INCREMENTAL SOFTWARE CONSTRUCTION WITH REFINEMENT DIAGRAMS

Ralph-Johan Back
Abo Akademi University

August 21, 2006

Home page: www.abo.fi/~backrj

Research / Current research / Incremental Software Construction

CREST
CENTRE FOR RELIABLE SOFTWARE TECHNOLOGY

Overview of Lectures

1. Incremental software construction
2. Refinement diagrams and diagrammatic reasoning
3. Reasoning about software components
4. Reasoning about software extension
5. Advantages of duplication
6. Software evolution

CREST
CENTRE FOR RELIABLE SOFTWARE TECHNOLOGY

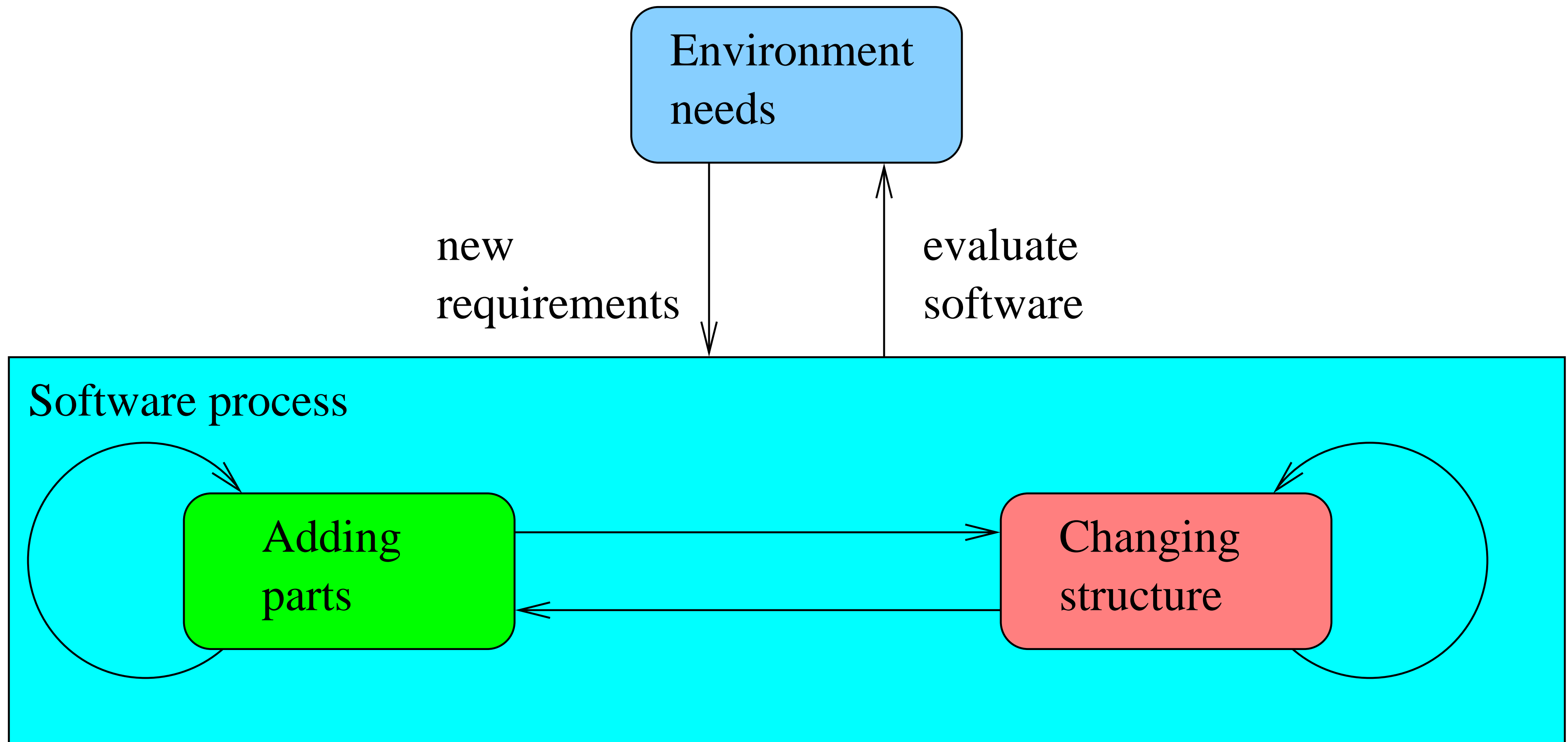
Incremental software construction

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
- However, adding increments accumulates *design errors*, which must be corrected by more or less frequent software *redesigns (refactorings)*

CREST
CENTRE FOR RELIABLE SOFTWARE TECHNOLOGY

Software evolution model



CREST
CENTRE FOR RELIABLE SOFTWARE TECHNOLOGY

Basic questions

- What is a suitable *conceptual model* for software and its evolution? ★
- What is a suitable *software architecture* to support evolving software?
- How do we *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?

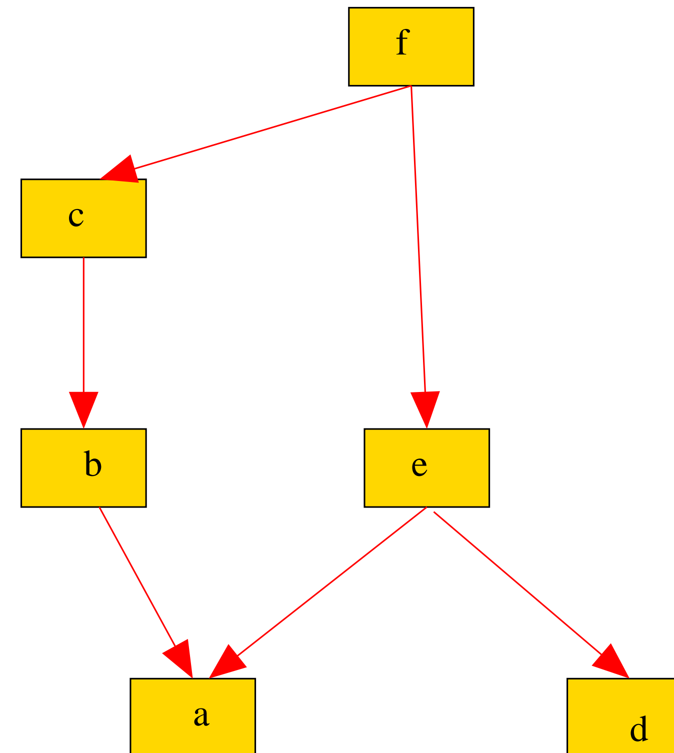
CREST
CENTRE FOR RELIABLE SOFTWARE TECHNOLOGY

Overview of Lectures

1. Incremental software construction
2. Refinement diagrams and diagrammatic reasoning
3. Reasoning about software components
4. Advantages of duplication
5. Reasoning about software extension
6. Software evolution

Posets

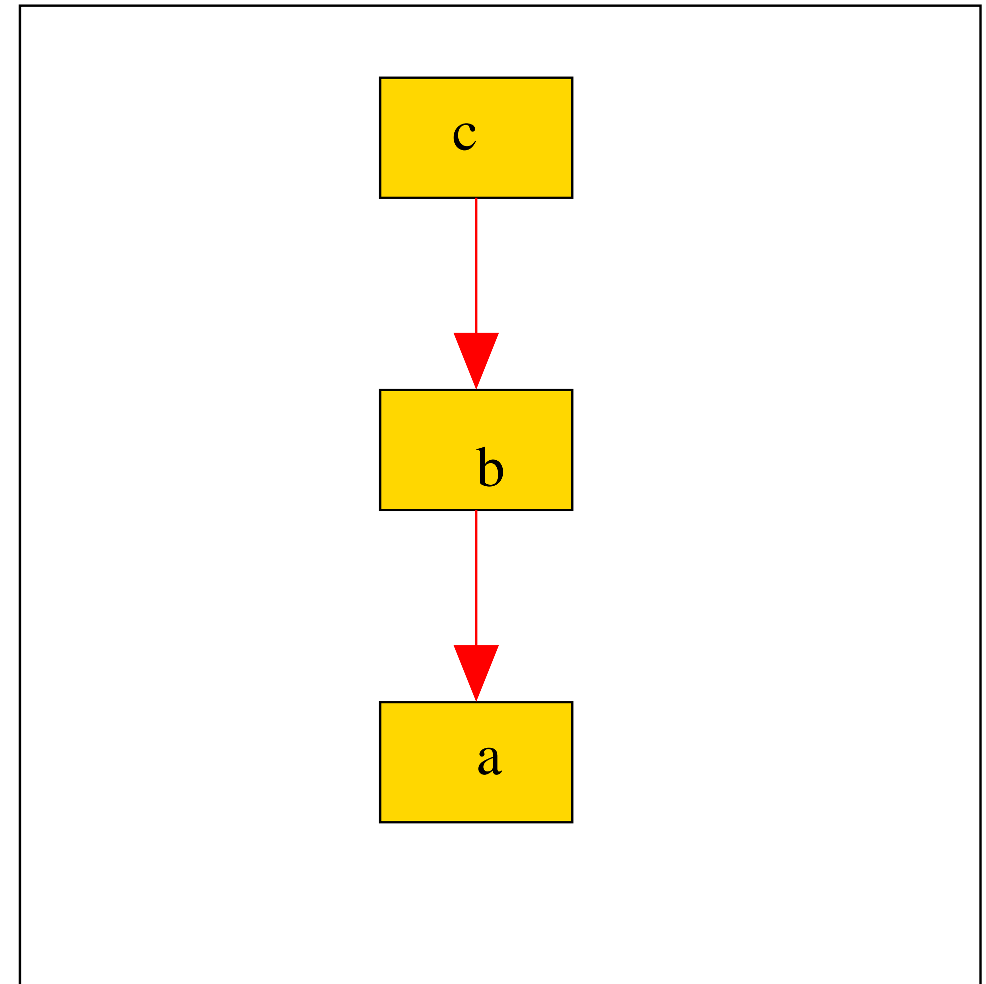
A *partially ordered set* (*poset*) is a set A together with an ordering $\sqsubseteq$ that is *reflexive*, *transitive* and *antisymmetric*.



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

The arrow describes the lattice ordering, e.g., $a \sqsubseteq b$ (arrow goes from larger to smaller element). We refer to is as refinement.

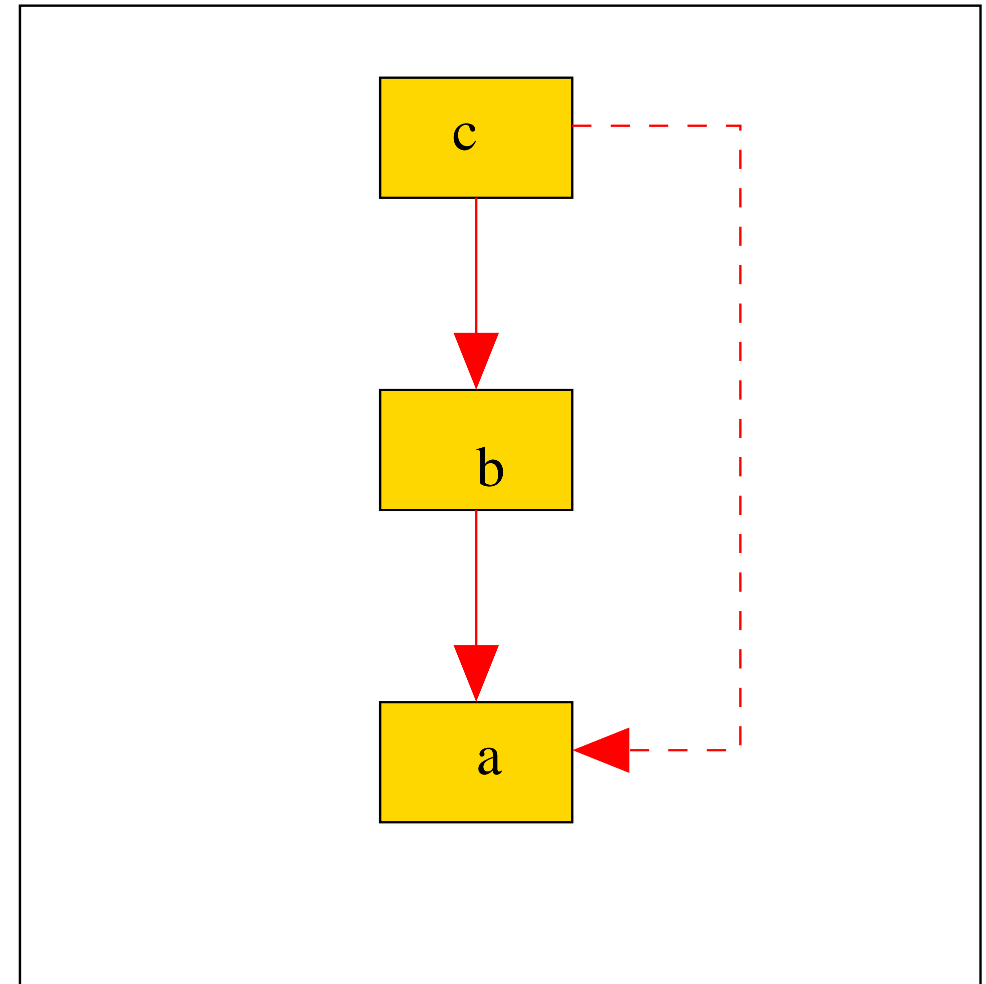
Transitivity rule (assumption)



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

CREST
CENTRE FOR RELIABLE SOFTWARE TECHNOLOGY

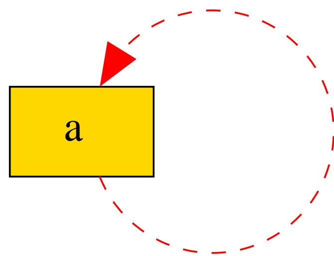
Transitivity rule (conclusion)

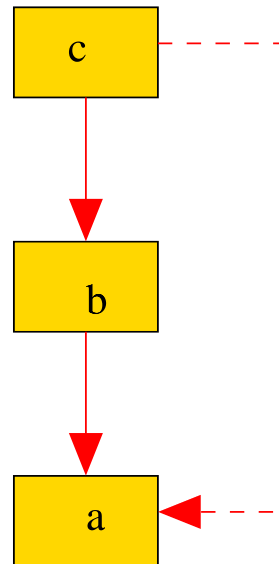


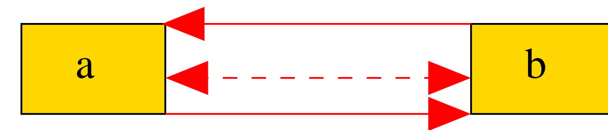
then we may deduce that $a \sqsubseteq c$

Refinement diagram rules

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







Reflexivity

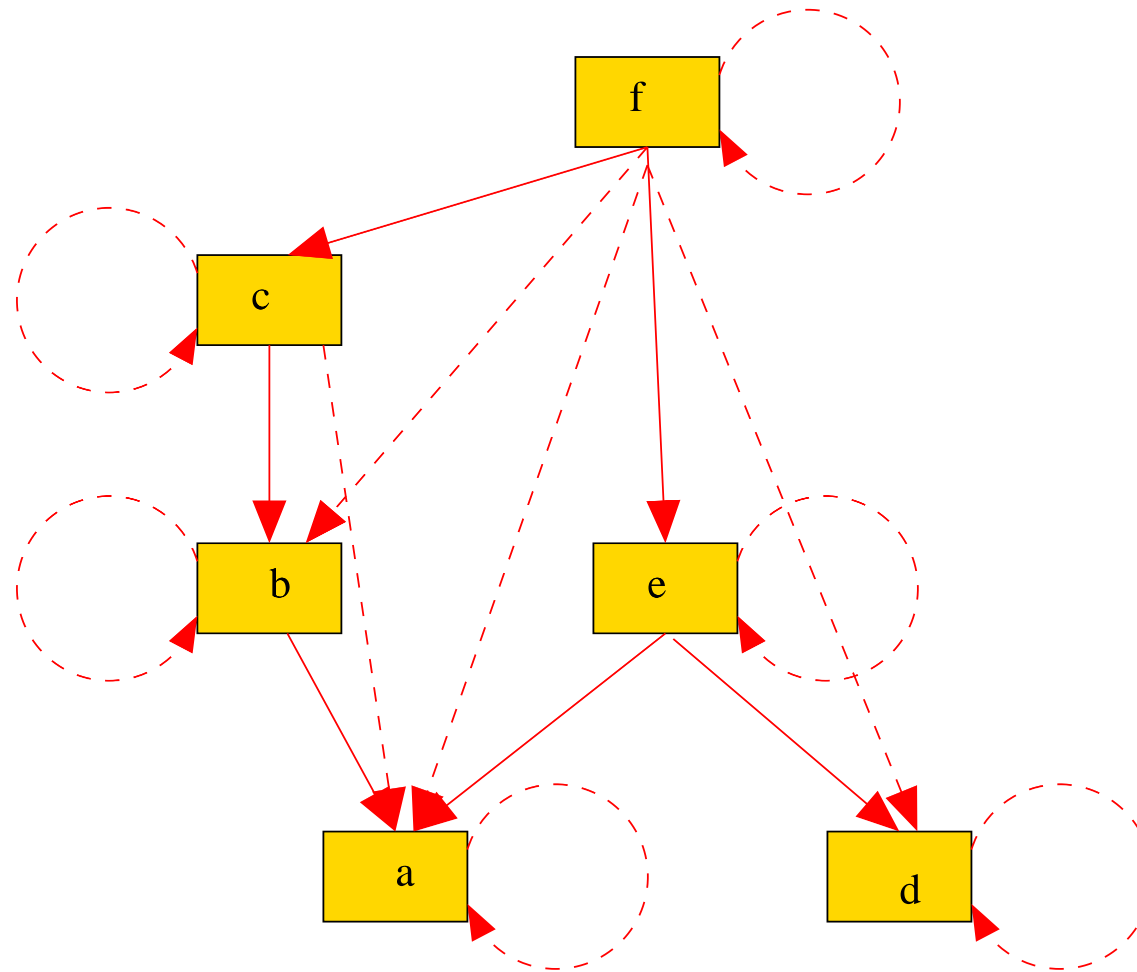
Transitivity

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.
- We use a double arrow to indicate equality of lattice elements.

