



Available at

www.ElsevierComputerScience.com

POWERED BY SCIENCE @ DIRECT®

Science of Computer Programming 49 (2003) 47–88

Science of
Computer
Programming

www.elsevier.com/locate/scico

The algebraic structure of interfaces[☆]

D.L.L. Rees^{a,1}, K. Stephenson^{b,*}, J.V. Tucker^c

^aBT Computing Partners, PP E1D, Enterprise House, 84-85 Adam Street, Cardiff CF24 2XF, UK

^bSystems Assurance Group, QinetiQ Trusted Information Management, Malvern WR14 3PS, UK

^cDepartment of Computer Science, University of Wales Swansea, Singleton Park, Swansea SA2 8PP, UK

Accepted 15 April 2003

Abstract

In this paper we examine formally the idea that the architecture of a system can be modelled by the structure of its interface expressed in terms of the interfaces for its components. Thus,

System Interface Architecture = Structured set of Sub-system Interfaces.

We specify an abstract model for interface definition languages (IDLs) based on this idea and the idea that an

Interface = Name + Imports + Body.

A set of interfaces is a *repository*. An interface architecture is a repository with some primary interfaces identified; the import dependencies between the interfaces of a repository are used to determine its structure.

The abstract model uses algebraic specifications to define the abstract syntax of a general IDL, and interface transformations using structural induction. We examine a flattening process which assembles a system interface from its components.

We use the general model to derive a simple IDL suitable for the design phase of object-oriented software development. This requires us to specify a form of *Body* that treats both data types and state, and in *Body* we explicitly distinguish between methods with and without side-effects, by *commands* and *queries*, respectively. We also consider alternative proposals for *Body* that yield new IDLs, including other object-oriented design languages and data type specification languages.

© 2003 Elsevier B.V. All rights reserved.

Keywords: Interface; Interface definition language; Imports; Flattening; Software architecture; Algebraic specification; Abstract syntax; Object-oriented architecture

[☆] Supplementary data associated with this article can be found in the online version, at doi:10.1016/j.scico.2003.04.001

* Corresponding author.

E-mail address: k.stephenson@swansea.ac.uk (K. Stephenson).

¹ Current address: Wireless Data Services, Alder Hills Park, Poole BH12 4AR, UK.

1. Introduction

In the design and development of large systems from smaller systems, there are different notions of interface and system architecture. These help combine large and small scale techniques for modelling, specification, programming, documentation, testing and validation. Most of the concepts of interface and system architecture are informal, of course, even in object-oriented software development.

Interfaces allow the composition of separate software components into larger interacting systems. The concept of a software interface has a long history in software engineering. It has emerged from several related concepts, e.g.:

- data hiding and abstraction (e.g., [31]);
- interfaces (e.g., [23,38]);
- specification languages (e.g., [29]);
- programming languages, both procedural (e.g., UCSD Pascal, Ada) and object-oriented (e.g., Simula, Smalltalk, Eiffel);
- object-oriented analysis and design (e.g., [13]);
- remote procedure call systems (e.g., [4]);
- component infrastructures (e.g., [39]); and
- object request brokers (e.g., [7,30]).

In the last few years, the interface concept has come of age. It has become clear that interfaces are an important organising idea at a number of levels of abstraction, from coding to the analysis and modelling phase of the software process. Timely questions are:

What is an interface? How do interfaces combine to form architectures?

At each level of abstraction, interfaces are intended to explicitly document aspects of the interaction between separate software systems. Interfaces provide a precise definition of the shared data definitions as well as the flows of control between separate components. Gathering together the components into a larger system, the interfaces determine a form of architecture for the system.

In this paper we present a simple model of the general notions of system interface and its associated architecture. This model is given axiomatically, being built from a series of algebraic specifications of its parts. Then, we use the general model to derive some interface definition languages, with a clear mathematical structure, suitable for the design phase for object-oriented software systems [5,10,26].

The general model is an abstract *interface definition language* (IDL) designed to define the interfaces of components and how they make the interface of a system. The IDL is developed from two ideas.

First, it analyses the idea that a

System Interface Architecture = Structured set of sub-system interfaces.

Thus, the IDL defines sets of interfaces called *repositories*, and gives them a structure to define system *architectures* in terms of interfaces.

Second, it is based on the idea that an

Interface = Name + Imports + Body.

The name can provide a unique identifier for the interface. The import list contains names of interfaces that may be required in the *Body*, but may or may not be in a repository.

The general model is defined axiomatically by giving an algebraic specification of the abstract syntax of the general IDL. The algebraic specification provides a structure for other IDLs which allows interface transformations to be defined by structural induction on abstract syntax.

An important transformation is that of *flattening* an interface architecture into a single interface. The act of flattening can be regarded as some process of assembling together the individual interfaces of a system's parts into a single interface that is an equivalent description of the whole system.

This flattening process helps our understanding by reducing notions of modularity or hierarchy present in system architecture via the *imports*. It is dependent on some operations

join and *tag*

on *Body*. Intuitively, one can think of the operation *join* as a form of textual substitution, and the operation *tag* as preserving the unique identity of components. Flattening has a rôle to play in specifying the semantics of a system in terms of the semantics of its component subsystems.

However, flattening is not straightforward. It raises interesting questions as to what we should do if:

- (i) an interface is dependent on another that is not present in a repository;
- (ii) names do not uniquely identify interfaces in a repository;
- (iii) we have already used an interface in the assembly process; or
- (iv) if interfaces are mutually dependent on one another?

Thus, in constructing the specifications, we need to add features to address such questions. In addition, there are several algorithms for flattening.

Next, we use the general model to derive two simple IDLs suitable for the design phase of object-oriented software development. Each IDL requires us to specify a form of *Body*. The concrete interfaces we derive are still more abstract than most practical IDLs, since we abstract away from supporting the pragmatic concerns of distributed computing. Our proposed notions of *Body* focus on the data types and states of systems and their components.

Most object-oriented IDLs provide a number of basic data types, from which new data types may be defined in an interface. Interfaces declare interactions between components by declaring a list of named *methods*. Typically these methods are declared with typed parameters and often return values. Our first object-oriented concept of an interface has been simplified by replacing this conventional notion of a method with the notion of:

- (i) *commands* that can change the internal state of an implementation, and
- (ii) *queries* that merely return values from a component without altering state.

Our notion of commands and queries is based on [43].

Our second object-oriented notion of an interface is based on the conventional notion of method without the distinction between queries and commands.

Here is the structure of the paper. In Section 2, we give some motivations for a general theory of interfaces and IDLs; this discussion extends beyond the limited contribution of this paper. In Section 3, we produce an algebraic specification for defining interfaces and IDLs. We provide the first application of this specification in Section 4, to provide an algebraic specification for an IDL for object-oriented design. We propose other applications of our specification in Section 5: an IDL for the more familiar model of methods, and an IDL for abstract data types. In Section 6, we reflect on our motivations, the abstract model and new directions for research.

We assume the reader is experienced in using algebraic specifications and abstract and concrete representations of syntax (e.g., [35,17,41]).

2. Informal description of models of interface definition languages

In this section, we reflect on interfaces and their uses, and prepare for our abstract models for IDLs. We return to this messy raw material in our concluding remarks (Section 6).

2.1. Examples of interfaces

2.1.1. Data hiding, abstract data types and algebraic specification languages

Data hiding is one of the primary concepts that needs a clear, explicit concept of interface. If we are to compose systems from parts, it is essential that we carefully design the interconnections between these parts. Interfaces provide a basic means of designing and documenting these interconnections, without complete information on how data is represented and operations implemented.

The development of modules in the 1970s stimulated research on abstract representations of code and components [45]. An important development was that of algebraic specification languages, where code is specified abstractly by a functional interface and a set of laws that operations obey. In such languages, modularity and importing specifications are natural, and lead to specification architectures for software. The algebraic theory of data type specifications uses signatures as interfaces, and sets of equations or conditional equations for laws. Although complicated, some general results about correctness and term-rewriting properties for modularity and importing are known [24].

What makes this approach important is

- (i) its strong theoretical foundation, laid using algebra and equational term rewriting [16,27,44,25], and
- (ii) the power of the slowly maturing software tools such as OBJ [19], ASF + SDF [1,41], Maude [9].

The notion of an interface in algebraic specification languages is both fundamental and clear, in theory and practice. It has been stable since the 1970s.

2.1.2. Object-oriented programming

In object-oriented programming, the notion of a class of objects defines a set of operations that can be applied to those objects [12,20]. The notion of class in many object-oriented programming languages subsumes the notion of interface, because classes often declare a set of permitted operations on objects and define the implementation and internal state that accompanies the declaration.

However, some languages have a more explicit notion of interface: classes that are explicitly declared with no attributes and no implementation code (for example, Java interface classes, Objective-C's protocol classes). These *implementation-free* classes exist only to be extended by other classes that provide a complete implementation for all the operations declared; such interfaces are useful for inheritance.

In object-oriented programming, classes are imported to reuse data types and methods. A class C is created from one or more superclasses P, Q, R, \dots which are inherited by C ; this importation process is called *multiple inheritance* and the superclasses P, Q, R, \dots are called *ancestors* of C . In particular, the interface of C is an extension of the interfaces of its superclasses P, Q, R, \dots and must implement all the declarations of its superclasses' interfaces. In programming, *implementation inheritance* takes this further by allowing the reuse of implementation. If class C does not require a different definition of a declaration then the absence of an implementation means that the implementation for the declaration in one of P, Q, R, \dots will be reused.

The ambiguity problem for multiple inheritance is: *Given a method declared in a class C and more than one superclass of C , determine which superclass method implementation is bound to the method declaration in C .*

In the object-oriented programming context, implementation-free interface classes also provide a simple solution to the ambiguity problem caused by multiple inheritance.

Many different disambiguation rules are possible. For example: the Python language [42] will transitively search each superclass in the order in which they are declared to be inherited; C++ requires explicit qualification of the invocation with the name of the explicit superclass from which the implementation is to be invoked; the Eiffel language adopts a renaming mechanism. The diversity of approaches to disambiguating multiple inheritance can make object-oriented models language-specific.

A simpler approach is to restrict implementation inheritance so that each class can inherit implementation from at most one class, thus avoiding ambiguity. Such implementations preserve the advantages of inheritance by allowing *multiple interface inheritance*. This multiple interface inheritance allows inheritance of special interface classes — classes which have declarations but provide no implementation and define no state. The Smalltalk-80 [20], Objective-C [11], and Java languages adopt this strategy.

The empty interface contains no declarations other than its own name. Systems involving interface inheritance can use empty interfaces to distinguish a particular class with a name. For example, the interface `java.io.Serializable` in [21] is used to indicate that all objects instantiated from classes inheriting `Serializable` are to be stored during serialization.

The notion of interface is fundamental but far less clear and stable in its theory and practice in object-oriented languages.

2.1.3. Object-oriented analysis and design

The notion of interface also plays a key part in object-oriented system analysis and design. Here a class is defined to participate in many rôles. When a class has several ancestors, it must implement all the operations declared for its ancestors, therefore it clearly must be usable in a number of different rôles, namely those of its ancestors and its own unique rôle. The Syntropy methodology [10] introduces the notion of *viewpoints*. A viewpoint is seen as a particular named subset of the interface supported by a class. We also note that the Unified Modelling Language (UML) has a notation for constraining the role of a class in an association to one of its interfaces (pp. 146–147 in [5]).

The notion of different viewpoints for a class can be used in practice. The Interface Segregation Principle encourages programmers to create a number of abstract interfaces along with an implementation class [26]. The interfaces explicitly define the rôles that the implementation class will perform. The implementation class then inherits the relevant interface classes, and thus can be used in a number of separate, named rôles. In typed languages, declarations are made using an appropriate interface type rather than the implementation class type. Due to the type system, the implementation class can then be used in that context, but access to features is limited by the interface. Thus, interfaces can provide a subtle form of data hiding in the design process.

The notion of interface is fundamental and has great scope and potential.

2.1.4. Components

Component infrastructures aim to provide concrete, cross-language standards for invoking methods in other components. The recent emergence of component based development infrastructures (for example [39]), has introduced a problem: *How can we re-use, extend, deploy and configure software at runtime?* Interfaces for components are fundamental for solutions to this problem.

