The Theory and Practice of Refinement
ed. J. McDermid.
Butterworth Scientific. 12-26

Interactive systems design and formal development are incompatible?

Alan Dix
Michael Harrison

Human-Computer Interaction Group
University of York

INTRODUCTION

We have argued elsewhere that formal methods are useful in the design of interactive systems [Dix 1985, Dix 1987a, Dix 1987b]. Further, we have argued that when formal methods are being used anyway for system design, it is essential that human factors considerations be explicitly included. However, when we consider the relationship between the requirements of the *process* of interactive system design, and the *process* of formal development, several conflicts occur. These problems are not unique to interactive systems design, but are inherent in the concept of formal development; it is just that the rigours of interactive systems intensify and bring these problems to notice.

After considering the differing requirements of the two domains of interactive systems design and formal refinement, we will proceed in a dialectic style. Conflicts will become apparent between the sets of requirements, which we will attempt to resolve, eventually leading to the need for, and proposals for techniques for structural transformation of systems at the module level during refinement. The technique of *interface drift* will be discussed in detail.

THE REQUIREMENTS OF INTERACTIVE SYSTEMS DESIGN

Interactive systems design gives rise to three problems:

- Formality gap
- Rapid turnaround - iterative design
- Fast prototypes

These are described below:

Formality gap

Designing a system involves a translation from someone's (the client's) informal requirements to an implemented system. Once we are within the formal domain we can *in principle* verify the correctness of the system. For example, the compiler can be proved a correct transformer of source code to object and we can prove the correctness of the program with respect to the specification. Of course, what can not be proved correct is the relation between the informal requirements and the requirements as captured in the specification. This gulf between the informal requirements and their first formal statement is the *formality gap* (*fig. 1*). Formalising HCI requirements is a complex process, not only are they not formally understood to start with, but it is likely that they are fundamentally unformalisable. We are thus aiming to only formalise some aspect of a particular requirement and it is difficult to know if we have what we really want.

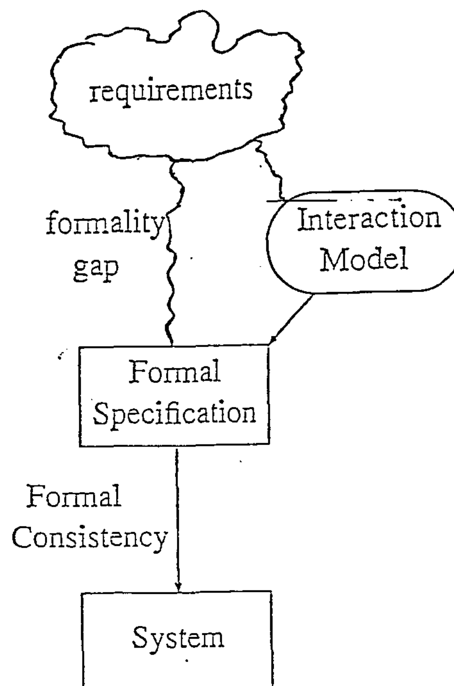


figure 1 - formality gap

We have suggested the use of an *abstract interaction model* [Dix 1985, Dix 1987a] to describe some desirable generic interface properties. If the abstract model is designed well for a particular class of principles, it can help bridge the formality gap. To achieve this, there must be a close correspondence of structure between the abstract model and the informal concepts. However the abstract model will only capture some of the requirements, and the same principle of *structural correlation* must apply to the entire interface specification.

Rapid turnaround

Because of the formality gap, we will never entirely capture the requirements for an interactive system [Monk 1988] and thus some form of iterative design cycle is usually suggested. That is a prototype system is built based on a first guess at the interface requirements, this is then evaluated, and new requirements are formed. The turnaround of prototypes must be fast for this process to be effective, perhaps days or even hours. Frequently this is done using mockups that have the immediate appearance of the finished product, but lack the internal functionality. However many of the interface properties we have studied permeate the whole design of the system, and hence we would see it as necessary to have some reasonable proportion of the functionality in this prototype.

