Specifica – parte IIID

Leggere Sez. 5.7 Ghezzi et al.

università di ferrara
DA SEICENTO ANNI GUARDIAMO AVANTI.

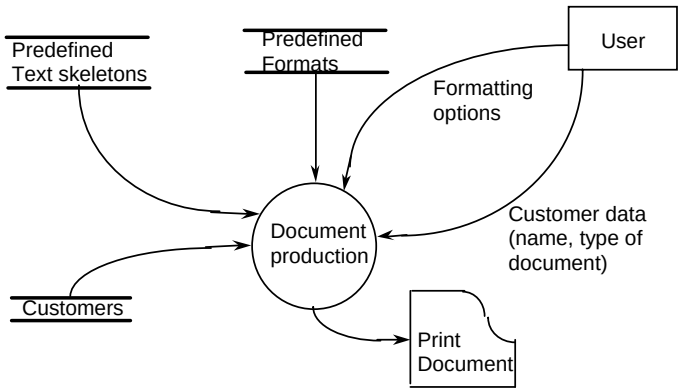
La specifica nei casi reali

- La complessità dei casi reali impone ulteriori requisiti, oltre all'espressività, al rigore e alla formalità
- E' necessario che i linguaggi usati supportino la modularizzazione delle specifiche

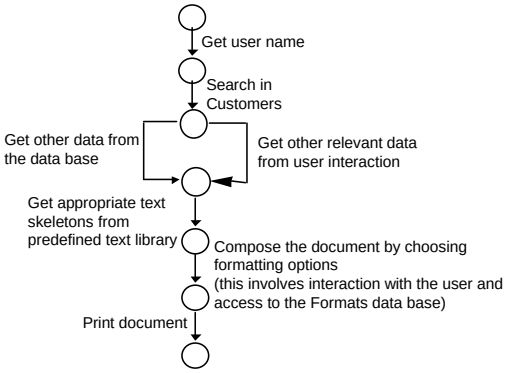
La specifica nei casi reali

- Separazione degli interessi: specificare separatamente aspetti diversi dello stesso problema.
- Esempio: produzione di un documento in un sistema di office automation.
- Specifica separata di:
  - flusso dei dati
  - controllo

Esempio: DFD



Esempio: FSM per controllo



(b)

Incrementalità

- Prima specifica dei requisiti più importanti poi di quelli secondari
- Prima specifica informali, poi semi-formali e infine formali
- Anche per le specifiche formali, versioni successive via via più dettagliate

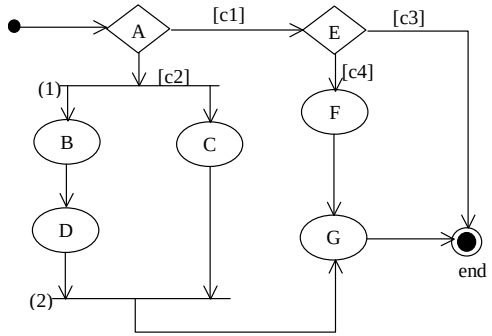
Rigore e formalità

- La formalità completa della specifica può non essere necessaria, soprattutto per le parti destinate agli utenti
- Anche nella specifica dello stesso sistema, è meglio utilizzare linguaggi diversi (informali, semi-formali o formali a seconda dell'uso.

Notazioni UML

- Forniscono diversi linguaggi per descrivere diversi aspetti di un sistema
- Viste diverse e complementari, ad es.
  - class diagram: descrizione statica dell'architettura software
  - statecharts, vedi seguito
  - activity diagram: mostra come le esecuzioni dei metodi possono essere combinate (sequenza o parallelo) e come possono essere sincronizzate (simili alle reti di Petri)

Activity diagram



rombi=attività decisionali, (1) esecuzione parallela di attività, (2) sincronizzazione di attivittà

Specifiche modulari

- Come nel caso della progettazione, la modularità di una specifica favorisce:
  - la gestione della complessità del sistema da definire
  - il riuso di componenti
- Costrutti appositi da inserire nei linguaggi di specifica

Specifiche algebriche modulari

- Alcuni tipi di dato come i numeri naturali o i valori logici sono utilizzati da molti altri tipi di dato
- E' utile poter riusare le algebre corrispondenti come componenti
- Gerarchie di algebre in Larch

