N91-22774

USING C TO BUILD A SATELLITE SCHEDULING EXPERT SYSTEM: EXAMPLES FROM THE EXPLORER PLATFORM PLANNING SYSTEM

David R. McLean and Alan Tuchman
Bendix Field Engineering Corporation

William J. Potter
NASA/Goddard Space Flight Center

ABSTRACT

Recently, many expert systems have been developed in a LISP environment and then ported to the real world C environment before the final system is delivered. This situation may require that the entire system be completely re-written in C and may actually result in a system which is put together as quickly as possible with little regard for maintainability and further evolution. With the introduction of high performance UNIX and X-windows-based workstations, a great deal of the advantages of developing a first system in the LISP environment have become questionable. This paper describes a C-based AI development effort which is based on a software tools approach with emphasis on reusability and maintainability of code.

The discussion starts with simple examples of how list processing can easily be implemented in C and then proceeds to the implementations of frames and objects which use dynamic memory allocation. The implementation of procedures which use depth first search, constraint propagation, context switching and a blackboard-like simulation environment are described. Techniques for managing the complexity of C-based AI software are noted, especially the object-oriented techniques of data encapsulation and incremental development. Finally, all these concepts are put together by describing the components of planning software called the Planning And Resource Reasoning (PARR) shell. This shell has been successfully utilized for scheduling services of the Tracking and Data Relay Satellite System for the Earth Radiation Budget Satellite since May of 1987 and will be used for operations scheduling of the Explorer Platform in November of 1991.

INTRODUCTION

The issue of "doing Artificial Intelligence (AI) in C" has been a topic of debate for a number of years now (Schildt, 1987). The primary motivation of this paper is not to demonstrate that it is possible to do AI in C but to demonstrate that there are definite advantages of doing AI in C. Because traditional approaches of software development (waterfall) have not emphasized its reusability, the products of this approach can only be utilized within a narrow range of applications. This is because the waterfall model does not accommadate the sort of evolutionary development made possible by rapid prototyping capabilities and forth-generation languages (Booch, 1991). Recently, NASA has been taking the software reusability issue seriously (Truszkowski, 1989) and there are those who argue that software reuse should be at the heart of the strategy for software maintenance (Longstreet, 1990). A related issue of concern is the need for software to accommodate change (Watson, 1990).

Because reusable software must accommodate changes in the desired behavior through easy reconfiguration, it also ensures that it is to some extent maintainable through the reconfiguration process. However, reusable software must also be fully integrated so that components can be added or deleted easily. Characteristics which improve maintainability include: use of a standard high-level language, modularity and standard coding conventions, which use meaningful names (Longstreet, 1990). Recently, object oriented languages have gone a long way toward allowing the software engineer to obtain the reusability goal.

Object oriented software development has evolved from the user interface technology which is often associated with AI (Goldberg, 1984). The availability of today's high performance workstations has allowed the software engineer to take advantage of some of the AI technology and put it to practical use. To the software engineer, AI technology is just another set of tools available to implement the requirements which eventually accomplish the desired software goals. However, it is easy to imagine the process of converting a LISP-based system to a C-based system as one being done as quickly as possible, without regard to the evolution of the C-based system. In any case, it is a major undertaking (Martin, 1990). Relying on vender support for modifications to an off-the-shelf AI shell may be undesirable and maintaining software which is composed of a mix of different kinds of languages is expensive (Schildt, 1987). Other developers and users of AI technology at JPL (Durham, 1990) have reported similar experience with software tools. To be more useful to the software engineer, AI tools should be part of an integrated set of software tools. Therefore, if a team of software engineers is to take full advantage of the new "AI technologies" it is desirable that the AI tools be directly accessible and written in the same language as the current tools.