Fast prototypes

Not only does the turnaround of prototypes have to be rapid, but the prototypes themselves must execute reasonably fast to be usable. A slow interface has a very different feel to a fast one, and hence wrong decisions can be made if the pace of the interaction is unreal. The evaluator may be able to make some allowance, but this ability is limited. First, it is very hard to evaluate an interactive system where you type for a few seconds, ~~then~~ have to go away for ten minutes and have a cup of tea before seeing what you have typed appear. The only thing to be said in its favour, is it might encourage predictability as a result of the "gone for a cup of tea" problem! [Dix 1985] Not only is it difficult to appreciate the system at all, but also very poor performance can encourage the wrong decisions to be taken, negating the benefits of prototyping. For example, imagine designing a word-processor. One design is a full screen, "what you see is what you get" editor, the other a line editor with cryptic single character commands. At the speed envisaged in the production version, the full screen editor would be preferable, but when executed a hundred times slower, the line editor, requiring fewer keystrokes and not relying so much on screen feedback, would appear better. It was for just such situations that line editors were developed! Of course, such a major shift would be obvious for the designer, however there may be many more subtle decisions wrongly taken because of poor performance.

Contrast with non-interactive systems

We can contrast the above requirements with the design of non-interactive systems such as data processing or numerical applications.

Whereas the requirements of interactive systems are very difficult to formalise, for a DP application like a payroll, this may not be too much of a problem. The requirements are already in a semi-formal form (eg. pay scales, tax laws) and they are inherently formalisable. Similarly, because these requirements are well understood, there is less need for iterative design, changes in requirements may be over

requirements contrast	
interactive system	DP or numerical application
wide formality gap	well understood requirements
rapid turnaround - iterative design	slow turnaround
fast prototypes - for usability	functionality sufficient

periods of months or years with corresponding turnaround times. Finally, with such applications it is sufficient to prototype the functionality only with little regard for speed. It is easy to produce a set of test data and then run a numerical algorithm overnight to check in the morning for correctness. Even complex distributed systems may be able to be simulated at a slow pace.

FORMAL AND CLASSICAL DEVELOPMENT COMPARED

In this section we will compare formal development with classical development. By classical I mean a non-formal approach to development (hacker rather than Homer) [Homer BC]. I will assume that the requirements are known and will consider ~~how~~ these requirements are used to produce a first implementation, through optimisation to a version that is suitable for use (perhaps as a prototype or even as a product). Then how these two processes respond to a changing requirements

Initial development

Classical

Given loosely stated requirements the programmer will, possibly using some intermediate graphical or textual plan, produce a first implementation of the program. Typically this will already be structured with efficiency in mind. However, if this is not fast enough several stages of optimisation may ensue before reaching a version suitable for release (*fig. 2*).

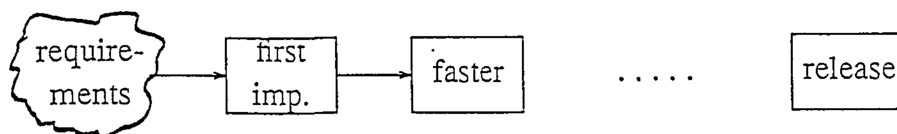


figure 2 - classical development

Interspersed with this optimisation process will be the debugging. In principle one might debug each version in turn until one is sure of correctness. In practice, this debugging will be distributed, errors present from early implementations being corrected only later in the process. These changes will probably never be reflected in those early points as the early versions are likely to be overwritten, or at best stored in a source control data base.

Formal

At this stage the formal development process is quite similar. The requirements will be first used to generate a specification. This first specification will then go through several levels of refinement within the formal notation, both to make it constructive enough for implementation and possibly as a first stage in moving towards an efficient formulation. The final product of these specifications will be used to derive the first implementation, which itself may then be optimised through several versions before a releasable version is produced (*fig. 3*).