Adding reflexivity and transitivity arrows



CREST
CENTRE FOR RELIABLE SOFTWARE TECHNOLOGY

Intended use of refinement diagrams

Refinement diagrams are intended to model software components and their relationships

- the 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.

CREST
CENTRE FOR RELIABLE SOFTWARE TECHNOLOGY

Instantiating refinement diagrams

Refinement calculus:

- Statements, procedures with algorithmic refinement
- Abstract data types, classes with data refinemenet and superposition refinement
- Action systems with trace refinement and superposition refinement

Other formalisms:

- CSP with trace refinement, ...
- Statements as relations with inverse implication
- etc.

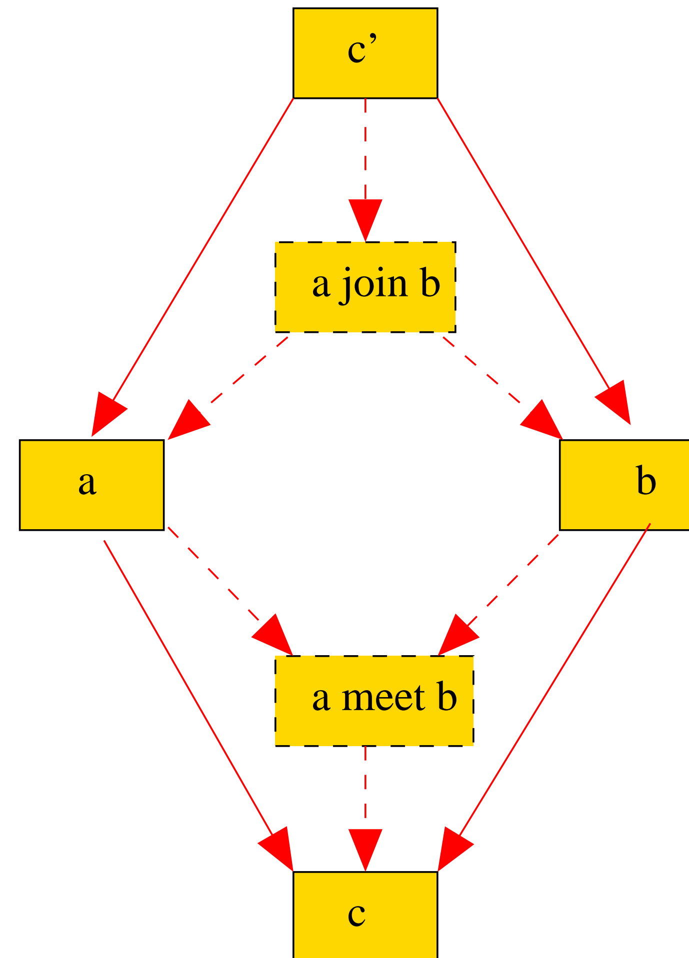
Lattices

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*) $a \sqcap b$.

- $a \sqsubseteq a \sqcup b$ 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)*

We will in fact assume that all our lattices are complete, i.e., any set of lattice elements have a meet and a join in the lattice.

Lattice properties as refinement diagram rules



Join properties:

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

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

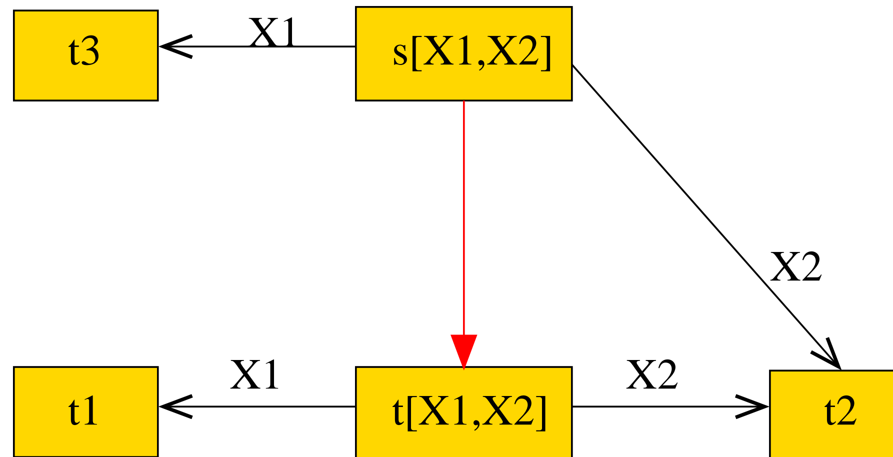
Terms and dependencies

- A *lattice term* is constructed by applying lattice operations to constants and variables that range over lattice elements.
- We write a lattice term as $t[X_1,...,X_m]$ to indicate that it depends on the variables $X_1,...,X_m$.
- 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 $t[t_1,t_2]$ is expressed in a refinement diagram as

- Can also write the term using explicit binding: $t[X_1,X_2][X_1 := t_1, X_2 := t_2]$

Refinement ordering and terms

Refinement between terms relate the meaning of terms, not the terms themselves:



Here the refinement arrow indicates that

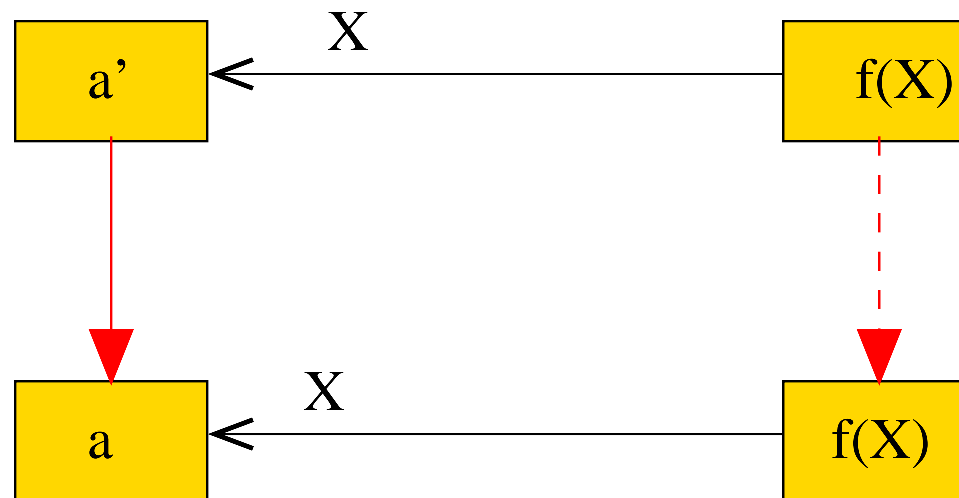
$$t[t_1, t_2] \sqsubseteq s[t_3, t_2]$$

CREST
CENTRE FOR RELIABLE SOFTWARE TECHNOLOGY

Monotonicity

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

$$a \sqsubseteq_A a' \Rightarrow f(a) \sqsubseteq_B f(a')$$



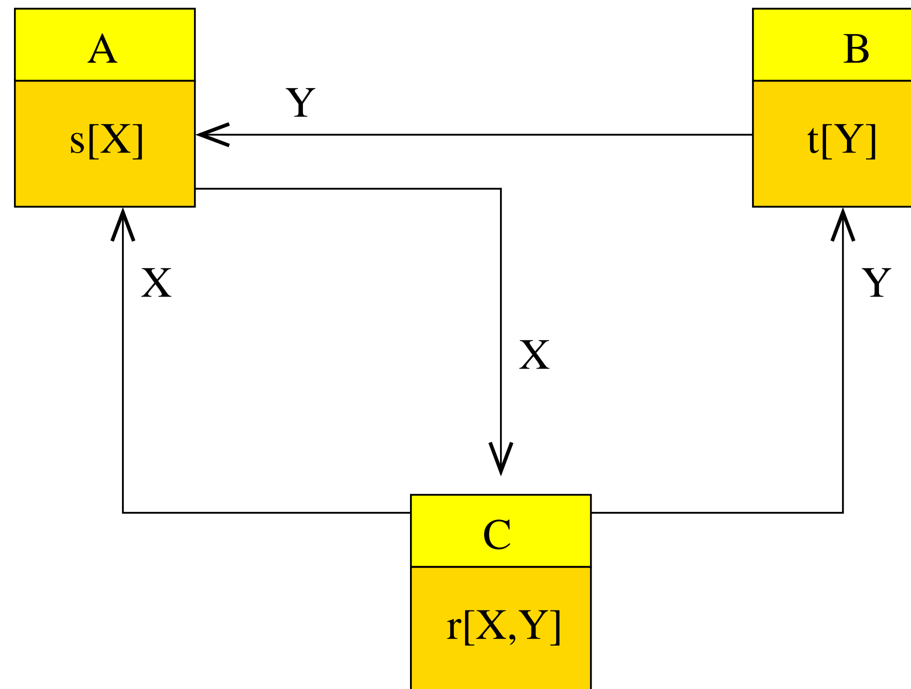
CREST
CENTRE FOR RELIABLE SOFTWARE TECHNOLOGY

Monotonic terms

- Assume a collection of *constants* that denote specific lattice elements.
- 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.

Circular dependencies

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

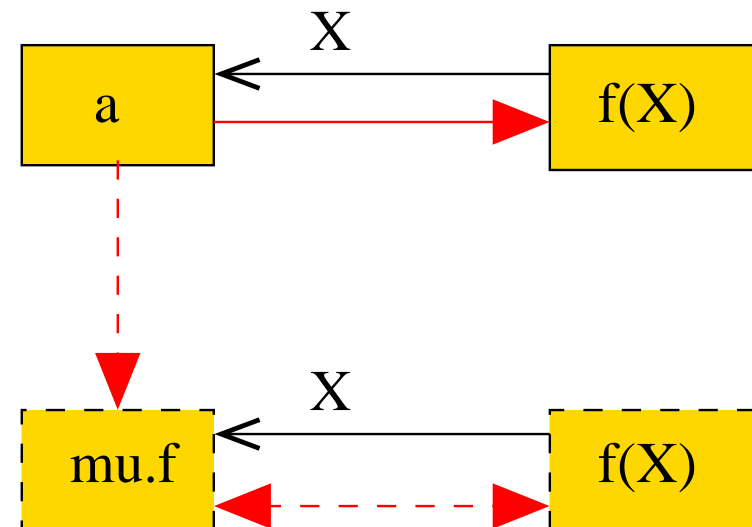


- Components $A = s[X]$, $B = t[Y]$, and $C = r[X, Y]$
- Bind X to C in $s[X]$, Y to A in $t[Y]$, and X to A, Y to B in $r[X, Y]$

Least fixed point

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

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

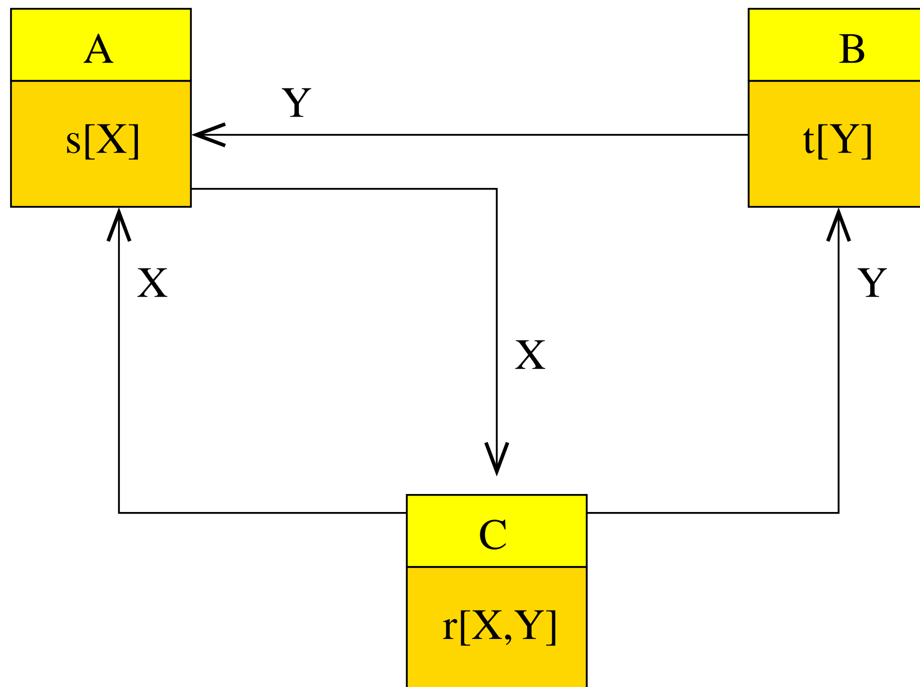


- $f(\mu.f) = \mu.f$ (i.e., $\mu.f$ *is a fixed point)*
- $f(a) \sqsubseteq a \Rightarrow \mu.f \sqsubseteq a$ *(least fixed point induction)*

CREST
CENTRE FOR RELIABLE SOFTWARE TECHNOLOGY

Environment mapping

Environment E maps part names to their terms, once bindings have been resolved.



$$\begin{aligned} E(A) &= s[C] \\ E(B) &= t[A] \\ E(C) &= r[A,B] \end{aligned}$$

CREST
CENTRE FOR RELIABLE SOFTWARE TECHNOLOGY

System as fixpoint of environment mapping

Environment mapping E

$$E\left(\begin{bmatrix} A \\ B \\ C \end{bmatrix}\right) = \begin{bmatrix} s[C] \\ t[A] \\ r[A,B] \end{bmatrix}$$

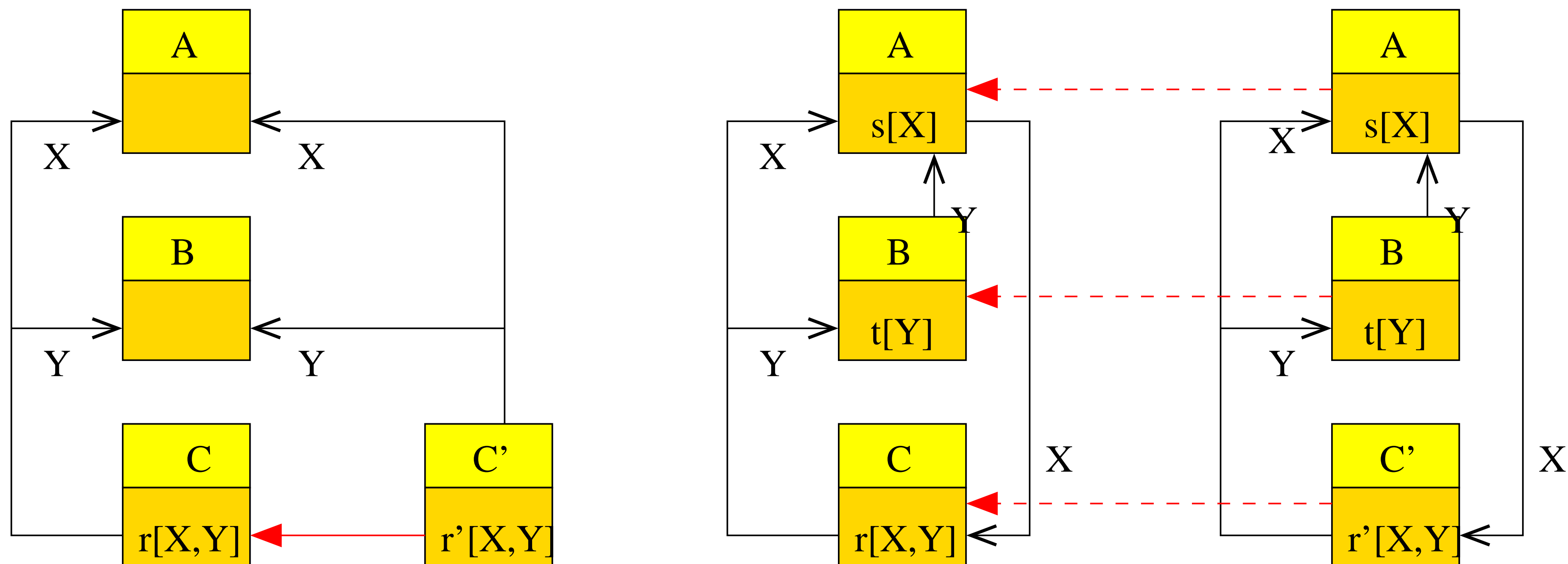
System is least solution to this equation, defined as μE (least fixed point of E).

$$\mu E = \begin{bmatrix} s^* \\ t^* \\ r^* \end{bmatrix}$$

Thus $(A,B,C) = (s^*,t^*,r^*)$ gives the meaning of the parts. The meanings are essentially the infinite unfoldings of the recursive definitions.

Local refinement gives global refinement

Refining a component will lead to a refinement of the system as a whole. Assume that we refine $r[X,Y]$ to $r'[X,Y]$. Then the meanings of the original system will be refined by the meanings of the new system.



- The empty box stands for an arbitrary term.

