisfaction of the substitutability principle.

Example: Order Sorted Algebras, work of Nierstraz, Puntigam, Wing & Liskov, etc.

Subtyping (con't)

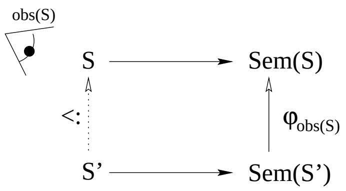
Principle of substitutability: An instance of a subtype can always be used in any context in which an instance a of a super-type was expected. (*Wegner and Zdonik, 1988*)

Achieving and using compatibility relations during software development process...

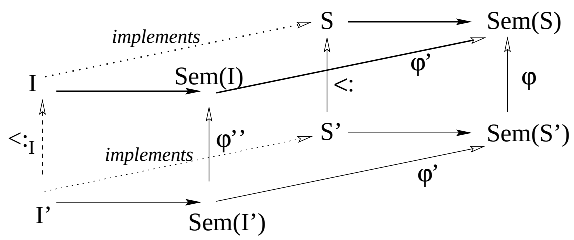


Compatibility relation	*How is it realizable?*
Name Compatibility	Easily realizable (given by the language's name search algo. and its associated renaming mechanism).
Signature Compatibility	Requires adapting objects interfaces to satisfy some set of rules constraining signatures: - covariant policy (doesn't ensure absence of type errors under substitution), - contravariant policy (may oppose to some view of data refinement)

Behavioral Compatibility	$\Rightarrow$ Aim of our work: *How do we put to work behavioral compatibility in the software development process? How is it useful?* - Definition of a semantical model for objects (in terms of transition systems). - Identification of various kinds of behavioral equivalence relations between objects of the semantics. Identification of properties preserved by each relation. - Formal definition of "subtyping" in a specification language context: subtype relations shall each be interpreted as specific kinds of observational behavioral equivalence relation in the semantical model.

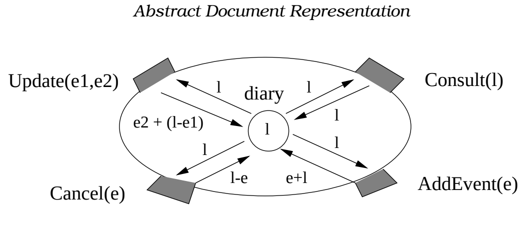


Behavioral Compatibility | Showing subtyping's usefulness for software verification:
- partial or complete validation of the development process (enrichment/refinement)
- validation of implementations



Journée scientifique ConForM - Juin 97

ADR's *Specification Diagram*



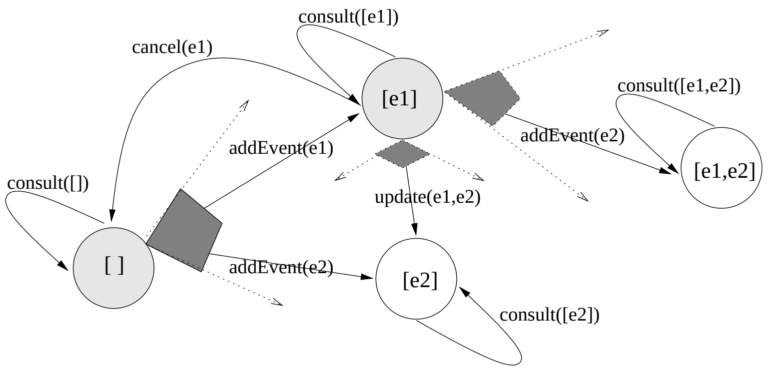
ADR's *Specification in COOPN*

CLASS ADR
INTERFACE
  USE Booleans, ListEvent, Event;
  METHODS
    Consult _ : listevent;
    AddEvent _: event, event;
    Update _ _: event, event;
    Cancel _ : event;
BODY
  PLACES
    diary: listevent;
  INITIAL
    diary [];

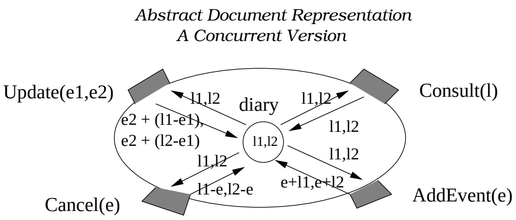
AXIOMS
  Consult(l) : diary l $\rightarrow$ diary l;
  AddEvent(e) : diary l $\rightarrow$ diary (e+l);
  e1 isin l = true $\Rightarrow$
    Update(l) :
      diary l $\rightarrow$ diary (e2 + (l-e1));
  e isin l = true $\Rightarrow$
    Cancel(e) : diary l $\rightarrow$ diary (l-e);
WHERE
  e, e1, e2 : event;
  l : listevent;
END ADR

ADR's *behavioral type*

(Transition System)



