Proceedings of the 1994 Winter Simulation Conference
ed. J. D. Tew, S. Manivannan, D. A. Sadowski, and A. F. Seila

MULTI-LAYERED ACTIVITY CYCLE DIAGRAMS AND THEIR CONVERSION INTO ACTIVITY-BASED SIMULATION CODE

Kurt A. Pflughoeft

Department of Information and Decision Sciences
University of Texas at El Paso
El Paso, Texas 79968-0544, U.S.A.

Kiran Manur

Department of Manufacturing Engineering
University of Texas at El Paso
El Paso, Texas 79968, U.S.A.

ABSTRACT

Activity cycle diagrams (ACDs) have long been used for the representation of the flow of entities within discrete-event systems. They can be used to manually simulate systems and their description serves as the basis for automatically generating activity-based simulation code. Although activity-based approaches are cited as one of the easiest ways to model systems their proliferation has been hindered by concerns of efficiency, support, and flexibility. In this paper, these concerns are addressed as well as simulation methods that support the multi-layered ACDs to assist in overcoming these issues.

1 INTRODUCTION

Activity cycle diagrams (ACDs) graphically show the flow of entities through discrete-event systems. These diagrams are useful for not only manually simulating systems but also ACD descriptions can be used to generate their closely-related activity-based simulation code. Although activity-based approaches may be the easiest way to represent systems, they have not enjoyed the popularity of other approaches (Balci, 1988, Law and Kelton, 1991, Paul 1993).

In the past ACD's advantages, most notably ease of understanding, were outweighed by its inefficiencies. For each change of the simulation clock, most activities must be scanned for feasibility under an activity-based approach. (Some activities could be skipped since they may be mutually exclusive with other activities. Other activities could be eliminated because of other logical considerations -- see cellular designs in Clementson (1990).) The need to scan most activities decreases the model's run-time efficiency which was a major concern in the early days of computing. With the advent of today's more powerful cpus and languages, which greatly diminishes the above bottleneck, the loss in efficiency is minimal for many discrete-event problems. For example, an activity-based implementation of a manufacturing simulator in C has allowed experimental designs that a few years ago would have been considered impractical using this approach (Pflughoeft, 1993, Hutchinson and Pflughoeft, 1994). The simulator was used to conduct thousands of experiments to evaluate manufacturing configurations, quantify the benefits of process plan flexibility, and formulate superior combination scheduling rules via search and simulation techniques.

Other disadvantages of the ACD approach include cumbersome diagrams for complex systems and few simulation packages which support this methodology. A multi-layered ACD approach is introduced to simplify the graphical representation for complex systems. The conversion of ACD descriptions into C/C++ is also discussed as a viable technique to simulate the flow of entities as depicted by ACDs.

2 MULTI-LAYERED ACD'S

Activity cycle diagrams have been discussed by several authors including: Paul (1993), Hutchinson and Clementson (1991) and Carrie (1988). Briefly, these diagrams represent each type of entity flow by unique line patterns. Within the flow each entity should alternate between two states: waiting and processing. Wait states are denoted as queues and graphically represented by circles. Processing states, activities, are represented by rectangles. For an activity to be feasible there must be at least one entity in each queue that is linked to the activity. For example in Figure 1, which depicts a simple manufacturing process, the *GetTool* activity can take place only when there is at least one entity in each of the following queues: *tlFree*, *wpFree*, and *wsFree*.

Unfortunately, systems may contain many activities and queues which can result in ACDs which are large, cumbersome, and cluttered. For example, Figure 1 represents a simplistic manufacturing process and as

