X11 : An Example in API Specification

Issue 1.0   (September 1993)

Author: Robert Andrews

Notice to Readers

TDF is a portability technology and an architecture neutral format for expressing software applications which was developed by the United Kingdom's Defence Research Agency (DRA). DRA has demonstrated that the TDF technology can support ANSI C on MIPS®, Intel 386™, VAX™, SPARC™ and Motorola® 680x0.

Requests for information about TDF should be directed to:

Dr. N E Peeling
Defence Research Agency
St. Andrews Road
Malvern
Worcestershire
United Kingdom WR14 3PS

Tel. +44 684 895314
Fax +44 684 894303

Internet peeling%hermes.mod.uk*relay*.mod.uk

March 1993

Preface

Document details :

- Title : X11 : An Example in API Specification
- Author : Robert Andrews
- Document version : Issue 1.0
- Date : September 1993

This is the first issue of this document. It describes work carried out at DRA by the author, David Hutchinson, Sally Rees and John MacCallum.

1 APIs and Portability

1.1 The Role of APIs in TDF

As was explained in [1], the TDF technology is designed to be an Architecture Neutral Distribution Format; that is to say, a form in which programs can be expressed and which may subsequently be compiled to any of a number of different target machines. At the simplest level this means that the C → TDF compiler (or producer) cannot make any assumptions about the target machine. It does not know, for example, the signedness of **`char`**, the number of bits in an **`int`**, or how types are laid out within structures. Instead it must represent these values abstractly. It is then up to the TDF installer on each target machine to substitute the actual local values for these abstractions.

There is however another set of target dependent values which the producer needs to represent abstractly, namely those which represent the interface between the program and the system libraries on the target machine. For example, the ANSI standard says that on any compliant machine there is a type called **`FILE`**, but that its precise details are implementation dependent. Thus the producer needs to know to represent **`FILE`** by an abstract value (actually a TDF token) which each installer can substitute by the particular value of **`FILE`** on that target machine. In order for this to be possible, the interface between the program and the target machine - the Application Programming Interface (API) - needs to be described to the producer in abstract terms. This description needs to be at the same level as the API specification. For example, it needs to be able to just express that "there is a type called **`FILE`**". In order to express this information, the producer extends the C syntax by using certain **`#pragma token`** statements (see [3] for details). For example, the fact that **`FILE`** is a type would be expressed in the form :

```
#pragma token TYPE FILE #
```

Therefore the role of the API is far more important, and far more explicit, in the TDF compilation system than with conventional compilers. The latter work directly with the API implementation in the form of system headers and system libraries, whereas the former has this abstract intermediate layer of **`#pragma token`** statements which reflects the actual API specification. This means that to compile a program using the TDF system is to effectively check the syntactic aspects of it against the abstract paper specification of the API, rather than a particular implementation (see [1]). Thus it provides a method of effectively API-checking these aspects of the program.

There is another type of API checking within the TDF system. The producer uses abstract values to represent objects within the API, but each individual installer needs to provide the actual local values of these objects. These local values are stored as TDF in a TDF library describing the API implementation. The construction of this library consists of comparing the abstract API specification, in the form of the **`#pragma token`** statements, with the local API implementation, as described in the system headers. Thus this process provides a syntactic check of the API implementation against the specification.

1.2 Describing APIs

Thus for the TDF system to effectively check a program against its API, this API must have been described in terms of the **`#pragma token`** syntax. Thus there has been a programme of API description at DRA designed to provide such descriptions of standard APIs such as ANSI, POSIX, XPG3/4, SVID3 etc.

It would be possible to describe APIs directly using **`#pragma token`** statements, however there are a number of drawbacks to this approach. Firstly, although the **`#pragma token`** syntax is extremely powerful and is capable of expressing extremely complex constructs, this also means that the syntax is necessarily complex and unfamiliar. The vast majority of constructs which APIs use are very simple and C-like, and so can be expressed using only a small subset of the **`#pragma token`** syntax. The second drawback to this approach is that there are no internal checks applied to the input, so that misprints and incomplete specifications can only be detected by extensive testing.

1.2.1 The tspec tool

It was with these considerations in mind that a tool, called *tspec*, was developed at DRA to aid in API description. *tspec* allows the API constructs to be entered in a form closer to C, which it then translates into the corresponding **`#pragma token`** statements (see [2] for details on *tspec*). During this translation process it checks not only for syntax errors, but also for the use of undeclared types, namespace clashes etc. These checks are applied not only to individual headers within the API, but globally to the entire API. Note that *tspec* is not intended to be an abstract specification language along the lines of Obscure or Clear, it is merely a tool which has been found useful for doing a specific task : translating paper API specifications into a form understood by the C $\rightarrow$ TDF producer.

As an example of *tspec* input, consider the following extract from the specification of the ANSI ***stdio.h*** header :

```
+TYPE FILE ;
+EXP FILE *stdin, *stdout, *stderr ;
+FUNC FILE *fopen ( const char *, const char * ) ;
+FUNC int fclose ( FILE * ) ;
```

This expresses that **`FILE`** is a type, **`stdin`**, **`stdout`** and **`stderr`** are expressions of type **`FILE *`**, and that **`fopen`** and **`fclose`** are functions with the given prototypes.

Many of the examples given in this paper will use the *tspec* notation, so it is worth familiarising oneself with it. Hopefully this should be fairly simple for anyone familiar with C.

1.2.2 API Description Guidelines

Essentially the process of describing an API specification to *tspec* consists of ploughing through the paper specification transcribing its descriptions of the objects comprising the API into their *tspec* forms. If the specification is available in machine readable form then parts of this can be partially automated, particular since the C-like terms used in the specification are similar to those used by *tspec*. This is particularly useful when the specification lists a large number of similar objects, such as the list of strings given in Appendix E of the Xt specification.

It is worth briefly discussing some of the general considerations involved in the API description process before moving on to consider the specific example of X11 (Release 5). The first consideration when starting an API description is how this API relates to those previously described. For example, XPG3 is an extension of POSIX, so the structure of the description should reflect this. It would be very easy to write an shell script which, for each header, ***x.h***, in the POSIX API, set up a header of the form :

```
+IMPLEMENT "posix", "x.h" ;
# Add xpg3 specific stuff here
```

to act as a skeleton for the XPG3 API. Objects which are in XPG3 but not in POSIX could then be added, as appropriate, either to these extension headers, or to XPG3-specific headers, which may be created as required.