Because maintaining software usually includes responding to new requirements and hence support of software evolution (Booch, 1991), the software reusability issue is even more important. Ideally, the software maintenance engineer will use existing software tools to modify a given system and change its capabilities. Therefore, the software tools should become part of the language through which new requirements are implemented. Because all software must eventually change or become obsolete, this kind of extensibility should be a primary goal of all software development projects. Software engineering teams which utilize this approach must become intimately familiar with existing software tools and libraries. This takes time and experience because gaining a working knowledge of the existing tools is similar to learning a new language. However, once this is accomplished, the engineers are in a position to develop systems in a fraction of the time that would otherwise be required. Thus, managers need to allocate time for new members of a software reuse team to learn the "new language" and also to value this expertise once it becomes available. There is a world of difference between an off-the-street C programmer and one who has learned to utilize software tools. It is important that a major effort toward this end be made so that generic software tools and reuse methodologies can be identified and utilized.

Getting started with the software tools approach may require that developers re-think some of their development paradigms because initial development may proceed from the bottom up. Some bottom up development is required because the lower level tools must exist before they can be utilized. Thus, the developers need to learn to think in terms of using and designing for reuse. This also means that software managers need to allow for reuse development and note that there is no need to generate an entire system from scratch. Also, because reusability developers will be using tools written by others, they will require some of the traits of the traditional maintenance engineers; humility and adaptability to the style and ideas of others (Parikh, 1986). With time, many of the distinctions between developers and maintainers may disappear.

This paper describes some of the development effort which has resulted in generic software tools, which include AI technologies, for use in solving scheduling problems. These tools are written in the C programming language (Kernighan, 1978) with an emphasis on object-like development methodology. C was chosen because of the primary maintenance goal of portability. When C++ (Stroustrup, 1986) class libraries become generally available (and reasonably standard), these tools will be re-written to takeadvantage of full fledged object oriented development methodology. The emphasis here will be on integrated AI tool development with examples which demonstrate how AI technology can be utilized with a traditionally non-AI language, such as C. Readers who are not interested in the implementation details may skip those parts without loss of continuity of the general methodology description. On the other hand, the detail reader will note that many of the AI paradigms which seem so exotic to the uninitiated can be implemented in a straight-forward manner.

GETTING STARTED

In 1985 a group of software engineers from Bendix Field Engineering Corporation, called the Interactive Experimenter Planning System (IEPS) group, were tasked with investigating AI tools and techniques to be utilized for a satellite planning system(McLean, 1987). The task started by looking for tools which might be useful, such as the language support libraries and other software currently available. Eventually, the IEPS software engineers created libraries for file I/O, string manipulation, date and time conversion and user interface tools. These user-defined libraries were written on top of the more or less standard language support libraries and have evolved continuously since their initial creation. The IEPS application developers, in turn, utilized these user-defined libraries (tools) to create prototypes and applications such as the Earth Radiation Budget Satellite System (ERBS) Tracking and Data Relay Satellite (TDRS) contact planning system (McLean, 1987).

For those readers who are farmiliar with LISP but not with C, the following example will demonstrate how some of the behavior of LISP can be simulated in C. In particular, this example shows how to simulate some of the behavior of the LISP primitives CAR and CDR. First consider a string which contains three tokens as follows:

first second third

In LISP, the first token is obtained by invoking the string with CAR and the remainder are returned by invoking CDR.

```
(CAR (first second third)) —> first
(CDR (first second third)) —> (second third)
```

Now consider a module written in C called "get_tok" which takes a pointer to a character string as the first argument, a string token buffer as the second argument and a character delimiter as the third argument. Get_tok also returns a pointer to the remainder of the string:

```
sptr = "first second third";
sptr = get_tok(sptr,token,BLANK);
```

Before invocation:

```
sptr —> "first second third"
```

After invocation:

```
token —> "first"
sptr —> "second third"
```

Because get_tok returns a NULL token when the end of the string is reached, it also provides iterative control as follows:

```
for(sptr =get_tok(sptr,tok,BLANK);
    *tok; /* while tok not empty */
    sptr = get_tok(sptr,tok,BLANK))
    do_something(tok);
```

All of the tools described in the subsequent sections utilize get_tok for string processing.