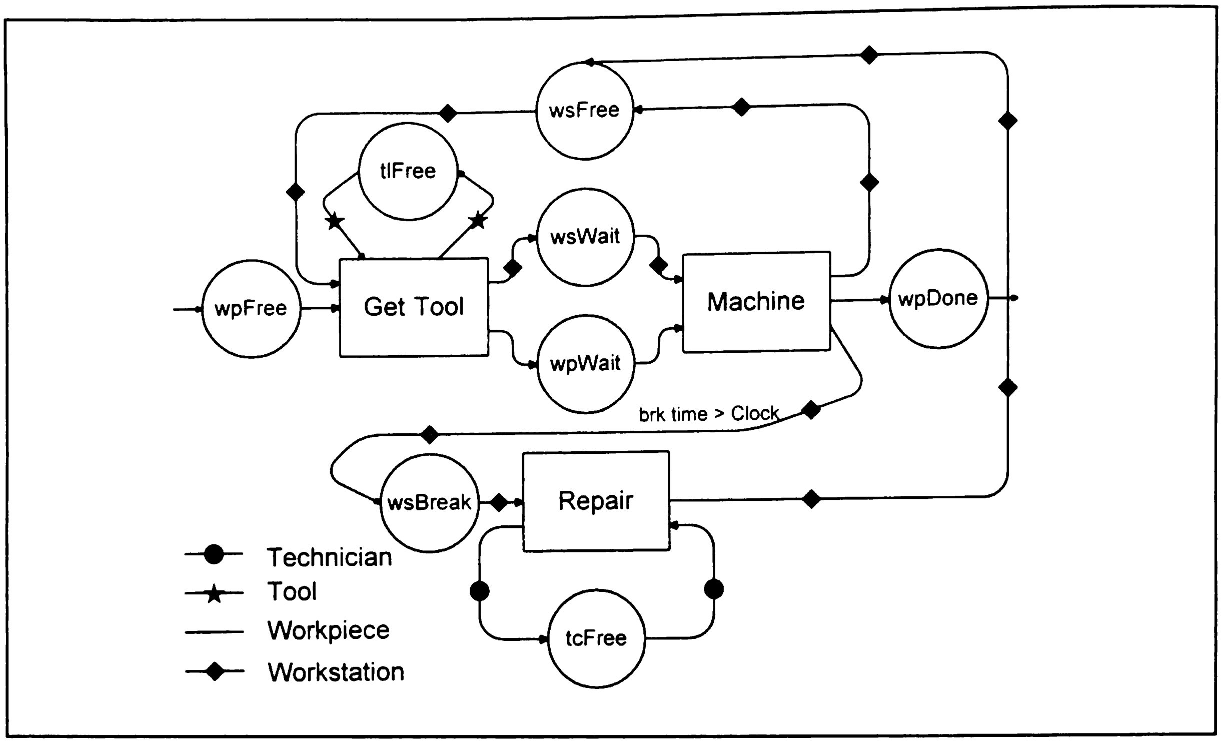

Figure 1: ACD for Workpiece Processing

the diagram is updated to model an actual system, the resulting ACD may contain hundreds of queues and activities. The standard approach in simplifying complex diagrams is to draw ACDs which show the flow of only one entity type. Thus, for Figure 1 we could have four separate diagrams for each entity type and their flows: **workstation** (*wsFree, GetTool, wsWait, Machine, wsFree* | *(wsBreak, Repair*, wsFree)), **workpiece** (*wpFree, GetTool, wpWait, Machine, wpDone*), **technician** (*tcFree, Repair, tcFree*), and **tool** (*tlFree, GetTool, tlFree*). When these individual diagrams are superimposed they would create the final ACD. The problem with this approach is that the individual diagrams do not show the interrelationship between entities and the final diagram requires the user to be versed with all the details of the system to be modeled.

In some situations the complexity of the final diagram can also be reduced by representing the flow of entities in other forms of logic such as conditional statements. However, the use of this approach must be limited if one wishes to maintain ease of understanding and not drift away from the ACD approach. Instead, a multi-layered ACD approach which decomposes the diagram by activities, instead of entity flows, is discussed. The multi-layered ACD approach is conceptually similar to explosion/implosion representation of processes within data flow diagrams (DFDs) (Gane and Sarson, 1979). DFDs allow each process to be represented at various levels of granularity as does multi-layered ACDs for activities. With a multi-layered approach the users, whether managers or engineers, are not forced to understand the system at the same level of detail. Allowing several levels of system representation simplifies understanding, expedites model verification/validation, and permits sensitivity analysis.

