INCREMENTAL SOFTWARE CONSTRUCTION

Ralph-Johan Back

7th August 2004

CREST
CENTRE FOR RELIABLE SOFTWARE TECHNOLOGY

OVERVIEW OF LECTURES

Mathematical framework for *incremental software construction* and *controlled software evolution*.

- *Refinement diagrams:* a visual way of presenting the *construction* and *architecture* of large software systems.

- Diagrams are based on lattice theory, allow reasoning about lattice element to be carried out directly with diagrams.

- *Refinement calculus:* the logic for reasoning about software systems. The calculus models software parts as elements in a lattice.

Apply framework to the incremental construction of large software system. Focus on:

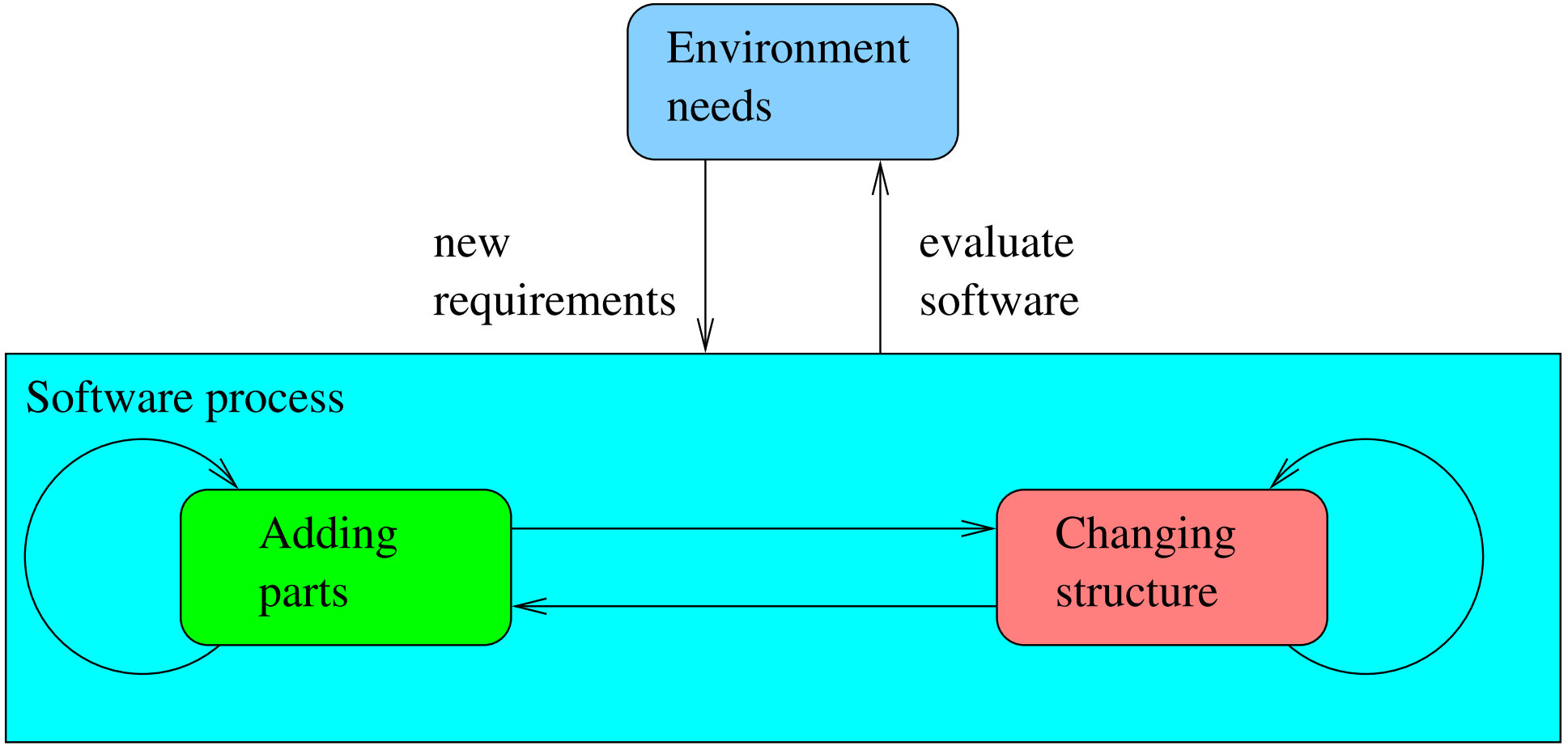
- *modularization* of software systems: component specifications and the role of information hiding
- *layered extension* of software: adding new features and the role of inheritance and dynamic binding
- *evolution of software:* preserving correctness during evolution and managing successive versions.

INCREMENTAL SOFTWARE CONSTRUCTION

Software is never ready, it *evolves* by *adapting* to a changing environment

- *Incremental software construction:* build and change the system in small steps
- *Correctness* is maintained by checking that each increment preserves the correctness of the system built thus far
- Adding increments accumulates *design errors*, which must be corrected by frequent software *redesigns (refactorings)*

Software evolution model



Software process

Software construction is in a continuous interaction with the environment that needs the software

- The present system is checked for conformance with the environment needs
- Based on this, new requirements are given that influence the further development of the system.

The system process alternates between

- adding new parts that implement required features and
- internal redesign of the software in order to meet new demands.

Basic questions

- What is a suitable *conceptual model* for software and its evolution?
- What is a suitable *software architecture* to support evolving software?
- How to *reason* about the *correctness* of evolving software?
- What kind of *software processes* support software evolution?
- What kind of *software tools* do we need to manage software evolution?

LATTICES AND REFINEMENT DIAGRAMS

Need a way of describing and reasoning about evolving software:

1. We define l*attices* and their basic properties

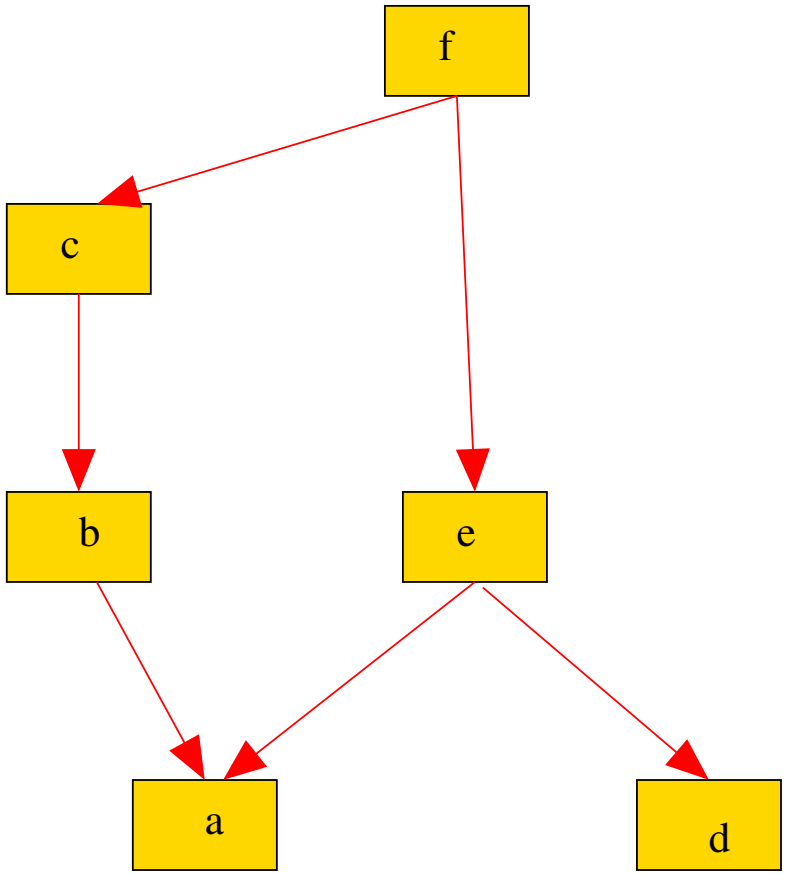
2. We introduce *refinement diagrams* as a way of describing and reasoning about lattice elements

Posets

A *partially ordered set* (or *poset*) is a set A together with an ordering $\sqsubseteq$ that is *reflexive*, *transitive* and *antisymmetric.* This means that for any elements a, b, c in the poset:

- $a \sqsubseteq a$ (*reflexivity)*
- $a \sqsubseteq b \wedge b \sqsubseteq c \Rightarrow a \sqsubseteq c$ (*transitivity)*
- $a \sqsubseteq b \wedge b \sqsubseteq a \Rightarrow a = b$ (*antisymmetry*)

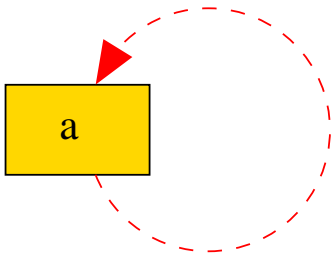
Example poset



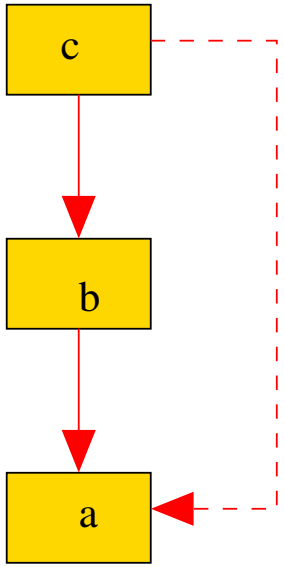
The arrow describes ordering, e.g., $a \sqsubseteq b$.

Refinement diagram rules

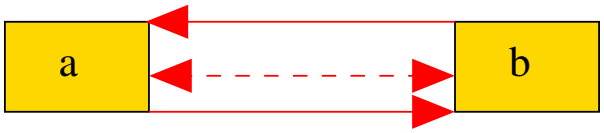
We capture the partial order properties with the following *refinement diagram rules*:



Reflexivity



Transitivity

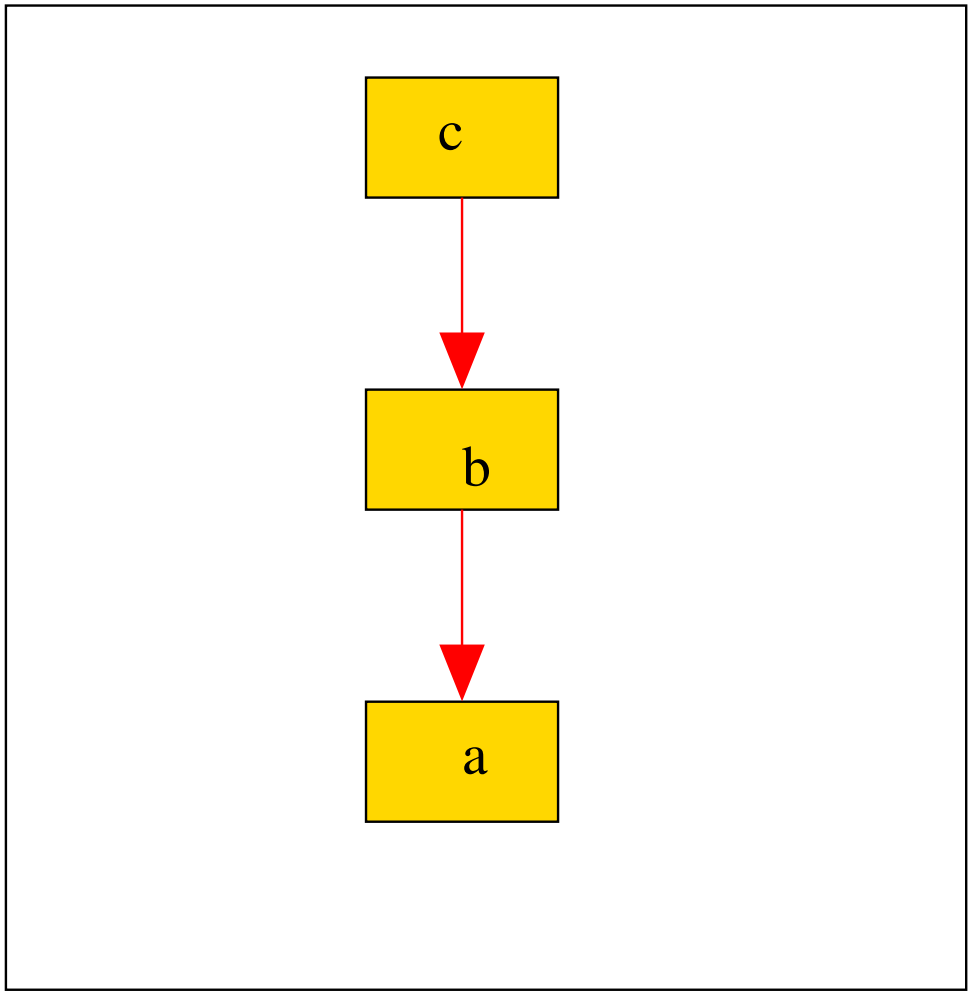


Antisymmetry

Conventions for refinement diagram rules

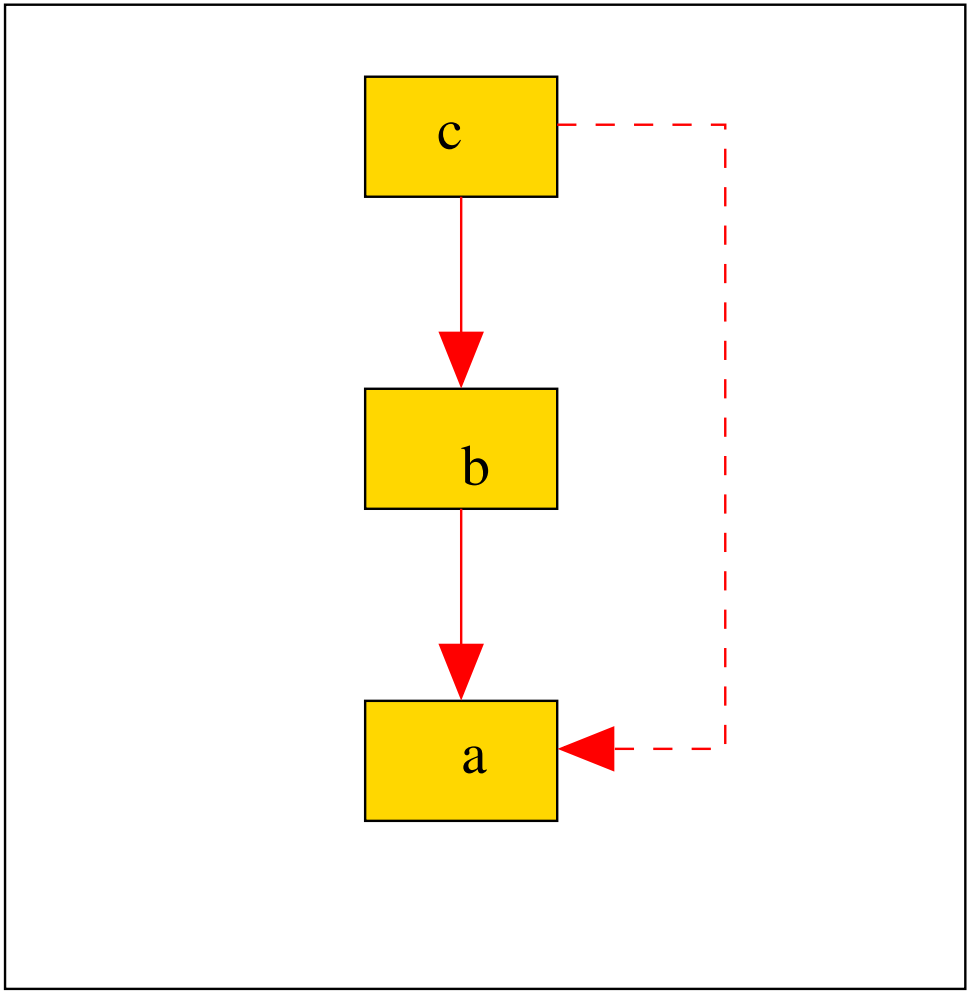
- Each diagram describes a universally quantified implication:
  - the solid arrows indicate assumed relationships,
  - the dashed arrows indicate implied relationships.
- The identifiers stand for arbitrary elements in the lattice.
- An arrow from b to a indicates that $a \sqsubseteq b$ holds (think of the arrow as a greater than sign).
- We use a double arrow to indicate equality of lattice elements.

Example: transitivity



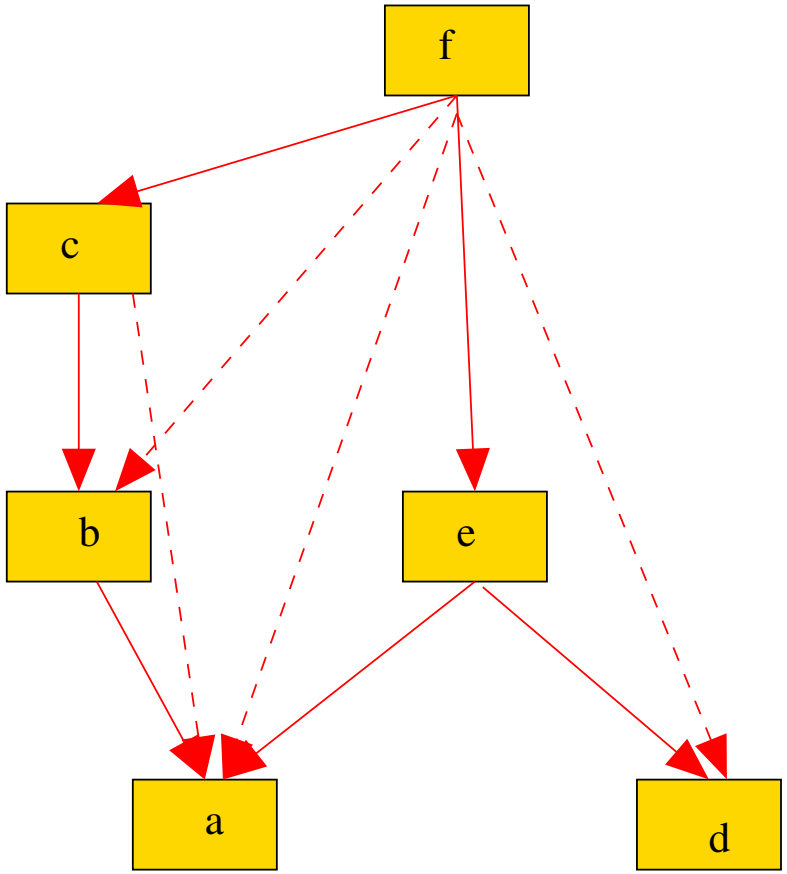
If we know that $a \sqsubseteq b$ and $b \sqsubseteq c$

Example: transitivity



then we may deduce that $a \sqsubseteq c$

Adding transitivity arrows



Intended use of refinement diagrams

Refinement diagrams are intended to model software components and their relationships

- the lattice elements are software parts,
- the ordering corresponds to *refinement* between software parts.

Intuitively, we can think of refinement as permission for replacement: $a \sqsubseteq b$ means that the part a may be replaced by part b in any context.

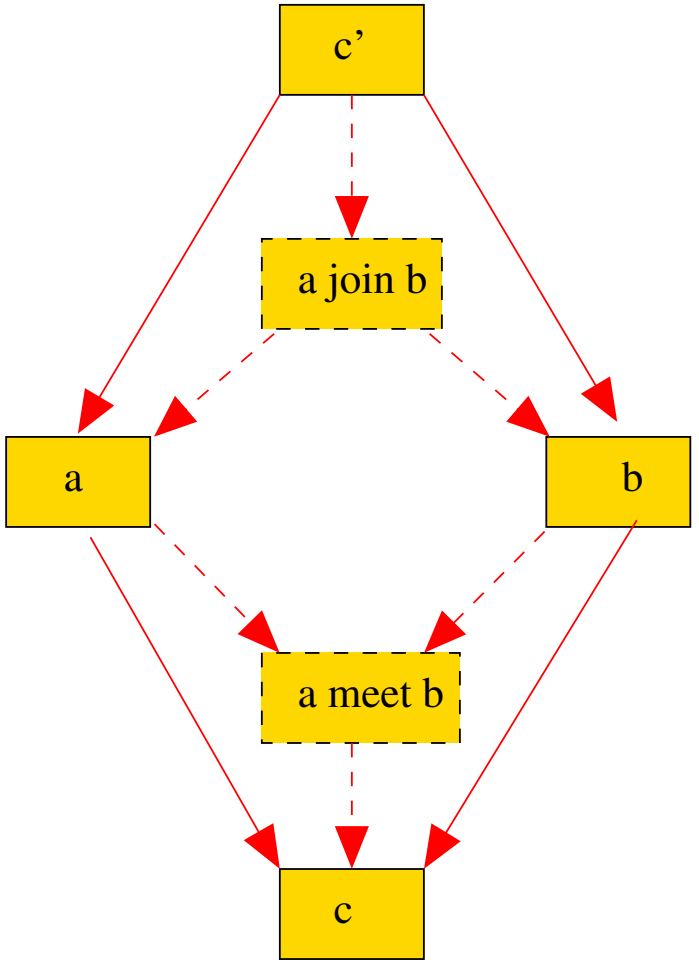
(The poset is generalized to a category, if we the arrows need to be labelled.)

Lattices

A poset is a *lattice*, if any two elements a and b in the lattice have a *least upper bound* (or *join*) $a \sqcup b$ and a *greatest lower bound* (or *meet*) $a \sqcap b$. A lattice is thus characterized by the following properties:

- $a \sqsubseteq a \sqcup b$ and $b \sqsubseteq a \sqcup b$ (*join is upper bound)*
- $a \sqsubseteq c \wedge b \sqsubseteq c \Rightarrow a \sqcup b \sqsubseteq c$ (*join is least upper bound)*
- $a \sqcap b \sqsubseteq a$ and $a \sqcap b \sqsubseteq b$ (*meet is lower bound)*
- $c \sqsubseteq a \wedge c \sqsubseteq b \Rightarrow c \sqsubseteq a \sqcap b$ (*meet is greatest lower bound)*

Refinement diagram rule



$a \sqsubseteq a \sqcup b$ and $b \sqsubseteq a \sqcup b$

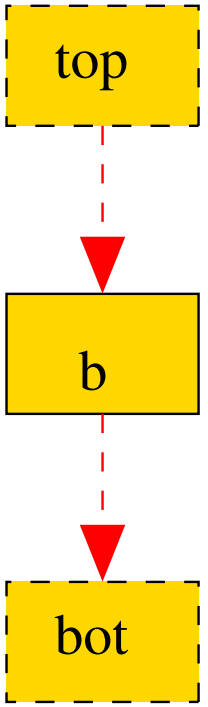
$a \sqsubseteq c \wedge b \sqsubseteq c \Rightarrow a \sqcup b \sqsubseteq c$

CREST
CENTRE FOR RELIABLE SOFTWARE TECHNOLOGY

Bounded lattice

A lattice is *bounded*, if there is a *least element* $\bot$ and a *greatest element* $\top$ in the lattice. This means that for any element b in the lattice, we have that

- $\bot \sqsubseteq a \sqsubseteq \top$ (*least and greatest element)*



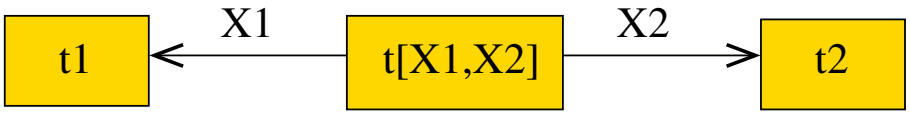
.

Complete lattices

- A lattice is *complete,* if any set of elements in the lattice has a least upper bound and a greatest lower bound.
- Any finite set will have this property in a lattice, but in an complete lattice also infinite sets and the empty set have a least upper bound and a greatest lower bound.
- In particular, we have that $\bot$ is then the least element of the whole lattice, while $\top$ is the greatest element of the whole lattice

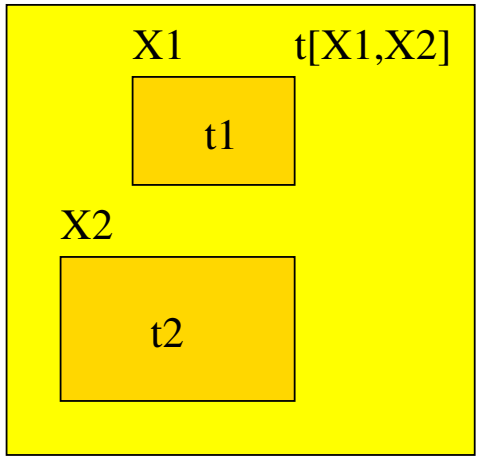
Terms and dependencies

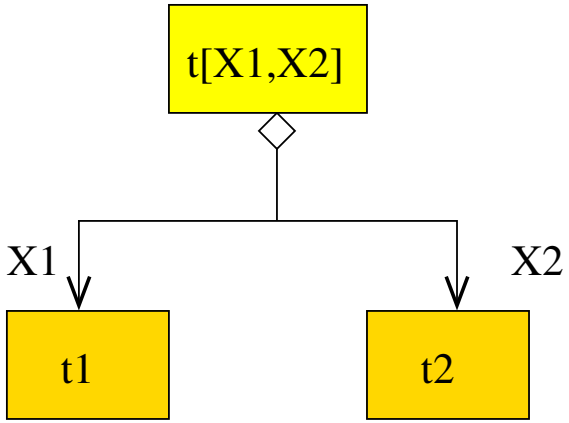
- A *lattice term* is constructed by applying lattice operations on lattice constants and variables over lattice elements.
- We write a lattice term as $t[X_1, \ldots, X_m]$ to indicate that it depends on the variables $X_1, \ldots, X_m$.
- In general, a box in a refinement diagram denotes a lattice term. We show a term as a box with dependency arrows, each arrow labeled with a lattice variable.
- Example: the term $t[t_1, t_2]$



Private subterms

The same term box with the subterms as nested boxes (left), or using aggregation as in UML (right):





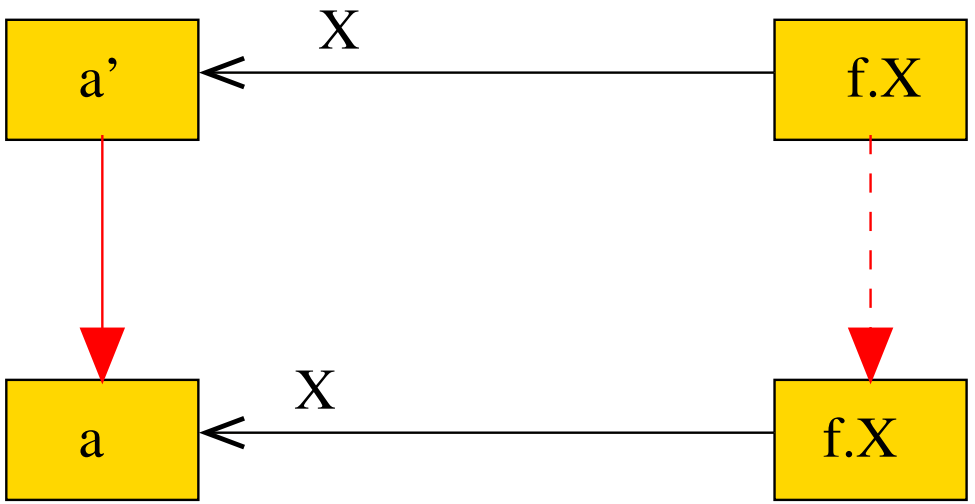
- Here the subterms are private to the enclosing box, in the previous representation they can be shared by other terms.

