Development Methods for Real-Time Systems

M. E. C. HULL*, P. G. O'DONOGHUE* AND B. J. HAGAN†

* *Department of Computing Science, University of Ulster, Newtownabbey, Co. Antrim, BT37 0QB, Northern Ireland, U.K.*

† *Software Ireland Ltd, Hydepark House, Mallusk Road, Newtownabbey, BT36 8WT, Northern Ireland, U.K.*

This paper examines and compares four methods for real-time system development. The recently proposed MOON method, HOOD and the established methods of JSD and MASCOT are compared using criteria important for real-time system development and a simple real-time example application. The implications of using the methods for larger scale projects are also considered.

Received December 1989, revised May 1990

1. INTRODUCTION

Jackson System Design (JSD)[6,8] and MASCOT3[2] have been proposed as methods for the development of real-time systems. While studying the effectiveness for system development of these two methods, this paper also considers a merged method combining both JSD and MASCOT3 – MOON.[3] The latter is a new method for the development of real-time systems. An alternative approach is HOOD (Hierarchical Object Oriented Design),[5] which combines two fairly complementary methods: AM (Abstract Machines) and OOD (Object Oriented Design).

An example of a simple real-time application is used to compare the four methods and illustrate their relative strengths and weaknesses. A set of criteria is also used to evaluate the four methods. How well each method meets each criteria is determined by the development of the example application, which tests all of the desired criteria. The structure of the paper is as follows:

1. Description of a simple real-time example application.
2. Development of the example using
   (*a*) JSD,
   (*b*) MASCOT3,
   (*c*) MOON,
   (*d*) HOOD.
3. Evaluation of the methods against given criteria.

2. AN EXAMPLE APPLICATION: A TRAFFIC-MONITORING/LIGHT-CONTROLLING SYSTEM

Fig. 1 shows a road running from West to East which has a section with one lane temporarily closed. The traffic is allowed to flow from one direction at a time controlled by a set of temporary traffic lights. The shaded areas are approach zones to the traffic lights. The circles show the positions of the lights and the triangles represent sensors that monitor traffic arriving and passing through the approach zones to the traffic lights.

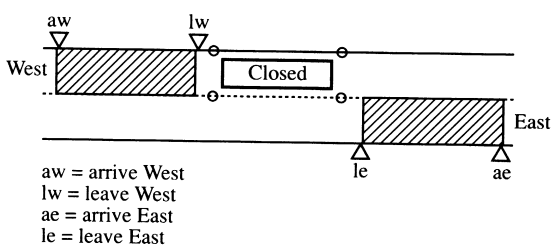

Fig. 1. The example system in its environment.

The lights alternate, allowing traffic to flow from the West and the East in turn. When traffic flows from one direction, it does so until 45 seconds have elapsed and there are cars waiting in the other approach zone. However, prior to 45 seconds elapsing, if the approach zone for the current direction empties and there are cars in the other approach zone, the lights will change.

Fig. 2 shows that there are 4 inputs (from the sensors) to the software system and one output from it (to a single device that changes all 4 lights). Figs 3 and 4 show the structure of the software which consists of an hierarchy of processes which communicate by message passing.

Fig. 3 shows how Monitor traffic is decomposed into two monitoring processes, each of which monitors the

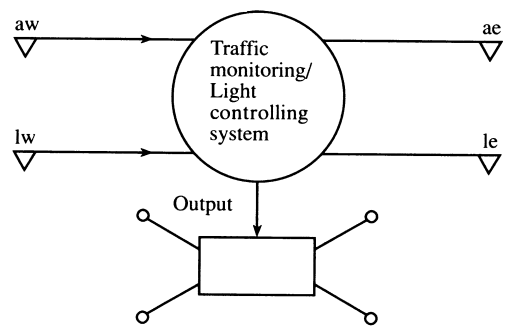

Fig. 2. The example system.

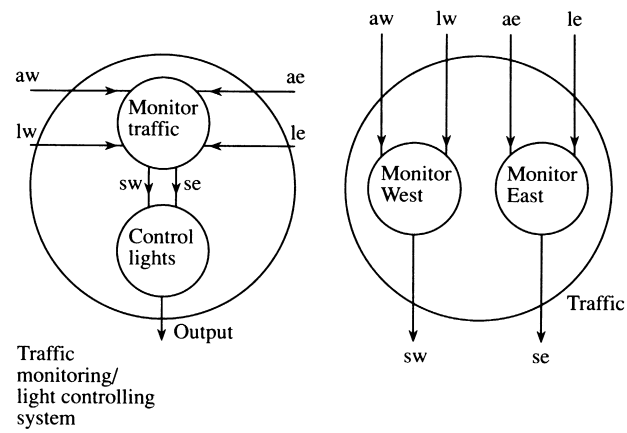

Fig. 3. The traffic monitoring subsystem.

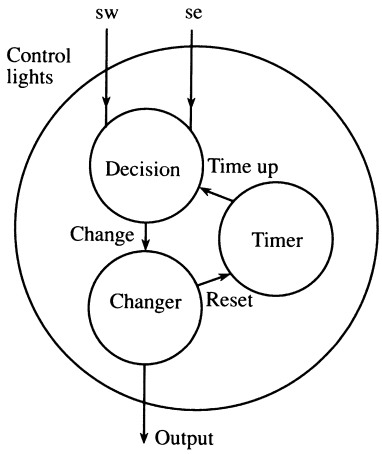

Fig. 4. The light-controlling subsystem.

number of cars in a particular approach zone. Fig. 4 shows that the Control lights process is decomposed into three communicating processes.

