Proc. of 10th IEEE ICDCS, Paris, May 1990.

A Constructive Approach to the Design of Distributed Systems

Jeff Kramer Jeff Magee Anthony Finkelstein

Department of Computing,
Imperial College of Science, Technology and Medicine,
180 Queen's Gate, London SW7 2BZ, UK.

ABSTRACT

The underlying model of distributed systems is that of loosely coupled components running in parallel and communicating by message passing. Description, construction and evolution of these systems is facilitated by separating the system structure, as a set of components and their interconnections, from the functional description of individual component behaviour. Furthermore, component reuse and structuring flexibility is enhanced if components are context independent ie. self-contained with a well defined interface for component interaction.

The Conic environment for distributed programming supports this model. In particular, Conic provides a separate configuration language for the description, construction and evolution of distributed systems. The Conic environment has demonstrated a working environment which supports system distribution, reconfiguration and extension. We had initially supposed that Conic might pose difficult challenges for us as software designers. For example, what design techniques should we employ to develop a system that exploits the Conic facilities? In fact we have experienced quite the opposite. The principles of explicit system structure and context independent components that underlie Conic have lead us naturally to a design approach which differs from that of both current industrial practice and current research. Our approach is termed "constructive" since it emphasises the satisfaction of system requirements by composition of components.

In this paper we describe the approach and illustrate its use by application to an example, a model airport shuttle system which has been implemented in Conic.

1. INTRODUCTION

Much of the current research on software development techniques and tools has emphasised automated software construction, specifically identifying the important role of formal specifications as the key to validation of user requirements and generation of the system itself [Balzer 85]. It is suggested that maintenance is not conducted directly on the system itself, but rather on the specification, with new versions of the system "automatically" re-generated from that new specification. We term this the "specification-driven" approach

Although this view of the software process is extremely appealing, it is unfortunately proving rather elusive. For instance, both automated verification and generation of software has proved to be far more difficult in practice than it seemed in those early promising days [Hoare 69, Burstall 77]. Formal specifications are still proving to be extremely difficult to develop, with subsequent interpretation being equally challenging. Automated deduction has made impressive strides forward, but we are still a long way from practical use. Declarative programming languages (such as logic and functional languages) certainly provide a sound base for research on transformation techniques to provide more efficient implementations, but leave much to be desired as specifications. They are, after all, programming languages. Part of the reason for our lack of impact on the 'real' world seems to be that, for any non-trivial system, no single specification technique seems to be adequate.

Although it may appear otherwise, the intention of this paper is not to criticise the motives or details of the research mentioned above. However, it is rather difficult to see how practitioners of software engineering for large and complex systems can take advantage of any of the advances made in system design without a complete revolution in the world of specifications. How then should we proceed?

In practice, parallel and distributed software systems are conveniently described, constructed and managed 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 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 should be used to dictate the structure of the resultant system. Evolution of the system can be achieved by making extensions or changes to the system configuration by the addition or replacement of components. Since it seems that many 'new' systems are created by modifying or evolving previous versions rather than by design and construction ab initio, there is a need to recognise system evolution as a major portion of the development process.

For a number of years, the Conic environment [Kramer 85, Magee 89] has supported the use of configuration languages in such an approach. Our experience is that those systems developed to accommodate physical distribution exhibit much more flexibility due to the enforced modularity, component independence, well-defined interfaces and explicit structure than those systems designed for centralised hardware. Since decisions on structure entail decisions on composition/decomposition, they are fundamental to the development process. The premise is that a structural (configuration) description is essential at each of the phases in the software development process, from system specification as a configuration of component specifications (eg. as in Inscape [Perry 89] or PAISLEY [Zave 82]), to evolution as changes to a system configuration. Hence, system structure should be recognised as the unifying framework upon which to hang specification, design, construction and evolution of systems. While some tools such as STATEMATE [Harel 88] have recognised the power of the structural view for system specification and modelling it is generally the case that system structure is pushed to either end of the system development process: at the top end, the architectural description of requirements and system context; at the bottom end, system management.

Our overall model for software development emphasises structural decisions and descriptions for **all** of the phases in the model. In contrast to the "specification driven" approach, our model is "constructive". The "specification driven" approach attempts to formalise the decomposition process based only on the system specification. We believe that this 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. Emphasis should rather be placed on the validation process viewed as "construction" of the system from components. The formal view of this constructive approach is that of system verification by showing satisfaction of the system specification as a composition of component specifications.

In this paper we describe the constructive design approach. The paper illustrates the approach using an example, a model airport shuttle system, which has been implemented in Conic.

2. THE CONSTRUCTIVE DESIGN APPROACH

The main principle on which the approach is based is that structure is fundamental to system design, construction and evolution. Structure should be explicitly described and preserved during the software development process ie. given an appropriate software architecture, system structure is stable and can be used to describe the system design, to construct the actual system and as the basis for system modification and evolution. Thus the main structural design is retained in the constructed system itself. We believe that an appropriate software architecture is provided by the Conic environment, based on the use of context independent component types with well-defined interfaces, and configurations of interconnected instances of components to form complex types.