Some approaches to component technology have involved defining interfaces in binary format between different components, whilst other approaches have been based on a standard declaration language, i.e., an IDL. An early example of the binary approach to component interoperability is the Microsoft Component Object Model (COM). An early and very influential example of the IDL approach is the object request broker CORBA (see Section 2.1.5).

Microsoft's COM [6] defines interfaces in terms of arrays of pointers to functions in the C programming language. It relies on binary format interfaces to minimise the additional performance cost of component-based systems. It allows system composition in terms of "interfaces" that are defined essentially in terms of C code declarations (such as `typedef`). The code-level definition forms the standard as opposed to the interface definition language, which exists as a convenient declaration language for developers and tool-makers, rather than providing the basis for standardising component interaction.

Both interfaces and implementations are registered in the Microsoft Windows system registry which provides the functionality of a repository.

Microsoft's COM uses many notions of name and identity. The key notion though is that of a *globally unique identifier* (GUID). GUIDs are 128 bit binary numbers.

A standard algorithm is used to generate GUIDs based on clock time, a counter and hardware serial numbers. Although conflicts are very unlikely, GUIDs do not guarantee global uniqueness.

2.1.5. Object request brokers

In order to allow runtime support and greater flexibility, some component infrastructures have been based on the *broker architectural pattern*, which involves an intermediate system, a *broker*, that connects and manages components in a potentially diverse software environment [7]. The notion of an IDL plays a central rôle in such systems, by providing a relatively simple common language for declaring interactions between components. The IDL provides abstraction from location, platform and implementation language concerns.

The Common Object Request Broker Architecture (CORBA). The CORBA standard [30] defines common protocols and infrastructure for connecting components together using interfaces. It relies on the syntax of an IDL and a common abstract object model. It defines conversion maps from interfaces to a number of programming languages such as C, C++ and Smalltalk.

Although distribution concerns can be abstracted from, by writing interfaces that can be used to communicate with remote and local objects, efficient distributed computing demands interfaces which declare fewer methods taking more parameters to reduce the communications overhead. Hence such interfaces focus on minimising calls rather than on the usual object-oriented programming notion of methods that provide a fundamental set of features for manipulating some class of objects.

In CORBA, interfaces are made unique by using a hierarchical system of user-assigned names. CORBA has separate repositories for interfaces and implementations. Interface repositories are necessary, since only some languages (usually the typed languages) will use interfaces declared directly in IDL. CORBA allows programmers to query interfaces and construct method invocations at runtime. This functionality allows systems to query and manipulate objects using interfaces that may not have been designed when the system was constructed. The interface repository provides the data necessary for such a dynamic service.

In component technologies, interfaces are so fundamental they achieve a conceptual independence. The notion of an interface is being clarified by practical developments.

2.2. Concepts

2.2.1. Some general notions about IDLs

The general notion of interface in this paper tries to identify *some* common ground between the various languages and systems above. It is more abstract than practical IDLs and it focuses on the architecture of a system.

First, we note the fundamental features of *naming* and *importing* in working with interfaces. All interfaces have a *unique name*, and an interface may acquire the features of an existing interface by naming the interface and using an *import* construct. There

are many forms of importation, of course (e.g., inheritance and instantiating generic types).

The declaration of interface names for importing requires a collection of named interfaces against which these import names can be resolved. Thus, naming and importing are “local” processes taking place in some “global” context.

We collect together a library of interfaces which we call a *repository*. Names refer to interfaces in a repository. In general, libraries can have elaborate indexing systems for cataloguing. Some programming languages and IDLs group interfaces together in a repository by attaching a common prefix to names. For example, this tagging accounts for the namespace concept in C++ and the CORBA IDL, and for the package notion in Java.

Since *import* is a transitive concept, we consider the notion of a *dependency tree* for some particular interface which records the interfaces of all components used in creating the interface. The dependency tree is the basis of an abstract notion of architecture.

2.2.2. Some basic notions about object-oriented IDLs

We refine this picture of a general IDL. Most object-oriented IDLs provide a number of basic data types and each interface *may* also define *new data types* constructed from the basic data types, for example records, sequences and enumerated types. (Although it may be controversial, we consider the notion of data and object types to be complementary. The modelling and separate implementation benefits of such a strategy can be found in [34].) Thus, declarations for data types mean that signatures are part of object-oriented interfaces.

An interface declares interactions between components by declaring a list of named *methods*. Typically these methods are declared with typed parameters which may return values.

Our first concept of an interface has been simplified to provide a semantically clearer model, by replacing this conventional notion of method with *commands* which are permitted to change the internal state of an implementation, and *queries* that merely return values from a component without causing state changes. Our notion of commands and queries is based on that used in Eiffel and given in [43], although similar notions exist in the Syntropy [10] and Catalysis methods [14]. This IDL is a simplification of one used in [33], where such IDLs are used to model the inheritance and encapsulation mechanisms found in C++, Eiffel and Java.

Our second concept of an interface is based on conventional methods.

Since our IDL involves the concept of mathematical signatures, we support the notion of “non-object” (or “non-component”) types by declaring new sorts in our interfaces. This mechanism provides a simple, programming language-neutral means of declaring “non-object” types.

2.2.3. Disclaimer on architectures

Software architecture is a developing field, which is not without its controversies [32,37]. We use the term interface architecture to constitute a structured set of interfaces, and consider this definition appropriate in the context of interface-based design.

We consider object-oriented systems and we see component-based software as an extension of basic object-oriented software.

It is clear that object-oriented principles can be used as a basis for specification and design of other architectural styles. Many architectural styles, however, do not require an object-oriented foundation. In [37] for example, object-oriented systems are seen as one of many different architectural styles.

Some architectural styles have both object-oriented and non-object-oriented implementations, for example, the pipe and filter style described in [28]: the Unix operating system implements notions of pipes and filters directly without recourse to object-oriented concepts. The Java core Application Programming Interface package `java.io`, however, follows the pipe and filter architectural style by specifying a collection of explicit interfaces and class interfaces for manipulating data streams. In our terms, the `java.io` package can be treated as a repository of related interfaces. Both approaches are valid and useful though we are primarily concerned with object-oriented architectures.

Other styles can be described in terms of interfaces but we do not claim that our IDLs are architecture description languages. (The limitations of IDLs for architectural description are considered in [36], where the formulation of a new concept, the “connector” is considered.)

3. An algebraic specification of general IDLs

An interface consists of declarations of a name, a list of imports and a body. We abstract away from what is inside an interface body to focus on the operations on bodies that characterise our general idea of interfaces and their architectures. To define a specific IDL, we define the nature of bodies.

3.1. Interfaces, repositories and architectures

We develop a language to define interfaces, sets of interfaces (repositories) and structured sets of interfaces (architectures). We begin by sketching these ideas using a concrete syntax for clarity. Shortly we will give an algebraic specification of an abstract syntax of this language.

We construct an *interface* in terms of other interfaces (its *imports*); an interface consists of three declaration sections:

```
interface    I
import      ... , J , ...
body        B
endinterface
```

An *architecture* of the form

```
architecture  A
primaries    P
repository   R
endarchitecture
```

is constructed from: a *repository*

```
repository  R
interfaces  ..., I, ...
endrepository
```

which lists the interfaces we have access to; and *primaries*

```
primaries  P
tops       ..., I, ...
endprimaries
```

that declares which of these interfaces form the “top” of a system.

The import dependencies of the interfaces within a repository determine a graph structure. The nodes of this graph are the interfaces of the repository, and each interface I with imports \dots, J, \dots gives the edges $\dots, I \rightarrow J, \dots$. The construction of the graph illustrates potential problems with the repository.

- *Repetition* of different interfaces with the same name.
- *Absence* of missing dependent interfaces.
- *Cyclicity* indicating mutual dependencies amongst interfaces.

For an interface, we can determine its subgraph. Editing this subgraph to deal with the above problems, we produce a non-cyclic *dependency tree*. By *flattening* the interfaces in a dependency tree we can assemble an interface. If this is self-contained, i.e., a *stand-alone interface*, we can proceed to ascribe it a semantics.

There are many algorithms for flattening, for example, using depth-first or breadth-first searches. We give two methods of flattening. The first process incrementally generates the flattened interface directly using recursion. The second method is indirect, involving the construction and traversal of dependency trees.

3.2. Preliminaries on specifying records and lists

Declarations such as interfaces, repositories and architectures are commonly made from records (of mixed types of syntax) and lists (of single types of syntax). Thus,

the common underlying form of the data is that of records and lists. We shall be using algebraic specifications in which data is generated by constructors, and whose design is abstracted from various flavours of records and lists. In this section we describe their general form which we will use to model the specifications that will be presented later.

3.2.1. Records

We construct records of length n over given specifications D_1, \dots, D_n , with an operator d ; we project out the component fields with operations d_1, \dots, d_n .

specification	$R(D_1, \dots, D_n)$
imports	D_1, \dots, D_n
sorts	$R(D_1, \dots, D_n)$
constants	
operations	$d : D_1 \times \dots \times D_n \rightarrow R(D_1, \dots, D_n)$ $d_1 : R(D_1, \dots, D_n) \rightarrow D_1$ \vdots \vdots $d_n : R(D_1, \dots, D_n) \rightarrow D_n$
equations	$d_1(d(x_1, \dots, x_n)) = x_1$ \vdots $d_n(d(x_1, \dots, x_n)) = x_n$

3.2.2. Lists

Suppose $\mathcal{S} = (\Sigma_{\mathcal{S}}, E_{\mathcal{S}})$ is a specification, and D a sort of \mathcal{S} with equality. So \mathcal{S} contains the sort *Bool*, constants *tt*, *ff* and operation *equalsD*: $D \times D \rightarrow \text{Bool}$. We assume that interpretations of \mathcal{S} use the standard model of Booleans with equality.

Then we construct lists that store data of type D using the empty list εD and by adding an element to a list with a function *addD*.

We check whether an element is present in a list with *inD*, and whether two lists are element-wise equal with *eqD*. We remove *all* occurrences of an element from a list with *cutD*. We join two lists L_1 and L_2 together, with either:

- *appD* to *append* the elements of L_2 after those of L_1 ; or
- *mrgD* to *merge* L_1 and L_2 together by appending those elements in L_2 that do not already occur in L_1 .

Structural induction. We define the behaviour of list processing functions by structural induction. Thus, the typical form of a function f operating on lists and some parameters

\bar{x} is split according to its behaviour on

- (i) empty lists (determined by g), and
- (ii) non-empty lists (determined by h):

$$\begin{aligned} f(\varepsilon D, \bar{x}) &= g(\bar{x}), \\ f(\text{add}D(d, L), \bar{x}) &= h(f(L, \bar{x}), d, L, \bar{x}). \end{aligned}$$

specification $List_D(S)$	
imports	S
sorts	D^*
constants	$\varepsilon D : \rightarrow D^*$
operations	$\text{add}D : D \times D^* \rightarrow D^*$ $\text{in}D : D \times D^* \rightarrow Bool$ $\text{eq}D : D^* \times D^* \rightarrow Bool$ $\text{cut}D : D \times D^* \rightarrow D^*$ $\text{app}D : D^* \times D^* \rightarrow D^*$ $\text{mrg}D : D^* \times D^* \rightarrow D^*$
equations	$\text{in}D(d, \varepsilon D) = ff$ $\text{equals}D(d, d') = tt \Rightarrow \text{in}D(d, \text{add}D(d', L)) = tt$ $\text{equals}D(d, d') = ff \Rightarrow \text{in}D(d, \text{add}D(d', L)) = \text{in}D(d, L)$ $\text{eq}D(\varepsilon D, \varepsilon D) = tt$ $\text{eq}D(\varepsilon D, \text{add}D(d, L)) = ff$ $\text{eq}D(\text{add}D(d, L), \varepsilon D) = ff$ $\text{equals}D(d, d') = tt$ $\Rightarrow \text{eq}D(\text{add}D(d, L), \text{add}D(d', L')) = \text{eq}D(L, L')$ $\text{equals}D(d, d') = ff$ $\Rightarrow \text{eq}D(\text{add}D(d, L), \text{add}D(d', L')) = ff$ $\text{cut}D(d, \varepsilon D) = \varepsilon D$ $\text{equals}D(d, d') = tt \Rightarrow \text{cut}D(d, \text{add}D(d', L)) = \text{cut}D(d, L)$ $\text{equals}D(d, d') = ff \Rightarrow \text{cut}D(d, \text{add}D(d', L)) = \text{add}D(d', \text{cut}D(d, L))$ $\text{app}D(\varepsilon D, L) = L$ $\text{app}D(\text{add}D(d, L), L') = \text{add}D(d, \text{app}D(L, L'))$ $\text{mrg}D(\varepsilon D, L) = L$ $\text{in}D(d, L') = tt \Rightarrow \text{mrg}D(\text{add}D(d, L), L') = \text{mrg}D(L, L')$ $\text{in}D(d, L') = ff \Rightarrow \text{mrg}D(\text{add}D(d, L), L') = \text{add}D(d, \text{mrg}D(L, L'))$

3.3. Algebraic specification of a general IDL

We construct a seven-sorted specification, using nine separate specifications, for the abstract syntax of our general IDL, as shown in Fig. 1.