CREST
CENTRE FOR RELIABLE SOFTWARE TECHNOLOGY

Diagrammatic reasoning

Refinement diagrams provide a visual way of reasoning about lattice elements. The three main ingredients are:

Refinement diagrams: describe a collection of lattice elements, how they depend on each other and how they are ordered (described above)

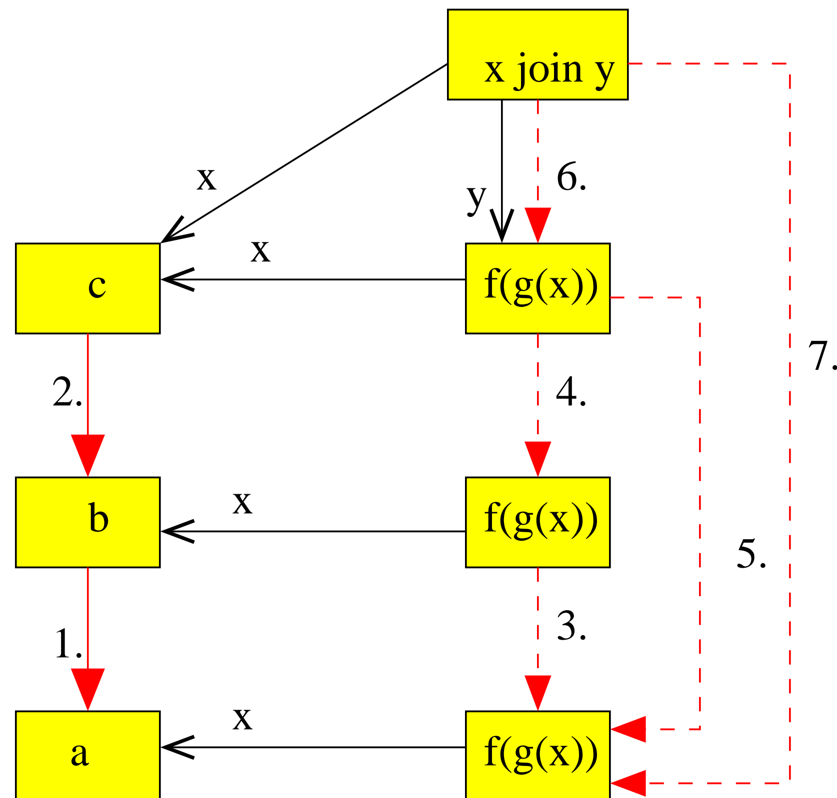
Refinement diagram rules: describe how to add new entities and ordering relations to a refinement diagram (described above)

Refinement diagram derivations: record the order in which the entities and ordering relations have been introduced (described next)

Refinement diagram derivation

1. A refinement diagram derivation is a refinement diagram where the ordering arrows are numbered by consecutive integers
2. The integers show the order in which the relations have been introduced in the diagram.
3. 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.
4. New entities may only be introduced into the diagram if justified by some proof rule.
5. No entities may ever be removed from the diagram.
6. The proof rules used in the diagram can be textually defined proof rules, or they can be refinement diagram rules.

Example



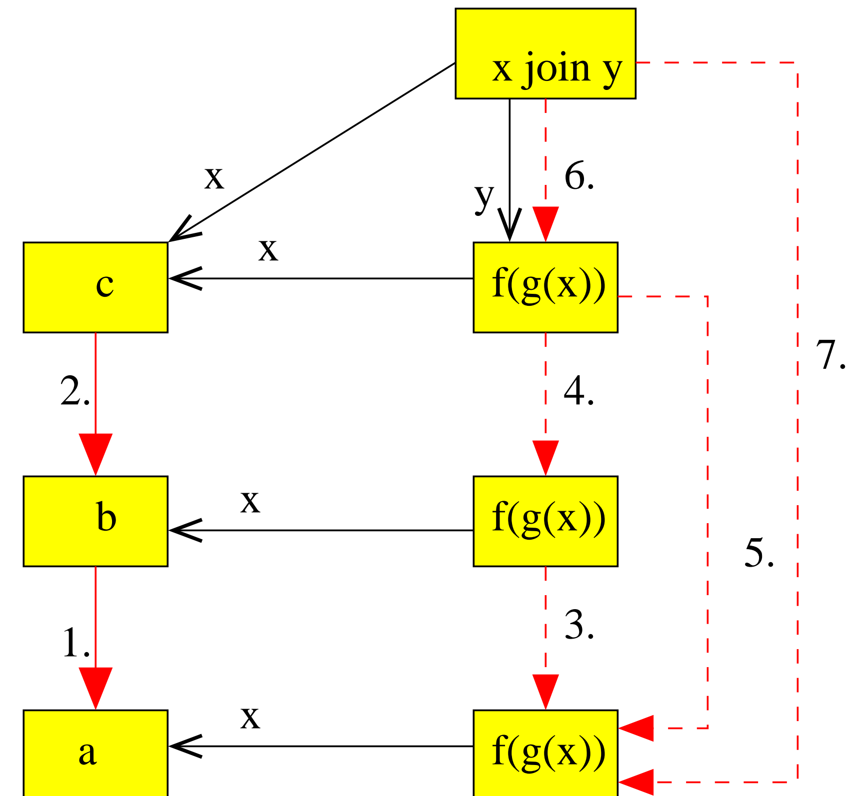
- Dashed arrows were inferred by inference rules.

Refinement diagram derivations and Hilbert-like proofs

- There is an equivalent textual presentation of a refinement diagram derivation, in the form of a Hilbert-like proof in 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)



CREST
CENTRE FOR RELIABLE SOFTWARE TECHNOLOGY

Software construction in the large and in the small

- Refinement diagrams used for software construction "in the large"
- Justification for each step may require quite a lot of work, and amounts 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.

CREST
CENTRE FOR RELIABLE SOFTWARE TECHNOLOGY

Alternative formalization

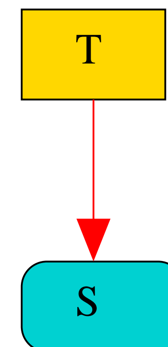
- The above formalization assumes that the software parts are always well formed and internally consistent.
- We can emphasize the construction of a well formed and consistent software part by introducing a separate judgment for this, e.g., $\vdash S$, that states that S *is consistent*.
- Then the diagrammatic 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 *check* that it satisfies its requirements.

Overview of Lectures

1. Incremental software construction
2. Refinement diagrams and diagrammatic reasoning
3. Reasoning about software components
4. Advantages of duplication
5. Reasoning about software extension
6. Software evolution

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
- Both specifications and implementations are parts

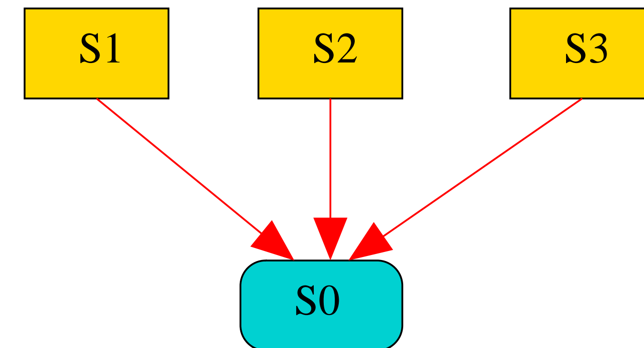


- A specification S is satisfied by an implementation T, if $S \sqsubseteq T$

CREST
CENTRE FOR RELIABLE SOFTWARE TECHNOLOGY

Multiple implementations

A specification S_0 can be satisfied by more than one implementation,
$S_0 \sqsubseteq S_1$, $S_0 \sqsubseteq S_2$, , $S_0 \sqsubseteq S_3$



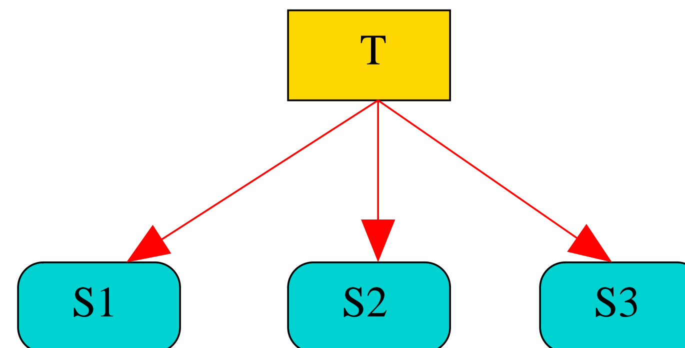
- S_0 could be a *standard* for some component, and S_1,S_2, S_3 could be different implementations of this standard which are provided by different vendors.

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.

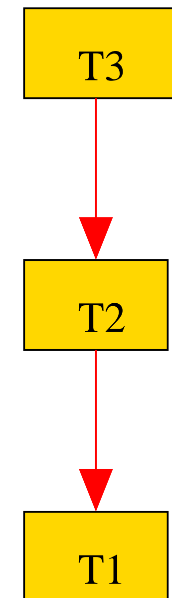
- A banking application may provide one interface for the bank customer and another interface for the bank clerk.



CREST
CENTRE FOR RELIABLE SOFTWARE TECHNOLOGY

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 $T_1 \sqsubseteq T_2$.

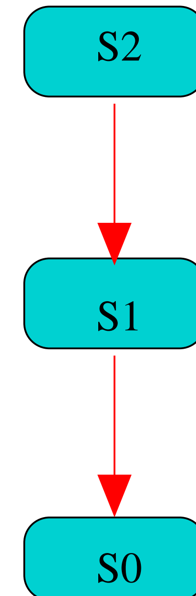


For instance,

- 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

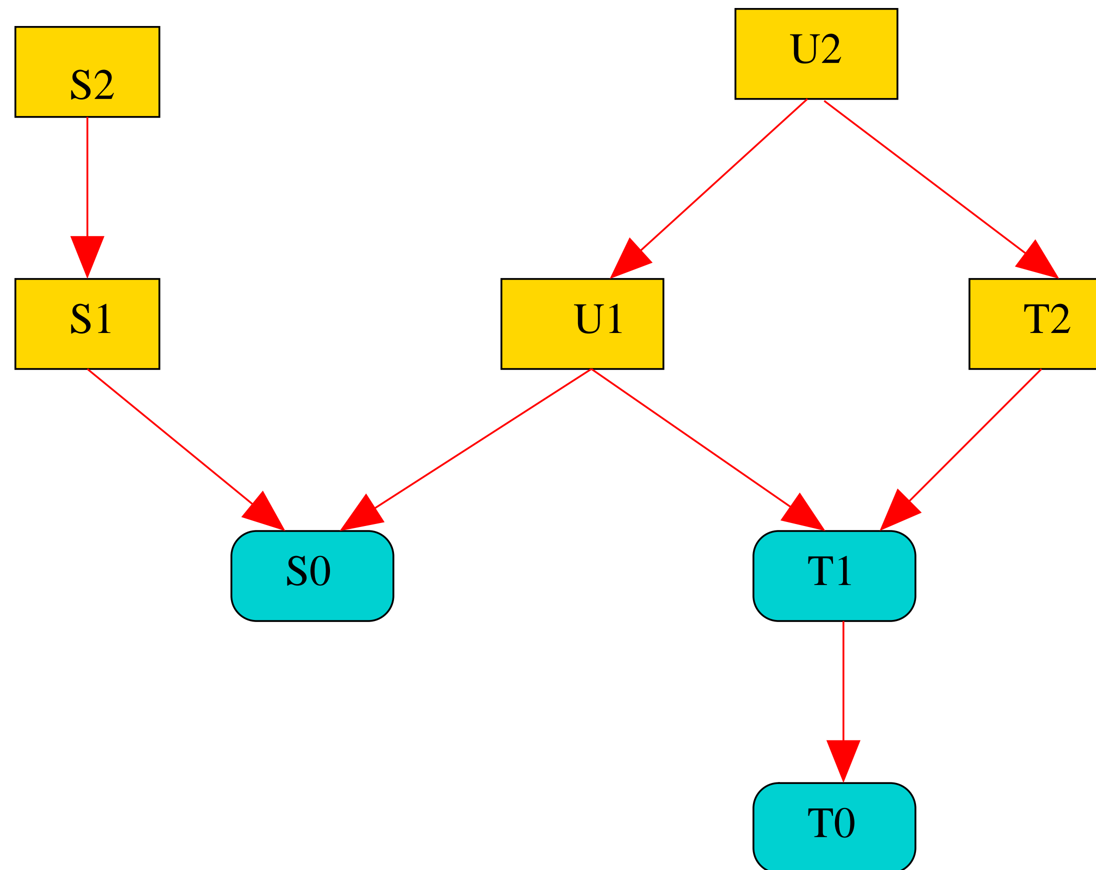


It is also possible that we have refinement between specifications, $S_1 \sqsubseteq S_2$.

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

CREST
CENTRE FOR RELIABLE SOFTWARE TECHNOLOGY

Specifications and implementations



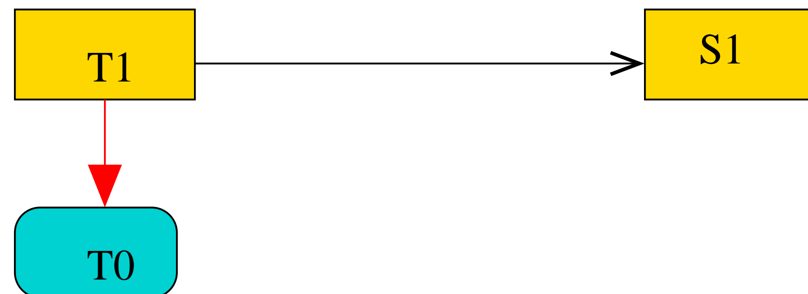
CREST
CENTRE FOR RELIABLE SOFTWARE TECHNOLOGY

Modularity and information hiding

- Specifications allow us to *modularize* software systems.
- Information hiding: a component knows only the specifications of another components
- The implementation of the used component can then be changed at will, as long as it still satisfies its original specification.

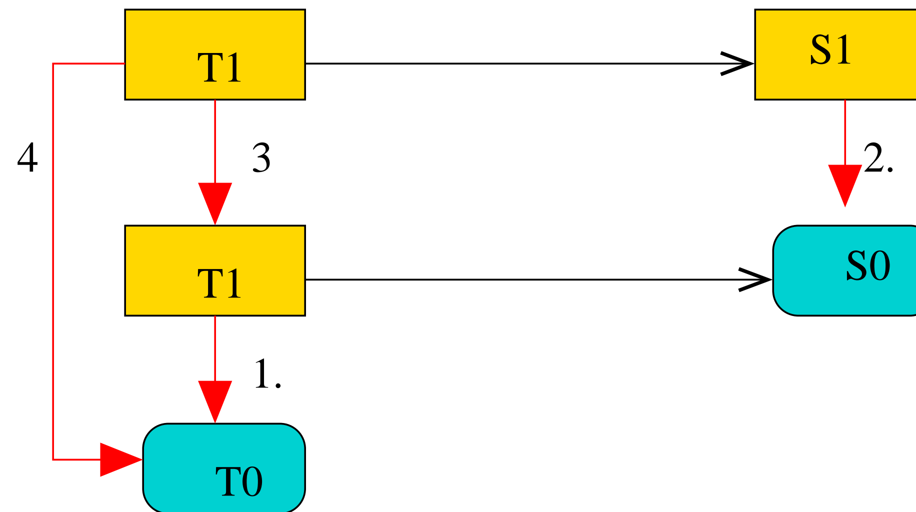
Example 1: Constructing systems with specifications

- 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.



CREST
CENTRE FOR RELIABLE SOFTWARE TECHNOLOGY

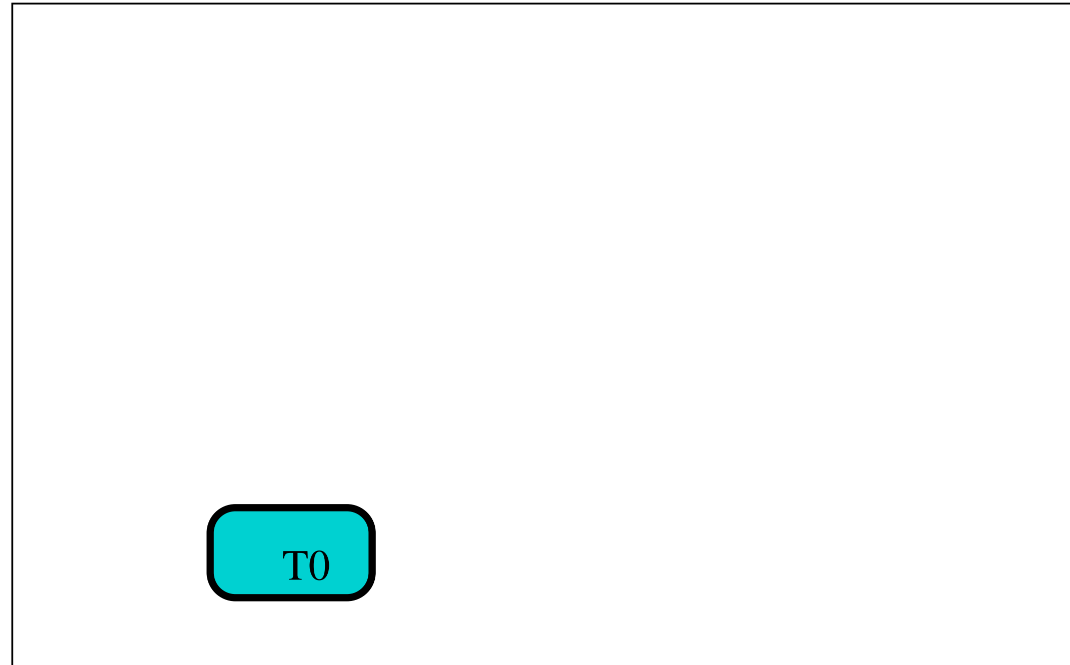
Refinement diagram derivation



- 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.