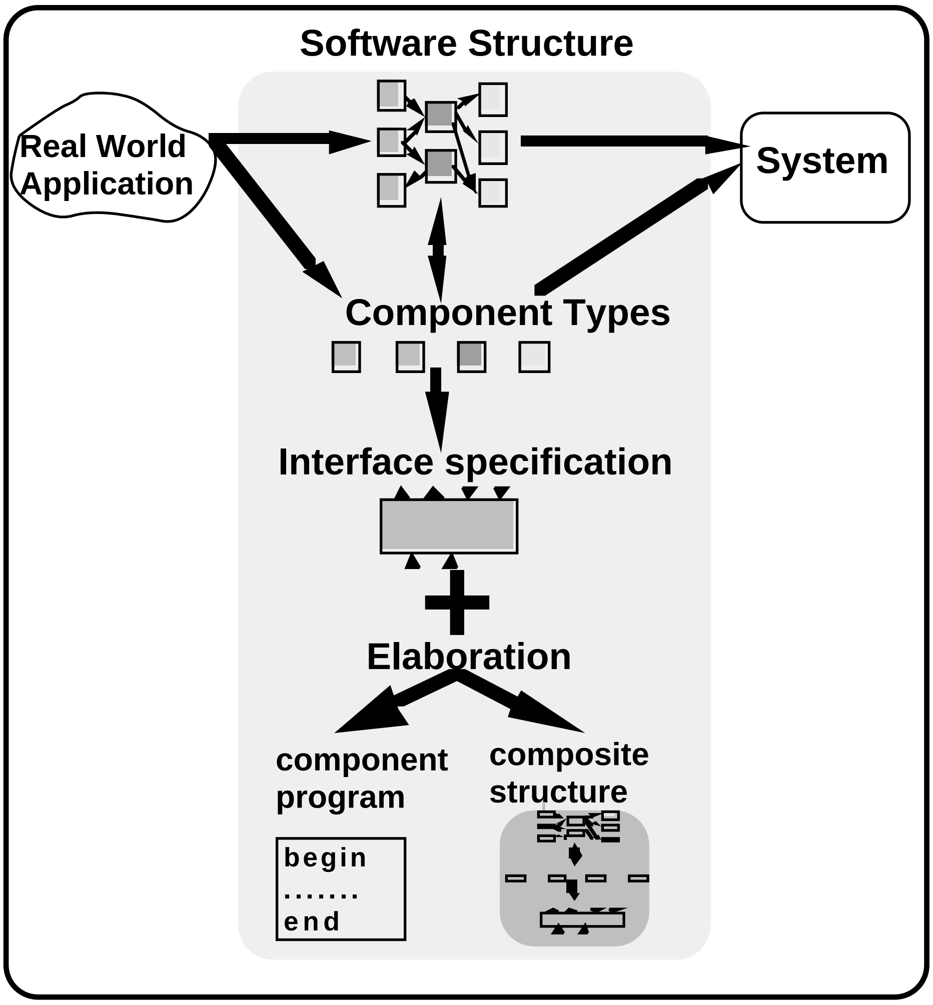


Figure 1. A diagrammatic representation of the Constructive Design Approach

The basic approach is recursive, and consists of the informal steps shown diagrammatically in figure 1 and outlined below:

1. *Structure and Component Identification:* Initial design aims to identify the main processing components and produce a structural description indicating the main data flows. The result is a configuration of processes, with interfaces giving the types of data flows, and outline descriptions of functions performed or entities modelled by the processes.

   This step provides an initial identification of component types. Furthermore, hierarchical decomposition of any of the component types into a configuration of subcomponents may be performed at this stage, or left for later refinement.

2. *Interface Specification:* Introduce control (synchronisation) between components and refine the configuration, component interface specifications (intercommunication) and component descriptions accordingly. The formulation of precise interface specifications permits the detailed design, implementation and testing of a component type to proceed independently from the rest of the system design.

3. *Component Elaboration* consists of elaboration of the component types, either by hierarchical decomposition of composite component types into a configuration of subcomponents (as in steps 1 and 2), or by detailed functional description of behaviour for primitive process components. As before, the identification of common process and composite component types is emphasised.

4. *Construction:* Node configuration by instantiation and interconnection of components to form distributable logical nodes, and system configuration by allocation and interconnection of node instances

5. *Modification and evolution* of the system is performed by the replacement or addition of nodes. Changes are specified as configuration changes applied to the operational system [Kramer 85, 88].

As in all realistic development processes, the steps tend not to be followed purely sequentially, and include both iteration and the opportunity to advance in the process on one part of the system while lagging behind on another. Modification and evolution are captured as changes to the software structure, as illustrated in figure 2.

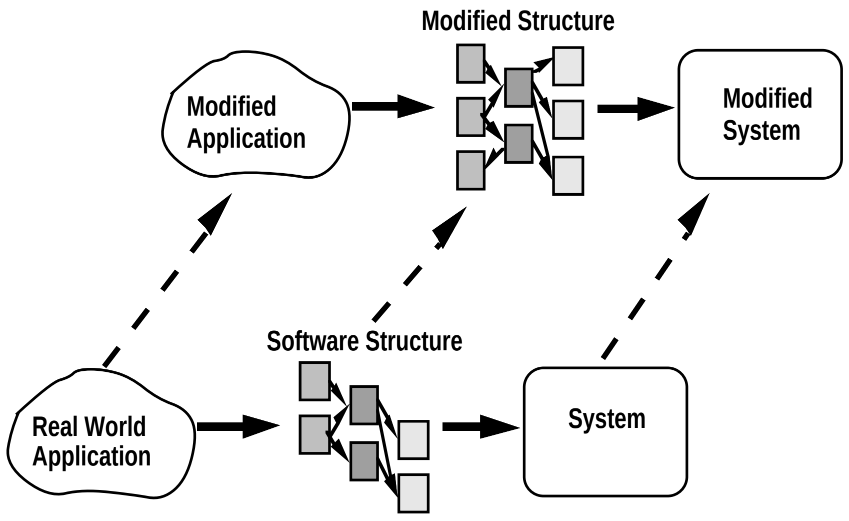


Figure 2. Evolution of a System by Structure Modification

The rest of this section discusses each of the development steps in more detail, and illustrates the approach using a model airport shuttle system described below. This example has been implemented and tested in the Conic environment, both as a simulation and on an actual model railway system.

Example: A Model Airport Shuttle System

Figure 1 depicts the track layout of a shuttle system intended to convey passengers between four terminals labelled NorthWest,NorthEast, SouthEast and SouthWest. Passengers signal their desired destination using a panel of buttons. Each button corresponds to a destination terminal. The system can support one to three passenger shuttle vehicles. Further details of this system and an alternative design may be found in [Atkinson 88].

