Augmenting the Design Process: Transformations from Abstract Design Representations

David Budgen and Grant Friel
Department of Computer Science
Keele University
Keele
Staffs. ST5 5BG

Abstract

Most of the software design practices that are in current use are based upon the use of graphical notations, and while providing a suitable vehicle for the designer's need to model abstract ideas about the structure of a solution and its expected behaviour, these forms lack any rigour in their syntax and semantics. This paper describes some of the work performed to transform abstract graphically-based design descriptions into a more formal notation that is capable of acting as a high-level prototype of the design. In particular, we describe the way in which these concepts have been encapsulated in an experimental "designer's workbench", and outline some possible future developments based upon using this as a framework for the development of further features and facilities.

1 Introduction

The development of a design for a large software-based system continues to form an area of difficulty for current software development technologies, both in terms of structuring the design process itself, and also of providing tools that can be used to support the design process. In this paper we explore some ideas about how the process of software design can be augmented and reinforced through the use of an experimental CASE tool, which assists both with developing a systematic graphical design representation, and also with transforming this into an executable prototype.

We can consider the process of software design as providing a 'how' stage in system development. Having determined, by means of some requirements analysis procedure, just 'what' is needed from a system in order to meet a customer's needs, the system developers then proceed to seek the means of achieving these aims. The planning process that results is generally considered to involve the design of a solution.

The current thinking about the nature of the design process in general is that this does not form an analytical process, as occurs in the domain of scientific research; but rather that it is one in which the designer postulates a model that represents their intentions, and then explores this to determine whether it can be developed to provide the required behaviour

[15, 10]. The Peters/Tripp model [16] is an example of how this view of design can be mapped on to the design procedures that are generally used for developing software. The design process is further complicated for the case of software design, since it involves the creation of an artifact which has both passive structure and dynamic behaviour, since software is represented by a *program* (with static qualities), which is then executed as a *process* (exhibiting dynamic behaviour).

The current widely-used software design practices make extensive use of graphical symbols, in conjunction with relatively unstructured text, in order to help the designer to handle highly abstract concepts in a relatively structured manner. The relatively abstract syntax and semantics of such forms are well suited to this initial stage in design, where the designer is building up their ideas about how a system might function (and then later considering how it might be structured). Experimental study of how designers work has indicated that they build 'mental models' of the intended system, and that during the process of development of a model, the designer will also 'mentally execute' the model in order to study its behaviour [3, 23].

More formal descriptive techniques that are based upon the use of mathematical notation, such as VDM and Z, provide a designer with a structure that can be used for reasoning about the behaviour of a system and about its other properties, in a more analytical manner than can be achieved through the use of systematic diagram-based forms. In particular, it is these features of formal notations that have made them of particular value for use in safety-critical systems, and in those systems where reliability of all kinds is of particular importance. However, the relatively intensive mathematics that is required for their use and understanding has so far limited the extent of their practical application.

A related field of work involves making use of formal mathematical descriptions in order to create executable prototypes of a system. Such a prototype then provides the designer with the means of 'executing' the description of a design at an early stage in its development. However, as is the case for almost all formal techniques, the skills that are required for developing such models are still relatively uncommon, and further, it is not evident that the more extensive syntax and semantics required to construct such models makes them particularly well suited to the initial exploration of the designer's ideas.

The general form of the design process that we have investigated can be summarised as follows:

1. Produce an initial design model using the MASCOT notation [1] to describe a network of cooperating parallel processes.

2. Extend the design model by providing behavioural descriptions for the MASCOT design objects, using State Transition Diagrams.

3. Transform the design model into the state-based CSP/*me too* form [11], and 'execute' this in order to explore the dynamic behaviour of the design.

(There are no specific 'method' assumptions to constrain this process.)

We have described elsewhere our work on design transformation that has helped towards putting the systematic description and 'formal' description techniques together in a manner that exploits the strength of each [11]. Using this approach, the more 'fuzzy' description forms based on diagrams are used for initial design modelling, and these are then transformed into a more rigorous form that can then be used as a prototype, as well as providing a basis for further refinement of the design (including the transformation to code). In many ways, this is the designer's equivalent of Pamela Zave's ideas about operational development versus lifecycle forms [24]. Her work has mainly concentrated on the prototyping of requirements specifications, but the general philosophy is very similar.

Our work in this area has identified a need for a degree of tool support if these ideas are to be explored further. In developing the architecture for a UNIX-based prototype of such a tool, termed the *Experimental Design Workbench* (EDW) we have also been able to integrate our ideas about design transformation with some other strands of our work. In this paper we report on:

- the work which has been done and is under way in the area of design transformation techniques;
- the way in which the framework of the EDW has enabled us to integrate this work with our previous experimental studies in design;
- our plans for future developments, both in terms of developing the technical ideas, and also in terms of widening their application;

and the rest of this paper is structured around the discussion of each of these three aspects.