CREST
CENTRE FOR RELIABLE SOFTWARE TECHNOLOGY

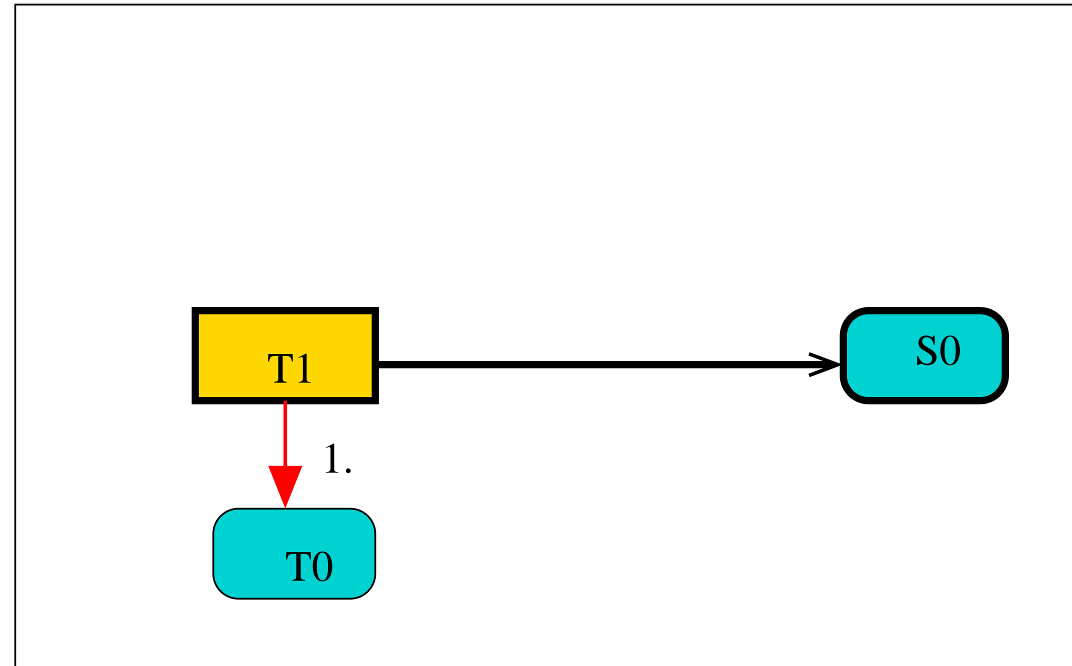
Animation of this construction, step 0



- Initially only the specification T_0 is provided

Step 1

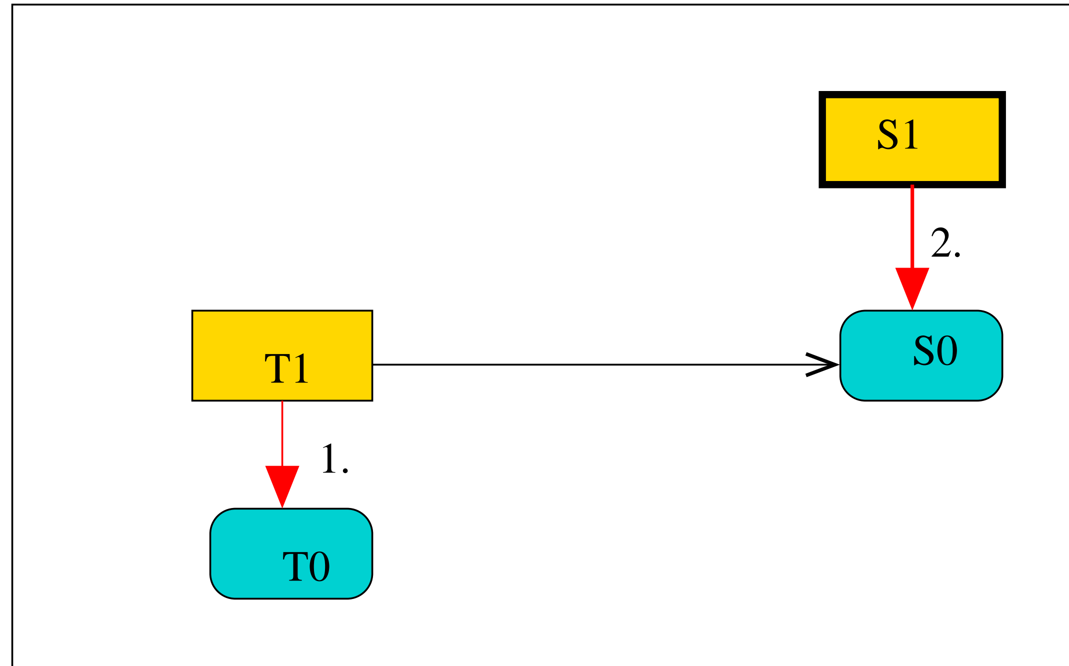
- We provide the specification of an auxiliary part S_0 and
- an implementation $T_1[S_0]$ of T_0.
- We show that this is a correct implementation.



CREST
CENTRE FOR RELIABLE SOFTWARE TECHNOLOGY

Step 2

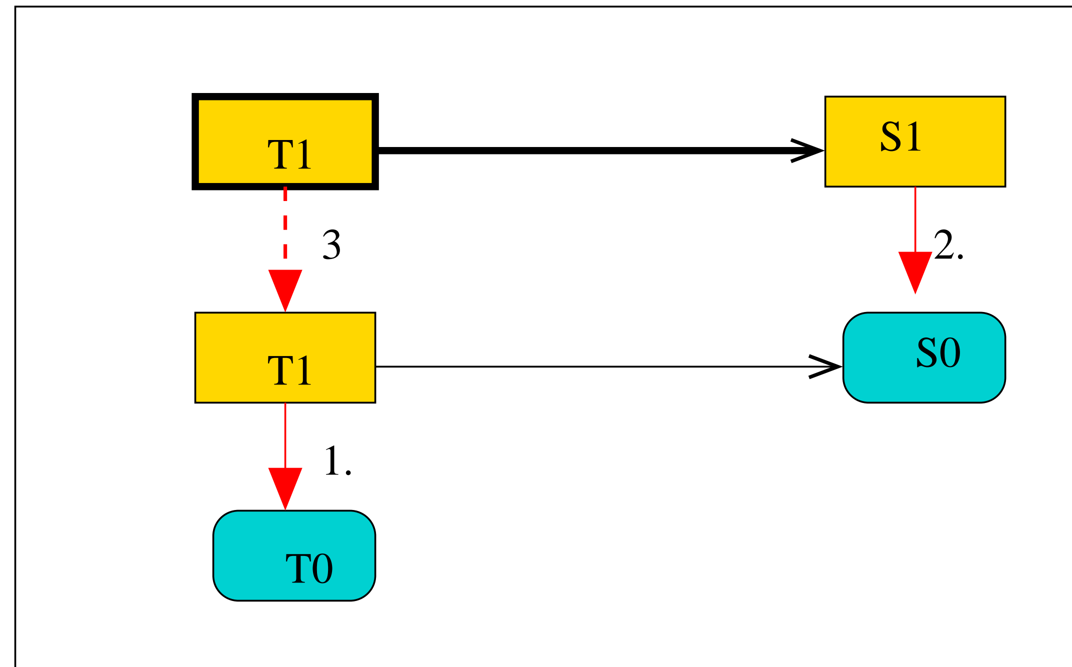
- We then provide an implementation S_1 of S_0
- We prove that this implementation satisfies the specification S_0.



CREST
CENTRE FOR RELIABLE SOFTWARE TECHNOLOGY

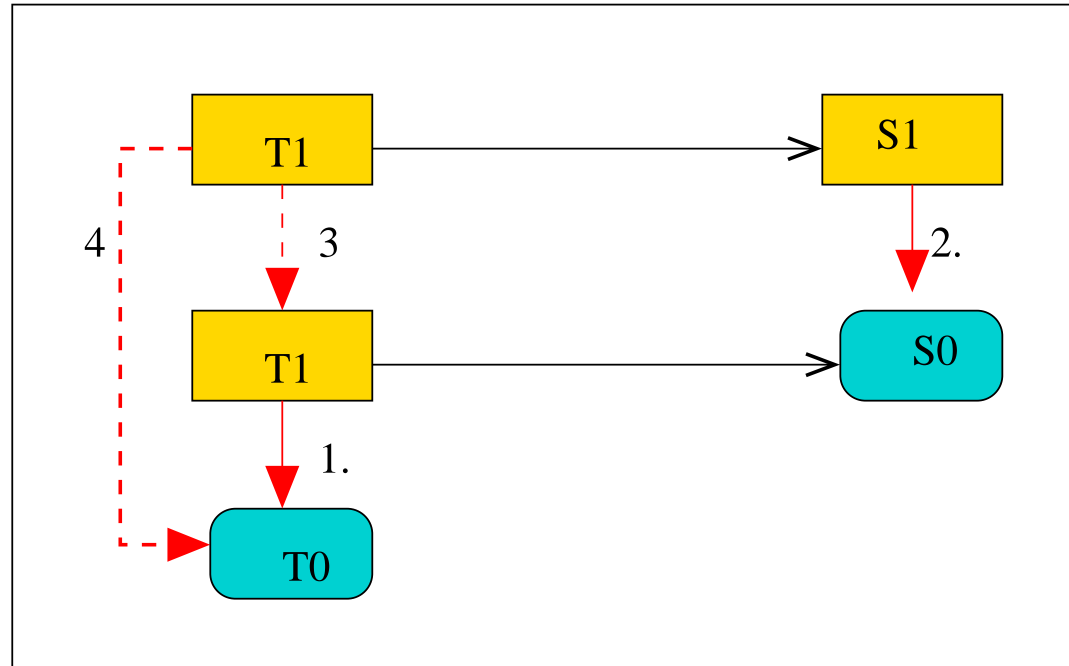
Step 3

- 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).



CREST
CENTRE FOR RELIABLE SOFTWARE TECHNOLOGY

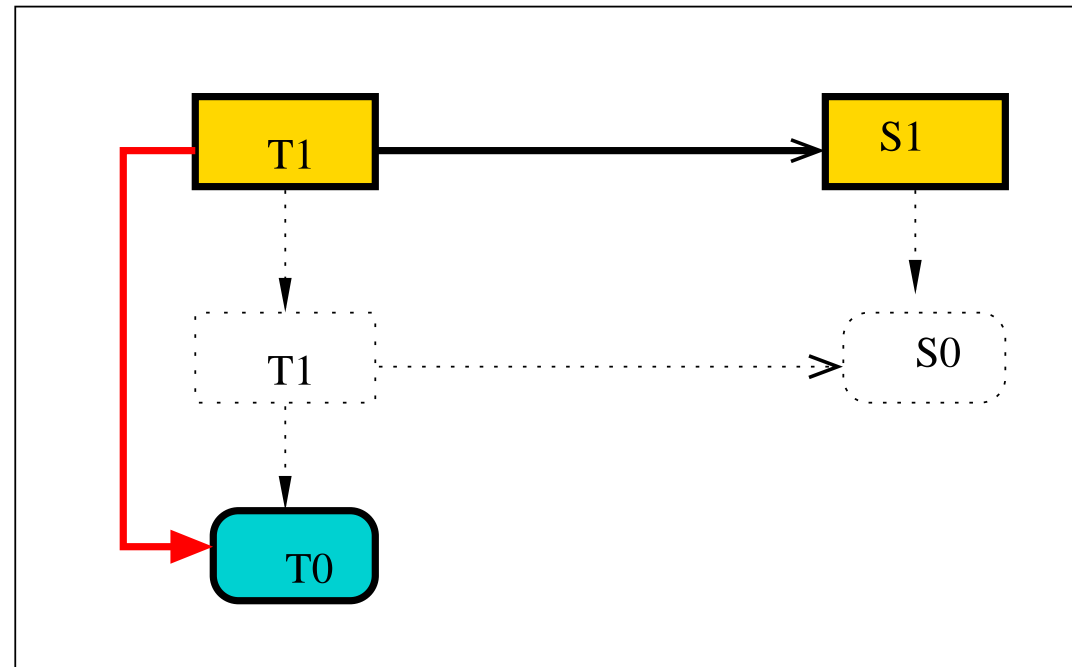
Step 4



- 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 hide them



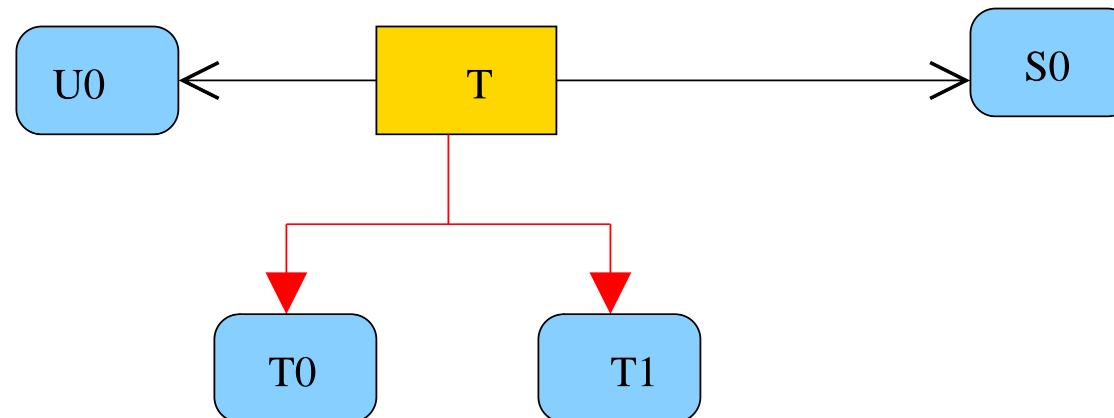
Construction as Hilbert-like proof

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

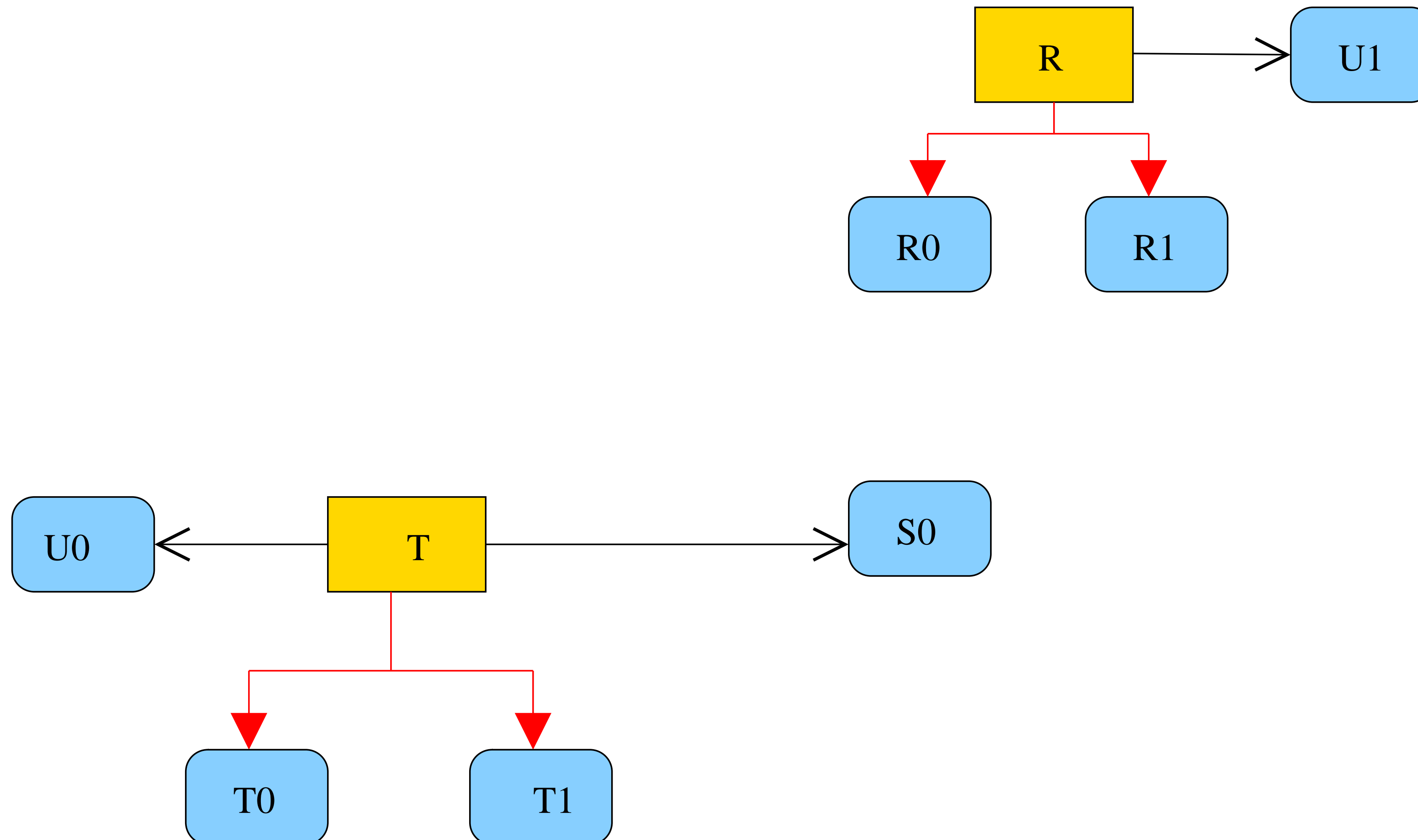
1. $T_0 \sqsubseteq T_1[S_0]$ (assumption or lemma)
2. $S_0 \sqsubseteq S_1$ (assumption or lemma)
3. $T_1[S_0] \sqsubseteq T_1[S_1]$ (by monotonicity from 2)
4. $T_0 \sqsubseteq T_1[S_1]$ (by transitivity from 1,3)