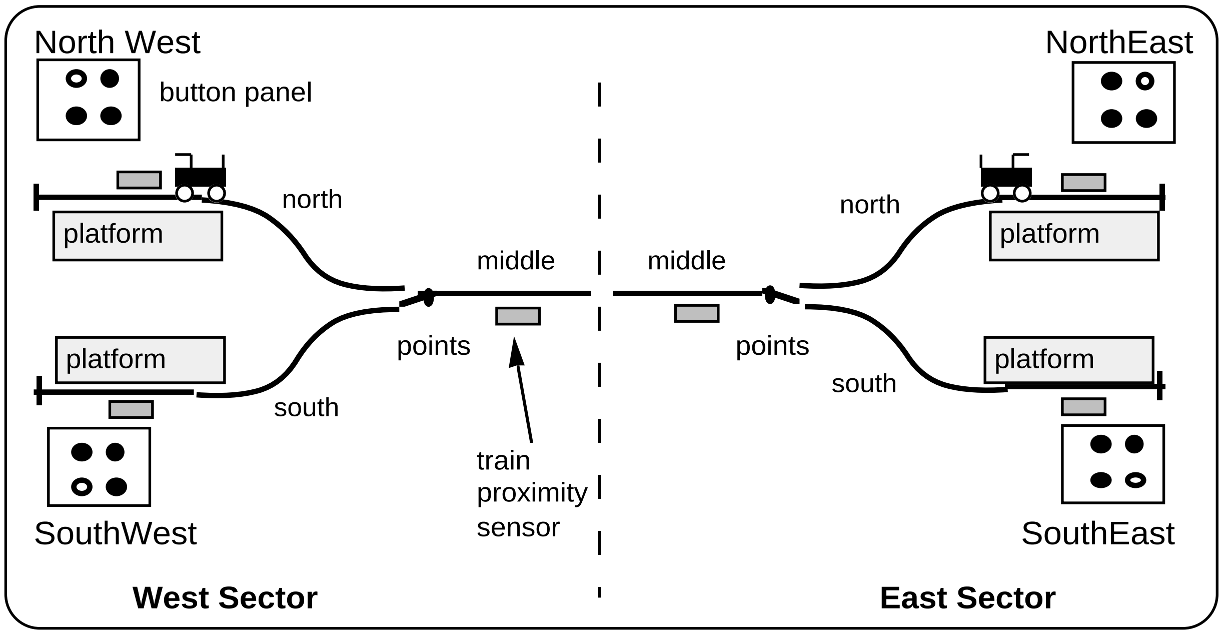


Figure 1 - Model Airport Shuttle System

Hardware

The hardware model of this system is constructed using "model railway" components. The track consists of six electrically isolated segments, three to each sector labelled north,middle and south. In each sector a set of points connects the north and south track segments to the middle track segment. Each sector has a serial interface which can be used by a computer system for control and monitoring. The functions which can be controlled are as follows:

Track Segments
- Power on / off
- Power polarity

Points
- up (connect north to middle) / down (connect south to middle)

The status which can be monitored is:

Track Sensors
- Train present/absent

Buttons
- pressed/not pressed

Trains can be moved by applying power with the appropriate polarity to track segments. The position of a train can only be directly sensed when it is over a proximity detector.

2.1 Structure and Component Identification: Initial design as Communicating Processes

There are two main approaches to this decomposition, functional and object-oriented. Functional decomposition is the approach popularised as SASD [Yourdon 78, de Marco 79]. The basis is the decomposition of the system functions into Data Flow Diagrams (DFD), which describe the system as a configuration of functional processes performing data transformations and communicating by flows of data. A Data Dictionary is used to build up definitions of the data types consumed and produced by the processes. Complex processes can be functionally decomposed into their constituent processes and data flows, thereby providing a description as a hierarchical configuration of processes. This approach is widely used, particularly for data processing applications.

The object-oriented approaches (such as [Booch 86]) are based on the identification of real world entities as objects, with state and defined interactions with other objects. Some approaches support the definition of objects in some hierarchy according to their properties, with identification of classes of objects to describe general properties, and subclasses to describe specialised properties. However, this inheritance hierarchy is not the same as the hierarchical decomposition described in DFDs, being more akin to a type definition hierarchy rather than an instance composition hierarchy. An example of a systematic design approach which follows the object-oriented philosophy, but not the inheritance hierarchy, is JSD [Jackson 83].

The approach we advocate tends to combine the data flow style of DFDs, with the entity modelling of object-oriented approaches. Object identification tends to be appropriate for embedded applications where the system components and structure mirrors the application, with functional decomposition being useful as the means for decomposing complex components. Both decomposition criteria can be used as appropriate; it is not crucial to our approach. What is important is that the identified components are context independent with no reference to external components other than as potential sources or destinations of messages. In order to reduce the coupling between components, state information is localised (cf. Information hiding [Parnas 72] rather than distributed across components. In addition, we emphasise the identification of reusable component types. The result is a description of the system design as a configuration of communicating process components modelling real-world entities, with identification of process types and the data types used in the information flows (cf. data dictionaries). Hierarchical (de)composition utilises the instance hierarchy of DFDs.

Initial Design of the Shuttle System

The following factors influence the first design step:

1) *Functional decomposition:* We wish to separate the scheduling strategy for trains from the mechanisms by which trains are moved from station to station. The design must have sufficient flexibility to allow the train scheduler to be easily replaced. From the specification, we can deduce that scheduling decisions depend on passenger requests and the location of trains. The scheduler also needs to know the set of destinations, but does not require detailed information on track layout, signalling arrangements etc.

2) *Entity modelling / component type identification:* We observe that the two sectors are essentially mirror reflections of each other. Consequently, it should be possible to consider the design of only one sector. The other sector will be essentially a parameterised second instantiation of this design. This early identification of component types considerably reduces the design and implementation effort.