Monotonicity

A function $f : A \longrightarrow B$ from poset A to poset B is *monotonic,* if for any $a, a' \in A$

$$a \sqsubseteq_A a' \Rightarrow f.a \sqsubseteq_B f.a'$$

($f.x$ stands for the function application $f(x)$).



Monotonic terms

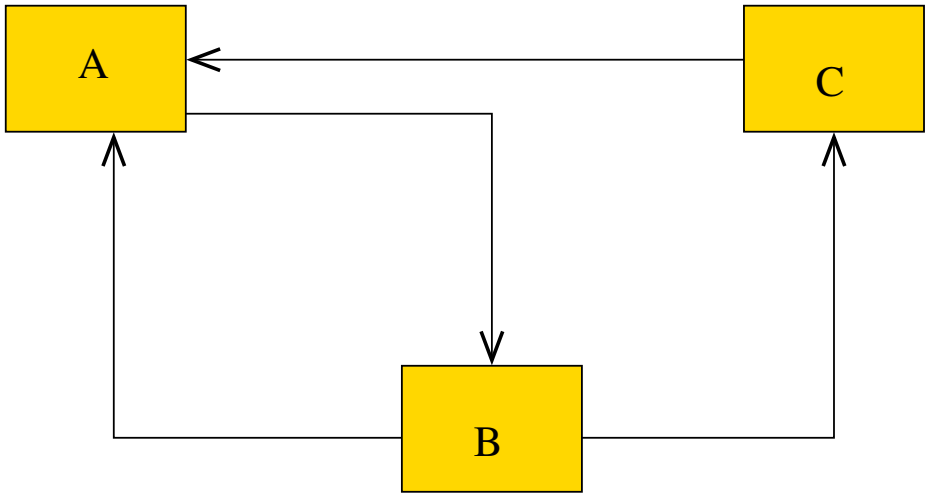
- Assume *constants* that denote specific lattice elements (like $\bot$ and $\top$).
- Assume a collection of monotonic functions (*operations*) on a given lattice.
- The composition of monotonic functions is also monotonic
- A lattice term is *monotonic*, if it is built out of constants and monotonic lattice functions.

RECURSION, ENVIRONMENTS AND SYSTEMS

- Recursion and fixed points
- Environments
- System as solutions of environment equation

Circular dependencies

Dependencies between terms in a diagram may be circular. Need a more elaborate notion of what a box denotes in such diagrams.

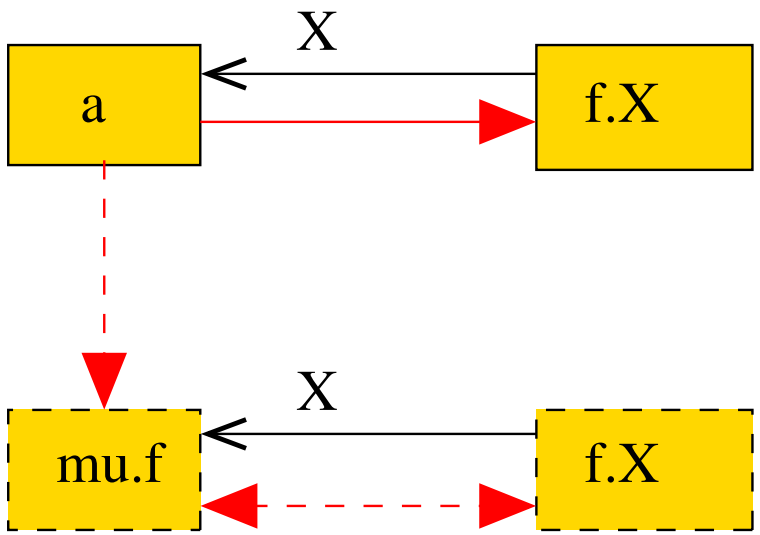


Circular dependencies are handled using *fixed points* of functions on lattices.

Least fixed point

A monotonic function $f : A \longrightarrow A$ on a complete lattice A has a unique *least fixed point,* denoted $\mu.f \in A$. The least fixed point has the following properties:

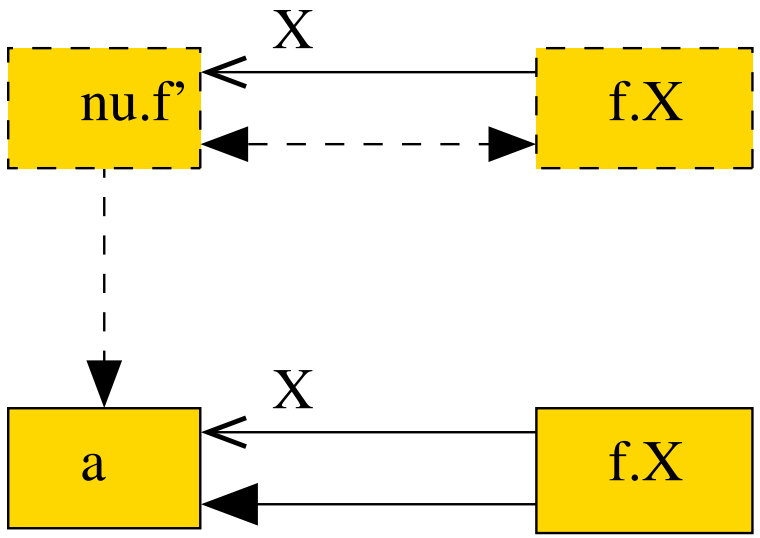
- $f.(\mu.f) = \mu.f$ ($\mu.f$ *is a fixed point)*
- $f.a \sqsubseteq a \Rightarrow \mu.f \sqsubseteq a$ (*least fixed point induction)*



Greatest fixed point

Similarly, there is also a unique *greatest fixed point* $\nu.f \in A$, which satisfies the following conditions

- $f.(\nu.f) = \nu.f$ ($\nu.f$ *is a fixed point)*
- $a \sqsubseteq f.a \Rightarrow a \sqsubseteq \nu.f$ (*greatest fixed point induction)*



Pointwise extension

- Consider the set of all monotonic functions from lattice A to lattice B. We denote this set by $A \rightarrow_m B$.

- The *pointwise extension* of the partial ordering on B to $A \rightarrow_m B$ is defined by

$$f \sqsubseteq f' \equiv (\forall a \in A \cdot f.a \sqsubseteq_B f'.a)$$

- The pointwise extension of a (complete) lattice is also a (complete) lattice.

Lattice products

- A special case of pointwise extension is *lattice product*.
- Let $A_1 \times \cdots \times A_m$ be the Cartesian product of lattices $A_1, \ldots, A_m$.
- Then we define a lattice ordering on $A_1 \times \cdots \times A_m$ by

$$(a_1, \ldots, a_m) \sqsubseteq (a'_1, \ldots, a'_m) \equiv a_1 \sqsubseteq a'_1 \wedge \ldots \wedge a_m \sqsubseteq a'_m$$

  This is a special case of the previous definition when each A_i denotes the same lattice B, and we choose $A = \{1, \ldots, m\}$.

- The product of a collection of (complete) lattices is a (complete) lattice.

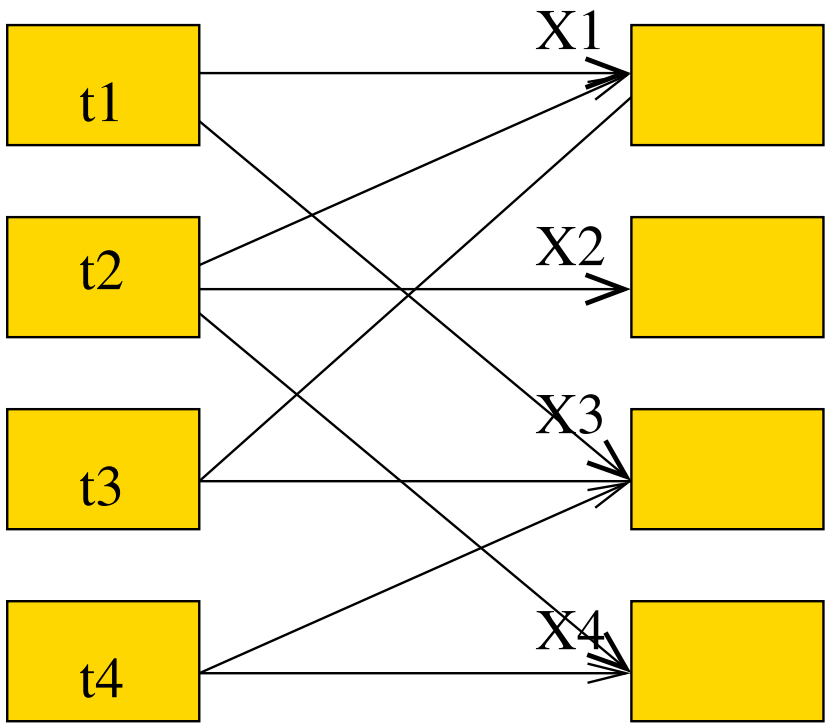
Monotonicity of fixed point operators

- Consider now the set $A \rightarrow_m A$.
- The least and greatest fixpoint operators are functions of type $\mu, \nu : (A \rightarrow_m A) \rightarrow A$.
- Both operators are monotonic with respect to lattice ordering, i.e., we have for any $f, g : A \rightarrow_m A$ that

$$f \sqsubseteq g \Rightarrow \mu.f \sqsubseteq \mu.g$$
$$f \sqsubseteq g \Rightarrow \nu.f \sqsubseteq \nu.g$$

Environments

- An *environment* is a tuple of monotonic lattice terms, $E = (t_1[X], \ldots, t_n[X])$
- Here $X = (X_1, \ldots, X_m)$ is a tuple of variables over lattice elements.



- Note that $t_i[X]$ has to use projection to access a specific variable in X. Projection is denoted by π_i^m, so $\pi_i^m.X = X_i$ for $i = 1, \ldots, m$.

Mutually recursive terms

- Consider the special case when $m = n$, i.e., $E = (t_1[X], \ldots, t_n[x])$ and $X = (X_1, \ldots, X_n)$.
- Then the function

$$\widetilde{E} = (\lambda X \cdot E)$$

  is a function of type $A^n \longrightarrow A^n$.

- This function is monotonic on the complete lattice $(A^n, \sqsubseteq)$.
- Hence, this function has a *least fixed point* $\mu E = \mu.\widetilde{E}$.

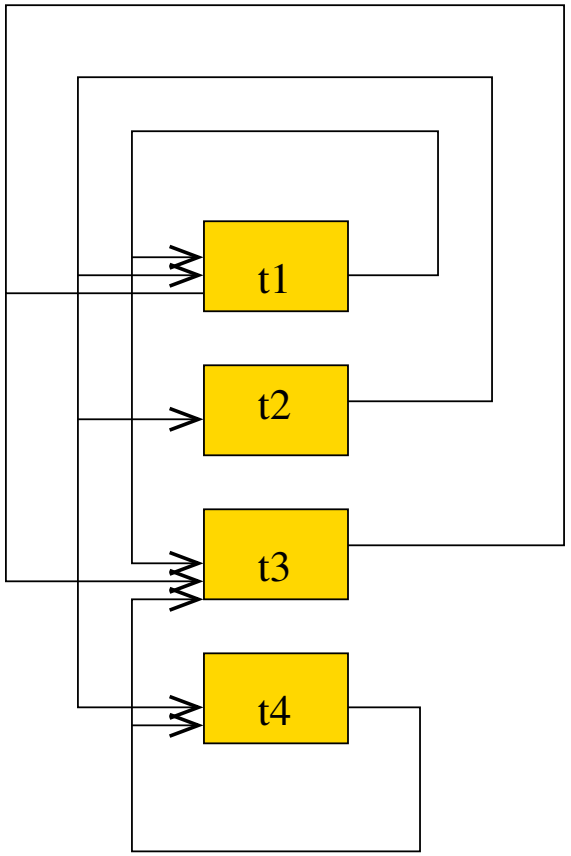
System defined by environment

- μE is the least solution to the equation

$$X = E$$

- We refer to μE as the *(least) system* defined by the environment E.

- We write $\mu E = (\mu E_1, \ldots, \mu E_n)$, i.e., μE_i is the ith lattice element in the system μE.



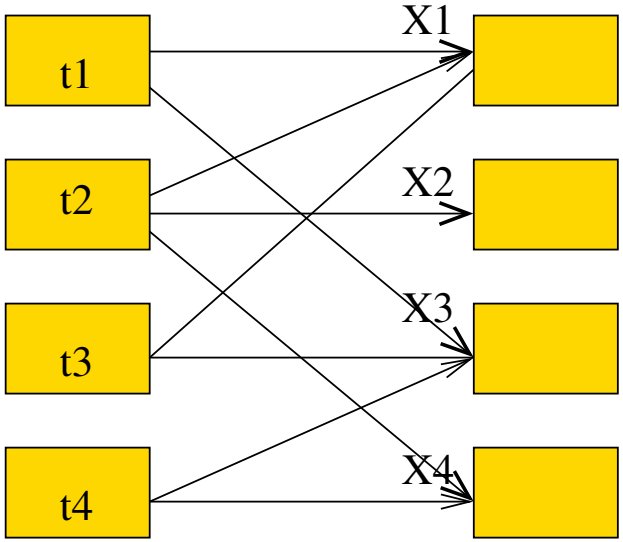
Unfolding

We can use unfolding to determine the meaning of an environment:

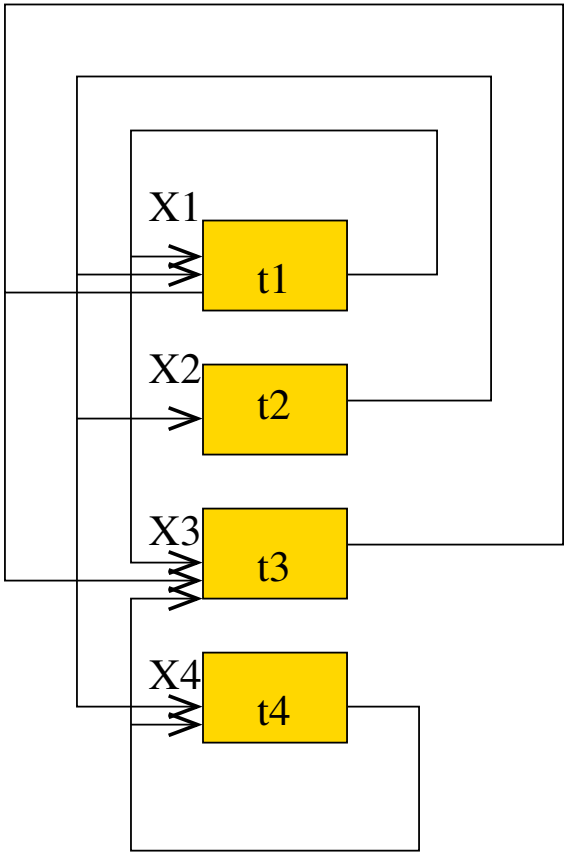
$$\mu E = (t_1[\mu E], \ldots, t_n[\mu E])$$

Intuitively, this means that the system μE is the infinite unfolding of the environment E .

Environment and system



Environment

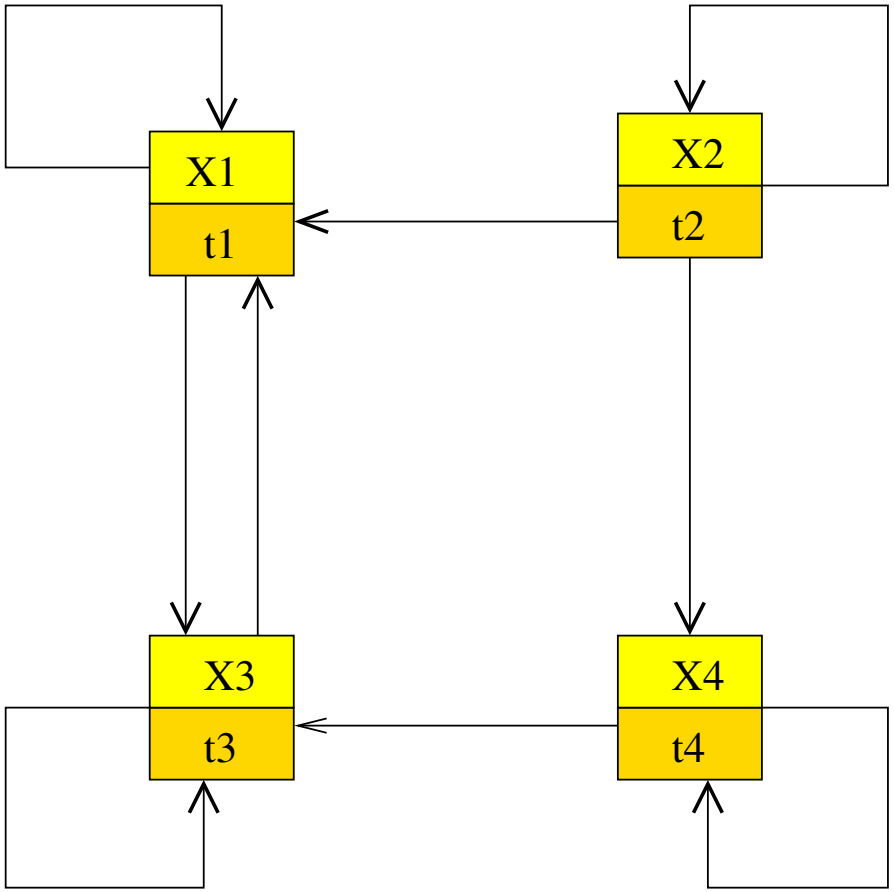


System

System equation

$$\begin{bmatrix} X_1 \\ X_2 \\ X_3 \\ X_4 \end{bmatrix} = \begin{bmatrix} t_1[X_1, X_3] \\ t_2[X_1, X_2, X_4] \\ t_3[X_1, X_3, X_4] \\ t_4[X_3, X_4] \end{bmatrix}$$

UML-like notation



Monotonicity of systems

We can define a lattice ordering on environments, by pointwise extensions:

$$E \sqsubseteq E' \equiv \widetilde{E} \sqsubseteq \widetilde{E'}$$

The monotonicity of the fixed point operator then gives us that

$$E \sqsubseteq E' \Rightarrow \mu E \sqsubseteq \mu E'$$

Refinement of complex systems

- This can be used to reason about refinement in a complex system.
- Assume that we have a term $t_i[X]$ in E, and that we make some small change to this term, to get $t_i'[X]$. The changed environment is E'.
- If this change is such that $t_i[X] \sqsubseteq t_i'[X]$, then $E \sqsubseteq E'$, so $\mu.E \sqsubseteq \mu.E'$
- This again means that $\mu E_i \sqsubseteq \mu E_i'$ holds. In other words, a refinement of one of the terms in the environment will result in a refinement of the meaning of each term in the system.

DIAGRAMMATIC REASONING

Refinement diagrams provide a way of reasoning in a visual way about lattice elements. We have three main ingredients here:

Refinement diagrams: describe a collection of lattice elements and how they are ordered

Refinement diagram rules: describe how to add new entities and ordering relations to a refinement diagram

Refinement diagram derivations: also records the order in which the entities and ordering relations have been introduced.

Refinement diagrams

1. The entities of the diagram are
   (a) lattice terms (shown as rectangels, but other graphical notation is also possible),
   (b) orderings (shown as refinement arrows) and equalities (shown as double arrows) between lattice elements, and
   (c) dependency relations (shown as usage arrows).

2. The lattice terms in a diagram form an environment E: each occurrence of a lattice term in the diagram is an element in the environment tuple.

3. The same lattice term can occur in two or more places in the diagram, but each occurrence corresponds to a separate element in the tuple.

4. The *meaning* of a term t_i in the diagram (environment) E is the system element μE_i.

5. An ordering indicated in the diagram holds between the meanings of the terms in the environment (not the terms themselves). The arrow goes from the larger to the smaller element.

6. A name X_i for a box with term t_i can be understood as the equation $X_i = t_i$. The collection of all such equations determine the system defined by the diagram.

7. A name associated with a dependency arrow must be one of the free variables in the term that is the source of the arrow. (The variables in the terms can be local names for terms, which are bound to the actual term by the dependency arrow).

Remarks

- May annotate entities. For instance, for software components the
  - refinement arrows would in general be annotated by an *abstraction function* (or relation) to determine the *data refinement* between the components.
- Nested boxes are interpreted as a dependency of the enclosing box on the inner box.
- Conventions (4) and (5) can be explicated as follows. Assume

$$E = (t_1[X], \ldots, t_n[X)$$

where the indexes $1, \ldots, n$ identify the different occurrences of terms in the refinement diagram that describes E. A refinement arrow from box i to box j means that

$$\mu E_j \sqsubseteq \mu E_i$$

where

$$\mu E_i = \pi_i^m(\mu.(\lambda X \cdot E[X]))$$

.

Refinement diagram rule

1. A rule gives a permission to add some new terms and ordering relations to a refinement diagram.

2. The elements and ordering relations that must be present are shown with solid lines.

3. The elements and ordering relations that may be added are shown with dashed lines.

Refinement diagram derivation

1. A refinement diagram derivation is a refinement diagram where the ordering relations are numbered by consecutive integers that show the order in which the relations have been introduced in the diagram.

2. With each number we associate a proof rule that justifies the introduction of this arrow, together with a possible side conditions that must hold for this inference to be valid.

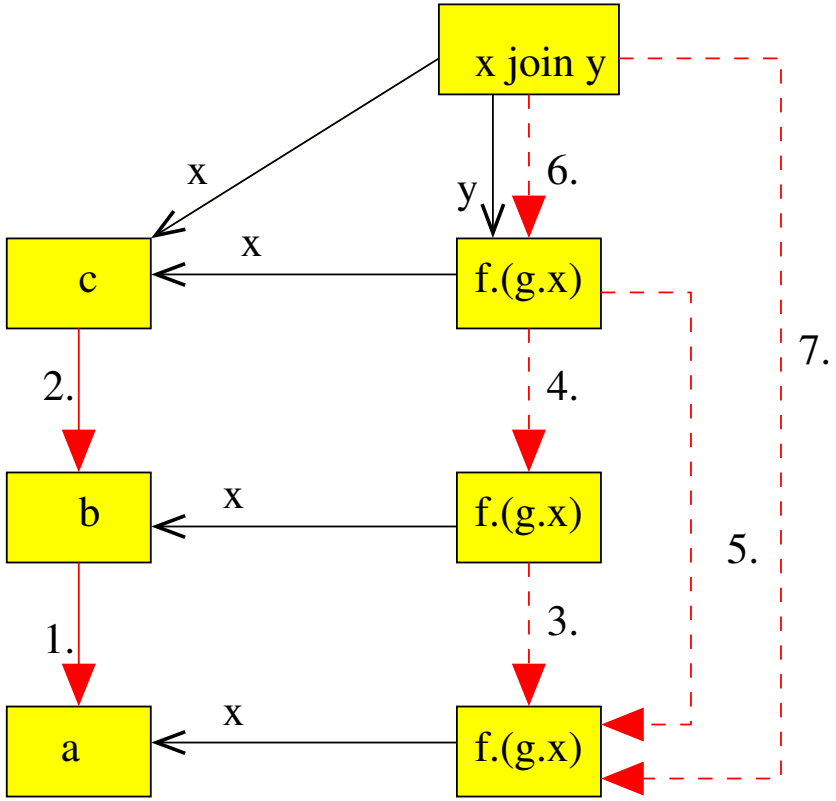
3. New entities may only be introduced into the diagram if justified by some proof rule.

4. No entities may ever be removed from the diagram.

5. The proof rules used in the diagram can be textual proof rules, or they can be refinement diagram rules.

6. Refinement diagrams provide a *construction logic* for constructing complicated lattice terms with specific properties.

Example



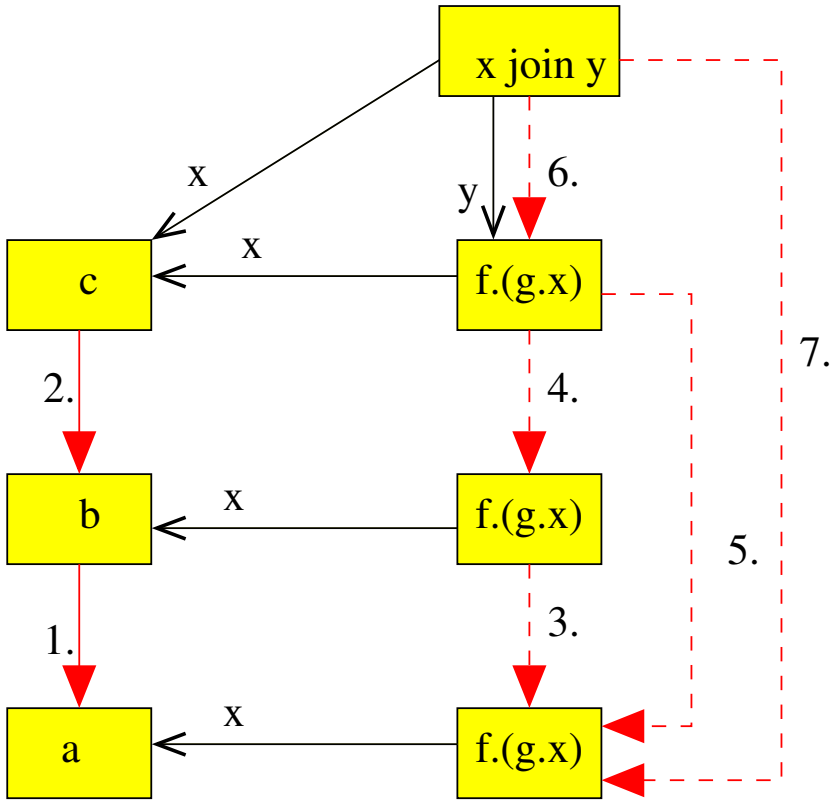
- Dashes show which parts had to be assumed and which parts could be inferred by inference rules.

Refinement diagram derivations and Hilbert-like proofs

- There is an equivalent textual presentation of the refinement diagram derivation, in the form of a Hilbert-like proof in a lattice theory.
- In this textual proof, each step is numbered, and is either justified
  - as an axiom,
  - as an assumption or
  - as an inference drawn from some previous steps using an inference rule.