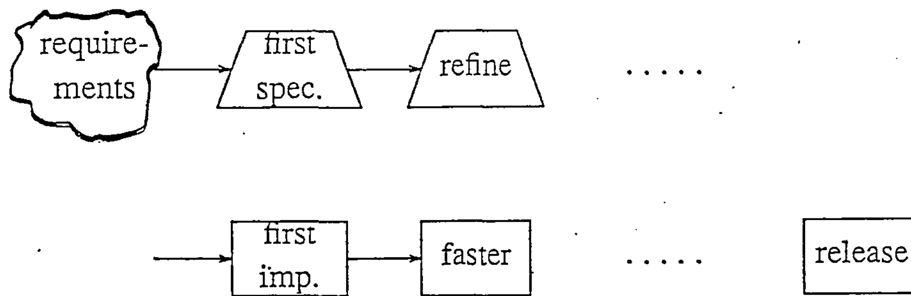


figure 3 - formal development

Again there will be debugging steps, both in the specification as each refinement is checked against the previous specification for correctness and in the implementation as this also is checked for correctness. The only difference here from classical development is that debugging is less likely to involve errors from previous versions. In particular, because the initial specification is much more directly derived from the requirements (it may in fact be a formal statement of the requirements) there are likely to be less changes needed to make the final product match the informal requirements. With some formal development paradigms, such as refinement by transformation, it could be argued that there is *never* any debugging as all versions are guaranteed to be correct. However, even here back-tracking in the transformation process is a form of debugging. From now on we will largely ignore these debugging steps, as they form a development process at a finer granularity than we are considering.

Changing requirements

Classical

If the changes in requirements are extreme, the programmer may be tempted to throw away all previous work and start from scratch. More usually, however, the most developed version of the system will be used and altered to fit the new needs (*fig. 4*). Because the fastest, optimised version is used there is unlikely to be much need for optimisation steps, except where radically new algorithms have been introduced. Note especially, that the changing requirements are not seen to affect at all the early un-optimised versions of the program, they are just history.

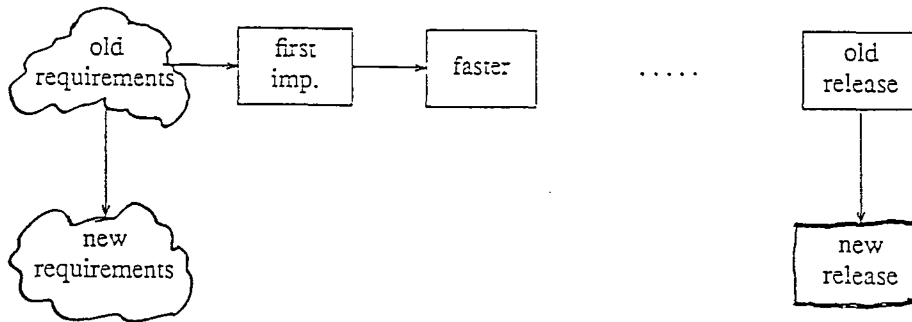


figure 4 - classical development, change in requirements

Formal

The formal situation is very different. The intermediate versions, and especially the first specification are the *proof* that the final version really does satisfy the requirements. There may well be testing and validation as well, but it is the process of development itself which is the major source of confidence in correctness. This is even (and especially) true when the process does not employ automatic checking. It is insufficient (although tempting) to change the specification a bit, change the final version a bit, and say the process is still formal. No, for formal correctness a change in requirements demands a complete rework of the whole development process from initial specification to final optimised system (*fig. 5*). Again in worst case, this may involve a complete rewrite, but usually will be obtained by propagating small changes through the stages.