USER INTERFACE TOOLS

The IEPS user interface tools were designed to be used independent of the application and to be as portable as possible between the PC and workstation hardware platforms. The bulk of these tools reside in a library called MEXLIB (Menu-based EXecutive LIBrary) (NASA-GSFC, 1988) which was designed to utilize an AI technology called Menu-based Natural Language Understanding (Tenant, 1983). These tools allow the application developer to describe the grammar, through which the user interacts with the application, in terms of menus, forms and other user interface objects (widgets). Thus, an application need only read the grammar file in order to know how to interact with a user and invoke the appropriate objects for command line building. Other user interface tools, such as the Transportable Applications Environment (NASA-GSFC, 1990) and others which utilize X Windows MOTIF or OpenWindows, create C source code which then must be compiled and linked to the application. MEX technology avoids this by dynamically creating the objects which are specified by the grammar file.

The LISP DEFSTRUCT data abstraction mechanism allows the user to create data structures. C also provides this capability and MEXLIB is built on these structures. When a user interface grammar file is read, MEX tools dynamically allocate these internal structures and fill in the slots of information specified by the file. For example, if the internal structure is of type menu then the options of the menu are read into a linked list and the appropriate interactive widget is assigned to the method slot. As each MEX structure is built, it is put into a hash table so that it can be looked up quickly by name. Once all the MEX structures have been built, the structure whose name is "main" displays itself to the user and initializes the interaction. After the "main" object has completed its interaction with the user it adds information to a command line which is then parsed. The parser examines each token in the command line and uses a depth first search to "expand" those tokens which match MEX structure names. Expansion is done by invoking the appropriate object which adds more information to the command line.

In a way similar to deriving new classes from base classes, C lets the developer derive new structures from more primitive structures. A simplified MEX data structure can be built upon two other structures, **list** and **form**. List represents a linked list of character strings and is used to hold the options of a menu or the default values of a form. **Form** represents a template with fields to be displayed to the user. **Form**, in turn, is built upon another structure called **form_element**. Each form element has a field name, value, row and column information. **Form** uses an array of form elements to represent the various fields on the form, a template (character page) which is displayed to the user and an index which represents the current field being processed. Examples of the **list**, **form_element** and **form** data structures are given below:

```
struct list {
char *line;
struct list *next;
};

struct form_element {
char *field_name;
char *value;
int row, column;
};

struct form {
struct form_element f[MAXF];
char *temp[MAXLINE];
int current_field;
};
```

In addition to the these structures, the **mex** structure also contains the name of the structure, a title, the menu option selected, the row and column position and a pointer to the interface widget to be invoked.

```
struct mex {
char *name;
char *title;
int i,j;
char *selected;

struct list *list;
struct form *form;

char *(*interface)();
};
```

As a mex grammar file is read, mex data structures are dynamically allocated by invoking mexalloc which uses the standard C library malloc tool. Some of the members are then set to default values until more detailed information is read.

```
struct mex *mexalloc()
{
struct mex *object;
  object = (struct mex *) malloc(sizeof (struct mex));
  object->name = NULL;
  object->title = NULL;
  object->list = list_alloc();
  object->form = NULL;
  object->i = object->j = EMPTY;
  object->selected = NULL;
  return(object);
}
```

Notice that the **list** member invokes a user defined **list_alloc** to allocate its initial dynamic space but that the **form** member is set to NULL until it is known that it will be used. (Forms use the **list** structure for default values but simple menus do not use the form structure.) Because dynamic memory is allocated only on an as needed basis, it is conserved.

Much of the work of parsing the MEX grammar file is accomplished by use of the **get_tok** tool. However, once the mex structures have been built and put into the hash table, the main MEX parser can be invoked to build command lines. A simplified version of the MEX parser is given below and described in the following paragraph:

```
char *parse(line)
char *line;
{
char head[80];
char *tail;
char *select;
    if(!*line)
        return(NULL);
    tail = get_tok(line,head,BLANK);
    if(object = mex_get(head)) {
        if(object->l->next->line)
            select = object->interface(object);
        else
            select = object->l->line;
        object->selected = select;
        parse(select);
    }
    else
        add_tok(head);
    return(parse(tail));
}
```

Parse is given a character string (**line**) and if it is empty, the value NULL is returned. Otherwise, **get_tok** is invoked to obtain the first token in the string (**head**). Next, the value of **head** is looked up in the hash table to see if it is the name of a mex structure. If so, then the structure (**object**) is retrieved and its linked list is examined to see of there is more than one option (which would require user interaction). If this is the case, then **object's** user interface is invoked to return the option selected. (In the case of a form, all fields are returned.) Once the option has been selected, a pointer to its value is placed in the "selected" slot of **object** and **parse** is invoked again (depth first) with that selection. If **head** is not the name of a mex structure then it is added to the command line being built by invoking **add_tok**. Finally, **parse** is again invoked on the remainder of the original string (**tail**).