The difference between the two monitoring processes is that they monitor different actual approach zones to the lights. The number of cars, in the approach zone being monitored by the process, is initially set to zero. Each time an arrive message is received, the number of cars is incremented (to reflect the fact that a car has entered the approach zone). Each time a leave message is received, the number of cars is decremented (to reflect the fact that a car has passed through the approach zone). Every time there is a change in the overall state of the approach zone (the overall state of the approach zone is either that there are cars in the zone or that there are not) the new state is output as a message. Therefore, every time the number of cars changes from 0 to 1, a 'cars' message is sent, and every time the number of cars changes from 1 to 0, the a 'nocars' message is sent.

The monitor time process inputs pulse messages on the reset channel and outputs pulse messages on the timeup channel. Every time a reset message is received (the lights have changed), a count down of 45 seconds starts. If 45 seconds has elapsed since the last input signal, a timeup message for the current direction is sent. If a reset message is received prior to the countdown reaching zero, the countdown starts again from 45.

The changer process merely replicates a message indicating that the lights have to change, passing it on to the timer process and the lights. Each time the change message is received (the signal that a decision has been made to change the lights) a reset message is sent as one output (for the countdown) and a signal message is sent as the other (to the lights).

The decision-making process changes the lights repeatedly basing the decision whether to do so on the state of the traffic, which is communicated to this process from 'cars' and 'nocars' messages from both directions as well as timeup messages from the monitor time process. It starts allowing traffic to flow from the West. It allows cars to flow from the current direction until 45 seconds or more has elapsed and there are cars waiting in the other approach zone or until there are no more cars in the current approach zone but there are cars waiting in the other one. When one of these conditions holds, a change message is sent to allow traffic to flow from the other direction.

3. DEVELOPMENT OF THE EXAMPLE USING THE METHODS

3.1. The JSD version

JSD is a method that allows the results of the system analysis and specification to be expressed. The required behaviour of the example application is expressed in terms of a sequence of externally observable events. The JSP chart in Fig. 5 shows the required behaviour of the traffic-monitoring and light-controlling system.

In JSD (Jackson System Development) a single network of model processes, channels and state vector inspections represents the structure of the overall system as shown in Fig. 6. Further decomposition of the sequential model processes is performed using a hierarchical decomposition method called JSP (Jackson Structured Programming). Each of the two monitor processes maintain the number of cars in a particular approach zone by incrementing it when a message is received on the arrive channel and decrementing it when a message is received on the leave channel. The information that is provided to the decision-making process by the monitor processes is the states of the approach zones. The state of an approach zone indicates whether or not there are any cars in it. The decision-making process only needs to know whether or not there are any cars in an approach zone and does not need to know how many cars there are. The decision-making process models the alternation of direction of traffic flow. It continually examines the states of the two approach zones as well as the state of the countdown timer. If the condition required to change the lights holds then it sends a message on the change channel and notes the change in direction.

The changer process merely outputs a message on the signal channel and sends a message to the countdown timer process on the reset channel each time it receives a message on the change channel. The countdown timer process provides the state of the time the traffic has been flowing from the current direction, which is either up to 45 seconds ('counting down') or over 45 seconds ('time up'). Each time the direction of traffic flow changes, a message is received on the reset channel and the count down is reset to 45 seconds. If the countdown reaches 0 before a reset message arrives, the state is changed to 'time up east' or 'time up west' reflect that over 45 seconds has elapsed since the traffic started flowing in the current direction.

The action taken by the five model processes identified in the JSD network are sequential actions designed using JSP. The two monitor processes are the same (except for the fact that the actual channels and state vector used are different) and so a single JSP chart (Fig. 7) is used to decompose the activity.

3.2. The MASCOT3 version

The MASCOT3 notation is more rigorous than JSD and implementation can be derived from the MASCOT3 design without difficulty. It also supports hierarchical system decomposition. The overall system is firstly decomposed into a monitor and a control system as shown in Fig. 8. These two subsystems communicate with each other through a composite path which consists

