Advanced Software Development Workstation

Effectiveness of Constraint-Checking

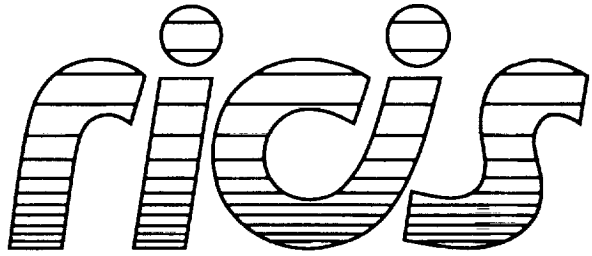
Michel Izygon

Barrios Technology, Inc.

July 1, 1992

Cooperative Agreement NCC 9-16
Research Activity No. SR.02

NASA Johnson Space Center
Information Systems Directorate
Information Technology Division



Research Institute for Computing and Information Systems
University of Houston-Clear Lake

INTERIM REPORT

The RICIS Concept

The University of Houston-Clear Lake established the Research Institute for Computing and Information Systems (RICIS) in 1986 to encourage the NASA Johnson Space Center (JSC) and local industry to actively support research in the computing and information sciences. As part of this endeavor, UHCL proposed a partnership with JSC to jointly define and manage an integrated program of research in advanced data processing technology needed for JSC's main missions, including administrative, engineering and science responsibilities. JSC agreed and entered into a continuing cooperative agreement with UHCL beginning in May 1986, to jointly plan and execute such research through RICIS. Additionally, under Cooperative Agreement NCC 9-16, computing and educational facilities are shared by the two institutions to conduct the research.

The UHCL/RICIS mission is to conduct, coordinate, and disseminate research and professional level education in computing and information systems to serve the needs of the government, industry, community and academia. RICIS combines resources of UHCL and its gateway affiliates to research and develop materials, prototypes and publications on topics of mutual interest to its sponsors and researchers. Within UHCL, the mission is being implemented through interdisciplinary involvement of faculty and students from each of the four schools: Business and Public Administration, Education, Human Sciences and Humanities, and Natural and Applied Sciences. RICIS also collaborates with industry in a companion program. This program is focused on serving the research and advanced development needs of industry.

Moreover, UHCL established relationships with other universities and research organizations, having common research interests, to provide additional sources of expertise to conduct needed research. For example, UHCL has entered into a special partnership with Texas A&M University to help oversee RICIS research and education programs, while other research organizations are involved via the "gateway" concept.

A major role of RICIS then is to find the best match of sponsors, researchers and research objectives to advance knowledge in the computing and information sciences. RICIS, working jointly with its sponsors, advises on research needs, recommends principals for conducting the research, provides technical and administrative support to coordinate the research and integrates technical results into the goals of UHCL, NASA/JSC and industry.

RICIS Preface

This research was conducted under auspices of the Research Institute for Computing and Information Systems by Dr. Michel Izygon of Barrios Technology, Inc. Dr. Rodney L. Bown served as the RICIS research coordinator.

Funding was provided by the Information Systems Directorate, NASA/JSC through Cooperative Agreement NCC 9-16 between the NASA Johnson Space Center and the University of Houston-Clear Lake. The NASA research coordinator for this activity was Ernest M. Fridge III, Deputy Chief of the Software Technology Branch, Information Technology Division, Information Systems Directorate, NASA/JSC.

Advanced Software Development Workstation

Effectiveness of Constraint-Checking
Interim Report

Prepared for
NASA-Johnson Space Center

July 1, 1992

Submitted by
Dr. Michel Izygon
Barrios Technology Inc.
1331 Gemini Av.
Houston, TEXAS 77058

ABSTRACT

This report summarizes the findings and lessons learned from the development of an Intelligent User Interface for a space flight planning simulation program, in the specific ara related to constraint-checking. The different functionalities of the Graphical User Interface part and of the rule-based part of the system have been identified. Their respective domain of applicability for error prevention and error checking have been specified.

Table of Contents

1. Introduction

A good user interface is critical to the successful use of a complex scientific application such as a space flight simulation, which typically involves very large sets of input data. Even an expert user may expend substantial effort to introduce the right data in the right manner. An Intelligent User Interface (IUI) uses knowledge-based technology to provide the user with the capability to easily prepare the input data without requiring prior extensive knowledge of the underlying software. An IUI is also commonly called a Knowledge-Based Front-End (KBFE). INTUIT (INTelligent User Interface development Tool) is a generic IUI shell that a knowledge engineer configures for a specific application by adding a knowledge base that includes input variable names which are immediately understandable by the users, the range of permissible data values, the structure and format of the data sets, and rules for error and consistency checking.