AN INFERENCE ENGINE

The inference engine developed by the IEPS group is called the Transportable Inference Engine version 1 (TIE1), (McLean, 1986). TIE1 utilizes MEXLIB tools for its user interface and is frame based (Minsky, 1975). Each frame represents a goal or concept which has a default value and a value which is to be sought by application of the rules of inference associated with the frame. The attributes which are referred to in the rules must be specified in the frame attribute list. Thus, frames consist of a frame name, a value, a default value, an attribute list and a rule list. A TIE1 Knowledge Base (KB) consists of a set of frames, one of which represents the goal and the remainder which represent subgoals.

Each simple rule represents a hypothetical instance of the goal or concept and is composed of a rule name which represents a potential value for the frame and attribute-relation-value triplets. For example:

N_eyes lt 8

(The number of eyes is less than eight.)

The attributes which make up the rules may be primitive (not decomposable) or they may represent other frames. Primitive attributes obtain their values by interacting with the user or by querying data bases. When a KB is to be used interactively, the KB engineer can specify the MEX-style user interfaces to be utilized for each attribute. Decomposable frame attributes obtain their values from the inference rules associated with its frame and thus represent the backward chaining component of the TIE1 architecture. Complex rules have additional attributes which are set to specified values when the rule is fired and thus provide the forward chaining capability of TIE1.

When TIE1 is invoked, the user specifies the KB to be used and the goal to be sought. TIE1 then reads and parses (via **get_tok**) the specified KB and dynamically allocates the data structures which represent each frame. After each frame is allocated, it is filled with the attribute and rule information specified in the KB and then placed in a hash table to allow quick look up by frame name. Finally, TIE1 considers the goal frame and starts the search for its value by testing each rule in this frame. In the simplified version of TIE1, the name of the first true rule is returned as the value of the goal being sought.

Because TIE1 uses MEX -style user interfaces for the primitive attributes, its frame data structures utilize a list of mex structures with their respective values to be sought:

```
struct alist {
  struct mex *ma;
  char *value;
  struct alist *next;
};
```

A rule list structure is also used and contains the name of the rule, a flag which is used during rule testing and an associated list of attribute-relation-value triplets:

```
struct rlist {
  char *name;
  int flag;
  struct list *triplet;
  struct rlist *next;
};
```

In addition to the attribute list and the rule list, each TIE1 frame structure also contains the name of the frame, its value (when known) and a default value:

```
struct tie {
  char *name;
  char *value;
  char *default;
  struct alist *alist;
  struct rlist *rlist;
};
```

A simplified TIE1 search algorithm, implemented in module "infer", which uses the TIE1 frame data structure is given below and described in the following paragraphs:

```
infer(tieobj)
struct tie *tieobj;
{
struct alist *a;
struct rlist *r;
struct mex *ma;
struct k *known;
int nhypots;
  if(known = get_known(tieobj->name)) {
      tieobj->value = known->value;
      return;
  }
  r = tieobj->rlist;
  for(nhypots=0; r->name; nhypots++, r = r->next)
      r->flag = TRUE;

  for(a=object->alist; a->ma; a = a->next) {
      ma = a->ma;
      if(known = get_known(ma->name))
          a->value = known->value;
      else
      if((newobj = tie_get(ma->name)) != UNKNOWN)
      {
          infer(newobj);
          a->value = newobj->value;
      }
      else
      a->value = user_select(ma);

      put_known(ma->name,a->value);
      nhypots=test_hypots(tieobj,ma->name,nhypots);

      if(nhypots == 0) {
          tieobj->value = tieobj->default;
          break;
      }
  }
  if(nhypots != 0) {
      for(r=tieobj->rlist; r->name; r = r->next)
          if(r->flag == TRUE)
            break;
      tieobj->value = r->name;
  }
  put_known(tieobj->name,tieobj->value);
}
```