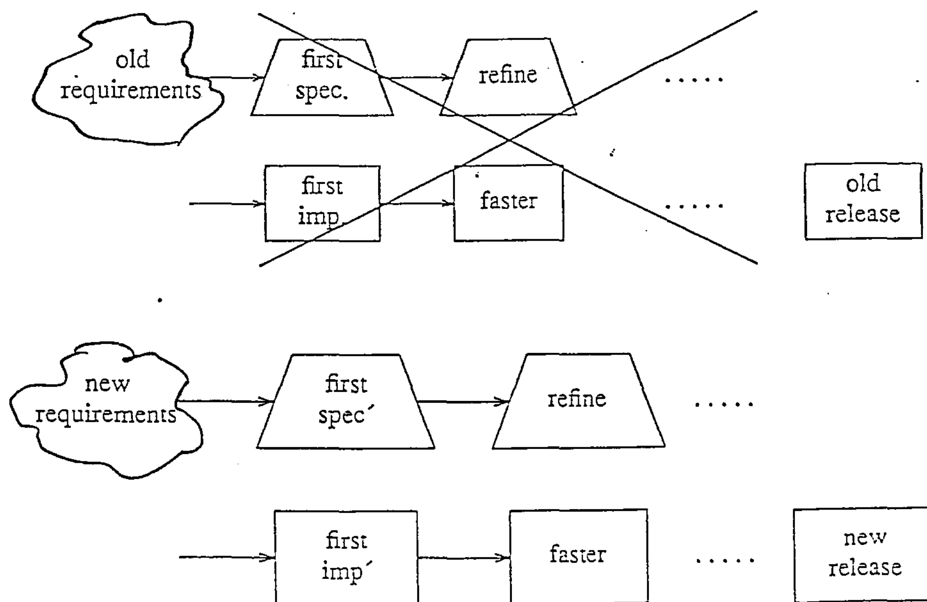


figure 5 - formal development, change in requirements

CONFLICT - rapid turnaround and refinement

How do the requirements for interactive systems design fit into this picture.

- rapid turnaround $\Rightarrow$ lots of changes in requirements, say m in all.
- fast prototypes $\Rightarrow$ lots of steps in the refinement process, say n steps.

That is the length (n) of the development chains, and the number of them (m) are far greater for interactive systems compared with non-interactive. These interact with the cost figures:

- *Classical* - this takes n steps to produce the first version, and one further step for each requirements change, so $m+n$ versions will have finally been produced.
- *Formal* - this too takes n steps to produce the first version, but then all requirements changes also require n bits of work, so the number of separate specifications and programs eventually produced will be $m \times n$.

A first glance the difference between these two cost figures, $m+n$ for classical development and $m \times n$ for formal development, is astounding. One wonders if formal development can ever be a practical proposition. A one-off cost can be acceptable, even if high, but a recurring cost like this could never be.

Happily, the situation is not as bad as it seems. First, the number of "changes in requirements" may be vastly different. For a system using classical methods, many of the requests coming in for changes will not be true changes in requirements, but restatements of parts of the original requirements the system does not satisfy. That is, long term debugging. As we've noted, formal development should reduce this to a minimum (or even zero if the requirements are a formal document themselves). Unfortunately this argument does not hold too well for interactive systems design, as we've noted that here formal methods will not perfectly capture informal requirements. Again the number of development steps may differ. Formal development may require the additional refinement steps, but the costs of each step will be much reduced as there will be less debugging.

Orthogonal development

It is clearly critical just how costly the changes to the intermediate specifications are when requirements change. In the best case, the n changes needed for formal development may be a distribution of the effort of the single change in the final implementation for classical development. We would hope that a small change in requirements, would only require a small change in each of the intermediate specifications.

Modularisation is the standard vehicle for such localisation. A small change in requirements is likely to only require changes in one or two modules of the specification. If the refinement and the implementation preserve the

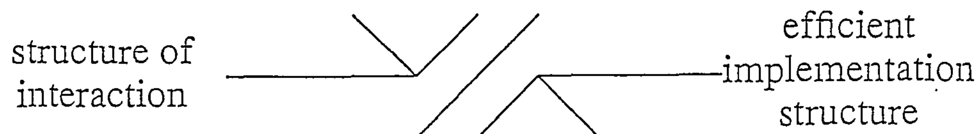
specification's module structure then we can alter the relevant modules in the refined specifications and implementation and reuse all other modules. That is the refinement process must obey the principle of *structural correlation.*

Cedar [Donahue 1985] uses such a system, carefully maintaining dependency information so that recompilation upon the change in a module is minimised. (Compilation is of course a form of refinement.) However the idea of locality in Cedar, as in most traditional languages such as Ada [Ichbiah 1983], is based on type correctness. The semantic repercussions of changes are not considered, and it is not obvious therefore that the rest of the system will behave as expected.