INTUIT has been used to develop an Intelligent Front-End (IFE) to a spaceflight design program named GNDSIM, which is used to simulate the rendezvous part of a shuttle mission. This report is aimed at summarizing the findings and lessons learned from the development of this IFE in the specific area related to constraints-checking. (Other papers describing other facets of INTUIT are given in the biblography.)

2. Constraint-Checking : An Overview

The goal of an IUI is to support the end-user of an application by taking over the tedious routine tasks, by providing assistance with the more complex tasks and by hiding the complexities of the underlying application from the user. Some of these complex tasks are related to the data integrity of the system. For example, in order to support the GNDSIM end-user, the main area where an IFE is very helpful is in preventing and checking user errors. In this section we will present what is involved in error checking.

2.1 Constraint-Checking Goals

Very often, in complex scientific applications, the end-user has to deal with a huge amount of interrelated data. As a consequence, he spends a lot of his time preparing the data and even more tracking his errors. An important feature of INTUIT is its constraints-checking. It provides the end-user with an automatic error checking mechanism. The main goal of the constraint-checking facility is to make sure that the user input are completely error-free and consistent before he executes the program. The practical benefits of the constraint-checking are :

- Increased productivity of the end-user who can spend his time working on his tasks rather than debugging the input data.
- The CPU time is not lost on useless executions caused by corrupted data.
- The user needs less training time, as the complexity of his tasks has been lowered.

The INTUIT constraints-checking capabilities result from the integration of Graphical User Interface (GUI) technology with Knowledge-Based technology. By mixing them we can cover a large range of potential errors. In fact, there are many different ways to ensure the integrity of the input data, depending on the type of error possible. We can prevent a mistake either by presenting only the correct values that an input can take, or by refusing a value if it is out of the allowable range. We can also accept a value but warn the user of a potential problem. We see that many errors can be prevented purely by developing a good user interface (UI). Characteristics of a good UI include good data organization and a self-explanatory way of presenting information to the end-user, i.e., in a language that talks to his domain expertise. When errors can not be prevented through the UI, then the knowledge base technology can be useful. Therefore we must first clearly understand what types of errors a good UI can prevent and what kind of errors it cannot prevent.

2.2 Scope of constraint-checking

• What are the different types of errors that are often found in these complex scientific programs?

An application program is considered by the end-user as a tool to achieve his domain specific goal. Very often this tool inherits some complexity from the domain it acts on; i.e., the number of inputs with which the end-user must deal is a direct function of the complexity of the domain and of the breadth of applicability of the tool.

We found in the literature many examples of complex programs that may be classified as error-prone. They are used in different areas such as Computational Chemistry, Ecological Modeling or Satellite Mission Support. It is interesting to analyze the reasons that make these programs error-prone for the end-user. The first possible reason is the large number of data to input. It is not uncommon to find a few tens to a few hundreds of inputs. It is impossible, for the end-user, to remember the meaning of each parameter, if their names do not convey their meaning clearly.

Another factor that makes these programs error-prone is the great difficulty associated with remembering the permissible values that a parameter can take, or its range if it is an integer or a real. Moreover, the parameters may be interrelated; i.e., the value of one parameter may affect the value of another parameter or its range, or its set of possible values .These relationships are hard to track and therefore often cause the end-user to overlook the implications that setting a parameter has on another parameter. These interrelationships may be in various forms : if a parameter has a certain value, another parameter has to be set, otherwise it does not have to be specified. Another type of interrelationship is found when the value of one parameter needs to be entered in multiple places within the input file.

• What errors can the GUI alone help to prevent?

Among the possible errors made by an end-user of a complex program, some can be prevented simply through the use of a good user interface. For instance, some presentation types such as radio buttons will not allow a wrong value to be input.

Consider an input parameter, for example a flag, that can take only a given number of values. A GUI can present this information to the user with a radio button or a scrollable list. Similarly, when the parameter has a given range of allowable values, a range checking facility in a GUI can handle this situation and thus prevent an out-of-range value from being input.

A GUI can also assume some part of the end-user's work such as tedious tasks or propagating the value of a parameter to every place it needs to be input. It can also handle hard formatting tasks.

Another standard feature of a GUI, On-line Help, can be very useful by simply reminding the user of the meaning of some parameter or explaining to a novice user what he is supposed to do at some point of his task.

