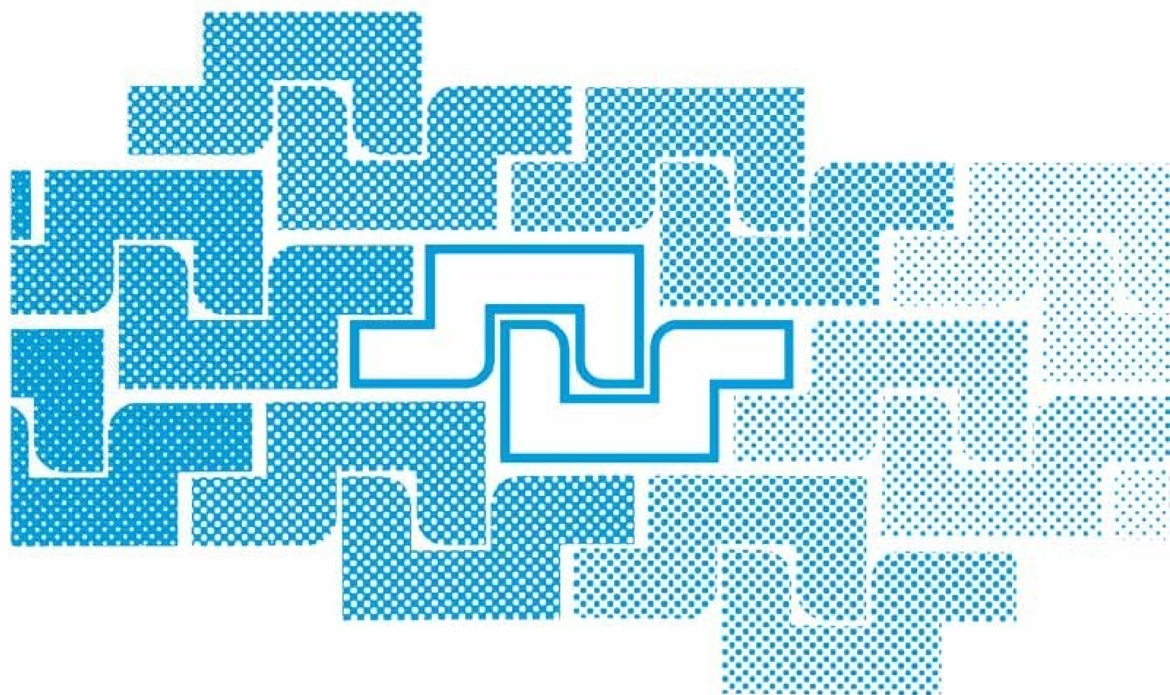


Engineering Theories of Software Intensive Systems

Edited by

Manfred Broy, Johannes Grünbauer,
David Harel and Tony Hoare

NATO Science Series

II. Mathematics, Physics and Chemistry – Vol. 195

Engineering Theories
of Software Intensive Systems

NATO Science Series

A Series presenting the results of scientific meetings supported under the NATO Science Programme.

The Series is published by IOS Press, Amsterdam, and Springer (formerly Kluwer Academic Publishers) in conjunction with the NATO Public Diplomacy Division.

Sub-Series

I. Life and Behavioural Sciences	IOS Press
II. Mathematics, Physics and Chemistry	Springer (formerly Kluwer Academic Publishers)
III. Computer and Systems Science	IOS Press
IV. Earth and Environmental Sciences	Springer (formerly Kluwer Academic Publishers)

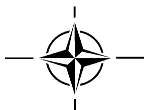
The NATO Science Series continues the series of books published formerly as the NATO ASI Series.

The NATO Science Programme offers support for collaboration in civil science between scientists of countries of the Euro-Atlantic Partnership Council. The types of scientific meeting generally supported are “Advanced Study Institutes” and “Advanced Research Workshops”, and the NATO Science Series collects together the results of these meetings. The meetings are co-organized by scientists from NATO countries and scientists from NATO’s Partner countries — countries of the CIS and Central and Eastern Europe.

Advanced Study Institutes are high-level tutorial courses offering in-depth study of latest advances in a field.
Advanced Research Workshops are expert meetings aimed at critical assessment of a field, and identification of directions for future action.

As a consequence of the restructuring of the NATO Science Programme in 1999, the NATO Science Series was re-organized to the four sub-series noted above. Please consult the following web sites for information on previous volumes published in the Series.

http://www.nato.int/science
http://www.springeronline.com
http://www.iospress.nl



Series II: Mathematics, Physics and Chemistry – Vol. 195

Engineering Theories of Software Intensive Systems

edited by

Manfred Broy
Technische Universität München,
Garching, Germany

Johannes Grünbauer
Technische Universität München,
Garching, Germany

David Harel
The Weizmann Institute of Science,
Rehovot, Israel

and

Tony Hoare
Microsoft Research,
Cambridge, U.K.

Springer

Proceedings of the NATO Advanced Study Institute on
Engineering Theories of Software Intensive Systems
Marktoberdorf, Germany
3–15 August 2004

A C.I.P. Catalogue record for this book is available from the Library of Congress.

