The Software Architect's Assistant - A Visual Environment for Distributed Programming

Keng Ng Jeff Kramer Jeff Magee Naranker Dulay

Department of Computing, Imperial College,
180 Queen's Gate, London, SW7 2BZ, UK
Email: {kn, jk, jnm, nd}@doc.ic.ac.uk.

Abstract

This paper describes current work on the application of visual techniques to the design and construction of parallel and distributed programs. In particular, we look at how the software architectural view can be effectively utilised to provide a common framework for integrating the various software development activities, ranging from early, informal program design to the evolution of the running program.

A prototype visual programming environment - the Software Architect's Assistant - has been built for the design and development of Regis distributed programs. It provides the user with automated, intelligent assistance throughout the software design process. Facilities provided include the display of integrated graphical and textual views, a flexible mechanism for recording design information and the automatic generation of program code and formatted reports from design diagrams. Software reuse is also supported through the use of component libraries. Support for graphical monitoring and management of running programs, currently provided by a complementary tool, will be integrated into the environment to provide a complete solution for visual distributed programming.

1 Introduction

The distributed systems community has reached some consensus on the recommended set of techniques and mechanisms for constructing distributed systems. Among the mechanisms agreed as useful are communication by remote procedure call, atomic group cast communication between server replicas to support availability and atomic actions to preserve data consistency in the presence of failures. This consensus is reflected in the inclusion of some of the above in commercially available distributed applications platforms such as ANSA [1] and OSF/DCE. However, none of the above mechanisms help in the design process of decomposing an overall application into a set of components nor in its subsequent construction. They are rather the "glue" used to compose components such that they may communicate and interact. What is missing is any notion of structure. Support is generally missing for the design and construction of applications exhibiting a structure which is any more complex than a simple client-server arrangement.

The premise of our approach is that a separate, explicit structural (configuration) description is essential for all phases in the software development process for distributed systems, from system specification as a configuration of component specifications to evolution as changes to a system configuration. Descriptions of the constituent software components and their interconnection patterns provide a clear and concise level at which to specify and design systems, and can be used directly by construction tools to generate the system itself. In many cases - particularly embedded applications - it is the structure of the application itself which is used to dictate the structure of the resultant system.

We use the neutral term "component" to mean a software entity which encapsulates some resources and provides a well defined interface in terms of the operations it provides to access the resources and the operations it requires to implement its functionality. Further, we require our components to be "context independent" in that they use only local names to communicate with their environment, thereby allowing them to be developed independently of the context in which they execute. Context independence makes it easy to plug a component into different programs since it specifies both the communication objects required as well as those provided.

This paper outlines current research on the provision of such an architectural framework for the engineering of parallel and distributed systems. In the following sections, we describe a constructive approach for designing and constructing distributed programs, followed by an introduction to the Regis environment for component-based distributed programming. An interactive graphical environment for the design and development of Regis programs - the Software Architect's Assistant - is then presented together with a case study to illustrate its use.

1060-3425/95 $4.00 © 1995 IEEE

2 Constructive Program Development

2.1 Design

In contrast to the 'specification-driven' approach, our approach to distributed systems design is 'constructive'. The specification-driven approach attempts to formalise the decomposition process based only on the system specification. We believe that the process of component identification remains informal as it requires design information not usually included in the system specification. Decomposition is best dealt with through design heuristics with the aid of more rigorous analysis on properties such as behaviour and performance. Emphasis should be placed on the validation process using analysis through composition of component behaviours analogous to "construction" of the system from components. This constructive approach is the means by which we gain confidence that our design is satisfactory. Our design approach is thus not restricted to a specific design method or technique such as SASD or OOD in the sense of enforcing those specific rules and method steps. Rather it supports a general approach to design consisting of the following general design activities:

• *Structure and Component Identification*: Initial design aims to identify the main processing components and produce a structural description indicating the main data flows.
• *Interface Specification*: This aims at introducing control (synchronisation) between components and refining the configuration, component interface specifications (intercommunication) and component descriptions accordingly.
• *Component Elaboration* consists of elaboration of the component types, either by hierarchical decomposition of composite component types into a configuration of sub-components, or by functional description of behaviour (formal or informal specifications). Primitive components must also be provided with implementation (code) descriptions. As before, the identification of common component types is emphasised.

Although this description emphasises the top-down approach, bottom-up (constructive) composition and component reuse can be used at any stage [2].

2.2 Construction

The design activity culminates in a structural description of the desired system and a set of primitive component types described in a programming language. From these, the executing distributed system can be constructed by invoking the appropriate compilation, linking and loading tools. Central to the construction activity is the structural description. It can be annotated with non-functional information such as location, availability and resource requirements during the design process to direct the construction phase.

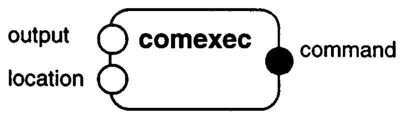
2.3 Analysis

In order to exploit the central architectural view, we adopt compositional techniques analogous to system construction except that the entities being manipulated are not software components but associated attributes giving specifications or models of their functional or timing behaviour. The objective is to ensure that the structure of the system and of its specification is the same. System architects should be free to select those specification techniques that are appropriate to the application. Currently we have focused on the use of Milner's work on the π-calculus [3] as a means for defining the semantics of our configuration language. More recently, we have been working on the use of labelled transition systems (similar to state transition systems) to describe component behaviour [4, 5]. Reachability analysis is then provided using an approximate but tractable technique for flow analysis of distributed programs and an improved exhaustive technique for compositional analysis. Both techniques are supported by automated software tools.