The question of which headers comprise an API, and which objects are specified in which headers, is another important one. Many APIs list their constituent headers at a conveniently early stage in the specification, and go to great lengths to describe where objects are declared. Others are vaguer on these matters, and the person performing the API description is left to try and work out a suitable logical structure. The *tspec* ***MASTER*** file lists all the headers comprising the API, and needs to be kept up to date as new headers are added.

It is also important to remember that the API description is meant to accurately reflect the API specification. Thus it is important to try to write down exactly what the specification says, and not to be unduly influenced by one’s knowledge of particular implementations. Probably the only legitimate reason to consult the implementation(s) during the description process proper is to help clarify the meaning of the specification when this is not clear. There will be a chance to review the finished API description against particular API implementations later, during the TDF library building process. Recall that this consists of comparing the API description with the API implementation. Although this was described above as being a check for the implementation, it may also be viewed as a check for the API description. For example, many mistypings in the API description will be picked up by *tspec*, but others will be detected during the library build. For example, if I had typed :

```
+TYPE FILE ;
+EXP PILE *stdin, *stdout, *stderr ;
```

in the example above, *tspec* would complain that the type **`PILE`** has not been declared. However if I had typed :

```
+TYPE FILE ;
+EXP FILE *stdon, *stdout, *stderr ;
```

tspec would not detect the error. However the object **`stdon`** would be left undefined in the library build since the implementation would hopefully have spelt it correctly. What neither process can detect at present would be if I had omitted **`stdin`** altogether, so it is important to try to avoid errors of this kind. This may however be possible in future. The API specification describes a space of all permitted names, and it should be possible to detect where the implementation steps outside this space. If I had omitted **`stdin`** from the specification then it will not be in this space, so its use by the implementation would be detected.

Where it is possible to detect such omissions is when using the generated **`#pragma token`** headers to compile an application which, for example, uses **`stdin`**. However any application will use only a small number of API constructs, so this is not a particularly effective way of detecting omissions.

1.3 The X11 (Release 5) API

The main purpose of this paper is to illustrate some of the concepts involved in API specification in relation to work carried out at DRA during September 1993 in describing the X11 (Release 5) API. Of course, X11 is an extremely important API, but from the point of view of this paper it is also an "interesting" API. Most of the APIs described previously tend to be rather similar - once you have described POSIX, SVID3 is unlikely to provide many surprises. However X11 raised a number of interesting issues, probably more than any API since the first to be described, ANSI (back in 1991).

There are a number of possible reasons for this, which will be discussed during the course of the paper.

- The X11 specification is looser than the more formal specifications issued by, for example, X/Open.
- It is also describing not one, but a hierarchy of APIs. Not only is there a split between, for example, Xlib and Xt, but also between "public" and "private" headers within Xt.
- In contrast to, say, POSIX, where there is a wide variety of implementations on different machines, most X11 implementations tend to be based on the standard MIT implementation, and so tend to be very similar. Thus there is less of a distinction made between specification and implementation in this case. If an application works with the MIT implementation, the programmer is less likely to

worry about other possible implementations. Because the TDF system is concerned with making the split between the specification and the implementation explicit, this attitude can cause problems. It could be said that the API the programmer is writing to is not X11, but some extension, MIT-X11.

Finally in this introductory section it is worth mentioning the point of view from which this paper is written. It is based on work carried out by the author based on preliminary studies by a number of colleagues. Neither I nor they are experts on X11. Our knowledge of it comes from reading the specification and from a study of how certain public domain X11 applications are written. For the purposes of describing the specification this means that we have not been unduly influenced by our knowledge of particular implementations, but have tried to reflect what is actually written.

2 Describing X11 - Preliminaries

As was mentioned in section 1.2.2, it is important to determine where an API fits into the hierarchy of previously specified APIs before attempting to describe it. X11 is not an extension of an existing API in the way that XPG3 is an extension of POSIX. Instead it is a bolt-on API which can be used in conjunction with a base API such as POSIX or XPG3. However, as mentioned above, X11 is not a single API, but itself consists of a number of sub-APIs.

The simplest division is that reflected in the documentation, with the hierarchy from the basic X Library (Xlib) to the X Toolkit (Xt), the Miscellaneous Utilities (Xmu) and the Athena Widgets (Xaw). These are arranged in a very simple progression :

$$\text{Xlib} \subseteq \text{Xt} \subseteq \text{Xmu} \subseteq \text{Xaw}$$

But there is also a hierarchy within, for example, Xt, given by the "public" and "private" headers. Normal application writers are intended to use only the public headers, such as ***X11/Intrinsic.h***, whereas people defining their own widgets may use the private headers, such as ***X11/IntrinsicP.h***. In the former the type **`Widget`**, for example, is a pointer to an opaque structure, whereas in the latter this structure is less opaque and certain fields have become visible. Thus we may think of the private headers as being an extension API of the public headers. From this point of view the term "private" is misleading, so we shall use the term P-headers to refer to these files.

A decision was made to specify Xlib, Xt, Xmu and Xaw as separate *tspec* APIs called ***xlib5***, ***xt5***, ***xmu5*** and ***xaw5***, but not to separate off the P-headers into separate APIs. Instead each P-header is booby-trapped with lines of the form :

```
#ifndef __X11_P_HEADERS
#error Unauthorized access to X11 P-headers
#endif
```

The macro **`__X11_P_HEADERS`** may be effectively hidden from the user by means of *tcc* environments (see [4]). In addition to the basic environments, **xlib5**, **xt5**, **xmu5** and **xaw5**, there is, for example, an environment **xt5p**, consisting of :

```
+FLAG "-Yxt5"
+FLAG "-D__X11_P_HEADERS"
```

The normal Xt application writer would then pass the flag **-Yxt5** to *tcc*, whereas the widget definer would use **-Yxt5p**. In this way the API requirements of a program may be made absolutely explicit.

3 Describing Xlib

Let us illustrate the description process by giving a detailed account of the description of the first couple of chapters of the Xlib specification. This is best understood by reading it in parallel with the Xlib specification, [5].