Happily, the situation in formal development can be much better. Modularisation based upon semantically defined interfaces leads to full semantic independence between modules. One module cannot affect another module without a change in the interface. This leads to an *orthogonal* development idiom. The specification consists of many modules, as do each of the intermediate stages to the final implementation. Development within any one of those modules is independent of all the rest. If a change is made in one specification module then an early, perhaps very inefficient, version of the corresponding implemented module can be used with highly optimised existing modules. This means that we can get a reasonably fast prototype back into the hands of the interface designer very rapidly.

CONFLICT - structural correlation and orthogonal development

To summarise where we have got to: The formality gap forces us to match the structure of interaction in the original specification. The conflict between the need for rapid turnaround and fast prototypes, and formal refinement forced us to use orthogonal development, with structural correlation throughout the refinement process. Thus we will arrive at an implementation with a structure still matching closely the structure of interaction. However at the end of the day, in order to obtain a fast prototype we will need an efficient implementation structure. Unfortunately the interaction structure and the structure required for efficiency do not usually agree.



It is worth noting that the experience of formal development is likely to be very different here from classical development. In the latter, the structural design will from the beginning be oriented towards efficient implementation. In fact, the

same can be done with formal specifications. But we argue that this is not what a specification is for and such trends are likely to yield errors in translating requirements to specification [Jones 1980]. In short, a good formal specification is likely to have problems with orthogonal development.

Before throwing away orthogonal development, it is worth noting how much can be done within the idiom. Many standard techniques can be used within a module. For instance, choosing efficient representations, caching results from other modules to reduce function call overheads and general algorithmic improvement. As a general rule most time is taken with the modules lowest in the usage hierarchy, so obviously most effort goes into improving these low-level modules. This will be tempered however with the use of a "don't use it" strategy - avoiding the need for a slow operation rather than speeding it up. Caching is one heavily used application of this principle.

However, despite large gains through intra-module optimisation, the time may come when the system is still too slow and inter-module optimisation is needed. the challenge is therefore to achieve this in as well structured a way as possible, preserving correctness, and making easy the reuse of existing modules as requirements change.

NON-ORTHOGONAL DEVELOPMENT - INTERFACE DRIFT

When is non-orthogonal development necessary? We consider the simple case of two modules, a *server* providing some functionality, and a *client* which uses this service. The need for intra-module refinement shows up in two ways.

- *Information* - the client module could perform further optimisation if the server could give it some more information. For instance, the use of locality information to tell the client about where the server has made changes. This locality information is not usually present in the original specification interface and is therefore not normally available to the client although the server "knows" it. This type of information may enable the client to optimise changes in its own data structures.
- *Services* - there may be computations in the client module involving many calls across the interface to the server. Such operations may well be performed far more efficiently by the server, reducing function call overhead at very least, but quite likely being more efficient in general due to the server's knowledge of representation and (again) information not available across the interface.

As we can see the two are not entirely independent, and both require *interface drift,* a movement of some information or functionality across the interface between two modules. In general, the information category requires a drift of

functionality across the interface *upwards* from server to client, as the server "opens itself up". Similarly, the services category requires a *downwards* drift of functionality, as the server "takes over" jobs previously performed by the client.

Informal interface drift

Interface drift is not a new phenomenon. However it is usually carried out in an informal manner: if services require the addition of extra operations at the module interface, these are added on the spur of the moment without too much thought. So long as there is some form of target language modularisation a change like this will be obvious. Other changes may slip through without notice: for instance, adding extra results from a function, or requiring additional parameters. The C language (pre-ANSI standard) [Kernighan 1978] will allow such changes without altering any header files (the interface documents) but this would be picked up with typed interfaces such as Cedar, Modula or Ada. More insidious still is when the type and number of parameters and result remain unchanged, but the semantics alter. For instance, imagine a stack module with a *top* function yielding the top value on the stack. It is noticed that all calls to *top* are followed immediately by a call to *pop* to remove that value from the stack. To save this additional function call, the *top* function is changed to *pop* the stack as a side effect. The interface looks identical, but its functionality has altered drastically.

Interface drift due to information requirements can be even harder to detect. When the information is passed explicitly it is very similar to the situations above. However, often general information about the semantics may be passed informally, especially if the same person is coding both modules.