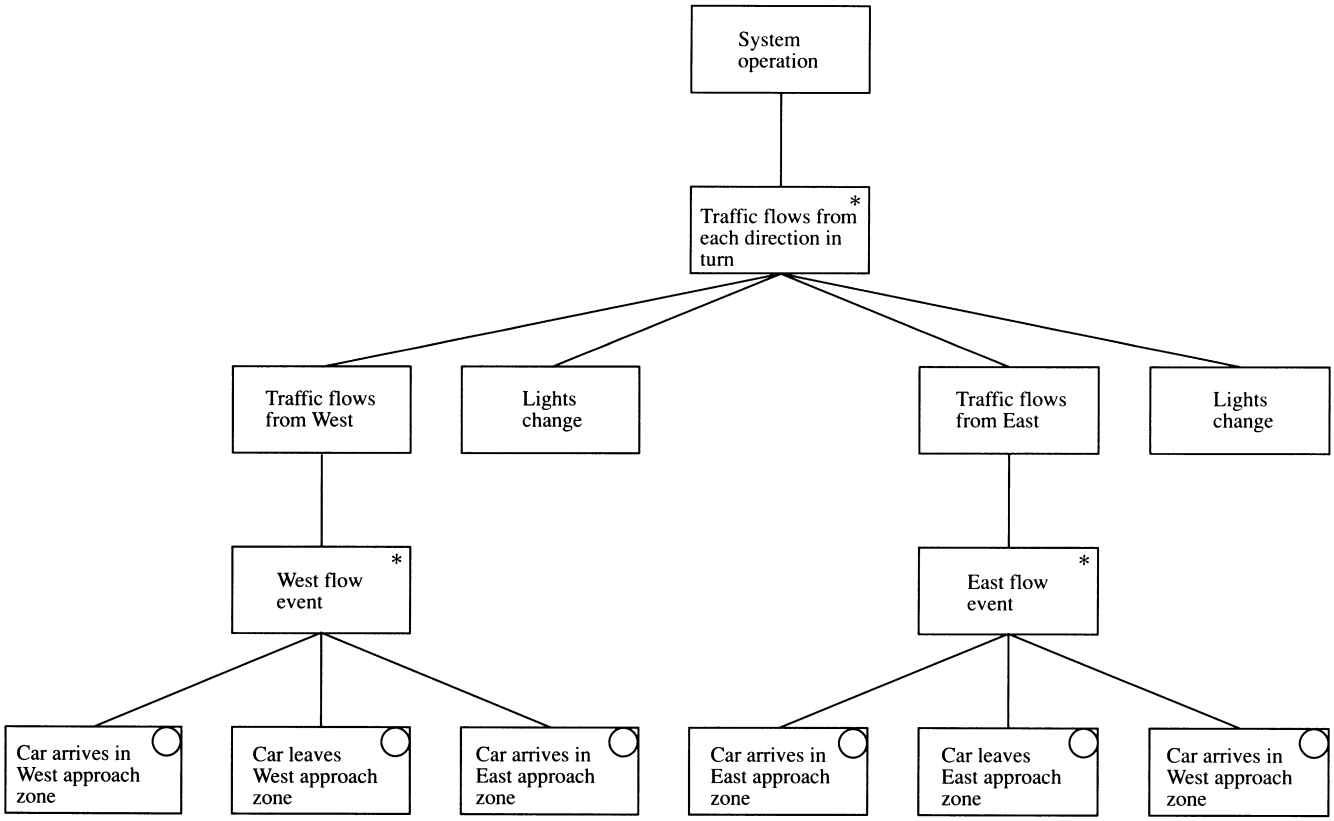

Fig. 5. Observable behaviour of intended system.

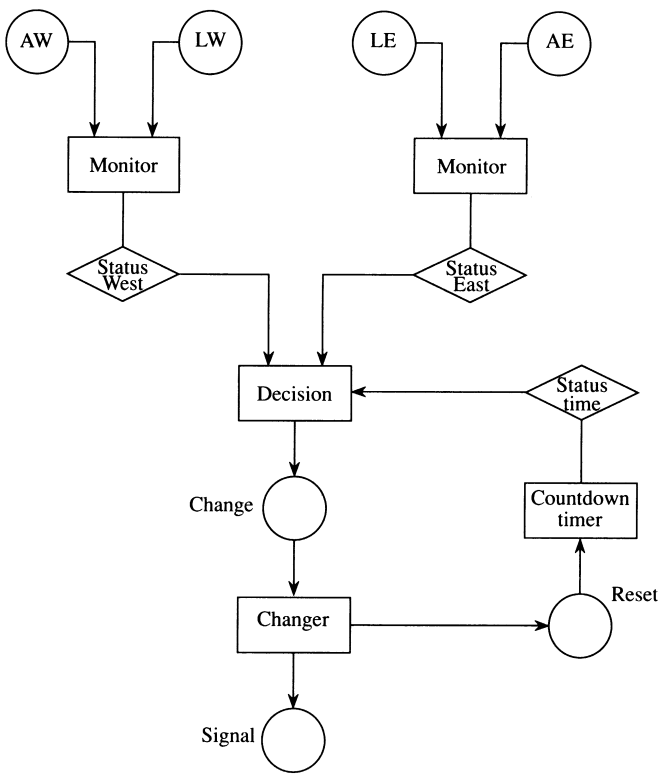

Fig. 6. The JSD design of the example system.

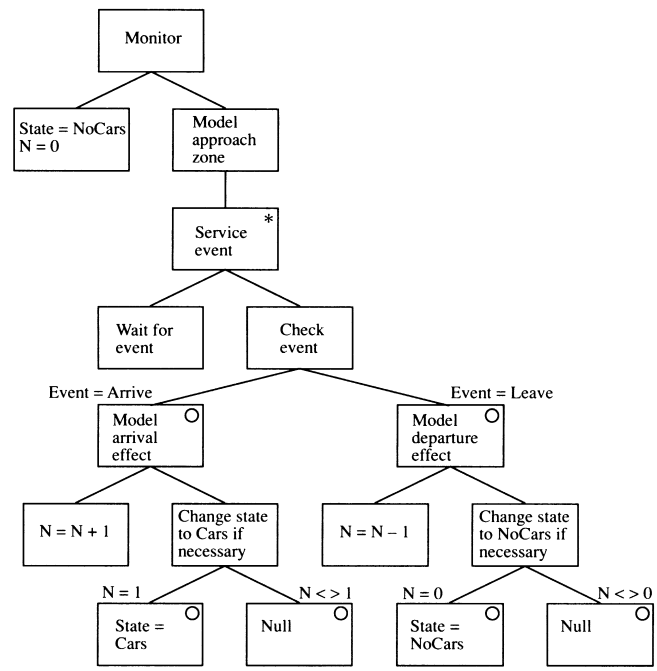

Fig. 7. JSP decomposition of the monitor subsystem.

of all lower level communications that we do not wish to outline at this level of abstraction.

The monitor subsystem template consists of a monitoring subsystem for each direction as shown in Fig. 9. These are both derived from the same template as both do the same task except on different devices and paths. The common template for monitoring an approach zone is shown in Fig. 10, illustrating how MASCOT3 helps overcome duplication of design work.

The Control subsystem template consists of a decision activity, a changer subsystem and a timer subsystem, as shown in Fig. 11. The changer and timer subsystems are broken down in Figs 12 and 13 respectively. This completes the hierarchical decomposition of the system.

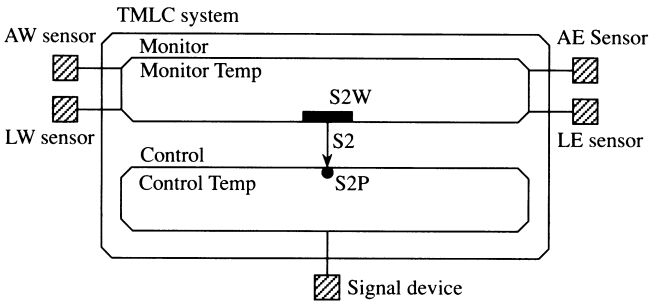

Fig. 8. MASCOT3 system diagram.