3.1 Chapter 1 of Xlib

Chapter 1 is primarily introductory in character, but contains the specification of some important objects. Section 1.1 is intended to be an overview, and most of the objects it mentions (printed in **bold**) are properly specified elsewhere. For example, referring to the index shows that `XSync` is specified on page 184. However the types `Window`, `Font` etc. are explicitly mentioned here for the one and only time. They are all synonyms for the same integral resource ID type. Skipping ahead to Section 1.4 reveals that this type is called `XID`, so we can specify our first objects in the header ***X11/X.h*** :

```
+TYPE ( int ) XID ;
+TYPEDEF XID Window ;
+TYPEDEF XID Font ;
+TYPEDEF XID Pixmap ;
+TYPEDEF XID Colormap ;
+TYPEDEF XID Cursor ;
+TYPEDEF XID GContext ;
```

Moving on to Section 1.2, the integral type `Status` is specified. Exactly which header it is defined in is not given - it is probably ***X11/X.h*** again. So we have :

```
+TYPE ( int ) Status ;
```

Section 1.3 contains the first chunk of substantial information, with a list of the headers comprising the Xlib API. The form of wording used, that "the following include files are part of the Xlib standard", seems to allow for implementations to add their own headers. However these would be implementation dependent, and so form no part of a description of the API in its most general form. Thus we can set up these thirteen headers and list them in the *tspec* ***MASTER*** file.

There are also some objects specified, and some relations between the headers spelt out in this section. For example, in ***X11/Xlib.h*** we have :

```
+USE "xlib5", "X11/X.h" ;
+DEFINE XlibSpecificationRelease 5 ;
```

The information on the kind of object specified in each header is also useful, and should be used as a basis for deciding where various objects should be put.

The header ***X11/keysym.h*** is completely specified in this section to be :

```
%%
#define XK_MISCELLANY
#define XK_LATIN1
#define XK_LATIN2
#define XK_LATIN3
```

```
#define XK_LATIN4
#define XK_GREEK
%%
+USE "xlib5", "X11/keysymdef.h" ;
```

however the contents of ***X11/keysymdef.h*** are never precisely specified within the Xlib specification. The actual **KeySym** values are standard (Latin-1 is ISO8859-1 etc.), but the actual **XK_** macros representing them are not given. The decision reached on how to handle this gap in the specification was to copy the names of the **KeySym**'s from the implementation, but to abstract their values by making them tokenised objects. Thus the *tspec* specification of ***X11/keysymdef.h*** has the form :

```
+IFDEF XK_MISCELLANY
+CONST KeySym XK_BackSpace, XK_Tab, XK_LineFeed ;
....
+ENDIF
....
```

Moving on, Section 1.4 specifies a number of generic values and types, including the generic resource ID type **XID** already mentioned. Presumedly these are meant to be defined in ***X11/X.h***. We have :

```
+TYPE ( int ) Bool ;
+CONST Bool True, False ;
+DEFINE None 0 ;
+TYPEDEF char *XPointer ;
```

Actually there is a bit of reading between the lines here. It is not actually stated that **Bool** is an integral type, but it is not a particularly radical assumption. The translation of "universal null resource ID or atom" to **0** is also probably acceptable.

The rest of Chapter 1 is concerned with the conventions used in Xlib and contains no further specification items.

3.2 Chapter 2 of Xlib

The real specification begins with the display functions in Chapter 2. We are immediately faced with a dilemma, since these functions are declared in the "traditional" style :

```
Display *XOpenDisplay ( display_name )
char *display_name ;
```

rather than as a prototype :

```
Display *XOpenDisplay ( char *display_name )
```

Obviously, since TDF is concerned with API checking, we wish to express the function argument information to allow the arguments to be type checked. This means that we want to convert all the function declarations to prototype forms. This presents no difficulties in the case above, since the two forms are exactly equivalent from the point of view of the C calling conventions. However this is not always necessarily the case. We have ignored this possibility for the time being and directly transformed all the function declarations into prototypes (this is probably justified by the second point in Section 1.8), however to exactly represent what is in the specification we could use the TDF producer's weak prototypes (see [3]).

The contents of Section 2.1 can now be summarised by adding :

```
+TYPE ( struct ) Display ;
+TYPE ( struct ) Screen ;
```

```
+FUNC Display *XOpenDisplay ( char * ) ;
```

to ***X11/Xlib.h***. We have had to reverse the order of specification of the function **`XOpenDisplay`** and the type **`Display`** so that the type is declared before it is used. Note that **`Display`** and **`Screen`** have been specified to be opaque structure types, thereby enforcing the use of the information macros and functions to access elements of these structures.

The display macros and functions in Section 2.2.1 are easily translated into :

```
+EXP unsigned long AllPlanes ;
+FUNC unsigned long XAllPlanes ( void ) ;
+MACRO unsigned long BlackPixel ( Display *, int ) ;
+FUNC unsigned long XBlackPixel ( Display *, int ) ;
+MACRO unsigned long WhitePixel ( Display *, int ) ;
+FUNC unsigned long XWhitePixel ( Display *, int ) ;
....
```

Why both versions are considered necessary is not entirely clear. In most APIs, functions may be implemented by macros, but there is an underlying function with the same effect which may be accessed by **`#undef`**-ing the macro. This is the meaning of the *tspec* **`+FUNC`** construct. X11's use of explicit macro forms seems superfluous.

The start of Section 2.2.2 presents another problem. A specification of the type **`XPixmapFormat-Values`** is given as follows : "The **`XPixmapFormatValues`** structure provides an interface to the pixmap format information that is returned at the time of a connection setup. It contains :

```
typedef struct {
      int depth ;
      int bits_per_pixel ;
      int scanline_pad ;
} XPixmapFormatValues ;
```

(page 14)". It is not clear from this whether **`XPixmapFormatValues`** is precisely the given structure, or whether it contains the given fields (but not necessarily in the given order and not necessarily comprising the entire structure). It is the word "contains" which troubles me. Both forms may be represented in *tspec*. In the former case, the statement :

```
+FIELD ( struct ) XPixmapFormatValues := {
      int depth ;
      int bits_per_pixel ;
      int scanline_pad ;
} ;
```

should be used, whereas if the definition is not exact the **`:=`** should be omitted. The only difference from the user's point of view is that in the former case the use of static initialisers such as :

```
XPixmapFormatValues v = { 1, 8, 0 } ;
```

is legitimate, whereas in the latter there is no correspondence between the initialising values and the structure fields, so the initialiser is not legitimate. The difference from the API specifier's point of view is that an exact structure can be an evolutionary dead-end. If in some future revision of the specification we wish to add a new field to this structure, then this is not necessarily backwards compatible in the exact structure case, although it is for the incompletely specified structure. The decision made on how to interpret this structure definition was that, since X11's form of specifying structures looks exact, the exact form is assumed unless otherwise stated in the specification.