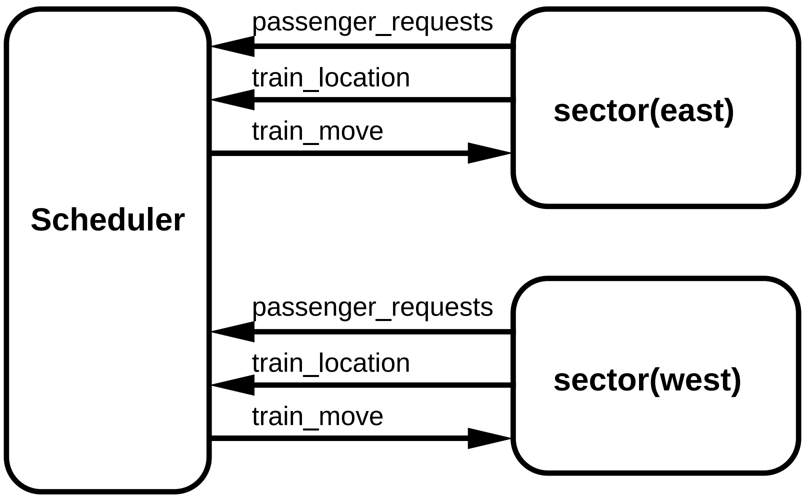


Figure 2 - Initial Design Structure of the Shuttle System

The above considerations lead to the initial design structure for the shuttle system depicted in figure 2. This is essentially a dataflow diagram showing the information flows between components. At this point in the design, control flows are ignored. We do not explicitly depict H/W interfaces. In the above system, these interfaces are encapsulated by the sector components. Arcs are labelled with the information type. An initial definition of these types is given below.

station = (NorthWest | SouthWest | NorthEast | SouthEast)

passenger_requests = list of (source,destination:station)

train_move = source,destination : station

train_location = **set of** station

It should be noted that the scheduler only needs to know the location of trains when they are in a station. The scheduler is not interested in transient location information such as train in a middle track segment. The passenger request list maintains the order in which requests were made since the scheduler requires this information to make fair scheduling decisions.

The result from this first step is thus a data-flow configuration of communicating processes, with identification of process and data types (cf. a data dictionary in Structured Analysis). For the sake of brevity, we do not provide the conventional outline description of components, but concentrate rather on the main issues in this paper: configuration structure, component types and interfaces. We now consider a possible component decomposition.

Decomposition : Sector Design

We now turn to the design of the sector component depicted in figure 2. The sector is concerned with the control of tracks and points in response to move commands from the scheduler. In addition it provides status information on the location of trains and the list of passenger requests. As in the initial design, the approach is to associate a software component with each physical component (including the computer interface) and identify the dataflows between these components. Where it seems obvious that the interface dataflows of the component being decomposed are not directly supported by these internal components, a controller or co-ordinator module is introduced. The sector component design is depicted in figure 3.