The concrete syntax described in Section 3.1 constitutes a model for the specifications.

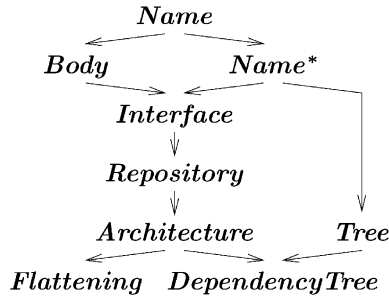


Fig. 1. Structure of the specification of IDLs.

3.3.1. Names

We need a specification *Name* of names to identify interfaces. Its precise nature is irrelevant, except we assume we can test for equality, and there exists a distinguished name *top*. In addition, we assume that we have operations *absent* and *repeat* on names which we shall use to flag the status of a name, and have no axioms.

```

signature Name
imports Bool
sorts Name
constants top :  $\rightarrow$  Name
operations equalsName :  $Name \times Name \rightarrow Bool$ 
            absent :  $Name \rightarrow Name$ 
            repeat :  $Name \rightarrow Name$ 
  
```

We also need a specification *Name** for lists of names built from that of *Name*. We apply the construction $List_D(S)$ of Section 3.2.2 to the specification $S = Name$ and the sort $D = Name$, so that we generate lists of elements of type *Name*, whilst retaining the tests on individual names. Thus, we generate the specification:

```

specification Name*
imports Name
sorts Name*
constants  $\varepsilon Name$  :  $\rightarrow Name^*$ 
operations addName :  $Name \times Name^* \rightarrow Name^*$ 
            inName :  $Name \times Name^* \rightarrow Bool$ 
            eqName :  $Name \times Name^* \rightarrow Bool$ 
            cutName :  $Name \times Name^* \rightarrow Name^*$ 
            appName :  $Name^* \times Name^* \rightarrow Name^*$ 
            mrgName :  $Name^* \times Name^* \rightarrow Name^*$ 
  
```

equations

$$\begin{aligned}
& \text{inName}(n, \varepsilon\text{Name}) = \text{ff} \\
\text{equalsName}(n, n') = \text{tt} \Rightarrow & \\
& \text{inName}(n, \text{addName}(n', N)) = \text{tt} \\
\text{equalsName}(n, n') = \text{ff} \Rightarrow & \\
& \text{inName}(n, \text{addName}(n', N)) = \text{inName}(n, N) \\
& \text{eqName}(\varepsilon\text{Name}, \varepsilon\text{Name}) = \text{tt} \\
& \text{eqName}(\varepsilon\text{Name}, \text{addName}(n, N)) = \text{ff} \\
& \text{eqName}(\text{addName}(n, N), \varepsilon\text{Name}) = \text{ff} \\
\text{equalsName}(n, n') = \text{tt} \Rightarrow & \\
& \text{eqName}(\text{addName}(n, N), \text{addName}(n', N')) = \text{eqName}(N, N') \\
\text{equalsName}(n, n') = \text{ff} \Rightarrow & \\
& \text{eqName}(\text{addName}(n, N), \text{addName}(n', N')) = \text{ff} \\
& \text{cutName}(n, \varepsilon\text{Name}) = \varepsilon\text{Name} \\
\text{equalsName}(n, n') = \text{tt} \Rightarrow & \\
& \text{cutName}(n, \text{addName}(n', N)) = \text{cutName}(n, N) \\
\text{equalsName}(n, n') = \text{ff} \Rightarrow & \\
& \text{cutName}(n, \text{addName}(n', N)) = \text{addName}(n', \text{cutName}(n, N)) \\
& \text{appName}(\varepsilon\text{Name}, N) = N \\
& \text{appName}(\text{addName}(n, N), N') = \text{addName}(n, \text{appName}(N, N')) \\
& \text{mrgName}(\varepsilon\text{Name}, N) = N \\
\text{inName}(n, N') = \text{tt} \Rightarrow & \\
& \text{mrgName}(\text{addName}(n, N), N') = \text{mrgName}(N, N') \\
\text{inName}(n, N') = \text{ff} \Rightarrow & \\
& \text{mrgName}(\text{addName}(n, N), N') = \text{addName}(n, \text{mrgName}(N, N'))
\end{aligned}$$
3.3.2. *Bodies*

Bodies are many and varied, and possess a variety of general and specific transformations.

signature	<i>Body</i>
imports	<i>Name</i> *
sorts	<i>Body</i>
constants	<i>null</i> : \rightarrow <i>Body</i>
operations	<i>tag</i> : <i>Name</i> * \times <i>Body</i> \rightarrow <i>Body</i>
	<i>join</i> : <i>Body</i> \times <i>Body</i> \rightarrow <i>Body</i>

As a minimum, we assume a specification **Body** of interface bodies has a constant *null* representing a body with no content; and operations *join* to “concatenate” two

bodies, and *tag* to rename a body's components by tagging them with locational information determined by a list of names. The operations *join* and *tag* characterise the nature of *import*.

Example. Typically, the structure of a body is given by a record specification $R(\mathbf{Decl}_1, \dots, \mathbf{Decl}_m)$ of component declarations with constructor function *body* and projections $decl_1, \dots, decl_m$, (recall Section 3.2). Furthermore, the structure of each component $Decl_i$ is that of *Body*, i.e., $Decl_i$ imports $Name^*$ and has operations

$$\begin{aligned} null_{decl_i} &: \rightarrow Decl_i \\ tag_{decl_i} &: Name^* \times Decl_i \rightarrow Decl_i \\ join_{decl_i} &: Decl_i \times Decl_i \rightarrow Decl_i \end{aligned}$$

In such cases, we define the operations *null*, *join* and *tag* of **Body** in terms of these constituent operations of the component elements. For example,

$$tag(N, B) = body(tag_{decl_1}(N, decl_1(B)), \dots, tag_{decl_m}(N, decl_m(B))).$$

We comment further on the operations of *tag* and *join* in Sections 3.3.3 and 3.3.4, but note that we need only postulate the existence of *tag* and *join*.

3.3.3. Tagging

The purpose of tagging is to record locational information. For flexibility, we store locational information as a list of names. For example,

$$tag(addName(m, addName(n, \varepsilon Name)), B)$$

indicates the relocation of body *B* from *n* to *m*. Thus, if we have no locational information, we specify that tagging has no effect:

$$tag(\varepsilon Name, B) = B.$$

Two natural methods of recording location are:

- (i) *local context tagging* or single tagging (typically the original location); and
- (ii) *global context tagging* or multiple tagging.

Definition (*Local context tagging*). Local context tagging is *innermost* if tag satisfies the *innermost tagging rule*:

$$tag(addName(m, addName(n, N)), B) = tag(addName(n, N), B).$$

Local context tagging is *outermost* if tag satisfies the *outermost tagging rule*:

$$tag(addName(n, N), B) = tag(addName(n, \varepsilon Name), B).$$

In this paper, local context tagging will be innermost.

Definition (*Global context tagging*). Global context tagging requires that there are no additional simplification axioms for tag.

3.3.4. Joining

Intuitively, the act of joining is that of adding body components together. Even at the level of abstract bodies, we can ask the questions: *What is the effect of joining a body to itself? What is the effect of joining a declaration to a body B that is already present in B?*

The act of tagging resolves many of the difficulties that could arise when we add the same body component from different locations. In the case that we add a body to itself, global context tagging will differentiate between the two contexts. For local context tagging though, we need to determine the outcome. Natural solutions are that we keep all or just one copy of B . The latter behaviour is specified by the following.

Definition (*Idempotent joining*). Joining is *idempotent* if join satisfies

idempotency rule: $join(B, B) = B$

commutativity rule: $join(B_1, B_2) = join(B_2, B_1)$

associativity rule: $join(B_1, join(B_2, B_3)) = join(join(B_1, B_2), B_3)$.

3.3.5. Modelling interfaces

An interface is a record of name, import name list and body. To the specification $R(\mathbf{Name}, \mathbf{Name}^*, \mathbf{Body})$ we add a function *extend*, which in conjunction with tagging we shall use to describe the meaning of import.

<p>specification <i>Interface</i></p> <p>imports $\mathbf{Name}^*, \mathbf{Body}$</p> <p>sorts $\mathbf{Interface}$</p> <p>constants</p> <p>operations $intf : \mathbf{Name} \times \mathbf{Name}^* \times \mathbf{Body} \rightarrow \mathbf{Interface}$ $name : \mathbf{Interface} \rightarrow \mathbf{Name}$ $imports : \mathbf{Interface} \rightarrow \mathbf{Name}^*$ $body : \mathbf{Interface} \rightarrow \mathbf{Body}$ $extend : \mathbf{Interface} \times \mathbf{Interface} \rightarrow \mathbf{Interface}$</p> <p>equations</p> $name(intf(n, I, B)) = n$ $imports(intf(n, I, B)) = I$ $body(intf(n, I, B)) = B$ $extend(I, J) = intf(name(I),$ $\quad \quad \quad mrgName(cutName(name(J), imports(I)),$ $\quad \quad \quad \quad \quad imports(J)),$ $\quad \quad \quad \quad \quad join(body(I), body(J)))$
--

We define $extend(I, J)$ to extend an interface I with the imports and body of J , whilst removing J 's name from I 's import list.

Thus, to define the flattening of an interface I with respect to an interface J , we extend I with the tagged interface J , producing the interface:

$$\text{extend}(I, \text{intf}(\text{name}(J), \text{imports}(J), \text{tag}(\text{addName}(\text{name}(J), \varepsilon\text{Name}), \text{body}(J))))),$$

which has:

- (i) I 's name;
- (ii) I 's imports with J 's name replaced with any new imports from J ; and
- (iii) I 's body joined to J 's body which has been tagged with J 's name.

Definition (*Stand-alone interfaces*). An interface with no imports is called a *stand-alone interface*.

The aim of flattening is to try to produce a stand-alone interface. Let

$$\text{StandAlone} = \{\text{intf}(n, \varepsilon\text{Name}, B) \mid n : \text{Name}, B : \text{Body}\}$$

be the set of terms denoting stand-alone interfaces.

Example. Cases involving self-referencing are interesting to consider. For example, given interfaces

```
interface  I
import    J
body      BI
endinterface
```

and

```
interface  J
import    I
body      BJ
endinterface
```

in concrete syntax, then flattening I with respect to J is the interface

```
interface  I
import    I
body      BI, J :: BJ
endinterface
```

where “,” denotes body concatenation and “::” tagging in the concrete syntax.

Flattening this interface with respect to I using local-context tagging, gives:

```
interface    I
import      J
body        BI, J :: BJ, I :: BI
endinterface
```

Clearly, if we were intending to remove imports then we would end up in a cyclic loop. Thus, we shall need some constraints when flattening interfaces.

3.3.6. Modelling repositories

A repository is a list of interfaces. Using our list construction with sort $D = \text{Interface}$, we add functions *yield* to pick an interface with a given name from a repository and *needs* to return its imports.