Infer is passed the TIE1 frame data structure (**tieobj**) whose name is the goal being sought. Module **get_known** is invoked first to see if the value of that goal (attribute) is already known and if it is, it sets **tieobj's** value to that known value and returns. Otherwise, **tieobj's** rule list is accessed and the values of all the flag slots are set to TRUE. This has the effect of treating all the rules as contending hypotheses which are initially assumed to be true. Then, the frames attribute list (**alist**) is accessed and each attribute's (**a**) value is sought according to the following ordered strategies:

Look up the attribute's name in the known facts hash table via module get_known and then return the value found there.

Look up the attribute's name in the frame hash table via module get_tie and then invoke module infer again (backward chaining) to obtain the value.

Ask the user or a data base for the value of the attribute.

Once a value is obtained for an attribute, its value is put into the facts hash table via module **put_known**. Then module **test_hypots** is invoked to test each rule in light of the new information obtained. Module **test_hypots** sets each rule's flag according to the success or failure of each rule and returns the total number of true rules (hypotheses). If the number of true hypotheses is zero, then the goal value of the frame is set to the default value and the attribute check loop is exited. Otherwise, the search and test strategy is continued for the remaining attributes in the list.

When the attribute search and test loop is exited, a check is made to see if the number of true hypotheses is zero. If this is not the case, then a search is made to find the first true hypothesis and when found this rule's name is assigned to the frame value. Finally, the frame's value is added to the facts hash table.

HEURISTIC SCHEDULING

The heuristic scheduler developed by the IEPS group is called the Planning And Resource Reasoning (PARR) shell (McLean, 1989). PARR's interactive mode utilizes MEXLIB tools for user interaction and acts like an intelligent assistant to the user. In the batch mode, it simulates the behavior of an expert human scheduler which has heuristics for where activities are to be placed on a timeline. These heuristics include specifications for the priorities, durations and how often the activities are to be scheduled. In addition, the resources, constraints and conflict resolution strategies may be specified. All of these specifications are placed in a KB which describes the way the expert human scheduler would schedule each general activity type (**activity class**).

PARR's architecture is somewhat like a blackboard model (Engelmore, 1988) which builds an activity timeline on a global blackboard and utilizes agents to perform constraint checking, resource management and conflict resolution. When PARR reads the KB, it dynamically allocates internal structures which represent each activity class and fills the slots with the appropriate generic values, thus PARR is also considered a frame based system. PARR's activity class structure is given below:

```
typedef struct {
  int type;
  char *name;
  int priority;
  int repeat;
  long duration;
  int offset;
  int shiftable;
  struct list *resources;
  struct list *constraints;
  struct list *strategies;
  char *subnames;
  struct list *misc_info;
} ACLASS;
```

Given the background of examples discussed so far, most of the members of ACLASS should be self explanatory and have been explained in detail elsewhere (McLean, 1989,1990). An exception is **subnames** which is a string of optional subactivity names.

When PARR creates an instance of an activity class (**ACLASS**) it dynamically allocates a different internal structure which will contain the detailed scheduling information about that particular instance. Among other things, the **EVENT** structure, as it is called, consists of a new structure (**t**) which represent time (**start** and **stop**) and also a pointer to the KB structure (**ACLASS**) which is used to generate the instance. The additional members are **label** and **flag** which are used to store associated information such as orbit numbers, **reslist** which is a resource list and **subacts** which is an array of optional subactivities. The **last** and **next** members are used to link the instances of a given class so that they can be kept in time order.

```
struct t {
  long seconds;
  int date;
};

typedef struct event {
  ACLASS *ac;
  struct t start;
  struct t stop;
  char *label;
  char flag;
  struct list *reslist;
  struct event *subacts[MAXSUBS+1];
  struct event *last;
  struct event *next;
} EVENT;
```