Figures 2 through 4 are an example of a multi-layered representation of Figure 1. Figure 2 shows the simplest interaction of entities required to process a workpiece, i.e., one workpiece and one workstation. Figure 3 is an explosion of the *Process* activity which reveals that this activity is subject to breakdowns. Figure 4 explodes the *Work* activity to show that the processing of a workpiece requires the selection of the appropriate tool and followed by the machining of the workpiece.

The development of ACDs is an integral part for increasing one's understanding of a system and simplifying the process of creating the required simulation code. The multi-layered variation of ACDs retains these characteristics as well as supporting various levels of understanding about a system.

Multi-layered ACDs, as with ACDs, do have some disadvantages. Both approaches are not comprehensive in the sense that they sometimes lack the capability of providing sufficient information to represent

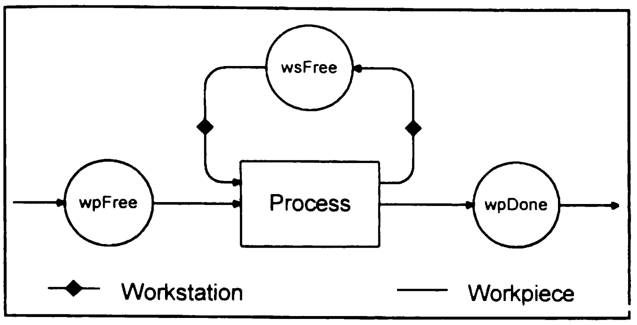

Figure 2: Abstract ACD of Workpiece Processing

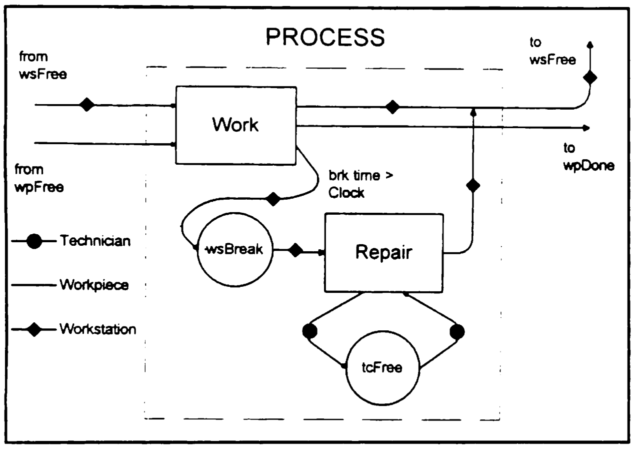

Figure 3: Explosion of the Process Activity

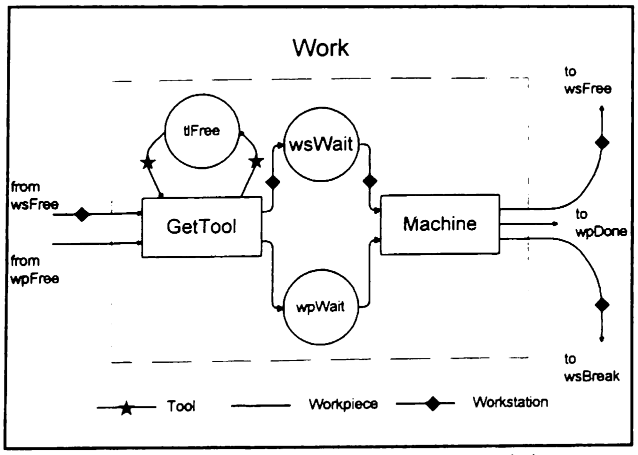

Figure 4: Explosion of the Work Activity

a system, e.g. different queuing principles such as LIFO and FIFO. This type of information and other logic which can't be easily portrayed via an ACD is best handled by other methods used together with ACDs. Multi-layered ACDs, which may be more suited to a prototype development approach than ACDs, also assume that other higher level activities are not changed by exploding a particular activity. For example, if one decides to explode the breakdown activity to allow it to occur after either the selection of a tool or machining, this could not be handled by simply exploding the repair activity. Instead, Figures 3 and 4 would have to be changed to reflect this situation.