The rest of Section 2.2 presents few difficulties. Starting with Section 2.2.2 it contains :

```
+FUNC XPixmapFormatValues ( Display *, int * ) ;
```

```
+MACRO int ImageByteOrder ( Display * ) ;
+FUNC int XImageByteOrder ( Display * ) ;
+CONST int LSBFirst, MSBFirst ;
+MACRO int BitmapUnit ( Display * ) ;
+FUNC int XBitmapUnit ( Display * ) ;
....
```

Note that the return values **`LSBFirst`** and **`MSBFirst`** for **`ImageByteOrder`** have been given. It would be easy to miss the specification of these values.

Section 2.3 raises another issue. The function **`XNoOp`** takes a **`Display *`** argument, but its result type is not specified. There are two possible ways of looking at this. The C rules state that **`XNoOp`** is assumed to return **`int`**. However any value returned is meaningless, so to stop it being used it can be argued that the function should be declared to return **`void`**. Whereas the former is technically correct, the latter is more useful from the point of view of API checking. Thus the latter solution has been used. When this is done the rest of Chapter 2 is straightforward. The *tspec* description begins :

```
+FUNC void XNoOp ( Display * ) ;
+FUNC void XFree ( void * ) ;
....
```

3.3 Chapter 3 of Xlib

Chapter 3 presents few new issues. Note however that the mask value macros **`CWBackPixmap`** etc. are explicitly defined, so it is necessary to directly quote these values :

```
%%
#define CWBackPixmap (1L<<0)
#define CWBackPixel (1L<<1)
#define CWBackBorderPixmap (1L<<2)
....
%%
```

Precisely why the exact values of these macros needs to be given in the specification is not clear. All that is really necessary is to specify that they exist as **`long`** constants :

```
+CONST long CWBackPixmap, CWBackPixel, .... ;
```

their precise values being implementation dependent. As with exact structures, exact values are potential blocks to an evolutionary approach to API specification.

The object **`CopyFromParent`** causes a few problems. When it is first mentioned, in Section 3.2, it is used as a default **`Pixmap`** or **`Colormap`**, so the specification :

```
+CONST XID CopyFromParent ;
```

seems appropriate. However in Section 3.3 several of the arguments to **`XCreateWindow`** can be **`Cop-yFromParent`**. These have types **`int`**, **`unsigned int`** and **`Visual *`**. So **`CopyFromParent`** can be an integer or a pointer without an explicit cast. Therefore the specification :

```
+DEFINE CopyFromParent 0 ;
```

seems inevitable.

The rest of the description process is similar. Some particular issues are worth mentioning :

3.3.1 The Region Type

The type **`Region`** specified in Section 16.5 has been made not, as described, an opaque type, but a pointer to an opaque type. This is because it seems to be universally assumed by application writers that **`Region`** is a pointer. For example, *xterm* has :

```
( Region ) NULL
```

Also, for example, in :

```
void XIntersectRegion ( Region, Region, Region ) ;
```

the third argument is used to return the intersection of the first two. Most types cannot be used to return values in this way, although pointers can.

3.3.2 The XESetFlushGC Function

There is an obvious misprint in Appendix C, where the prototype of **`XESetFlushGC`** should be :

```
int ( *XESetFlushGC ( Display *, int, int (*) () ) () ;
```

i.e. the function argument should be the same type as the function returned.

3.3.3 The X Protocol Types

The status of the types in ***X11/Xproto.h*** is not clear. In Appendix C it is stated that for each of the X requests listed in Appendix A there is a request number, **`X_DoSomething`**, and there may also be a request structure, **`xDoSomethingReq`**, and a reply structure, **`xDoSomethingReply`**. However for which requests these structures exist, and the precise form they take is only given in the X Protocol documentation, not in Xlib. Therefore it was decided to put the constants **`X_DoSomething`** in ***xlib5***, but to reserve the structures for some extension API, ***xproto5***. The **`GetReq`**, **`GetReqExtra`**, **`GetResReq`** and **`GetEmptyReq`** macros also more properly belong to this extension.

3.3.4 The X10 Compatibility Functions

The X10 compatibility functions have been omitted. They are obsolete and should not be used.

3.4 Running tspec on Xlib

Once the description had been entered in this way, *tspec* was be run to convert it into the corresponding **`#pragma token`** statements. This is done by means of the command-line :

```
tspec -u -v xlib5
```

tspec bears the same relationship to the description as a compiler does to a program, so it is not surprising that the first attempt to run *tspec* resulted in a number of errors. Most of these were simple misprints in the description, which are easily corrected. Others involve types being used before they are declared, and were solved by reordering the code. However the following types were detected as never being declared, because they are not properly specified in the specification :

- **`GC`**,
- **`XcmsCCC`**,
- **`XrmHashTable`**,
- **`ScreenFormat`**.

What the specification does have to say about these types seems consistent with the first two being pointers to opaque types, and the latter two being opaque types, so these specifications were added to the description.

3.5 Building the Xlib TDF library

Once the API description has been run through *tspec*, the next test is to compare it with an implementation of that API through the TDF library building process. The model implementation of X11 used for this process was that provided with a pre-release version of System V Release 4.2 running on an ICL SPARC machine. The errors shown up by the library building process are of four types :

- Idiosyncrasies of the library building process,
- Misprints in the specification,
- Errors in the description process,
- Errors and omissions in the implementation.

The first type of error generally arises because the library building process uses the system headers describing the API implementation in a non-standard manner. For example, consider the API specification :

```
+TYPE Complex ;
+MACRO int Real ( Complex ) ;
+MACRO int Imag ( Complex ) ;
```

and the implementation :

```
#define Real( c ) ( ( c ).real )
#define Imag( c ) ( ( c ).imag )
typedef struct {
      int real ;
      int imag ;
} Complex ;
```

For most purposes the macros **`Real`** and **`Imag`** in the implementation are just stored by the preprocessor and expanded when required. However in the library build they are not just macros, they are definitions of the tokens **`Real`** and **`Imag`**. As such they are expanded when they are encountered. But at this stage we only know that **`Complex`** is an opaque type, not that it is the given structure. Thus the definition of **`Real`** will fail with an error to the effect that **`Complex`** does not have a field called **`real`**. To conclude : when it comes to token definitions, the declare before use rule even applies to the body of macro definitions. In the model implementation of X11 the headers ***X11/Xlib.h*** and ***X11/Xcms.h*** need a little hacking around to get round this problem.