First Example
`ADR_concur`*'s Specification Diagram*

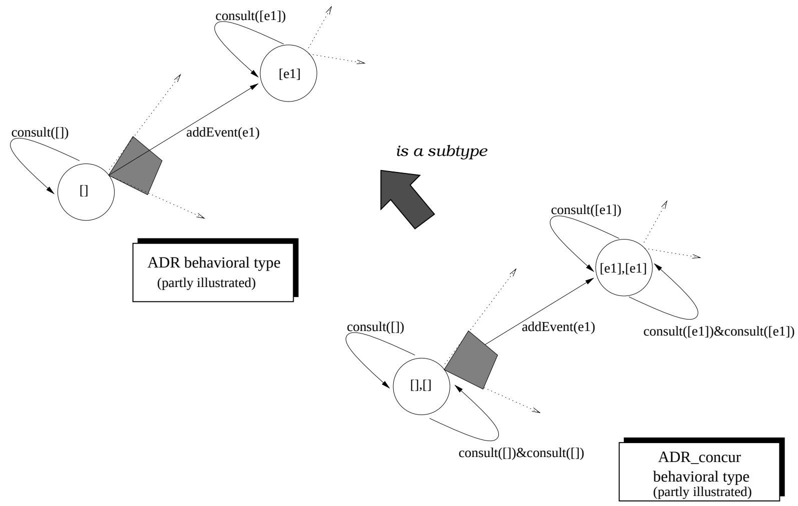


First Example
Enrichment of `ADR` *:* `ADR_concur`

```
CLASS ADR
INTERFACE
  USE Booleans, ListEvent, Event;
  METHODS
    Consult _ : listevent;
    AddEvent _: event, event;
    Update _ _: event, event;
    Cancel _ : event;
BODY
  PLACES
    diary: listevent;
  INITIAL
    diary [];
AXIOMS
  Consult(l) : diary l → diary l;
  AddEvent(e) : diary l → diary (e+l);
  e1 isin l = true ⇒
    Update(l) :
      diary l → diary (e2 + (l-e1));
  e isin l = true ⇒
    Cancel(e) : diary l → diary (l-e);
```

```
CLASS ADR_concur
INTERFACE
  USE Booleans, ListEvent, Event;
  METHODS
    Consult _ : listevent;
    AddEvent _: event, event;
    Update _ _: event, event;
    Cancel _ : event;
BODY
  PLACES
    diary: listevent;
  INITIAL
    diary [], diary [];
AXIOMS
  Consult(l) : diary l → diary l;
  AddEvent(e) : diary l1, diary l2 →
        diary (e+l1), diary (e+l2);
  e1 isin l = true ⇒
    Update(l1) :
      diary l1, diary l2 →
      diary (e2 + (l1-e1)), diary (e2 + (l2-e1));
  e isin l = true ⇒
    Cancel(e) : diary l1, diary l2 →
        diary (l1-e), diary (l2-e);
```

ADR_concur *as a subtype of* ADR



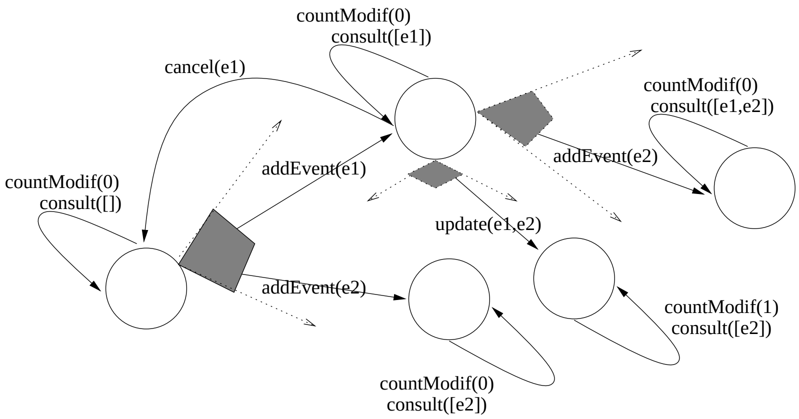
Second Example
Enrichment of ADR : ADR_count

```
CLASS ADR
INTERFACE
  USE Booleans, ListEvent, Event;
  METHODS
    Consult _ : listevent;
    AddEvent _: event, event;
    Update _ _: event, event;
    Cancel _ : event;
BODY
  PLACES
    diary: listevent;
  INITIAL
    diary [];
AXIOMS
  Consult(l) : diary l → diary l;
  AddEvent(e) : diary l → diary (e+l);
  e1 isin l = true ⇒
    Update(l) :
      diary l → diary (e2 + (l-e1));
  e isin l = true ⇒
    Cancel(e) : diary l → diary (l-e);
```

```
CLASS ADR
INTERFACE
  USE Booleans, Naturals, ListEvent, Event;
  METHODS
    Consult _ : listevent;
    AddEvent _: event, event;
    Update _ _: event, event;
    Cancel _ : event;
    CountModif _ : natural;
BODY
  PLACES
    diary: listevent;
    nbModif: natural
  INITIAL
    diary []; nbModif 0;
AXIOMS
  Consult(l) : diary l → diary l;
  AddEvent(e) : diary l → diary (e+l);
  e1 isin l = true ⇒
    Update(l) :
      diary l, nbModif n →
      diary (e2 + (l-e1)), nbModif (n+1);
  e isin l = true ⇒
    Cancel(e) : diary l → diary (l-e);
  CountModif(n) : nbModif n → nbModif n;
```

`ADR_count`*'s behavioral type*

(Transition System)



Some Questions to Conclude...

- ◇ Comment agreement needed on
  - ◇ What is a valid *enrichment* step?
  - ◇ What is a valid *refinement* step?
- ◇ Apart from the *substitutability principle*, can we identify useful variants of this principle?
- ◇ Given some semantical object model, it is possible to define various kinds of *behavioral equivalences* (strong, weak, observational, etc.) Among them, which one are particularly interesting? Which variant of the substitutability principle do they ensure?

- ◇ What do "*being a subtype of*" precisely mean in a specification language? How can subtyping help in software development (refinement/enrichment verification)?
- ◇ Proving subtyping : hard, undecidable? Can this proof work be partly automatized and/or be guided by some well-defined methodology?