Interesting problems arise when we process repositories.

Repetition If there is more than one interface with a given name in a repository, then we choose to take the first interface that we locate.

Absence If an interface with name n is absent from a repository R , we choose to flag this by returning the interface which has imports *absent*(n) and empty body.

Cyclicity The mutual (including self-) dependency between interfaces impacts on their use in defining flattening as described in Section 3.3.8.

```
specification Repository
imports    Interface
sorts     Repository
constants  $\varepsilon\text{Interface} \rightarrow \text{Repository}$ 
operations  $\text{addInterface} : \text{Interface} \times \text{Repository} \rightarrow \text{Repository}$ 
               $\text{needs} : \text{Name} \times \text{Repository} \rightarrow \text{Name}^*$ 
               $\text{yield} : \text{Name} \times \text{Repository} \rightarrow \text{Interface}$ 
equations
               $\text{needs}(n, R) = \text{imports}(\text{yield}(n, R))$ 
               $\text{yield}(n, \varepsilon\text{Interface}) = \text{intf}(n, \text{addName}(\text{absent}(n), \varepsilon\text{Name}), \text{null})$ 
 $\text{equalsName}(n, \text{name}(I)) = \text{tt} \Rightarrow$ 
               $\text{yield}(n, \text{addInterface}(I, R)) = I$ 
 $\text{equalsName}(n, \text{name}(I)) = \text{ff} \Rightarrow$ 
               $\text{yield}(n, \text{addInterface}(I, R)) = \text{yield}(n, R)$ 
```

3.3.7. Modelling architectures

We use our record construction on Name^* and Repository to construct a specification for architectures. Note that because names are present in repositories, we do not bother to import them separately.

specification	<i>Architecture</i>
imports	<i>Repository</i>
sorts	<i>Architecture</i>
constants	
operations	$arch : Name^* \times Repository \rightarrow Architecture$ $primaries : Architecture \rightarrow Name^*$ $repository : Architecture \rightarrow Repository$
equations	$primaries(arch(P, R)) = P$ $repository(arch(P, R)) = R$

We shall define operations of flattening and forming dependency trees over this structure in Sections 3.3.8 and 3.3.9.

3.3.8. Flattening

The act of flattening an architecture replaces all references to interfaces with their instantiations, wherever possible. More specifically, flattening joins the tagged bodies of the primaries, with all their recursively flattened imports.

Note that different definitions of tagging (Section 3.3.3) and joining (Section 3.3.4) will yield different flattening algorithms.

Direct Flattening Algorithm. We axiomatise this algorithm with a function

$$flatten : Architecture \rightarrow Interface$$

whereby $flatten(A)$ gives the flattened interface of the architecture A . We define $flatten$ in terms of a sub-function

$$f : Architecture \times Interface \times Name^* \rightarrow Interface$$

such that $f(A, I, V)$ constructs the flattened interface of the architecture A using the incrementally generated interface I , and list of visited interfaces V . Let R_{-V} be the repository R with the interfaces named in V removed. Then,

$$f(arch(P, R), I, V) = \text{the interface } I \text{ joined with the tagged result of flattening the primaries } P \text{ on the repository } R_{-V}.$$

So, to compute $flatten(arch(P, R))$, we evaluate the function f on an interface $I = intf(top, P, null)$ and the list $V = addName(top, \epsilon Name)$ of names:

$$flatten(arch(P, R)) = f(arch(P, R), intf(top, P, null), addName(top, \epsilon Name))$$

We define f by double recursion on names. In the first recursion, we join the result of flattening the first name n in the list of primaries on the repository R_{-V} to that of flattening its dependents on the repository $R_{-addName(n, V)}$. To this interface, we join that

from the second recursive stage which flattens the remaining names in the list with their dependents on the repository R_V .

specification *Flattening*

imports *Architecture*

sorts

constants

operations *flatten* : *Architecture* \rightarrow *Interface*

f : *Architecture* \times *Interface* \times *Name*^{*} \rightarrow *Interface*

equations

flatten(*arch*(P, R))
 = f (*arch*(P, R), *intf*(*top*, P , *null*), *addName*(*top*, ε *Name*))

f (*arch*(ε *Name*, R), I, V) = I

eqName(*needs*(n, R), *addName*(*absent*(n), ε *Name*)) = *tt*

\wedge *inName*(n, V) = *ff*

$\Rightarrow f$ (*arch*(*addName*(n, N), R), I, V)
 = f (*arch*(N, R),
 intf(*name*(I),
 addName(*absent*(N), *cutName*(N , *imports*(I))),
 body(B)),
 V)

eqName(*needs*(n, R), *addName*(*absent*(n), ε *Name*)) = *ff*

\wedge *inName*(n, V) = *ff*

$\Rightarrow f$ (*arch*(*addName*(n, N), R), I, V)
 = f (*arch*(N, R),
 f (*arch*(*needs*(n, R), R),
 extend(I ,
 intf(n ,
 needs(n, R),
 tag(*addName*(n, V), *body*(*yield*(n, R))))),
 addName(n, V)),
 V)

inName(n, V) = *tt*

$\Rightarrow f$ (*arch*(*addName*(n, N), R), I, V)
 = f (*arch*(N, R),
 intf(*name*(I),
 addName(*repeat*(n), *cutName*(n , *imports*(I))),
 body(I)),
 V)

Let us consider how flattening works in the three interesting situations identified in Section 3.3.6, together with the case where the flattening process can be completed.

Note that these cases are not mutually exclusive, except an interface cannot be flattened in the cases where absence or cyclicity arise.

Repetition The presence in the repository of more than one interface with a given name is not directly evident from flattening as we did not flag this in Section 3.3.6. This could be treated though in a similar fashion to that of absence if wanted.

Absence Recall that if the repository R does not contain an interface n we defined $yield(n, R)$ to return an interface with name n , imports $absent(n)$ and empty body. As a consequence, the flattening process will produce an interface with an import $absent(n)$ for any interface n absent from R .

Cyclicity Similarly, if we detect a cycle $n \dots n$ we flag this by returning $repeat(n)$ in the imports list of the flattened interface. Note that we only flag the first occurrence of cyclic repetition, and not all the interfaces involved in this cyclicity.

Flattenable If all the interfaces required to flatten an architecture are present in a repository, we will return a stand-alone interface. We define

$$Flattenable = \{arch(P, R) \mid P : Name^*, R : Repository, \\ flatten(arch(P, R)) \in StandAlone\}.$$

Examples. Consider the sample cases illustrated in Fig. 2, where we have used local context tagging and idempotent joining.

In case (a), we have an example of a system in which flattening produces a stand-alone interface as all the interfaces required are present, and there is no cyclicity. Flattening starts by taking the interface with name top , imports n_1 and n_4 (the primaries of the architecture) and no body. The process flattens the import n_1 , creating the interface with name top , imports n_4 and body the joined and tagged components from the interfaces n_1 , n_2 , n_3 and n_4 . Then we repeat the process for the import n_4 , so that we remove n_4 from the import list and add n_4 's body components.

Note that we add the interface n_4 twice to the flattened interface that we construct, although this is not evident from the result we return because of the idempotent joining and local context tagging. In a global context tagging, this would be evident as the contexts n_4 in n_1 in top , and n_4 in top .

In case (b), we have an example of an interface that we cannot flatten into a stand-alone interface as we are missing the interface n_3 , and n_1 and n_2 are mutually dependent. Thus, we return an interface which has found n_1 , n_2 and n_4 , but is missing n_3 and there is a cyclic dependency involving n_1 .

Complexity. Suppose there are N interfaces in a repository and that the size of an interface is $O(M)$. In the worst case, the primaries of the architecture will be dependent (directly or indirectly) on every interface in the repository.

Suppose that the functions $join$ and tag are both $O(M)$. Then, in defining flattening, the functions $needs$ and $yield$ are dependent on N , all other basic functions called are $O(M)$ or $O(1)$, and $mrgName$ is $O(N^2)$. Thus, the complexity of flattening is $O(N * (N^2 + M))$, which if $M = O(N)$, is $O(N^3)$.

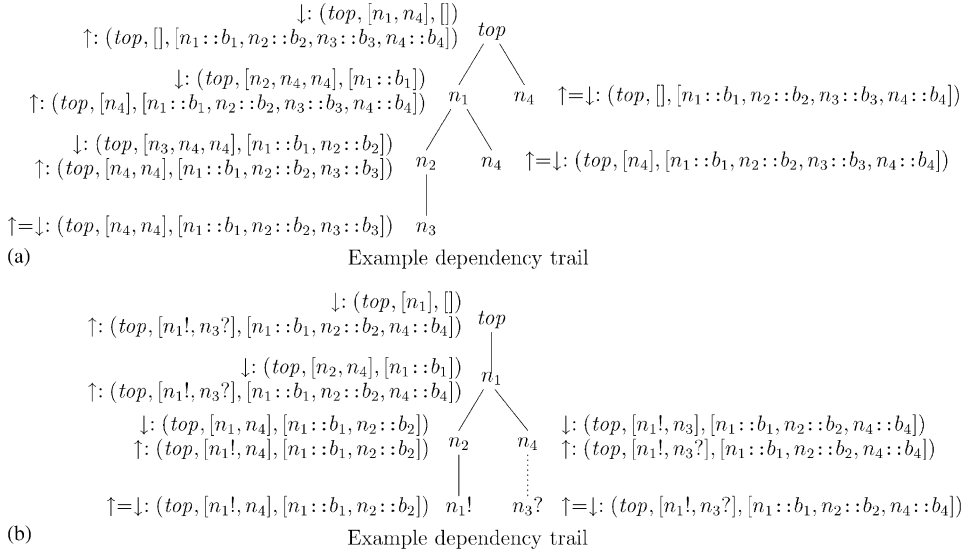


Fig. 2. Sample flattening cases. The symbol \downarrow denotes the initiation of a function call, and \uparrow the returned value. The progress of the function evaluation can be tracked through the tree by following the \downarrow symbols down the tree branches, to the leaves where upon the \uparrow symbols are then traced up through the branches. $(n, [i_1, \dots, i_m], [b])$ denotes an interface with name n , imports i_1, \dots, i_m and body b . In particular, tagging of body components is indicated by $::$, and joining by commas. In each case, we assume that an interface with name n_i has body b_i . Dotted lines in the dependency tree indicate missing interfaces (denoted $n?$) or cyclic repeated interfaces (denoted $n!$).

We now consider another algorithm for flattening, which is less efficient but more intuitive.

3.3.9. Dependency trees

We can represent the import dependencies between the interfaces of a repository with a dependency graph. We shall produce a finite tree representation of this graph to give the *dependency tree* of a system.