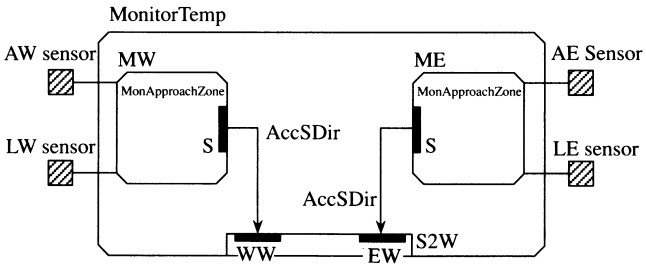

Fig. 9. Monitor traffic subsystem in MASCOT3.

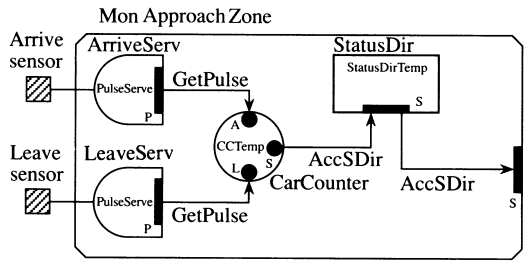

Fig. 10. Monitor approach zone subsystem in MASCOT3.

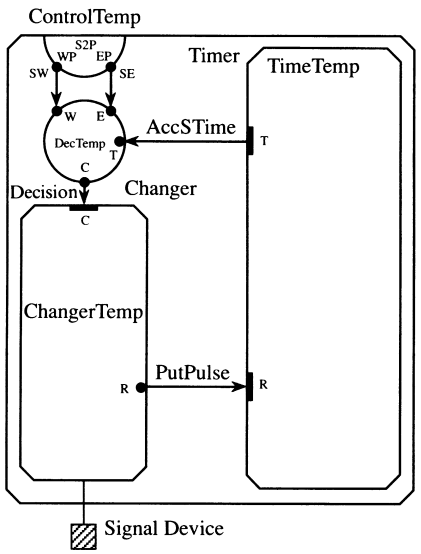

Fig. 11. Control lights in MASCOT3.

Ultimately, at the bottom of this hierarchy is a network of activities that are decoupled by inter-communication data areas (IDAs). In the same way as JSD forced the description of the system in terms of channels and state vectors, MASCOT3 forces the description of system in terms of IDAs. Direct communication (such as message passing) is not allowed.

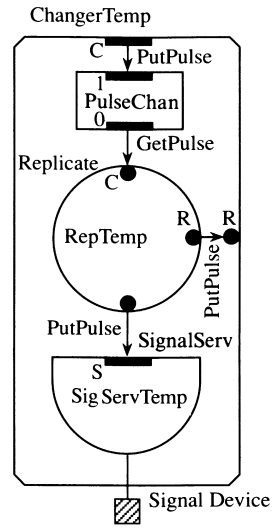

Fig. 12. Changer subsystem in MASCOT3.

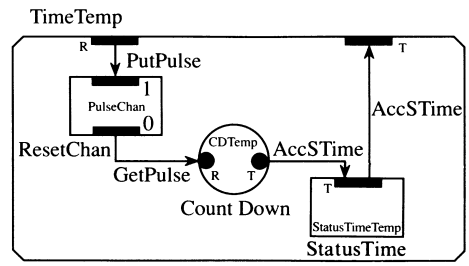

Fig. 13. Timer subsystem in MASCOT3.

Figs 8–13 also show that there are many items that must be named in MASCOT3. Though this may be inconvenient, the naming of components, ports, windows and access paths is necessary for a complete and unambiguous breakdown which avoids duplication. The next stage of the MASCOT3 method involves coding (using a PASCAL-like notation) of the access interface specifications. Once this has been completed the activities which require them and the IDAs which provide them can be coded. As the access interfaces, activities and IDAs are related within the hierarchy of systems and subsystems, MASCOT3 provides a textual view of a subsystem which can be used as a basis for developing the components within it. The textual view of the Mon subsystem template (its graphical view is shown in Fig. 9) is as follows.

```
SUBSYSTEM MonAproachZone;
 PROVIDES S : GetSDir;
 USES PulseServ, CCTemp, StatusDirTemp;
  SERVER    ArriveServ : PulseServ;
  SERVER    LeaveServ  : PulseServ;
  POOL      StatuDir   : StatusDirTemp;
  ACTIVITY  CarCounter : CCTemp
                         (A = ArriveServ.P,
                          L = LeaveServ.P,
                          S = StatusDir.S);
  S = StatusDir.S
END.
```

The access interfaces of the internal paths, specified by the port-window connections above, are named in the component templates.

```
SERVER PulseServ;
 PROVIDES P : GetPulse;
END.
POOL StatusDirTemp;
 PROVIDES S : AccSDir;
END.
ACTIVITY CCTemp;
 REQUIRES A, L : GetPulse;
          S    : AccSDir;
END.
```

The two types of access interface are specified so as the CCTemp activity, the StatusDirTemp pool and the PulseServ server can be implemented in terms of the access facilities provided.

```
ACCESS INTERFACE GetPulse;
 FUNCTION Pulse : Boolean;
END.
ACCESS INTERFACE AccSDir;
 PROCEDURE PutStatus(Status :
                     CarsState);
 FUNCTION Status : CarsState;
END.
```

3.3. The MOON version

The MOON version is decomposed using an extension of the MASCOT3 notation as shown in Figs 14–19. The extended notation abstracts the traffic-monitoring and light-controlling system to a subsystem of an overall system which uses it. Therefore there are no servers; instead IDAs provide net system input/output to external systems which contain or interact with the traffic-monitoring and light-controlling system. The MOON method is as rigorous as MASCOT3, but is different not only because there are no servers, but also because arrays of components have been introduced and composite activities are not used in the extended version of MASCOT3.