The integration of these 3 phases of software development through the Software Architect's Assistant is summarised by the diagram in figure 3.

3 The Regis Distributed System

Regis is a programming environment which supports the configuration approach [6] to the design, construction and maintenance of parallel and distributed systems. The configuration approach dictates a clear separation between the description of program structure and the specification of the component behaviours which constitute that program. The approach has been successfully used by both the authors and others in programming environments for distributed systems (Conic, Rex, Polylith etc.). Regis inherits from these predecessors the use of a separate configuration language and context independent components. However, it permits the design and construction of a much less restricted class of program architectures than its predecessors through its support for dynamic structures and its flexibility in accommodating a wide range of component interaction mechanisms [7]. Programs in Regis are constructed from interconnected instances of component types. The behaviour of a component type is implemented in the C++ programming language. This paper is concerned largely with the structural aspects of program design and construction and consequently we will not deal with component level programming (details may be found in [7]). Program structures in Regis are described either graphically as outlined later in this paper, or textually in the Darwin configuration language [8]. An example of the structural view of a component is depicted in both these forms in Figure 1. Examples in this paper are taken from an Active Badge system [9] implemented by one of the authors in the Regis programming environment. Active Badges emit and receive infrared signals which are received/transmitted by a network of infrared sensors connected to workstations. The system permits the location and paging of badge wearers within a building.

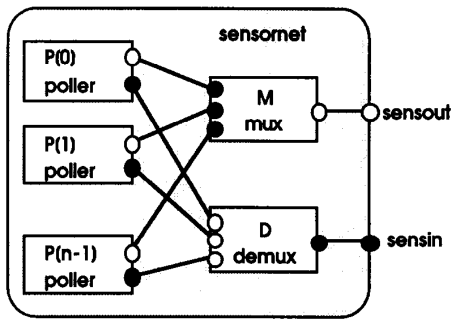


```
component comexec {
   require location <event bstatus>;
           output   <port smsg>;
   provide command  <entry comT repT>;
}
```

Fig. 1 Component Type

The component of Figure 1 provides one service (the filled in circle) and requires two external services to implement that service (the empty circles). The service provided is badge command execution. Commands are issued to badges to set off its internal beeper or to illuminate LEDs. The Darwin component interface specification specifies the set of services required and provided by a component type together with types of these services (enclosed in angle brackets). By convention, the first word of the type specification is the interaction mechanism class. For example, command accepts entry calls with a request of type *comT* and a reply of *repT*. To execute a command, it is necessary to first locate a badge. Consequently, the command execution component requires the location service. Location information in the badge system is an event stream where an event represents a change of badge location. Consequently, the interaction mechanism for location is *event* and the data type of each event is *bstatus*. Similarly to execute the badge once found, the component must send a message to the sensor network. The requirement for this service is represented by *output* which uses the Regis port message transmission primitives. Note that the component *comexec* does not need to know the names of external services or where they may be found. It may be implemented and tested independently of the rest of the badge system.

Systems in Regis are constructed by composing component instances. The composition binds services required by component instances to the services provided by other instances. The composition may be treated as a single component type at higher levels of system description. Figure 2 is an example of a composite component. It controls the interface to the network of infrared badge sensors. Each requirement (empty circle) in this example is for a port (named *output*) to send messages to, and each provision (filled in circle) is a port from which a component receives messages (named *input*). Requirements which cannot be satisfied inside the component can be made visible at a higher level by binding them to an interface requirement as has been done in the



```
component sensornet( int n) {
   provide sensin   <port smsg>;
   require  sensout <port smsg>;

   array P[n]:poller;
   inst
       M:mux;
       D:demux;
   forall i:0..n-1 {
       inst P[i] @ i+1;
       bind
           P[i].output -- M.input[i];
           D.output[i] -- P[i].input;
   }
   bind
      M.output -- sensout;
      sensin   -- D.input;
}
```

Fig. 2 Composite Component Type

example for multiplexer *M* requirement output which is bound to *sensout*. Similarly services provided internally which are required outside are bound to an interface service provision e.g. *sensin--D.input*.

From the example, it can be seen that components may be parameterised and that parameters can be used to determine the internal structure of composite components. In this case the parameter determines the number of poller instances. In addition to replicated structures, conditional, recursive and dynamic structures may be defined. Services may be exported to and imported from an external name service. Bindings may be made by a third-part (configuration manager) dynamically at runtime. The visual design tool supports all these forms and is described below.