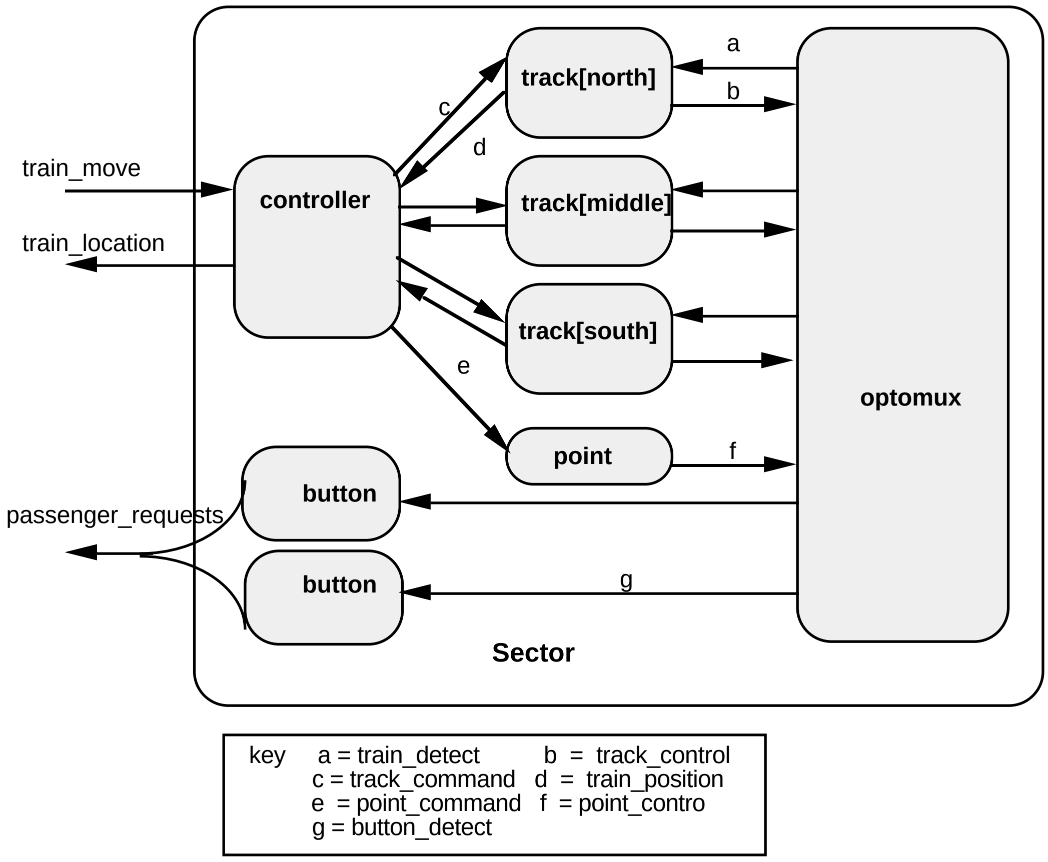


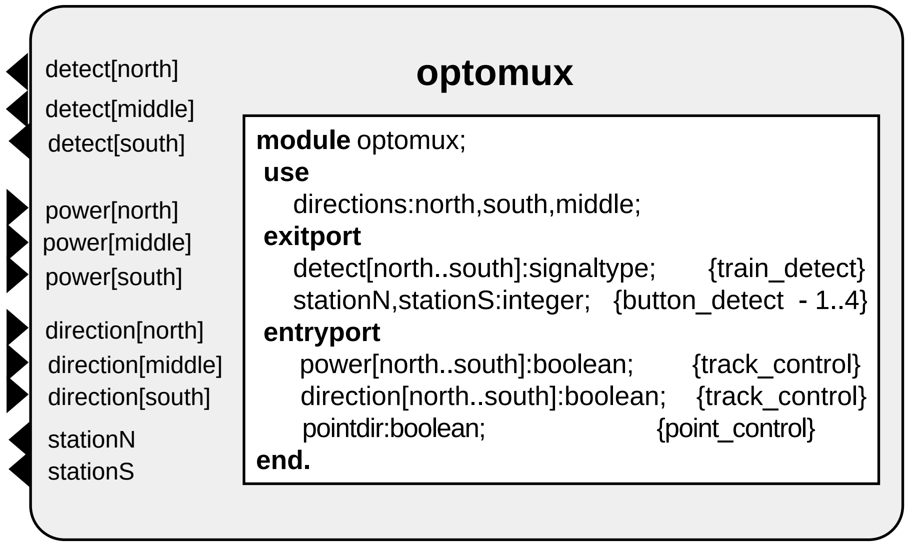
Figure 3. - Sector Design

The *controller* component translates scheduler move commands into operations on points and tracks. To reduce the level of detail exposed at this level of design, we have chosen to combine the management of track segment and track sensors into one component type - *track.* If necessary, the track component can be further decomposed. The *optomux* component provides the interface to the hardware controller (industrial "OPTOMUX" controllers). Again, component types have been identified to reduce the design effort. *Sector* is composed from the component types - *controller, track, point, button* and *optomux*.

The level of design at which decomposition stops is obviously problem dependent. For the shuttle system, the level depicted in Figure 3. is appropriate for moving to the next stage in system design - introducing control flow, leading to a precise specification of each component interface.

2.2 Interface Specification

Interfaces are specified with the assumption that components are concurrently executed and that dataflows are implemented by message passing. At this stage, issues of control flow are considered in the sense that we impose a communication transaction structure on the dataflows that have been identified, for example, whether some dataflows occur as the responses to other dataflows - a request-response transaction. We will first consider the specification of the interface to the optomux component which drives the hardware controller interface to the shuttle system.



```
module optomux;
 use
    directions:north,south,middle;
 exitport
    detect[north..south]:signaltype;      {train_detect}
    stationN,stationS:integer;  {button_detect  - 1..4}
 entryport
     power[north..south]:boolean;       {track_control}
     direction[north..south]:boolean;    {track_control}
      pointdir:boolean;                    {point_control}
end.
```

Figure 4. - Optomux Module interface

Interfaces are described in the Conic language. Messages are sent out of modules via **exitports** and received into modules via **entryports**. Definitions which must be shared by a number of modules are imported from definition units. For example, in figure 4, the definitions of the constants *north*, *middle* and *south* are imported from the definitions unit *directions*. Comments in figure 4 relate the interface to the dataflow depicted in figure 3. Note that in the case of this interface the communication control flow is exactly that of the dataflow - the providers of data initiate the communication transactions to transfer that data as messages.

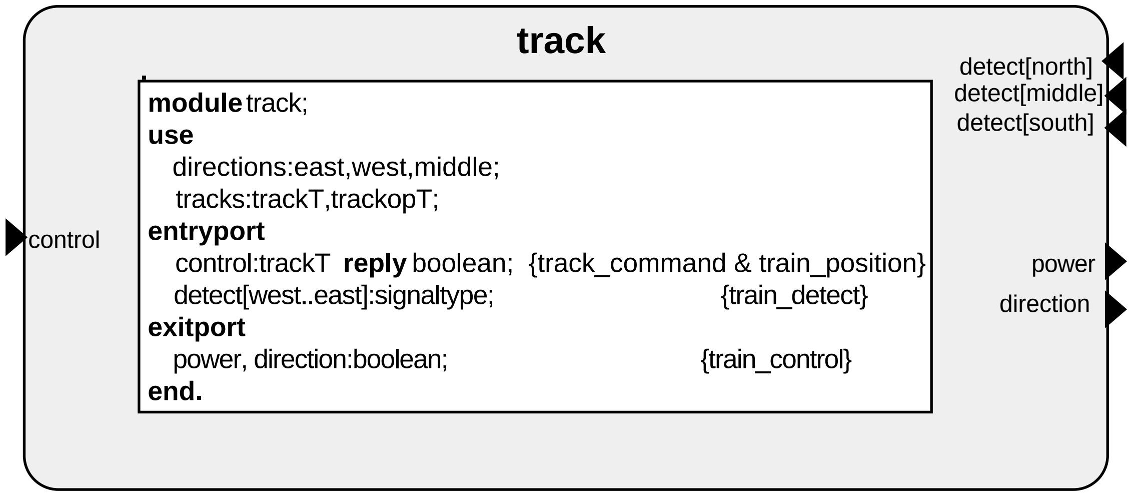
Interface Specification of the Track Component

We now consider the interface specification of the track component. Note that while we have been illustrating use of a top-down decomposition for design, some components can be designed bottom-up. In practice, lower level design is often a combination of both top-down and bottom up. Once interfaces have been specified component implementation can proceed in parallel with the design of other parts of the system.

In the previous section, we saw that shared definitions are declared in definition units. In that case, the shared definitions were concerned with the system wide representation of compass points and directions. Before specifying the interface to the *track* module, we must define the types of the dataflows between *track* modules and the *controller* since these definitions are shared. We avoided this step in the previous section by using only the standard primitive types *signaltype*, *boolean* and *integer*.

```
define tracks: trackT,trackopT;
   type
       trackopT = (on_com,query_com,capture_com);
       trackT= record                                      {track_command}
              command: trackopT;
               direction:integer; {either east or west - directions are of type  integer}
           end;
end.
```

The interface to the track module can now be specified in terms of types defined by *tracks* and primitive types (figure 5)