MOON, like MASCOT3, provides a textual view of its graphical notation. In MASCOT3, the access interfaces of internal paths are not named in the appropriate system/subsystem text. Instead, the internal component's templates must be examined to determine the access interface types. Therefore, a MASCOT3 text is not exactly equivalent to a diagram, though the complete set of texts is equivalent to the complete set of diagrams in a system decomposition. So as the software engineer can decompose a system in a top-down manner, using either the graphical or textual version of MOON, the textual representation of MOON has been amended to allow inclusion of internal path's access interfaces as shown in the text equivalent to the top level system.

```
SYSTEM TMLCSystem;
 PROVIDES A[West..East], L[West..East]
          : IPPulse;
 PROVIDES Out : LPulse;
 USES MonitorTemp, ControlTemp;
  SUBSYSTEM Monitor : MonitorTemp;
  SUBSYSTEM Control : ControlTemp;
  (S2P:S2 = Monitor.S2W);
  A[i:West..East] = Monitor.A[i];
  L[i:West..East] = Monitor.L[i];
  Out = Control.Out
END.
```

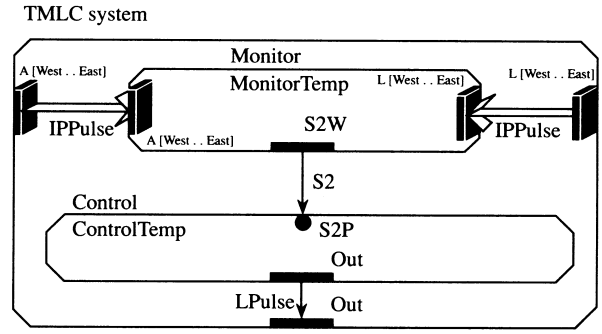

Fig. 14. The example system in MOON.

The monitor subsystem template consists of a monitoring subsystem for each direction as shown in Fig. 15. Instead of deriving the two components from the same template separately, MOON allows an array of two components to be instantiated from that template. Though the advantage over MASCOT3 is not that significant in this example, where a greater number of components are derived from the same template, the array facility of MOON is beneficial.

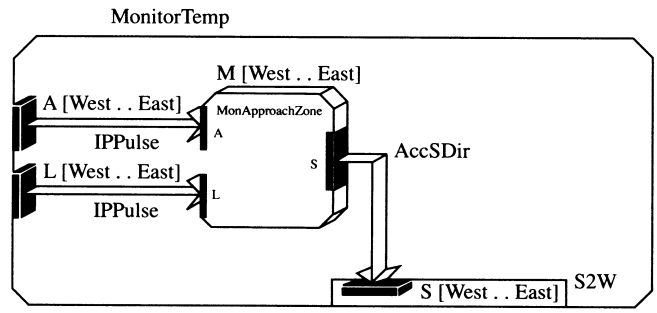

Fig. 15. Monitor traffic subsystem in MOON.

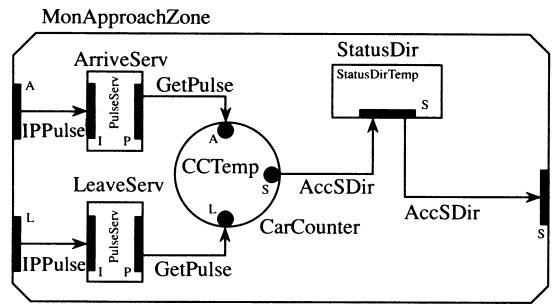

Fig. 16. Monitor approach zone subsystem in MOON.

Fig. 16 shows that the control subsystem template is decomposed as it was in MASCOT3. The connection of the LPulse path to external systems propagates down to the lowest level from the top instead of being localised at the bottom level by a server. The MASCOT3 approach would, therefore, save specifying external connections at all levels. The MOON decomposition is completed in Figs 17–19.

At the bottom of the MOON hierarchy is a network of activities that are decoupled by inter-communication data areas (IDAs). Rather than decomposing these activities as composite activities in terms of dataflow, MOON provides a mechanism for further decomposition using an extension of JSP. Consistency between the two

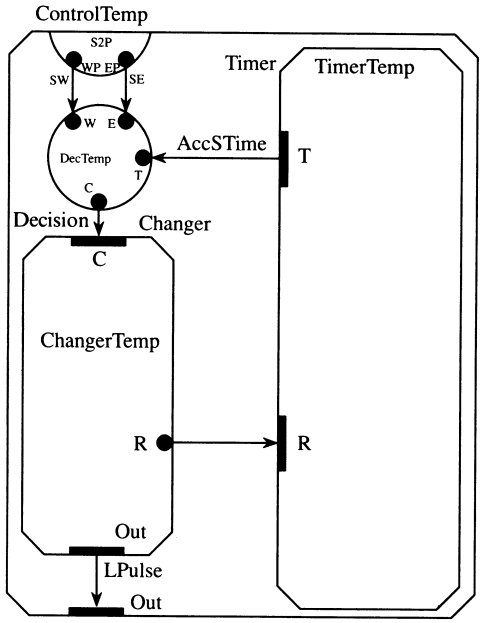

Fig. 17. Control lights subsystem in MOON.