A similar problem concerns duplicate macros. Normally a macro may be defined any number of times, provided the definitions are consistent. However if these macros are being used to define tokens they can lead to multiple token definition errors.

Some of the implementation errors shown up by the library building process are technical is nature. For example it does not allow tokens representing types (or field selectors) to be defined by macros. This is because these objects do not occupy the macro namespace (see [3]), so such implementations are illegal. The model implementation defines the types **`Bool`** and **`Status`** by macros, so it is necessary to change these into **`typedef`**'s.

All other errors show up genuine contradictions between the description of the API and the implementation. On the first run through the vast majority of these will be because there has been an error in the description process and the description does not reflect what is actually in the specification. What is more interesting is when the specification and the implementation disagree.

There are only a couple of such disagreements in Xlib. The first class are the undefined token errors. These occur when an object is specified in the *tspec* description of the API, but is not defined in the API implementation. As mentioned above, many of these are due to mistypings in the description. Others are due to the library build looking in the wrong place - if an object is defined in a different header than the one prescribed by the *tspec* description it may be necessary to modify the library build to make it look in the alternative location. This sort of error is particularly common when the specification is vague as to which headers objects are defined in. However some undefined token errors are due to genuine omissions from the implementation. In the Xlib library build the following such omissions were discovered :

- The atoms **XA_WM_COLORMAP_WINDOWS**, **XA_WM_PROTOCOLS** and **XA_WM_STATE** are not defined. These values are best left undefined so that an attempt to install a program which uses them will fail. This is done by adding the following lines to the implementation :

```
#pragma ignore XA_WM_COLORMAP_WINDOWS
#pragma ignore XA_WM_PROTOCOLS
#pragma ignore XA_WM_STATE
```

- The value **XcmsInitFailure** is not defined. A definition of it to be **XcmsInitDefault** (which seems reasonable) was provided.

- The cursor **XC_dot_box_mask** is not defined in the implementation. I suspect that this is because the name of this cursor should be **XC_dotbox**.

- The value **EPERBATCH** is not defined. It is not clear what its value should be, so it was left undefined.

The second class of errors occur when the specification and the implementation both specify an object, but their specifications disagree. A number of such errors occurred with Xlib. The first concerns the type **XcmsColorFormat** which is specified to be **unsigned long** but defined to be **unsigned int** in the implementation. Not a vital error I agree, but one requiring attention. The solution decided upon was to relax the specification slightly by making **XcmsColorFormat** a general integral type in the API description.

The remaining errors detected concern function prototypes. In the specification **XrmCombineFile-Database** returns **void**, whereas in the implementation it returns **Status**. A careful reading of the specification however reveals that it is later assumes that the function does indeed return a status value, so that the **void** return value is a misprint in the specification.

The function **XcmsStoreColors** is declared in the specification to be :

```
Status XcmsStoreColors ( Display *, Colormap, XcmsColor [],
      int, Bool [] ) ;
```

whereas the implementation has :

```
Status XcmsStoreColors ( Display *, Colormap, XcmsColor [],
      unsigned int, Bool [] ) ;
```

This difference is very minor (no programs are likely to break because of it), but I suspect that the implementation is correct. Other similar functions with names beginning with **Xcms** seem to have **unsigned int** for their **ncolors** argument, although those beginning with just **X** seem to have **int**.

The function **XcmsAllocNamedColor** presents a similar difficulty. The specification has :

```
Status XcmsAllocNamedColor ( Display *, Colormap, char *,
      XcmsColorFormat, XcmsColor *, XcmsColor * ) ;
```

whereas the implementation has :

```
Status XcmsAllocNamedColor ( Display *, Colormap, char *,
      XcmsColor *, XcmsColor *, XcmsColorFormat ) ;
```

I suspect that the implementation is correct by analogy with **`XcmsAllocColor`** and other functions, in which the colour format argument comes last.

4 Describing Xt

We now move on to consider the X Toolkit Intrinsics (Xt) as specified in [6]. This will be described in less detail than Xlib.

4.1 Widgets

4.1.1 Hiding Data

The Xt widgets provide any example of how much easier it is to hide data with TDF than with just C. For example, in ***X11/Intrinsic.h***, we can write :

```
+TYPE ( struct ) CoreRec ;
+TYPEDEF CoreRec *CoreWidget ;
```

and in ***X11/IntrinsicP.h*** :

```
+USE "xt5", "X11/Intrinsic.h" ;
+FIELD CoreRec {
      CorePart core ;
} ;
```

So that if ***X11/Intrinsic.h*** is included we know that **`CoreWidget`** is a pointer to the opaque structure type **`CoreRec`**, whereas if ***X11/IntrinsicP.h*** is included we get the additional information that **`CoreRec`** has a field **`core`** of type **`CorePart`**.

In fact, **`CoreRec`** consists of precisely this one field (it does not appear with the non-exact structures, such as **`CorePart`**, listed at the end of Section 1.5). Therefore what we should actually write in ***X11/IntrinsicP.h*** is as follows :

```
+USE "xt5", "X11/Intrinsic.h" ;
+FIELD CoreRec := {
      CorePart core ;
} ;
```

The distinction between this and the previous specification is interesting. In both cases we are extending the API given in ***X11/Intrinsic.h*** by ***X11/IntrinsicP.h***. However in the first case, it is by adding a new object, the field selector **`CoreRec.core`**, whereas in the second case it is by specialising an existing object, **`CoreRec`**, to the structure :

```
struct {
      CorePart core ;
} ;
```

Alternatively expressed : in the first case we are adding a generator, in the second we are adding a relation. *tspec* is designed primarily for adding generators; its support for adding relations is an area which needs development.

Another interesting point is that if I write :

```
#include <X11/Intrinsic.h>
CoreRec c ;
```

then the producer will use the token **`CoreRec`** to represent the type of **`c`**, whereas if I write :

```
#include <X11/IntrinsicP.h>
CoreRec c ;
```

it will use the explicit structure definition. Of course it is important that the two representations are equivalent, that is that the definition of the token **`CoreRec`** is the given structure. But the definition of this token corresponds to the definition of the structure **`CoreRec`** in the implementation, so all we actually require is that the implementation and the description agree as to the definition of **`CoreRec`**. Unfortunately, in this one case, the TDF library building process is not able to check this, because the C rules for structure definitions get in the way. What we effectively have is two definitions for the same structure, which we want checked for consistency. Unfortunately C does not allow for even consistent redefinitions of structures, so it is not possible to check for consistency during the library build.