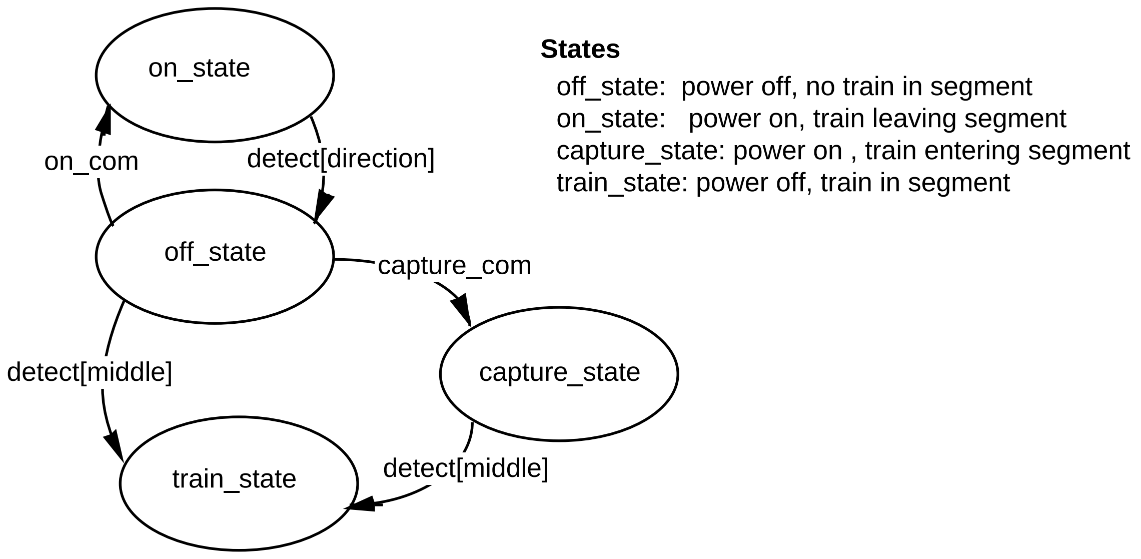
```
module track;
use
   directions:east,west,middle;
   tracks:trackT,trackopT;
entryport
   control:trackT  reply boolean;  {track_command & train_position}
   detect[west..east]:signaltype;                          {train_detect}
exitport
   power, direction:boolean;                              {train_control}
end.
```

Figure 5. - Track Module interface

Note that we have chosen to make the dataflow train_position a response to track_commands. This is because train position as represented by a boolean (true when a train is present in the track segment) can only be reliably reported after a track command has been executed.

2.3 Component Elaboration

In this step, the outline description of each of the low-level processes is elaborated to give a precise description of its behaviour. In many design techniques, pseudo-code is used at this stage. Since the Conic Programming language is a superset of Pascal, including the required communication primitives, we have found it convenient to program these processes directly. As always, refinement of the more complex actions of the process can be encapsulated into procedures.



States

off_state: power off, no train in segment
on_state: power on, train leaving segment
capture_state: power on , train entering segment
train_state: power off, train in segment

```
task module track;
use
  directions:east,west,middle;
  tracks:trackT,trackopT;
entryport
  control:trackT reply boolean;
  detect[west..east]:signaltype;
exitport
  power, direction:boolean;
var
  state:(on_state,off_state,train_state,capture_state);
  T:trackT; i:integer;
begin
  send false to power; state:=off_state;
  loop
    select
      for i:=west to east do  receive signal from detect[i]=>
          case state of
          off_state: if (i=middle) then state:=train_state;
          on_state: if (i=T.direction) then begin
                        state:=off_state;    send false to power;   reply false to control;
                       end;
          capture_state:if  (i=middle)  then begin
                        state:=train_state;  send false to power;   reply true to control;
                        end;
          train_state:;
          end;
    or
      receive T from control=>
          case T.command of
          query_com:reply (state=train_state) to control;
          on_com: begin
                        send (T.direction=east) to direction; state:=on_state;   send true to power;
                       end;
           capture_com: if (state=train_state) then    reply true to control
                  else begin
                       send (T.direction=east) to direction;   state:=capture_state;   send true to power;
                  end;
      end;
    end;
  end;
end.
```

Figure 6. Track Module implementation

The complete Conic source of the track module and a diagram of the state transitions it implements are shown in Figure 6. The track module is implemented as a single sequential program. In Conic, this is a task module. It should be noted that a track module requires signals not only from its own track sensor *detect[middle]* but also signals from the track sensors to the east and west of it. These signals are required so that a track knows when a train has left it. This requirement introduces a new interface dataflow in figure 3 since we require train detect signals from the middle track segments to be passed between sectors.

2.4 Construction

When all interfaces of components types within a higher level component have been specified as modules (these need not necessarily be implemented) we can describe the higher level component as a composition of instances of these module types using the Conic configuration language. The Configuration language description of the sector module type is outlined in figure 7.

```
group module sector(longtitude:integer); {parameterised with either east or west}
use
    directions:east,west,north,south,middle,trains,destinations,movet;
entryport
    move:movet reply trains;                                    {train_move & train_location}
    getrequestN,getrequestS:signaltype reply destinations;      {passenger_requests}
    detectin:signaltype;                                        {train_detect_in}
exitport
    detectout:signaltype;     {train_detect_out}
use
    point;  track; button; controller; optomux;
create
    point;
    stationN:button;
    stationS:button;
    controller(longtitude);
    optomux;
create forall i:[north..south]
    track[i];
link  {interface}
    move to controller.move;
    getrequestN to stationN.getrequest;
    getrequestS to stationS.getrequest;
    detectin to track[middle].detect[-longtitude];              {- reverses direction}
    optomux.detect[middle] to detectout;
        {internal}
    controller.point to point.control;
    point.direction to optomux.pointdir;
    optomux.stationN to stationN.input;
    optomux.stationS to stationS.input;
link forall i:[north..south]
    controller.track[i] to track[i].control;
    track[i].power to optomux.power[i];
    track[i].direction to optomux.direction[i];
    optomux.detect[i] to track[i].detect[middle];
link
    optomux.detect[north] to track[middle].detect[longtitude];
    optomux.detect[south] to track[middle].detect[longtitude];
    optomux.detect[middle] to track[north].detect[-longtitude];
    optomux.detect[middle] to track[south].detect[-longtitude];
end.
```

Figure 7. - Sector Module configuration

The **use** construct specifies the set of message types necessary to declare a group module interface and the set of task and/or group module types necessary to construct the group. Instances of task (or group) types are specified by the **create** construct. The **link** construct declares the interconnections between instance exitports and entryports. The replicator **forall** declares arrays of instances and links.