Example as Hilbert like proof

1. $a \sqsubseteq b$ (assumption)
2. $b \sqsubseteq c$ (assumption)
3. $f.(g.a) \sqsubseteq f.(g.b))$ (mon. 1)
4. $f.(g.b) \sqsubseteq f.(g.c)$ (mon.2)
5. $f.(g.a) \sqsubseteq f.(g.c)$ (trans.3,4)
6. $f.(g.c) \sqsubseteq c \sqcup f.(g.c)$ (lub prop)
7. $f.(g.a) \sqsubseteq c \sqcup f.(g.c$ (trans. 5,6)



REFINEMENT CALCULUS

- Predicate transformers
- Refinement relation as lattice ordering
- Refinement calculus hierarchy
- Reasoning in the hierarchy

Background

- How to describe software in terms of lattices
- Base our approach on the *refinement calculus*
  - Back Ph.D. thesis, MC tract, articles 1978 - 1982
  - further developed by Back, Morgan and others 1988 –
  - presentation here based on book by Back & von Wright 1998.
- Refinement calculus is in turn based on the *weakest precondition approach* by Dijkstra 1976 –

Predicate transformers

A *predicate* is a property of a state. Hence, we can identify a predicate with a set of states.

A *predicate transformer* maps predicates to predicates.

A predicate transformer can be understood as the semantics of a program statement [Dijkstra]:

- For any programming language statement S, we define its meaning $wp.S$ as a predicate transformer.
- $wp.S$ determined for any predicate (*postcondition*) q on the state space another predicate (*precondition)* $wp.S.q$
- $wp.S.q$ is the *weakest precondition* for statement S to terminate in a state satisfying q.

Example statements

$abort$	can fail to terminate in any initial state: $wp.abort.q = false$ for any q
$magic$	is guaranteed to always terminate and establish any postcondition: $wp.magic.q = true$ for any q
$skip$	does not change the state at all: $wp.skip.q = q$
$x := e$	an assignment statement: $wp.(x := e).q = q[x := e]$
$S_1; S_2$	sequential composition: $wp.(S_1; S_2).q = wp.S_1.(wp.S_2.q)$

$(b \rightarrow S_1|S_2)$ conditional composition:

$$wp.(b \rightarrow S_1|S_2).q = (b \cap wp.S_1.q) \cup (\neg b \cap wp.S_2.q)$$

Refinement ordering

Refinement $S \sqsubseteq T$ says that

any postcondition that S can establish can also be established by T .

In this sense T is better than S (or at least as good as S) . We define

$$S \sqsubseteq T \equiv (\forall q \cdot wp.S \subseteq wp.T)$$

- Intuitively, this means that any user of the statement S who only is interested in the functional properties of this statement, should not notice any difference if S is replaced by T.
- The monotonic predicate transformers form a complete lattice with the refinement ordering.

Refinement lattice

- The meet $S \sqcap T$ is the *demonic choice* between executing S or executing T.
  - One of these alternatives is chosen, but we have no influence of which alternative that is chosen.
- The join $S \sqcup T$ is the *angelic choice* between executing S or executing T.
  - We can choose the alternative that suits our purpose better

Refinement calculus interpretation

- The refinement calculus interprets software systems as elements in a lattice (of, e.g., predicate transformers).
- Simpler lattice interpretations also possible: e.g., consider program statements as relations on the state space.
- More complicated lattice interpretations used to, e.g., model classes in object oriented systems, or interactive systems, real-time systems or concurrent systems.
- In fact, one often forces the semantics into a lattice framework, in order to handle recursion with fixed points.

A theory of program parts

- We will therefore postulate here that a software system can be understood as the system defined by an environment on a lattice, as explained above.
- The terms can be seen as the collection of *parts* of the system.
- A part can *depend on* (or *use)* other parts.
- A part can be *refined* by another part, in the sense that any user of this part does not see the difference if that part is replaced with the refining part.

- We can think of parts as real physical entities, like machine parts or building components or similar things.

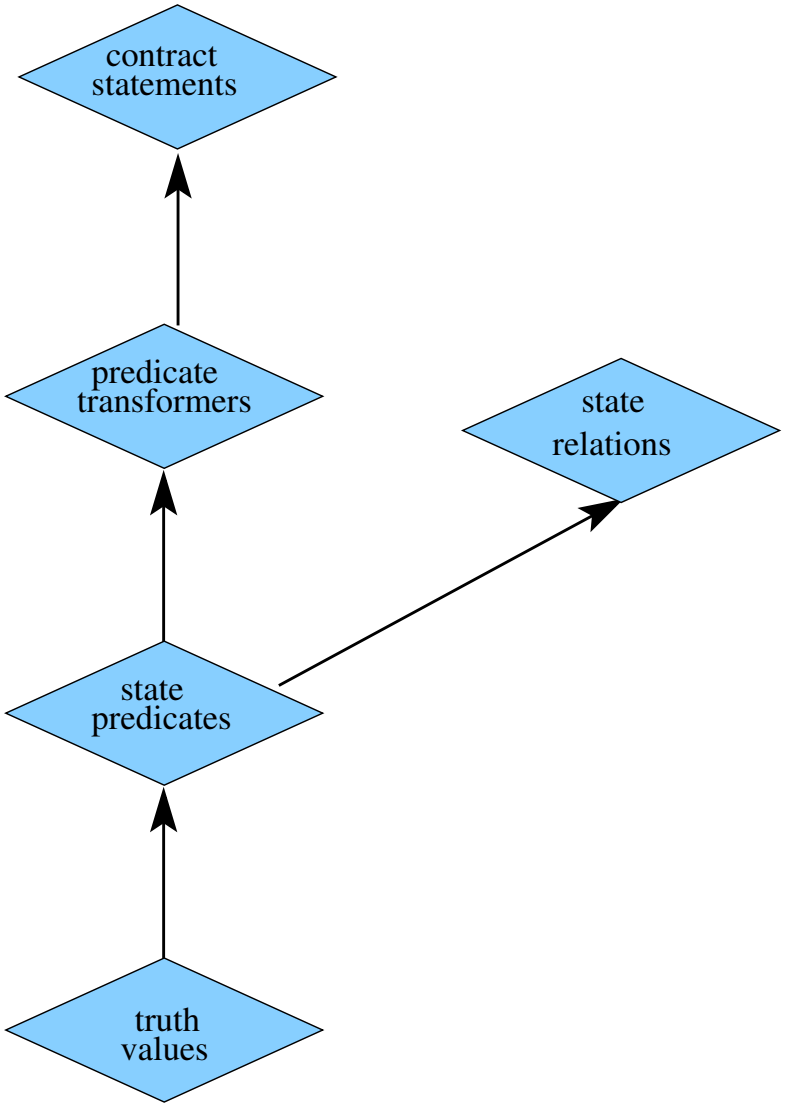
- We can also think of parts as software components, like procedures, functions, expressions, classes, modules, libraries, or packages. The latter interpretation is the one that we are here primarily interested in.

The refinement calculus hierarchy

A hierarchy of complete lattices that allow sone to reason about complex software systems at different level of detail.

The hierarchy is built on top of an arbitrary collection of state spaces Σ, Γ, and collection of *agents* Ω (for contracts).

Check Back & von Wright 1998 for details.



Truth value lattice

- The *truth value lattice*

$$Bool = \{T, F\}$$

.

- The ordering is *implication*, i.e., $b \sqsubseteq b' \equiv (b \Rightarrow b')$.
- The smallest element is falsity F and the largest element is truth T.
- Meet is defined by $a \sqcap b = a \wedge b$ and join is defined by $a \sqcup b = a \vee b$.

State predicate lattice

- The *state predicate (or subset) lattice*

$$Pred(\Sigma) = \Sigma \rightarrow Bool$$

.

- The ordering is *subset inclusion*, $p \sqsubseteq q \equiv p \subseteq q$.
- The smallest element is the *universally false predicate* $false = \emptyset$ and the largest element is the universally true predicate $true = \Sigma$.
- Meet is intersection, $p \sqcap q = p \cap$q, and join is union, $p \sqcup q = p \cup q$.

State relation lattice

- The *state relation lattice*

$$Rel(\Sigma, \Gamma) = \Sigma \rightarrow Pred(\Gamma)$$

.

- The ordering is relational inclusion, $P \sqsubseteq Q \equiv P \subseteq Q$.
- The smallest element is the universally false (empty) relation $False = \emptyset$ and the largest relation is the true (universal) relation $True = \Sigma \times \Gamma$.
- Meet is intersection of relations and join is union of relations.

Predicate transformer lattice

- The *predicate transformer lattice*

$$Mtran(\Sigma, \Gamma) = Pred(\Gamma) \rightarrow_m Pred(\Sigma)$$

- The ordering is *refinement*, defined by $S \sqsubseteq T \equiv (\forall q \cdot S.q \subseteq T.q)$.
- The least element is the predicate transformer $abort = (\lambda q \cdot false)$and the greatest element is the predicate transformer $magic = (\lambda q \cdot true)$.
- Meet is the predicate transformer $S \sqcap T = (\lambda q \cdot S.q \cap T.q)$ and join is the predicate transformer $S \sqcup T = (\lambda q \cdot S.q \cup T.q)$.

Contracts lattice

- The *contracts lattice*

$$Cont(\Omega, \Sigma, \Gamma) = Pred(\Omega) \rightarrow Mtran(\Sigma, \Gamma)$$

.

- The ordering is $F \sqsubseteq G \equiv (\forall c \cdot F.c \sqsubseteq G.c)$.
- The least element is $Abort = (\lambda c \cdot abort)$ and the greatest element is $Magic = (\lambda c \cdot magic)$.
- Meet is defined by $F \sqcap G = (\lambda c \cdot F.c \sqcap G.c)$ and join is defined by $F \sqcup G = (\lambda c \cdot F.c \sqcup G.c)$.

- Contracts model systems where a number of independent agents with possibly conflicting goals participate in making decisions about how the system should behave.

Pointwise extension

- The implication ordering of truth values forms the basis for the refinement calculus hierarchy. The other lattice orderings are defined by pointwise extension of lower orderings.
- Subset inclusion is the pointwise extension of implication, and relation inclusion is the pointwise extension of subset inclusion.
- Refinement is also the pointwise extension of subset inclusion, to a different domain than relations.
- Finally, contract ordering is the pointwise extension of the refinement ordering.

- The refinement calculus hierarchy also contains other, more exotic lattices, which we will not describe in detail here.

Other operations in the lattices

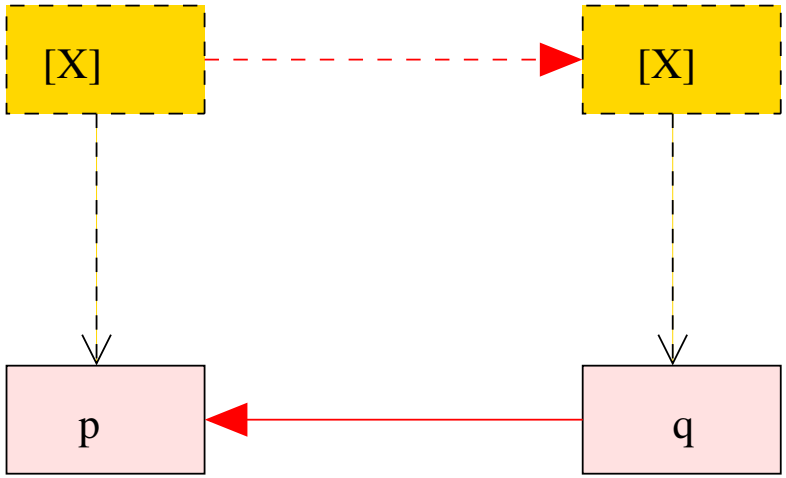
- The refinement calculus hierarchy contains, in addition to the lattice operations, also other operations that are defined on these domains.
- In particular, we usually need some operation for sequential composition of relations and predicate transformers.
- In addition, there are a number of homomorphic embeddings between the lattices in the hierarchy.

Reasoning in the refinement calculus hierarchy

- Reasoning with refinement diagrams in the refinement calculus hierarchy typically involves reasoning on different levels in the hierarchy simultaneously.
- It is possible to show reasoning in different lattices in the same refinement diagram.
- Transfers between different levels in the hierarchy are achieved by using homomorphic embeddings of a lower level lattice in a higher level lattice.

Example: assume statement

- assume statement $[p]$ models a guard
- $wp.[p].q = \neg p \cup q$
- we have that $p \subseteq q \Rightarrow [q] \sqsubseteq [p]$



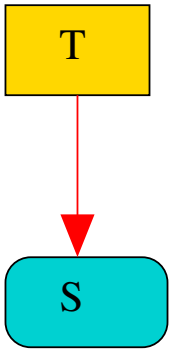
COMPONENTS AND SPECIFICATIONS

- Specifications and implementation of components
- Recursive components
- Information hiding

Specifications and implementations

- A *specification* is a description of a the functional (and sometimes also non-functional) behavior of a software component. It describes *what* the component does, but not *how* it does it.
- An *implementation* is a software component that realises the functional behavior described by the specification
- We consider a specification to be a part in a software system, in the same way that an an implementation is a part in a software system
- We assume that there is a (possibly idealized) sense in which the specification can be executed

Satisfying a specification



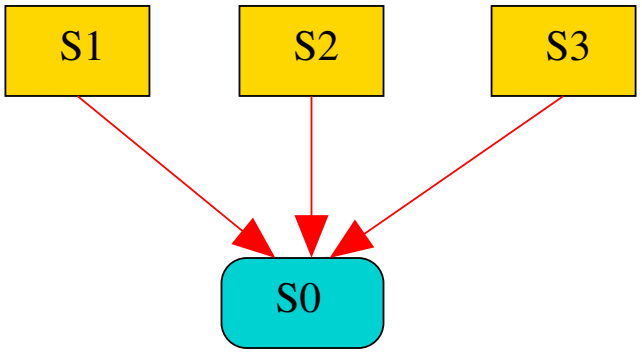
- A specification S is satisfied by an implementation T, if $S \sqsubseteq T$
- Intuitively, this means that we are allowed to replace the specification S by the implementation T in any context.
- We indicated specifications by rounded boxes to emphasize the intended use of these components.
- (In practice, one would often have *data refinement* between the components rather than simple algorithmic refinement.)

Multiple implementations

A specification S_0 can be satisfied by more than one implementation,

$S_0 \sqsubseteq S_1$, $S_0 \sqsubseteq S_2$, , $S_0 \sqsubseteq S_3$

- S_0 could be a *standard* for some component, and S_1,S_2, S_3 could be different implementations of this standard which are provided by different vendors.

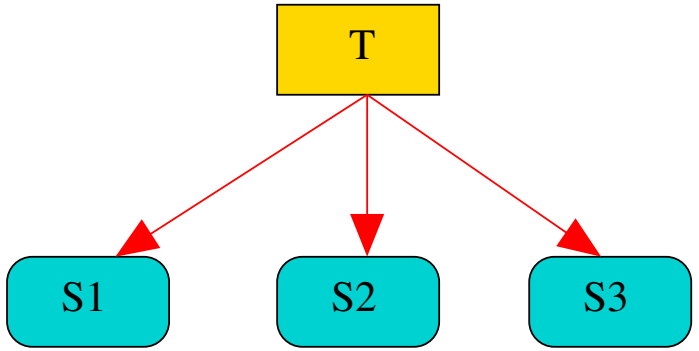


Multiple interfaces

An implementation can also satisfy more than one specification, $S_1 \sqsubseteq T$, , $S_2 \sqsubseteq T$, , $S_3 \sqsubseteq T$ etc.

- Then we often talk about multiple *interfaces* to the same software component.

- A banking application may provide one interface for the bank customer and another interface for the bank clerk.

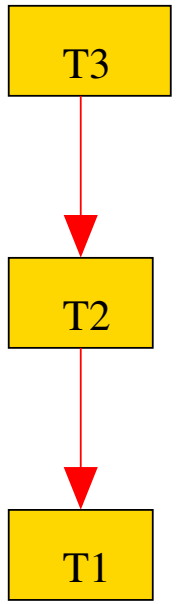


Refining implementations

It is also possible that an implementation T_1 is seen as a specification of another implementation T_2, in which case we require $T_1 \sqsubseteq T_2$.

For instance,

- T_2 could be a more efficient implementation of T_1,
- T_2 could be an adaptation of T_1 to a different platform, or
- T_2 could be the object code of the source code component T_1.

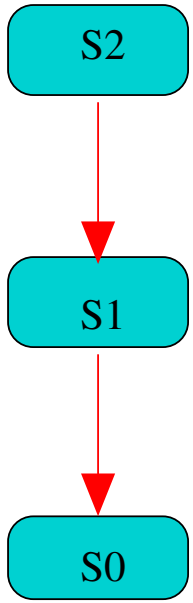


Stepwise refinement is based on this idea.

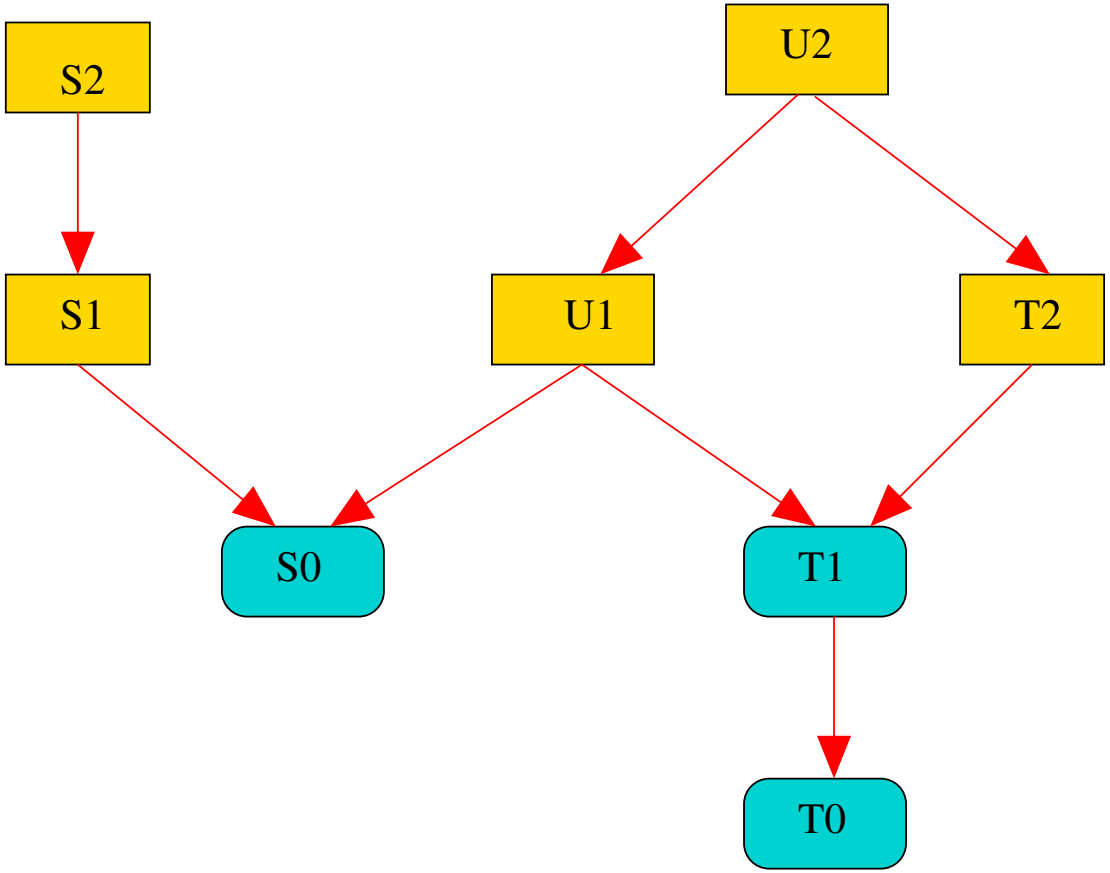
Refining specifications

It is also possible that we have refinement between specifications, $S_1 \sqsubseteq S_2$.

- we add functionality to a specification, or
- we add further detail to the specification.



Specifications and implementations

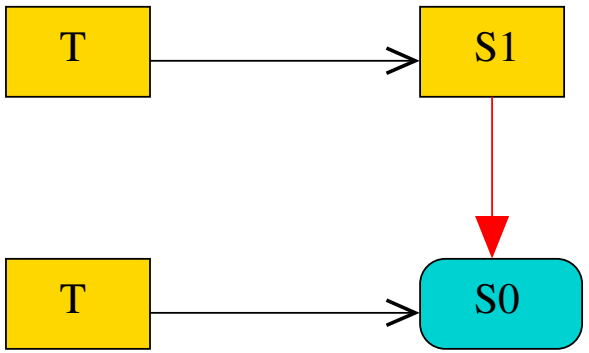


Modularity and information hiding

- Specifications allow us to *modularize* software systems.
- If a component only knows about the specifications of other components, then the implementation of a used component can be changed at will, as long as it still satisfies the original specification.
- This *information hiding* technique is a powerful technique for building loosely coupled systems
- It allows us to build different parts of the system independently (e.g., by different people or at different times), as long as we do not change the specifications of the parts in the system.

Specifications and correctness

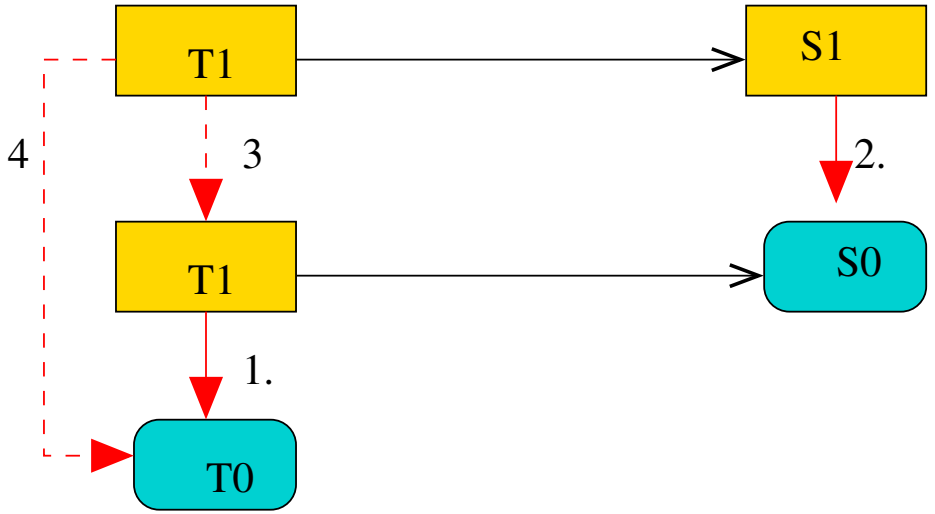
- Specifications are also important for verifying that a software system is correct.
- A specification S_0 is usually more abstract and simpler to reason about than an implementation S_1.
- If T is another component that depends on the S component, $T[S_1]$ is likely to be a much more complex term than $T[S_0]$, and hence much more difficult to reason about.



Constructing systems with specifications

- We have a specification T_0 of a part that we want to build
- We want to implement this with a part T_1 that uses another part S_1
- The implementation must be correct, i.e., $T_0 \sqsubseteq T_1[S_1]$ must hold.

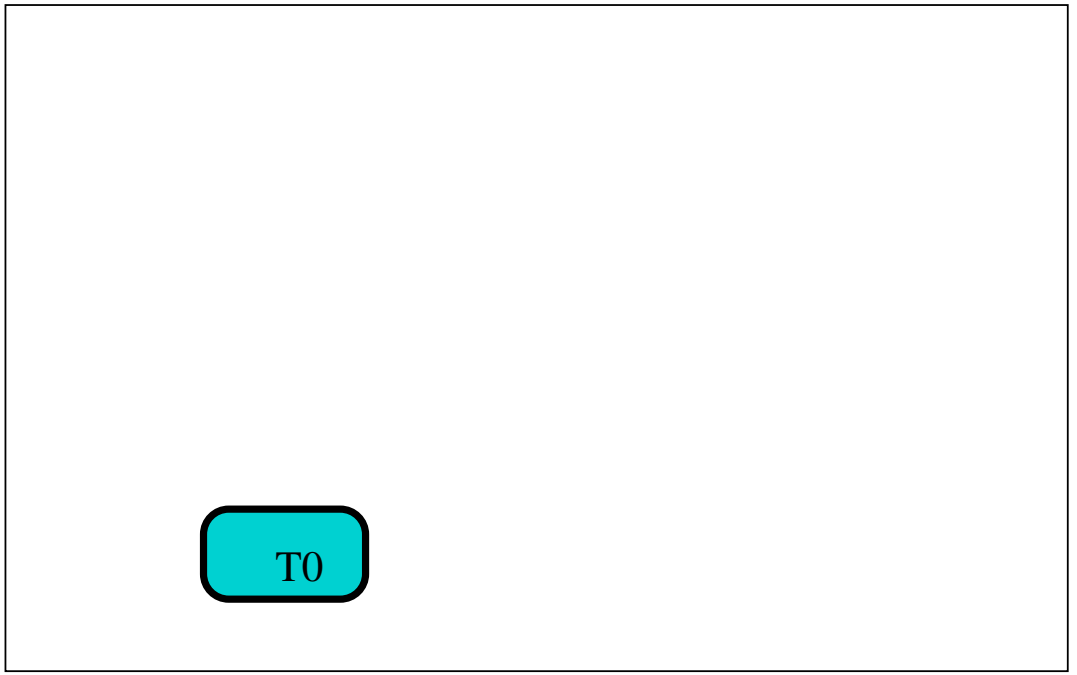
Refinement diagram derivation



- The proof shows that we have used a specification S_0 of S_1 to make it easier to check the correctness of the constructed system.