4 The Software Architect's Assistant

The Software Architect's Assistant is a program development environment which supports the design and construction of Regis distributed programs. It has a highly visual and interactive user interface, and provides a framework in which software design and associated information can be captured, viewed and modified easily and quickly. Program source code in the form of Darwin code for composite components and C++ code skeletons for primitive components are generated automatically from the program design. The Assistant has the capability to

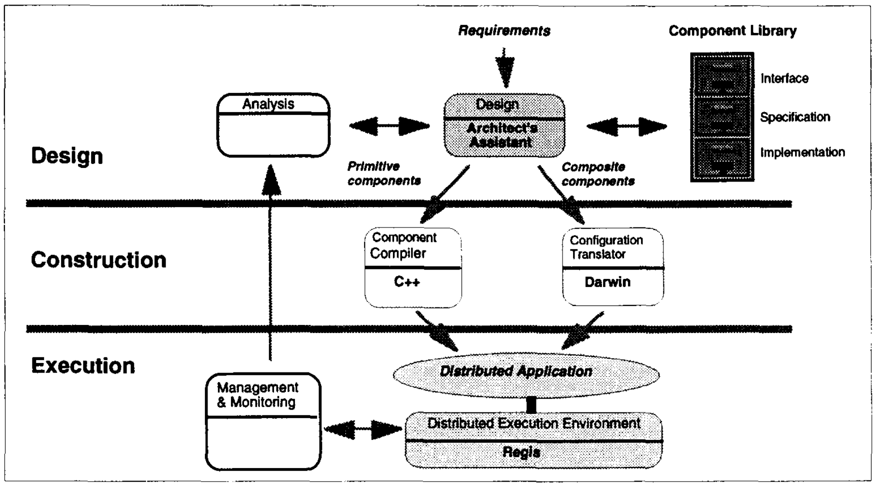


Fig. 3 Integration using the Architect's Assistant

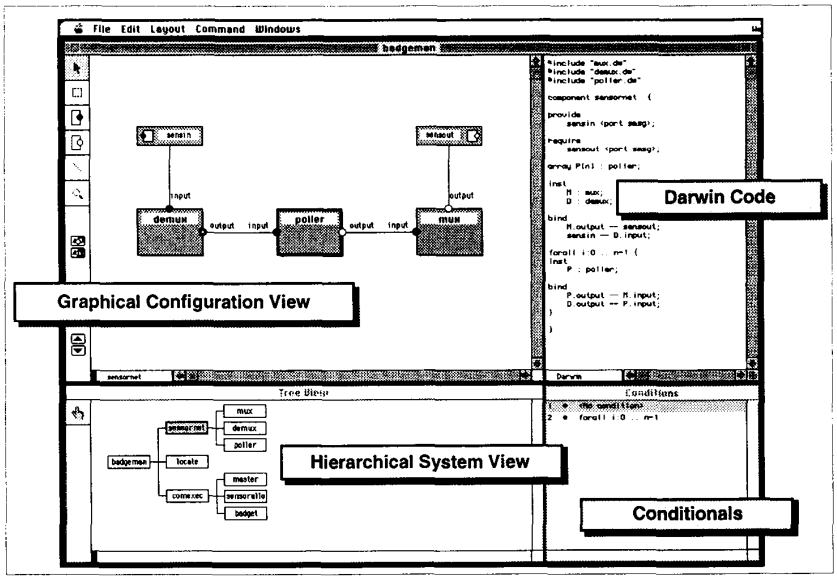
software architecture in the configuration programming paradigm [11]. Furthermore, given that for many applications the basic software structure is fairly stable throughout the development process, it also provides an ideal framework for integrating the various software development and management activities and facilitates the presentation of a uniform and consistent user interface.

manipulate structural (Darwin) descriptions in both graphical and textual forms and is intended to transform one to the other (currently only graphical to textual). Its functionality can be split into three complementary and interacting areas. These are support for design, analysis, and construction. The Assistant is intended to act as a front end for specifying the structure and those component specification attributes selected by the designer. It should then be possible to invoke the relevant compositional analysis or verification tools associated with any or all of the provided attributes (figure 3).

One of the novel features of the Assistant is the provision of intelligent, automated assistance throughout the design process, including facilities for the stepwise refinement of program design, automated layout aids to simplify the task of diagram manipulation and support for navigating and traversing through complex hierarchies. Software reuse is also supported through the use of component libraries.

The user interface of the Assistant is based on the theme of structural visualisation. This emphasis is especially relevant in the case of distributed programs where much of the underlying conceptual constructs are topological in nature, and can be naturally captured and presented in a graphical form [10]. It is also compatible with the user's mental model of his program, and reflects the strong emphasis placed on the

The Assistant allows the software architecture to be viewed from multiple integrated views (figure 4). The Configuration Window contains a sketchpad in which the program architecture can be mapped out using the appropriate tools from the tool palette. The sketchpad displays the graphical configuration view of Darwin, and is where all editing of the program structure takes place. Controls are provided for traversing the hierarchy within this view. Multiple configuration windows can be opened to allow the viewing of different parts of the program at the same time. Each of these can in turn be split into 2 resizable viewing panes, with the right hand pane used for displaying, among other information, the Darwin code corresponding to the diagram being drawn. This code is read-only and is updated on the fly as the diagram is modified.



```
#include "mux.dw"
#include "demux.dw"
#include "poller.dw"

component sensornet  {

provide
    sensin <port smsg>;

require
    sensout <port smsg>;

array P[n] : poller;

inst
    M : mux;
    D : demux;

bind
    M.output — sensout;
    sensin — D.input;

forall i:0 .. n-1 {
inst
    P : poller;

bind
    P.output — M.input;
    D.output -- P.input;
}

}
```

```
1  ◆  <No condition>
2  ●  forall i:0 .. n-1
```

Fig. 4 The Main Windows of the Architect's Assistant

While each sketchpad allows the display and editing of a single component at a time, the associated Tree Window presents the entire program in a hierarchical tree structure. This is a type hierarchy which shows all the component types used in the program and the 'include' relationship between them. It is updated automatically whenever the program structure is modified in the sketchpad. It is useful not only as an indicator of the overall program structure but can also be used as a navigational aid for browsing the different parts of the system - selecting the appropriate type in this view will bring up the structure of that component type in the sketchpad. In addition, it serves as a 'where am I' indicator by highlighting the part of the tree diagram which corresponds to the component being displayed in the front-most sketchpad.