Example 2: Component reuse

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

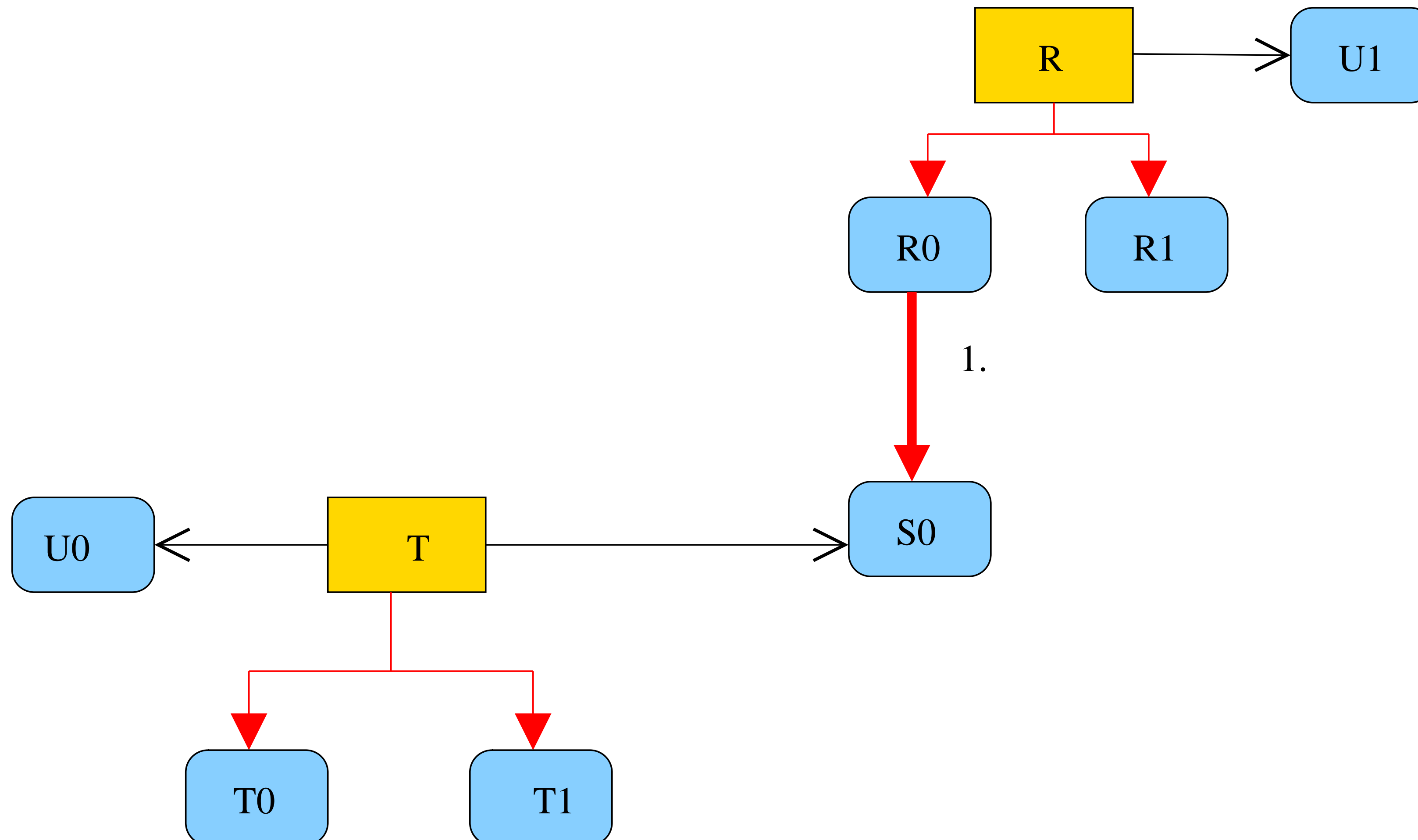


Another component R



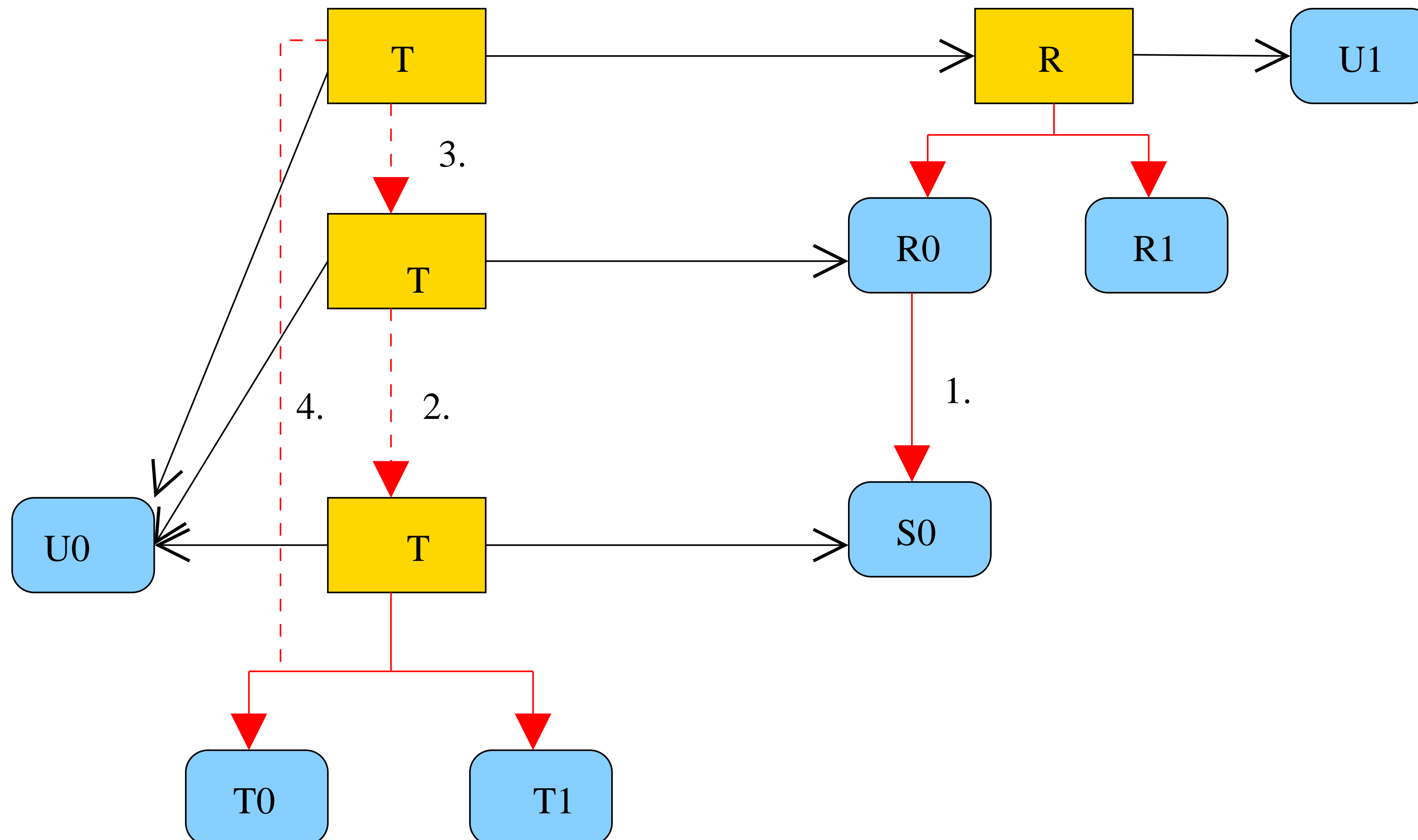
CREST
CENTRE FOR RELIABLE SOFTWARE TECHNOLOGY

Use R in T



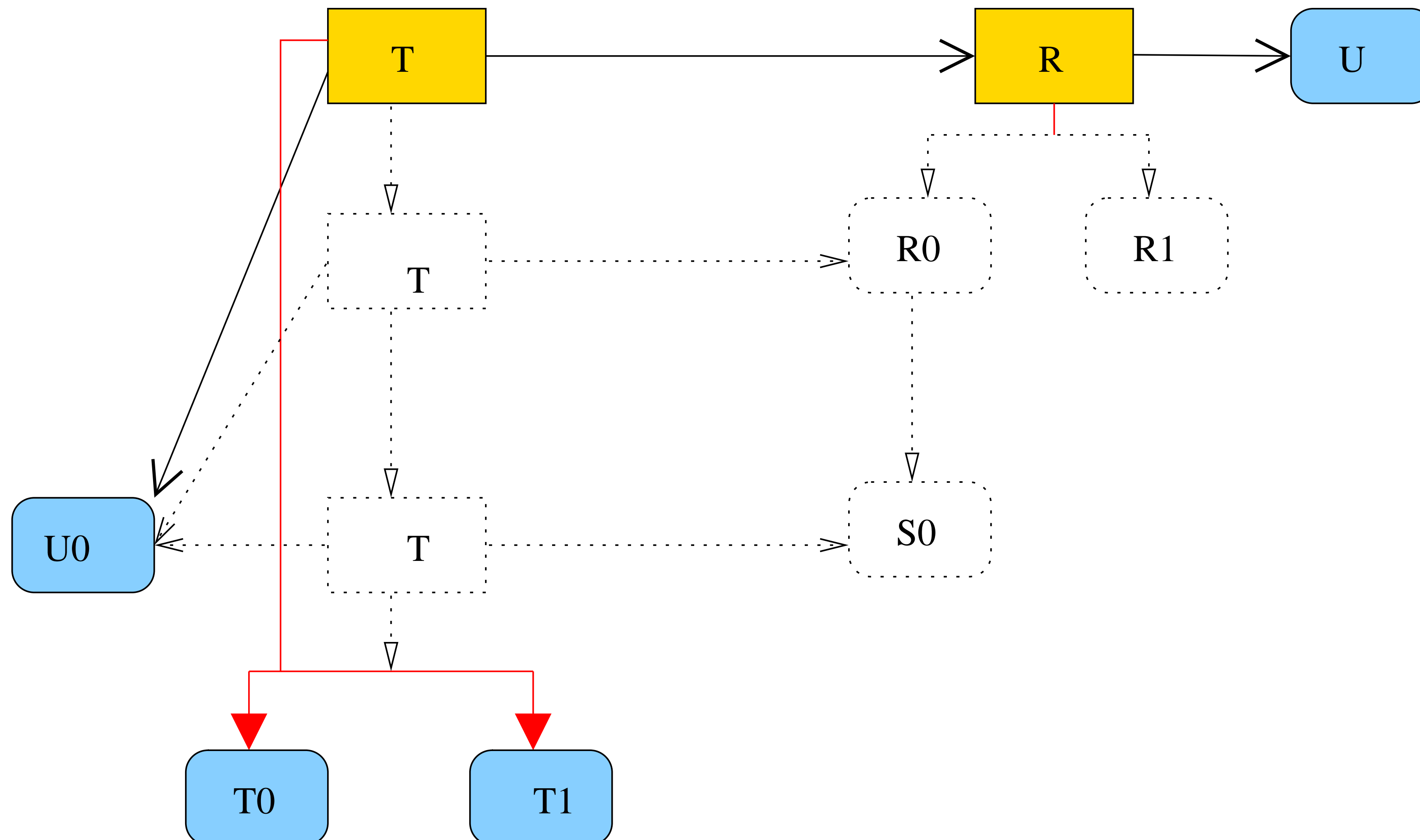
CREST
CENTRE FOR RELIABLE SOFTWARE TECHNOLOGY

Use monotonicity and transitivity



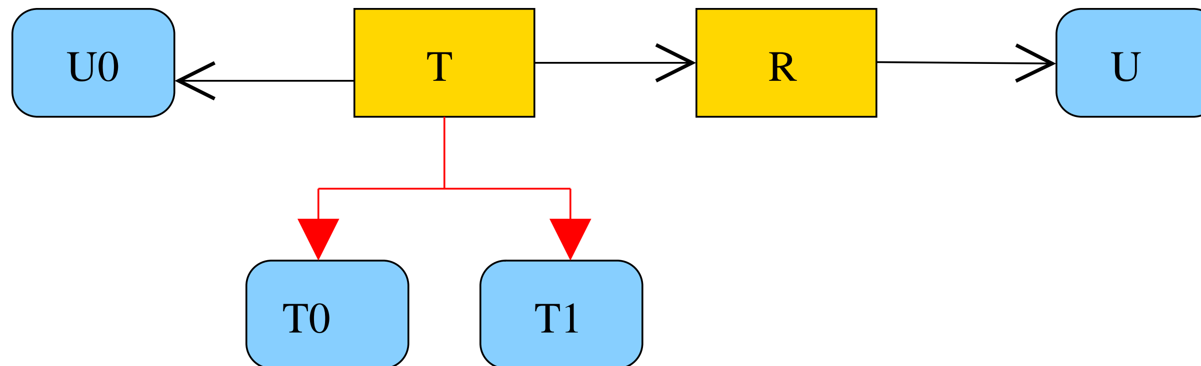
CREST
CENTRE FOR RELIABLE SOFTWARE TECHNOLOGY

Final result



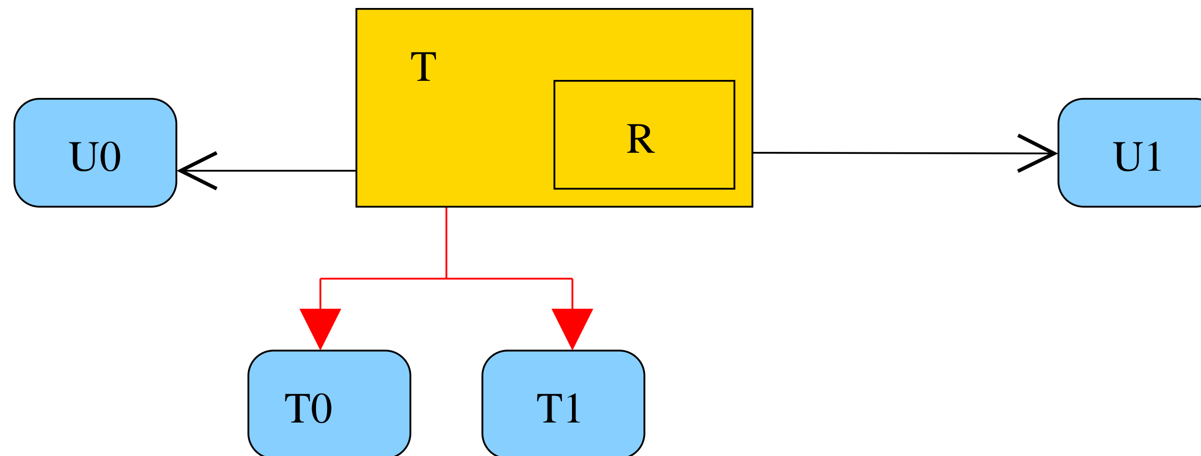
CREST
CENTRE FOR RELIABLE SOFTWARE TECHNOLOGY

Hide derivation



CREST
CENTRE FOR RELIABLE SOFTWARE TECHNOLOGY

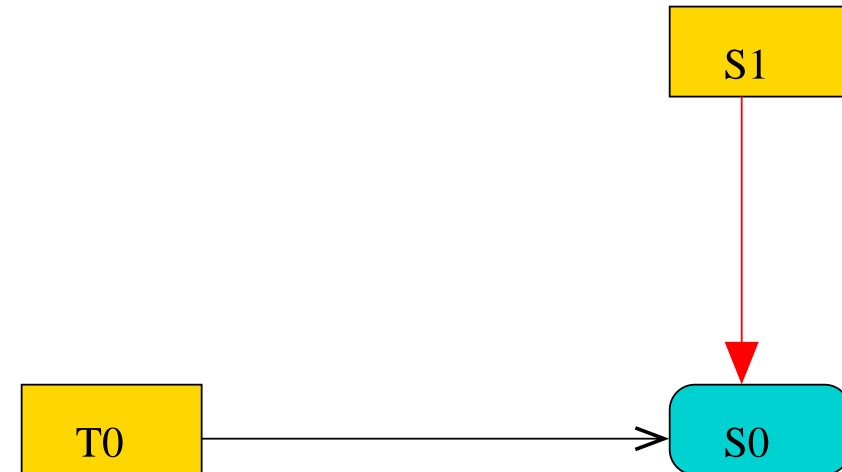
Package into bigger component



CREST
CENTRE FOR RELIABLE SOFTWARE TECHNOLOGY

Example 3: Information hiding / information revealing

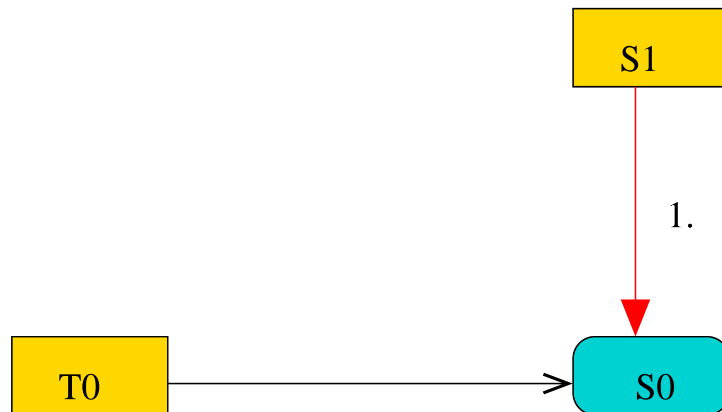
- T_0 knows (uses) the specification S_0
- T_0 does not know the implementation S_1 of S_0 (i*nformation hiding*)
- We want to refine T_0 to a new part T_1.