2.4 System Construction - Overall system description and allocation

The hardware configuration for the model airport shuttle system consisits of a Sun workstation connected by Ethernet to two Motorola 68020 target systems. These target systems are connected by serial links to the OPTOMUX controllers which power on/off tracks, switch points etc. The mapping of the software components of the control system to this hardware is dictated by the location of hardware interfaces (as is often the case in this class of system). Each target runs a sector component and the scheduler is executed on the Sun workstation. We could of course choose to run the scheduler on one of the targets. However, its implementation (not discussed here) includes a display of current requests and moves; consequently it requires the display interface provided by the Sun. This mapping is expressed in the system configuration description of Figure 9 by **at** clauses. The current configuration of the operational system can be displayed (figure 10) using the Conic graphical configuration manager, ConicDraw [Kramer

89a].

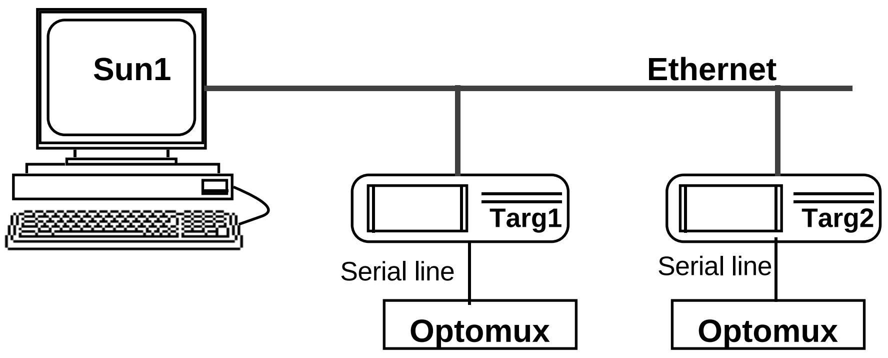


Figure 8. - Control Hardware Configuration

```
system shuttle;
use
        sector;
        scheduler;
create
        EastSec:sector(east) at targ1;
        WestSec:sector(west) at targ2;
        scheduler at Sun1;
link
        EastSec.detectout to WestSec.detectin;
        WestSec.dectectout to EastSec.detectin;
        scheduler.west to WestSec.move;
        scheduler.east to EastSec.move;
        scheduler.getNW to WestSec.getrequestN;
        scheduler.getSW to WestSec.getrequestS;
        scheduler.getNE to EastSec.getrequestN;
        scheduler.getSE to EastSec.getrequestS;
end.
```

Figure 9. - Overall System Configuration

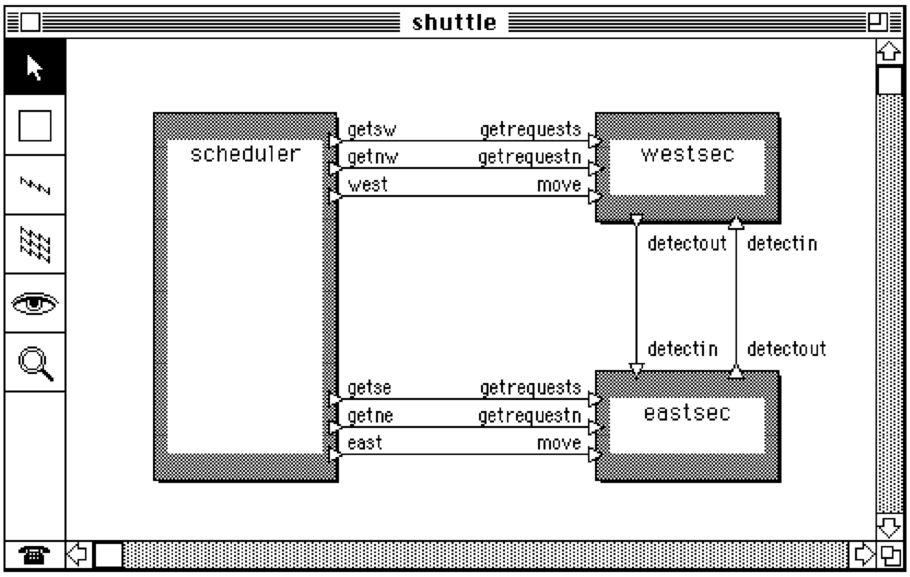


Figure 10. - ConicDraw Display of the System Configuration

2.5 Modification and Evolution

The changes which which can be applied to this system are limited by the inflexibility of the physical track layout. However, it is likely that we will want to experiment with the scheduling stratgey by which shuttles are moved in response to passenger requests. The following expresses the replacement of the scheduler component.

```
change shuttle;
use
        newscheduler;
remove
        scheduler;
create
        scheduler:newscheduler at Sun1;
link
        scheduler.west to WestSec.move;
        scheduler.east to EastSec.move;
        scheduler.getNW to WestSec.getrequestN;
        scheduler.getSW to WestSec.getrequestS;
        scheduler.getNE to EastSec.getrequestN;
        scheduler.getSE to EastSec.getrequestS;
end.
```

Note that we can apply this change dynamically to the running system without losing passenger requests and with minimum disturbance to the shuttle service by using the change management protocol described in [Kramer 89b].

3. DISCUSSION AND CONCLUSIONS

The main principles on which our constructive approach is based is that of explicit system structure and context independent components. Structure is explicitly described and preserved during the software development process, from initial design to actual system construction and evolution. Thus the main structural design information is retained in the constructed system itself. The second principle, that of context independence of components, reduces the design and implementation effort by facilitating early identification of component types and component interface specifications. Once its interface has been specified, the detailed design, implementation and testing of a component type can proceed independently from the rest of the system design. Component types can be multiply instantiated in the final system and hierarchically composed to form composite component types. Furthermore, the use of precise interface specifications can provide the basis for formal specification of required/provided component behaviour, and system specification as a configuration (composition) of component specifications.