The Condition Window enables the creation and modification of Darwin conditional guards for conditional components and/or bindings used in the component that is displayed in the front-most sketchpad.

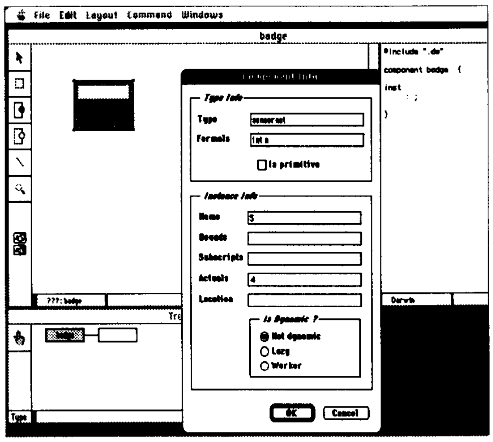
In the following sections, a simple case study - the active badge program described earlier - is used to highlight the main facilities of the Assistant and to illustrate the typical usage of the tool in the course of designing and constructing a Regis program.

4.1 Program Design with the Architect's Assistant

The principal activity at the design stage entails the identification and elaboration of the main processing components within the overall program. This is carried out within the Assistant's sketchpad which is an 'intelligent' diagram editor with built-in knowledge of the Darwin syntax. Therefore only the entry of design diagrams which correspond to legal Darwin code are allowed. The user is alerted with a warning beep whenever an illegal operation is attempted. Furthermore, everytime the diagram is modified, a built-in tidy-up algorithm is invoked on-the-fly to minimise the crossovers of lines. This helps to alleviate the tedium commonly associated with visual programming.

When drawing in the sketchpad, the designer is essentially defining a component type in terms of instances of components and the bindings between them. Initially, the Assistant presents a blank or undefined type which corresponds to the 'root' of the program we are building. We have named the system 'badge', as indicated by the name of the window. The badge component type is elaborated by sketching out its internal structure within this window. A component is created by drawing a box with the component tool. Information related to this component, such as its type and instance name, array bounds, actual and formal parameters, is entered through a dialog box (figure 5). Any of this information can be left undefined initially and filled in later.

A binding between two components is specified by drawing a line from the component requiring a service to



Type Info
Type: sensornet
Formals: int n
☐ Is primitive

Instance Info
Name: S
Bounds:
Subscripts:
Actuals: 4
Location:

Is Dynamic ?
● Not dynamic
○ Lazy
○ Worker

```
#include ".dw"
component badge {
inst
}
```

Fig. 5 Entering Component Information

the one providing the service. The appropriate communication objects are automatically created on the components at each end of the binding. These snap to the appropriate sides of the components and are automatically positioned to avoid any crossovers with existing bindings. A communication object can also be drawn independently by clicking inside the appropriate component with the provide or require tool. It will appear on the face of the component nearest to where the mouse was clicked. Figure 6 below shows the badge component type after its has been fully defined. Composite components are drawn as shadowed boxes, as opposed to plain boxes for primitive components. The thin rectangular boxes (e.g. location) are the open systems interfaces of the badge system which enable communication with external programs, and are to be registered with a name server [12]. As can be seen from the figure, the formatted Darwin code for badge has been generated from the design diagram.

Elaboration of the composite component *sensornet* is done by first double-clicking on its box. This drops the user into that component whose internal structure is initially empty, apart from its interface which is represented by the interface boxes *sensin* and *sensout*. These are generated automatically by the Assistant based on what has previously been defined at the level above. For the *sensornet* component we would like to reuse the component type *tcp* which had been previously defined in a different program. This can be done by calling up the library browser which presents a list of all component types defined in a program together with their corresponding diagrams and descriptions. Using the standard copy-and-paste technique, any component can be copied from the library and included into the badge program under development. By reusing a component, we pick up not only its diagram and code but also any additional information associated with it. Once added, multiple instances of *tcp* can be created by choosing the Duplicate command.

An alternative way of reusing pre-defined components is to import a textual Darwin program file using the Darwin translator built into the Assistant. This facility enables Darwin components written outside of the Assistant environment to be incorporated into the current program. An instance of the imported type will be added to the component being displayed in the sketchpad. The automated layout algorithms in the Assistant (see section 5.1) ensures that a clean, readable diagram is produced which can be further edited by the user. This import facility is also useful simply for generating Darwin configuration diagrams out of old programs.

4.2 Program Construction

Primitive components in Regis such as *mux* are programmed in C++. In the Assistant, when a component is specified as a primitive in its Info dialog (figure 5), a skeleton C++ code in the form of its class constructor is generated automatically. This can be expanded upon by choosing *Implementation* from the pop-up menu under the right-hand pane of a sketchpad. This switches the pane over to display its C++ code instead of its Darwin code. The text editor provided in this pane can then be used to complete the C++ implementation of the component.

In order to compile and execute the badge program just entered, we first need to save it to disk. This produces a project file which records all information that has so far been entered, as well as individual .dw and .cc files for the appropriate components. These files can then be picked up by the Darwin and C++ compilers directly for compilation. If further modifications are made to the program within the Assistant, only the files of modified components will be saved. This incremental save ensures that no unnecessary recompilation is done when the *Make* tool is next invoked.