3 GENERATING SIMULATION CODE FROM ACDS

As stated before the description of ACDs provides enough information to automatically generate simulation code. Activities can easily be represented as functions, queues as linked lists, and entity types as a hierarchical data structures. The only other requirement to generate the code is to have a simulation library which supports a clock. Below is a brief description showing a subset of the resulting C code generated from the multi-layered ACDs. The purpose is to give the reader an overview of this process as there are few packages which support this methodology.

As early as 1976, the CAPS component (Computer-Aided Programming System) of the ECSL (Extended Control Simulation Language) package provided the ability to automatically generate simulation code from ACD descriptions (Clementson, 1990). CAPS demonstrated that the above process is straightforward requiring little user training. However, CAPS is designed to work with single-layered ACDs and this package is relatively little known.

KBSL (Knowledge-Based Simulation Language) is a package that can translate ACD descriptions into either high-level compatible ECSL code or directly into C code. Using the multi-layered diagrams shown in Figures 2 - 4, the KBSL code for those diagrams is presented in Appendix 1. The code retains its modularity and allows the model to be run at several levels of representation. If efficiency is still a major consideration the *if* tokens, denoted by the explosion comment, can be replaced with corresponding *ifdef* precompiler statements. The cost of the latter technique requires recompilation for a change of the constant.

The multi-layered activity-based approach also works well for the generation of C++ code and is currently under development. The entity descriptions are converted from struct into class descriptions. However, for a single entity there will most likely be a base class followed by one or more derived classes. The class descriptions are determined by the hierarchical relationships that may be present between entities and the required entity attributes for each activity. For example, hierarchical relationships may include deriving workpieces from their respective orders. The determination of required entity attributes for activities is also used in building class descriptions to restrict which entity attributes can be accessed in an activity. These attributes and the activity itself become members of one of the classes.

4 CONCLUSION

Activity-based approaches for code and graphical representations of systems are a viable technique in the 1990's. Multi-layered techniques work well for complex systems and the ability to convert their description into C/C++ expands the scope of this tool. Models can be supported at various levels of granularity that expedites understanding and testing besides supporting a form of sensitivity analysis.

ACKNOWLEDGEMENTS

The authors would like to acknowledge George K. Hutchinson for his insightful ideas and comments regarding activity cycle diagrams.

APPENDIX: KBSL-GENERATED C CODE*

```
/* Entity & Queue descriptions */

extern struct TECH_tag {
  id_type id;
  } *TECH;

extern QUEUE *tcFree;

extern struct WP_tag {
  id_type id;
  } *WP;

extern QUEUE * wpDone, *wpFree, *wpWait;

extern struct WS_tag {
  id_type      id;
  clock_type break_time;
  clock_type repair_time;
  clock_type time;
  seed_type   time_seed;
 } *WS;

extern QUEUE  *wsBreak, *wsFree, *wsWait;

extern struct TOOL_tag {
  id_type  id;
  } *TOOL;

extern QUEUE *tlFree;
```

```
/* Activity Descriptions */

int process(void) {

GetTool();
Machine();
if(REPAIR_DEFINED)  /* Explosion */
  Repair();
}

int GetTool(void) {

clock_type duration; id_type  tool_sub,wp_sub,ws_sub;
cnt_type   GetTool_cnt=0;

GetTool_label:

if(find_first(&wp_sub,wpFree) == FALSE)
  return(1);
if(find_first(&ws_sub,wsFree) == FALSE)
  return(2);
remove_elem(wp_sub,wpFree);
remove_elem(ws_sub,wsFree);
duration=0;
if(TOOL_DEFINED) {  /* Explosion */
  if(find_first(&tool_sub,tlFree) == FALSE)
    return(3);
  else {
    remove_elem(tool_sub,tlFree);
    duration=Tool_time;
    add_elem(tool_sub,tlFree,duration);
    }
  }
add_elem(wp_sub,wpWait,duration);
add_elem(ws_sub,wsWait,duration);
if(++GetTool_cnt <= MAX_REPEAT)
  goto GetTool_label;
else
  error("GetTool - too many repeats");
}

int Machine(void) {

clock_type duration; id_type    wp_sub,ws_sub;
cnt_type   Machine_cnt=0;

Machine_label:

if(find_first(&wp_sub,wpWait) == FALSE)
  return(1);
if(find_first(&ws_sub,wsWait) == FALSE)
  return(2);
remove_elem(wp_sub,wpWait);
remove_elem(ws_sub,wsWait);
duration=neg_exp(WS[ws_sub].time,WS[ws_sub].time_
seed);
if(REPAIR_DEFINED) {  /* Explosion */
  if(WS[ws_sub].brk_time > Clock)
    add_elem(ws_sub,wsBreak,duration);
  else
    add_elem(ws_sub,wsFree,duration);
  }
else
  add_elem(ws_sub,wsFree,duration);
add_elem(wp_sub,wpDone,duration);
if(++Machine_cnt <= MAX_REPEAT)
  goto Machine_label;
else
  error("Machine - too many repeats");
}

int Repair(void) {

clock_type duration; id_type  tech_sub,ws_sub;
cnt_type Repair_cnt=0;

Repair_label:

if(find_first(&tech_sub,tcFree) == FALSE)
  return(1);
if(find_first(&ws_sub,wsBreak) == FALSE)
  return(2);
remove_elem(tech_sub,tcFree);
remove_elem(ws_sub,wsBreak);
duration=WS[ws_sub].repair_time;
add_elem(tech_sub,tcFree,duration);
add_elem(ws_sub,wsFree,duration);
if(++Repair_cnt <= MAX_REPEAT)
  goto Repair_label;
else
  error("Repair - too many repeats");
}
```