We start by describing the structure of trees as a tree of names with a post-order traversal function *traverse*, which records the exact position within the tree of every node: the function *rd* adds the parent to the path of each child.

specification	Tree
imports	$(Name^*)^*$
sorts	$Tree, Tree^*$
constants	$\varepsilon Tree \rightarrow Tree^*$
operations	$addTree : Tree \times Tree^* \rightarrow Tree^*$
	$tree : Name \times Tree^* \rightarrow Tree$
	$traverse : Tree \rightarrow (Name^*)^*$
	$traverses : Tree^* \rightarrow (Name^*)^*$
	$rd : Name \times (Name^*)^* \rightarrow (Name^*)^*$

equations

$$\begin{aligned}
\text{traverse}(\text{tree}(n, T)) &= \text{appName}^*(\\
&\quad \text{rd}(n, \text{traverses}(T)), \\
&\quad \text{addName}^*(\text{addName}(n, \varepsilon\text{Name}), \varepsilon\text{Name}^*)) \\
\text{traverses}(\varepsilon\text{Tree}) &= \varepsilon\text{Name}^* \\
\text{traverses}(\text{addTree}(t, T)) &= \text{appName}^*(\text{traverse}(t), \text{traverses}(Ts)) \\
\text{rd}(n, \varepsilon\text{Name}^*) &= \varepsilon\text{Name}^* \\
\text{rd}(n, \text{addName}^*(M, Ms)) &= \text{addName}^*(\\
&\quad \text{appName}(M, \text{addName}(n, \varepsilon\text{Name})), \\
&\quad \text{rd}(n, Ms))
\end{aligned}$$

We create an artificial root *top* for the dependency tree, with children, the primaries. If these interfaces import any other interfaces, they will appear as their children, and so forth. Circularities $n \dots n$ within the dependency graph are indicated by the single repetition of n as a leaf node of a given branch. We use a function *structure* to construct a dependency tree for an architecture. We define *structure* using two mutually dependent functions *dts* and *dt*: *dt* will construct the dependency tree for a single name, and *dts* will construct (by recursion) the dependency tree for a list of names.

specification *DependencyTree***imports** *Architecture, Tree***sorts****constants****operations** *structure* : *Architecture* \rightarrow *Tree**dts* : *Name*^{*} \times *Name*^{*} \times *Repository* \rightarrow *Tree*^{*}*dt* : *Name* \times *Name*^{*} \times *Repository* \rightarrow *Tree***equations**

$$\text{structure}(\text{arch}(P, R)) = \text{tree}(\text{top}, \text{dts}(P, \text{addName}(\text{top}, \varepsilon\text{Name}), R))$$

$$\text{dts}(\varepsilon\text{Name}, L, R) = \varepsilon\text{Tree}$$

$$\text{inName}(n, V) = \text{ff} \Rightarrow$$

$$\text{dts}(\text{addName}(n, N), V, R) = \text{addTree}(\text{dt}(n, V, R), \text{dts}(N, V, R))$$

$$\text{inName}(n, V) = \text{tt} \Rightarrow$$

$$\text{dts}(\text{addName}(n, N), V, R) = \text{addTree}(\text{tree}(\text{repeat}(n), \varepsilon\text{Tree}), \text{dts}(N, V, R))$$

$$\text{eqName}(\text{needs}(n, R), \text{addName}(\text{absent}(n), \varepsilon\text{Name})) = \text{tt} \Rightarrow$$

$$\text{dt}(n, V, R) = \text{tree}(\text{absent}(n), \varepsilon\text{Tree})$$

$$\text{eqName}(\text{needs}(n, R), \text{addName}(\text{absent}(n), \varepsilon\text{Name})) = \text{ff} \Rightarrow$$

$$\text{dt}(n, V, R) = \text{tree}(n, \text{dts}(\text{needs}(n, R), \text{addName}(n, V), R))$$

Examples. For example, Fig. 2 illustrates the dependency trees for two different architectures.

In case (a), the repository contains at least: the primary interface n_1 which depends on n_2 and n_4 ; the stand-alone primary interface n_4 ; the interface n_2 which depends on n_3 ; and the stand-alone interface n_3 .

In case (b), the repository contains at least: the primary interface n_1 which depends on n_2 and n_4 ; the interface n_2 which depends on n_1 ; and the interface n_4 which depends on the absent interface n_3 .

Indirect Flattening Algorithm. We can give an alternative definition for flattening ($flatten'$) with an explicit dependency tree. Using $extends$, we extend, from right-to-left, each of the interfaces given in the list of locations provided by the post-order traversal of an architecture's dependency tree.

```

specification IndirectFlattening
imports      DependencyTree
sorts
constants
operations   $flatten' : Architecture \rightarrow Interface$ 
                $extends : (Name^*)^* \times Repository \rightarrow Interface$ 
equations
 $flatten'(A) = extends(traverse(structure(A)), repository(A))$ 
 $extends(addName^*(addName(n, \varepsilon Name), \varepsilon Name^*), R)$ 
            $= (n, \varepsilon Name, null)$ 
 $equalsName(n, repeat(m)) = tt \vee equalsName(n, absent(m)) = tt$ 
 $\Rightarrow extends(addName^*(addName(n, N), M), R)$ 
            $= intf(name(extends(N, R)),$ 
                  $cutName(m, imports(extends(N, R))),$ 
                  $body(extends(N, R)))$ 
 $equalsName(n, repeat(m)) = ff \wedge equalsName(n, absent(m)) = ff$ 
 $\Rightarrow extends(addName^*(addName(n, N), M), R)$ 
            $= extend(extends(N, R),$ 
                  $intf(n,$ 
                      $depends(n, R),$ 
                      $tag(addName(n, N), body(yield(n, R))))))$ 

```

3.4. Term rewriting

For clarity, we have given a modular structure to the specifications. The meaning of the word import in our algebraic specifications is simply that of disjoint union.

(Or following the terminology of the paper, it is a local context tagging with idempotent join.)

Consider, for example, the specification for architectures with flattening but without explicit dependency trees. This is a 6-sorted specification with constructor functions:

- (i) *top*, *absent* and *repeat* producing terms of sort *Name*;
- (ii) ε *Name* and *addName* producing terms of sort *Name**;
- (iii) *null*, *join* and *tag* producing terms of sort *Body*;
- (iv) *intf* producing terms of sort *Interface*;
- (v) ε *Interface* and *addInterface* producing terms of sort *Repository*; and
- (vi) *arch* producing terms of sort *Architecture*.

In addition it will have accessor operations such as *name*, *imports*, *body*, and manipulator operations such as *cutName* and *flatten*.

The normal forms of architectures will be terms

$$\text{arch}(P, R),$$

where *P* and *R* are terms of sort *Name** and *Repository*, respectively, in normal form. As these are both list structures, these terms can be reduced down to:

$$\begin{aligned} P &= \text{addName}(n, N) \quad \text{or} \quad P = \varepsilon\text{Name}, \\ R &= \text{addInterface}(I, R) \quad \text{or} \quad R = \varepsilon\text{Interface}, \end{aligned}$$

where *n*, *N*, *I*, *R* are terms in normal form of the appropriate types. If we take the free model of bodies (i.e., with unqualified tagging and joining), then the normal forms of interfaces will be terms

$$\text{intf}(n, N, B),$$

where *n*, *N* and *B* are terms of sort *Name*, *Name** and *Body*, respectively, in normal form. The normal forms of stand-alone interfaces will be terms

$$\text{intf}(n, \varepsilon\text{Name}, B).$$

Note that no interfaces have sub-terms of sort interface.

3.5. Specifying interfaces for systems and their semantics

The abstract syntax for interface architectures can be used to construct an interface definition language for a particular class of systems. First, to obtain an abstract syntax for the particular language, we must choose an appropriate abstract syntax for *Name*, *Name** and, especially, *Body* and combine them with the abstract syntax for interface

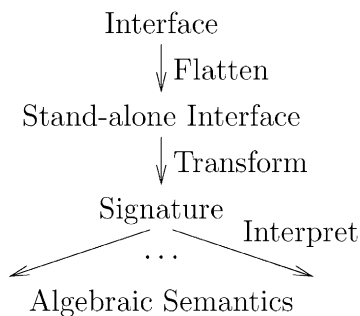


Fig. 3. The semantics of an architecture.

architectures. An interface architecture is a term $arch(P, R)$ with subterms P and R of sort $Name^*$ and $Repository$, respectively. These are built from terms of sort $Name$, $Name^*$ and $Body$.

Flattening models the process of building a system from components. The architecture term $arch(P, R)$ is a description of the system in terms of certain components, and the flattened architecture interface term $flatten(arch(P, R))$ is a description of an assembled system. Thus, the semantics of $arch(P, R)$ may be defined to be that of $flatten(arch(P, R))$ when this is stand-alone.

In defining the semantics $\llbracket \cdot \rrbracket$ of system interfaces in terms of the semantics of its components, flattening plays a rôle, as follows.

Suppose we have an IDL as above, and a semantics $\llbracket \cdot \rrbracket_0$ for its stand-alone system interfaces. Then, for a well-formed architecture $arch(P, R)$, we define

$$\llbracket arch(P, R) \rrbracket =_{\text{def}} \llbracket flatten(arch(P, R)) \rrbracket_0.$$

We are interested in the case that $\llbracket \cdot \rrbracket_0$ is based on a process which constructs algebraic signatures from stand-alone interfaces, and interprets the signature by an algebra as shown in Fig. 3. We illustrate the process in Section 4.4.

4. A simple object-oriented IDL

We use the general model of IDLs produced in Section 3 to derive a simple object-oriented IDL by specifying the *Body* component. The IDL that we produce captures the interactions between components, but abstracts away from supporting the pragmatic concerns of distributed computing of most practical IDLs. We separate these components into *commands* that can change the internal state of an implementation, and *queries* that merely return values from a component without altering the state (cf. [43]).

4.1. Informal description using concrete syntax

The IDL declares the data types and the programs that compute over them. Using a simple concrete syntax, we describe the form of an interface in the IDL. The *Body* is constructed from:

data type declarations of the usual form:

```
sorts ... , s , ...
constants ... , c : -> s , ...
operations ... , f : s(1)* ... * s(l) -> s , ...
```

for its data sets, constants and operations; and

program module declarations which we split into:

the state-altering modules

```
commands ... , p : s(1)* ... * s(m) , ...
```

that can be implemented using program procedures of the form:

```
procedure p in  $\bar{x}$  body S endprocedure
```

the state-querying modules

```
queries ... , q : s(1)* ... * s(n) -> s , ...
```

that can be implemented using program functions (restricted to having no side-effects) of the form:

```
function q in  $\bar{x}$  out y body S endfunction
```

Substituting these declarations for bodies in our general IDL of Section 3.1 yields an interface with seven declaration sections, of the form:

```
interface    I
import      ... , J , ...
sorts       ... , s , ...
constants   ... , c : -> s , ...
operations  ... , f : s(1)* ... * s(l) -> s , ...
commands    ... , p : s(1)* ... * s(m) , ...
queries     ... , q : s(1)* ... * s(n) -> s , ...
endinterface
```

We now specify the abstract syntax of this IDL.

4.2. Bodies

The abstract syntax of *Body* is constructed from component specifications: *Sort*^{*}, *Const*^{*}, *Opn*^{*}, *Command*^{*} and *Query*^{*}.

4.2.1. Component declarations

The structure of the specification of *Body* is that of typical general bodies described in Section 3.3.2. Thus, each of the component specifications *Decl*_{*i*}:

- (i) imports the specification *Name*^{*};

- (ii) has a constant $null_{decl_i} : \rightarrow Decl_i$;
- (iii) has a join operation $join_{decl_i} : Decl_i \times Decl_i \rightarrow Decl_i$; and
- (iv) has a tagging operation $tag_{decl_i} : Name^* \times Decl_i \rightarrow Decl_i$.

4.2.2. Component specifications

The structure of the body components is:

- **Sort**^{*} for a list of names;
- **Const**^{*} for a list of records of name and sort;
- **Opn**^{*} for a list of records of name, sort list and sort;
- **Command**^{*} for a list of records of name and sort list; and
- **Query**^{*} for a list of records of name, sort list and sort.

As each of these components has list structures, we define operations on the components by recursion on the constructor operations of lists (cf. Section 3.2). We illustrate the components' specification by considering **Sort**^{*} and **Opn**^{*}; the specifications for **Const**^{*}, **Command**^{*} and **Query**^{*} are similar to **Opn**^{*}.

Sorts. As the structure of **Sort**^{*} is a list of names, we import the specification **Name**^{*}. We define: $null_{sorts}$ to be the empty list of names; tag_{sorts} to add a name to a list if it is not already present (see Section 4.4 for motivation); and $join_{sorts}$ to be the merge operation of **Name**^{*}. Note that we have defined tag_{sorts} for an innermost local context tagging scheme.

specification Sort[*]	
imports	Name [*]
sorts	
constants	$null_{sorts} : \rightarrow Name^*$
operations	$tag_{sorts} : Name^* \times Name^* \rightarrow Name^*$
	$join_{sorts} : Name^* \times Name^* \rightarrow Name^*$
equations	
	$null_{sorts} = \varepsilon Name$
	$tag_{sorts}(N, N') = join_{sorts}(N, N')$
	$join_{sorts}(N, N') = mrgName(N, N')$

Operations. The structure of **Opn**^{*} is a list of individual operations, the structure of each is a record $R(Name, Name^*, Name)$ of name, list of sorts, and sort. Note that we deliberately ignore a natural desire to ensure the domain of operations is non-empty, to give a light touch to the discussion.