2 Formalising the initial design model

The design of most software systems involves the development of a 'mental model' in the mind of the designer at an early stage in the design process. A large part of this model will be made up of fuzzy, or hazy notions about the intended system, and at this stage it contains little, if any detail and dwells primarily on gross system characteristics — the *formalism dimension* of the Peters-Tripp model [16]. During the initial stages of the software design process this 'mental model' has to be 'captured' and expressed in a form which can be used as a basis for evaluating the designer's ideas, and then for eventually developing the design. In the development of reactive, or real time systems these early stages in the design process are becoming of greater importance as the demand for high quality and increasingly complex software grows.

The more 'traditional' approach to recording the details of a design during software development has been to use natural language, usually with some imposed structure, and sometimes supplemented with diagrams, tables and formulae [13]. Clearly a more rigorous, or formalized approach that reflects the need of the designer is desirable.

More recently, a great deal of effort has been put into formalizing these more abstract aspects of the design process [21], the advantages of which are already well documented [18, 24]. The 'ideal' situation is therefore to have a formal representation of the initial design model from which the design can be developed. But, due to the mathematical nature of formal description languages and the lack of any well defined design procedure for using them, there is often great difficulty in expressing these initial design forms in this way.

Our recent work, described in [11], has shown how a formal design model can be generated through the use of informal design techniques. Using this approach the initial design structure is described by using the informal, but systematic design representation of MASCOT 3 [1], and then by using rule-based transformation techniques, we are able to generate a more formal CSP/*me too* description [9] for the design.

The transformation process involved carries out a change in form of description, and so can also be thought of as a transformation from one design viewpoint to another. Our original design form is described using MASCOT, which expresses the design from an architectural, or structural viewpoint (ie. the system is described in terms of both its internal and external components and the lines of communication, or connections between them). Using the automated design transformation process, this 'static' representation is then transformed into a state based description. That is, each of the identified design components is described in terms of a process and its *local state* and the operations which are carried out to effect a change in that state. These operations represent the *events* which take place within the system. What, in effect, we now have is an event driven, or behavioural viewpoint based upon an elaboration of the original design description — a 'dynamic' representation.

A CSP/*me too* specification is executable, therefore by using this technique the designer can effectively be provided with an experimental prototype at an early stage in the development of a design. Using this, it is then possible to model the *dynamic* behaviour of the system that corresponds to the design, and to compare this to the designer's intentions. In other words, it enables the designer to explore and refine the *mental model*, and thus provides a means of identifying and highlighting those areas where the original design description needs closer inspection and/or modification.

The *Experimental Designer's Workbench* was developed partly as a vehicle for exploring the ideas on design transformation. The need for such a support environment became apparent as a result of developing the design transformation tool described above.

3 The Experimental Designers Workbench

During the process of devising the transformation rules for generating CSP/*me too* structures from MASCOT, it became possible to identify the major limitations imposed by our use of the MASCOT representation. In particular, the dataflow-oriented MASCOT viewpoint provides no means of expressing the *behavioural* aspects of a design. To get around this problem, and therefore to enable the generation of a more complete and accurate CSP/*me too* specification, it became necessary to explore ways of supplementing the MASCOT representation with the required behavioural description (this is currently achieved by using State Transition Diagrams [22]). The workbench itself then provides the framework that is needed in order to integrate these components.

The *Experimental Designer's Workbench* (EDW) is an experimental platform on which a designer can develop and explore ideas during the various stages of the software design process. As discussed in the previous section, it was originally conceived as a means of testing out ideas about design tools and their roles in the design process, and especially those involving transformations between different design forms and viewpoints.

The workbench started off with the work described in [20], which included the design and implementation of the basic architecture of the MASCOT ACP diagram editor and devising the initial structure of the *Design Object Base* (*DOB*). The *DOB* is a form of *object-based* 'database' which holds all the information on each of the design objects, or components. It is discussed more fully in section 3.6.

A major influence in this work and in the early stages of developing the workbench was the MDSE [1] project [7, 5]. This was an Alvey project that investigated some ideas about the use of Knowledge Based Systems in design development. The EDW is in some ways a 'second go' at the MDSE, as we have made use of some of the ideas which it incorporated (as well as using MASCOT as the basic form of design description). A second major influence was the general development in design tool thinking that is reflected in Demo-CAEDE [19], which has similarly sought to support 'visual design' techniques.