Informal interface drift that is not properly recorded can lead to disastrous results. For instance, when a colleague was implementing some windowing routines for an application, one of the authors 'let on' that only the currently selected window ever changed between user interactions. This information was used to improve the speed of the windowing. Later, inevitably, the same windowing routines were used when the application had changed slightly. Now other windows were also subject to change, requiring a major rewrite of parts of the windowing code.

We can imagine similar problems to the above with the stack example. But surely, if properly documented with interface specifications amended to record changes in semantics or assumptions made about behaviour, interface drift is an acceptable part of formal development? In fact, this is not the case.

Reuse and non-orthogonal development

Interface drift is a form of non-orthogonal development. Orthogonal development assumes that changes to one module affect no others, however with interface drift an interdependence is set up between different module development histories. Orthogonal development was said to give us fast turnaround and extensive reuse of optimised modules. So we would expect to get problems of reuse caused by interface drift.

How do these problems manifest themselves? Imagine the following scenario. We have two specification modules, A and B. A is the client and uses operations defined in B using an interface specification *B_def*. A and B are refined (by several steps) to A_1 and B_1 respectively. The interface is still of course *B_def*. At this stage interface drift is seen as necessary. New modules A_2 and B_2 are produced with interface *B_def'*. So long as we prove that A_2 using B_2 behaves identically to A_1 using B_1 correctness is preserved. After this, more optimisation is carried out yielding A_3 and B_3 still with the interface *B_def'*. Any changes to other parts of the specification leave A and B's development unaffected. Problems come when B's specification is altered to B^*. The development of B must be redone and a new first implementation of B^* is produced. It still satisfies *B_def* and so can be used with A_1. Unfortunately it *cannot* be used with the heavily optimised version A_3. Perhaps later a version of B^* can be developed that mimics the development of B and satisfies *B_def'*, but for the moment the work done in producing A_3 cannot be reused.

In short, non-orthogonal development hampers reuse.

Mechanisms for interface drift

Some simple points of refinement style can help reduce the impact of non-orthogonal development. First, it should always be left as late as possible in the refinement and optimisation process. This means that if some of the development of a module cannot be reused after another module with which it is linked by interface drift has changed, at least the amount of work wasted is not too great. Second, the desire to perform such tuning must be tempered by the recognition that one or other module may be required for use elsewhere or be subject to a change in requirements, and only changes that are thought to be of general usefulness should be performed until the system has begun to solidify. However, these are councils of perfection. How should we proceed when interface drift is deemed to be necessary?

A general picture of interface drift is as follows. Module A invokes B via an interface *B_def*, and is transformed to two new modules A', B' with interface *B_def'*.

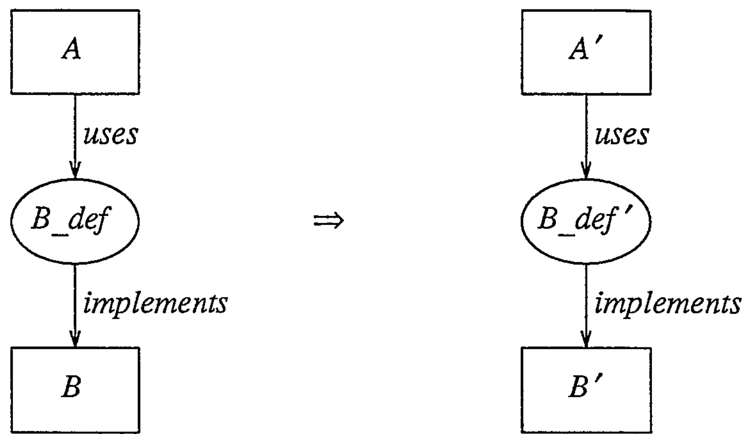


figure 6 - non-orthogonal development step

As we've said, the two composites A-using-B and A'-using-B' must be proved equivalent. This is not ideal, not only because of problems with reuse, but also because it is a large proof involving both A and B, and further the whole idea of having loose interface specifications is to hide unwanted detail. In many cases the following easy and clean method can be used:

The transformation from *B_def* to *B_def′* is often such that the operations in *B_def′* can be defined in terms of those in *B_def*. We can define a new transformation module C, which comprises exactly those definitions, and we get the following situation.