• What errors cannot be prevented or checked by typical GUIs?

The more complex type of errors that we will describe now cannot be prevented or detected by a typical GUI.

A parameter may have multiple valid ranges. For instance, it might take any value between 0 and 25, and then between 50 and 100.

Interrelationship between variables is another type of error that cannot be handled by a typical GUI.

As a final example, when a block of input data is to be entered with a rigorous syntax, and cannot be simplified in order to remove the strict formatting, the GUI provides no assistance for such a task.

3. Constraint-Checking Mechanisms available in INTUIT

3.1 Rule Objects

In INTUIT, in order to ensure data integrity, as objects are created and modified, there exist modification rules within the system which can propagate new attribute values or check the validity of existing values. These rules are the source of INTUIT's complex constraint-checking capabilities. The knowledge-based technology chosen to implement them is ART-IM (Automated Reasoning Tool for Information Management) by Inference Corp. Modification rules in the knowledge base are themselves represented as objects (or schemas) and are transformed at initialization time into ART-IM rules. The definitions of the object class rule and the instance compile_rule, as supplied in kernel.art are as follows:

```
(defschema rule
(is-a Object)
(compile compile_rule_method)
(name "Rule"))

(defschema compile_rule
(instance-of rule)
(text "   (defrule compile_rule
          (declare (salience -100))
          (schema ?rule&~ compile_rule
          (instance-of rule) (text ?))
          =>
          (send compile ?rule))"))
```

At initialization of INTUIT, the ART-IM rule defined in the text slot of compile_rule is compiled. This creates a production rule, itself called compile_rule, which will cause the value of the text slot for each object which is an instance-of a rule to be compiled. In this way, rules are generated which can modify other objects in the knowledge base. More details about the INTUIT rule system are provided in the Appendix.

3.2 Constraints and Argument Lists

A subclass of the class rule is the class constraint. The definition of the constraint class is as follows:

```
(defschema constraint
(is-a rule)
(attribute)
(constraint-of)
(name "Constraint"))
```

The class constraint in turn has subclasses formula and predicate. A formula is an object used to specify how to compute the value of an attribute of a particular class of objects in terms of other attributes of that object and its subobjects. (A subobject of a given object is an object which is the value of an attribute of the given object.) For example, a class of objects of type phase has attributes day, hour, min, and sec, whose values represent the time since launch for the beginning of this phase of the mission. Phase also has an attribute, tevent, whose value is also the time since launch for the beginning of this phase of the mission, but tevent is expressed in seconds. Clearly tevent can be calculated from the values of the day, hour, min, and sec slots.

A predicate is used to specify how the system will verify relationships between the values of attributes of an object and its subobjects. A function is specified whose

arguments are these values and which returns T if the relationship constraints are satisfied and NIL if they are not. If the test fails, the object is marked as having a constraint violation.

Any formula or predicate schema must specify a value for the constraint-of slot. This value is the name of the object class to which the constraint is intended to apply. The computation specified by one of these constraints will only be invoked for objects which are instances of the specified class.

Formulas and predicates rely on user-specified functions, either (in the case of a formula) to compute an attribute value or (in the case of a predicate) to test for a constraint violation. The knowledge engineer must specify how to determine the arguments to these functions . This is done through the value of the arguments slot. This value must be a sequence of n elements, where n is the number of arguments which will be required by the user-specified function. Each element in the sequence must be either a symbol or a sequence of symbols, each of which is an attribute name.

The use of this specification is perhaps best illustrated by an example. Suppose a constraint is intended to apply to all objects in the class rendezvous, and that this class has attributes propagator-select and omp-model-select. The values of these attributes are intended to be instances of propagation-selection and omp-model schemas, each of which will have a text attribute, whose values are intended to be used as arguments to the constraint function. The following code fragment shows how this would be specified:

```
(constraint of rendezvous)
(arguments ( (propagator-select text) (omp-model-select text)))
```

A constraint function will be applied to a specific object only if all arguments for that function exist. In general, if the *i-th* element of the argument list is a symbol representing a slot, then the *i-th* argument for the constraint function will be the value of the slot in the object in question. If the *i-th* value of the argument list is a sequence of symbols, al, a2,..., then the argument is found by first taking the subobject which is the value of the slot al, then taking the value of the slot a2 in that subobject, assuming it also exists, etc. If the entire chain of subobjects exists, then the final element in the chain is the value passed to the constraint function. The number of arguments to a formula or predicate function is limited to 40.