* Represents only a subset of the generated code needed for model execution.

REFERENCES

Balci, O. 1988. The Implementation of Four Conceptual Frameworks for Simulation Modeling in High-Level Languages. In *Proceedings of the 1988 Winter Simulation Conference*, ed. M. Abrams, P. Haigh, and J. Comfort, 287-295.

Carrie, A. 1988. Simulation of Manufacturing Systems. Chichester: John Wiley & Sons Ltd.

Clementson, A.T. 1990. *Extended Control and Simulation Programmer's Manual.* University of Birmingham.

Gane, C., and T. Sarson. 1979. *Structured System Analysis and Design Tools and Techniques.* Englewood Cliffs, N.J.: Prentice Hall.

Hutchinson, G.K. and A.T Clementson. 1991. Static Analysis of Systems - A Methodology Based on Timed Petri Nets. *International Journal of Production Planning and Control* 2, 1: 110-115.

Hutchinson, G.K. and K.A Pflughoeft. 1994. Flexible Process Plans: their Value in Flexible Automation Systems. *International Journal of Production Research* 32 , 3: 707-719.

Law, M.L. and W.D. Kelton. 1991. *Simulation Modeling and Analysis*, New York: McGraw-Hill.

Paul, R.J. 1993. Activity Cycle Diagrams and the Three Phase Method. In *Proceedings of the 1993 Winter Simulation Conference*, ed. G.W. Evans, M. Mollaghasemi, E.C. Russell and W.E. Biles, 123-131.

Pflughoeft, K.A., 1993. A Knowledge-Based Approach to Automate and Support Dynamic Decision Making for the Control of Complex Discrete Systems. Ph.D. Dissertation. University of Wisconsin, Milwaukee, Wisconsin.

AUTHOR BIOGRAPHIES

KURT A. PFLUGHOEFT is an Assistant Professor in the Information and Decision Sciences Department at the University of Texas at El Paso. He received his M.S. and Ph.D. degrees from the University of Wisconsin-Milwaukee in 1986 and 1993. He has worked as a systems analyst and computer consultant for numerous companies. His research interests include FMS design and scheduling, simulation, knowledge-based systems, information metrics, and object-oriented design principles.

KIRAN MANUR is a graduate student in the Department of Manufacturing Engineering. He received his B.S. from Bangalore University in 1990. His thesis addresses the conversion of ACDs into an object-oriented representations. His other research interests include database and interface design.