We envisaged the workbench developing as a set of loosely coupled tools sharing access to a central 'Design Object Base' (the *DOB*), based on the UNIX philosophy of using separate processes to perform individual tasks. The workbench in its initial form (encompassing both design editor and object base as one monolithic process) had to be modified to conform to this, so the object base was modified to use the UNIX System V shared memory facilities, allowing it to maintain an existence as a separate entity. The design editor then became a separate process which could interrogate and update the *DOB* through its access procedures. Using this as a framework it then became a straightforward procedure to integrate further tools into the workbench.

3.1 The toolset

Currently, the workbench consists of a set of prototype tools, including two graphical design editors, the design transformation tool, and a static design assessment tool (the **`Advisor`**), each of which shares access to the *DOB*. The workbench architecture is illustrated by the schematic diagram shown in Figure 1. Originally, the workbench was developed on a Hewlett Packard UNIX (HP-UX)-based system using the HP-GKS graphics library. Current development is now being carried out on a Sun SPARC-Station platform, to which the workbench was successfully ported, and using the X-GKS library. This is an implementation of GKS which has been built around the X-Windows environment.

The toolset is intended to support the designer during the process of developing a formal representation of their *mental*, or conceptual model at an early stage in the design process, while placing the minimum degree of constraint upon how the designer organises this. (This is a similar philosophy to that used in the Designer's Notepad [12].) To achieve this, the model is first described using the MASCOT ACP diagram editor which results in a design form in terms of the identifiable objects within the system and its environment. Further elaboration of the design description can then be explored by using the State Transition Diagram (STD) editor, using a state machine form to describe the design's behavioural characteristics. This makes it possible for the designer to provide a behavioural description for each of the previously identified design components/objects (ie. each MASCOT system element can have an STD associated with it).

Using the design transformation tool this combined graphical design representation can then be used to generate the more formal description based upon CSP/*me too*. Note that is not necessary for all, or any, of the MASCOT system elements to have been further described using STD's for the transformation tool to be used, although clearly the quality and extent of the CSP/*me too* description will be improved if STDs are available.

The following is a brief description of each of the design tools and the *DOB*.

[1]MASCOT Design Support Environment

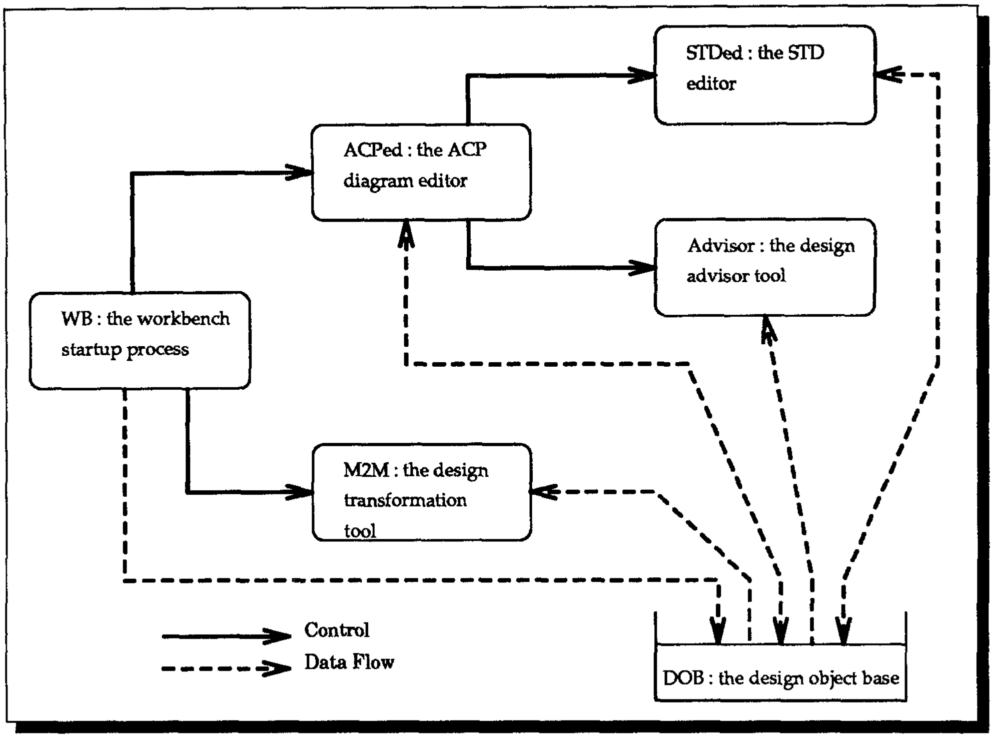
STDed : the STD editor
ACPed : the ACP diagram editor
Advisor : the design advisor tool
WB : the workbench startup process
M2M : the design transformation tool
Control
Data Flow
DOB : the design object base

Figure 1: A schematic diagram of the EDW

3.2 The MASCOT ACP diagram editor

The MASCOT ACP diagram editor (**ACPed**) is a graphical design editing tool based on the MASCOT 3 design representation. MASCOT (an acronym for Modular Approach to Software Construction, Operation and Test) describes a model of a real-time system which contains distributed components and (potentially) many complex devices. This model is based on a network of cooperating parallel processes, using well defined interfaces.