ISBN-10 1-4020-3531-4 (PB)
ISBN-13 978-1-4020-3531-9 (PB)
ISBN-10 1-4020-3530-6 (HB)
ISBN-13 978-1-4020-3530-2 (HB)
ISBN-10 1-4020-3532-2 (e-book)
ISBN-13 978-1-4020-3532-6 (e-book)

Published by Springer,
P.O. Box 17, 3300 AA Dordrecht, The Netherlands.

www.springeronline.com

Printed on acid-free paper

All Rights Reserved
© 2005 Springer
No part of this work may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, microfilming, recording or otherwise, without written permission from the Publisher, with the exception of any material supplied specifically for the purpose of being entered and executed on a computer system, for exclusive use by the purchaser of the work.

Printed in the Netherlands.

Contents

Preface vii

Part I Architectures, Design and Interfaces

Incremental Software Construction with Refinement Diagrams 3
Ralph-Johan Back

Service-oriented Systems Engineering: Specification and Design of Services and Layered Architectures 47
Manfred Broy

Interface-based Design 83
Luca de Alfaro, Thomas A. Henzinger

The Dependent Delegate Dilemma 105
Betrand Meyer

Part II System and Program Verification, Model Checking and Theorem Proving

Formalizing Counterexample-driven Refinement with Weakest Preconditions 121
Thomas Ball

A Mechanically Checked Proof of a Comparator Sort Algorithm 141
J. Strother Moore, Bishop Brock

Keys in Formal Verification 177
Amir Pnueli

On the utility of canonical abstraction 215
Mooly Sagiv, Thomas W. Reps, Reinhard Wilhelm, Eran Yahav

Part III Process Algebras and Experimental Calculi

Process Algebra: a Unifying Approach 257
Tony Hoare

Computation Orchestration 285
Jayadev Misra

A Tree Semantics of an Orchestration Language 331
Tony Hoare, Galen Menzel, Jayadev Misra

Part IV Security, System Development and Special Aspects

Model Driven Security 353
David Basin, Jürgen Doser, Torsten Lodderstedt

Some Challenges for System Development: Reactive Animation, Smart Play-Out and Olfaction 399
David Harel

Preface

Today software systems are a pervasive factor in industry, science, commerce, and communication. In consequence of the wide distribution of software, the high dependency of its functioning and quality, our understanding of software engineering foundations is still weak. A lot of progress has been achieved, but much more has to be done to keep pace with the speed of innovations and new applications. The foundations of software technology lie in models allowing us to capture application domains, requirements, but also to understand the structure and working of software systems like software architectures and programs. These models have to be expressed in techniques of discrete mathematics including logics and algebra. However, according to the very specific needs in applications of software technology, formal methods have to serve the needs and the quality of advanced software engineering methods, especially taking into account security aspects in Information Technology. The lectures of the Marktoberdorf Summerschool address these topics and teach state-of-the-art ideas how to meet these challenges.

This book is divided into four parts:

Part I: Architectures, Design and Interfaces. Constructing large software systems requires solutions to the central problem of managing their complexity. The larger a systems becomes, the more difficult it is to extend the system and to adapt it to changing requirements. *Ralph-Johan Back* presents in his contribution the handling of these processes as a stepwise feature introduction. Interfaces play a central role in the design of hard- and software systems. Two further articles take this into account: First, based on the FOCUS theory, *Manfred Broy* introduces a formal model to specifying and designing large systems using services and layered architectures. Second, *Thomas A. Henzinger* presents his work on interface-based design, where interface theories are axiomated to cover certain requirements. Finally, *Bertrand Meyer* presents a promising solution to the "Dependent Delegate Dilemma", based on a simple correctness rule.

Part II: System and Program Verification, Model Checking and Theorem Proving. Building reliable soft- and hardware is still a challenge today. Even so, there are some areas where strong achievements to systems are made, e.g. in air planes and air traffic control, most soft- and hardware is still error-prone. In this part, approaches to develop correct systems are shown. *Thomas Ball* presents a method how to model-check temporal safety properties of programs. *J Strother Moore* shows how to prove "little theorems" without addressing a complete system, *Amir Pnueli* presents a set of techniques for the verification of reactive infinite-states systems. And finally, *Shmuel Shagiv* presents a survey on a parametric abstract domain called *canonical abstraction*.

Part III: Process Algebras and Experimental Calculi. Mathematical foundations of software engineering help to describe and understand the processes how software should behave. In this part, some experimental work is shown. The article of *Tony Hoare* presents a unifying theory of concurrency that combines the advantages of process algebras. *Jayadev Misra* shows in his contribution a novel approach for combining different web-services as well as general distributed transactions to form complex services over the internet. In a joint paper, Tony Hoare, Galen Menzel and Jayadev Misra present a formal semantics of Jayadev Misra's language "Orc".

Part IV: Security, System Development and Special Aspects. Security is getting a leading role in software engineering. *David Basin* shows how to specify high-level system models along with their security properties and use tools to generate system architectures from the models automatically, including complete, configured security infrastructures. The article of *David Harel* revolves topics that are seemingly peripherical to the *classical* notion of system development. Reactive animation covers a method which allows enriching models of reactive systems with an animated, interactive and intuitive frontend. A method, called "smart play-out", helps to run a program by utilizing verification techniques and tools. With olfaction a set-up for an odor communication and synthesis system is proposed.