3.3 Formulas

A formula schema results in the creation of a rule which is used to compute a value for some attribute of a particular object class. A formula schema is of the following form:

```
(defschema formula
(constraint of)    ;class to which formula applies
(arguments)
(attribute)        ;attribute for which value is computed
(function)         ;name of function used to produce attribute value
(name)             ;for documentation
```

The value of the function slot of a formula instance is a symbol representing the name of a user-supplied function. This is normally a def-art-fun, but could be a def-user-fun or an ART-IM system function.

In order for a formula to be applied, there must be an object in the knowledge base which is an instance of the class given by the value of the constraint_of slot. The values of attributes of this object and its subobjects as specified by the value of the arguments slot must exist. These values are then passed as arguments to the function specified in the

function slot. The return value from this function is then used to modify the value of the object attribute specified by the value of the attribute slot.

3.4 Predicates

A predicate schema results in the generation of a rule which is used to perform a procedural test on an object. If the test fails, the object is marked as having a constraint violation. This is done by asserting the name of the function as a value in the violates_constraint slot of the object.

A predicate schema is of the following form:

```
(defschema predicate
(constraint of)          ;class to which predicate applies
(arguments)
(boolean-function)       ;name of function to be used for test
(name)                   ;for documentation
```

As with a formula, the value of the boolean-function slot of a predicate instance is a symbol representing the name of a user-supplied function. This is normally a def-art-fun, but could be a def-user-fun or an ART-IM system function.

4. Types of constraints checked by INTUIT

In this section we will present the different types of errors that can be checked with the INTUIT constraints-checking capabilities. Some of these are directly derived from the GNDSIM experience.

Many problems faced by the end-user were caused by the difficulty to remember the meaning, the allowable values, the range and the type of the different parameters. Applying pure GUI techniques allowed us to solve these problems. What we are interested in analyzing in this section are the remaining types of errors, which we could not solve with a GUI alone.

4.1. Phase Ordering

In GNDSIM, the user has to enter a list of phases corresponding to each shuttle trajectory modification. One of the problems faced by the end-user is that the phases must be ordered according to their phase number. Moreover, in some phases the time at which the event occurs has to be defined, and generally this time value has to be greater than the time value of any preceding phase. Therefore we have two separate constraints that must be satisfied. The first one, phase ordering according to phase number, is a hard constraint, i.e. it must be enforced. The second one, phase ordering according to the time of the event is a soft constraint; i.e., it should be checked and reported to the end-user but not enforced by the system (It is possible, but rare for GNDSIM to propagate backward in time). In order to implement these two constraints we used two different approaches. For the first one, we used a mechanism available in INTUIT that automates the phase ordering according to phase number, thereby preventing this error. The mechanism used is called multi-valued slot ordering. When a multi-valued slot is specified for an object, there is the option of specifying a function which will be used to order the values in this slot for display and in formula functions. As shown below, this ordering function helps to order the multiple values (in this case, phase objects) found in the slot PHASE_ of the object RENDEZVOUS according to the value of the slot PHASNUM that is found in each of the phase objects listed in PHASE_. Thus, the mis-ordering of phases is not possible any more. Specifically, phase_-ordering-function compares the values of the phasnum slots of two phase objects, and INTUIT then uses the results of all these pairwise comparisons to correctly order the phase objects listed in PHASE_.

```
(def-art-fun phase_-ordering-function (?s1 ?s2)
    (bind ?valid1 (and (symbolp ?s1) (schemap ?s1) (slotp ?s1 phasnum)))
    (bind ?valid2 (and (symbolp ?s2) (schemap ?s2) (slotp ?s2 phasnum)))
    (bind ?t1 (if ?valid1 then (get-schema-value ?s1 phasnum) else NIL))
    (bind ?t2 (if ?valid2 then (get-schema-value ?s2 phasnum) else NIL))
    (if (not ?t1) then
        (if ?t2 then -1 else 0)
    else (if (not ?t2) then 1
        else (if (< ?t1 ?t2) then -1
            else (if (eq ?t1 ?t2) then 0
                else 1)))))
```