Within the 'MASCOT machine' there are three principal components: a set of abstract concepts, represented by the system elements; a diagramatical representation of a network of system elements; and a runtime executive that supports realisation of the system elements and that provides support for their interactions.

There are three basic types of the above mentioned MASCOT system elements: the *Activity*, which is a single sequential process that performs the algorithmic functions of the system; the *Channel*, which acts as a 'pipeline' mechanism for the transmission of data (messages) between Activities; and the *Pool*, which provides a mechanism for giving Activities access to shared 'static' information. The only means provided for different Activities to communicate with one another, or to share data objects, are by using explicitly defined connections through Channels or Pools.

A MASCOT design is represented graphically by an ACP diagram (Activity,Channel, Pool), which is a network diagram used to show the interconnections between the system elements.

The **ACPed** is an *object based* style of editor, with a menu and mouse driven interface. By *object based* we mean that the 'design objects' (in this case Ports, Windows, Activities, Data Flows etc) once created, can be manipulated either individually or collectively in groups, and are not fixed on initial placement, ie the editing space is treated as a set of building blocks and not as a drawing canvas. The user interface makes use of two main types of menu, namely *pull-down* menus and *context-sensitive pop-up* menus.

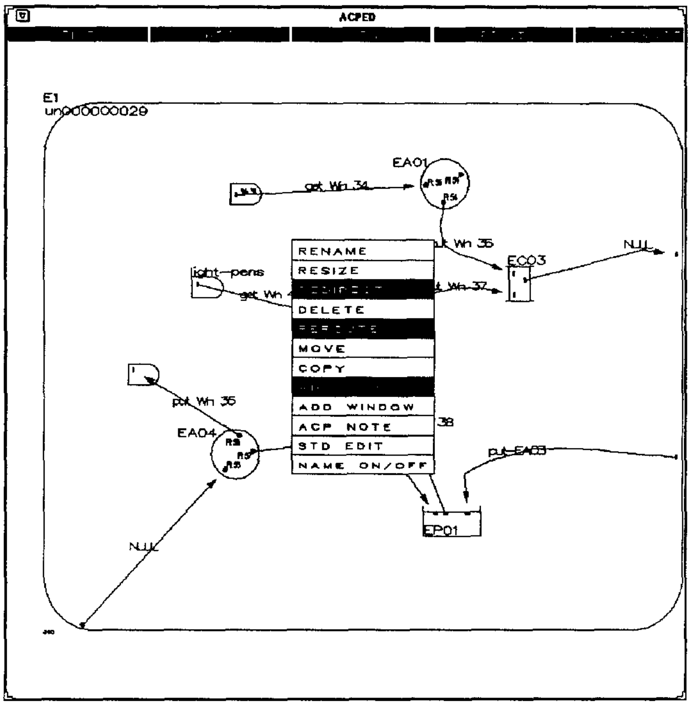

Figure 2: The ACP diagram editor (showing pop-up menu)

Pull-down menus are used for non-editing functions such as file handling, creating new design views, and invoking representation-specific applications (eg. the **`Advisor`**). They are continuously available via a set of pull-down menu 'buttons' positioned across the top of design editing window. An illustration showing the ACP editor in use is provided in Figure 2.

Pop-up menus facilitate the design editing functions and are *context sensitive* in the sense that the menu which appears depends on the particular type of object that has been selected, or where the mouse pointer is located when *clicked*. Selecting a particular design object will cause a menu to pop up that offers editing operations which are appropriate to that class of object. If appropriate, the options to invoke further design tools are included in these *pop-up* menus, as is for example the STD editor (see section 3.3). Other examples include provision for the future development of a *note editor* (to facilitate the attachment of notes to design objects) and a data dictionary description facility (to describe the forms of data used by design objects).

Alternatively, clicking on an area of empty space within the diagram will result in a pop-up menu which offers non-object-specific editing operations, such as the creation of a Channel, Activity or Pool.