The contributions in this volume emerged from lectures of the 25th International Summer School on *Engineering Theories of Software Intensive Systems*, held at Marktoberdorf from August 3 to August 15, 2004. More than 100 participants from 25 countries attended, including students, lecturers and staff. The summerschool has been two weeks of learning, discussing and developing new ideas, and a gainful event, from the professional as well from the social aspect.

It is our pleasure to thank all lecturers, staff, hosts in Marktoberdorf, and especially our secretaries Ingrid Luhn and Katharina Spies for their great and gentle support.

Furthermore, we thank Wil Bruins from Kluwer Publishing and Katharina Spies for their help in editing this volume. We also thank Britta Liebscher, who did a great job in typesetting some articles in LaTeX.

We thank the photographers Katharina Spies, Sonja Werner, David Harel and Dan Barak for always being in the right place at the right time.

The Marktoberdorf Summer School was arranged as an Advanced Study Institute of the *NATO Security Through Science Programme* with support from the town and county Marktoberdorf and the Deutsche Akademische Austausch Dienst (DAAD). We thank all authorities involved.

THE EDITORS

Part I
Architectures, Design and Interfaces

Ralph-Johan Back

Manfred Broy

Thomas A. Henzinger

Bertrand Meyer

INCREMENTAL SOFTWARE CONSTRUCTION WITH REFINEMENT DIAGRAMS

Ralph-Johan Back
Abo Akademi University and Turku Centre for Computer Science
Turku, Finland
backrj@abo.fi

Abstract We propose here a mathematical framework for *incremental software construction* and for *controlled software evolution*. The framework allows incremental changes of a software system to be described on a high architecture level, but still with mathematical precision so that we can reason about the correctness of the changes. The framework introduces *refinement diagrams* as a visual way of presenting the architecture of large software systems. Refinement diagrams are based on lattice theory and allow reasoning about lattice elements to be carried out directly in terms of diagrams. A refinement diagram proof will be equivalent to a Hilbert like proof in lattice theory.

We use *refinement calculus* as the logic for reasoning about software systems. The calculus models software parts as elements in a lattice of predicate transformers. . In this way, we can use refinement diagrams to reason about the properties of software systems. We show here how to apply refinement diagrams and refinement calculus to the incremental construction of large software system. We concentrate on three topics: (i) *modularization* of software systems with component specifications and the role of information hiding in this approach, (ii) *layered extension* of software by adding new features one-by-one and the role of inheritance and dynamic binding in this approach, and (iii) *evolution of software* over time and the control of successive versions of software.

Keywords: Incremental software construction, refinement calculus, stepwise feature introduction, refinement diagrams, layered software, software evolution, version control, class diagrams, lattices, diagramatic reasoning, object oriented programming

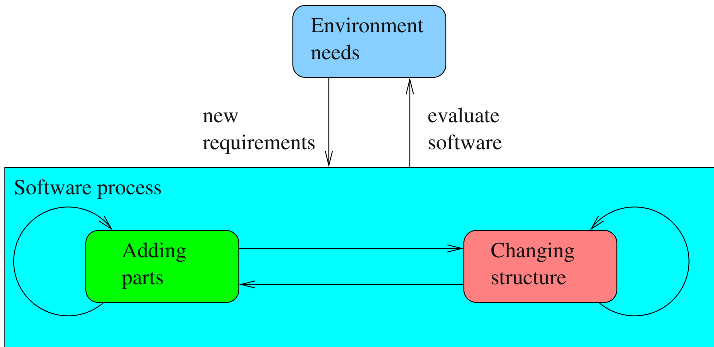
1. Introduction

We are interested here in a logical framework that will support the construction of large, correct software systems in an *incremental* and *layere*d fashion. This means that the software system is built in small increments, part by part, always checking that the correctness of the system is preserved by the exten-

M. Broy et al. (eds.), Engineering Theories of Software Intensive Systems, 3–46.
© 2005 *Springer. Printed in the Netherlands.*

sion. We will assume that the system has not been completely specified when we start building it. Rather, the requirements on the system are influenced by the system built thus far, and by changing expectations in the environment. Hence, the framework must also support an *evolutionary* approach to software construction: the system is never ready, but continuously evolving to meet different demands. At the same time as we are adding new increments to the software system, we are also accumulating design errors, which make further increments more and more difficult. Hence, the systematic extension of the software must be punctuated by frequent *redesigns* , which improve the software architecture and allow for further extensions.

The following diagram shows in a very simple way the overall evolution of a software system:



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, and based on this, new requirements are given that influence the further development of the system. The system itself is built by alternating between adding new parts that implement required features and internal redesign of the software in order to meet new demands.

This view of software evolution should be contrasted with an alternative view, where we start from a well-defined and complete specification (or set of requirements) for the software to be built, and then proceed to build it in a software project without changing the specifications any more. This view is not inconsistent with the above, because there is a difference of scale here. A project would typically add one or more features to an existing software system, by a sequence of increments. We do not want the specifications to change too much while implementing these new features. The project can also be a major restructuring of a system that has grown out of phase with the changing requirements. Thus, the sequence of successive software projects constitute the software evolution. The important thing here is that while carrying out a software project, one should bear in mind that there will be other software projects

that continue from the point where this project ended, and that the software should therefore be structured so that it can evolve smoothly.