As mentioned, the approach evolved from our experience in the development of distributed systems using the Conic environment. This suggested that those systems developed to accommodate physical distribution exhibit much more flexibility than centralised designs by requiring modularity, component independence, well-defined interfaces and explicit structure. Flexible distribution relies on the use of independent components in which information is localised. If this is not achieved, it leads to problems of maintaining consistency across distributed components. Hence we emphasise the partitioning and localisation of state information.

Our approach is similar in many respects to that of [Schneidewind 89], which also emphasises the use of independent components which communicate using messages. He too asserts that the discipline of designing to this model aids reuse by the enforced independence of the components, thereby gaining reusability, modularity, maintainability and understandability. The object oriented paradigm described in [Lee 88] uses a similar approach. DARTS/DA [Gomaa 89] offers a pragmatic method based on developing a data-flow model. The method provides some sound structuring criteria for forming distributable subsystems (components). However, we add an emphasis on explicit configuration structure, using component types to define composite types and the system itself in a uniform and constructive fashion. In addition, we provide the underlying support environment to directly translate this design into an operational system, and subsequently to evolve it as required. The mapping to software environments other than Conic would involve the sort of additional transformation as performed by Lee in transforming objects to Ada packages.

As in Luqi's approach to software evolution through prototyping [Luqi 89], our systems can be constructed as cut down versions of the final system and evolved to meet the full requirements. Again, we make use of structural extension and modification to perform the evolution.

Although our design approach has been extensively used, including the design of the Conic support environment itself, this paper is the first attempt to document it. Further work is necessary to make the method more systematic and to refine the method description by documenting many of the design decisions which we seem to make implicitly. Experience in teaching the method to other users of the Conic environment will also contribute to the method refinement. The intention is then to provide CASE tool support based on ConicDraw [Kramer89] and integrated into the Conic environment.

Acknowledgements

The authors would like to acknowledge discussions with our colleagues Morris Sloman, Naranker Dulay, Kevin Twidle and Keng Ng during the formulation of these ideas. We gratefully acknowledge the SERC ACME Directorate under grant GE/E/62394, and the CEC in the REX Project (2080) for their financial support.

REFERENCES

[Atkinson 88] "Ada for Distributed Systems", The Ada Companion Series, Cambridge University Press, Edited by Colin Atkinson, Trevor Moreton and Antonio Natali.

[Balzer 85] R. Balzer, "A 15 Year Perspective on Automatic Programming", IEEE Transactions on Software Engineering, SE-11 (11), Nov. 1985

[Barbacci 88] M.R.Barbacci, C.B.Weinstock, and J.M.Wing, "Programming at the Processor - Memory - Switch Level", Proc. of 10th IEEE Int. Conf. on Software Engineering, Singapore, April 1988.

[Booch 86] G. Booch, "Object-Oriented Development", IEEE Transactions on Software Engineering, SE-12 (2), February 1986, pp. 211-221.

[Burstall 77] R.M.Burstall, J. Darlington, 'A Transformation System for Developing Recursive Programs', JACN, 24, Jan.1977.

[De Marco 79] T. De Marco, "Structured Analysis and Structured Specifications", Prentice Hall 1979.

[de Remer 76] F.DeRemer, H.H.Kron. "Programming-in-the-large Versus Programming-in-the-small, IEEE Trans. Software Engineering", Vol. SE-2, 2, June 1976.

[Gomaa 89] H.Gomaa. "A Software Design Method fro Distributed Real Time Applications", Journal of Software Systems, Feb. 1989.

[Harel 88] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtul-Trauring, "STATEMATE: A Working Environment for the Development of Complex Reactive Systems", Proc. 10th. Int. Conf. on Software Eng., Singapore April 1988, pp. 396-406.

[Hoare 69] C.A.R.Hoare, " The Axiomatic Basis of Computer Programming", Communications of the ACM 12, October 1969, pp. 576-583.

[Jackson 83] M.A.Jackson, "System Development", Prentice Hall 1983.

[Kramer 85] J.Kramer, J.Magee, "Dynamic Configuration for Distributed Systems", IEEE Transactions on Software Engineering, SE-11 (4), April 1985, pp. 424-436.

[Kramer 88] J.Kramer, J.Magee, "A Model for Change Management", Proc.of IEEE Workshop on Trends of Distributed Computing Systems in the 1990s, Hong Kong, pp.286-295, Sept. 1988.

[Kramer 89a] J.Kramer, J.Magee, and K.Ng, "Graphical Support for Configuration Programming", Proc. 22nd HICSS, Vol.II , Hawaii, pp. 860-870, January 1989.

[Kramer 89b] J.Kramer, J. Magee, and A. Young, "A Refined Model for Change Management in Distributed Systems", 3rd Workshop on Large Grain Parallelism, SEI/CMU Pittsburgh, October 1989.

[Lee 88] K.J.Lee et al, "An OOD Paradigm for Flight Simulators, 2nd ed.", Technical Report, CMU/SEI-88-TR-30, September 1988.

[Luqi 89] Luqi, "Software Evolution through Rapid Prototyping", IEEE Computer, 22 (5), May 1989, pp. 13-25.

[Magee 89] J.Magee, J.Kramer, and M.Sloman, "Constructing Distributed Systems in Conic" IEEE Transactions on Software Engineering, SE-15 (6), June 1989.

[Parnas 72] D.Parnas, "On the criteria to be used in decomposing systems into modules", Comm. ACM, Vol. 15 (2), pp. 1053-1058.

[Perry 89] D.E.Perry, "The Inscape Environment", Proc. of 11th IEEE Int. Conf. on Software Engineering, Pittsburgh, May 1989.

[Schneidewind 89] N.FSchneidewind, "Distributed System Software Design Paradigm with Application to Computer Networks", IEEE Transactions on Software Engineering, SE-15 (4), April 19859, pp. 402-412.

[Yourdon 78] E.Yourdon, L.Constantine, "Structured Design", Yourdon Press, 1978.

[Zave 82] P.Zave, "An Operational Approach to Requirements Specification for Embedded Systems", IEEE Trans. on Software Engineering, SE-8 (3), 1982.