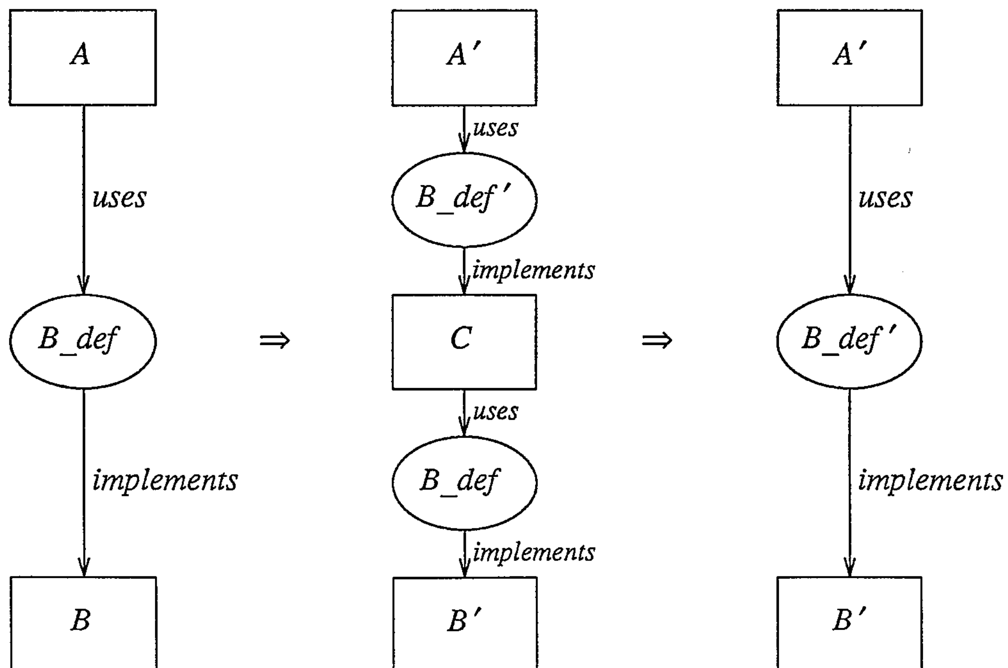


figure 7 - interface drift

In the first stage we only have to prove the equivalence of A and A'-using-C,

which is considerably simpler since C is likely to be small. Similarly in the second stage we need only satisfy ourselves that C–using–B is equivalent to B'. Thus we have factorised the proof effort and obtained a much more controlled breakdown of the interface barrier.

As well as factoring proof effort this also gives us a means of enhancing reuse and ensuring continuity in implementation. The transformation module, if coded, can be a means to smooth implementation change (whether or not formal methods are used). Further, if changes in requirements mean the specification of a new module D satisfying B_def then we can initially code this very simply as D_coded and then C_coded–using–D_coded will interface properly to A'_coded, and thus we quickly have a working prototype with the new module. As time allows we can define a module D' equivalent to C-using-D to bring the system back up to speed.

This technique is comparable at the system level to program transformation techniques such as fold/unfold or tail-recursion removal [Feather 1982].

Zdonik uses a somewhat similar technique in his object-oriented database for version management [Zdonik 1986]. He allows object type definitions to change and uses *filter* functions to translate between old and new types. These filters are the parallel of the module C. The technique proposed there is only applied to a system employing (effectively) single sorted algebras whereas the scheme proposed here is for multi-sorted algebras and could even involve large changes such as the addition or removal of data-types from the interface.

CONCLUSIONS - an interface for formal development environments

Software Engineering

An orthogonal development paradigm fits most closely with traditional software engineering practice, and is necessary for the frequent turnaround of requirements for interactive systems. However it conflicts with the need for structural correlation between requirements and specification. A technique of structural transformation, interface drift, has been described which helps to control the complexity of non-orthogonal development aiding proof and reuse. It is however still more complex than simple development, so the general advice of putting off such steps till late in refinement still holds (and where possible avoiding it entirely). In particular, overlapping cases of interface drift, whilest being far better behaved then typical unstructured development clearly pose problems.