It is worth noting at this point that one of the facilities provided within the ACP editor (as a pull-down menu option) is a pop-up static design display window, ie. without editing facilities. This can be used to display either the context of the subsystem currently being edited, or the top level of the ACP diagram.

As a MASCOT design is being constructed, checks are made within the editor to ensure that the diagram *syntax* is correct, for example that no two Channels or Pools are directly connected. At present, these checks are limited to simple syntax checking procedures and no attempt has been made to ensure more general consistency throughout the design. It is intended that separate tools for carrying out more extensive checking procedures will be incorporated at a later date.

3.3 The State Transition Diagram editor

The form of interface provided for the STD editor (**STDed**) follows the same basic philosophy as that of the ACP diagram editor, the main difference being that it is based on the graphical STD design representation [22]. It makes use of the same form of *context-sensitive pop-up menu* as the ACP editor, but has no pull-down menus.

Invocation of the STD editor is achieved through one of the options in the pop-up menus attached to the relevant design objects in the ACP diagram editor. By using this approach STDs are directly associated with the appropriate MASCOT system element and it also integrates of the two design editing tools in a logical manner. In effect, both editors will appear to the user to operate as a single tool.

3.4 The Design Transformation Tool

The Design Transformation Tool (**M2M**) transforms a MASCOT design representation into that of CSP/*me too* [9, 8]. This is an extension of the *me too* method of software design [4] and was developed by the DESCARTES Esprit Project, primarily for the design and development of embedded Ada systems. The *me too* method is concerned with the specification and development of sequential software systems and has been extended by the addition of CSP [14] to enable it to deal with concurrent systems.

Briefly, each MASCOT system element (ie Activity, Channel, Pool and Server) is translated into a CSP process and the data flows are translated into CSP events. For example, the Channel *Chan_1* in Figure 3 would be represented in CSP by:

```
Chan_1 = (     (sig_1_out  →  Chan_1)
            [] (sig_2_out  →  Chan_1)
            [] (new_sig  →  (Chan_1_reset  →  Chan_1))
```

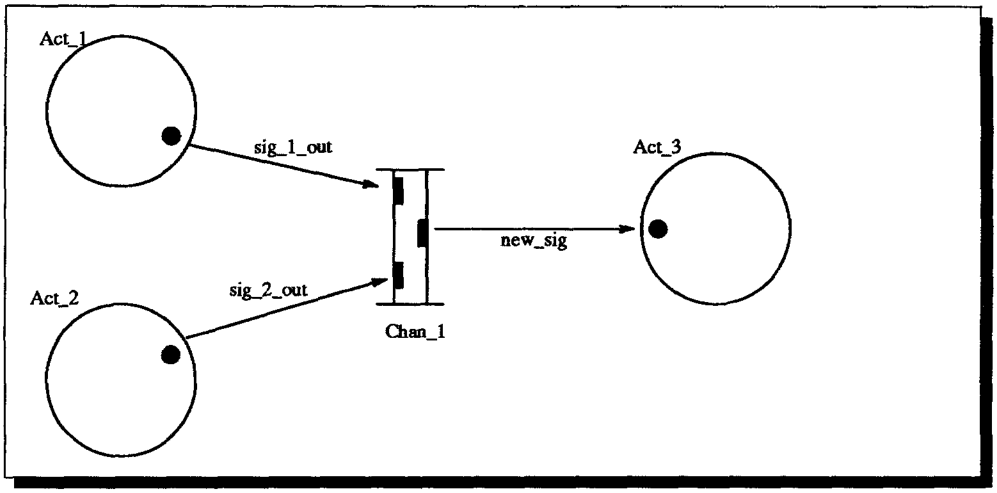

Figure 3: Section of a MASCOT ACP Diagram

Each of these CSP processes is represented by a *me too* module which has its own local state, defined in terms of sets, sequences,maps relations, etc., and which exports operations on that state. These operations are formally specified by the natural mathematical operations allowed on the forementioned structures. In effect, *me too* describes each of these processes in terms of its state and the operations on that state, while CSP constrains the ordering of these operations. These processes are mapped on to physical processors and expected execution times are allocated to the CSP events. The default processor/time configuration, as generated by the **M2M** tool, can be easily modified manually if required.

Once a complete CSP/*me too* specification has been obtained, it is then used by the CSP/*me too* interpreter to model the behaviour of the design using a 'state machine' representation. This specification can be used to experiment with the effects of different process to processor configurations and event execution times.

This process of design transformation and execution of the prototype is discussed more extensively in [11], where there is an appendix providing a fuller worked example.

3.5 The Design Advisor

The Design `Advisor` is a rule-based design assessment tool which is used to comment on the static structures of the design. It is currently invoked from one of the pull-down menu options within the ACP diagram editor, and provides a report based on the current state of the MASCOT design.