Thus, we import the specification **Name**^{*} where the empty declaration is the empty list of operations, i.e., we have both εOpn and $null_{opn}$ to emphasise the list structure of operations, whilst retaining the general structure of body components. We define

tagging to be on operation names using a function

$$\text{tag}_{\text{name}} : \text{Name}^* \times \text{Name} \rightarrow \text{Name}.$$

We choose to define joining as concatenating lists of operations together. Note that other variations on join_{opn} are also natural, such as only adding operations that are not already present, either by name and type, or just name.

<p>specification Opn^*</p> <p>imports Name^*</p> <p>sorts Opn</p> <p>constants $\text{null}_{\text{opn}} : \rightarrow \text{Opn}^*$</p> <p>operations $\text{opn} : \text{Name} \times \text{Name}^* \rightarrow \text{Opn}$ $\text{opname} : \text{Opn} \rightarrow \text{Name}$ $\text{dom} : \text{Opn} \rightarrow \text{Name}^*$ $\text{rng} : \text{Opn} \rightarrow \text{Name}$ $\text{join}_{\text{opns}} : \text{Opn}^* \times \text{Opn}^* \rightarrow \text{Opn}^*$ $\text{tag}_{\text{opns}} : \text{Name}^* \times \text{Opn}^* \rightarrow \text{Opn}^*$ $\text{tag}_{\text{opn}} : \text{Name}^* \times \text{Opn} \rightarrow \text{Opn}$ $\text{tag}_{\text{name}} : \text{Name}^* \times \text{Name} \rightarrow \text{Name}$</p> <p>equations</p> $\text{opname}(\text{opn}(n, d, r)) = n$ $\text{dom}(\text{opn}(n, d, r)) = d$ $\text{rng}(\text{opn}(n, d, r)) = r$ $\text{null}_{\text{opn}} = \varepsilon \text{Opn}$ $\text{join}_{\text{opns}}(O, O') = \text{appOpn}(O, O')$ $\text{tag}_{\text{opns}}(N, \varepsilon \text{Opn}) = \varepsilon \text{Opn}$ $\text{tag}_{\text{opns}}(N, \text{addOpn}(o, O)) = \text{addOpn}(\text{tag}_{\text{opn}}(N, o), \text{tag}_{\text{opns}}(N, O))$ $\text{tag}_{\text{opn}}(N, o) = \text{opn}(\text{tag}_{\text{name}}(N, \text{opname}(o)), \text{dom}(o), \text{rng}(o))$
--

This specification can be extended to specify the different forms of joining and tagging discussed in Section 3.3.2.

4.2.3. Body specification

We represent the abstract syntax of **Body** by records constructed from declarations for sorts, constants, operations, commands and queries. Our specification $\mathbf{R}(\text{Sort}^*, \text{Const}^*, \text{Opn}^*, \text{Command}^*, \text{Query}^*)$ for records applied to these bodies introduces operations bdy to create bodies, and $\text{sorts}, \dots, \text{queries}$ to project out their five declaration sections.

With this abstract structure for bodies, we can define the operations null , join and tag postulated in Section 3.3.2: we construct an empty body from empty declarations; we join bodies by joining their respective declaration sections; and we tag a body by tagging each of its declaration sections.

specification	<i>Body</i>
imports	<i>Sort*</i> , <i>Const*</i> , <i>Opn*</i> , <i>Command*</i> , <i>Query*</i>
sorts	<i>Body</i>
constants	<i>null</i> : \rightarrow <i>Body</i>
operations	<i>bdy</i> : $Sort^* \times Const^* \times Opn^*$ $\times Command^* \times Query^* \rightarrow Body$
	<i>sorts</i> : $Body \rightarrow Sort^*$
	\vdots \vdots
	<i>queries</i> : $Body \rightarrow Query^*$
	<i>join</i> : $Body \times Body \rightarrow Body$
	<i>tag</i> : $Name^* \times Body \rightarrow Body$
equations	
	$sorts(bdy(S, \dots, Q)) = S$
	\vdots \vdots
	$queries(bdy(S, \dots, Q)) = Q$
	$null = bdy(null_{sorts}, \dots, null_{queries})$
	$join(B_1, B_2) = bdy(join_{sorts}(sorts(B_1), sorts(B_2)), \dots,$ $join_{queries}(queries(B_1), queries(B_2)))$
	$tag(N, B) = bdy(tag_{sorts}(N, sorts(B)), \dots,$ $tag_{queries}(N, queries(B)))$

4.3. Specification of the IDL

We create a specification of the abstract syntax of the IDL sketched in Section 4.1 by simply substituting the specification for object-oriented bodies in Section 4.2 for the signature for bodies in Section 3.3. The architecture of the resulting specification is shown in Fig. 4 (cf. Fig. 1).

4.4. Semantics

We outline an algebraic semantics of the object-oriented IDL following the method sketched in Section 3.5.

Given an object-oriented interface architecture A , providing that

$$flatten(A) \in StandAlone,$$

then we define the semantics of A to be the semantics of $flatten(A)$. Thus, we give an algebraic semantics to the stand-alone interfaces.

The purpose of an object-oriented IDL is to provide interfaces for an object-oriented programming system. The semantics of a stand-alone interface I will be that of the component objects in the system which implement I .

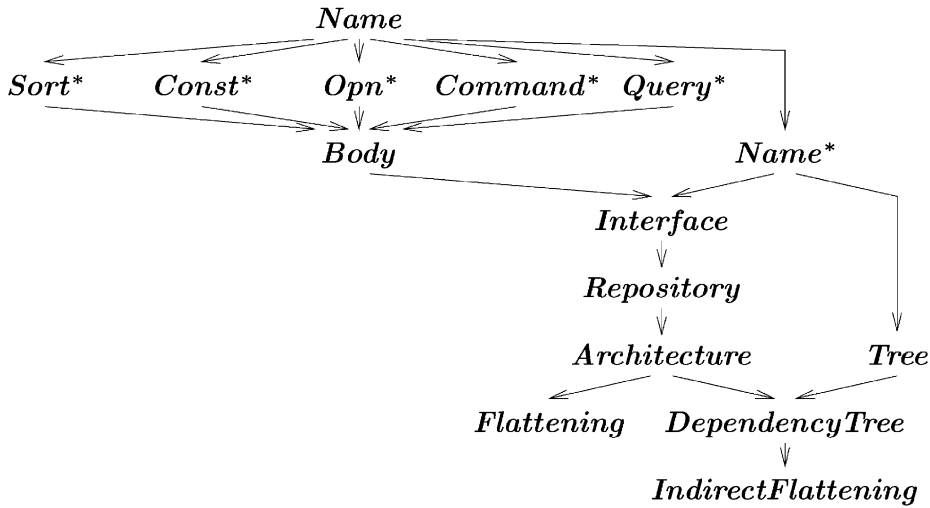


Fig. 4. Structure of the specification for the command-query IDL.

Mathematically, to model algebraically the semantics of a stand-alone interface I we must construct a signature Σ_I . (A specification for signatures can be obtained by reducing that of Section 5.2.) Algorithmically, to translate a stand-alone interface to a signature, we transform the commands and queries of an interface into functions, making explicit the internal computation state of the programming system: commands can alter states; queries can produce values from accessing the state.

In designing a semantics for an object-oriented system, we assume the existence of a set of objects indexed by a set OID of object *identifiers*, e.g.,

$$\{\text{nullid}, 0, 1, 2, \dots\}.$$

We suppose there is a mapping

$$\beta : OID \rightarrow \text{StandAlone}$$

that binds objects in the programming system to interfaces in the IDL, where $\beta(i)$ is the interface for the object indexed by i .

The global state, or *world*, of an object-oriented system is built from the local states of all the objects in the system. A world is a snapshot of the state of all objects in the system at an instant; its precise structure is complicated and highly system dependent. However, suppose the set of all worlds has the form

$$\text{World} = [OID \rightarrow \text{State}].$$

We need not go into detail on the structure of object states. In a simple case, the state of an object is given by

- (i) data for the sorts named in its interface;

- (ii) other objects that can be recovered (via their identifiers) from the same world value; and
- (iii) the name of the class from which the object was instantiated.

Consider a semantics of the IDL based on a simple single-threaded object-oriented language.

A method invocation can change

- (i) the states of any number of objects present in the system, and
- (ii) the number of objects in the system, by creating or destroying objects.

To model the semantics of method calls, we need a function that returns the updated world value which represents the system after invoking the method. This function depends on the initial world, the object on which the method is executing, and parameters. Each command method p needs a function

$$P : World \times OID \times A_{s(1)} \times \cdots \times A_{s(n)} \rightarrow World$$

and each query method q has a function

$$Q : World \times OID \times A_{s(1)} \times \cdots \times A_{s(n)} \rightarrow (World \times A_s).$$

Creating these functions for each method in a stand-alone interface gives an algebra that models the objects associated with interfaces.

(Recall from Section 4.2, that we tagged sorts by adding the interface name to the sort list if it was not already present. The reason for this decision is that the purpose of every interface is to present some object. Consequently, this information may be omitted from non-flattened interfaces, on the basis that it is implicit. When flattening, we need to extract this information as it is needed for the semantical analysis.)

Thus, in concrete syntax terms, given an interface repository and stand-alone interface I we can create the signature Σ_I to model an associated object-oriented programming system and object.

signature	
sorts	$\dots, s, \dots, oid, world$
constants	$\dots, c : \rightarrow s, \dots$
operations	$\dots, f : s(1) \times \cdots \times s(l) \rightarrow s, \dots$ $\dots, p : world \times oid \times s(1) \times \cdots \times s(m) \rightarrow world, \dots$ $\dots, q_{world} : world \times oid \times s(1) \times \cdots \times s(n) \rightarrow world, \dots$ $\dots, q_{data} : world \times oid \times s(1) \times \cdots \times s(n) \rightarrow s, \dots$

The study of constructions of this kind is, of course, a topic in itself.

5. Other IDLs

We consider some variations on the IDL given in Section 4, namely an alternative object-oriented IDL using a more conventional notion of method (Section 5.1), and an IDL for abstract data types (Section 5.2).

5.1. A conventional object-oriented method IDL

5.1.1. Informal description using concrete syntax

As for commands and queries, we declare the data type components and the program modules that compute over it, but now modules are all of the form:

```
methods ... , m: s(1)* ... * s(k) -> s', ...
```

Methods not returning a value are distinguished by a special return sort `void`.

Substituting these declarations and those for sorts, constants and operations, for bodies in our general IDL of Section 3.1 yields an interface with six declaration sections, of the form:

```
interface    I
import      ... , J , ...
sorts       ... , s , ... , void
constants   ... , c : -> s , ...
operations  ... , f : s(1)* ... * s(l) -> s , ...
methods     ... , m : s(1)* ... * s(k) -> s' , ...
endinterface
```

The method return type s' ranges over the sorts \dots, s, \dots and `void`. This is the only place where `void` is allowed to be used.

5.1.2. Bodies

We model the bodies for method-IDLs in a similar manner to that for commands and queries.

We import a specification $Method^*$ to model the method declarations; its structure is a list of records of name, sort list, and sort. Thus, the specification of $Method^*$ is similar to that of Opn^* given in Section 4.2. The null method is the empty list of methods (as for operations, we keep both $\varepsilon Method$ and $null_{method}$), joining methods appends one list of methods to another, and tagging attaches the interface name to each method's name.