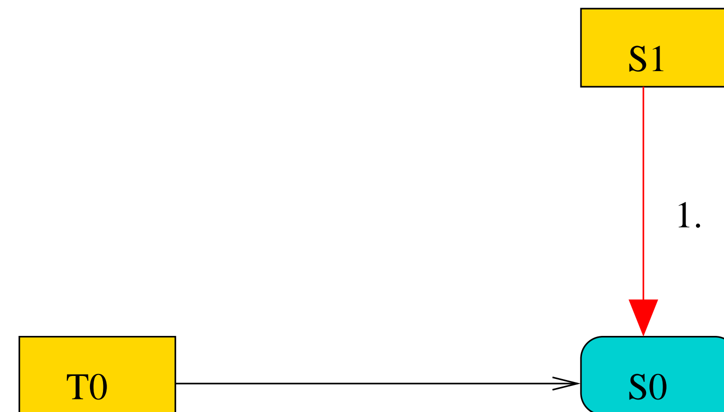
CREST
CENTRE FOR RELIABLE SOFTWARE TECHNOLOGY

Information hiding in refinement: two scenarios

Should the new part T_1 know about the implementation S_1 or not? Consider two different scenarios



Hiding information



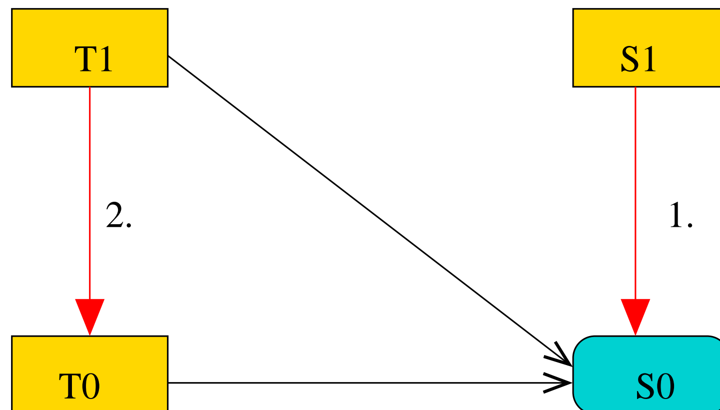
Not hiding information

CREST
CENTRE FOR RELIABLE SOFTWARE TECHNOLOGY

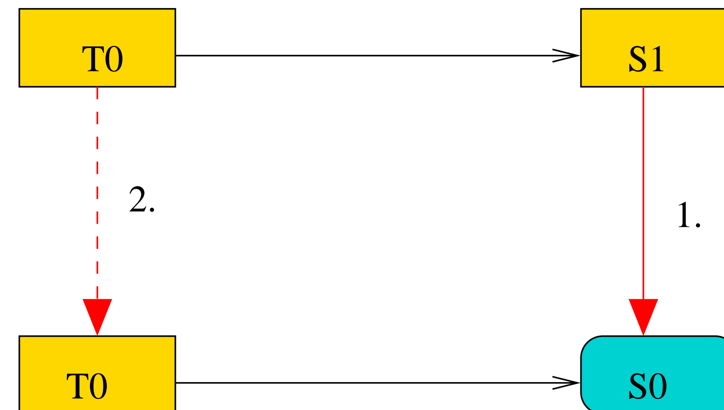
Information hiding in refinement (cont.)

Hiding: Refinement T_1 only knows specification S_0. We prove that $T_0[S_0] \sqsubseteq T_1[S_0]$

Not hiding: Allow T_0 to know the implementation S_1. By monotonicity, we have $T_0[S_0] \sqsubseteq T_0[S_1]$



Hiding information



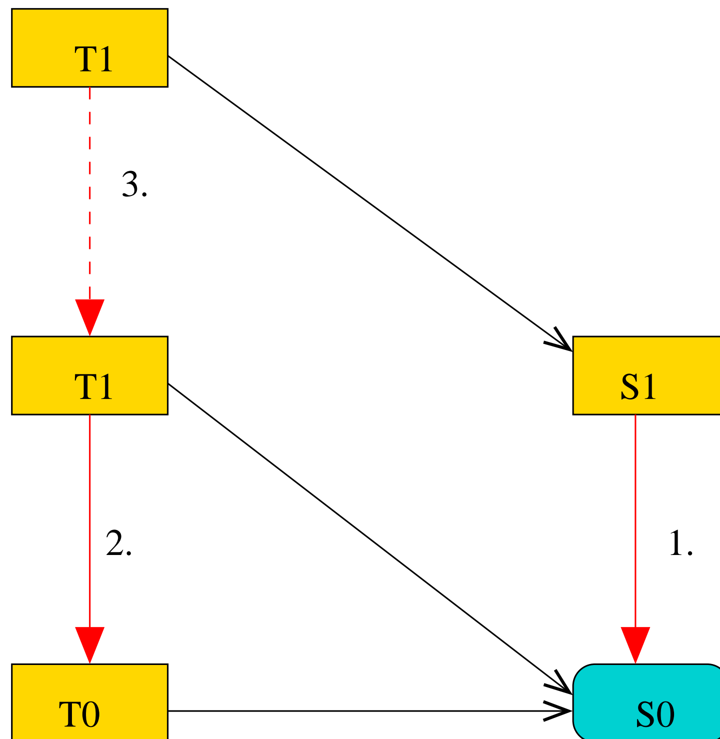
Not hiding information

CREST
CENTRE FOR RELIABLE SOFTWARE TECHNOLOGY

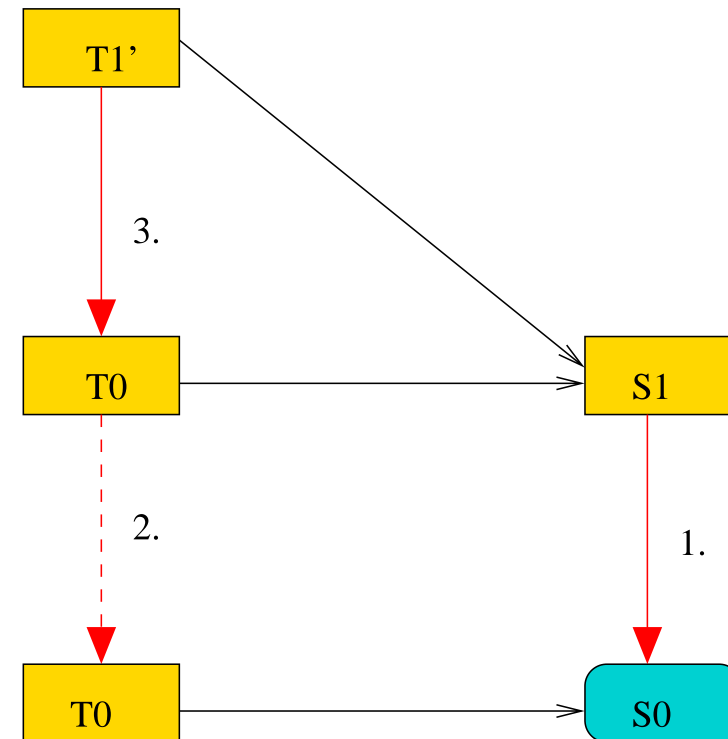
Information hiding in refinement (cont.)

Hiding: By monotonicity, we have that $T_1[S_0] \sqsubseteq T_1[S_1]$

Not hiding: We prove that $T_0[S_1] \sqsubseteq T_1'[S_1]$. Now T_1' may use information about S_1.



Hiding information

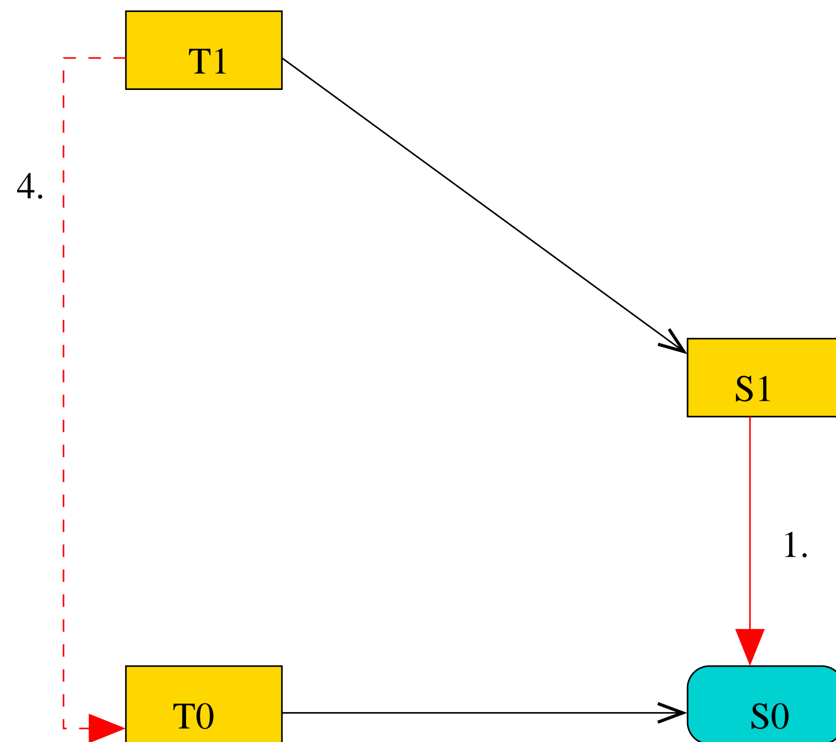


Not hiding information

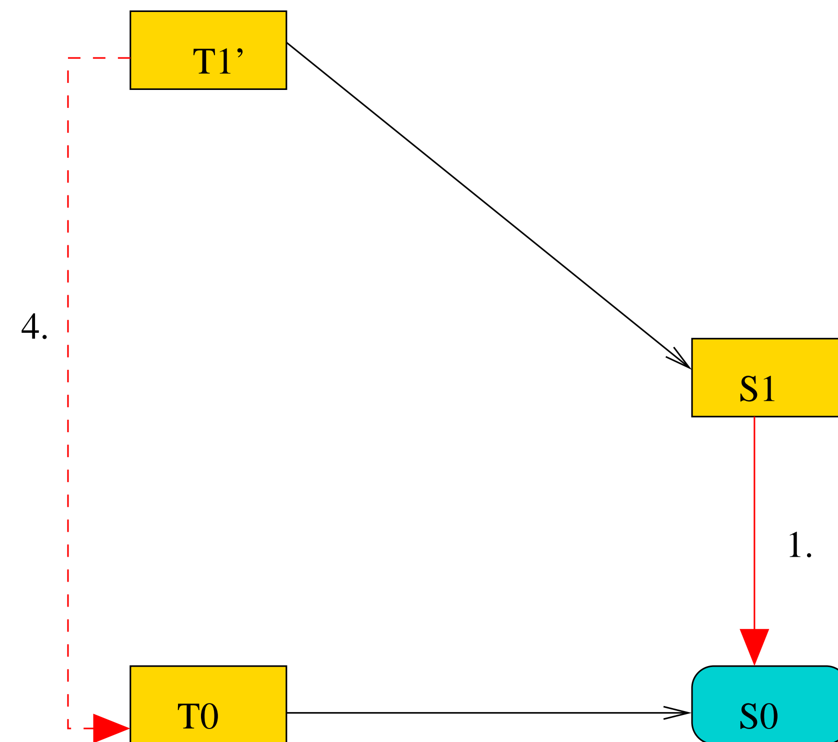
CREST
CENTRE FOR RELIABLE SOFTWARE TECHNOLOGY

Information hiding in refinement (cont.)

Hiding and not hiding gives similar result. But T_1 does not use any information about S_1 while T_1' may make use of information about S_1.



Hiding information



Not hiding information

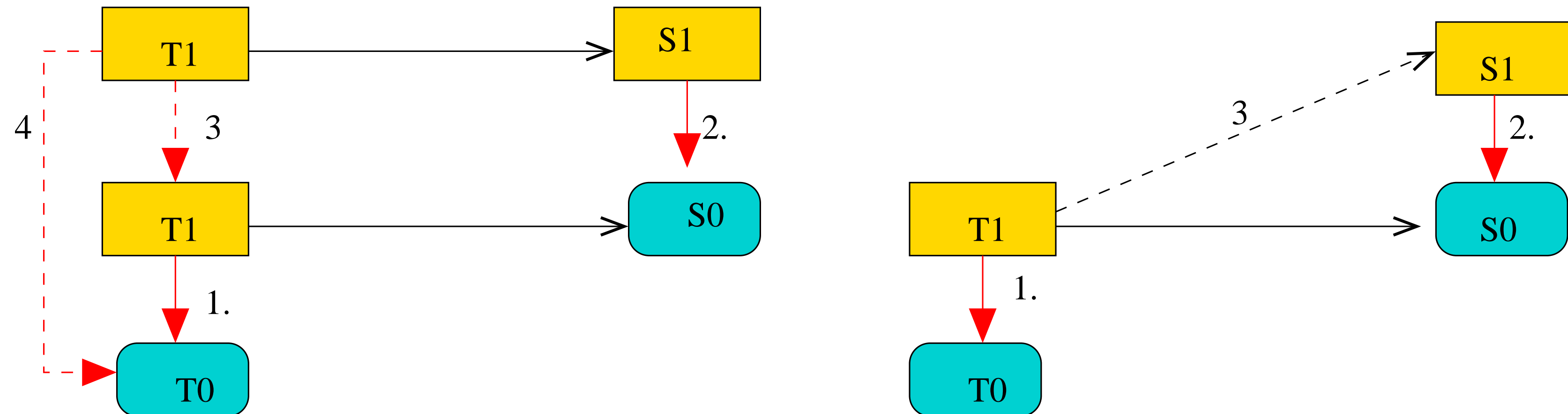
CREST
CENTRE FOR RELIABLE SOFTWARE TECHNOLOGY

Overview of Lectures

1. Incremental software construction
2. Refinement diagrams and diagrammatic reasoning
3. Reasoning about software components
4. Advantages of duplication
5. Reasoning about software extension
6. Software evolution

Duplication vs redirection

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



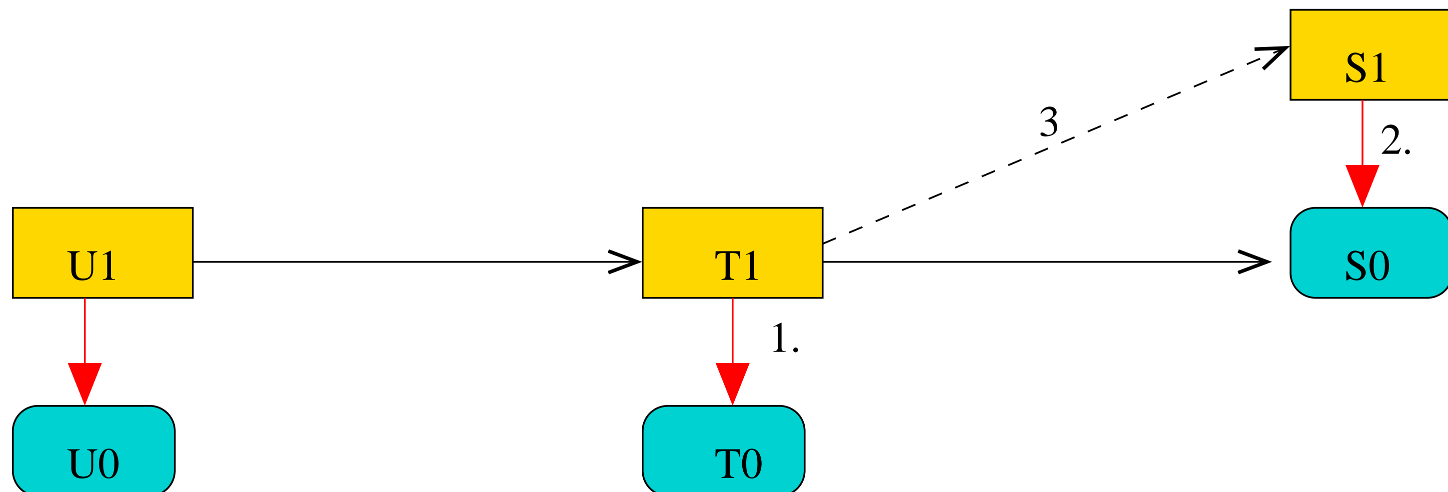
It would seem more economical to redirect the arrow in the derivation rather than duplicating the whole entity (like in right figure)

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.