The issues that need to be addresses when designing a framework for software evolution include the following:

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

We are here going to concentrate on the first three questions. We are presently also working on answers to the last two questions [Back, 2002, Anttila et al., 2002, Back et al., 2002], but this is beyond the scope of this paper.

Basic approach. We will use *refinement calculus* as the basic framework for software evolution. Originally, this calculus was proposed as a programming logic for stepwise program refinement [Back, 1980, Back, 1988, Morgan, 1990, Back and von Wright, 1998], but its applicability has been extended considerably over the years .It has been used for modeling different kinds of software, from distributed and parallel systems to asynchronous circuits to object oriented systems and UML diagrams (e.g., [Back et al., 1996, Back et al., 2000, Back et al., 1999b, Back and Sere, 1991]) .

Refinement calculus is based on looking at program statements as predicate transformers, as originally proposed by Dijkstra [Dijkstra, 1976, Dijkstra and Scholten, 1990]. In refinement calculus, we introduce a *refinement ordering* on predicate transformers. Intuitively, the refinement ordering models replacement: a statement S is refined by another statement T, denoted $S \sqsubseteq T$, if S can be replaced by T in any program context without compromising program correctness. The predicate transformers form a complete lattice with refinement as the lattice ordering.

The predicate transformer approach is, however, more versatile than just modelling program statements. Different domains that are central to software construction fit together into a hierarchy of lattices that are connected by *pointwise extension* [Back and von Wright, 1998]. Reasoning about software can be carried out in this hierarchy, at different levels of detail. In this hierarchy, we can fit simple reasoning about state functions and state relations, as well as more complex reasoning about classes and concurrent or interactive processes.

One of the basic methods for mastering the complexity of large engineering systems is to use diagrams and drawings. The same approach is also used for software, in particular when describing software architecture. UML, the unified modelling language, has become particularly popular in the last ten years. However, very often these software diagrams are rather informal, based on more or less intuitive software concepts. This should be contrasted with, e.g., construction drawings for buildings. Such drawings have a very precise meaning, so that the systems can be built from these without even consulting the people who have made the drawings.

We want to achieve the same level of precision for software architecture diagrams. We will do this by providing a *diagrammatic way* of reasoning about the construction and correctness of software systems in the refinement calculus. Essentially, what we will provide is a diagrammatic way of reasoning about lattice elements, and then apply this to software construction. Because of the intended application area, we refer to the diagrams that we use for reasoning as *refinement diagrams*. However, the refinement diagrams can be used for reasoning about any kind of lattice elements, not only software constructs.

Refinement diagrams will provide a simple way of visualizing the architecture of a software system, and allows us to reason about the construction and correctness of software at the architecture scale. As we will show below, they are also very precise: a refinement diagram derivation is essentially isomorphic to a Hilbert-like proof in lattice theory.

Usually, reasoning in a mathematical structure like a lattice means establishing some general properties for the structure in question (*analyzing* properties of the structure). This would be the mathematicians way of working. In software engineering, the focus is different. There we are interested in building a piece of software that satisfies specific requirements (*constructing* elements in the structure). The requirements have been laid down before the construction, or they become evident during the construction process. The mathematicians viewpoint is still needed, however, because we need to show that the construction does indeed satisfy the requirements (*correctness*).

The term that describes the required lattice element can be very complex, so the construction needs to be carried out in an *incremental* manner. The property to be established for the term can also be quite complex, and is preferably checked during the construction of the term rather than after the construction. Thus, our logical framework must support incremental construction of large lattice terms, and allow for proofs of specific properties about these terms to be carried out during the construction. We will below show that refinement diagrams do satisfy these conditions.

The paper is structured as follows. In the next section, we introduce *lattices* as an algebraic structure. We give the basic properties of lattices, and we introduce refinement diagrams as a way to describe lattice properties and to

reason about lattice elements. In Section 3, we introduce a *diagrammatic proof method* for refinement diagrams and show that a refinement diagram proof is equivalent to a Hilbert-like proof in lattice theory.

Section 4 gives a brief overview of refinement calculus from a lattice theoretic point of view, to show how lattices can be used as the semantic basis for software systems. We introduce the refinement calculus hierarchy, to show that we may need to carry out reasoning in different lattices at the same time.

In Section 5, we use refinement diagrams to analyze modular software construction, in particular the use of specifications and information hiding in building large systems. We consider situations where information hiding is advantageous and situations where information hiding should not be enforced. In Section 6, we look at software extension, and show how to use refinement diagrams to describe the extension of software with new features. The central technique to be used here is *layering*, which provides for the extension of existing software using inheritance and dynamic binding. Section 7 then finally ties together these threads into an overall view of software evolution with refinement diagrams, and proposes a *software editor* or *versions control system* based on refinement diagrams.