The role of a tool of this form was explored in the MDSE project. In this, we used a Knowledge-Based 'Design Advisor' that incorporated three classes of rules:

1. Rules based upon the MASCOT syntax and semantics.
2. Rules based upon design theory and upon ideas about quality metrics.

3. Heuristic rules that were based upon experience with the use of MASCOT ('rules of thumb').

Since we view the third class as essentially being used to provide a bound upon the 'solution space' available to a designer, we prefer to term these *constraints*. A fuller discussion of these rules is given in [5].

The use of a general-purpose Expert System shell in the MDSE did not prove to be an efficient way of constructing such a tool. For the `Advisor` used in the workbench we have therefore used a more simple rule-based approach. So far, we have translated about half of the MDSE rule-set into this form, and it will shortly be extended to include assessment rules which can make use of the information provided in the STD descriptions of objects.

The inclusion of the `Advisor` provides a good example of the benefits that can be obtained from using the distributed architecture of the workbench. The relatively monolithic structure of the MDSE made it difficult to extend the rule-set of the `Advisor` because of the manner in which it was integrated into the MDSE toolset. (In particular, the MDSE had no provision by which any facility for capturing the state-transition information could easily be added to the design model.) By contrast, the workbench structure makes it relatively easy to add and integrate any further knowledge elicitation tools that might be needed for the development of the `Advisor`, and to add further structures to the *DOB* to support this if necessary.

3.6 The Design Object Base

The *DOB* is a database, in the loosest sense, which can be accessed by each of the workbench tools. It holds all of the information available for each of the design *objects*, or components, and it provides a set of access procedures which conceal the internal form of the *DOB* and act as the *object base* interface. The term *object base* is used in preference to the term 'database' as the latter carries with it many implicit implications of structure and characteristics which we are not concerned with at this point. Similarly, we have refrained from using the term 'Object Oriented' as it is increasingly being used, with varying interpretations, and hence is liable to lead to misunderstandings!

At present the *object base* is a simple text file which is divided up into four main sections, each of which contains its own type of data record. These sections are outlined below:

Access Interfaces The intention was that there should be a record that would correspond to each *Access Interface* within the MASCOT ACP diagram. As yet, the *Access Interface* feature has not been fully implemented, and so this section of the object base is currently not used by the workbench tools. It was included as part of the initial object base implementation, because it forms part of the MASCOT standard and was implemented in the MDSE, and so has been included here for completeness.

Templates A template record is normally associated with a MASCOT Subsystem and contains references to each of the Subsystem's components, including the *doors* which act as its external interface. The use of this feature differs somewhat from that specified by the MASCOT standard, where templates are used to facilitate the reuse of design components.

Objects Each record of this type corresponds to an individual component within the design. As well as the particular attributes of a given object, each object record also holds the information required to reconstruct the ACP and STD design forms. This includes information such as the data connections it is associated with, its parent object, and any subcomponents it might possess. Conceivably, an object record could contain any information (or a pointer to it) relating to that particular object, for example design notes or even outline implementation code. Tools to utilise this potential can easily be integrated into the workbench as has already been demonstrated in section 3.2.

Indexes The main purpose of the records in this section is to provide a means of referencing the various objects within the object base.

As the EDW toolset expands and the demands placed on the object base increase, it will probably become necessary to review its structure and form and to extend this to make use of more complete database management techniques.

3.7 Using the workbench

The EDW is started up as a single process which initialises the *DOB* and reads in a design, if one was specified on the command line (the *Workbench Start Up* process in Figure 1). Then, depending on the other command line arguments, either the ACP diagram editor or the Design Transformation tool is invoked. Ideally, the EDW start up process should provide a menu from which the required tool can be selected. Such a menu would be displayed continuously while the workbench was active, therefore enabling the user to invoke another tool while one is already being used, for example running the Design Transformation tool while the ACP diagram editor is being used.

4 Future Developments

In the preceding sections, we have described the general principles behind the use of the Design Transformation Technique, as well as the structure of the prototype design support environment (the EDW) that has been developed around this. In this section, we briefly discuss some of our ideas for further work, and in particular, we consider how these might be developed within the framework of the Experimental Designer's Workbench.

In terms of future developments in the use of Design Transformations, we have identified four fairly major themes that can usefully be investigated with the aid of the EDW. These are:

Verification of a design by evaluating it against the requirements specification.

Resource modelling in order to predict the processing requirements of the final system as it will be implemented.

Design Refinement by providing facilities that can be used to expand the *me too* structures in a controlled manner (including checking for consistency with the original design in systematic form).