4.1.2 The Widget Hierarchy

Xt describes a hierarchy of widgets (and widget-like objects), illustrated in Fig. 1. This hierarchy may be extended by defining new widgets. This can be done either by extension APIs, such as Xaw and Motif, or by individual application writers. The hierarchy gets more refined as one moves down the diagram : the set of **`CompositeWidget`**'s is a subset of the set of **`CoreWidget`**'s etc.

The C type system does not allow for compile-time checking of this widget hierarchy. For example, if we had :

```
extern void UseCoreWidget ( CoreWidget ) ;
void UseCompositeWidget ( CompositeWidget w )
{
      UseCoreWidget ( w ) ;
      return ;
}
```

this will fail to compile, because the conversion from **`CompositeWidget`** to **`CoreWidget`** implied by the widget hierarchy cannot be expressed to the compiler. It would of course be possible to put in an explicit cast. But explicit casts can equally well go against the widget hierarchy.

For this reason, the X11 specification and X11 application writers do not explicitly spell out the class of each widget, but instead use the generic type, **`Widget`**. If Fig. 1 is thought of as showing all the legitimate conversions between classes of widgets (by following the lines up), the use of the generic type **`Widget`** effectively breaks the hierarchy. Any class of widget can be converted to, and from, **`Widget`**, therefore it is possible to convert between any two classes of widgets.

4.1.3 Conversion Tokens

TDF actually allows for type hierarchies such as those in Fig. 1 to be expressed. Suppose that we have types **`One`**, **`Two`**, **`Three`**, **`Four`** arranged in the hierarchy shown in Fig. 2. This may be expressed as follows :

```
+TYPE One, Two, Three, Four ;

+MACRO One __Convert21 ( Two ) ;
+MACRO Two __Convert32 ( Three ) ;
+MACRO Two __Convert42 ( Four ) ;
+MACRO One __Convert41 ( Three ) ;
+MACRO One __Convert41 ( Four ) ;
```

```
%%
#ifndef __BUILDING_LIBS
#pragma accept conversion __Convert21
#pragma accept conversion __Convert32
#pragma accept conversion __Convert42
#pragma accept conversion __Convert31
#pragma accept conversion __Convert41
#endif
%%
```

The **`__Convert`** macros express all the legimate conversions allowed by the hierarchy in an explicit fashion. However the **`#pragma accept conversion`** statements make them into implicit conversions. When the C → TDF producer has to convert a value of one type to another type, it firstly checks all the normal C conversion rules and then all the tokens registered using **`#pragma accept conversion`**. For example, in :

```
One x ;
Two y = x ;
```

the conversion token **`__Convert21`** is used to transform **`x`** from a value of type **`One`** to a value of type **`Two`**. In effect the program is transformed to :

```
One x ;
Two y = __Convert21 ( x ) ;
```

As another example, suppose that we have :

```
extern void UseTwo ( Two ) ;
void Use ( One a, Three b )
{
      UseTwo ( b ) ;
      UseTwo ( a ) ;
      return ;
}
```

then the **`Three`** → **`Two`** conversion on line 4 will be allowed (using **`__Convert32`**), but the **`One`** → **`Two`** conversion on line 5 will not.

The **`__Convert`** tokens are implementation-dependent, and must be explicitly defined during the TDF library building process. (Note that the **`#pragma accept conversion`** statements are protected by the **`__BUILDING_LIBS`** macro, which is by convention defined during library building - this is to stop the conversion tokens being defined recursively.) An example implementation might be :

```
typedef struct { int a ; } *One ;
typedef struct { int a, b ; } *Two ;
typedef struct { int a, b, c ; } *Three ;
typedef struct { int a, b, d ; } *Four ;

#define __Convert21( x ) ( ( One ) ( x ) )
#define __Convert32( x ) ( ( Two ) ( x ) )
#define __Convert42( x ) ( ( Two ) ( x ) )
#define __Convert31( x ) ( ( One ) ( x ) )
#define __Convert41( x ) ( ( One ) ( x ) )
```

A point worth making is that it is necessary to express all the possible conversions in this way. The producer cannot handle transitivity in conversion tokens. Transitivity is decidedly an *n^2* problem; for example the 11 basic conversions in Fig. 1 expand to 54 when transitivity is taken into account.

So the TDF extensions to C can express the widget hierarchy "properly". However, since the hierarchy cannot be exploited from C and so is not used by application writers, and is anyway broken by the use of **`Widget`** as a generic type, it is probably not worth the effort.

4.2 The XtArgVal Type

The implementation-specific types described in Section 1.5 of the Xt specification raise a number of issues. **`XtArgVal`** is designed to be a generic type large enough to hold most pointer and integral types, and examining various X11 applications shows that both pointers and integers are converted to **`XtArgVal`**'s without explicit casts. The solution to this is try to write down exactly what is in the specification. **`XtArgVal`** is an opaque type, and for certain integral and pointer types **`t`** the conversions :

```
XtArgVal → t
t → XtArgVal
```

are allowed. This can be done using the conversion tokens mentioned above. The *tspec* description takes the form :

```
+TYPE XtArgVal ;
+TOKEN __MyXtArgValToPtr
      %% PROC { TYPE t, EXP rvalue : XtArgVal : e |\
      EXP e } EXP rvalue : t * : %% ;
+TOKEN __MyPtrToXtArgVal
      %% PROC { TYPE t, EXP rvalue : t * : e |\
      EXP e } EXP rvalue : XtArgVal : %% ;
+TOKEN __MyXtArgValToInt
      %% PROC { VARIETY v, EXP rvalue : XtArgVal : e |\
      EXP e } EXP rvalue : v : %% ;
+TOKEN __MyIntToXtArgVal
      %% PROC { VARIETY v, EXP rvalue : v : e |\
      EXP e } EXP rvalue : XtArgVal : %% ;
%%
#ifndef __BUILDING_LIBS
#pragma accept conversion __MyXtArgValToPtr
#pragma accept conversion __MyPtrToXtArgVal
#pragma accept conversion __MyIntToXtArgVal
#pragma accept conversion __MyXtArgValToInt
#endif
%%
```

Note that this is one of the rare occasions when we need to step outside the *tspec* syntax and directly access the full power of the **`#pragma token`** syntax. For example the first **`+TOKEN`** statement translates to :

```
#pragma token PROC { TYPE t, EXP rvalue : XtArgVal : e |\
      EXP e } EXP value : t * : __MyXtArgValToPtr #
```

This means that **`__MyXtArgValToPtr`** takes an expression **`e`** of type **`XtArgVal`** and returns an expression of type **`t *`** for some type **`t`**. The actual value of **`t`** will be deduced by the C → TDF producer from the context in which `__MyXtArgValToPtr` is used. Thus `__MyXtArgValToPtr` is a generic `XtArgVal` to pointer conversion token. (See [3] for more details.)