Specifiche algebriche modulari

```
algebra BoolAlg;
introduces
        sorts Bool;
        operations
                true () → Bool;
                false () → Bool;
                not : Bool → Bool;
                and: Bool, Bool → Bool;
                or: Bool, Bool → Bool;
                implies: Bool, Bool → Bool;
                ≡ : Bool, Bool → Bool;
constrains true, false, not, and, or, implies, ≡ so that
Bool generated by [true, false]
for all [a, b: Bool]
    not (true) = false;
    not (false) = true;
    a and b = not (not (a) or not (b));
    a implies b = not (a) or b;
...
end BoolAlg.
```

Specifiche algebriche modulari

```
algebra NatNumb;
introduces
        sorts Nat, Bool;
        operations
            0: () → Nat;
            Succ: Nat → Nat;
            + : Nat, Nat → Nat;
            - : Nat, Nat → Nat;
            = : Nat, Nat → Bool;
...
constrains 0, Succ, +, -, =,..., so that
NatNumb generated by [0, Succ]
...
end NatNumb.
```

```
algebra CharAlg;
introduces
        sorts Char, Bool;
        operations
            'a': () → Char;
            'b' : ()→ Char;
            ...
end CharAlg.
```


imports

- **imports** consente di utilizzare le algebre definite nelle specifiche importate
- È necessariamente una gerarchia
- I sort importati non possono essere ridefiniti

Specifiche algebriche modulari

```
algebra StringSpec;
        imports BoolAlg, NatNumb, CharAlg;
        introduces
            sorts String, Char, Nat, Bool;
            operations
                new: () → String;
                ...
end StringSpec.
```


Algebra Container

```
algebra Container;
imports DataType, BoolAlg, NatNumb;
introduces
        sorts Cont;
        operations
            new: () → Cont;
            insert: Cont, Data → Cont;
                {Data is the sort of algebra DataType, to which
                elements to be stored in Cont belong}
            isEmpty: Cont → Bool;
            size: Cont → Nat;
constrains new, insert, isEmpty, size so that
Cont generated by [new, insert]
for all [d: Data; c: Cont]
        isEmpty (new ()) = true;
        isEmpty (insert (c, d)) = false;
        size (new ()) = 0;
end Container.
```

assumes

- Come imports, permette l'utilizzo dei sort importati e delle loro operazioni
- Come imports, è transitiva: permette l'utilizzo dei sort e delle operazioni delle algebre importate da quella che si importa
- Diversamente da imports, permette la ridefinizione della semantica

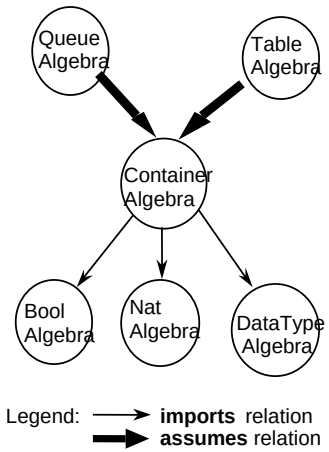
Specializzazione

```
algebra TableAlg;
assumes Container;
introduces
        sorts Table;
        operations
                last: Table → Data;
                rest: Table → Table;
                equalT : Table, Table → Bool;
                delete: Table, Data → Table;
constrains last, rest, equalT, delete, isEmpty, new, insert so that
for all [d, d1, d2: Data; t, t1, t2: Table]
        last (insert (t, d)) = d;
        rest (new ()) = new ();
        rest (insert (t, d)) = t;
        equalT (new (), new ()) = true;
        equalT (insert (t, d), new ()) = false;
        equalT (new (), insert (t,d)) = false;
        equalT (t1,t2) = equalD(last (t1), last (t2)) and
        equalT (rest (t1),rest (t2));
        delete (new (), d) = new ();
        delete (insert (t,d),d) = delete (t, d);
        if not equalD(d1, d2) then
        delete (insert (t, d1), d2) = insert (delete (t, d2), d1);
end TableAlg.
```