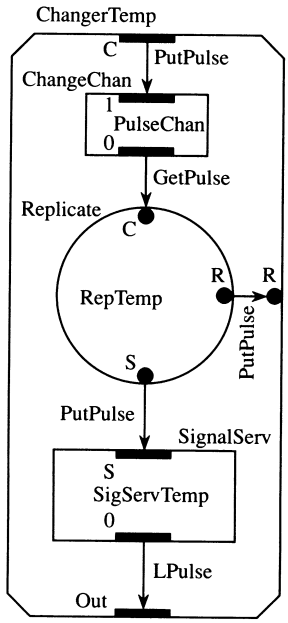

Fig. 18. Changer subsystem in MOON.

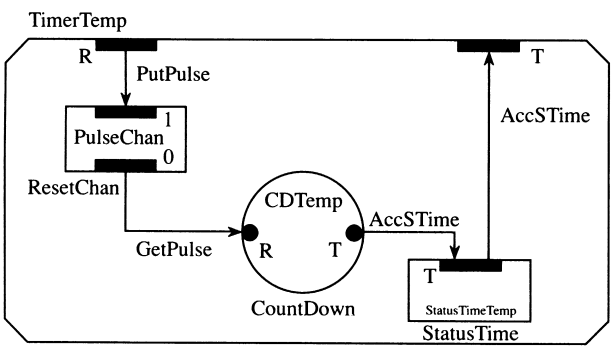

Fig. 19. The timer subsystem in MOON.

levels of abstraction is preserved by representing port facility access as primitive actions to the JSP decomposition. MOON uses an extension of MASCOT3 instead of JSD as it allows hierarchical decompositions of large systems. However, once basic networks have been identified, a method such as JSD could be used by other merged methods, prior to the use of JSP. This difference in the hierarchical decomposition is shown by comparing Fig. 20 (the MOON decomposition of the approach zone monitoring process) with Fig. 7 (the JSD equivalent).

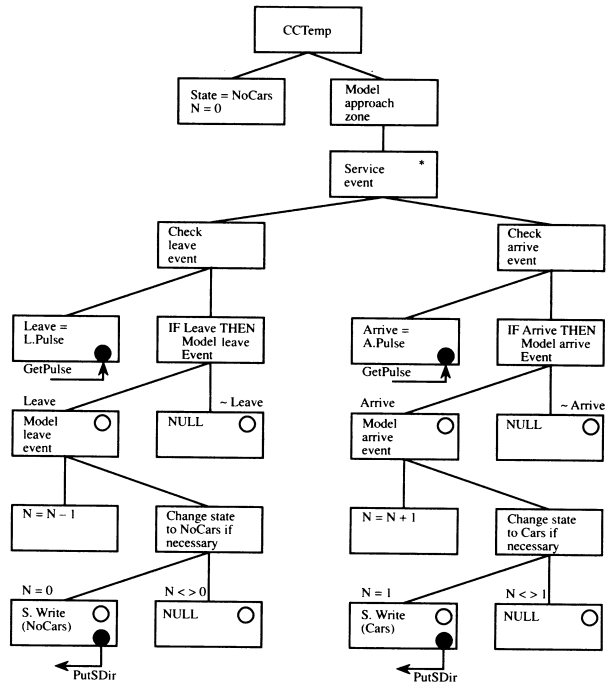

Fig. 20. MOON decomposition of the monitor approach zone activity.

MOON provides other extensions to MASCOT3 and JSP not illustrated by the example. The extensions to MASCOT3 include further primitive IDAs for man-machine-interface, file and database organisations and manual access. Because MOON is used for systems to be executed by a variety of processor types, the physical environment of processors and storage units is described using the physical version of the extension to MASCOT3 instead of the logical version. Dataflow between logical elements can also be shown in MOON using a Yourdon-style notation.[9]

The extensions to JSP provided by MOON allow rigorous control of iteration and selection, and instantiation of actions (and data) from action (and data) templates. There is also a textual version of the extension to JSP, just as there is one to the extension of MASCOT3.

3.4. The HOOD version

HOOD uses a different notation and approach to concurrent system design than MASCOT3. The system is described as a hierarchy of objects each of which could be active or passive. The higher level objects include lower level ones of which they are composed. An object can access the visible facilities of another with the use relationship, which is represented as an arrow between

the using object and the one it uses. A passive object is one which provides facilities through its Operation Control Structure (OPCS), whereas an active object also has control flow interactions with other objects. Objects have a visible interface as well as hidden information.

In MASCOT3, MOON and JSD, the system is decomposed into a network of active and passive elements which execute or exist in parallel. The system/subsystem hierarchical decomposition provided by MASCOT3 and MOON, merely groups separate elements into a subsystem. In HOOD, when an active object is decomposed, it not only has a hierarchical relationship with its sub-objects, but also controls the order of their execution. The basic MASCOT3 activities, IDAs and access paths, at the bottom of the MASCOT3 decomposition, can execute as a 'flat' JSD-like network without the hierarchy of subsystems that structure them together. In HOOD, on the other hand, the upper level objects are necessary as they contain control information vital to system execution. This control flow information is not shown in HOOD's graphical notation, but is described textually in the active object's Object Control Structure (OBCS). Fig. 21 shows the top level system as one which interacts with another system (its environment). Though this aspect of systems is not addressed by MASCOT3 in this way, MOON does recognise that a system is part of or used by external systems.

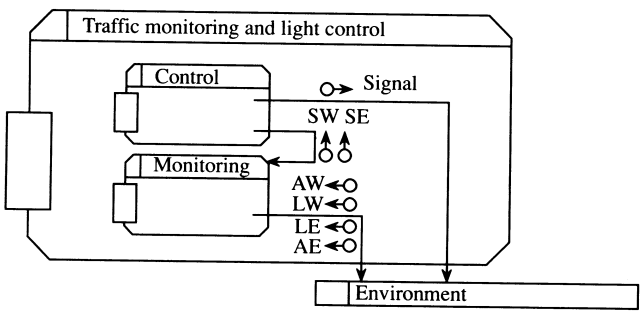

Fig. 21. The HOOD version of the example system.