When an instance (EVENT) is to be created, the information in the activity class is examined so that the start and stop time of the activity can be set. The PARR controller then consults as many as three agents; the constraint checker, the resource manager and the conflict resolver. When invoked, each agent examines the appropriate slot in the activity class structure and performs its specific task. Status messages are then returned after each of the agents has performed its task and the controller makes a decision as to how to proceed with the scheduling of that particular activity. When the activity has passed all its constraint checks and all its resources have been allocated, it is placed on the timeline. The internal representation of this timeline is an array of EVENT structures:

```
EVENT *timeline[MAXCLASSES];
```

The constraint checker uses a rule representation similar to TIE1 (attribute-relation-value triplets) but does not include the implicit backward and forward chaining capabilities because of the simplicity of this type of constraint check. If any rule is violated, a message is constructed which states the constraint rule that was violated and specifies the conflicting value, otherwise a status of OK is returned.

If constraint checking has been passed and resources are required then the resource agent is consulted which, in turn, consults the appropriate resource model. At present, PARR supports a simplified power model and two different types of tape recorder models. If any of the resource models consulted return a status other than OK, a message is built which explains why the resource allocation failed.

If either the constraint checker or the resource allocation agent returns a status other than OK then the conflict resolution agent is consulted. This agent consults the status message and the conflict resolution slot of the activity class and tries to resolve the conflict by either rescheduling the current activity or rescheduling the conflicting activities. To describe the implementation of all of these strategies is beyond the scope of this paper. However, a description of the general approach to conflict resolution may give some insight into how PARR manages conflict resolution by consulting the strategies list and the conflict messages returned from the constraint checker and the resource manager.

THE CONFLICT RESOLUTION AGENT

The following is a simplified version of the conflict resolution agent which is discribed in the following paragraphs:

```
resolve_conflict(ew)
EVENT *ew;
{
EVENT *rwndo;
struct list *strat;
int status;
int strategy;
char *duration, *newact;

strat = ew->ac->strats;
rwndo = get_resources(ew);
strat = next_strat(strat,&strategy);

  for(status = NOTOK;
      status == NOTOK && startegy != EMPTY;
      strat = next_strat(strat,&strategy)) {

  if(context(strategy,conflict_msg) != OK)
      continue;

  switch(strategy) {
      case START:
          start(&ew,rwndo);
          break;
      case END:
          end(&ew,rwndo);
          break;

      case BEFORE:
          if(before(&ew) == EMPTY)
              continue;
          break;
      case AFTER:
          if(after(&ew) == EMPTY)
              continue;
          break;
      case DELETE:
          if(delete(ew,conflict_msg) == EMPTY)
              continue;
          break;
      case NEXT:
          if(!next(&ew,rwndo))
              continue;
          break;
      case PRIOR:
          if(!prior(&ew,rwndo))
              continue;
          break;
      case DURATION:
          duration = get_duration(strat->line);
          next_time(&ew->stop,ew->start,duration);
          break;
      case BUMP:
          duration = get_duration(strat->line);
          bump_time(&ew->start,duration);
          bump_time(&ew->stop,duration);
          break;
      case ACTIVITY:
          newact = get_newact(strat->line);
          if(activity(ew,newact) == NOTOK)
              continue;
          else
          return(OK);
          break;
      case SHIFT:
          if(shift(ew,conflict_msg) == NOTOK)
              continue;
          break;
  }
  status = do_insert(ew);
 }
 if(status == OK)
    report_success(ew->start,ew->stop);
 return(status);
}
```

Resolve_conflict is passed the activity's data structure (ew) that contains its activity class (ac) with the list of conflict resolution strategies (**strats**). Initially, **get_resource** is invoked to return the event data structure (**rwndo**) which is the primary resource window (for example, Daylight view) used by this activity. Then, a loop is initialized which processes the strategies list while the status of each try is unsuccessful and strategies remain. In this loop, module **context** is invoked to determine the suitability of the strategy to be tried in view of the conflict message. For example, if the **BEFORE** strategy is to be used and the conflicting activity is a tape dump and the activity to be scheduled uses tape then the strategy may not be suitable because there probably won't be enough tape remaining just before a tape dump. If the context is not suitable then the strategy is skipped.