Related and earlier work. An earlier version of refinement diagrams has been described in [Back, 1991]. The present version is influenced by UML class diagrams. The way we reason about refinement diagrams is obviously also influenced by category theory diagrams [Barr and Wells, 1990]. 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* [Back, 2002] 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 [Back et al., 1999a, Back and Sere, 1991, Back et al., 2000] .

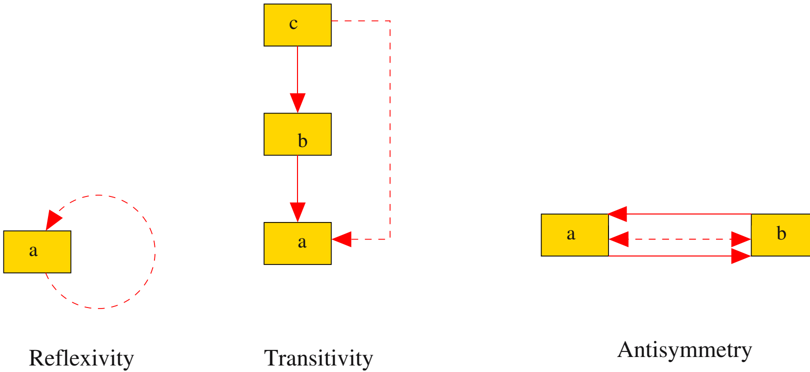
2. Lattices and refinement diagrams

We will below define lattices and their basic properties (see e.g. [Birkhoff, 1961, Davey and Priestley, 1990] for an overiew of lattice theory, or [Back and von Wright, 1998] for a shorter overview of lattices). At the same time, we will introduce *refinement diagrams* as a way of expressing the lattice properties in an intuitive and visual way.

Posets and categories. A *partially ordered set* (or *poset*) is a set A together with an ordering $\sqsubseteq$ that is *reflexive*, *transitive* and *antisymmetric*. This means that the following holds in any poset, 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*)

We capture these properties in the following diagrams:



Here the first figure shows reflexivity, the second transitivity and the third antisymmetry. Each diagram describes a universally quantified implication: the solid arrows indicate assumed relationships, while 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.

As an example, consider the middle diagram. It says the following: if we know initially that $a \sqsubseteq b$ holds (the lower arrow) and that $b \sqsubseteq c$ holds (the upper arrow) for some arbitrary elements a, b, and c, then we may deduce that $a \sqsubseteq c$ also holds (the dashed arrow from c to a).

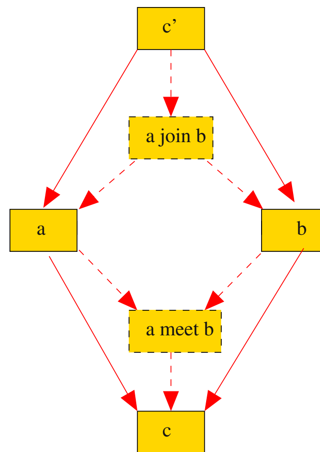
We refer to these diagrams as *refinement diagrams*. This name indicates the intended use of the diagrams: the lattice elements are program parts, and the ordering corresponds to *refinement* between program parts. Intuitively, we can think of refinement as permission for replacement: $a \sqsubseteq b$ means that the part a can be replaced by program part b (in any context).

The poset can be generalized to a category, if we want that the arrows are labelled. The diagrammatic representation of algebraic entities are familiar from category theory. Here we will emphasize the use of this kind of diagrams for reasoning about software construction and software architecture in a refinement calculus framework.

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)*

These properties are illustrated by the following diagram.

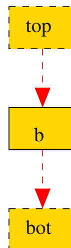


Here the meet and join elements are shown as dashed, because their existence is inferred.

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 a in the lattice, we have that

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

This property is shown in the following diagram:



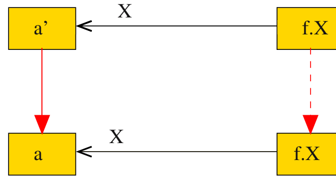
The top and bottom elements are dashed here, because their existence can be inferred by the boundedness property.

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 and the greatest element of the empty set, while $\top$ is the greatest element of the whole lattice and the least element of the empty set.

Functions and terms on lattices. A function $f : A \to B$ from poset A to poset B is *monotonic,* if for any elements $a, a' \in A$

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

(We write function application using dot-notation: $f.x$ stands for the more familiar $f(x)$). The partial ordering on A is indicated by $\sqsubseteq_A$ and the partial ordering on B by $\sqsubseteq_B$. Monotonicity is expressed by the following diagram:



Note that the ordering on the left is in A and the ordering on the right is in B. We indicate the *dependency* of $f.X$ on a by an open arrow from $f.X$ to a. Here $f.X$ is a *lattice term* that is constructed by applying the lattice function f to the variable X that ranges over lattice elements.

A lattice term is in general constructed by applying lattice operations on lattice constants and lattice variables. We write a lattice term as $t[X_1, \ldots, X_m]$ to indicate that it depends on the lattice variables $X_1, \ldots, X_m$. An example of a lattice term is, e.g., $f.((\bot \sqcup X_1) \sqcap g.X_2)$, where f and g are two operations on lattices.