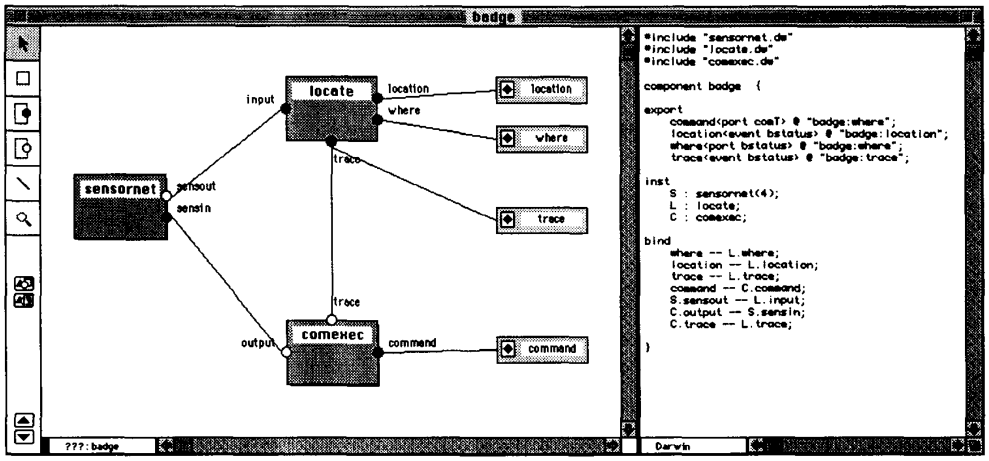
5 Salient Features of the Architect's Assistant

Our overall objective in developing the Assistant was to provide a design environment which is 'fun' to work in. This entails an environment which is flexible and forgiving, which automates all mundane clerical tasks and has good performance.

Flexibility. We have deliberately made the use of the Assistant very flexible and informal by not imposing any unnecessary restrictions on when information should be entered, or the order in which things should be done. The aim is not to enforce strict conformance to any pre-defined workplan, but to allow the user to work in the style that he is comfortable with. For instance, there is no requirement to complete the specification of one component before starting on another. The user is free to leave any part of a program partially defined and return to it at a latter date. This makes it easier and quicker to record ideas and information in a more ad-hoc fashion. In addition, unlike many other CASE tools which impose a strict top-down decomposition approach, support is provided in the Assistant for the design of a program to be carried out in a top-down, bottom-up or middle-out fashion, or even a combination of the three. This, when combined with the facility for component reuse, makes for a very flexible and powerful design environment.

Given that any software design inevitably needs to go through several iterations and refinements, we have been careful to ensure that any decisions made can be easily undone. This not only reduces the effort required in error-recovery, but also helps to encourage the designer to explore alternative design solutions. The ability to alter the compositional structure of components through drag-and-drop (see section 5.5), the commands for flipping the directionality of communication objects and the auto-propagation of data types are examples of this.

Automation. A software tool, like any other tool, is useful only if it increases the productivity of the user. One of the



```
#include "sensornet.dw"
#include "locate.dw"
#include "comexec.dw"

component badge  {

export
    command<port comT> @ "badge:where";
    location<event bstatus> @ "badge:location";
    where<port bstatus> @ "badge:where";
    trace<event bstatus> @ "badge:trace";

inst
    S : sensornet(4);
    L : locate;
    C : comexec;

bind
    where -- L.where;
    location -- L.location;
    trace -- L.trace;
    command -- C.command;
    S.sensout -- L.input;
    C.output -- S.sensin;
    C.trace -- L.trace;

}
```

Fig. 6 The top-level badge component

most effective way of achieving this is to exploit the computer's processing power for tasks which would otherwise have required human intervention. Our philosophy has been that any task that can be performed by the tool should be automated, particularly tedious mechanical activities that can be accomplished much quicker by the tool and with fewer errors. To this end, the Assistant has provided two levels of automatic drawing of configuration diagrams, automatic tree drawing of the program hierarchy, code generation and various syntactic and type checking. In general, any required data that can be inferred from existing data is automatically generated. This enables the user to concentrate more effort on the creative aspects of program design and construction, and less on routine clerical activities. Automation, of course, also has the desirable side-effect of reducing human errors.

The Assistant further ensures that consistency is always maintained across the different parts of the program at all times. If, for instance, the definition of a component's interface is altered in one part of the program, the change will be reflected immediately in all instances of that type. Similarly, type-checking is performed whenever a binding is drawn to connect two communication objects. If there was a type clash, the user is given the option of either cancelling the operation, or copy the 'correct' type from one to another. If the last option was chosen, this information will also be propagated to all other connected objects throughout the program hierarchy. If the object type is subsequently changed, this new information will again be picked up automatically by all connected objects.

Performance. Speed of performance directly affects the usability of a tool. While it may not be as crucial to programs designed to run in batch mode, it is a key requirement of any interactive program as the difference in response time of a fraction of a second is often perceptible to the user. A sluggish tool does not only aggravate user frustration, but tends to break up the user's flow of thought and concentration. Considerable care has therefore been taken in the development of the Assistant to ensure that it is responsive to the user at all times. This is all the more crucial because of the high level of automation provided which might potentially slow down its performance, such as on-the-fly diagram tidy, tree diagram updating, Darwin code generation and syntax and error checking, each of which is potentially invoked everytime a diagram is modified.