On the other hand if the strategy is suitable, the appropriate strategy handler is invoked so that the event structure can be modified accordingly. This modification usually includes changing the start and stop times of the activity. If this adjustment is not successful then the strategy is abandoned and control returns to the next strategy. If the strategy is successful then module **do_insert** is invoked with the adjusted start and stop times. **Do_insert** consults the constraint checker and the resource manager again and adds the activity to the timeline if all goes well. The status of **do_insert** is returned and processing continues depending upon its value. If the status is not OK then the next strategy is tried. If the status is OK then the loop is exited, a message is logged and **resolve_conflict** returns the final status.

The following is a brief description of the conflict resolution strategies used by PARR:

START

Reschedule the activity at the start of a specific resource window by setting the start time of the activity to the start time of the resource window. Alternatively, reschedule the activity at the start of the specified time.

END

Reschedule the activity at the end of a specific resource window by setting the start time of the activity to the end (stop time) of the resource window.

BEFORE

Reschedule the activity before the conflicting activity by adjusting the stop time accordingly.

AFTER

Reschedule the activity to occur after the conflicting activity by adjusting the start time accordingly.

NEXT, PRIOR

Reschedule the activity in the next or prior resource window by adjusting the start and stop times accordingly.

DELETE

Delete the conflicting activities. Start and stop times of the current activity are not adjusted. Care is taken not to delete an activity of higher priority or a required resource replenishment.

DURATION

Shorten the duration of the activity.

BUMP

Bump the start and stop times by a specified amount (plus or minus) to avoid the conflicting region.

ACTIVITY

Schedule an alternative activity instead of the current activity type. This strategy temporarily abandons trying to schedule an instance of the current activity class and tries to schedule an instance of another class. When successful, module resolve_conflict returns immediately with a success status. When not successful, the next strategy in the current activity class is tried. Switching to another activity class amounts to a context switch for controlling the behavior of PARR because each activity class contains its own heuristics which are used to create instances of a particular class. Thus, when module **activity** returns, the context of the current activity class (and strategies list) is restored.

SHIFT

Reschedule the conflicting activities. This strategy also does not change the current activity's start or stop times. When shifting is attempted, the activity class of the conflicting activity is examined to make sure that shifting is allowed. If shifting is allowed then the conflicting activity is temporarily deleted and the start and stop times are adjusted so that it may be rescheduled out of the conflicting range of the current activity. Then module **do_insert_resolve** is invoked with the conflicting activity's adjusted event structure. **Do_insert_resolve**, in turn, checks the constraints and resources for this adjusted activity and also consults with the conflict resolution agent if required. The case may be that more conflicts will occur and that the shifting strategy be applied again to resolve those conflicts. Thus, this type of conflict resolution demonstrates the constraint propagation problems which PARR attempts to solve by use of recursive application of context dependent strategies.

CONCLUSIONS

The C-based AI technology presented here is not only clearly possible but is in actual use (McLean, 1987). Because this AI technology is part of an integrated set of tools, the experienced software engineer can readily make use of it to build new applications. It is this merging of the AI technology with the standard tools and techniques of experienced software engineers which makes the AI technology so readily usable.

Traditional software development efforts take years to accomplish their goals and start the process by building the system components from scratch. The future requirements for NASA missions will be even more demanding in terms of the number, complexity and configurability of software. In order to solve these problems, software engineers and managers need to get serious about the software reuse issue. This means that not only do the engineers need to be aware of and design for reuse but also that managers allow for a methodology which supports this effort. This methodology includes building systems through reuse of existing software tools, through iterative refinement and prototyping.

The ERBS scheduling system has demonstrated the utility of the software tools approach to maintain an expert planning system (McLean, 1991). This software reuse approach is also being used to develop the Explorer Platform Planning System (EPPS) (McLean, 1990) which will be used by the flight operations team to schedule mission support activities. EPPS is being built by reusing and enhancing the ERBS scheduling system software tools. Although much of the engineering methodology for reuse technology has been defined, the management methodology is lagging and needs further exploration and development.