HOOD employs a Yourdon-like approach[9] in showing the data that flows between objects. There is also a notation in HOOD used for exception flows, which is against the normal flow of control. It is represented by a line crossing the use relationship. Fig. 22 shows the HOOD decomposition of the Monitor object. Though this object has been decomposed into two similar sub-objects, an array of two objects could have been specified using HOOD's class instances. This structuring facility has the same advantages as MOON's equivalent over the MASCOT3 method. Each HOOD object provides its visible facilities through a single interface. Therefore, different views of an object cannot be expressed in HOOD. MASCOT3 provides a window for each separate view.

Fig. 23 shows the decomposition of the control object into a decision, a timing and a light-changing object. Now that all of the terminal objects have been identified, they can be implemented. The guidelines for implementing a HOOD object in the ADA language, have been outlined in the HOOD reference manual.[5]

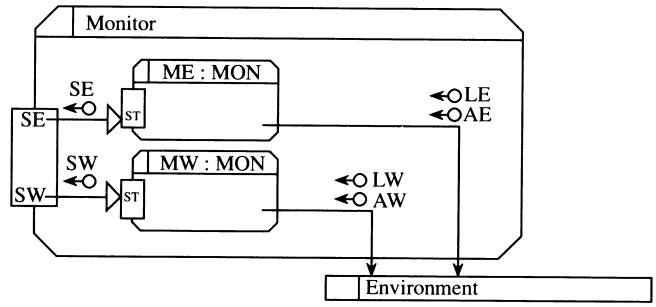

Fig. 22. The monitor traffic subsystem in HOOD.

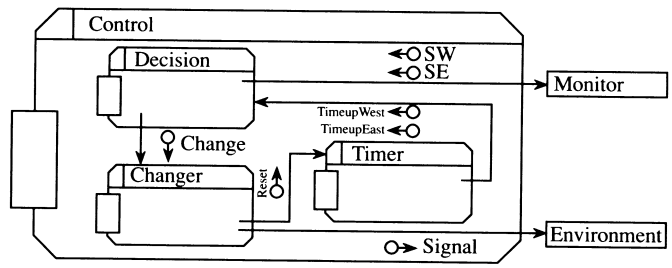

Fig. 23. The control lights subsystem in MOON.

4. EVALUATION OF THE METHODS

The four methods for real-time system design that have been examined, using the traffic monitoring and light controlling example, can be evaluated using a set of suitable criteria and considering their performance in the development of the example. The criteria are listed below.

(*a*) Ease of use and understanding;
(*b*) Unambiguous;
(*c*) Concise;
(*d*) Concurrency/real-time Support;
(*e*) Ability to derive implementation from design;
(*f*) Support for top-down development;
(*g*) Ability to express requirements and constraints;
(*h*) Ability to express different views of the system;
(*i*) Consistency;
(*j*) Graphical/diagrammatic;
(*k*) Addressing different phases of software life-cycle;
(*l*) Modularity.

(a) Ease of use and understanding

Ease of use is important to allow the software engineer to communicate the development of the system without unnecessary effort spent on syntactic details of the method. Ease of understanding allows modification of systems developed using the given techniques, by staff other than the original software engineer. JSD was the easiest notation used to develop the example application and is also the easiest to understand, as the whole system structure can be expressed and viewed on a single diagram. The JSP charts are, however, cumbersome for the simple actions of the system and a textual version would have been easier to produce. The large number of components, templates, ports, windows and paths that have to be named in MASCOT3 and MOON gives rise to difficulty. Though it is hard to establish so many unique and meaningful names, all of the names are

necessary to allow consistency and completeness while avoiding duplication. Another factor which effects comprehension of the MASCOT3 design of the example is the use of a separate diagram for each subsystem in the hierarchy. For this example JSD is preferable but for a larger project, system decomposition aids understanding and system clarity. Thus MOON and MASCOT3 are easer to understand than JSD when used for large projects. The example application was easier to develop in HOOD than in MASCOT3 or MOON. This is due to not having to generate names for so many items. Because it supports hierarchical decomposition, it is easier to use and understand than JSD for large software projects. Though the HOOD version for this simple example used hierarchical decomposition, it could have been abstracted to a 'flat' network of objects that would not have been significantly harder to create or understand than the JSD version.

(b) Unambiguous

The MASCOT3 and MOON versions, due to their rigorous development, are totally unambiguous. The JSD version lacks the rigor of MASCOT3, but a unique interpretation of the structure of the system is made by the observer. The JSP notation does not force exact specification of the conditions of selection of iteration and selection which leads to ambiguity. The HOOD structure of the system requires examination of the OBCS of each active object to determine the exact order of sub-object execution.

(c) Concise

The hierarchical decomposition provided by all methods except JSD allows separation of a large system specification into a number of more concise and less cumbersome specifications. In MOON the instantiation of action and data from templates also allows separation of complex structures using a number of more concise designs. The array facilities provided by both HOOD and MOON also aid conciseness. JSD is concise for small systems such as the traffic-monitoring and light-controlling example.

(d) Concurrency/real-time support

All of the methods reviewed support concurrency. However, only HOOD addresses other real-time facilities such as exception handling and time outs.

(e) Ability to derive implementation from design

Because MASCOT3 is very rigorous with a PASCAL-like notation for activity, IDA and access interface specification, implementing the designed system is straightforward. It is the intention that MOON specifications can be compiled into executable subsystems by a set of tools called an automated system development generator.[3] Thus implementation is automatic. Guidelines have been provided to help the software engineer implement HOOD designs in Ada. JSD lacks the rigor of the other methods and therefore it requires much greater effort to derive an implementation from a JSD specification than from any of the other methods.