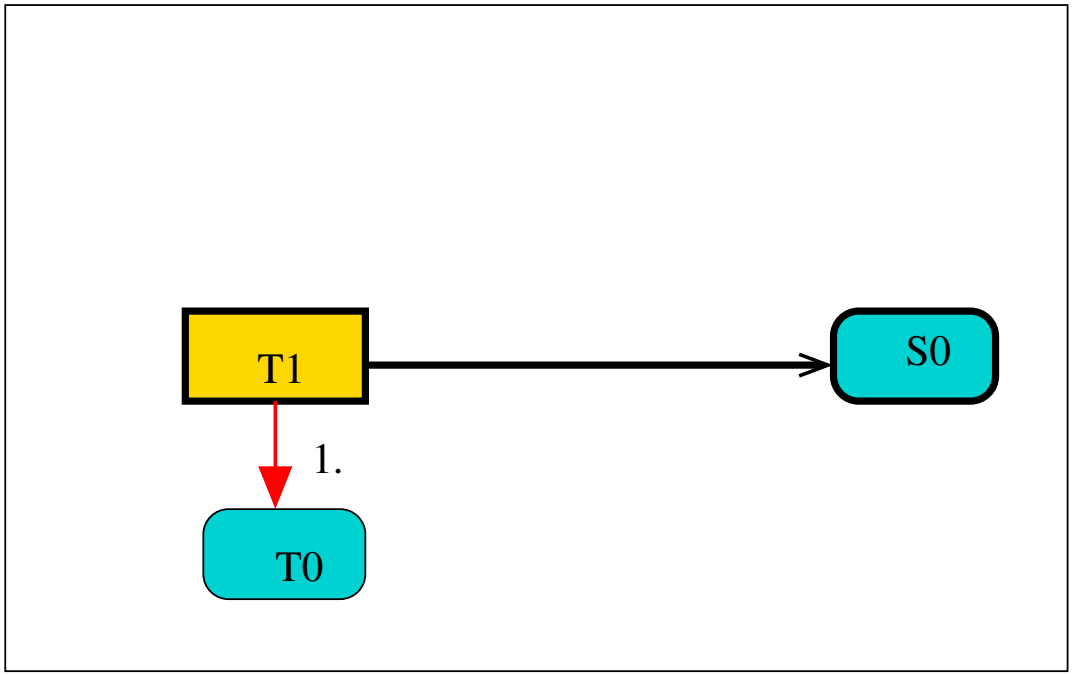
Animation of construction, step 0

Initially only the specification T_0 is provided



Step 1

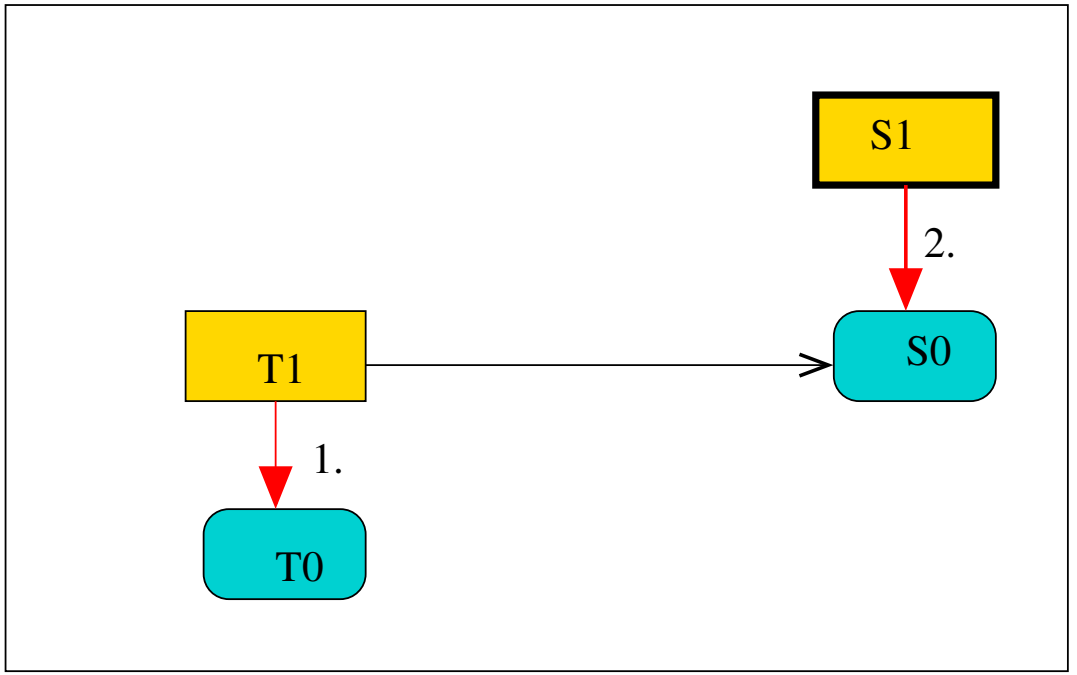
We provide the specification of an auxiliary part S_0 and an implementation $T_1[S_0]$ of T_0. We show that this is a correct implementation.



Step 2

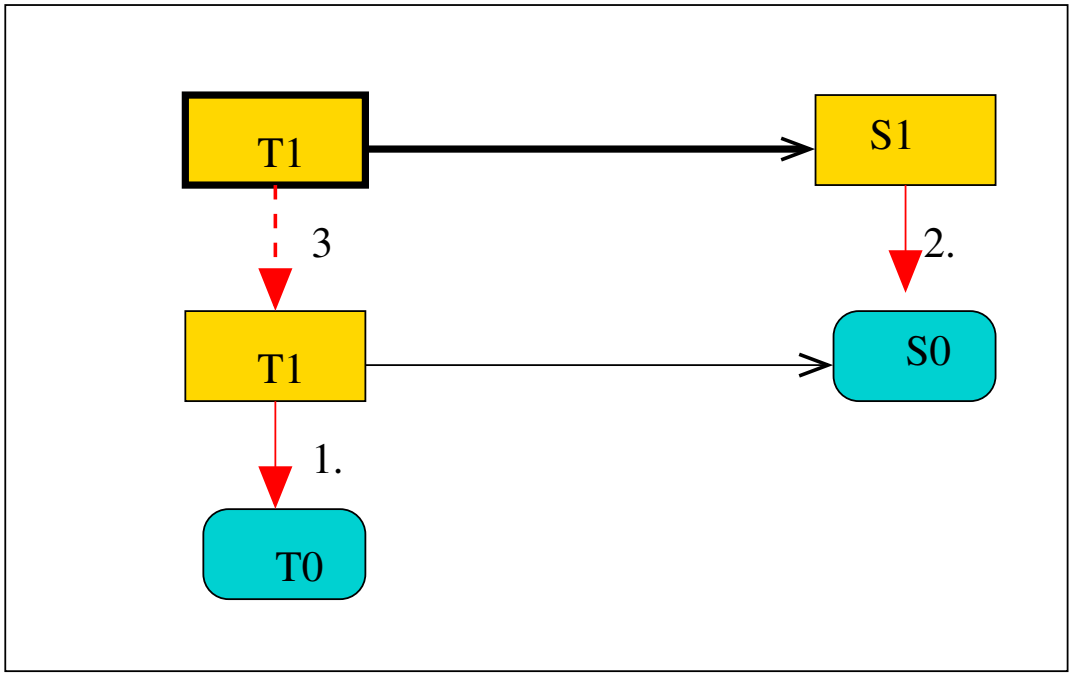
We then provide an implementation S_1 of S_0

We prove that this implementation satisfies the specification S_0.



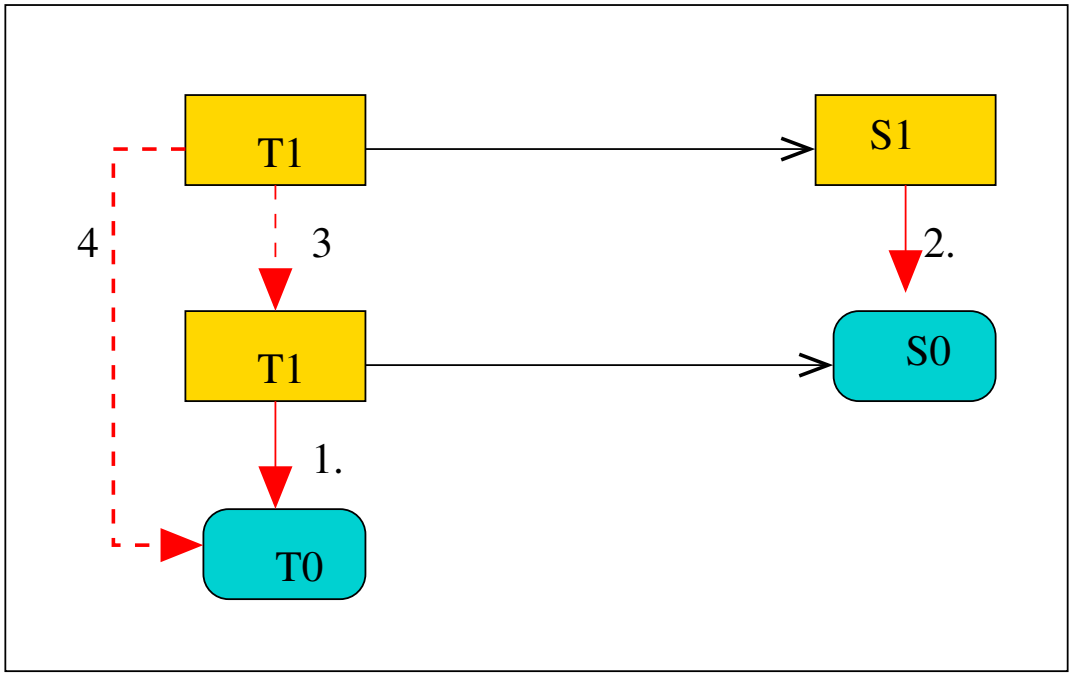
Step 3

We redirect T_1 to use the implementation S_1 rather than the specification S_0.
This is a correct refinement of the previous version of T_1 (by monotonicity).



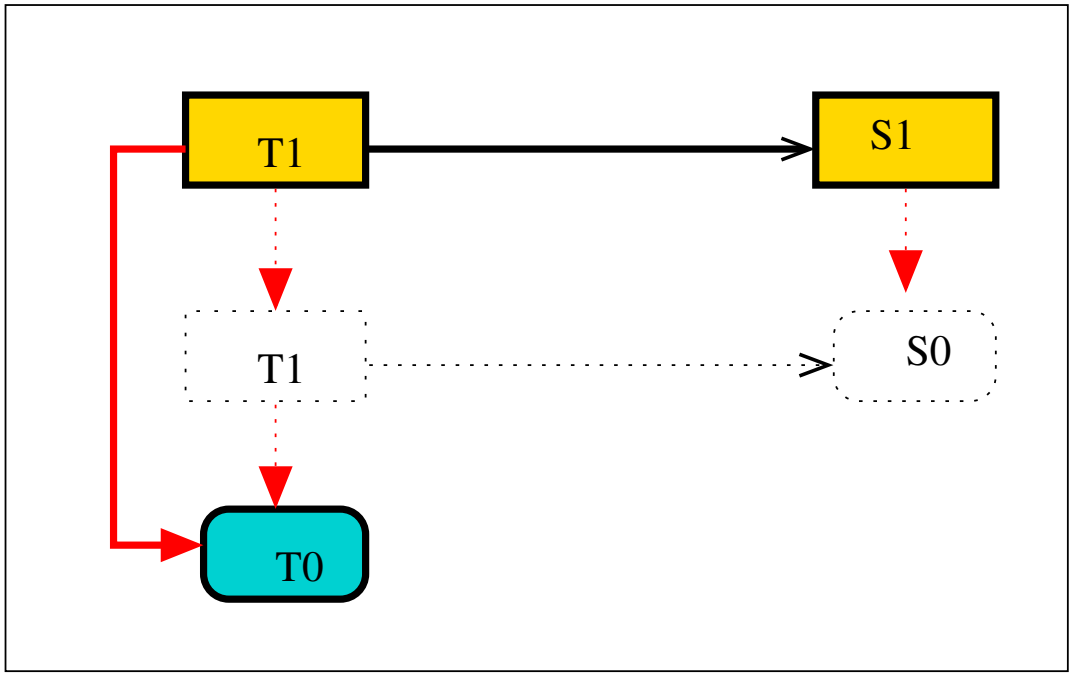
Step 4

We now have a correct implementation $T_1[S_1]$ of the original specification T_0(by transitivity).



Hiding intermediate steps

The specification S_0 and the previous version of T_1 that used S_0 are now obsolete, so we can ignore them in future steps



Construction as a log

The verbal explanations above can be seen as a log of the software construction process:

1. We provide the specification of an auxiliary part S_0 and an implementation $T_1[S_0]$ of T_0. We show that this is a correct implementation.

2. We then provide an implementation S_1 of S_0, and prove that this implementation satisfies the specification S_0.

3. We redirect T_1 to use the implementation S_1 rather than the specification S_0. This is a correct refinement of the previous version of T_1 which used the specification S_0 (by monotonicity).

4. Finally, we notice that we now have a correct implementation $T_1[S_1]$ of the original specification T_0(by transitivity). The specification S_0 and the previous version of T_1 that used S_0 are now obsolete, so we can forget about them.

Construction as Hilbert-like proof

We can also express the construction as a proof in lattice theory:

1. $T_0 \sqsubseteq T_1[S_0]$ (assumption or lemma)
2. $S_0 \sqsubseteq S_1$ (assumption or lemma)
3. $T_1[S_0] \sqsubseteq T_1[S_1]$ (by monotonicity from 2)
4. $T_0 \sqsubseteq T_1[S_1]$ (by transitivity from 1,3)

Three different presentations

We thus have three different ways of describing the same construction:

- a diagrammatic way based on refinement diagrams (intuitive, visual overview)
- a software process log describing the successive development steps (shows and explains the actions taken)
- a formal proof in the refinement calculus (mathematical, precise)

Software construction in the large and small

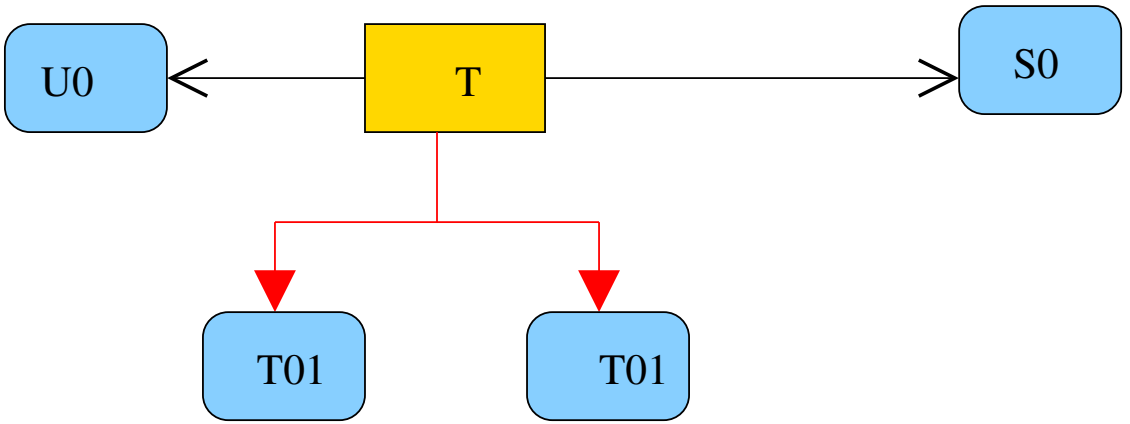
- Refinement diagrams used for software construction “in the large”
- Justification for each step may require quite a lot of work, and amount to software construction “in the small”
- “assumption” or “lemma” as justification in the proof indicate that these steps may have been established in a different proof framework, and are here taken as lemmas or assumptions.

Alternative formalization

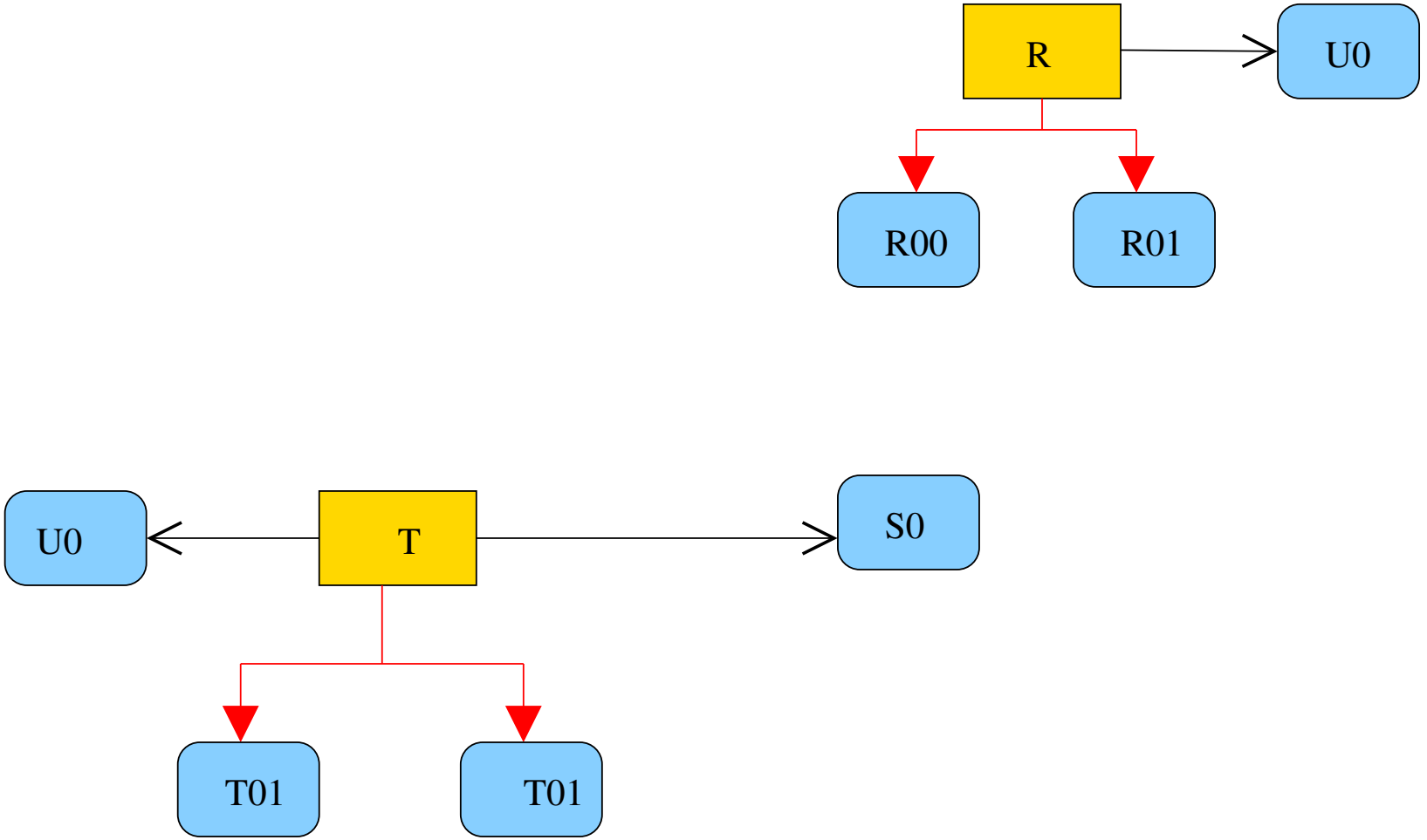
- Above formalization assumes that the terms describing the software parts are always well formed and internally consistent.
- We can emphasize the construction of a well formed and consistent software part by introducing a separate judgment for this, e.g., $\vdash S$, that states that S *is consistent*.
- Then the diagrammatic proof and the corresponding Hilbert like proof will have two kinds of judgements, $\vdash S$ and $\vdash S \sqsubseteq T$
- Paradigm: first *construct* a consistent part and then c*heck* that it satisfies its requirements.

Components

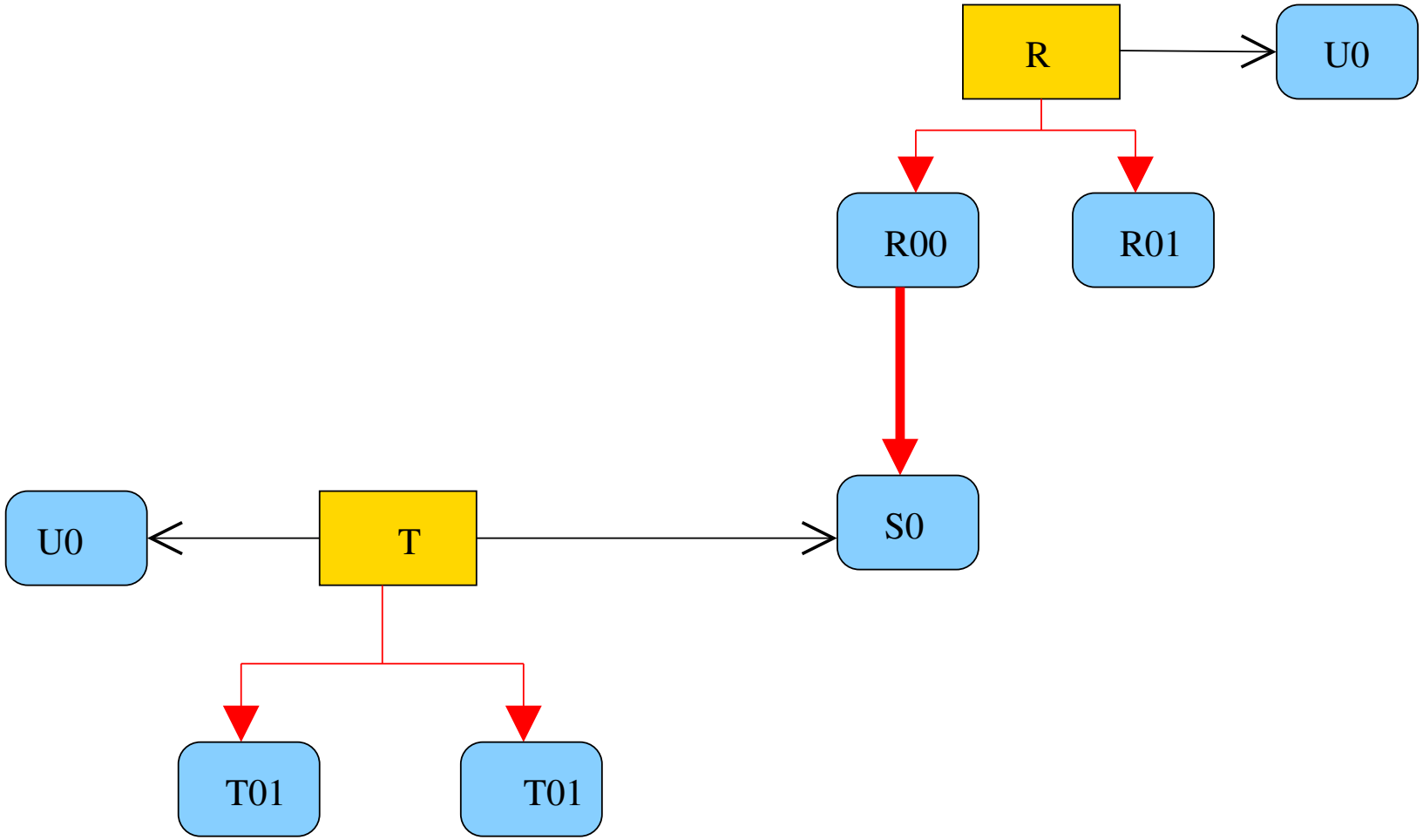
A component satisfies some interfaces (specifications) and depends on some other interfaces (specifications)