For the second constraint concerning ordering the phases according to time, we used a regular formula mechanism described in section 3.3. A warning message is displayed that the user has to acknowledge. The user may still misorder the phases according to the time slot, which is desirable in the case of a soft constraint. The formula mechanism currently only handles soft constraints. It allows a warning to be issued but does not enforce hard constraints; i.e., it does not force the user to resolve a real error. Other mechanisms must be used to enforce hard constraints. The formula schema and the associated function for this phase time constraint are shown below:

```
(defschema formula_order_phase
    (instance-of formula)
    (arguments (name))
    (attribute violates_constraint)
    (constraint_of rendezvous)
    (function check-phase-time-order))

(def-art-fun check-phase-time-order(?name)
    (bind ?rdv (read-from-string ?name))
    (bind ?phases (get-schema-value ?rdv PHASE_))
    (bind ?ordered-phases
        (reorder-values ?phases phase_-ordering-function))
    (bind ?knt 1)
    (bind ?violcon NIL)
    (bind ?kntmax (length$ ?ordered-phases))
    (for ?p in$ ?ordered-phases do
        (if (= ?knt ?kntmax) then
            (bind ?knt (- ?knt 1)))
        (if (/= (get-schema-value ?p covarmat) NO_PERTURBATION) then
            (bind ?t1 (get-schema-value ?p tevent))
            (for ?k2 from (+ ?knt 1) to ?kntmax do
                (if (and (/= (get-schema-value (nth$ ?ordered-phases ?k2) covarmat) NO_PERTURBATION)
                    (> ?t1 (get-schema-value (nth$ ?ordered-phases ?k2) tevent))) then
                    (bind ?violcon order-of-phase-time-is-bad)
                                (bind ?t2 (get-schema-value (nth$ ?ordered-phases ?k2) tevent))
                                (bind ?p2 (nth$ ?ordered-phases ?k2))
                    (bind ?msg (sprintf "\nWARNING: inconsistency in object %a \ntevent, %a, \nin phase
                      %a \nis greater than tevent, %a, \nin the phase %a " ?rdv ?t1 ?p ?t2 ?p2))
                    (add-warning-item ?rdv order-of-phase-time-is-bad USER NIL ?msg NIL)
                    (printf NIL "%a" ?msg)
                )
            )
                    (bind ?knt (+ ?knt 1))
        else (bind ?knt (+ ?knt 1))
        )
    )
    ?violcon)
```

The apparent complexity of the function is caused by the fact that all the phases do not have a time of event that is meaningful. The function has to check for the phases where the value represents a meaningful time of event, and to apply the check accordingly.

It should be mentioned here that the way ordering is handled by the system is not completely satisfactory. The problem arises from the fact that the ordering parameter has to be an attribute of the phase object schemas. This has the following implication: whenever the end-user wants to change the order of a given phase object, and assuming that this schema is a saved object, the end-user will have to edit this phase object, to modify the value of the ordering parameter, then to create a new object by modifying the phase object name, and finally to change the pointer in the PHASE_ slots so that it points to this new phase object. When this type of task needs to be often performed, the end-user will rapidly find the number of steps involved to be a real burden for him. Other solutions should be researched in order to offer an easier way of ordering a set of objects.

4.2 Complex Range Checking

It can happen that a parameter has a more complex valid range than just from value A to value B. It can, for example, have a multiple (i.e., disconnected) valid range such as 0 to 10 and 20 to 50. Another possibility is that the range of the parameter depends on the value of another parameter. A third and slightly different situation exists in GNDSIM, where a set of parameters can take any value within a single (i.e., connected) valid range, but the user has to be warned if one of the values is different from its default value. These three types of range checking can be easily implemented with the available INTUIT mechanisms. We will now describe how this can be achieved.

4.2.1. Multiple Range.

The multiple range can be resolved using the Predicate mechanism introduced in section 3.4. The associated function checks the value of the specified parameter against the allowed ranges. If the parameter is out of range, the Predicate returns a constraint-violation value for the object, thus prompting a warning message to be displayed. Here is an example of such a multiple range check predicate, that applies to the attribute "attribute_1" of the class "obj_a", and for which the valid range is [liminf1,limax1]or[liminf2,limax2].

```
(defschema multi-range_predicate
(instance-of predicate)
(constraint_of  obj_a)
(arguments  attribute_1)
(boolean_function  check-multi-range))
```

The associated function is :

```
(def-art-fun check-multi-range (?arg)
  (if (or (and (> ?arg liminf1) (< ?arg limax1)) (and (> ?arg liminf2) (< ?arg limax2)) then T else NIL))
```

The range values (liminf1, limax1,liminf2 and limax2) can be entered directly in the function, or input as attributes of the object, and passed as arguments to the predicate function.