Again, this specification can be extended to specify the different forms of joining and tagging discussed in Section 3.3.2.

```

specification Method*
imports      Name*
sorts       Method
constants    $\varepsilon\text{Method} \rightarrow \text{Method}^*$ 
                $\text{null}_{\text{method}} : \rightarrow \text{Method}^*$ 
operations   $\text{addMethod} : \text{Method} \times \text{Method}^* \rightarrow \text{Method}^*$ 
                $\text{method} : \text{Name} \times \text{Name}^* \rightarrow \text{Method}$ 
                $\text{mdname} : \text{Method} \rightarrow \text{Name}$ 
                $\text{dom} : \text{Method} \rightarrow \text{Name}^*$ 
                $\text{rng} : \text{Method} \rightarrow \text{Name}$ 
                $\text{join}_{\text{methods}} : \text{Method}^* \times \text{Method}^* \rightarrow \text{Method}^*$ 
                $\text{tag}_{\text{methods}} : \text{Name}^* \times \text{Method}^* \rightarrow \text{Method}^*$ 
                $\text{tag}_{\text{method}} : \text{Name}^* \times \text{Method} \rightarrow \text{Method}$ 
                $\text{tag}_{\text{name}} : \text{Name}^* \times \text{Name} \rightarrow \text{Name}$ 

equations
 $\text{mdname}(\text{method}(n, d, r)) = n$ 
 $\text{dom}(\text{method}(n, d, r)) = d$ 
 $\text{rng}(\text{method}(n, d, r)) = r$ 
 $\text{null}_{\text{method}} = \varepsilon\text{Method}$ 
 $\text{join}_{\text{methods}}(M, M') = \text{appMethod}(M, M')$ 
 $\text{tag}_{\text{methods}}(N, \varepsilon\text{Method}) = \varepsilon\text{Method}$ 
 $\text{tag}_{\text{methods}}(N, \text{addMethod}(m, M))$ 
   $= \text{addMethod}(\text{tag}_{\text{method}}(N, m), \text{tag}_{\text{methods}}(N, M))$ 
 $\text{tag}_{\text{method}}(N, m) = \text{method}(\text{tag}_{\text{name}}(N, \text{mdname}(m)), \text{dom}(m), \text{rng}(m))$ 

```

Assembling all the declarations of method bodies, we get:

```

specification Body
imports      Sort*, Const*, Opn*, Method*
sorts       Body
constants    $\text{null} : \rightarrow \text{Body}$ 
operations   $\text{bdy} : \text{Sort}^* \times \text{Const}^* \times \text{Opn}^* \times \text{Method}^* \rightarrow \text{Body}$ 
                $\text{sorts} : \text{Body} \rightarrow \text{Sort}^*$ 
                $\vdots$ 
                $\vdots$ 
                $\text{methods} : \text{Body} \rightarrow \text{Method}^*$ 
                $\text{join} : \text{Body} \times \text{Body} \rightarrow \text{Body}$ 
                $\text{tag} : \text{Name}^* \times \text{Body} \rightarrow \text{Body}$ 

```

equations

$$\begin{aligned}
 \text{sorts}(\text{bdy}(S, C, O, M)) &= S \\
 &\vdots \\
 \text{methods}(\text{bdy}(S, C, O, M)) &= M \\
 \text{null} &= \text{bdy}(\text{null}_{\text{sorts}}, \dots, \text{null}_{\text{queries}}) \\
 \text{join}(B_1, B_2) &= \text{bdy}(\text{join}_{\text{sorts}}(\text{sorts}(B_1), \text{sorts}(B_2)), \dots, \\
 &\quad \text{join}_{\text{methods}}(\text{methods}(B_1), \text{methods}(B_2))) \\
 \text{tag}(N, B) &= \text{bdy}(\text{tag}_{\text{sorts}}(N, \text{sorts}(B)), \dots, \\
 &\quad \text{tag}_{\text{methods}}(N, \text{methods}(B)))
 \end{aligned}$$

5.1.3. Specification of the IDL

The architecture of the specification of the IDL is shown in Fig. 5: by substituting the specification of method bodies for the signature for bodies in Section 3.3, we produce a specification constructed from 13 separate specifications.

5.1.4. Semantics

Following the semantics of commands and queries interfaces, we first flatten an interface architecture for method-interfaces into a single method-interface. If this produces a stand-alone interface, then we translate those methods that are declared as returning

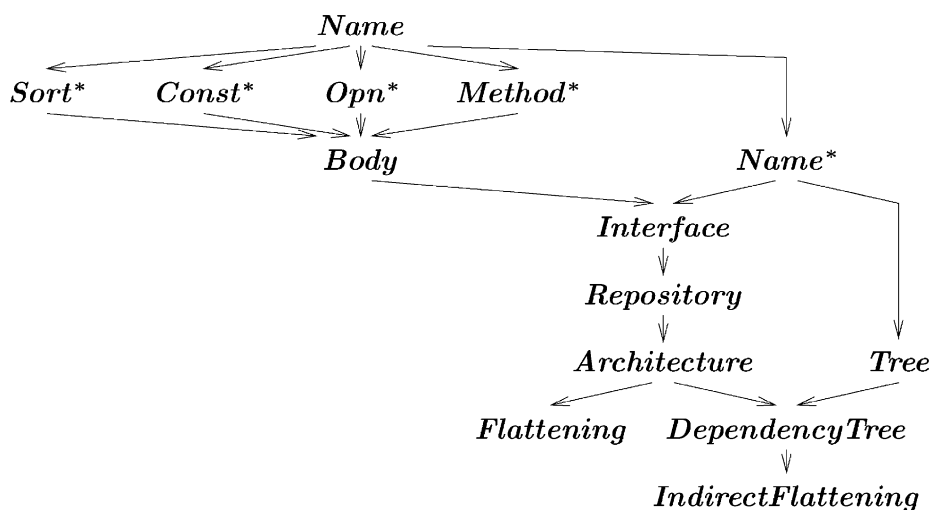


Fig. 5. Architecture of the specification of the method IDL.

void as we did for commands in Section 4:

$$I :: m : world \times oid \times s(1) \times \cdots \times s(k) \rightarrow world;$$

and we translate those methods that are declared to return a non-void value as we did for queries in Section 4, but we also return a (possibly) altered state:

$$\begin{aligned} I :: m_d : world \times oid \times s(1) \times \cdots \times s(k) &\rightarrow s \\ I :: m_s : world \times oid \times s(1) \times \cdots \times s(k) &\rightarrow world \end{aligned}$$

This produces a signature; the semantical interpretation of which, is the semantics of the interface.

5.2. An abstract data type IDL

We can produce an IDL model for abstract data types by creating the set of all *named importing signatures* as an IDL. The body of the IDL is just the usual mathematical notion of a signature, i.e., a collection of sorts, constants and operations:

```
interface    I
import      ..., J, ...
sorts       ..., s, ...
constants   ..., c : -> s, ...
operations  ..., f: s(1)* ... * s(l) -> s, ...
endinterface
```

Thus, the specification **Body** of well-formed bodies for named importing signatures is a natural adaptation of a reduct of the specification of bodies for interfaces of Section 3.3.2, with structure $R(\mathbf{Sort}^*, \mathbf{Const}^*, \mathbf{Opn}^*)$. We specify the language of named importing signatures by

```
specification Body
imports      Sort*, Const*, Opn*
sorts        Body
constants    null : -> Body
operations   bdy : Sort* x Const* x Opn* -> Body
               sorts : Body -> Sort*
               const : Body -> Const*
               opns : Body -> Opn*
               join : Body x Body -> Body
               tag : Name* x Body -> Body
```

equations

$$\begin{aligned}
& \text{sorts}(\text{bdy}(S, C, O)) = S \\
& \text{consts}(\text{bdy}(S, C, O)) = C \\
& \text{opns}(\text{bdy}(S, C, O)) = O \\
& \text{null} = \text{bdy}(\text{null}_{\text{sorts}}, \text{null}_{\text{consts}}, \text{null}_{\text{opns}}) \\
& \text{join}(B_1, B_2) = \text{bdy}(\text{join}_{\text{sorts}}(\text{sorts}(B_1), \text{sorts}(B_2)), \\
& \quad \text{join}_{\text{consts}}(\text{consts}(B_1), \text{consts}(B_2)), \\
& \quad \text{join}_{\text{opns}}(\text{opns}(B_1), \text{opns}(B_2))) \\
& \text{tag}(N, B) = \text{bdy}(\text{tag}_{\text{sorts}}(N, \text{sorts}(B)), \\
& \quad \text{tag}_{\text{consts}}(N, \text{consts}(B)), \\
& \quad \text{tag}_{\text{opns}}(N, \text{opns}(B)))
\end{aligned}$$

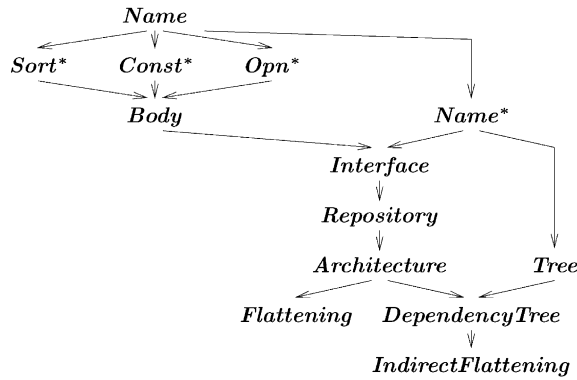


Fig. 6. Architecture of the specification of the abstract data type IDL.

The architecture of the specification of the IDL is shown in Fig. 6: we have a specification constructed from 12 separate component specifications.

5.2.1. Semantics

To define the semantics of an architecture of named importing signatures we flatten the architecture. If this produces a stand-alone signature named N , then we define its semantics to be some algebra of that signature (on removing the name from N).

6. Concluding remarks

Specifically, in this paper, we have

- (i) considered some problems, concepts and applications for a theory of interfaces (Section 2);
- (ii) given an axiomatic analysis of interface architectures by means of an algebraic specification of the abstract syntax of a language for general interfaces and defined a general flattening transformation by means of structural induction (Section 3); and
- (iii) presented some examples of object-oriented interfaces and their structure by means of IDLs (Sections 4 and 5).

To conclude, we suggest four directions for further work.

6.1. Insight into practical IDLs

Notions of interface, and their properties, are very varied. Our short account of practical developments in Section 2 emphasises some relevant interests in object-oriented programming and design. Even there, the word interface is hard to pin down. In fact, it is *not* practical to model *faithfully* “real” IDLs for at least three different reasons, namely:

- legacy issues (for example, COM’s use of C pre-processors and bit-level operations),
- political compromises (for example, the OMG policy of trying to merge alternative standard proposals), and
- non-essential features (for example, “context expressions” in CORBA).

These effect the features that make up “real” IDL standards and lead to unwieldy models which obscure our picture of the *core* conceptual abstractions. The core concepts, together with a proper understanding of their scope and limits, can lead to new insights into practical IDLs. One is reminded of the many notions of data type, and their properties, used in the decade 1968–1978 (see, for example, [22]). They were the raw material for the theory of abstract data types, whose core concepts have had a profound influence on programming.

In short, languages in practical use are complicated, and evolve further in the hands of different vendors and user groups. Theory can choose aspects of practical IDLs to reflect on, model and analyse independently. In this rich and dynamic area there are many new core abstractions to be identified and studied.

6.2. Evolving a framework

What all the practical standards have in common amounts to almost nothing. From their motivation and intentions to their detailed constructs, practical languages (such as Eiffel and Java) and systems (such as COM and CORBA) are different and the style of their interfaces reflect these differences. Therefore, in considering the idea of interface

we are *not* looking for a *universal model* that can unify current practice. Nor are we proposing a (historically, politically or scientifically) *correct model* for practice.

Rather than presenting a unifying model of IDLs, we introduce a simple framework based on precise concepts and techniques that can be used to model *some* aspects of *some* specific IDLs. The abstract model we discuss is simple, almost trivial, from the point of view practical standards. But it is also easily understood, and can be built upon in disparate ways.

The IDLs present in the various methods and tools of software engineering require adaptations of our general model. The simple notion of importing can be extended with variants such as adding the concepts of

- (i) visibility (export, hide),
- (ii) data-only importing,
- (iii) inheritance, and
- (iv) genericity

by choosing different forms of *join* and extending the simple notion of importing in the current model. The simple notion of repository can be extended with richer indexing systems. For example, new tagging operations on names can give the repository a tree structure.

The extensive literature on abstract data types and wide-spectrum algebraic specification languages provides a rich source of motivation for other concrete models of IDLs and extensions to our general framework. Examples of sources include [2,15,18,25,3,29].