In the following sections we describe some of the more specific features of the Assistant which have not been covered in its description above.

5.1 Graphical Layout Support

A comprehensive set of automated and manual aids is provided for the editing of Darwin configuration diagrams. Our aim is to enable the desired layout of a diagram to be achieved with the minimum of effort on the part of the user.

Two levels of automated support is available. The first deals with the untangling of crossovers between bindings, and is done by relocating the communication objects at the ends of bindings and then distributing them along each face of a component box to give a regular appearance. The component boxes themselves are left unchanged. This algorithm is based on our earlier work on ConicDraw [13] and is invoked after every single modification to the diagram layout such as the moving or resizing of a box. Consequently, configuration diagrams in the sketchpad are always in the tidied state.

The second level of automated support takes care of the auto-placement of the component boxes in a configuration graph, and is invoked only on demand by the user. The objective is to produce a compact diagram with minimal crossovers between bindings and component boxes. The heuristic-based algorithm is used to generate the initial diagrams of imported Darwin text, and is also useful in situations where the large number of components and bindings makes it difficult for the human eye to easily come up with an aesthetically pleasing layout.

A number of node alignment aids are also available as further aids to the designer. A flexible template facility allows a group of boxes to be aligned either horizontally, vertically or in a ring. For instance, to arrange the 4 boxes (A, B, C and D) in figure 7 into a ring, the user first selects them and then draws a circle with the 'magnet' tool. This will pull the boxes onto the circle and distribute them evenly along its circumference. As with all other layout changes, the bindings in the diagram were automatically cleaned up after the operation.

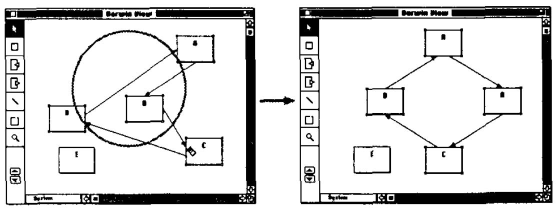


Fig. 7 Box alignment tool

5.2 System Navigation

In a complex system consisting of nested components, it is important to have quick and easy access to the information relating to any part of the program. This is a problem of navigation, and is very similar to traversing the directories of a hierarchical file system such as that of UNIX. We have provided various ways of navigating within each of the views described above as well as the option of controlling the navigation of all views at once from any one view. Within the sketchpad, navigation is supported by the up and down buttons below the tool palette. Double-clicking on a component will also drop the user into its internal configuration. An alternative method for moving up the system hierarchy is to use the 'level' pop-up menu at the bottom left hand corner of the sketchpad. This contains the path from the top of the system hierarchy down to the current level. Selecting the name of a component in this menu switches the display of the window to the internal configuration of that component. Since the name of the current component is always visible in the menu, it also serves as an indication of where we are in the system hierarchy.

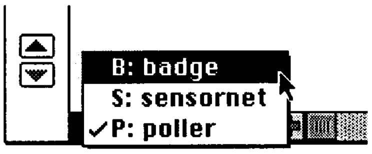


Fig. 8 The 'Level' pop-up menu

For larger systems, the tree view allows the user to go to any part of the system hierarchy without the need to traverse the intermediate levels.

5.3 User-defined Attributes

The Assistant provides a flexible attribution mechanism for information capture, allowing snippets of information to be associated with objects in the design diagram. This process is rather akin to sticking notes on a notice board (Figure 9). Typical attribute types may include formal specification, design rationale, resource requirements and functional descriptions. As described earlier, *Darwin* and *Implementation* are built-in attribute types for components for storing their Darwin and C++ code respectively.

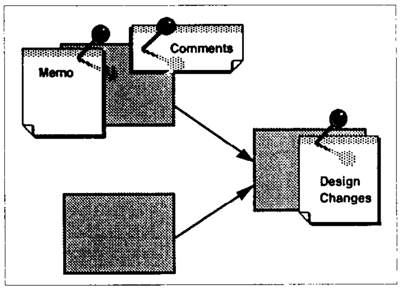


Fig. 9 Attaching information to components

New attribute types can be created by the user for storing new kinds of information. These can either be associated with all object types or be made specific to only particular types of objects. This extra flexibility allows attributes such as 'comments' to be created for all objects but limits other attributes such as 'data dictionary' to just data flows. The removal of redundant attributes and renaming of existing attributes are also supported.

5.4 Design Validation

A Darwin specification essentially describes a 'generic' structure of a component type. The instantiated structure at run-time is determined by the actual parameters passed to the component instance and the evaluation of its conditional guards, and can be difficult to visualise at design time. As a design validation aid, work is under progress to incorporate a Darwin interpreter into the Assistant to facilitate the testing of component descriptions with different parameter values, obtaining as results the instantiated configurations of the component which are reflected back to the user in a graphical form. This kind of 'what-if' scenario testing should allow many errors to be caught at an early stage without needing to go through the full compilation and execution cycles of program development (figure 10). It is particularly useful for the validation of complex components which make use of Darwin's more advance facilities such as the parameterisation of components, replication of component instances and interface ports, conditional configurations with guards, and even recursive definition of components.

5.5 Construction of Program Hierarchy

One of the common flaws of CASE tools which support hierarchical structures is the unnecessary restrictions imposed on the way the hierarchy can be constructed and modified. Many tend to enforce a strict top-down decomposition approach, and offer no easy way of altering the grouping of the objects once it has been specified. Experience has shown that in practice, a combination of top-down, bottom-up and middle-out construction styles are often adopted in switching between the different levels of abstractions. Furthermore, for a system like Regis, it is not uncommon for a designer to begin work on the various component hierarchies independently and then bring them together to form the overall program.