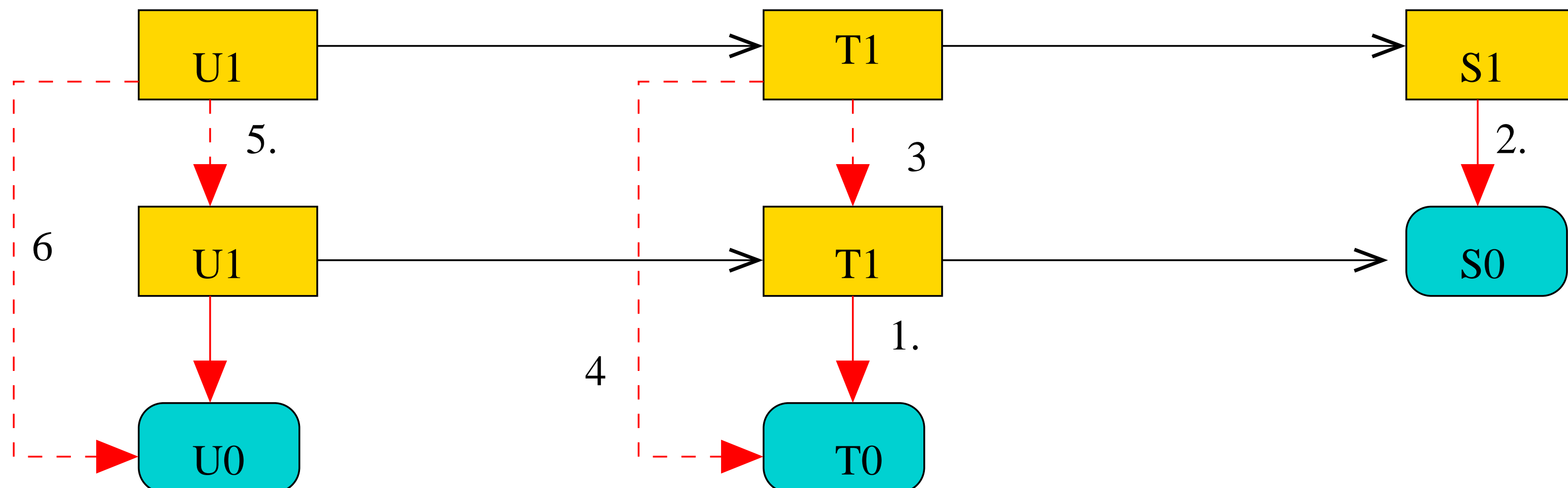
- A change in T_1 (to use S_1 rather than S_0) will also mean that U_1 is changed, from $U_1[T_1[S_0]]$ to $U_1[T_1[S_1]]$. But this change is difficult to notice here.

CREST
CENTRE FOR RELIABLE SOFTWARE TECHNOLOGY

Duplication is good

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

- Compare above to the same derivation with duplication:

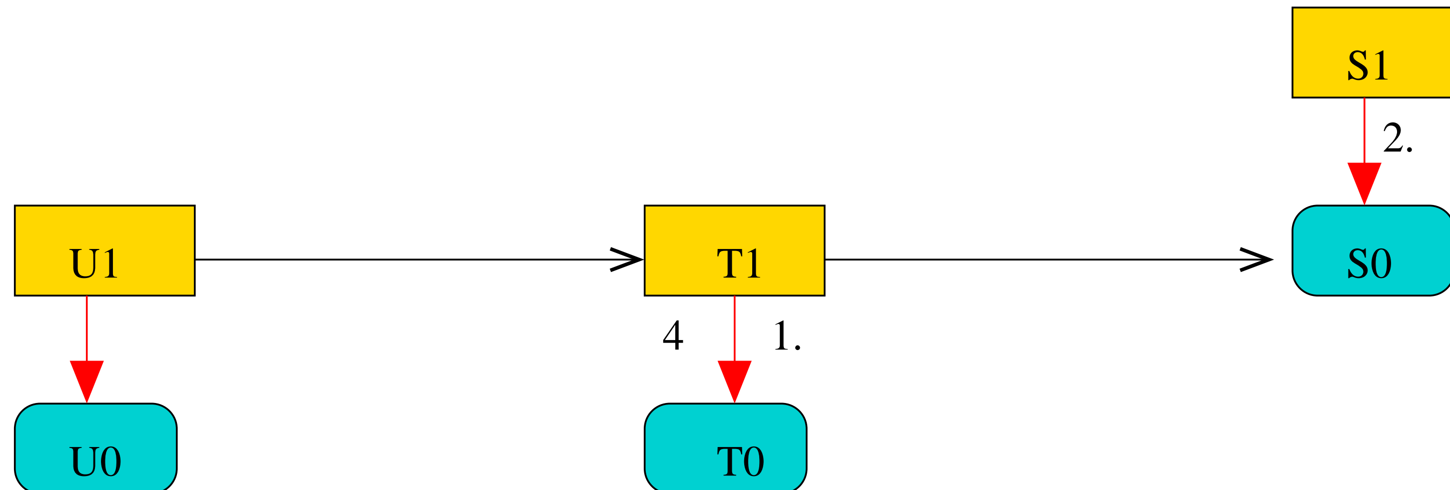


- New terms are shown explicitly, $U_1[T_1[S_0]]$ and $U_1[T_1[S_1]]$ both occur in the diagram.

CREST
CENTRE FOR RELIABLE SOFTWARE TECHNOLOGY

Compacting refinement diagrams

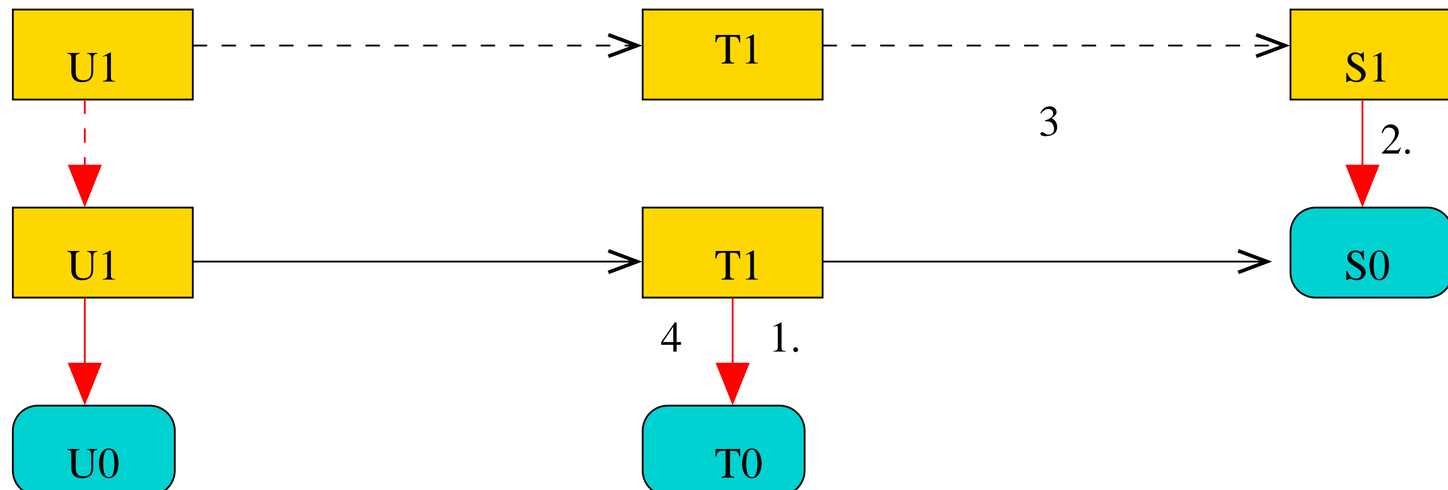
Avoid making inferences unless they are explicitly needed. Example:



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.

Overview of Lectures

1. Incremental software construction
2. Refinement diagrams and diagrammatic reasoning
3. Reasoning about software components
4. Advantages of duplication
5. Reasoning about software extension
6. Software evolution

Software increments

Increment an existing system by either

- adding a new *component*, (described above), or
- adding a new *extension*, increasing the functionality of an existing component (described below).
- components and extensions are both parts in the system

CREST
CENTRE FOR RELIABLE SOFTWARE TECHNOLOGY

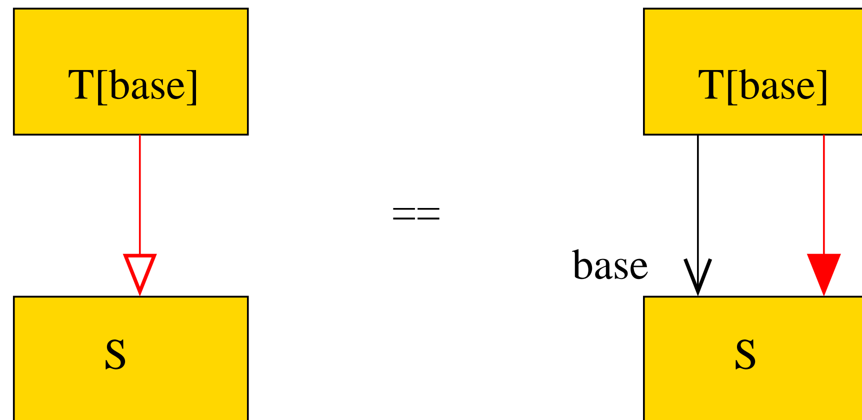
Extension

- Let S be some part, and let $T[base]$ be another part that extends S. The parameter $base$ indicates the use of S in T.
- We write $S \triangleleft T[base]$ for the component that we get by *extending* component S by component $T[base]$, i.e. $S \triangleleft T[base] = T[S]$
- We will require from an extension that $S \sqsubseteq T[S]$ holds, i.e., the extension should preserve the functionality of the original component (*superposition refinement*).
- Example: an extension class may add new attribute and methods, but behavior of old methods on old attributes must remain the same.

CREST
CENTRE FOR RELIABLE SOFTWARE TECHNOLOGY

Extension in refinement diagrams

We introduce a special arrow for superposition refinement.



Note that S and T may dependend on other parts in the environment.

Example 4: 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.

Example application:Teenage girl diary

Teenage Girl Diary

Image

Week 12	Mon 23	Tue 24	Wed 25	Thu 26	Fri 27	Sat 28
8 - 9						
9 - 10	Biology				Cooking	
10 - 11	Biology				Cooking	
11 - 12	English	Math	Geography			
12 - 13	English	Math	Geography	Swedish		Sun 29
13 - 14		Gymnastics		Swedish		
14 - 15		Gymnastics				
15 - 16						
		Riding			Jen's party	

Next week | Last week | Clear week | View term

Teenage Girl Diary

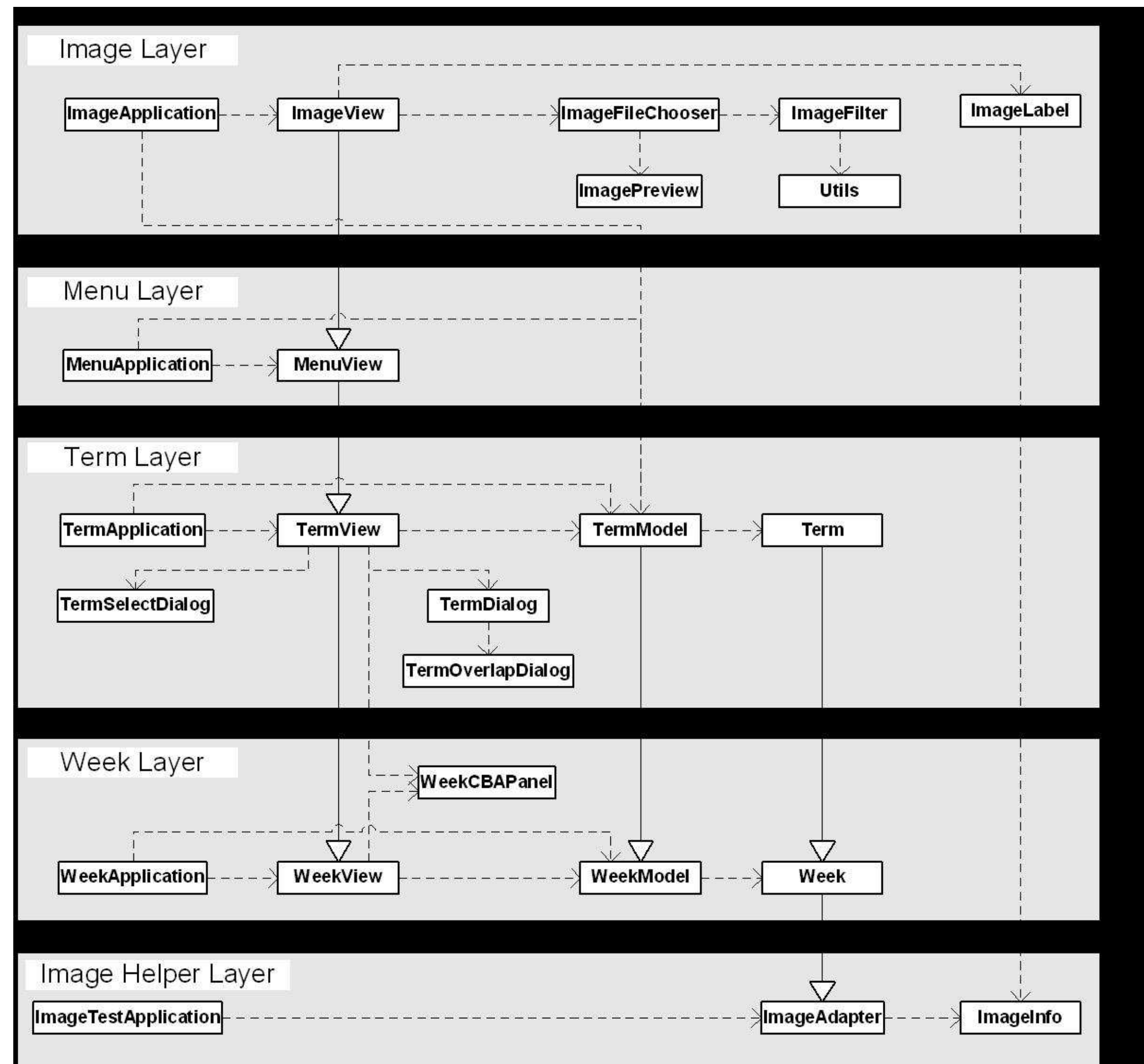
Week 2	Mon 12	Tue 13	Wed 14	Thu 15	Fri 16	
8 - 9						Sat 17
9 - 10	Biology				Cooking	Party.
10 - 11	Biology				Cooking	
11 - 12	English	Mathematics	Geographics			
12 - 13	English	Mathematics	Geographics			
13 - 14		Gymnastics		Swedish		Sun 18
14 - 15		Gymnastics		Swedish		More party.
15 - 16						
		Riding.			Cider drinking. Disco.	

Next week | Last week | Clear week | View term

tk

Week 49	Monday	Tuesday	Wednesday	Thursday	Friday	
8 - 9	Math	English				Saturday
9 - 10	Math	Finnish	Drawing	Math	E.S.	Katarina's birthday. Skiing 20 km.
10 - 11	History	Finnish	Drawing	Physics	E.S.	
11 - 12	Gym	Biology	R.E.	Physics	Biology	
12 - 13		English	R.E.	Geography	Swedish	
13 - 14						Sunday
14 - 15						
15 - 16						
	Skiing, 15 km.	Skiing 10 km.	Running 10 km.	Rest.	Buy six-pack. Warm up sauna.	

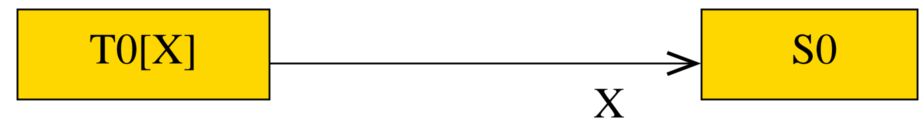
Layered structure



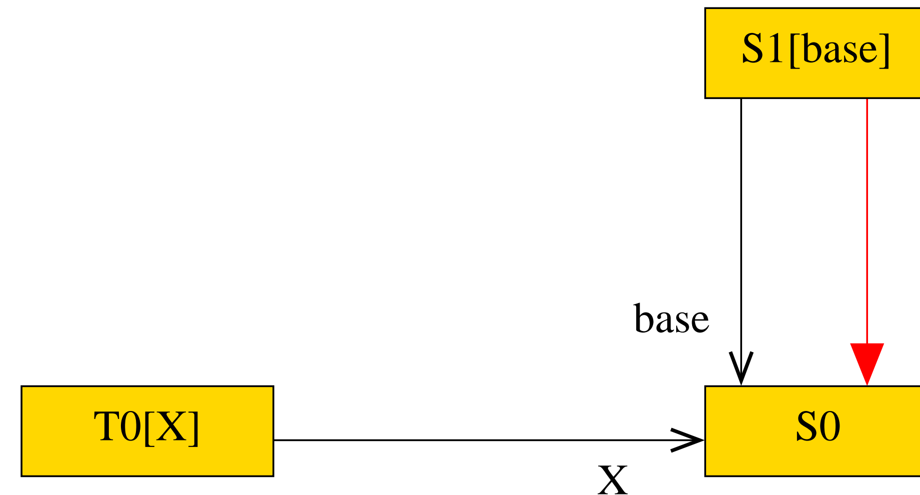
CREST CENTRE FOR RELIABLE SOFTWARE TECHNOLOGY

Initial layer

- Start with a system consisting of $T_0[X]$ and S_0
- X is bound to S_0.