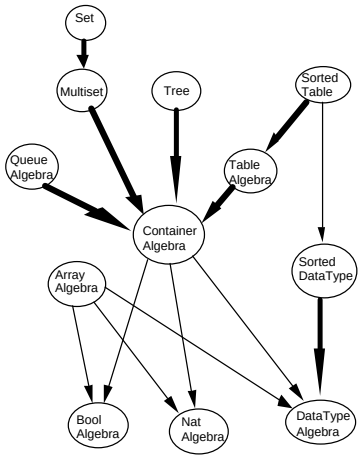

Altra specializzazione

```
algebra QueueAlg;
assumes Container;
introduces
        sort Queue;
        operations
                last: Queue → Data;
                first: Queue → Data;
                equalQ : Queue , Queue → Bool;
                delete:Queue → Queue;
constrains last, first, equalQ, delete, isEmpty, new, insert so that
for all [d: Data; q, q1, q2: Queue]
        last (insert (q, d)) = d;
        first (insert (new(), d) = d
        first (insert (q, d)) = if not isEmpty (q) then first (q);
        equalQ (new (), new ()) = true;
        equalQ (insert (q, d), new ()) = false;
        equalQ (new (), insert (q, d)) = false;
        equalQ (insert (q1, d1), insert (q2, d2)) = equalD (d1, d2) and
        equalQ (q1,q2);
        delete (new ()) = new ();
        delete (insert (new (), d)) = new ();
        if not equalQ (q, new ()) then
        delete (insert (q,d)) = insert (delete (q), d);
end QueueAlg.
```


Rappresentazione grafica



Una gerarchia più ricca



Implementazione in Larch/Pascal

```
type String exports isEmpty, add, append,...
based on Boolean, integer, character
    function isEmpty (s: String) : Boolean
            modifies at most [ ] {i.e., it has no side effects};
    procedure add (var s : String; c : char)
            modifies at most [s]
            {modifies only the string parameter s};
    function length (s: String) : integer
            modifies at most [ ] ;
    procedure append (var s1, s2, s3: string)
            modifies at most [s3]
            {only s3, the result of the operation, is modified};
end StringSpec.
```

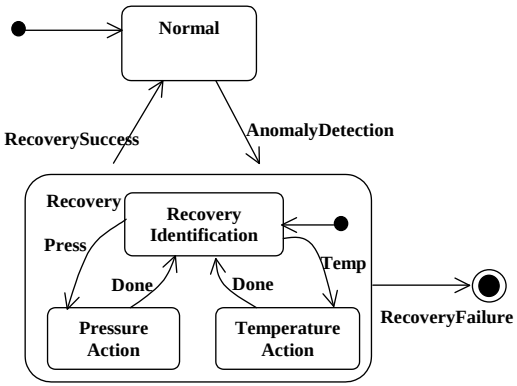

Modularizzazione di FSM: Statecharts

- Notazione incorporata in UML
- Supporta la definizione modulare per mezzo di *superstati*, stati che possono essere a loro volta scomposti in FSM.
- Transizioni e superstati:
  - in entrata: entra nello stato iniziale del superstato
  - in uscita: esce da uno qualsiasi degli stati interni

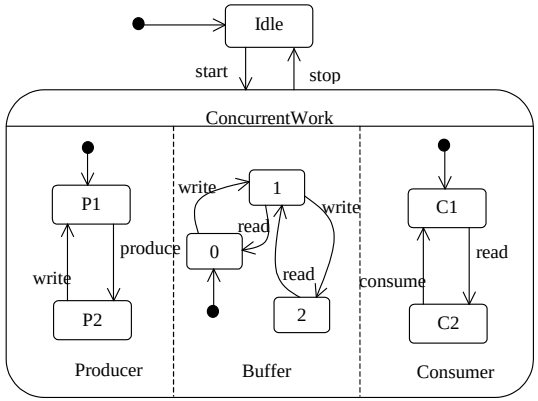
Statecharts

- Transizioni etichettate con uno o più valori di <event, guard, action>
- *event*: l'evento che rende attivabile la transizione
- *guard*: un predicato che deve essere vero affinchè si possa attivare la transizione
- *action*: azione atomica che viene eseguita durante la transizione.

Esempio



Scomposizione parallela



Statecharts con guards