ACKNOWLEDGEMENT

The authors wish to thank Patricia Lightfoot at NASA-GSFC/Code 514 and Ellen Stolarik at Bendix Field Engineering Corporation for their support of this work. This work was supported by NASA contracts NAS5-31000 and NAS5-27772.

REFERENCES

Booch, G. (1991), **Object Oriented Design With Applications**, Benjamin/Cummings.

Durham, R., Reilly, N. B. and Springer, J. B. (1990), "Resource Allocation Planning Helper (RALPH): Lessons Learned," **Proceedings of the 1990 Goddard Conference on Space Applications of Artificial Intelligence.** Engelmore, R. and Morgan, T. (1988), Blackboard Systems, Addison-Wesley.

Goldberg, A. (1984), **Smalltalk-80: The Interactive Programming Environment**, Addison-Wesley.

Kernighan, B. W. and Ritchie, D. M. (1978), **The C Programming Language**, Prentice-Hall.

Longstreet, D. (1990), "Introduction," **Software Maintenance and Computers**, IEEE Computer Society Press.

Martin, R. G. (ed.), D. J. Atkinson, M. L. James, D. L. Lawson, H. J. Porta (1990), "Spacecraft Health Automated Reasoning Prototype (SHARP)," **A Report on SHARP and the Voyager Neptune Encounter**, JPL Pulbication.

McLean, D. R. (1986), "The Design And Application Of A Transportable Inference Engine (TIE1)," **Telematics and Informatics**, J. Liebowitz (ed.), Vol 3 No. 3.

McLean, D. R., Littlefield, R. G., and Macoughtry, W. O. (1987), "Defining and Representing Events in a Satellite Scheduling System: the IEPS (Interactive Experimenter Planning System) Approach," **Proceedings of the 1987 International Telemetering Conference** Vol 23.

McLean, D. R., Littlefield, R. G., and Beyer D. S. (1987), "A Exper Syste fo Schedulin Request fo Communication Link Betwee TDR an ERBS," **Telematic an Informatics** J. Liebowitz (ed.), Vol 4 No 4.

McLean D R. Yen W L (1989), "PS PARR Pla Specificatio Tool An Plannin An Resourc Reasonin

Shel Fo Us I Satellit Missio Planning," **Proceeding of the 1989 Goddard Conference of Space Applications of Artificial Intelligence.**

McLean, D. R., Page, B. J. Potter, W. J. (1990), "The Explorer Platform Planning System: An Application of a Resource Reasoning Planning Shell," **Proceedings of the First International Symposium on Ground Data Systems for Spacecraft Control.**

McLean, D. R. (1991), "Maintaining An Expert Planning System: A Software Tools Approach." To be published in **Institutionalizing Expert Systems: A Short Handbook for Managers**, J. Liebowitz (ed.).

Minsky, M. (1975), "A Framework for Representing Knowledge," **Psychology of Computer Vision**, P. H. Winston (ed.), McGraw-Hill.

NASA-GSFC (1988), **MEX Portabl Menu-Base Executive.**

NASA-GSFC (1990), **TAE Plus User Interface Developer's Guide.**

Parikh, G. (1986), **Handbook of Software Maintenance**, John Wiley & Sons.

Schildt, H. (1987), **Artificial Intelligence Using C**, McGraw-Hill.

Stroustrup, B. (1986), **The C++ Programming Language**, Addison-Wesley.

Tenant, H. R., Ross, K. M., Saenz, R. M., Thompson, C. W., and Miller, J. R. (1983), "Menu-based Natural Language Understanding," **Proceedings of the Association for Computational Linguistics**, MIT.

Truszkowski, W. (1989), "Prototype Software Reuse Environment at Goddard SFC," **Software Reuse Issues**, Proceedings of a workshop sponsored by NASA Langley Research Center.

Watson, W. (1990), "Introduction," Space Network Control Conference on Resource Allocation Concepts and Approaches Presentations at GSFC, NASA.