The conversion tokens need to be defined in the implementation. If `XtArgVal` is `long`, as in the model implementation, this takes the form :

```
#define __MyXtArgValToPtr( x ) ( ( void * ) ( x ) )
#define __MyPtrToXtArgVal( x ) ( ( XtArgVal ) ( x ) )
#define __MyXtArgValToInt( x ) ( x )
#define __MyIntToXtArgVal( x ) ( x )
```

Note that the implicit conversion tokens have been defined in terms of explicit conversions.

4.3 Omissions from the Xt Specification

The following omissions from the Xt specification were discovered when running *tspec* on the Xt description.

- The type `Pixel` is not specified. It is probably an integral type.
- The types `XtValueMask`, `XtGeometryMask`, `XtEventMask` and `XtGCMask` are not specified. They are probably integral types.
- The types `XtInputId`, `XtIntervalId` and `XtWorkProcId` are not specified. They are probably integral types (see section 6.4).

In addition the following contradictions between the specification and the implementation were discovered :

- In the specification `XtPending` returns `Boolean`, whereas in the implementation it returns `XtInputMask`. I suspect that the implementation is correct.
- In the specification `XtDestroyGC` has two arguments, a `Widget` and a `GC`, whereas in the implementation it only has the `GC` argument. This is a discrepancy documented in the text. It can be worked round by ignoring the given implementation of `XtDestroyGC` and redefining it in terms of `XtReleaseGC` as follows :

```
#define XtDestroyGC( w, gc ) XtReleaseGC ( w, gc )
```

5 Describing Xmu and Xaw

The description of Xmu and Xaw from [7] and [8] raises few new issues. There are a couple of omissions and misprints in Xmu :

- The type `XmuWidgetNode` is not specified. It has been made an opaque type.
- The type `CloseHook` is not specified, but since it can take the value `NULL` it must be a pointer type.
- In the specification `XctData` is described as a structure with certain fields, whereas in the implementation it is a pointer to this structure. The implementation is almost certainly correct.

- There is a misprint in the specification of **`XmuStandardColormap`** - its first argument should be a **`Display *`**.

In addition, Xmu uses the type **`caddr_t`** as a generic pointer to data. This is one of my bugbears since everyone uses it, but nobody specifies it. For example, in XPG3 it is optional. SVID3 also describes it as being optional, but then uses it in its RPC section. Its use in Xmu is unnecessary since Xlib specifies **`XPointer`**, which is a similar generic pointer to data (I think the use of **`caddr_t`** is a hang-over from a previous release). This illustrates one of the considerations in API specification : the dependency on other APIs. The only other external types imported by X11 are **`wchar_t`**, **`size_t`** and **`FILE`**, which all appear in ANSI.

In Xaw we have the following contradiction between the specification and the implementation :

- **`XawSimpleMenuClearActiveEntry`** returns **`Widget`** in the specification, but **`void`** in the implementation. I suspect that the implementation is correct.

6 X11 Applications

The next stage of the testing process is to try to compile a few X11 applications using the **`#pragma token`** headers generated from the *tspec* description. These applications were all compiled to target independent TDF on a 68040 based machine running HP-UX, and this TDF was installed on the ICL SPARC machine running System V 4.2 previously used to build the TDF libraries. Because this was merely an experiment and not a serious porting exercise, the default *tcc* compilation mode, **-Xa**, was used, rather than the stricter **-Xc** or **-Xs** (see [4] for details).

All the applications compiled come from the MIT X11 Release 5 demos.

6.1 maze

maze is a very simple Xlib application, so it should be possible to specify its API using the **-Yxlib5** flag to *tcc*. However there is a problem : the program uses the header ***X11/Xos.h*** and a couple of bitmaps which do not form part of Xlib proper, but are specific to the MIT implementation.

For this reason it was decided to set up a simple extension API, ***xmit5***, containing some commonly used, MIT specific headers. The contents of ***xmit5*** are as follows :

- The header ***X11/Xfuncs.h***, which includes ***string.h*** and defines **`index`**, **`rindex`**, **`bcopy`**, **`bzero`** and **`bcmp`**.
- The header ***X11/Xos.h***, which includes the header above. It also includes ***time.h*** and defines **`struct timeval`** and **`struct timezone`**. If the API extends POSIX, it also includes ***sys/types.h***, ***fcntl.h*** and ***unistd.h***.
- Most of the standard bitmaps supplied with the MIT release.

With this extension, *maze* may be compiled using the **-Yxlib5 -Yxmit5** flags to *tcc*. No further difficulties are encountered.

6.2 puzzle

The API for *puzzle* differs only from that of *maze* in that it uses POSIX as its underlying base API, rather than ANSI. This may be expressed by passing the **-Yposix -Yxlib5 -Yxmit5** flags to *tcc*.

There is a problem which occurs during the compilation. *puzzle* assumes that the structure `Visual` has a field `visualid`, whereas the specification asserts that it is an opaque type. This is the TDF compiler earning its money by detecting that the application does not conform to the Xlib API.

Having detected this problem, how do we work around it? There are several possible approaches. Perhaps the easiest is for the application writer to say, OK I know what Xlib says, but ignore that and assume that `Visual` is defined as in the standard MIT implementation. This may be expressed as follows :

```
#ifdef __ANDF__
#pragma ignore Visual
typedef struct {
      XExtData *ext_data ;
      VisualID visualid ;
      ....
} Visual ;
#endif
```

Note that the application writer is here using a simple extension API of Xlib. Using this extension reduces the portability of the program - it will only be portable to machines which have this particular definition for `Visual`.