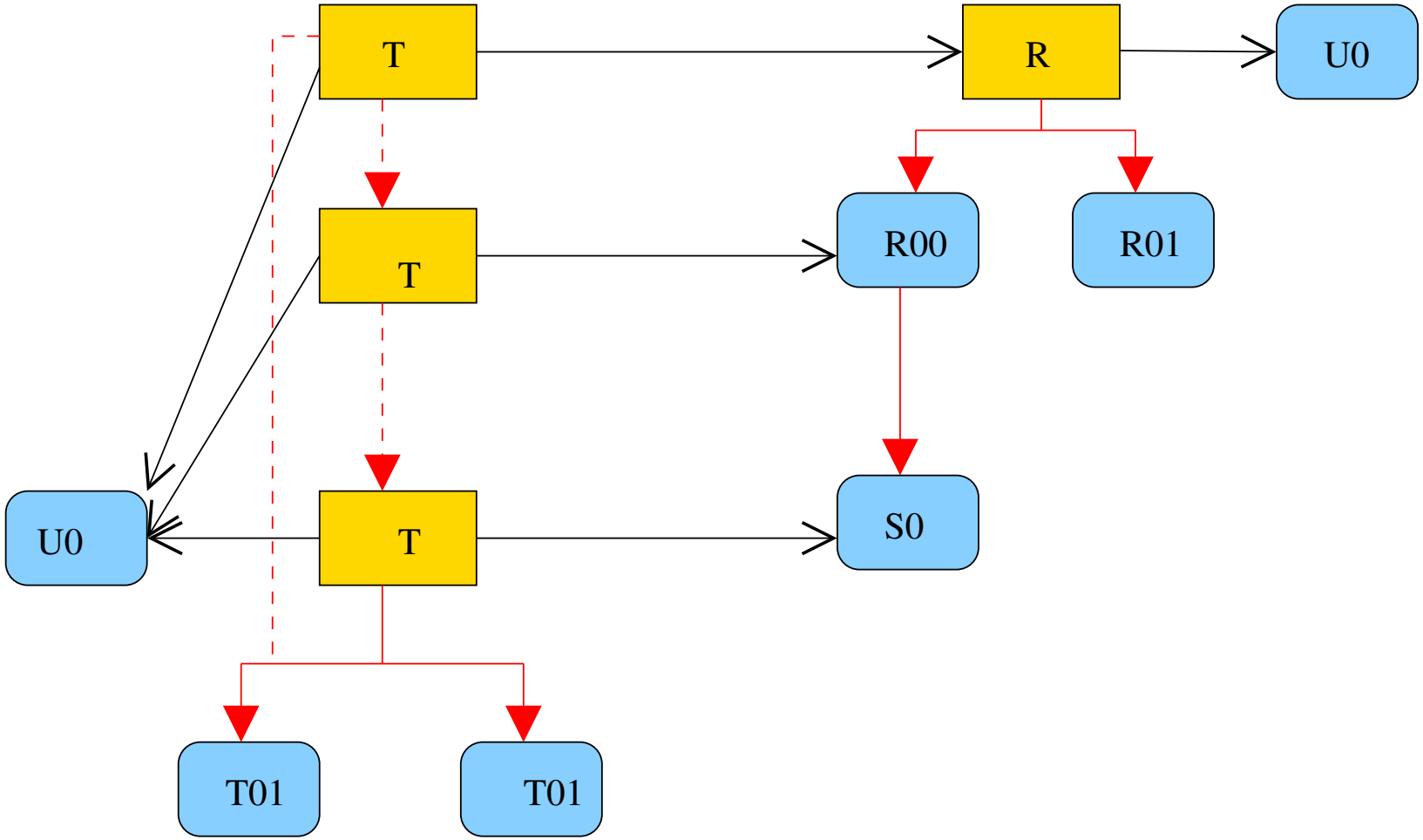
Another component R



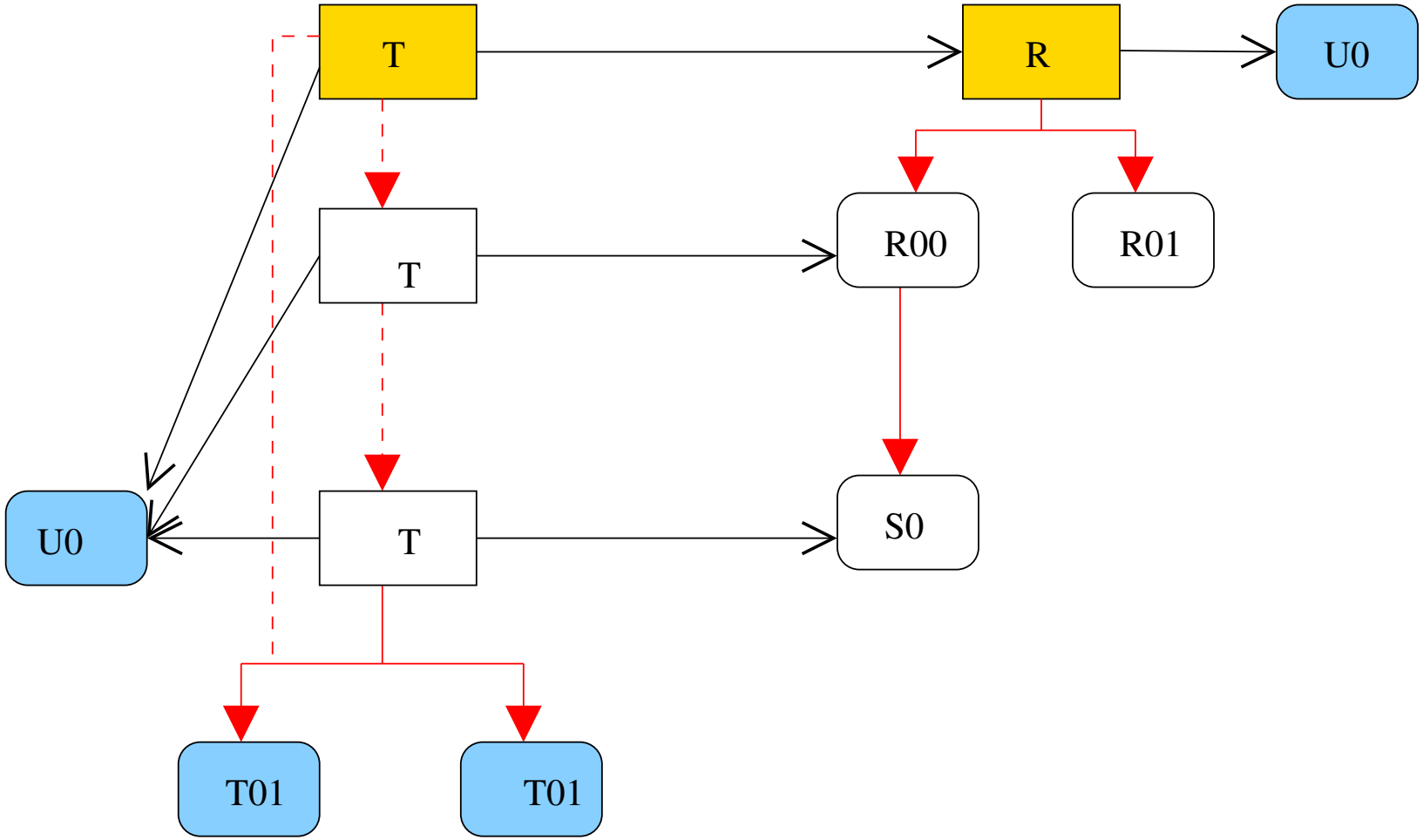
Use R in T



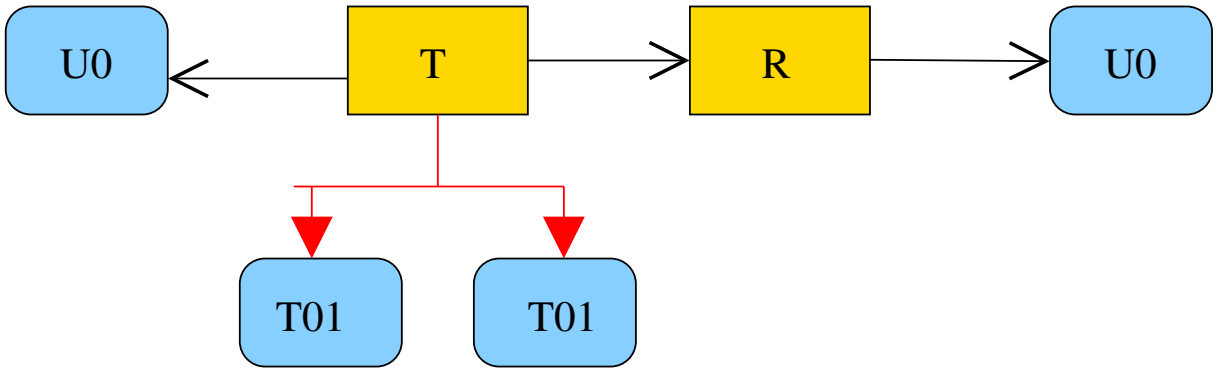
Use monotonicity and transitivity



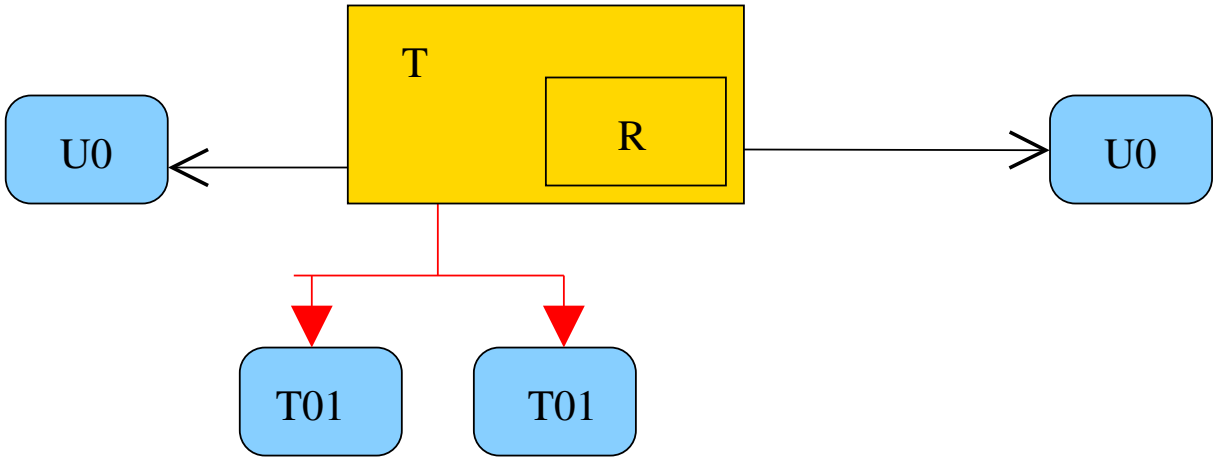
Final result



Hide derivation



Package into bigger component



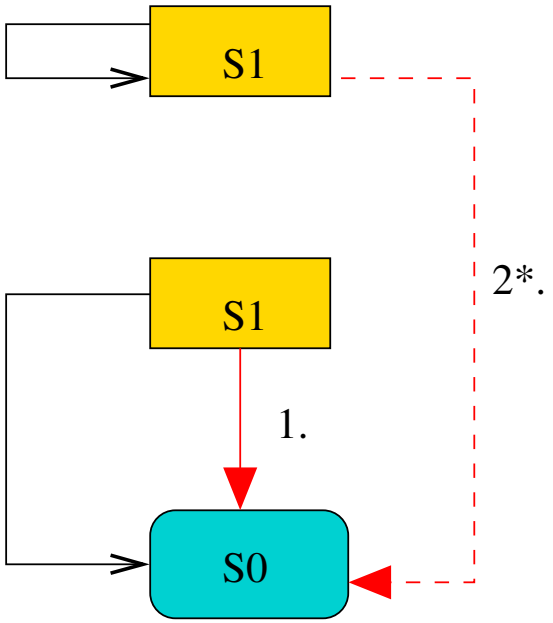
Implementing recursive components

The following *recursion rule* can be used to reason about a recursive construct:

$$S_0 \sqsubseteq S_1[S_0] \Rightarrow S_0 \sqsubseteq (\mu X \cdot S_1[X]) \quad (*)$$

The star indicates that there is a side condition for this rule (usually some kind of termination or well-foundedness condition).

Refinement diagram /Hilbert-like derivation



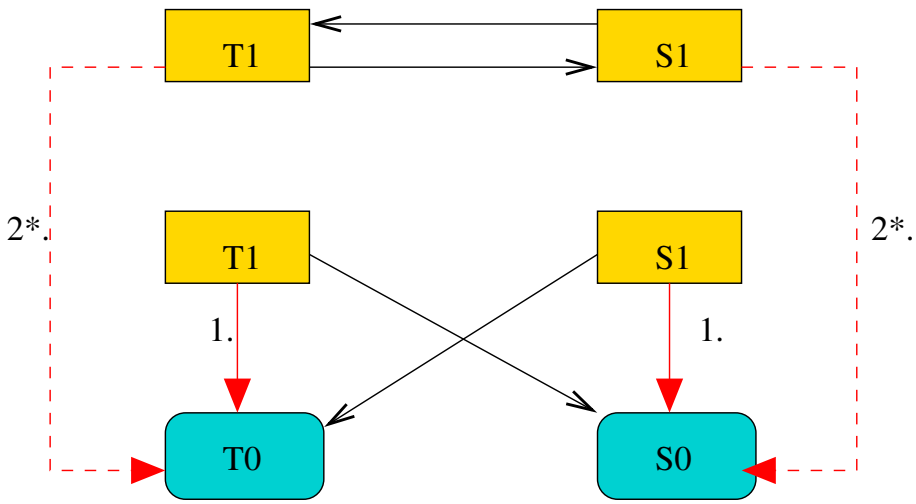
.

1. $S_0 \sqsubseteq S_1[S_0]$
2. $S_0 \sqsubseteq (\mu X \cdot S_1[X])$ (by recursion rule)

Mutual recursion

- Initially, we have two specifications, S_0 and T_0 .
- Assume that we implement T_0 with T_1 that uses the specification S_0 and S_0 with S_1 that uses the specification T_0.
- Want to show that the system where these two statements call each other directly is a correct implementation of the specifications S_0 and T_0.

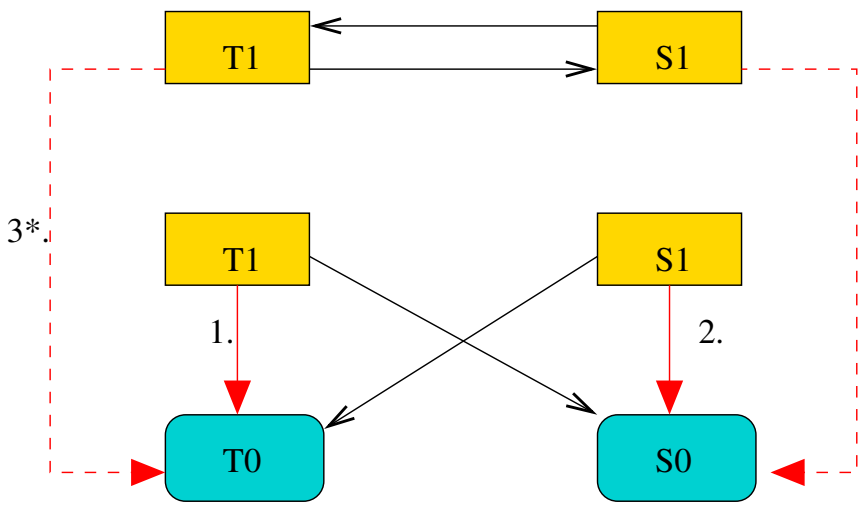
Refinement diagram / Hilbert-like derivation



1. $(T_0, S_0) \sqsubseteq (T_1[S_0], S_1[T_0])$
2. $(T_0, S_0) \sqsubseteq (\mu X, Y \cdot (T_1[Y], S_1[X])$

(by recursion rule)

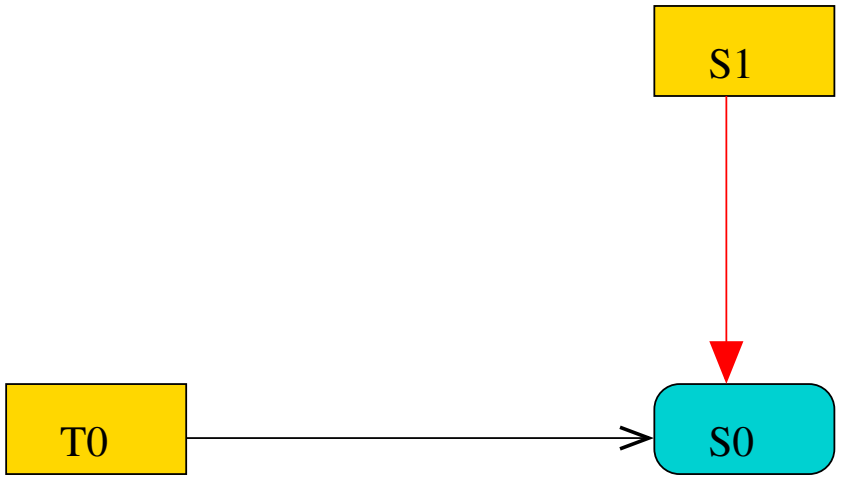
One step at a time



1. $T_0 \sqsubseteq T_1[S_0]$
2. $S_0 \sqsubseteq S_1[T_0]$
3. $T_0 \sqsubseteq (\mu X, Y \cdot (T_1[Y], S_1[X]))_1$ (by recursion rule)
4. $S_0 \sqsubseteq (\mu X, Y \cdot (T_1[Y], S_1[X]))_1$ (by recursion rule)

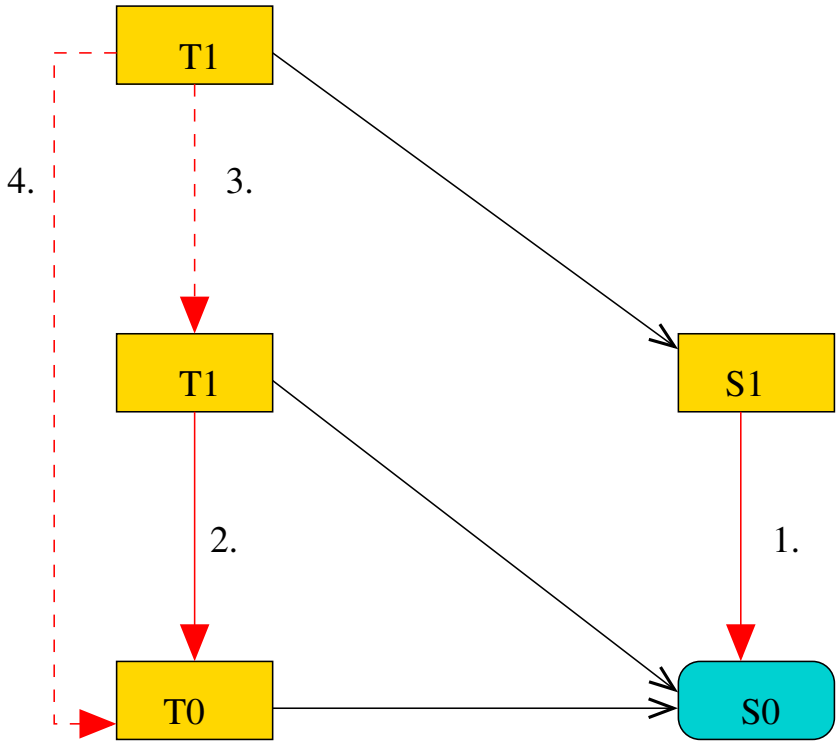
Refining a component and information hiding

- The T - component knows the specification S_0 of the S- component,
- T component does not know the implementation S_1 of the S component (i*nformation hiding*)
- We want to refine T_0 to a new part T_1.

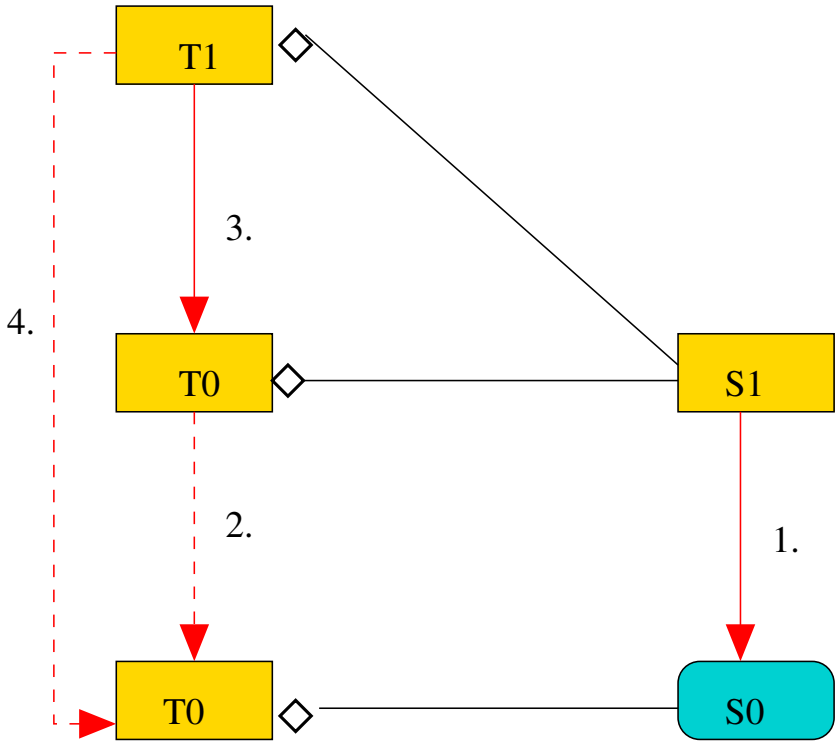


Information hiding in refinement

Should the new part T_1 know about the implementation S_1 or not?



Hiding information



Not hiding information

□

Remarks

- On the left $T_1[S_0]$ occurs as an intermediate step in the derivation, so T_1 cannot have any information about the implementation S_1.
- On the right, $T_0[S_1]$ occurs as an intermediate step, so T_1 may use information about the implementation S_1.
- The final diagrams are the same in both cases, but the derivation shows the differences between the two diagrams.
- Information hiding is good when components are shared, not necessary when we have private components.

Reasons not to respect information hiding

- If we insist on information hiding in the refinement step above, then it follows that the implementation T_1 of T_0 cannot make use of the implementation S_1.

- In many cases, it may be desirable that T_1 does make use of the implementation, e.g., because of efficiency reason, or because it needs direct access to the data representation in S_1, or because it wants to utilize new functionality provided by S_1.

- In this case, we would prefer to implement S_0 and T_0 together, and therefore break the information hiding principle.

Bottom up and top down construction

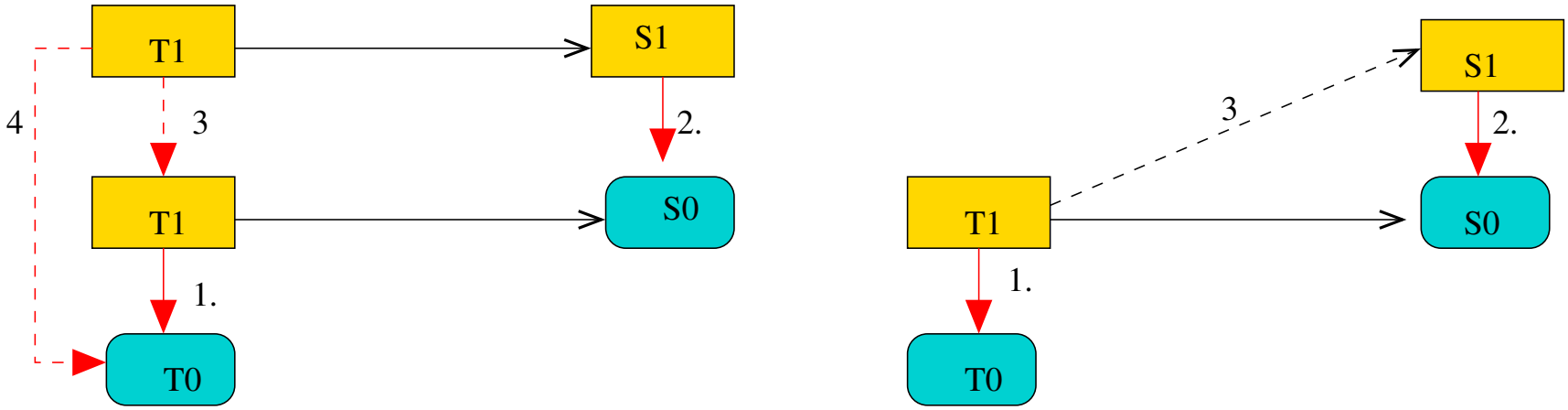
- In both these cases, the ordering of the steps 1 and 2 are not important.
- If step 1 comes before step 2, then we proceed *bottom up*, first defining the components that are used before using them.
- If step 2 comes before step 1, then we are proceeding *top down*, first defining the user of a component, before implementing the component.
- We can also think about the construction as proceeding by first placing all the entities on the diagram, before starting to connect them by refinement arrows (*off the shelf* components).

ON DUPLICATION OF TERMS

- Duplication vs redirection
- Ambiguity with redirection
- Compacting refinement diagrams

Duplication vs redirection

Derivation can seem overly complex, because we are duplicating some entities (T_1 in left figure)



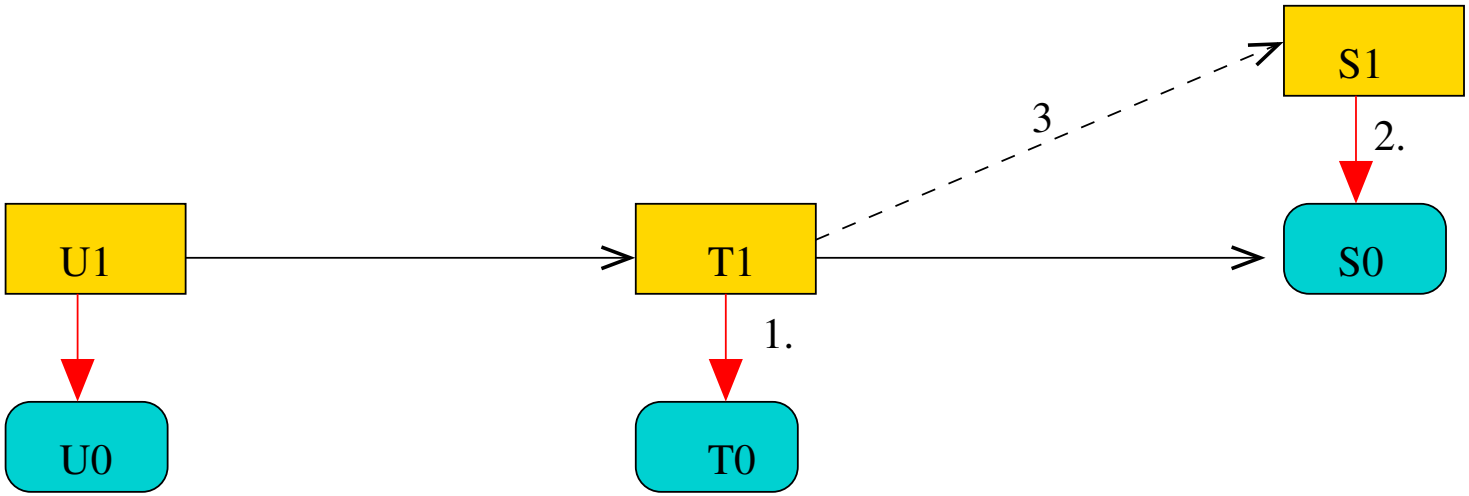
It would seem more economical to redirect the arrow in the derivation rather than duplicating the whole entity (right)

Redirecting arrows

- This figure shows the third step as just a redirection of the solid arrow from T_1 to S_1.
- Implicitly could state that this redirection is ok, in the sense that all relations that held before are still valid.
- In particular, this would mean that $T_1[S_1]$ would still be an implementation of T_0
- Advantage: the derivation becomes more compact, the use of duplicates is avoided, and the layout of the class diagram is unchanged, we just move arrows around.

Why not to use redirection

- the meaning of a box becomes ambiguous. Consider a user of T_1, say U_1.

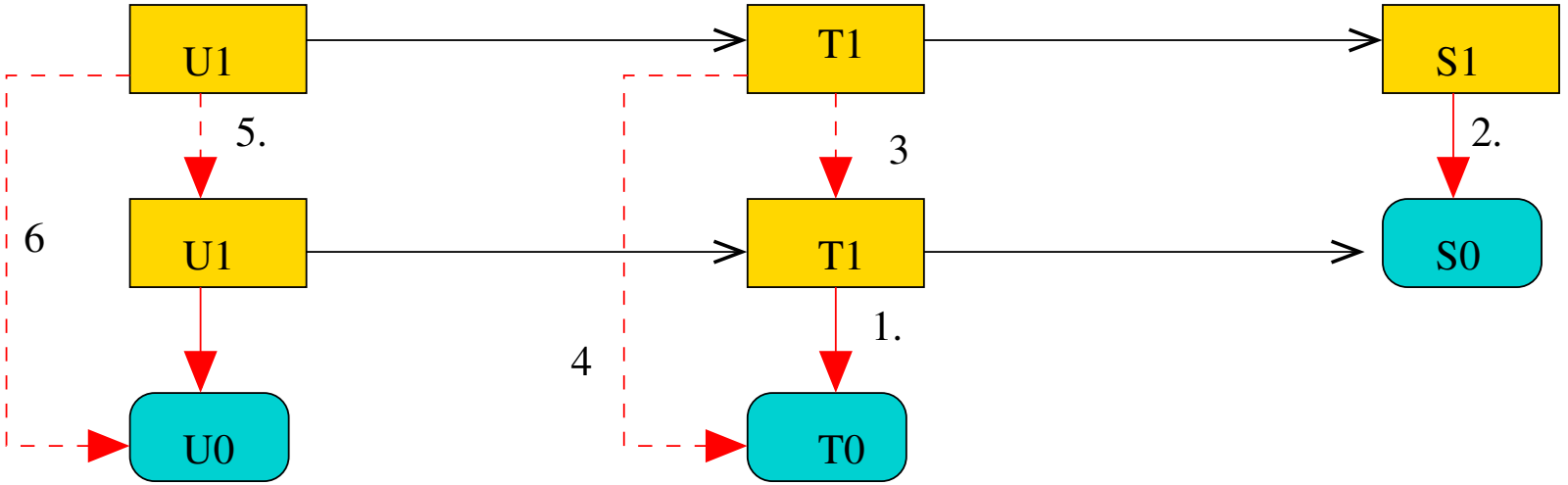


- A change in T_1 (to use S_1 rather than S_0) will also mean that U_1 is changed, from $U_1[T_1[S_0]]$ to $U_1[T_1[S_1]]$. But this change is difficult to notice here.

Duplication is good

- Duplication of terms avoids hidden, uncontrolled and unwanted changes in the software system.