In the Assistant, all these different modes of working are supported : top-down decomposition is carried out by repeatedly exploding a component and elaborating its substructure, while bottom-up construction is achieved by zooming up while at the root of the hierarchy - a new root will be created at the top of the hierarchy as the parent of the current root. Middle-out construction is simply a combination of the top-down and bottom-up approach. The user can also choose to start a new component hierarchy within the program with the 'New component' command. This creates a new component tree which can later be merged into the main program tree.

If a component ever becomes too complicated, instances within it can be easily grouped into a single component by dragging and dropping selected instances into a new instance. Any bindings to the selected instances will be automatically re-bound to the new one. This operation can be reversed by 'unwrapping' the new instance.

5.6 Report Generation

Design documentation is an essential part of the design process. Given the multitude and volume of information that is recorded during this process, the ability to collate and present this information in a coherent and formatted layout is therefore important. The report generation facility of the Assistant takes the drudgery out of report collation and formatting by automatically gathering the stored attributes of the design and writing them out in a formatted file in Rich Text Format (RTF), a document interchange format. The report includes, for each composite component type, its configuration diagram and Darwin code together with all attributes belonging to its sub-components in a tabular format. For a primitive component, the report records its implementation in C++ and all the attribute information. All diagrams are automatically scaled to fit the printed page if necessary. Since the RTF format is readable by most popular commercial word processors, further editing to the contents and layout is possible after the report has been generated.

We intend to allow the user to customise the layout and contents of a report from within the Assistant, similar to the way database designers describe the layout of database views. This is not yet supported in the current version of the tool.

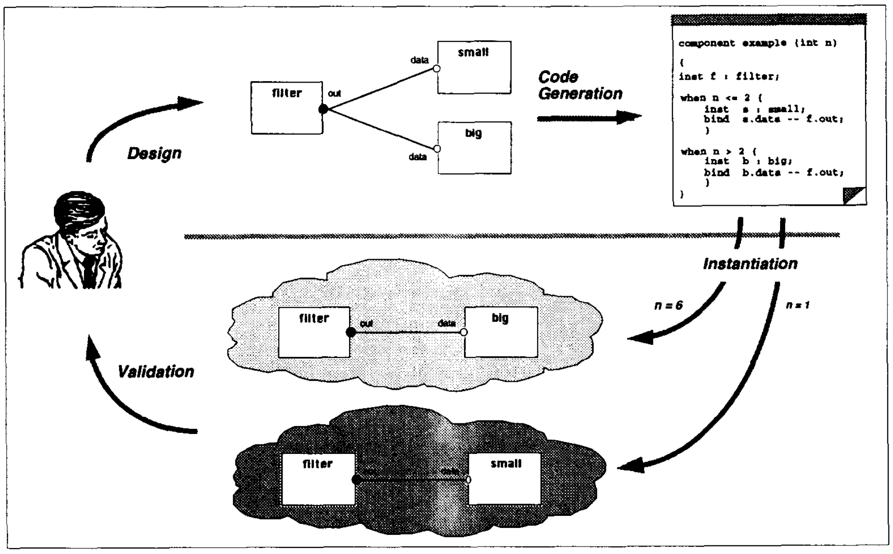
6 Related Work

Standard design methods such as JSD [14] and Structured Analysis (Data Flow Diagrams) [15] produce designs as system configurations but fail to carry these notions through to distributable implementations with explicit configurations. HOOD [16] and JSD+ [17] address some of these issues but are tailored to the Ada language. Environments such as STATEMATE [18] recognise the power of the structural view for system specification and modelling, but tend to be weak in their support for component-based distributed system construction. STILE (STructured Interconnection Language and Environment) [19] advocates and provides good support for graphical component-based design and construction, but does not provide particular support for distribution. The Designer's Notepad [20] shares many of our concerns on design capture and expression, and provides flexible means for note-taking, documentation and design exploration. However, being a generic design tool, it is weak in method-specific automated support. ROOM [21] and its associated CASE tool ObjecTime is targeted at distributed real-time systems. It embodies many of the same concepts as Darwin/Regis, including the separation of system structure from its behaviour. As in Darwin, the system architecture is specified in terms of hierarchically-structured components which communicate through message ports. The dynamics of a system is modelled using a variation of Harel's StateCharts [22].

7 Conclusion

This paper has presented an architectural design methodology in which distributed systems are described, modelled and constructed in terms of their software structure. Descriptions of the constituent software components and their interconnection patterns provide a clear and concise level at which to specify, design and analyse systems, and can be used directly by construction tools to generate the system itself. An overview has also been given of the Regis distributed programming environment which embodies such a component-based constructive approach to the development of parallel and distributed programs.

The emphasis on the structural view of systems makes them particularly amenable for expression in a graphical form. The Software Architect's Assistant is our attempt at providing tool support for the design and construction of such systems through a highly visual and interactive interface. Using the software structure as the central theme, it provides automated support for the sketching of design diagrams from which program code is automatically generated. At the same time, facilities are provided for the recording of design information, report generation and the reuse of



```
component example (int n)

{
inst f : filter;

when n <= 2 {
    inst  s : small;
    bind  s.data -- f.out;
    }

when n > 2 {
    inst  b : big;
    bind  b.data -- f.out;
    }
}
```

Fig. 10 Validation of a component description

pre-defined software components. In building the Assistant, one of our main objectives has been to maximise the level and scope of automated, intelligent assistance that is given to the user, ranging from diagram layout and consistency maintenance to the generation of code and reports. Where possible, new information is derived from existing information in order to save time and reduce human error.