Another problem encountered when compiling *puzzle* is a very common one. The source has :

```
#include <errno.h>
extern int errno ;
```

which the TDF producer objects to, because `errno` need not be an `extern int`, but a modifiable lvalue of type `int`. The easiest workaround is to change the program to :

```
#include <errno.h>
#ifndef errno
extern int errno ;
#endif
```

6.3 x11perf

The API for *x11perf* is again given by **-Yposix -Yxlib5 -Yxmit5**. The application compiles first time, apart from one error. The source has :

```
time_t t = time ( ( long * ) NULL ) ;
```

which is only correct if `time_t` is `long`. This should be corrected to :

```
time_t t = time ( ( time_t * ) NULL ) ;
```

6.4 xgas

xgas is a more interesting application since it uses the Athena widgets and the P-headers from Xt. This API is given by passing the **-Yxaw5p** flag to *tcc*. There are a couple of problems. Firstly **`NULL`** is not defined, so it is necessary to insert a couple of :

```
#include <stdio.h>
```

statements. This shows the peril of including something, which includes something, which declares what you want - it isn't always portable. The second problem is that *xgas* consistently uses **`caddr_t`** when it means **`XtPointer`**. A global replacement solves this problem. *xgas* also assumes that **`XtIntervalId`** is an integral type, which was used to help make the decision on this type mentioned in section 4.3

6.5 xeyes

xeyes also defines its own widget, and also uses the non-rectangular window shape extension. It was a simple task to describe the latter as part of a separate extension API, ***xext5***. The API for *xeyes* is then described by passing the **-Yxpg3 -Yxmu5p -Yxext5 -Yxmit5** flags to *tcc*.

The first problem encountered is that *xgas* applies functions which are expecting a **`Widget`**, such as **`XtDisplay`**, directly to its own widget, **`EyesWidget`**, without an explicit cast. As was mentioned above, this is disallowed by the C prototype rules. There are two possible solutions. The dull one is to insert all the explicit casts. The interesting one is to define a conversion token which tells the producer that **`EyesWidget`** $\rightarrow$ **`Widget`** conversions are allowed. This is done as follows :

```
#ifdef __ANDF__
#pragma token PROC ( EXP rvalue : EyesWidget : ) |\
      EXP rvalue : Widget : CrossEyes #
#define CrossEyes( w ) ( ( Widget ) ( w ) )
#pragma accept conversion CrossEyes
#endif
```

Note that we are providing both the conversion token and its definition.

The only other problem encountered is that *xeyes* steps outside the Xlib specification by directly accessing the **`root`** field of the opaque structure **`Screen`** in the expression :

```
XtScreen ( toplevel )->root
```

Because **`Screen`** is genuinely opaque in the **`#pragma token`** headers this error is detected by the C $\rightarrow$ TDF producer. The screen access macros should be used instead :

```
RootWindowOfScreen ( XtScreen ( toplevel ) )
```

7 Conclusions

In addition to providing a useful end product - the **`#pragma token`** headers which allow the C $\rightarrow$ TDF producer to API-check applications - the API description process is a useful exercise in its own right because of the information it provides on the underlying paper specification.

While the paper specification remains merely as a paper document it is an entirely passive object. Transforming it into the *tspec* form makes it an active entity, by allowing applications and implementations to be checked against it, as well as allowing for internal checks to be applied. Any omissions from the specification become immediately obvious, as do any dependencies on other APIs.

8 References

[1] *TDF and Portability*, DRA, 1993.
[2] *tspec - An API Specification Tool*, DRA, 1993.
[3] *The C to TDF Producer*, DRA, 1993.
[4] *tcc User's Guide*, DRA, 1993.
[5] *Xlib - C Language X Interface*, MIT, 1991.
[6] *X Toolkit Intrinsics - C Language X Interface*, MIT, 1991.
[7] *Xmu Library*, MIT, 1991.
[8] *Athena Widget Set - C Language Interface*, MIT, 1991.

9 Figures

Fig. 1. The Widget Hierarchy for Xt.

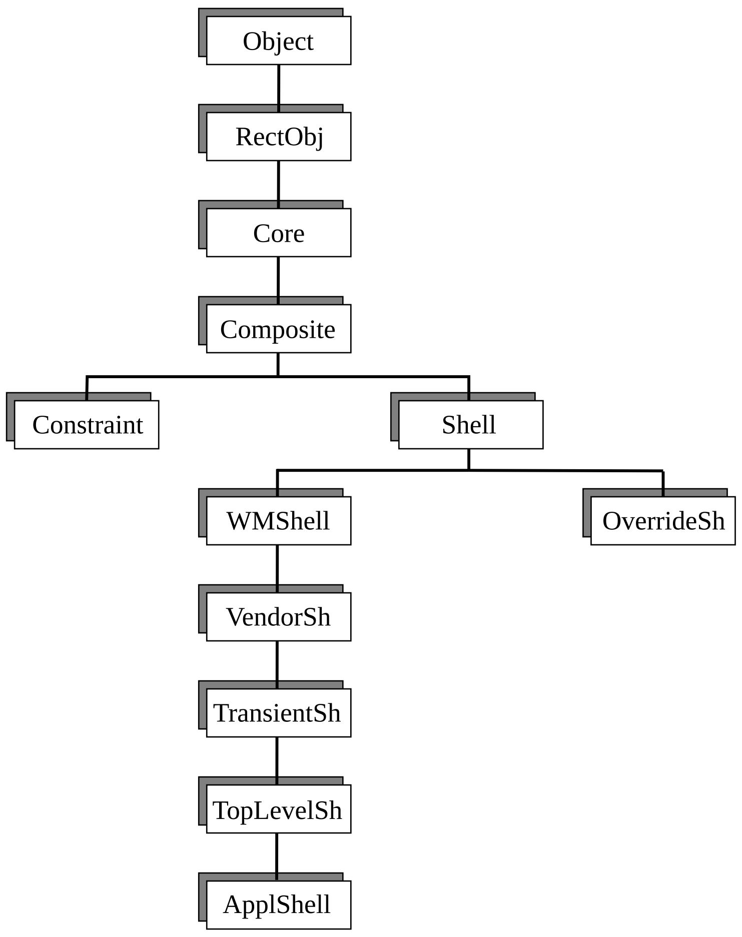
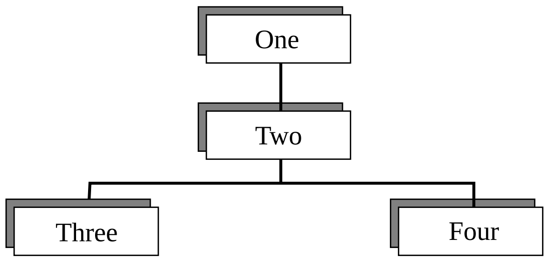


Fig. 2. Example Type Hierarchy


One
Two
Three
Four