The techniques have been presented from the view of the particular pairs of modules concerned. Elsewhere a more coherent description is given of the high level representation of such steps within a system development database [Dix 1989].

Human Interface (of development environment)

The discussion has been focused on the rigours placed on formal development when the subject of that development is an interactive system. However it leads to important questions about the cognative demands of the resulting process, and the user interface to any support system. Recalling the comparison between classical and formal development processes, the classical took $m+n$ steps to the formal's $m \times n$. Although, we argued that the actual effort expended would not be as extreme as this suggests, it is likely that the number of *documents* (formal and informal) produced will approach this level. That is, the complexity of formal specification 'in the small' is likely to approach that of classical programming 'in the large'.

Large scale programming has in the past been supported by well documented analysis and management structures and currently project support environments are being developed to aid this process. However, the time-scales involved in interactive systems design preclude these approaches requiring instead environments more akin to exploratory programming [Goldberg 1984]. Marrying these conflicting styles will require exceptional organisation and ingenuity in the environment and its user interface.

Whilest the development process is totally orthogonal, the complexity is unlikely to be too bad, as the documents can be located in a matrix, the dimensions of which a reasonably small. However non-orthogonal development steps significantly complicate this picture. They make the structure more complex, introducing problems of naming and representation. In the example given of interface drift, we have assumed that the two resulting modules inherit the names of the original two, but it may be that the amount of functionality transferred across the interface in C is such that we feel that the new server should really be the natural successor of the A stream whilest the new client is a mere stub, constituting a new name space. Similar issues arise if a module stream completely disappears, or if a module is decomposed. Similar issues arise when considering graphical representations. Whilest such issues are apparently insignificant formally, they will have a major effect on the usability of formal development environments.

REFERENCES

Dix 1985. A.J. Dix and C. Runciman, "Abstract Models of Interactive Systems", pp. 13-22 in *People and Computers: Designing the interface*, ed. P. Johnson & S. Cook, Cambridge University Press (1985).

Dix 1987a. A.J. Dix, M.D. Harrison, C. Runciman, and H.W. Thimbleby, "Interaction models and the principled design of interactive

systems'', pp. 127-135 in *Proceedings of European Software Engineering Conference*, Springer-Verlag (1987).

Dix 1987b. A.J. Dix, *Formal Methods and Interactive Systems: Principles and Practice*, D.Phil. thesis, Department of Computer Science, University of York (1987).

Dix 1989. A.J. Dix, *Software engineering implications for formal refinement*, (submitted to ESEC'89) (1989).

Donahue 1985. James Donahue, ''Cedar: An environment for experimental programming'', pp. 1-9 in *Integrated project support environments*, ed. J McDermid, IEE Software Engineering Series (1985).

Feather 1982. Feather, M.S., ''A System for Assisting Program Transformation'', *ACM Transactions on Programming Languages and Systems* **4**(1), pp. 1-20 (1982).

Goldberg 1984. Adele Goldberg, *Smalltalk-80, The interactive programming environment*, Addison-Wesley (1984).

Homer BC. Homer, *Iliad*, circa 900 BC.

Ichbiah 1983. J. D. Ichbiah and *et al* (eds.), ''Reference Manual for the Ada Programming Language'', ANSI/MIL-STD-1815A-1983 (1983).

Jones 1980. C. B. Jones, *Software Development: A Rigorous Approach*, Prentice-Hall (1980).

Kernighan 1978. B. W. Kernighan and D. M. Ritchie, *The C programming language*, Prentice Hall (1978).

Monk 1988. A. F. Monk, P. Walsh, and A. J. Dix, ''A comparison of hypertext, scrolling and folding as mechanisms for program browsing'', pp. 421-436 in *People and Computers: From Research to Implementation - proceedings HCI'88*, ed. D.M.Jones & R.Winder, Cambridge University Press (1988).

Zdonik 1986. S. B. Zdonik, ''Version management in an object-oriented database'', pp. 405-422 in *Advanced Programming Environments*, ed. Reidar Conradi, Tor M. Didriksen and dag H. Wanvik, Springer-Verlag, Lecture Notes in Computer Science 244 (1986).