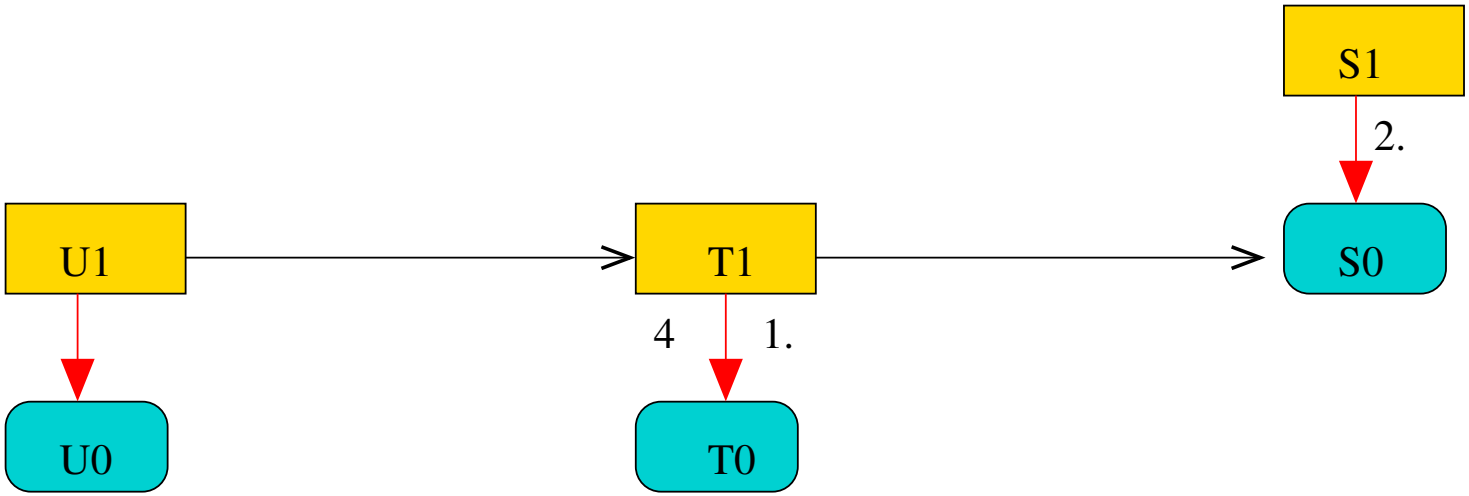
- Compare above to the same derivation with duplication:



- New terms are shown explicitly, $U_1[T_1[S_0]]$ and $U_1[T_1[S_1]]$ both in diagram.

Compacting refinement diagrams

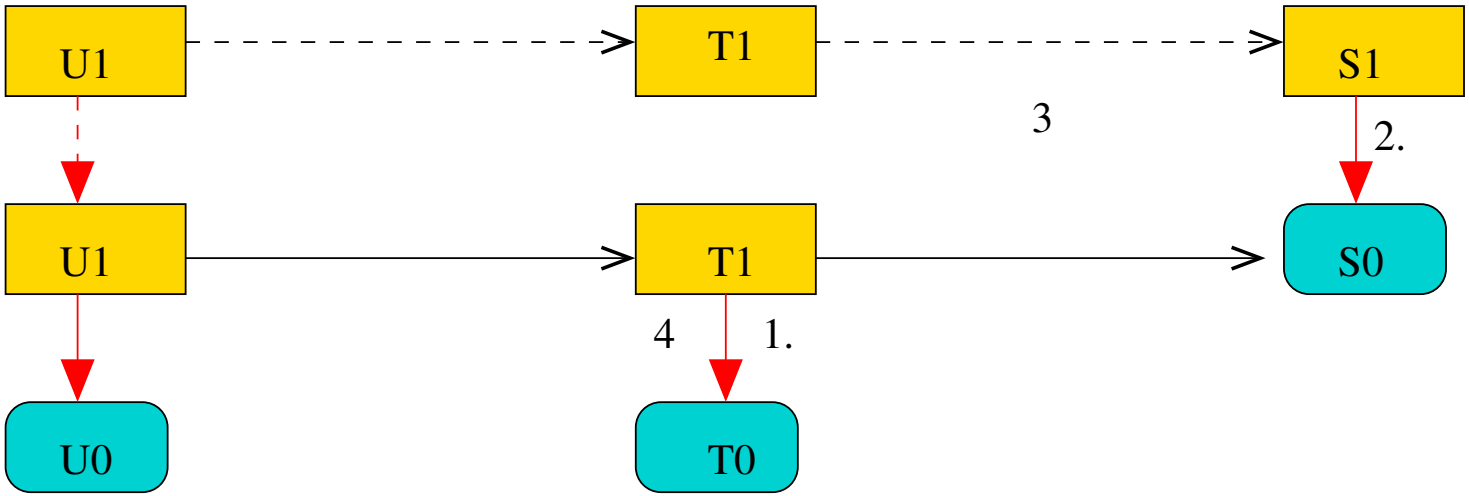
Avoid making inferences unless they are explicitly needed. Example:



The refinement of S_0 by S_1 is shown, but we have not drawn the consequences. Inferred terms and arrows can be indicated later, if they are needed.

Combine inference steps

Alternatively, we could combine a number of inference steps into a single step:



Here we only show the desired conclusion, that $U_1[T_1[S_0]]$ is refined by $U_1[T_1[S_1]]$. Intermediate transitivity and monotonicity steps are implicit, and are easy to see by arrow chasing.

Conclusion on compactness

- Duplication of terms is needed, to avoid ambiguity in the derivations
- But one does not have to draw all the inference arrows and intermediate terms that are possible, only those that are relevant for the final result.
- The refinement derivation is a proof, so it must be unambiguous and show all the necessary information
- After the proof is done, then one need only to display the part of the diagram that is interesting for the present purpose. The rest can be hidden.

EXTENDING SOFTWARE SYSTEMS

Increment an existing system by either

- adding a new *component*, (described above), or
- adding a new *extension*, on top of an existing component (described below).

Extension

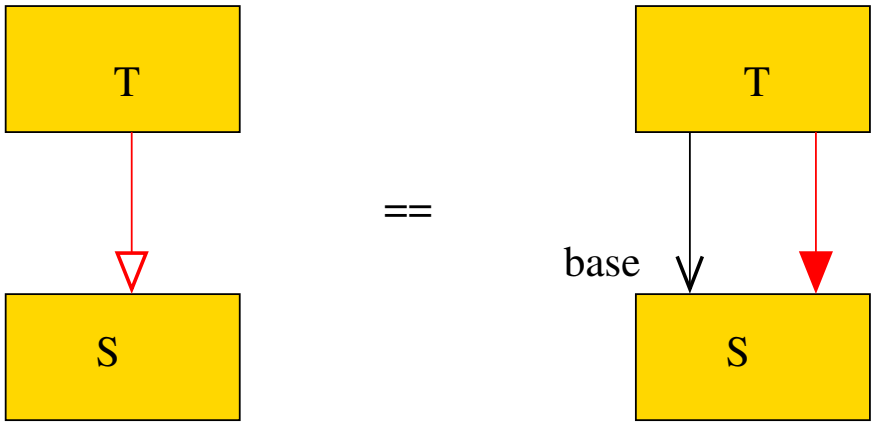
- We write $S \triangleleft T$ for the component that we get by *extending* component S by component T .
- The extending component T refers to the extended component S by the name $base$, $T = T[base]$ (could also call the base part $super$).
- We model extension by usage: $S \triangleleft T = T[S]$, i.e. an extension is a component that uses another component.

Refinement requirement

- We will require from an extension that $S \sqsubseteq T[S]$ holds, i.e., the extension should preserve the functionality of the original component.
- This holds if the extension is a *superposition refinement* of the original statement. E.g., for classes,
  - the extended statement can introduce some new attributes, but cannot remove old attributes
  - it can add new methods, an redefine old methods, but must preserve the effect of the old methods on old attributes
- We write $S \preceq T$ (S is *superposition refined* by T) for the statement $S \sqsubseteq S \triangleleft T$.

Extension in refinement diagrams

We introduce a special arrow for superposition refinement.



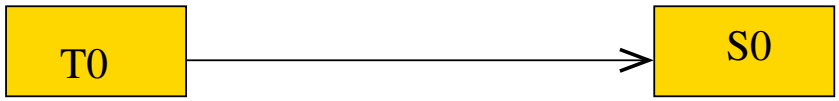
Note that $(S \triangleleft T)[X] = S[X] \triangleleft T[X] = T[X, S[X]]$, i.e., both S and T can also be dependent on other parts in the environment.

Adding new functionality to a system

- We have built a basic system, consisting of a collection of parts (e.g., classes) that use each other. This system provides some basic functionality.
- Next, we want to extend the functionality of the system with some new features
- Often, it is not sufficient to just extend a single part, the new functionality may require that a number of components are extended simultaneously
- Essentially, we want to build a new *layer* of functionality on top of the basic system layer, where the new layer provides the added functionality.

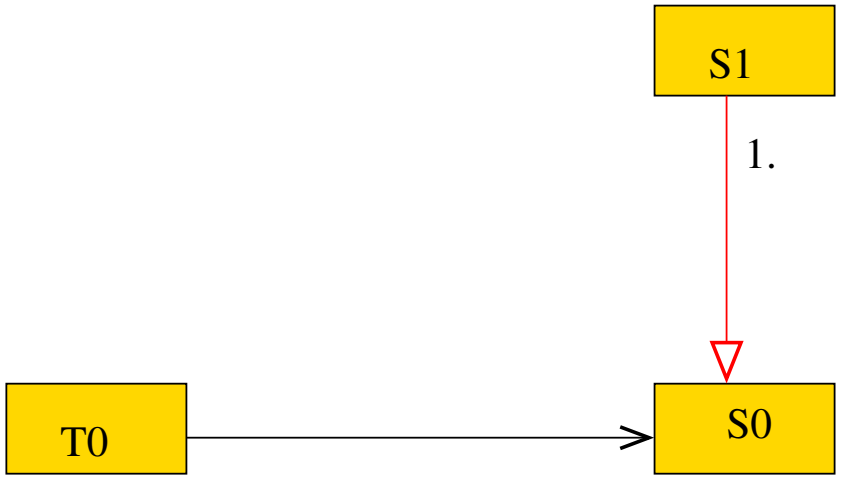
Initial layer

Start with the system consisting of $T_0[S_0]$ and S_0 (the basic layer).



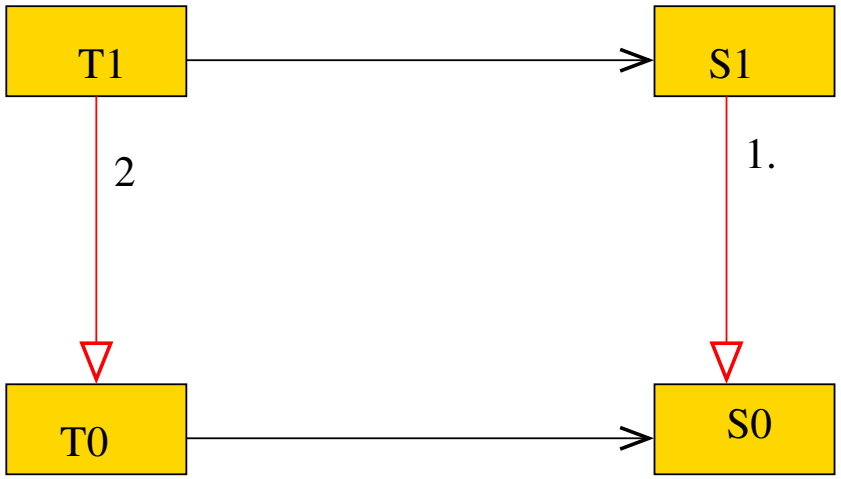
Extend used component

Introduce a part S_1 such that $S_0 \preceq S_1$



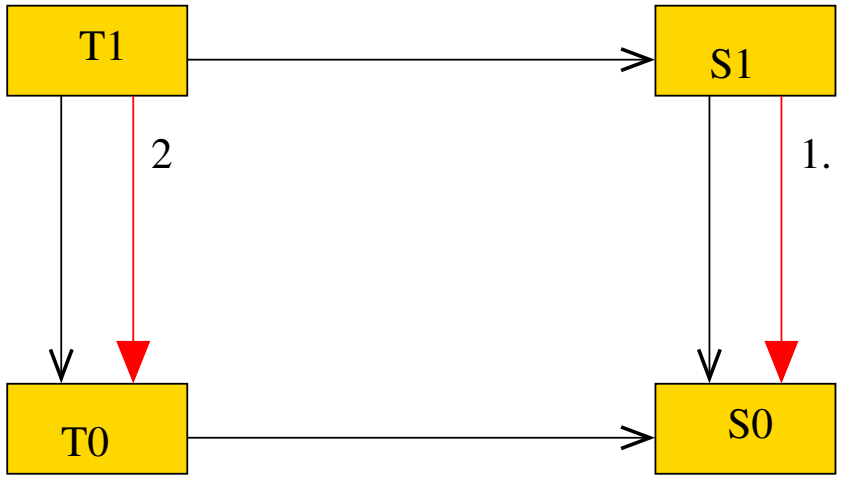
Extend using component

Then introduce a new part T_1 such that $T_0[S_0] \preceq T_1[S_0 \triangleleft S_1]$



Derivation in terms of usage alone

- Start with the system consisting of $T_0[S_0]$ and S_0 (the basic layer).
- Introduce a part S_1 such that $S_0 \sqsubseteq S_0 \triangleleft S_1$
- Then introduce a new part T_1 such that $T_0[S_0] \sqsubseteq T_0[S_0] \triangleleft T_1[S_0 \triangleleft S_1]$



Static and dynamic binding

- This layering uses *static binding* for the extended parts: in the extended system

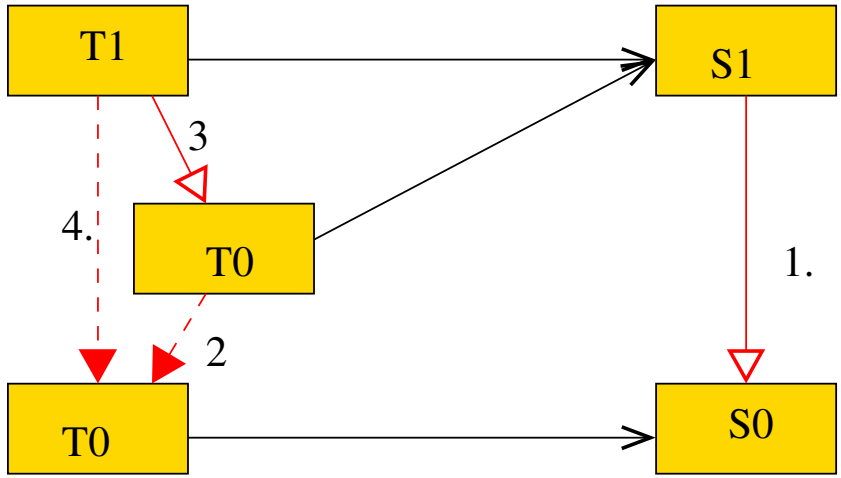
$$T_0 \triangleleft T_1 = T_0[S_0] \triangleleft T_1[S_0 \triangleleft S_1]$$

  the base part T_0 continues to use the base part S_0, even if there is an extension $S_0 \triangleleft S_1$ of this part available.

- We model *dynamic binding* by requiring that the extended version is used in all extension layers.

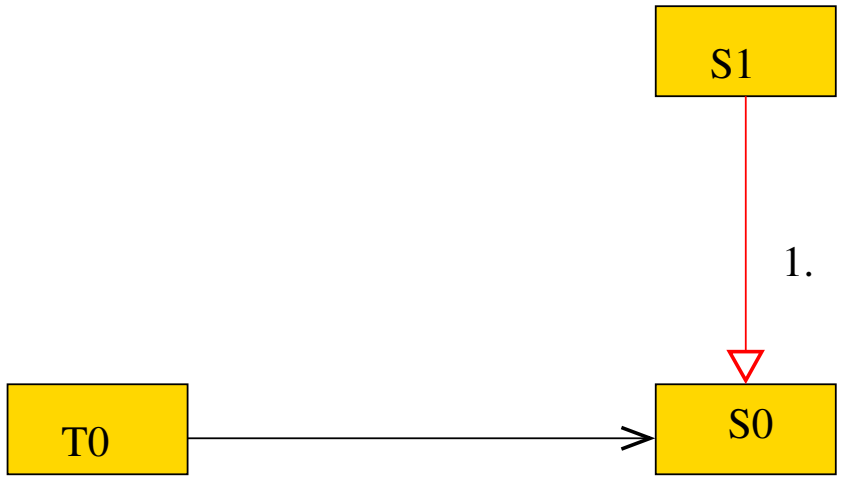
Extension with dynamic binding

- The following derivation achieves dynamic binding:



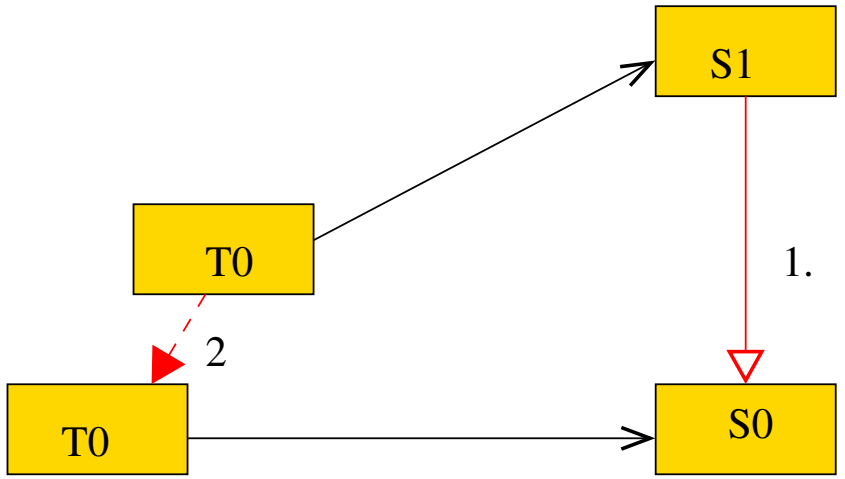
Step 1

First step same as before, introduce extension S_1 of S_0



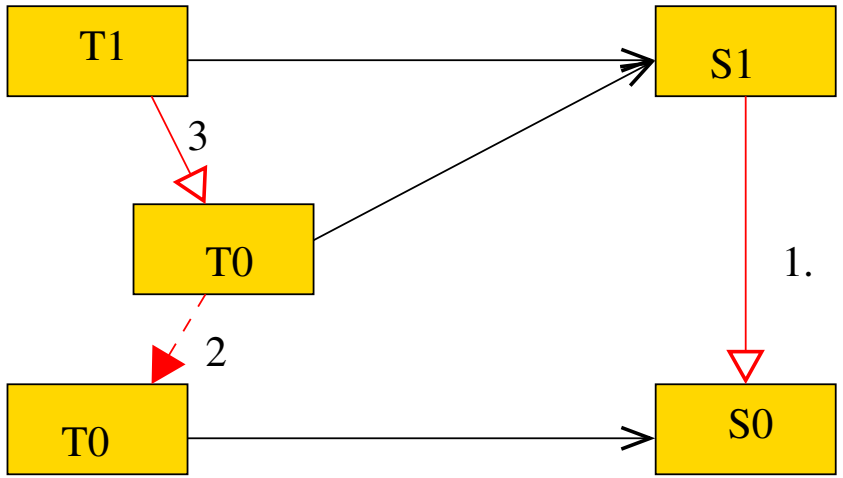
Step 2

Step 2 uses monotonicity to derive $T_0[S_0] \sqsubseteq T_0[S_0 \triangleleft S_1]$



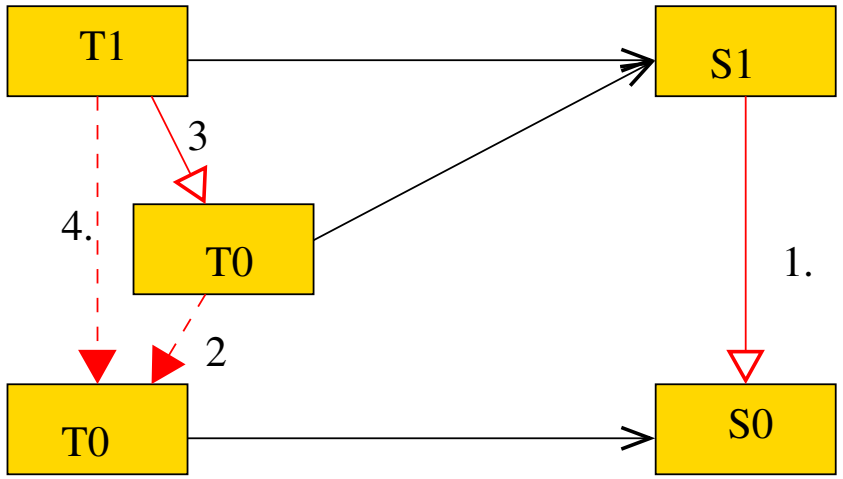
Step 3

In step 3 we show that $T_0[S_0 \triangleleft S_1] \sqsubseteq T_0[S_0 \triangleleft S_1] \triangleleft T_1[S_0 \triangleleft S_1]$

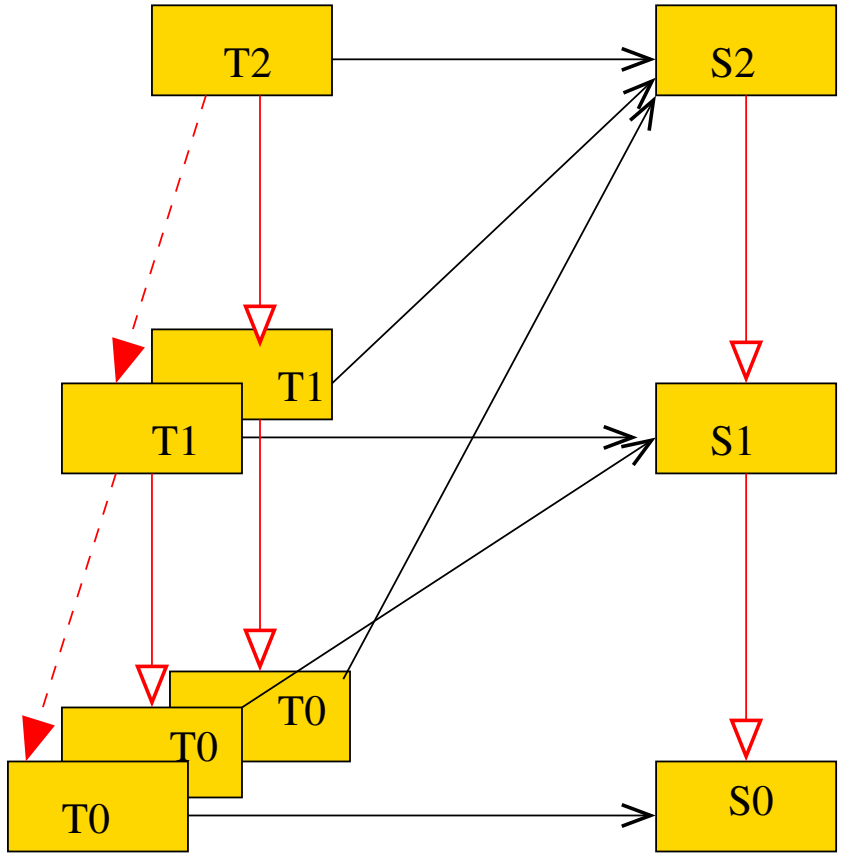


Step 4

Step 4 gives required result by transitivity, $T_0[S_0] \sqsubseteq T_0[S_0 \triangleleft S_1] \triangleleft T_1[S_0 \triangleleft S_1]$



Dynamic binding with successive extensions

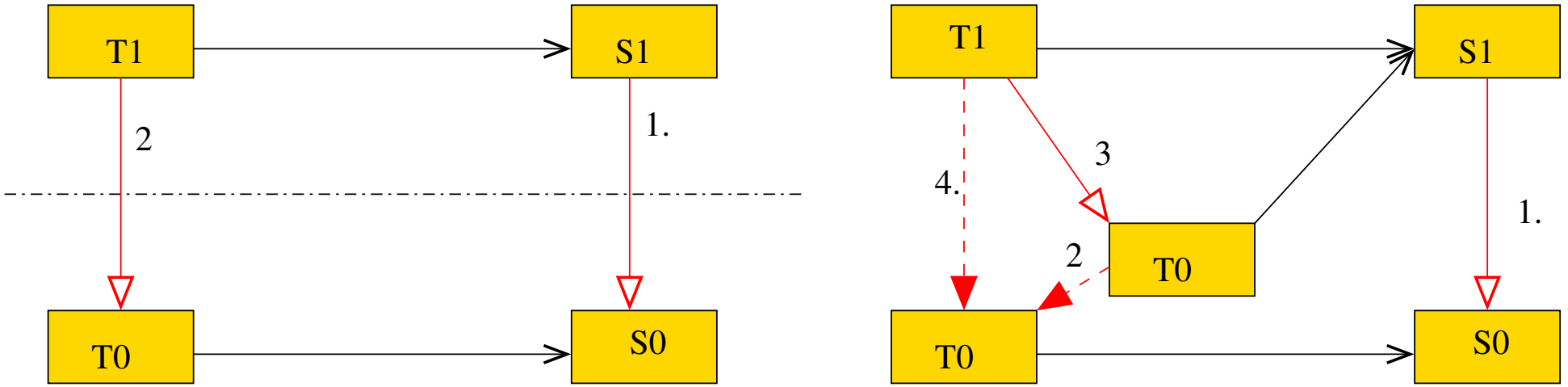


Layers

- The extra steps that are required by dynamic binding can be inferred and do not have to be proved explicitly. This suggests that we could introduce *implicit* dynamic binding.
- We introduce *layers:* a collection of extensions that are to be used together.
- A reference to a part at a lower level of extension it is taken to refer the extension in the current layer (i.e., all calls are bound to extensions in the current layer).
- We indicate a layer with a dashed outline in the diagram.