(f) Support for top-down development

All of the methods support top-down development with the exception of JSD. MASCOT3 supports top-down design using its graphical notation but the textual form for implementation appears to be oriented to a bottom up implementation. MOON has amended the textual notation used in MASCOT3 in a way which allows both top-down design and implementation as each system's diagrammatic form is equivalent to the text version, which includes detail of internal connections. Because an object's OBCS can be developed along with its graphical representation, HOOD also supports top-down design and development.

(g) Ability to express requirements and constraints

JSD starts much earlier in the software life-cycle than the other methods and can be used to model the required behaviour of the system being developed. Though MASCOT3 does not address requirements analysis, the CORE method[7] has been proposed as a front end to the MASCOT3 method. The only constraints that can be provided in MOON are hardware to software mapping constraints. If we consider the example cited in this paper, a constraint is that a car cannot leave the eastern approach zone until the lights allow traffic to flow from east to west. Though the methods can be used to design a system to meet this constraint, none of the methods have formally specified it.

(h) Ability to express different views of the system

The use of windows in IDAs and subsystems in MASCOT3 and MOON allow different views of the facilities provided by a given template to be provided by different access interfaces. A HOOD object provides all of its visible facilities through a single interface made available to all using objects. This does not support separate views of the system as with MASCOT3 and MOON. A JSD process could provide several state vectors to inspecting processes. Each state vector would provide a different view of the model process.

(i) Consistency

The consistency between a MASCOT3 system and its subsystems is preserved by ensuring the ports, windows and their access interfaces are exactly the same as the nett ports, windows and access interfaces of its components. The activities at the bottom of the hierarchy require ports which are consistent with those specified in the PASCAL-like development of the activity's template. Similarly the IDAs at the bottom of the hierarchy provide windows which are consistent with those specified in the PASCAL-like development of the IDA's template. In MOON, instead of using a PASCAL-like notation, an extension of JSP is used which has port facility access as a primitive action. This helps preserve consistency between the MASCOT and JSP levels of MOON. JSD has a process chart for each process identified in the network. However, consistency between the network and process levels is not enforced with the same rigor as in the other methods. Consistency between a HOOD object and its sub-objects is preserved by mapping the parent's operations to its children's operations.

(j) Graphical/diagrammatic

All of the methods provide graphical views of system development which are beneficial for system decomposition to process level. However, the example shows that at the lower levels, where sequential actions are being decomposed, diagrams can become cumbersome and text should be used to express the action.

(k) Addressing different phases of software life-cycle

JSD addresses the specification and design phases of the software life-cycle, but lacks the rigor to be used for implementation. MASCOT3 starts at the design phase and allows development to a level where implementation is straightforward. For the earlier stage of requirements analysis, CORE[8] has been recommended. MOON addresses as much of the system development process as MASCOT3. Both of these methods are also used for the development of test harnesses and test plans. HOOD is a design method that allows development to a stage where implementation can be continued in Ada.

(l) Modularity

MOON and MASCOT3 are modular methods, allowing IDAs and related activities to be encapsulated within subsystems. Furthermore, IDAs are expressed in the two methods as collections of hidden data and visible access facilities. The only claim JSD could make to being at all modular is that it allows model processes to hide data (e.g. the number of cars in an approach zone) and provide state inspection vectors to external processes. HOOD objects can be composed of both active and passive sub-objects within constraints, and is therefore a modular approach.

5. CONCLUSION

The design methods that have been reviewed all support diagrammatic notations for expressing the system being developed. JSD and MASCOT3 are two widely used methods that have adopted the shared variable approach to process synchronisation. The MOON method, which is a combination of these two complementary methods, has overcome some of the disadvantages of each. JSD does not provide expression of the decomposition of the system until process level and MASCOT3's graphical notation for process decomposition does not show control flow.

MOON provides the hierarchical network decomposition of MASCOT3 and the process decomposition of JSP. The rigor of MOON and MASCOT3 allows implementation to follow design with less effort than from a JSD design. However, the informal nature of JSD makes it more useful for the early stages of analysis and modelling than the other methods.

Though MOON has taken advantage of the best features of MASCOT3 and JSD, it does not provide a server symbol as MOON regards all systems as subsystems of larger systems. This is clearly not as good as MASCOT3, where external access to the system is localised at a single level. However, it does allow reusable components to be integrated into larger systems without conversion.

The example application is conceptually a dataflow example and is expressed well in HOOD, a method promoting the alternative message passing approach. However, for applications that do not naturally abstract to a message passing model, the other methods allow expression of the design without compromising the conceptual nature of the application.

Acknowledgement

MOON was developed as part of a SERC-funded research project under the ALVEY programme. The participating bodies are University of Ulster and Software Ireland Ltd; ALVEY project SE/080; 'An Automated System Development Generator'.

REFERENCES

1. M. A. Jackson, *Principles of Program Design*. Academic Press (1975).
2. Joint IECCA and MUF Committee on MASCOT (JIMCOM), *The Official Handbook of MASCOT*. Her Majesty's Stationery Office (June 1987).
3. M. E. C. Hull, P. G. O'Donoghue and B. J. Hagan, MOON – modular object oriented notation. Accepted for publication in Software Engineering Journal.
4. L. Ingervaldsson, *JSP: A Practical Method of Program Design*. Chartwell Bratt Ltd (1979).
5. HOOD Working Group, *HOOD Reference Manual*, draft B issue 3.0 (June 1989).
6. J. R. Cameron, An overview of JSD. *IEEE Transactions on Software Engineering*, **12** (2) (February 1986).
7. G. Mullery, CORE – A method for controlled requirements specification. *Proceedings of the 4th International Conference on Software Engineering* (Sept. 1979).
8. M. A. Jackson, *System Development*. Prentice Hall International (1983).
9. E. Yourdon and L. L. Constantine, *Structured Design*. Yourdon Press (1978).