Knowledge-Based Support can potentially be incorporated into the transformation process, making it possible to use transformation rules that are more context-sensitive in terms of the problem domain.

The next two sub-sections describe the first two of these ideas in a little more detail.

4.1 Design Verification

Verification is concerned with identifying whether the system as designed will meet the requirements originally specified by the customer ("are we building the system right"?). A major problem for verification lies in the very different viewpoints involved in the descriptive forms that are used to record requirements specifications and design specifications, since:

- the requirements documents are concerned with *problem*-centred issues;
- the design documents are concerned with *solution*-centred issues.

Not surprisingly, these often involve the use of rather different viewpoints. In particular, the requirements documents are concerned with describing required system *behaviour*, whereas design documentation is generally concerned with recording the *structure* of the intended system.

The distinctive feature of the design representation that is generated from our Design Transformation tool is that it is executable, and hence it can be used to model the behaviour of the system for particular scenarios. In matching the outputs from the modelling to the required behaviour specified in the requirements documents, the problem which needs to be resolved is how to match the information output from the *me too* interpreter to the forms used in requirements descriptions.

We are currently investigating this topic using a similar strategy to that used in testing code (although the context and form are very different). Our aim is to be able to use a requirements specification document to generate:

- the sequence of events;
- the predictions of behaviour;

for a number of scenarios, using some suitably prescribed format, and then to use the first of these to generate the inputs for the CSP/*me too* model. The outputs from executing the model will then be compared with the predicted results, in order to provide some basic degree of verification.

While the approach is not rigorous in any sense (at present), and inherits from testing techniques the problem of selecting suitable scenarios, we believe that it has considerable potential. In particular, it unifies the forms of description used for both specification and design, and hence makes some degree of comparison possible.

4.2 Resource Modelling

An important feature of the original MDSE system [7], was that it included a resource modelling facility that allowed a designer to produce predictions of likely cpu usage of the resulting system. This was considered to be particularly valuable for large systems and embedded systems, where such predictions are extremely desirable in terms of determining final configurations for systems. However, producing a model involved generating skeleton Ada code, which then had to be compiled and executed in order to generate the required predictions.

The availability of the high level prototype of the system means that by adding the facilities to attach resource usage estimates to the design objects, the 'execution' of a prototype can be used to predict the consumption of cpu and other resources in the final system. As with the original MDSE toolset, this can be parameterised to take account of a number of different models for the implementation form (for example, processor types, kernal characteristics and process to processor mappings). In particular, the model is easy to change, and can be refined in parallel with the development of the final design.

Adding this feature is likely to require the integration of a further form of editor (or making an extension to the ACP editor), together with some additions to the *me too* interpreter. There may also be some potential for including this performance information in the verification process described in the previous sub-section.

4.3 Refinements to the Designer's Workbench

The EDW itself is also undergoing continuing experiment and refinement, and in particular we are planning to:

- provide a *design animation* facility that will allow the execution of the *me too* design to be described in terms of the original systematic design forms;
- add checking mechanisms to provide consistency checking between the different viewpoints of an object and the expanded descriptions of objects;
- improve the user interface, so that the designer is not constrained by the tool in terms of how they manage the design process;
- adding a 'notes' facility that will allow the designer to append notes to design objects (the use of note-making is an observed practice of designers [3] and it is important to provide this to allow designers to develop the design in an opportunistic form [23]);
- provide a more refined means for saving and restoring designs.

Taken together, these will turn the workbench into a prototype that can be used for demonstrating our ideas on a reasonably large scale problem.

4.4 Related Developments

The EDW described in this paper is in many ways a prototype for use in exploring ideas about design support tools. However, even in this form it exhibits elements of many of the forms of tool integration defined by Wasserman [17]. (Platform Integration, Presentation Integration and Process Integration are certainly included, Data Integration is a lengthier objective, and only Control Integration is really lacking at this stage of development.) In this concluding section we briefly describe some related areas of work that are likely to lead to a more fully integrated system.

In the longer term, we are seeking to make use of our experiences with using the Workbench in order to identify the particular features of our present representation forms that make it possible to perform the various tasks required by the techniques that have been described in this paper. By associating 'methods' (or properties) with 'objects' in this manner, we can then seek to generalise the support for these techniques within the work of the GOOSE project (Generalised Object Oriented Support Environment) [2]. This is an externally funded project which will seek to take a more generalised view of design while trying to provide broadly similar facilities to those described here.

A related thread of work is connected with extending and developing our use of knowledge-based techniques in both the areas of static design analysis (as typified by the MDSE *Advisor* tool) and also of the Design Transformation Technique. Again, we hope to be able to generalise these ideas and to incorporate them into the GOOSE environment.