Although the graphical support provided by the Assistant has proven to be an attractive as well as effective alternative to the more traditional use of text for software design and construction, we recognise that the description of regular graph structures such as replication and recursion can often be done more easily and concisely in a textual form. Hence we intend to provide better support for the mixed usage of graphics and text within the Assistant so that either may be used when appropriate.

Another area we will be exploring is the publishing of attributes for use by tools other than the Assistant. In the same way that the Darwin and Implementation attributes are currently used by the Darwin and C++ compilers respectively during program construction, it is intended that other attribute types such as component specification in the form of labelled transition diagrams should be usable by the analysis tool during the analysis phase. A standard mechanism for attribute publishing will enable the Assistant to be used as a general graphical front-end for driving more specialised external tools.

Whereas the Assistant has focused primarily on program design and development, our earlier work on ConicDraw [13] has demonstrated the feasibility and usefulness of utilising graphics for the run-time monitoring and management of distributed systems. It allowed the structure of a running program to be viewed and dynamically reconfigured through an interface similar to that of the Assistant. Our intention is to incorporate similar facilities into the Assistant to provide a complete solution for visual distributed programming

References

1. "ANSAware Release 3.0 Reference Manual", A.P.M. Ltd. 1991, Poseiden House, Castle Park, Cambridge CB3 0RD, U.K.

2. J. Kramer, J. Magee and A. Finkelstein. "A Constructive Design Approach to the Design of Distributed Systems", in Proceedings of the 10th International Conference on Distributed Computing Systems. May 1990, pp 580-587.

3. S. Eisenbach and R. Paterson. "π-Calculus Semantics for the Concurrent Configuration Language Darwin", in Proceedings of the 26th Hawaii International Conference on System Sciences, January 1993, Vol II (Software Technology) pp. 456-462.

4. S.C. Cheung and J. Kramer. "Enhancing Compositional Reachability Analysis with Context Constraints", in the 1st ACM SIGSOFT Symposium on the Foundation of Software Engineering, ACM SIGSOFT, Los Angeles, California, December 1993, pp 115-125.

5. S.C. Cheung and J. Kramer. "Tractable Flow Analysis for Anomaly Detection in Distributed Programs", in 4th European Software Engineering Conference (ESEC'93), Germany, September 1993, LNCS 717, Springer-Verlag, pp. 282-300.

6. J. Kramer, J. Magee and M. Sloman. "Dynamic Configuration for Distributed Systems", IEEE Transactions on Software Engineering Vol. 11 No. 4, April 1985, pp. 424-436.

7. J. Magee, N. Dulay and J. Kramer. "A Constructive Development Environment for Parallel and Distributed Programs", in International Workshop on Configurable Distributed Systems, Pittsburg, USA, March 1994.

8. J. Magee, N. Dulay and J. Kramer. "Structuring Parallel and Distributed Programs", IEE Software Engineering Journal Vol. 8 No. 2, March 1993, pp. 73-82.

9. A. Harter and A. Hopper. " A Distributed Location System for the Active Office", IEEE Network Special Issue on Distributed Systems for Telecommunications, January 1994.

10. D. Harel. "Biting the Silver Bullet - Toward a Brighter Future for System Development", IEEE Computer Vol 25 No 1, January 1992, pp. 8-20.

11. J. Kramer, "Configuration Programming - A Framework for the Development of Distributable Systems", in Proceedings of IEEE International Conference on Computer Systems and Software Engineering (CompEuro 90), Tel-Aviv, Israel, May 90, pp. 374-384.

12. J. Kramer, J. Magee, and M. Sloman. "Configuring Distributed Systems", in 5th ACM SIGOPS Workshop on Models and Paradigms for Distributed Systems Structuring", Mont Saint-Michel, France, September 1992.

13. J. Kramer, J. Magee, and K. Ng. "Graphical Configuration Programming", IEEE Computer, October 1989, pp. 53-65.

14. M.A. Jackson, "System Development", 1982, Prentice-Hall.

15. T. DeMarco. "Structured Analysis and Structured Specifications", 1979, Prentice Hall.

16. "HOOD Reference Manual", European Space Agency, September 1989,

17. "JSD+ Reference Manual", Sema Group, Sema Scientific, June 1991.

18. D. Harel, et al. "STATEMATE : A Working Environment for the Development of Complex Reactive Systems", in Proceedings of the 10th International Conference on Software Engineering, 1988, pp. 396-406.

19. M.P. Stovsky and B.W. Weide. "STILE : A Graphical Design and Development Environment", COMPCON Spring '87, IEEE, San Francisco, February 1987.

20. N. Haddley and I. Sommerville. "Integrated Support for System Design", IEE Software Engineering Journal Vol. 5 No. 6, November 1990, pp. 331-338.

21. B. Selic et al. "ROOM: An Object-Oriented Methodology for Developing Real-Time Systems", Bell-Northern Research Ltd., Ottawa, Canada.

22. D. Harel. "A Visual Formalism for Complex Systems". Science of Computer Programming, 1987.