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. As an example, consider the diagram

This shows the term $t[X_1, X_2]$, where X_1 is instantiated with the term t_1 and X_2 is instantiated with the term t_2. The middle box thus denotes the term $t[t_1, t_2]$. The explicit indication of the lattice variables on the arrows can be omitted, if it is clear from the context what is meant by the arrows.

We can also show the same term box with the subterms as nested boxes (on the left below), or using aggregation as in UML (on the right):

The difference between this representation and the previous is that the subterms in the latter representations cannot be shared. In the previous representation, we can have two or more terms using the same subterm.

In general, we assume that there is a collection of monotonic functions (*operations*) available on a given lattice. As the composition of monotonic functions is also monotonic, this means that we can build more complex monotonic functions out these simpler functions using composition. In addition, we assume that there are *constants* that denote specific lattice elements ($\bot$ and $\top$are two constants that always exist in any bounded lattice, but we usually need other constants as well). We say that a lattice term is *monotonic*, if it is built out of monotonic lattice functions.

Least fixpoints. In general, the dependencies between terms in a diagram may be circular. In that case, we need a more elaborate notion of what a box denotes in the diagram. For this purpose, we need to introduce the notion of fixed points of functions on lattices.

A monotonic function $f : A \to A$ on a complete lattice A has a unique *least fixed point,* denoted $\mu.f \in A$ (here μ is the fixpoint operator that gives the least fixed point for any monotonic function f). 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)*

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)*

The least fixed point and the greatest fixed point are illustrated in the following diagram:

We can use fixpoint induction to establish properties of fixpoint. Another proof technique is given later, in section 5.2.

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.

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 $A = \{1, \ldots, m\}$. As above, the product of a collection of (complete) lattices is a (complete) lattice. [1]

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

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

Lattice homomorphisms. Consider a function $h : A \rightarrow B$, where A and B are lattices. Then h is a *lattice meet homomorphism* if

$$h.(a \sqcap b) = h.a \sqcap h.b$$

In a similar way, we define *lattice join homomorphism,* a *bottom homomorphism*, a *top homomorphism* and so on. The function is a *complete join homomorphism* if

$$h.(\sqcup A) = \sqcup h.A$$

holds for any set of lattice elements A.

We can define homomorphism also for other operations on lattices, in the same way as we defined it for meet and join.

Mutual recursive definitions. An *environment* is a tuple of monotonic lattice terms, $E = (t_1[X], \ldots, t_n[X])$, where $X = (X_1, \ldots, X_m)$ is a tuple of variables that range 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 $i = 1, \ldots, m$.

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 $A^n \to A^n$. This function is monotonic on the complete lattice $(A^n, \sqsubseteq)$. Hence, this function has a *least fixed point* $\mu E = \mu.\widetilde{E}$. Thus, 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. (Dually, we can define the *greatest system* νE defined by the environment).

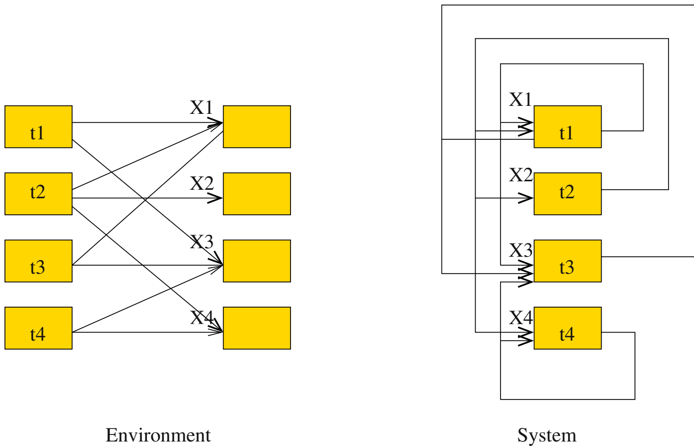
A consequence of this is that we can use unfolding to determine the meaning of an environment:

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

Intuitively, this means that the system defined by the environment is the (potentially infinite) unfolding of the environment.

We will usually describe a system by an equation $X = E$. The system described is then the solution $X = \mu E$, where $X_i = \mu E_i$ now denotes the ith element of the system.

The following figure illustrates an environment E (on the left) and the system μE (on the right) that is defined by the environment:



The system is here the solution to the equation

$$(X_1, X_2 X_3, X_4) = (t_1[X_1, X_3], t_2[X_1, X_2, X_4], t_3[X_1, X_3, X_4], t_4[X_3, X_4])$$

(we have here indicated the occurrence of the components of X in each term rather than just X). We see how the dependency arrows are bent back to the terms themselves in the system, thus providing the infinite unfolding semantics for the system defined by the environment.

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

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

This can be used to reason about refinement in a complex system. Assume that we have a term $t_i[X]$ in the system, and that we make some small change to this term, to get $t'_i[X]$. If this change is such that $t_i[X] \sqsubseteq t'_i[X]$, then we may infer that $E \sqsubseteq E'$ holds for the changed environment E',and therefore that $\mu.E \sqsubseteq \mu.E'$ holds for the new system that is determined by the change. 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 the terms.