In that sense, the EDW is providing a valuable prototype that is operating within a specialised domain, which can then be used to help us construct a prototype for a more general design support environment.

We have argued elsewhere that the software design support environments of the next generation will need to contain tools that do more than just record a designer's ideas and check them for syntactic consistency [6]. A similar view is reflected in the work described in [19]. The work described in this paper is a further step in this direction, and it has already begun to demonstrate the potential benefits of using well-structured transformations for the development of a design.

Acknowledgements

Many people have contributed to the work that is described in this paper. Thanks are particularly due to Cor Yong Thed, who built the first prototype elements during his MSc project, and to Dr Mustafa Marashi and Mr Sam Nelson who provided ideas and contributions to the design and implementation of the workbench. Dr Robert Clark provided help with the development of the Design Transformation Tool. The support of SERC through the provision of a Research Studentship is also acknowledged.

References

[1] Special Issue on MASCOT. *Software Engineering Journal*, Vol. 1(No. 3), May 1986.

[2] Generalised object oriented support environment (goose), 1991. Three year research project.

[3] B Adelson and E Soloway. "The Role of Domain Experience in Software Design". *"IEEE Transactions on Software Engineering"*, SE-11(11):1351–1360, November 1985.

[4] H Alexander and Val Jones. *Software Design and Prototyping using me too*. Prentice-Hall, 1990.

[5] David Budgen and Mustafa Marashi. "MDSE Advisor: Knowledge based techniques applied to software design assessment.". *Knowledge Based Systems*, Vol 1(No. 4), September 1988.

[6] David Budgen and Mustafa Marashi. "Knowledge Use in Software Design". In Kathy Spurr and Paul Layzell, editors, *CASE on Trial*, pages 163–179. John Wiley, 1990.

[7] J A Chattam, R K James, H Patel, D Budgen, A G O'Brian, and M J Looney. MASCOT design support environment - final report. Technical Report MDSE/GEN/FR/1.1, Alvey Project SE/044, March 1989.

[8] Robert G Clark. The csp/me too method. Technical Report TR.61, Department of Computing Science, University of Stirling, May 1990.

[9] Robert G Clark. "The Design and Development of Embedded Ada Systems". *Software Engineering Journal*, Vol. 5(No. 3):175–184, 1990.

[10] N. Cross, editor. *Developments in Design Methodology*. John Wiley, 1984.

[11] Grant Friel and David Budgen. "Design Transformation and Abstract Design Prototyping". *Information and Software Technology*, November 1991.

[12] Neil Haddley and Ian Sommerville. Integrated support for systems design. *Software Engineering Journal*, pages 331–338, 1990.

[13] K.L. Heninger. "Software Requirements for Complex Systems: New Techniques and their Applications". *IEEE Transactions on Software Engineering*, SE-6:2–13, January 1980.

[14] C.A.R. Hoare. *Communicating Sequential Processes*. International Series in Computing Science. Prentice Hall, 1985.

[15] J. Christopher Jones. *Design Methods: seeds of human futures*. Wiley Interscience, 1970.

[16] Lawrence J.Peters. *Software Design : Methods and Techniques*. Yourdon Press, 1981.

[17] Anthony J.Wasserman. "Tool Integration in Software Engineering Environments". In Fred Long, editor, *Software Engineering Environments*. Springer Verlag : Lecture Notes in Computer Science No. 467, 1990.

[18] S. Patel, R.A. Orr, M.T. Norris, and D.W. Bustard. "Tools to Support Formal Methods". In *Proccedings of 11th International Conference on Software Engineering*. IEEE/ACM/CMU, May 1989.

[19] R.J.A.Buhr, Gerald M.Karam, C.J.Hayes, and C.M.Woodside. "Software CAD: A Revolutionary Approach". *IEEE Transactions on Software Engineering*, Vol. 15(No. 3):235–249, March 1989.

[20] Cor Yong Thed. An experimental designer's workbench. Master's thesis, Department of Computing Science, University of Stirling, Stirling, Scotland, 1991.

[21] K.J. Turner. "Towards better specifications". *ICL Technical Journal*, pages 33–49, May 1984.

[22] Paul T.Ward and Stephen J.Mellor. *Structured Development for Real-Time Systems*, volume 1 : Introduction & Tools. Yourdon Press, 1985.

[23] W Visser and J-M Hoc. *Expert Software Design Strategies*, pages 235–249. Academic Press, 1990.

[24] Pamela Zave. "The Operational Versus the Conventional Approach to Software Development". *Communications of the ACM*, Vol. 27(No. 2):104–118, February 1984.