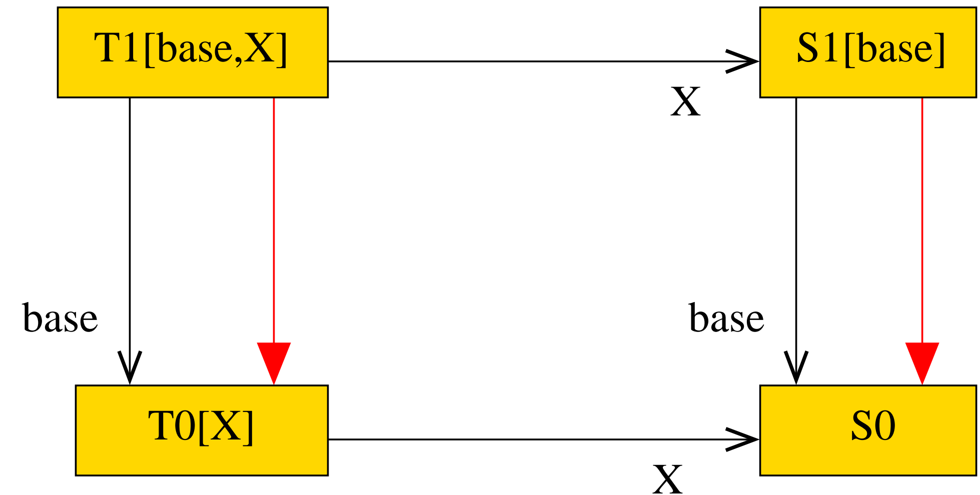
Extend used component



- Introduce a part S_1 that extends S_0

CREST
CENTRE FOR RELIABLE SOFTWARE TECHNOLOGY

Extend using component



- Then introduce a new part T_1 that uses the extension S_1 and extends T_0

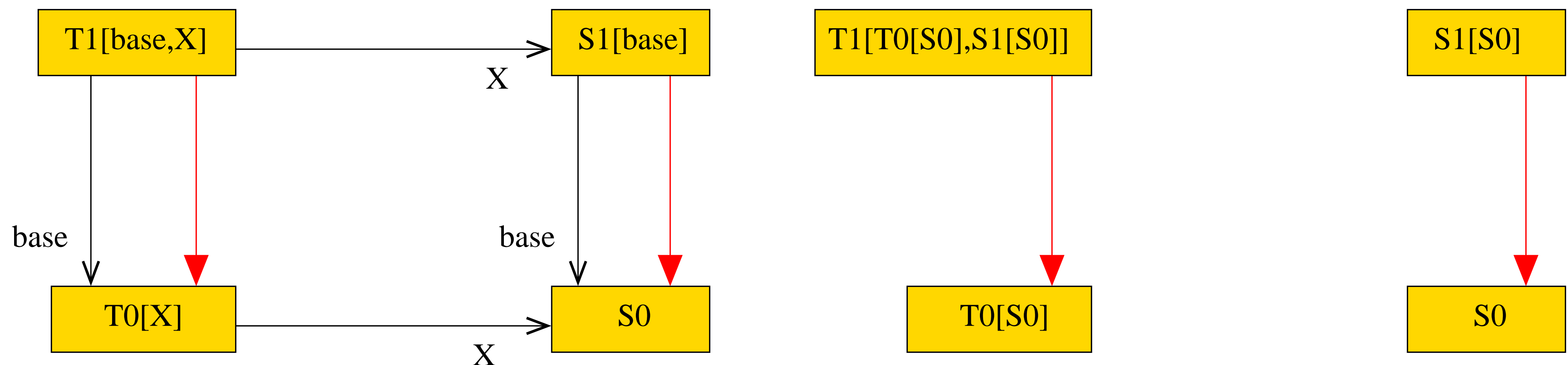
CREST
CENTRE FOR RELIABLE SOFTWARE TECHNOLOGY

Static and layered binding

- The meaning of the parts are determined here by carrying out the substitutions according to the bindings
- We get *static binding* if we bind X and $base$ at the same time (extension and usage bound at the same time).
- We get *layered binding* (dynamic binding?) if we first bind $base$ and then bind X (extension bound before usage).

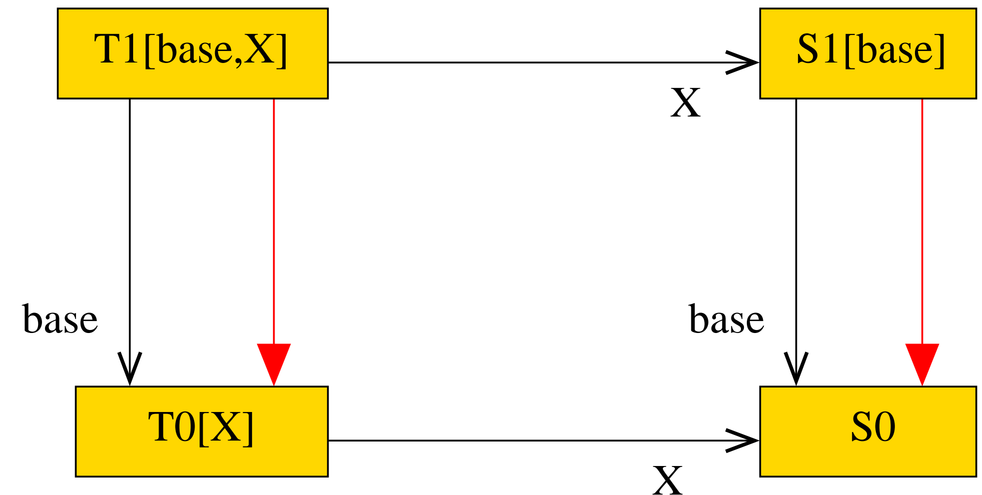
Static binding

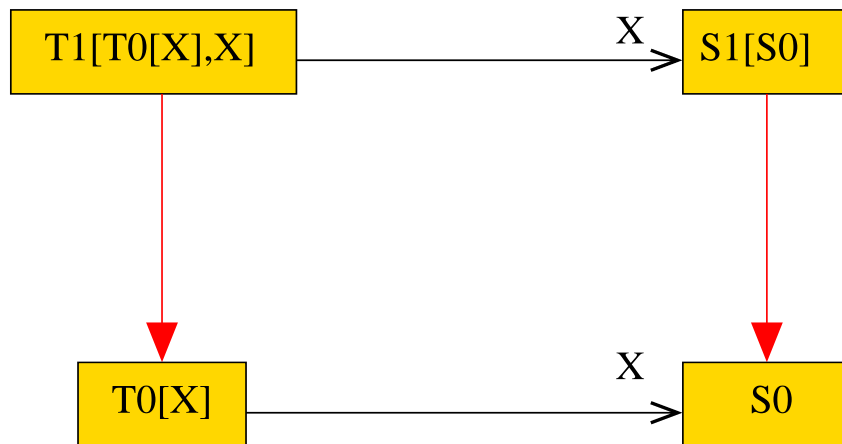
- $T_0[X][X := S_0] = T_0[S_0]$
- $S_1[base][base := S_0] = S_1[S_0]$
- $T_1[base, X][base := T_0[S_0], X := S_1[S_0]] = T_1[T_0[S_0], S_1[S_0]]$

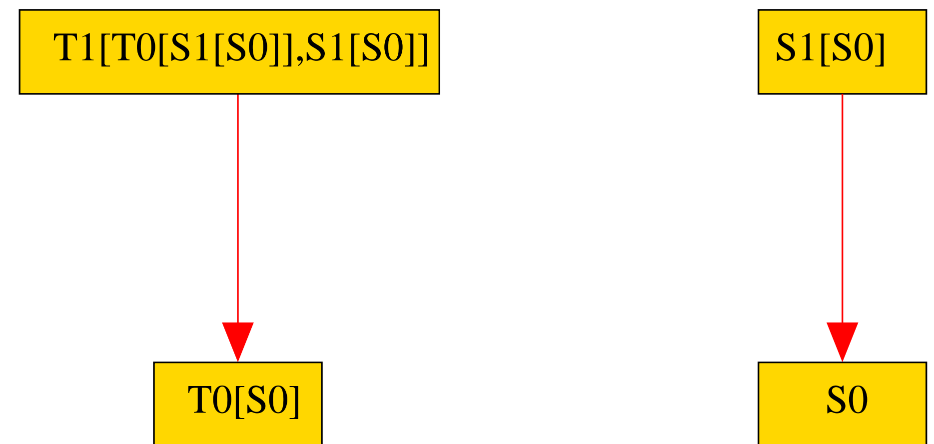


Layered binding

- $S_1[base][base := S_0] = S_1[S_0]$
- $T_1[base, X][base := T_0[X]] = T_1[T_0[X], X]$



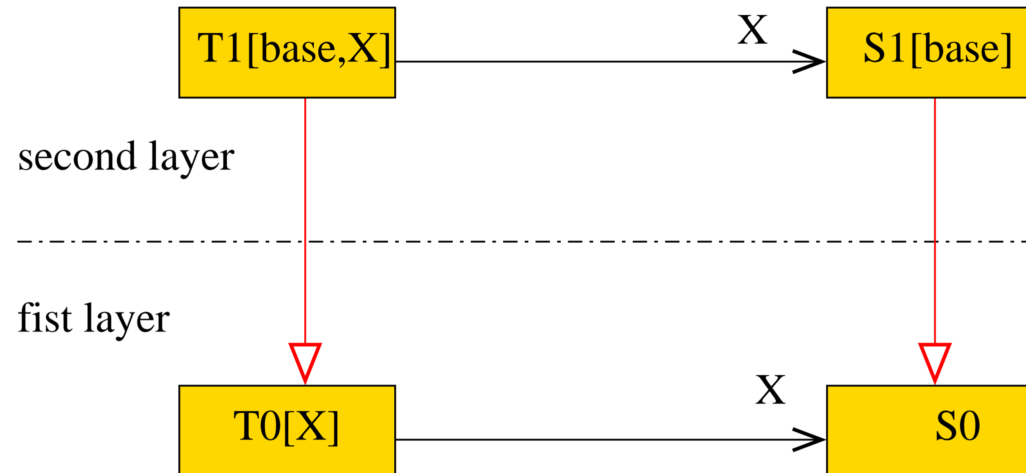




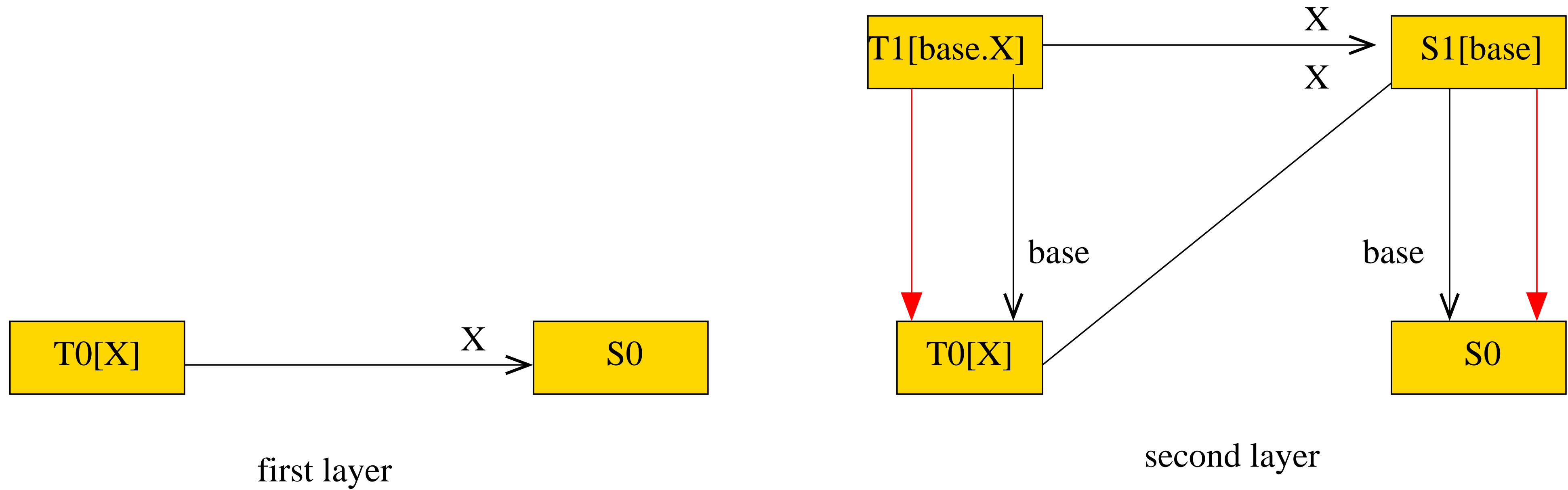
Layers

- Layered binding gives here a layered structure, where a *layer* is a collection of extensions that are to be used together.
- A reference to a part at a lower level of extension is taken to refer the extension in the current layer (i.e., all calls are bound to extensions in the current layer).
- We will use the extension symbol (white arrowhead) when we want to have layered binding
- We indicate a layer with a dashed outline in the diagram.

Two layers



Two systems



Overview of Lectures

1. Incremental software construction
2. Refinement diagrams and diagrammatic reasoning
3. Reasoning about software components
4. Advantages of duplication
5. Reasoning about software extension
6. Software evolution

Software evolution

- A refinement diagram proof models the evolution of software over time
- Each new addition to the diagram increases the (logical) time counter by one.
- 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
- The construction of the software system can be played back like a movie, showing how each step adds to the construction.

No deletions in refinement diagrams

- We are only permitted to add elements to the diagram; we do not permit any elements to be removed from a refinement diagram.

- Removing elements may make the corresponding Hilbert proof inconsistent

- 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

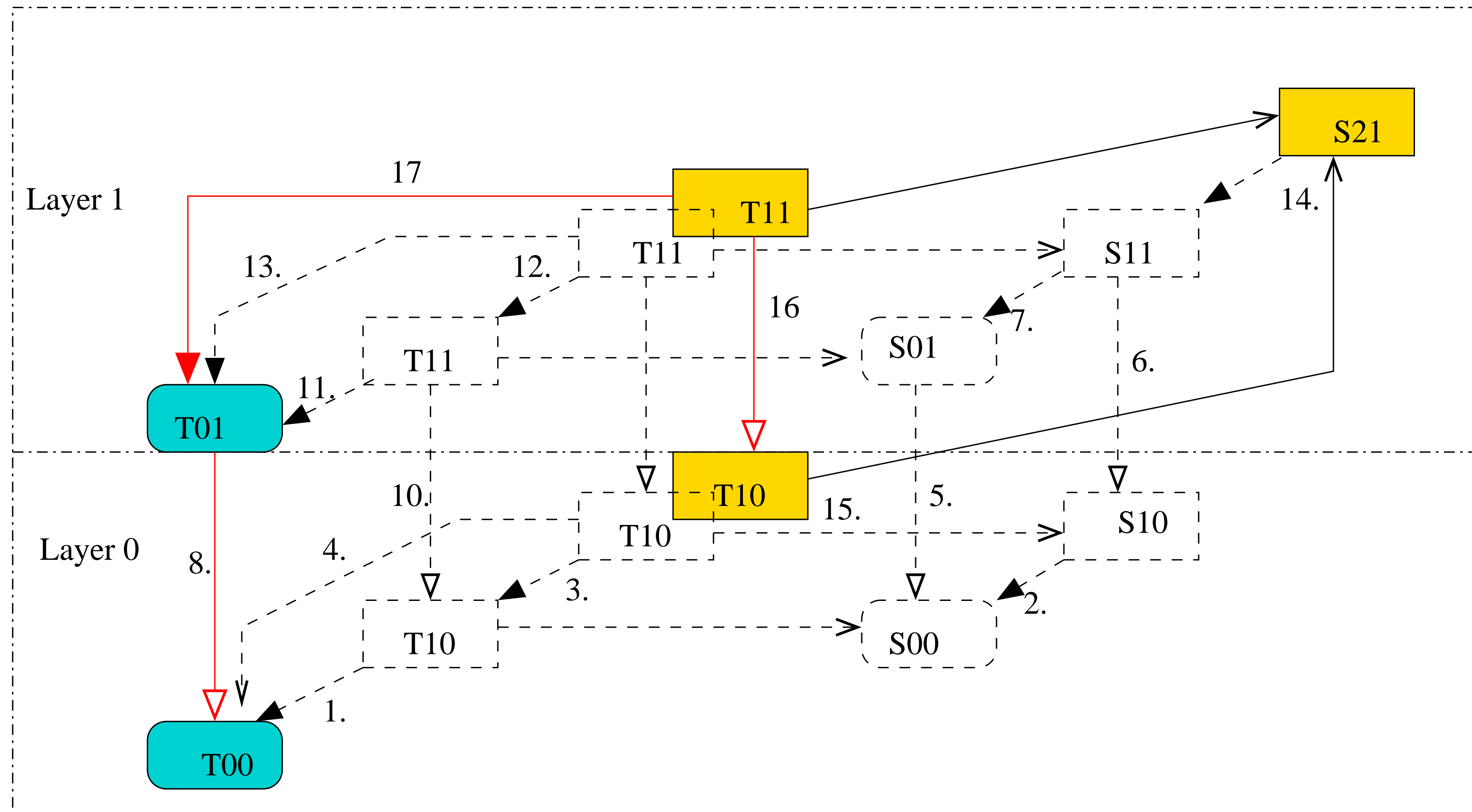
Keeping (but hiding) the history

- The diagram shows the historic development
- The parts which are not relevant for our present purposes may be hidden, but not removed– they may be needed later.
- A step in the derivation that we ignore may have to be revisited later,
  - 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 the 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.
- Redesign is also done by adding new elements to the diagram.
- The now obsolete elements (describing the earlier design) are not removed. They remain in the diagram, but are on paths that will be ignored in later construction phases.

Example redesign



Refinement diagram editor

- Many of the operations described above become rather cumbersome if done by hand
- In particular, need support for selective showing and hiding of sections of the refinement diagram.
- A refinement diagram editor provides an environment for building and manipulating refinement diagrams
- Can also work as a code base, proof environment, testing environment, documentation environment, version control system
- Presently working on a 3-d refinement diagram editor, Socos (Software Construction Site).

Thank you!