3. Diagrammatic reasoning

The diagrammatic notation that we have used to describe the lattice rules can be extended to a proof system for refinement diagrams. The basic idea of such a *refinement diagram proof* (or *derivation*) is the following:

1 The entities of the diagram are lattice terms (shown here as rectangels, but we can also use other graphical notation for terms), lattice ordering (shown as a refinement arrow), lattice equality (shown by double arrow) and dependency relations (shown as usage arrow). Nested boxes are interpreted as a dependency of the enclosing box on the inner box.

2 We assume that the lattice terms in a diagram together form an environment E , with each occurrence of a lattice term t in the diagram corresponds to some element t_i in the environment tuple. The same lattice term can occur in two or more places in the diagram, but each occurrence has its own index in the tuple.

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

4 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.

5 We may associate names with the diagram boxes. These names can be understood as the variable names in X . 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).

6 An entity in the diagram may be annotated. The annotation may provide additional information about the entity. For instance, for software components the refinement arrows would in general be annotated by an *abstraction function* (or relation) that is needed to determine the *data refinement* between the components.

The conventions (3) and (4) are very important. They can be explicated as follows. Assume that we have an environment $E = (t_1[X], \ldots, t_n[X])$, where the indexes $1, \ldots, n$ now identify the different occurrences of terms in the refinement diagram that describes E. Then a refinement arrow from box i to box j means that $\mu E_j \sqsubseteq \mu E_i$ holds. As already explained, we write μE_i for $\pi_i^m(\mu.(\lambda X \cdot E[X]))$.

A *refinement diagram proof* (or *derivation*), is a refinement diagram where the ordering relations in the diagram are numbered by consecutive integers. This numbering shows the order in which the relations have been introduced in the diagram. With each number we associate a proof rule that justifies the introduction of this arrow, together with the possible side conditions that must hold for this inference to be valid. New entities may only be introduced into the diagram if justified by some proof rule. No entities may ever be removed from the diagram.

The proof rules used in the diagram can be textual proof rules, or they can be diagrammatic ones, like the inference rules we have presented above. In the latter case, we can apply the proof rule if the solid part of the rule diagram matches the proof diagram; and we may then add any or all of the dashed entities in the rule diagram to the proof diagram.

As an example, consider the refinement diagram proof in Figure 1: Note that the infered rules remain dashed in the proof diagram, indicating which parts of the diagram had to be specifically assumed and which parts could be infered by some inference rules.

The numbering of the entities in the diagram means that there is an equivalent textual presentation of the proof as a Hilbert-like proof in the theory of lattices. In this textual proof, each step is numbered, and is either justified as an axiom, as an assumption or as an inference that is drawn from some previous steps using some inference rule. As an example, this is the Hilbert like proof that corresponds to the above derivation:

1. $a \sqsubseteq b$ (assumption)

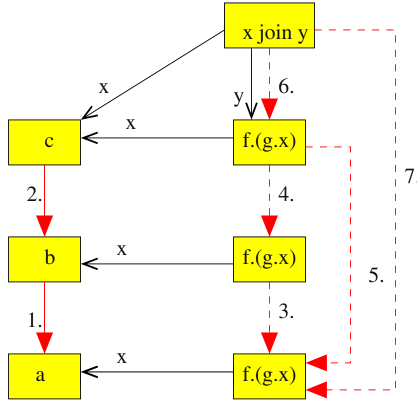


Figure 1. A refinement diagram proof

2. $b \sqsubseteq c$ (assumption)

3. $f.(g.a) \sqsubseteq f.(g.b))$ (from 1 by monotonicity)

4. $f.(g.b) \sqsubseteq f.(g.c)$ (from 2 by monotonicity)

5. $f.(g.a) \sqsubseteq f.(g.c)$ (from 3 and 4 by transitivity)

6. $f.(g.c) \sqsubseteq c \sqcup f.(g.c)$ (least upper bound)

7. $f.(g.a) \sqsubseteq c \sqcup f.(g.c)$ (from 5 and 6 by transitivity)

Note that the Hilbert proof contains more detail, because it also justifies each step. We will assume that the justifications in the diagram proof are given in a separate document (or with hyperlinks), because writing out the justification in the diagram itself is likely to be messy.

Rather than considering diagrammatic reasoning as a logic for establishing truth (a proof system), we will usually apply it as a *logic of construction*. This means that we construct some complex environment step by step, by building up the diagram for this environment and all the time checking that the environment built so far satisfies our requirements to the extent that they can. The purpose of building up the environment is that we are interested in defining (constructing) some specific term with the help of the environment, the term of interest itself being part of the environment.

A diagrammatic proof (with justifications for the steps) can be shown in a rather convenient manner as an animated presentation of the diagrams. Then one shows the diagram building up step by step, and carefully identifies the new entities that are placed at the diagram and also shows the justification for this.

Constructing such a presentation is rather simple using standard presentation software.

4. Lattice of program parts

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, an idea that was introduced by Dijkstra [Dijkstra, 1976]. For any programming language statement S, we define its meaning $wp.S$, which is a predicate transformer. This predicate transformer computes for any predicate (*postcondition*) q on the state space another predicate $wp.S.q$, which is the *weakest precondition* for statement S to terminate in a state satisfying q.