6.3. Formalisation

The problem of analysing the concept of interface is challenging and timely. Our curiosity about interfaces originates in thinking about simple languages for writing algebraic specifications, and has been stimulated recently by trying to understand current theories of object-oriented design methods.

Our algebraic approach here was inspired by the seminal paper on module algebra [2]. This provides a first axiomatisation of modules, guided by the case of algebraic specifications. Our paper addresses interfaces, removes some technical ideas from module algebra, and adds the following:

- (i) a notion of unique interface identity;
- (ii) a treatment of dependency management, using the ideas of repositories and dependency trees, and their properties.

Interface identity is a key concept in object oriented programming languages, and dependency a key concept for components as units of deployment [40].

Some further import constructs need to be added and analysed to enrich the dependency, and the analysis of further operators on interfaces would be useful. Developing other abstract approaches to interfaces (e.g., using (i) module algebra and (ii) graphs) are obvious tasks.

The simple model in Section 3 applies to interfaces found in different stages of the software process. A problem is: Develop a hierarchical theory of interfaces and use it to analyse system architectures at different levels of abstraction. The questions concern correctness between architectures for the analysis, modelling, design and programming of systems.

Let us observe that flattening is intimately connected to assembling. The assumptions we have made for interfaces, and hence the axiomatic specifications proposed, are quite general. They might apply to unexpected situations where systems are made from units or components that are indexed and kept in a library. The connection between interfaces and implementations needs to be analysed to establish the scope of these formalisations.

6.4. Tools

Plenty of software tools already exist for producing interfaces from models and some exist for producing interfaces from code. Some systems also provide packaging tools that determine which compiled Java classes and interfaces need to be deployed together in a Java archive file. The tools are more specific than the languages they service. So, in making tools for the abstract models, our goal is still to understand the core concepts and properties present in IDLs.

A prototype of our specification has been developed in Prolog which has been used to test the axiomatisation. A next step is to use the general model of Section 3 to design a metalanguage for the definition of IDL syntax and equip it with general flattening and other basic syntax processing tools.

However consider a few of the many levels at which interfaces exist.

1. We see the concept of interface as one subsumed by the notion of class. For example, the Eiffel notion of visibility leads to an interface composed of “subinterfaces”—each of which has a different visibility. (This is explored in [33]).
2. IDLs are used for API standards. Such API standards allow developers to depend on standard interfaces rather than definitions or classes supported only by one vendor. The W3C consortium has used an IDL to specify a standard for accessing XML syntax trees, and Sun and IBM both have DOM-compliant Java parsers for XML documents.
3. Interfaces are often used in design where a class is expected, but no concrete class is specified—only a named list of attributes and operations. For example, see [26] on the Interface Segregation Principle, and [8] on reusable domain objects.

Can tools for these kind of applications be suggested or even derived from tools for the theoretical models? This is connected to the problems, mentioned above, of discovering core abstractions and mapping aspects of programming languages and IDLs to them.

Appendix

An electronic appendix containing a Prolog implementation of our specification can be accessed from ScienceDirect at doi:[10.1016/j.scico.2003.04.001](https://doi.org/10.1016/j.scico.2003.04.001).

References

- [1] J.A. Bergstra, J. Heering, P. Klint, *Algebraic Specification*, ACM Press, New York, 1989.
- [2] J.A. Bergstra, P. Heering, P. Klint, Module algebra, *J. ACM* 37 (2) (1990) 335–372.
- [3] M. Bidoit, D. Sannella, A. Tarlecki, Architectural specifications in CASL, in: A.M. Haerberer (Ed.), *Proc. 7th Int. Conf. on Algebraic Methodology and Software Technology*, Lecture Notes in Computer Science, vol. 1548, Springer, Berlin, 1999, pp. 341–357.
- [4] A.D. Birrel, B.J. Nelson, Implementing remote procedure calls, *ACM Trans. Comput. Systems* 2 (1) (1984) 39–59.
- [5] G. Booch, J. Rumbaugh, I. Jacobson, *The Unified Modelling Language User Guide*, Addison-Wesley, Reading, MA, 1998.
- [6] K. Brockshmidt, *Inside OLE*, 2nd Edition, Microsoft Press, Redmond, WA, 1995.
- [7] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal, *Pattern Oriented Software Architecture: A System of Patterns*, Wiley, New York, 1996.
- [8] F. Civello, Rooted class diagrams: a notation for context-independent models, *J. Object-Oriented Programming* 2 (11) (1998) 55–65, 69.
- [9] M. Clavel, S. Eker, P. Lincoln, J. Meseguer, Principles of Maude, in: J. Meseguer (Ed.), *Proc. 1st Internat. Workshop on Rewriting Logic*, *Electron. Notes Theoret. Comput. Sci.*, vol. 4, Elsevier, 1996, pp. 65–89.
- [10] S. Cook, J. Daniels, *Designing Object Systems: Object-Oriented Modelling with Syntropy*, Prentice-Hall, Englewood Cliffs, NJ, 1994.
- [11] B.J. Cox, A.J. Novobilski, *Object-Oriented Programming: An Evolutionary Approach*, 2nd Edition, Addison-Wesley, Reading, MA, 1991.
- [12] O.-J. Dahl, K. Nygaard, Simula: an ALGOL-based simulation language, *Comm. ACM* 9 (9) (1966) 671–678.
- [13] P.L. Deutsch, Design reuse and frameworks in the smalltalk-80 programming system, in: T.J. Biggerstaff, A.J. Perlis (Eds.), *Software Reusability I: Concepts and Models*, ACM Press, New York, 1989.
- [14] D.F. D’Souza, A.C. Wills, *Objects, Components and Frameworks with UML: The Catalysis Approach*, Addison-Wesley, Reading, MA, 1998.
- [15] H. Ehrig, B. Mahr, Fundamentals of algebraic specification 2: module specifications and constraints, in: *EATCS Monographs on Theoretical Computer Science*, vol. 21, Springer, Berlin, 1990.
- [16] J.A. Goguen, J.W. Thatcher, E.G. Wagner, An initial algebra approach to the specification, correctness, and implementation of abstract data types, in: R.T. Yeh (Ed.), *Current Trends in Programming Methodology*, vol. IV, Data Structuring, Prentice-Hall, New York, 1978, pp. 80–144.
- [17] J.A. Goguen, J.W. Thatcher, E.G. Wagner, J.B. Wright, Initial algebra semantics and continuous algebras, *J. ACM* 24 (1977) 68–95.
- [18] J.A. Goguen, W. Tracz, An implementation-oriented semantics for module composition, in: G. Leavens, D.M. Sitaraman (Eds.), *Foundations of Component-Based Systems*, Cambridge University Press, New York, 2000, pp. 231–263.
- [19] J.A. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, J.-P. Jouannaud, Introducing OBJ, in: J.A. Goguen, G. Malcolm (Eds.), *Software Engineering with OBJ: Algebraic Specification in Action*, Kluwer, Dordrecht, 2000, pp. 3–167.
- [20] A. Goldberg, D. Robson, *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, Reading, MA, 1983.
- [21] J. Gosling, F. Yellin, The Java Team, *The Java Application Programming Interface*, vol. 1 (Core Packages), Addison-Wesley, Reading, MA, 1996.
- [22] D. Gries (Ed.), *Programming Methodology*, Springer, Berlin, 1978.
- [23] K. Heninger-Britton, R.A. Parker, D.L. Parnas, A procedure for defining abstract interfaces for device interface modules, in: *Proc. 5th Int. Conf. on Software Engineering*, 1981, pp. 195–204. (Reprinted in: *Software Fundamentals. Collected Papers by David L. Parnas, D.M. Hoffman, D.M. Weiss* (Eds.), Addison-Wesley, Reading, MA, 2001, pp. 295–314).
- [24] J.W. Klop, Term rewriting systems, in: D. Gabbay, S. Abramsky, T.S.E. Maibaum (Eds.), *Handbook of Logic in Computer Science*, Oxford University Press, Oxford, 1992.

- [25] J. Loeckx, H.-D. Ehrich, M. Wolf, *Specification of Abstract Data Types*, Wiley/Teubner, New York/Stuttgart, 1996.
- [26] R. Martin, The interface segregation principle: one of the many principles of OOP, *The C++ Report*, August 1996.
- [27] J. Meseguer, J.A. Goguen, Initiality, induction and computability, in: M. Nivat (Ed.), *Algebraic Methods in Semantics*, Cambridge University Press, Cambridge, 1985, pp. 459–541 (Chap. 14).
- [28] R. Meunier, The pipes and filters architecture, in: J.O. Coplien, D. Schmidt (Eds.), *Pattern Languages of Program Design*, vol. I, Addison-Wesley, Reading, MA, 1995, pp. 427–440 (Chap. 22).
- [29] P. Mosses, CASL: a guided tour of its design, in: J.L. Fiadeiro (Ed.), *Proc. 7th Int. Workshop on Recent Trends in Algebraic Development Tools*, Lecture Notes in Computer Science, vol. 1589, Springer, Berlin, 1999, pp. 216–240.
- [30] OMG, Common object request broker: architecture and specification, Technical Report 99-10-08, Revision 2.3.1, The Object Management Group, Framingham, Massachusetts, October 1999. Available at: <http://www.omg.org/library/specindx.html>.
- [31] D.L. Parnas, On the criteria to be used in decomposing systems into modules, *Comm. ACM* 15 (15) (1972) 1053–1058.
- [32] D.E. Pery, A.L. Wolf, Foundations for the study of software architecture, *Software Eng. Notes* 17 (4) (1992) 40–52.
- [33] D.L.L. Rees, A theory of software interfaces, Ph.D. Thesis, Department of Computer Science, University of Wales Swansea, 2001.
- [34] D. Riehle, W. Siberski, C. Lilienthal, D. Megert, K.-H. Sylla, H. Züllighoven, Values in object systems, Technical Report 98.10.1, UBI Lab, UBS AG, Zurich, Switzerland, 1998 (Available from <http://www.riehle.org>).
- [35] T. Rus, Σ S-algebra of a formal language, *Soc. Sci. Math. République Socialiste Roumaine Bull. Math.* 15 (2) (1971) 227–235.
- [36] M. Shaw, Procedure calls are the assembly language of system interconnection: connectors deserve first-class status, in: D.A. Lamb (Ed.), *Proc. Workshop on Studies in Software Design*, Lecture Notes in Computer Science, vol. 1078, Springer, Berlin, May 1996, pp. 17–32.
- [37] M. Shaw, D. Garlan, *Software Architecture: Perspectives on an Emerging Discipline*, Prentice-Hall, Englewood Cliffs, NJ, 1996.
- [38] R. Snodgrass, *The Interface Description Language*, Computer Science Press, Rockville, MD, 1989.
- [39] C. Szyperski, *Component Software: Beyond Object-Oriented Programming*, Addison-Wesley, Reading, MA, 1997.
- [40] C. Szyperski, C. Pfister, Component-oriented programming, in: M. Mühlhäuser (Ed.), *Special Issues in Object-Oriented Programming: Workshop Reader of the 10th European Conf. on Object-Oriented Programming ECOOP '96*, dpunkt.verlag, Heidelberg, 1997, pp. 127–130.
- [41] A. van Deursen, J. Heering, P. Klint (Eds.), *Language Prototyping: An Algebraic Specification Approach*, vol. 5, AMAST Series in Computing, World Scientific Publishing, Singapore, 1996.
- [42] G. Van Rossum, *Python Reference Manual*, Release 1.5.2, 1999. Also available from www.python.org.
- [43] K. Walden, J.-M. Nerson, *Seamless Object-Oriented Software Architecture: Analysis and Design of Reliable Systems*, Prentice-Hall, Englewood Cliffs, NJ, 1994.
- [44] M. Wirsing, Algebraic specification, in: A. van Leeuwen (Ed.), *The Handbook of Theoretical Computer Science*, vol. B, Elsevier, Amsterdam, 1990, pp. 675–788.
- [45] W.A. Wulf, Abstract data types: a retrospective and prospective view, in: P. Dembinski (Ed.), *Proc. 9th Symp. on the Mathematical Foundations of Computer Science*, Lecture Notes in Computer Science, vol. 88, Springer, Berlin, 1980, pp. 95–112.