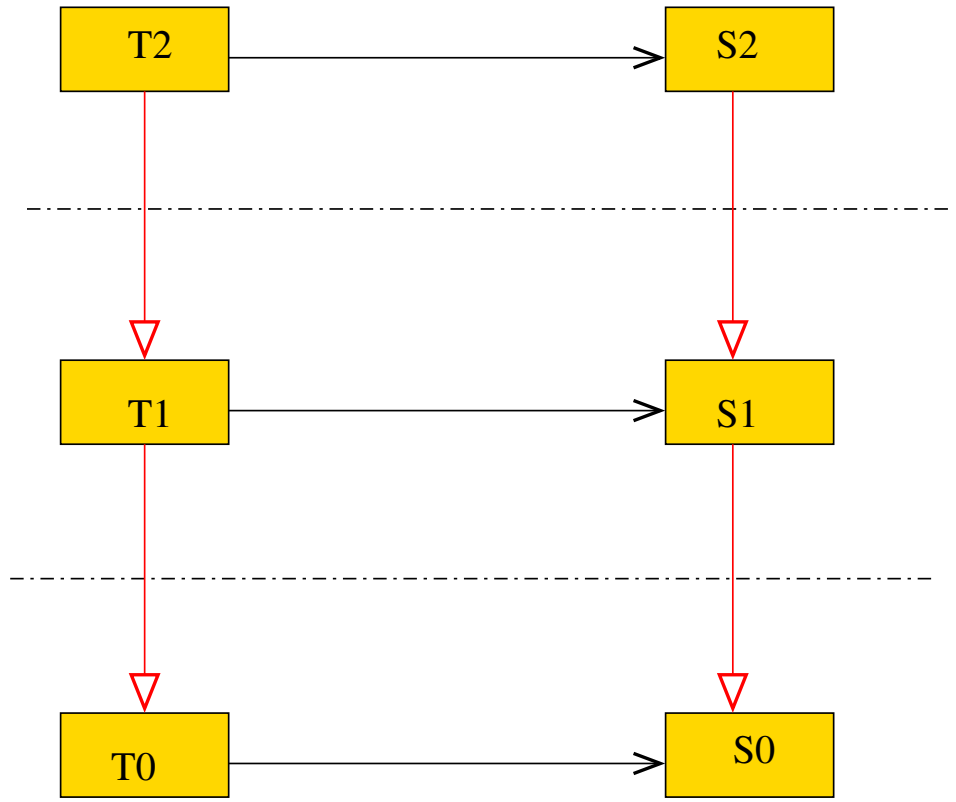
Two layers

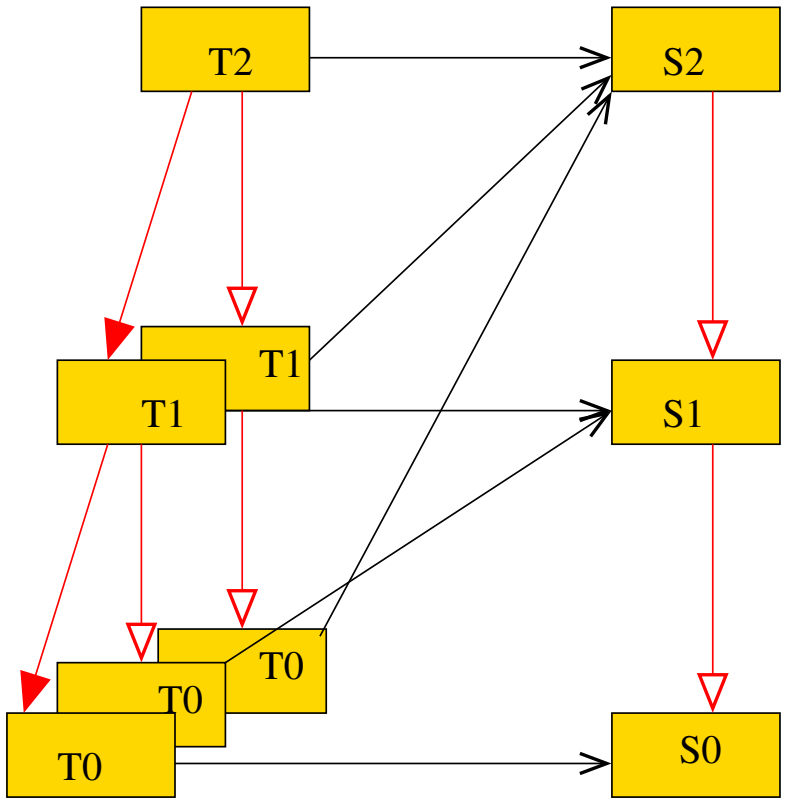
Layers (left), no layers (right)



Three layers

:

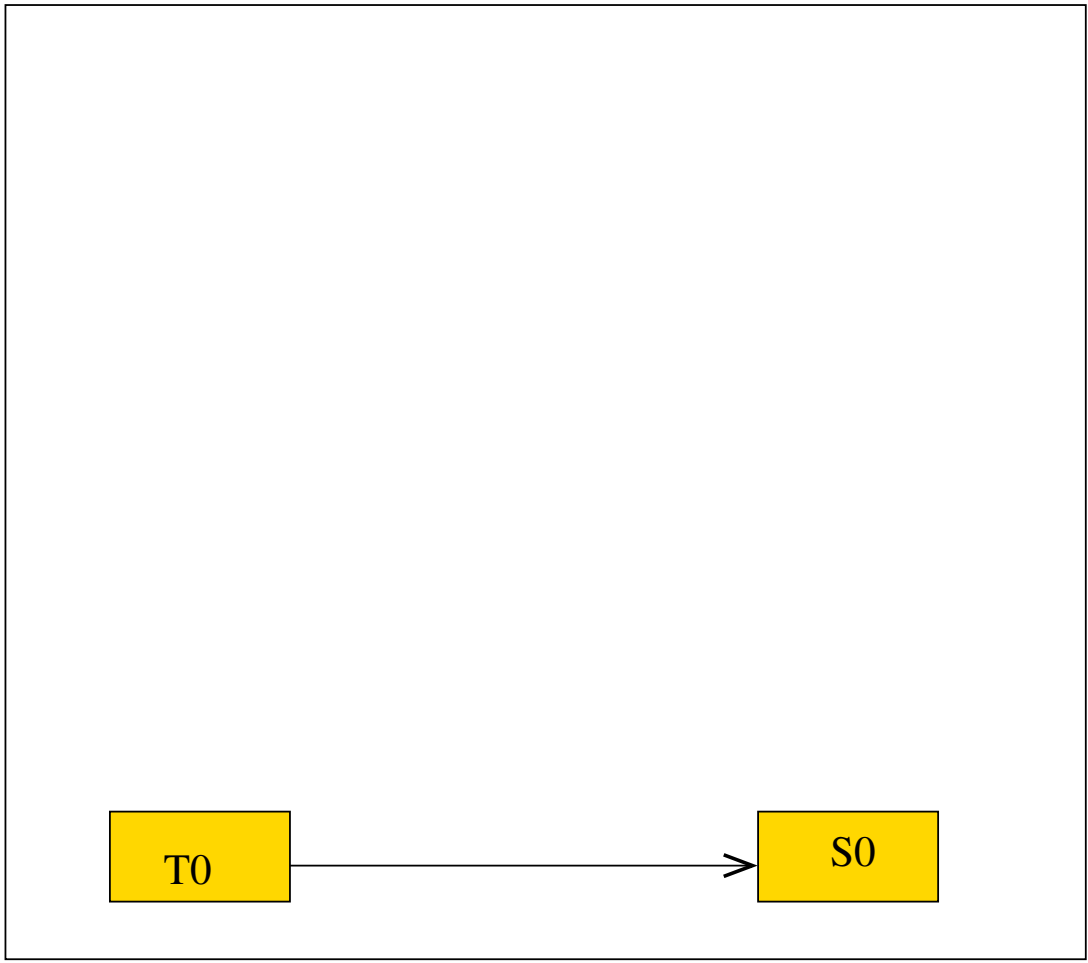




Each layer defines a different system

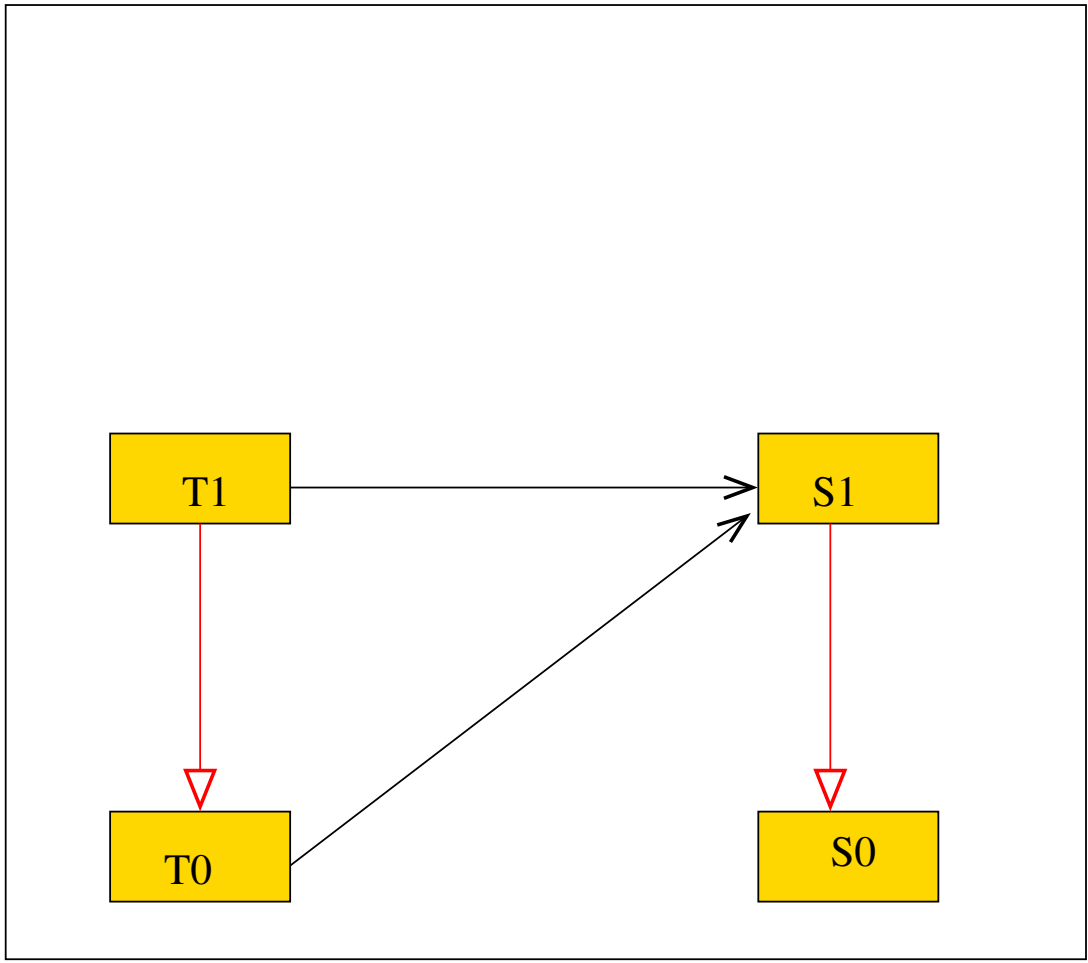
Basic system

- The basic system is started by using T_0 as the main program.
- It provides some basic functionality, and makes use of S_0 as an auxiliary part.



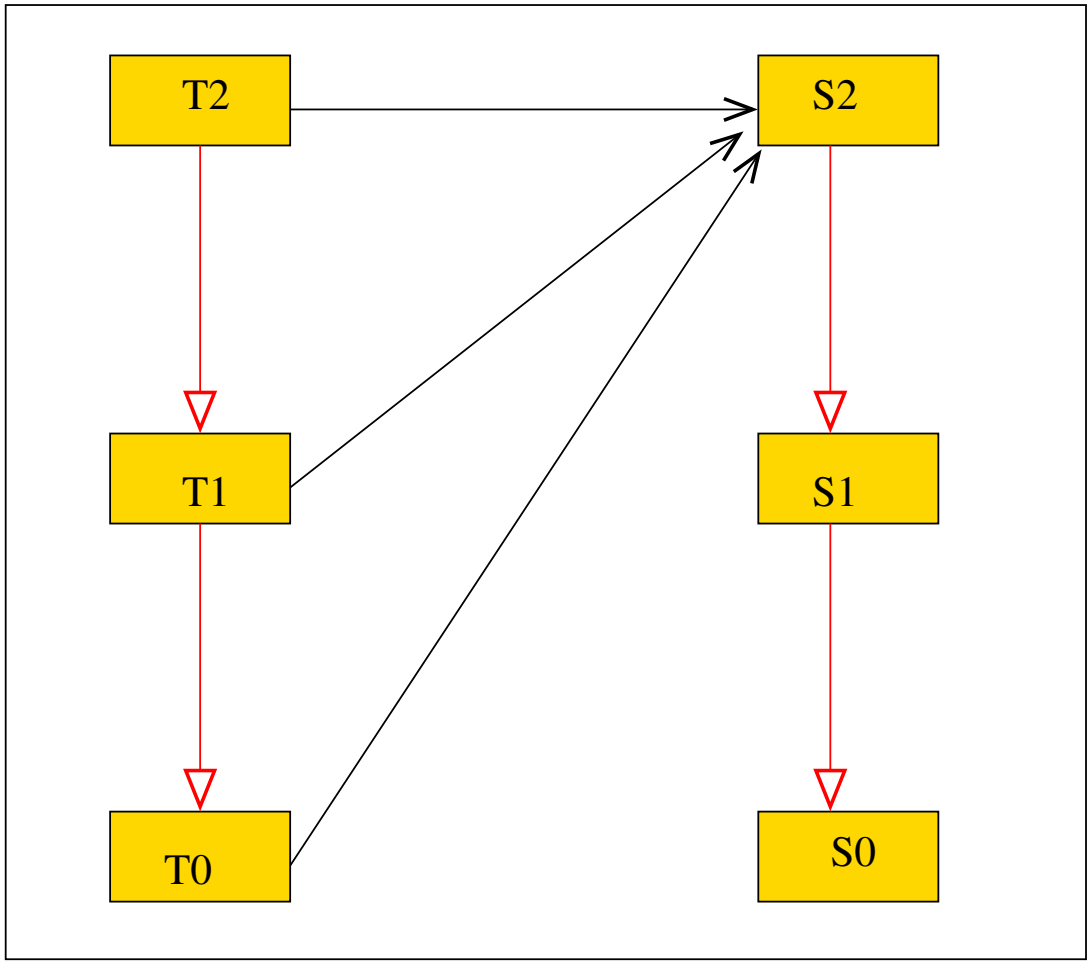
Intermediate system

- The intermediate system is started using T_1 and it makes use of the extension S_1 of S_0.
- All calls to S_0 are redirected to S_1.



Final system

- The most advanced system is started from T_2 and makes use of the extension S_2 of S_1 .
- Calls to S_0 or S_1 are redirected to the extension S_2 .



Loose ends

- We assume that the layers in the system have a tree like structure, so that for each layer there is a unique *previous* (father) layer.
- A part may only reference a part in a *preceding* layer (which means either a previous layer or a layer that precedes the previous layer).
- For any used component, the *most recent* extension below or in the present layer is used
- These conventions correspond to the *single inheritance* principle in object oriented systems.

Remarks

- Note that the layering construct allows a number of different extension hierarchies to co-exist at the same time.
- At the same time, it prevents extensions in different layers to be used at the same time.
- In many situations, this is exactly what we want. There are, however, also situations where we do not want this. For such situations, we may use both kinds of calls: *dynamic calls* that are redirected by layering, and *static calls* that cannot be redirected.
- We then need a differentiating notation for these two calls

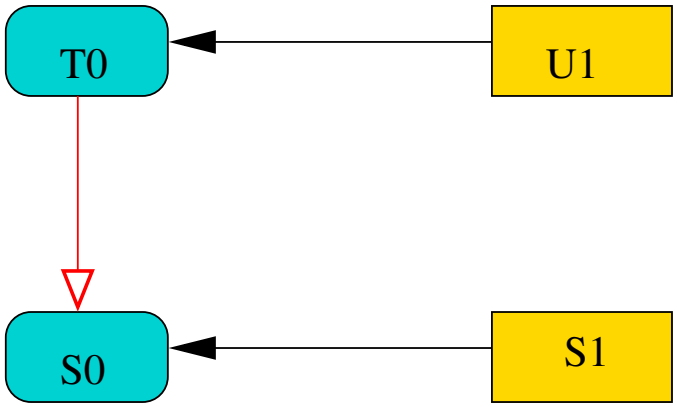
LAYERED SPECIFICATIONS AND COMPONENTS

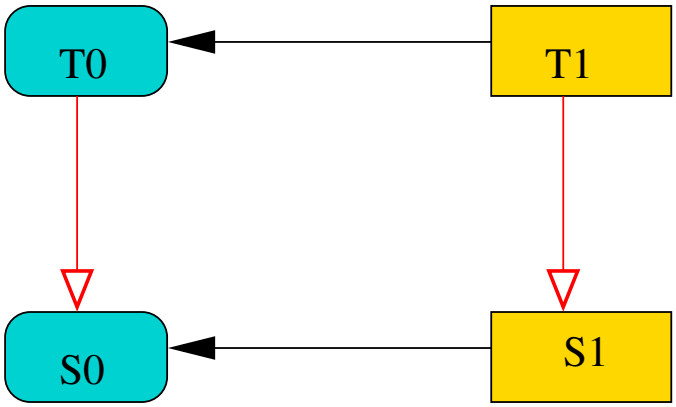
- Let us next consider the relationship between extension and implementation.
- Assume that we have a preliminary specification S_0 which we have implemented by S_1.
- Assume now that we want to extend the specification by some new features T_0. This now gives us a new *layered specification* $S_0 \lhd T_0$.

Implementing layered specifications

We could implement this layered specification directly by

- a new implementation U_1 (left), or
- by an extension T_1 of the original implementation S_1, giving us $S_1 \triangleleft T_1$ (right).





Comparison

- Left, the presence of the new features in T_0 requires a change of the original implementation of S_0, and so the new and old features are therefore implemented anew, as U_1.

- Right, the new features in T_0 do not require a reimplementation of the features in S_0, it is sufficient to just extend the implementation S_1 with an implementation T_1 for the new features of T_0.

- Right approach works if the new feature are rather orthogonal to the old features

Advantage of layered implementation

- One can reuse the implementation of the S_0 features
- Need to check that
  - the new features in T_0 are correctly implemented and
  - that their implementation does not invalidate the implementation of old S_0 features.

Extension, non-recursive implementation and usage

- A problem with the construction of layered system above is that the proof is not local.
- Thus, when we add extension layers to the system, we are forced to prove refinement between larger and larger terms.
- For instance, we have to prove in step 3 that $T_0[S_0 \triangleleft S_1] \sqsubseteq T_0[S_0 \triangleleft S_1] \triangleleft T_1[S_0 \triangleleft S_1]$.
- If S_0 and S_1 are non-trivial statements, then this can require proofs involving very large terms. If these statements in turn call other statements, the terms get even bigger.

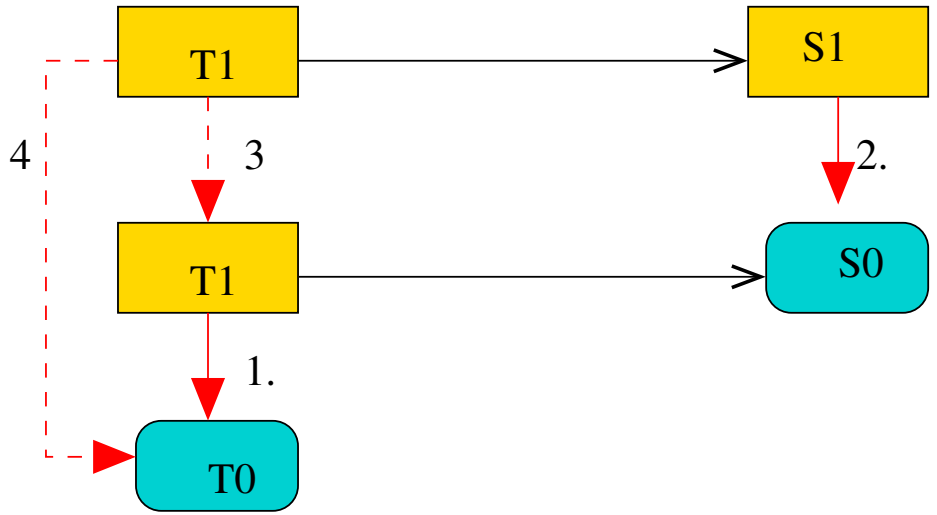
Modularizing extensions

- We therefore need to use more local reasoning and modularize the proof, in order to keep it of manageable complexity.
- The solution is again to introduce specifications for components.
- This complicates the proof, because we have to come up with a whole new set of constructions, the specifications, and we have to establish many more properties.
- However, the terms in the propositions are now smaller, and calls only refer to specification statements.

- 16a

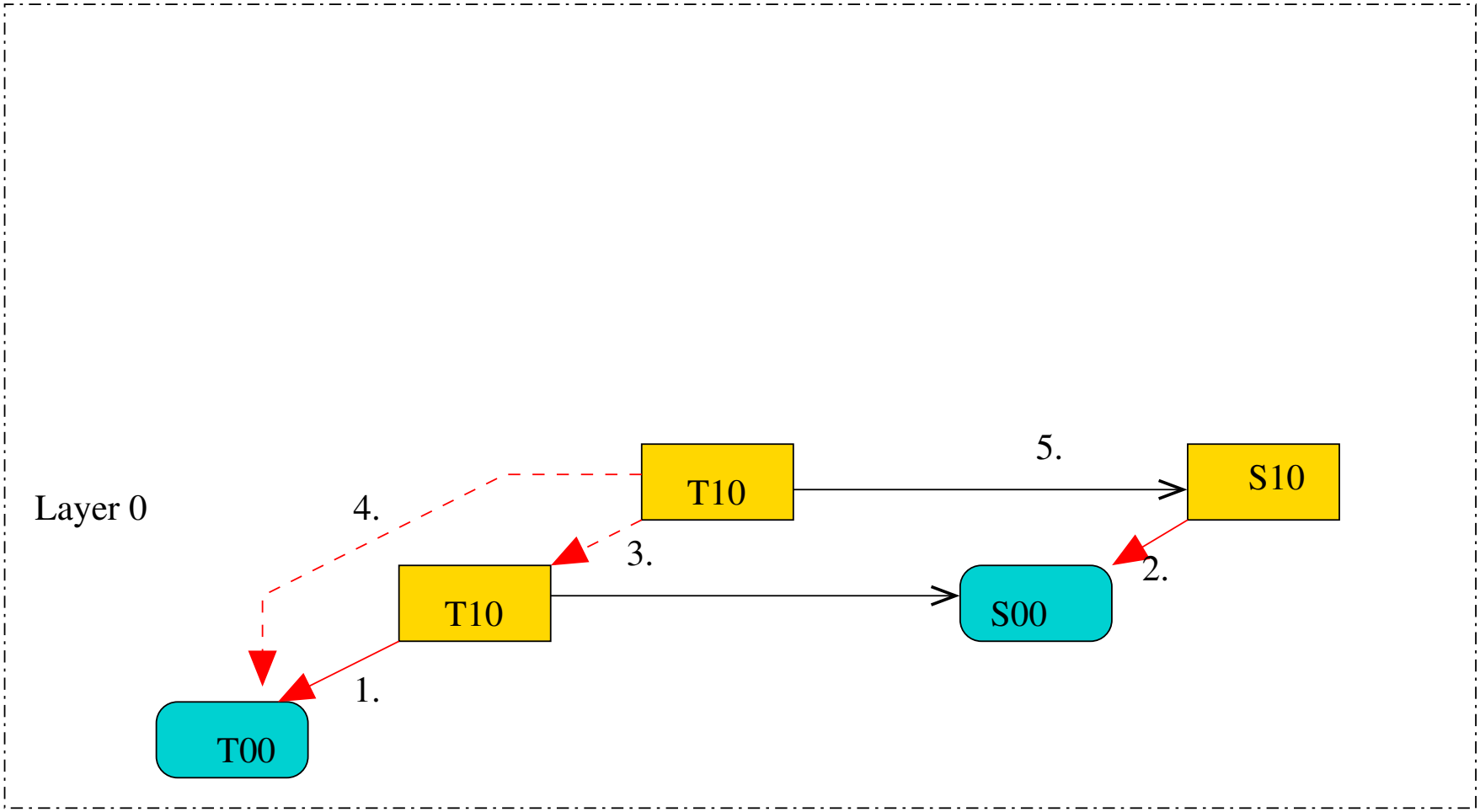
Constructing first layer

Earlier derivation for implementing a specification using an auxilliary part.



Turn figure sideways, add 0 to component as layer index (S_1 becomes S_{10} etc).

Constructing first layer - 2



We write here S_{jk} for component S in implementation j and layer k, and similarly for component T .