A statement that can fail to terminate in any initial state then describes the predicate transformer $abort$ (defined by $abort.q = false$ for any q), while a statement that is guaranteed to always terminate and establish any postcondition describes the predicate transformer $magic$ (defined by $magic.q = true$ for any q). There cannot be any such statement in reality, because among other things, it would also guarantee that the program terminates in a final state that satisfies $false$, which is impossible. Hence the name $magic$ for this predicate transformer. It turns out that it is a good mathematical notion, even if it does not exist in reality.

The refinement ordering $S \sqsubseteq T$ essentially 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 this relation by

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

[Back, 1980, Back and von Wright, 1998]. It also means that any user of the statement S that only is interested in the functional properties of this statement, should not notice any difference if S is replaced by T.

The meet and join in the predicate transformer lattice also have very strong analogues in program behavior. The meet $S \sqcap T$ is the *demonic choice* between executing S or executing T. One of these alternatives is chosen, but we (the person who is interested in the result of the program) have no influence on which alternative is chosen. The join $S \sqcup T$ is the *angelic choice* between executing S or executing T. In this case, we can choose the alternative that suits our purpose better (and choose differently for different purposes).

The refinement calculus interprets software systems as elements in a lattice (of, e.g., predicate transformers). A simpler lattice interpretation of program statements is to interpret these as relations on the state space. On the other hand, we can build more complicated models to capture, e.g., classes in object oriented systems, or interactive systems, real-time systems or concurrent systems. A common feature of these models is that they often can be seen

as having some lattice theoretic properties. In fact, one often has to force the semantics into a lattice framework, because most systems will allow one form of recursion or another, and the simplest way of modelling recursion is as fix-points. However, fixed points require an underlying lattice structure (or something very close to a lattice, like a cpo), if one is to reason about the properties of the fixed points in any reasonable manner.

We will therefore here postulate 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 parts 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. The *refinement calculus* provides a hierarchy of complete lattices that together allow one to reason about complex software (and also hardware) systems. The refinement calculus hierarchy is built on top of an arbitrary collection of state spaces Σ, Γ, etc. In addition, we assume a collection of *agents* Ω for contracts. The hierarchy and its properties are discussed in much more detail in [Back and von Wright, 1998].

Figure 2 shows the main lattices in the refinement calculus hierarchy. The basic lattices in the hierarchy are

- 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 $a \sqcup b = a \vee b$.

- The *state predicate (or subset) lattice*

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

  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, $p \sqcup q = p \cup q$.

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

  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

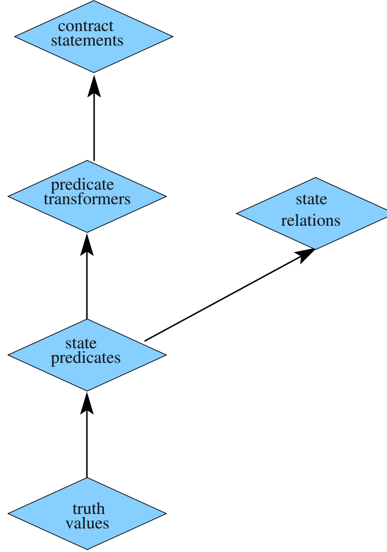


Figure 2. The refinement calculus hierarchy

largest relation is the true (universal) relation $True = \Sigma \times \Gamma$. Meet is intersection of relations and join is union of relations.

- The *predicate transformer lattice*

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

The ordering is *refinement*, defined as stated above by $S \sqsubseteq T \equiv (\forall q \cdot S.q \subseteq T.q)$. This is the pointwise extension of the ordering of predicates. 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 $S \sqcup T = (\lambda q \cdot S.q \cup T.q)$.

- The *contracts lattice*

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

Here 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.

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. Thus, 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. 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 with refinement diagrams in the refinement calculus hierarchy would typically involve reasoning on different levels in the hierarchy simultaneously. It is possible to show reasoning in different lattices in the same diagram. This can be very useful, but in the applications that we have here in mind, we do not use this facility.

5. Modularity and specifications

Let us start by considering software components with specifications. A *specification* is a description of a the functional (and sometimes also non-functional) behavior of a software component that describes *what* the component does, but not *how* it does it. We will consider a specification and an implementation as both parts (i.e., lattice terms) in a software system, and assume that there is a (possibly idealized) sense in which the specification can be executed, in the same way as an implementation can be executed.

A specification S is satisfied by an implementation T , if $S \sqsubseteq T$ holds. Mathematically, this means that $S \sqsubseteq T$ holds in the refinement calculus. In practice, one would usually have *data refinement* [Back, 1980, Hoare, 1972, Gardiner and Morgan, 1993, Back and von Wright, 2000] between the components rather than simple algorithmic refinement, but we will skip the distinction here. Intuitively, this means that we are allowed to replace the specification S by the implementation T in any context. Let us consider the refinement relation in somewhat more detail.

- A specification S can be satisfied by more than one implementation, $S \sqsubseteq T_1$, $S \sqsubseteq T_2$, $S \sqsubseteq T_3$, and so on. For instance, S could be a *standard* for some component, and T_1, T_2, T_3 could be different implementations of this standard which are provided by different vendors.
- 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