Hilbert proof for first layer

1. $T_{00} \sqsubseteq T_{10}[S_{00}]$ (prove)

2 $S_{00} \sqsubseteq S_{10}$ (prove)

3. $T_{10}[S_{00}] \sqsubseteq T_{10}[S_{10}]$ (by monotonicity)

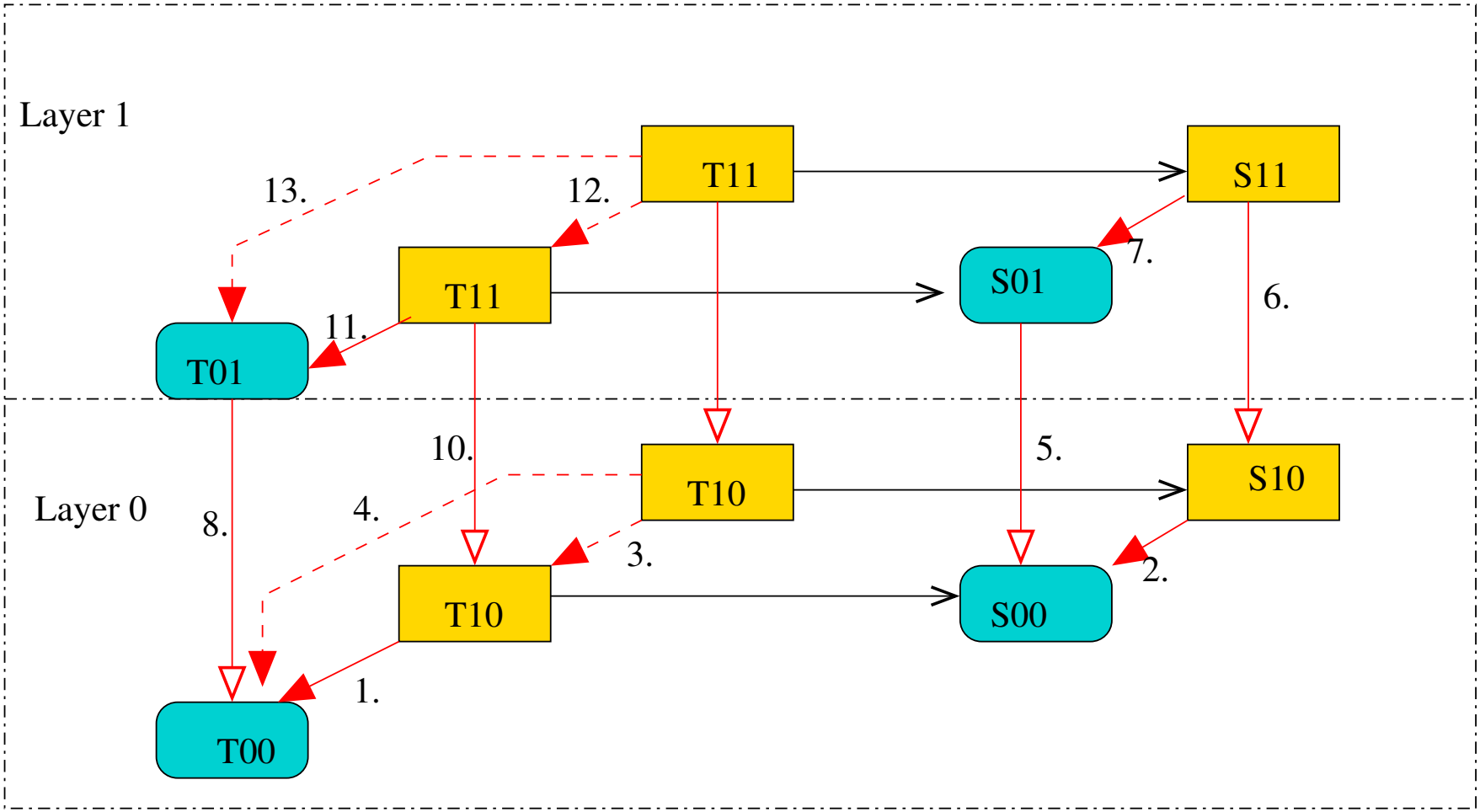
4. $T_{00} \sqsubseteq T_{10}[S_{10}]$ (transitivity)

Verification of individual proof steps

The steps indicated by “proof” are verified in some lower level formalism:

- possibly refinement diagrams at a lower level, or
- textual proofs in refinement calculus, Hoare logic, ..., or
- interactively by proof checkers, or
- automatically by automatic deduction or model checking, or
- just handchecked
- or by testing

Second layer construction



Hilbert proof (second layer)

5. $S_{00} \preceq S_{01}$ (prove)

6. $S_{10} \preceq S_{11}$ (prove)

7. $S_{00} \triangleleft S_{01} \sqsubseteq S_{10} \triangleleft S_{11}$ (prove)

8. $T_{00} \preceq T_{01}$ (prove)

9. $T_{10}[S_{00}] \sqsubseteq T_{10}[S_{00} \triangleleft S_{01}]$ (monotonicity)

10. $T_{10}[S_{00} \triangleleft S_{01}] \preceq T_{11}[S_{00} \triangleleft S_{01}]$ (prove)

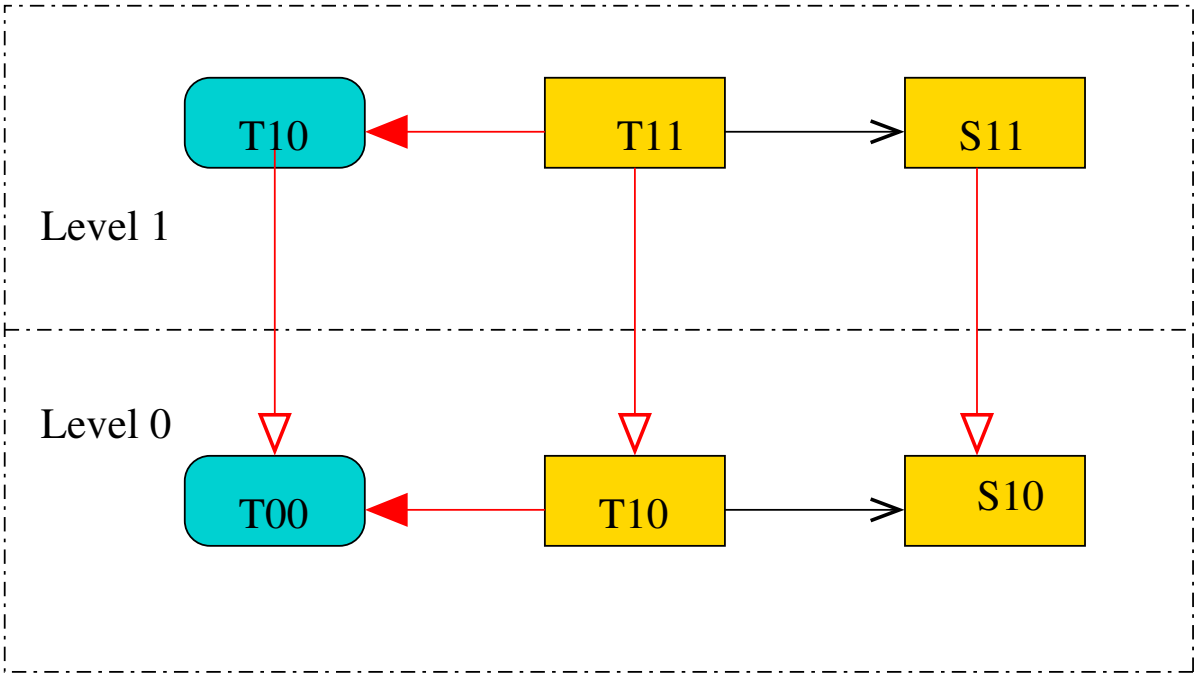
11. $T_{00} \triangleleft T_{01} \sqsubseteq T_{10}[S_{00} \triangleleft S_{01}] \triangleleft T_{11}[S_{00} \triangleleft S_{01}]$ (prove)

12. $T_{10}[S_{00} \triangleleft S_{01}] \triangleleft T_{11}[S_{00} \triangleleft S_{01}] \sqsubseteq T_{10}[S_{10} \triangleleft S_{11}] \triangleleft T_{11}[S_{10} \triangleleft S_{11}]$ (monotonicity)

13 $T_{00} \triangleleft T_{01} \sqsubseteq T_{10}[S_{10} \triangleleft S_{11}] \triangleleft T_{11}[S_{10} \triangleleft S_{11}]$ (transitivity)

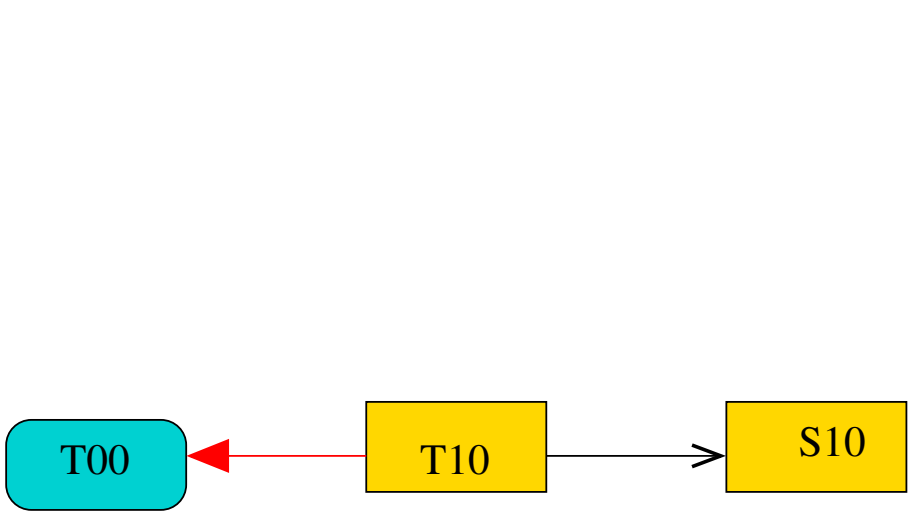
Derived system

The final system that we have derived is the following:

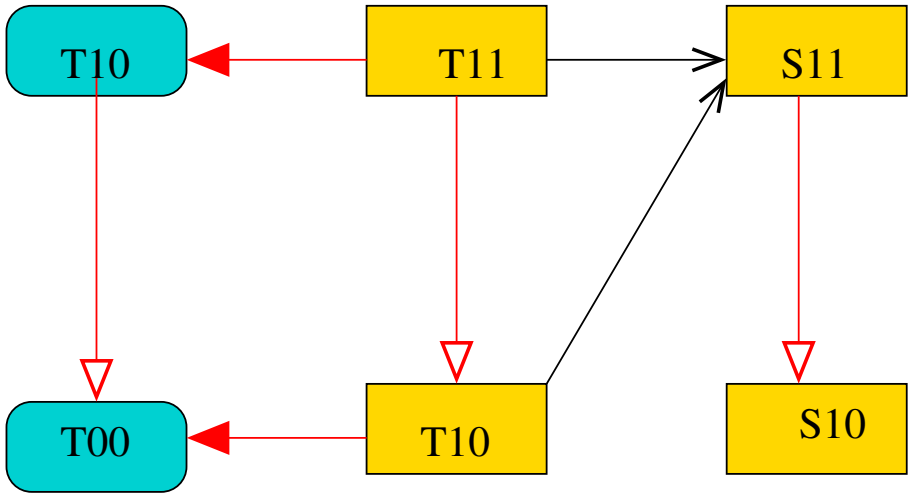


Two systems constructed

In fact, this describes two different systems, Layer 0 and Layer 1:



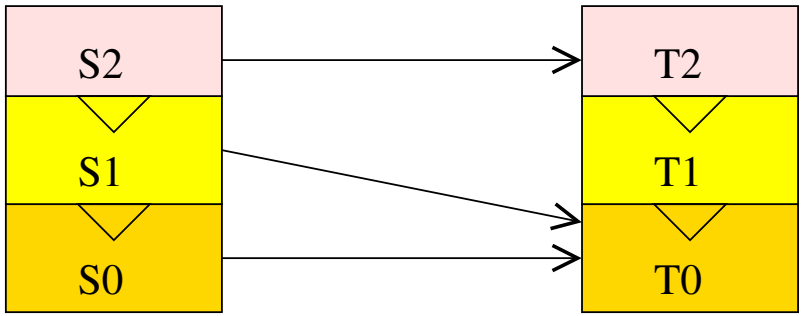
Level 0



Level 1

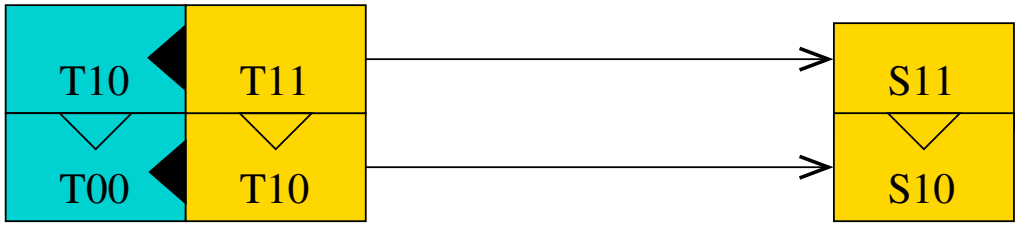
More compact notation

- Extension adds a new dimension to software diagrams.
- Diagrams can become quite large, so need to describe extensions in a more concise way
- One approach: stack extensions on top of each other.
- This is only notational abbreviations, it does not change the underlying logic of the derivations.



Layers and implementations

- We can also show implementations as boxes to the right of the original boxes.
- Example: the result of the derivation above can be compressed into the following figure:



- Considerably more compact than the previous derivation.
- Shows that the system essentially consists of two components, a T and an S component.

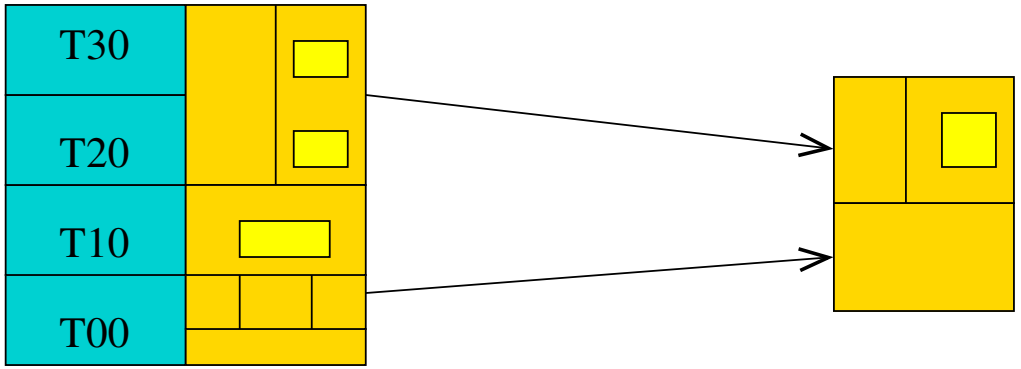
Layered components

- The figure shows that the T component is internally constructed in two layers, and for both layers we have a specification and an implementation.
- Similarly, the S component is constructed in two layers.
- We refer to these components as *layered components.*
- In addition, this figure shows that the bottom layer of the T component uses the bottom layer of the S component.
- Thus we can allow a layered component to be used on different levels, with increasing functionality.

- The T component also provides a layered specification of the component, and shows explicitly that there are two levels on which the component can be used.

Different layerings

- The layering of the different components does not have to be the same.
- Moreover, the layering of specifications and implementations need not be directly corresponding.



Remarks

- Here the specification T_{00} is implemented by two layers in the implementation, whereas T_{20} and T_{30} are implemented at the same time by a single layer.
- The layering of the S component is here independent of the layering of the T component.
- Moreover, the implementation of extensions T_{20} and T_{30} has been further optimized by providing further implementations.
- In addition, some of the implementations use private components, that are not visible to the outside.

Combining proximity and arrows

- The example has been constructed to exemplify the different possibilities for using proximity rather than arrows to indicate usage, implementation and extension.
- The abbreviated notation is convenient when we are describing simple structures, but can be restrictive for larger systems. is quite restrictive and can also be ambiguous.
- In more complex situations we will need both proximity and arrows.
- Proximity gives compactness, arrows give generality.

SOFTWARE EVOLUTION

- Software evolution and proofs
- Redesigning software
- Version control

Evolution over time

- The refinement diagrams model the evolution of software over time by numbering the inference steps.
- Each new inference step increases the (logical) time counter by one.
- This time dimension is then the same as the step number in a Hilbert like proof system.
- The fact that the time steps correspond to proof steps help maintain consistency of the construction:
  - we cannot refer to a part that has not been constructed yet or to a relation that has not yet been established

Growth of refinement diagrams

- The construction of software can be played back like a movie, showing how each step adds to the construction.
- Only permitted to add elements; we do not permit any elements to be removed from a refinement diagram.
- Over time the diagram will be filled with elements that are not needed anymore.
  - stepping stones in the derivation that have served their purpose, or
  - alternative approaches that we have abandoned

Hiding details

- Parts of the diagram that reflect the historic development but are not relevant now may be hidden, but not removed– they may still be needed later.
- A step in the derivation that can usually be ignored may have to be revisited,
  - if we find an error in the proof,
  - or if we are considering an alternative development that could be based on this version.
- Keeping the trail of the software development may be useful for auditing purposes, for certification purposes, or for backup purposes.

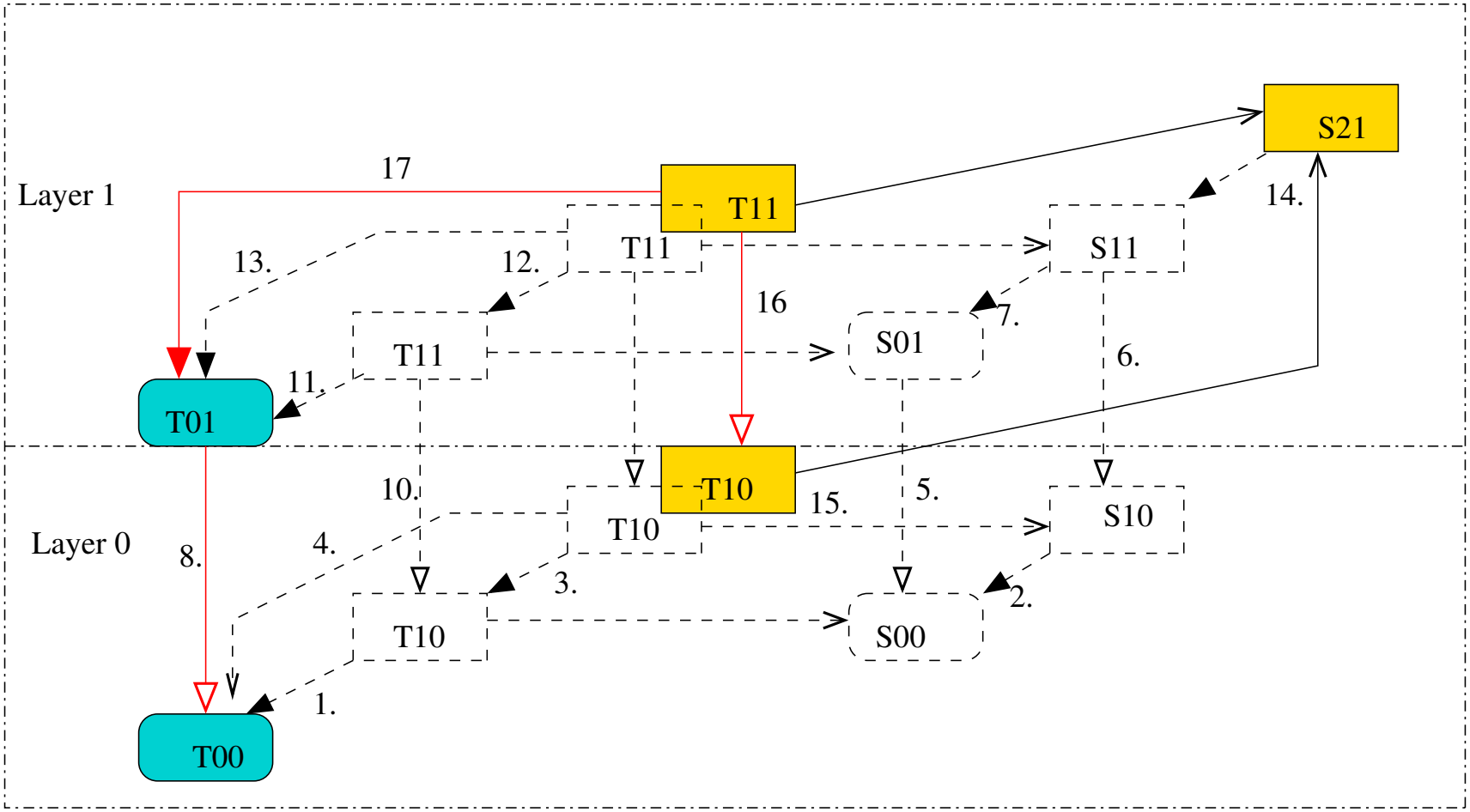
Redesign of system

- In practice, it is often necessary and desirable to *redesign* the system, i.e., change the software architecture without necessarily changing the functionality of the system.

- This means that the refinement diagram is extended with new elements, and some of the old elements become obsolete.

- These obsolete elements are not, however, removed. They remain in the diagram, but are on paths that will be ignored in later construction phases.

Example redesign



Continued derivation

14. We decide that the implementation $S_{10} \triangleleft S_{11}$ is too inefficient or too complicated, and we want to improve it by implementing the S component directly without layering. For this purpose, we introduce a new class S_{21}. We show that this new class is a correct implementation of $S_{10} \triangleleft S_{11}$.

15. Because of this, we are now allowed to deduce that T_{10} using S_{10} is refined by T_{10} using S_{21}instead (this requires two applications of monotonicity and one application of transitivity)

16. Next, we show that T_{11} using $S_{21}$is a correct extension of T_{10} using S_{21}.

17. Finally, we show that $T_{00} \triangleleft T_{01}$ is correctly refined by $T_{10}[S_{21}] \triangleleft T_{11}[S_{21}]$.

Remarks

- We started with a quite strict layered construction
- Having managed to get this to work, we decided that we needed a more efficient version, so we refactored the system by reimplementing the layered component $S_{10} \triangleleft S_{11}$ by a non-layered component S_{21}.
- The T components were changed to use the new component instead, and we showed that the changed T components still satisfied the layered specification.
- We did not change the layered structure of the T component .

Associated information

- The refinement diagrams emphasise the structure of software, but do not properly discuss the information associated with the different structure elements.
- Typically, we would associate program code with the components, as well as other information (e.g., protection).
- We would associate proofs with refinement and extension arrows, in addition to, e.g., abstraction relations with refinement arrows.
- We may also associate test sets (e.g., automatic unit tests) with the implementation and extension arrows.

- We could associate usage restrictions (e.g., method preconditions) with usage arrows (or with the methods themselves), and so on.

A software editor

- Currently implementing a software editor where the software is described as a refinement diagram
- Editor also provides more compact views of the software, along the lines explained above
- Allows code to be associated directly with the parts, and to execute a refinement diagram
- Allows proofs / tests to be carried out in order to check the correctness of the refinement arrows.
- Provides a *version control system* system based on refinement diagrams.

Software editor design

- A specialized editor constructs and browses the refinement diagram
- Other specialized editors are used to construct and inspect the information associated with the structure elements:
  - source code editor for writing program text,
  - interactive proof editor for checking correctness of refinement steps,
  - unit test framework to execute the tests automatically,
  - compiler for executing the software system, ...

Tentative design

- The refinement diagram as presented above essentially equate a refinement arrow with a true refinement proposition.
- In practice, one may want to make a difference between a refinement arrow that should be true (the intention) and one that has been shown to be true (the established fact).
- We can decorate an *tentative entity (*part or usage or refinement arrow) with a question mark, and an established refinement arrow with an exclamation mark.
- The exclamation mark may further be qualified by the way in which the

truth of the refinement has been established: by inspection, by testing, by a manual proof, or by a formal, possibly machine checked proof.

Committing entities and relations

- Tentative elements may be changed at will, and even deleted.
- When the relation has been established, it is *committed* to the diagram, and cannot be changed any more
- The committed diagram elements provide the historic trace of the construction, a k o *version control system*.

Applications for incremental software construction

- **Stepwise feature introduction**: a *programming technique* and a *software architecture*
- **Extreme programming**: a *software process* that supports incremental construction
- **The ladder process**: incremental *software design*
- **Automated unit testing**: incremental *testing of superposition*

Conclusions

- We have above shown how to extend the refinement calculus with a diagrammatic notation that allows large software systems to be constructed in a rigorous and (in our opinion) quite intuitive way.
- The refinement diagrams that we introduce for this purpose are essentially tools to reason about lattice elements, but can be used for software by interpreting software components as elements in lattices, as is done in refinement calculus.
- We have shown that the refinement diagram proofs are equivalent to Hilbert like proofs in a lattice theory.
- We have applied this framework to analyze a collection of important problems in software engineering.

- The importance of specifications has been highlighted, and we have shown the importance of specifications when deriving large systems.

- We have also discussed the rational for the information hiding principle when constructing large software systems, that this principle should be used when applicable, but that there are situations when it should not be used.

- We have also shown how to formalize and reason about systems that are built by extension layers, where the layering is based on inheritance and dynamic or static binding.

- We have proposed a new software construct, layered component, and discussed how to reason about such components.

- Finally, we have described how the refinement diagram proofs provide a high level view of the evolution of the software system, and that a version control system could be based on this kind of diagrams.

Acknowledgments

A number of colleagues have been very helpful in discussing the issues described here. In particular, I want to thank Marcus Alanen, Johannes Eriksson, Luka Milovanov, Herman Norrgrann, Viorel Preoteasa, and Joakim von Wright.

References

[1] Ralph-Johan Back. Refinement diagrams. In J. M. Morris and R. C. Shaw, editors, *4th Refinement Workshop*, Workshops in Computer Science, pages 125–137. Springer-Verlag., 1991.

[2] Ralph-Johan Back. Software construction by stepwise feature introduction. In Bowen J.P. Henson M.C. Robinson K. Bert, D., editor, *ZB 2002: Formal Specification and Development in Z and B, proceedings of the 2nd International Conference of B and Z Users*, LNCS, pages 162–183, Grenoble, France, January 2002. Springer Verlag. Also appeared

as TUCS Technical Report 496.

[3] Ralph-Johan Back, Leonid Mikhajlov, and Joakim von Wright. Formal semantics of inheritance and object substitutability. Technical Report 337, TUCS - Turku Centre for Computer Science, Turku, Finland, March 2000.

[4] Ralph-Johan Back, Anna Mikhajlova, and Joakim von Wright. Reasoning about interactive systems. In J. Woodcock J. Wing and J. Davies, editors, *Proc. of the World Conference on Formal Methods (FM'99),Toulouse, France.*, volume 1709 of *Lecture Notes in Computer Science*, pages 1460 – 1476. Springer-Verlag, 1999.

[5] Ralph-Johan Back and Kaisa Sere. Stepwise refinement of action systems. *Structured Programming*, 12:17–30, 1991.

[6] Ralph-Johan Back and Joakim von Wright. *Refinement Calculus: A Systematic Introduction*. Springer-Verlag, 1998.

Related and earlier work

- An earlier version of refinement diagrams has been described in [1].
- The present version is influenced by UML class diagrams. The way we reason about refinement diagrams is obviously also influenced by category theory diagrams.
- The new thing here is to use refinement calculus as the underlying logic for the diagrams, and to use the diagrams to reason about software architecture and correct refinement.
- The theory described here is intended to support the *stepwise feature introduction methods* [2] for constructing layered software, where each layer introduces only one new feature in the system. We are not in this

paper going into ways for modelling more complicated software notions in the refinement calculus, like concurrent and interactive systems, or object oriented systems.

- Some references are to be found in [4, 5, 3] .

Recursion rule

Consider first a system with a single recursive component. We assume that we have a specification S_0 of a component, and we want to implement this with the component $(\mu X \cdot S_1[X])$. We can use the following induction principle to reason about properties of a fixed point.

Assume that f is a monotonic function on a lattice, and that we have a monotonically increasing sequence $x_0 = \bot \sqsubseteq x_1 \sqsubseteq x_2 \sqsubseteq \ldots$ such that $x = \bigsqcup_{i=0}^{\infty} x_i$. Assume further that

$$x_{n+1} \sqsubseteq f.x_n$$

holds for any $n \geq 0$. We then have that

$$x \sqsubseteq \mu.f$$

This result is the basis for a proof rule for recursive procedures described in [6].

The proof of this is a rather simple exercise in lattice theory. We first show that $x_n \sqsubseteq \mu.f$ holds for any $n \geq 0$. We prove this by induction. For $n = 0$, we have $x_0 = \bot \sqsubseteq \mu.f$. Next, assume that $x_n \sqsubseteq \mu.f$ holds. We then have that

$$x_{n+1} \sqsubseteq f.x_n \sqsubseteq f.(\mu.f) = \mu.f$$

From this then follows that

$$\bigsqcup_{n=0}^{\infty} x_n \sqsubseteq \mu.f$$

on associated with the different structure elements. Typically, we would associate program code with thas the limit is the least of all fixed points of the sequence. (Note: for the general case, we need to carry on the argument to transfinite induction).

This result would be used in the following way. The sequence $x_0 = \bot \sqsubseteq x_1 \sqsubseteq x_2 \sqsubseteq \ldots$ provides better and better approximations of the specification of the component, such that $x = \bigsqcup_{n=0}^{\infty} x_n$ is the complete specification of the component. We prove for an arbitrary approximate specification x_{n+1}, that $x_{n+1} \sqsubseteq f.x_n$ holds. In other words, the specification x_{n+1} can be replaced by the body of the component, which calls some specification lower down in the approximation hierarchy. This means that any sequence of unfoldings will eventually terminate.