4.2.2. Range Dependency.

This situation represents the case when the valid range of an attribute is modified according to the value of another attribute. For example, the range of attribute A is from liminf1 to limax1 when the attribute B has a value of n1 and from liminf2 to limax2, when B is equal to n2. In order to implement such a range check, we can use either a Formula or a Predicate. In both cases we assume that the attributes A and B belong to the same object.

If these attributes belong to different objects, the constraints-checking would be implemented as explained in section 4.3.2 on Interrelationship of parameters. The way to solve this problem is very close to the previous example, but a slightly different function is associated with the predicate.

```
(defschema  range-dependent_predicate
(instance-of predicate)
(constraint_of  obj_a)
(arguments  attr_A  attr_B)
(boolean_function  check-depend-range))
```

The associated function is :

```
(def-art-fun check-depend-range (?A ?B)
  (if (or  (and (and (> ?A liminf1) (< ?A limax1))  (= ?B n1))
           (and (and (> ?A liminf2) (< ?A limax2))  (= ?B n2))
           then T else NIL))
```

When the condition on these attribute values is violated, the predicate returns a NIL and then INTUIT sets the value for the attribute "constraint-violation" of the object "obj_a", thus prompting a warning message to be displayed.

4.2.3. Allowed Range, with warning if non default-value.

The third of these complex range checking capabilities is aimed at informing the end-user that an attribute has a correct value but is not equal to its specified default value. This type of check is solved via a different feature of INTUIT. It is possible to define a default value for each parameter and also a default and a non-default color for any panel. If one of the parameters included in such a panel is assigned a value different from its default value, the panel color is changed to its non-default color, thereby indicating to the end-user that a parameter has a different value from its default value. The schemas involved in this type of constraint-checking are as follows:

```
(defschema montecarlo_initialization
     (is-a application_component)
     (has-form-specification form-for-mcinit)
     (jran 1)
     (mtcp 1)
     (nran1)
     (nref 2)
     (nsam 30)
     (name "montecarlo_initialization"))

(defschema panel-for-mc_init
     (instance-of panel-spec)
     (has-item-specs jran-spec mtcp-spec nran-specs nref-specs nsam-specs)
     (panel-id)
     (default-background-color "white")
     (non-default-background-color "red"))
```

We can see that in the definition of the class montecarlo_initialization, the attributes jran, mtcp, nran, nref and nsam are initialized at their default values. If any of these parameters takes a value different from the defined default value, the corresponding panel will change from its default color, white, to its defined non default color, red. This allows the user to be aware of the change without displaying a warning message that the user would have to acknowledge.

4.3 Interrelation of Parameters.

One of the main areas where GUI techniques cannot ensure data integrity is where the input parameters are interrelated, i.e., the value of one parameter is dependent on the values of other parameters. This is a known reason for many errors, as the end-user has to make sure that the parameter values are consistent. The mechanisms provided by INTUIT, described in section 3, allow to eliminate this source of errors. Two different cases can occur. The first one occurs when the attribute that is being calculated and the attributes it depends on are encapsulated within the same object. The second case occurs when the attributes cannot be encapsulated in the same object.

4.3.1. Attributes encapsulated

The way to implement this type of constraint is to employ the formula mechanism as explained in section 3.3. The arguments of the formula are the parameters on which the calculation is based, and the attribute of the formula object is the parameter being derived. The formula is attached to the object containing all the attributes. As an example, consider an object "phase" that contains an attribute tevent that is derived from the values of the attributes day, hour, min and sec belonging to the same class:

```
(defschema phase
(is-a application_component)
(ph_number)
(ph_title)
(icoast)
(covar_mat)
(day)
(hour)
(min)
(sec)
(tevent)
(name "Phase"))
```

The following is a formula designed to compute the value of the slot tevent, given values for the day, hour, min, and sec slots:

```
(defschema phase-formula
(instance-of formula)
(arguments(day hour min sec))
(attribute tevent)
(constraint-of phase)
(function calc-sec))

(def-art-fun calc-sec (?day ?hr ?min ?sec )
(+ (+ (+ (* (* ?day 24) 3600) (* ?hr 3600)) (* ?min 60)) ?sec))
```

4.3.2. Attributes not encapsulated

If the parameters cannot be encapsulated in the same object, the solution is trickier to build. Basically, in order to apply one of the available mechanisms, there needs to be some way to relate all the objects containing all these attributes. This is done through the object architecture; i.e., there must be an object whose attribute values point to all of the objects containing these attributes. This type of problem occured in the GNDSIM knowledge base. In order to convert the times of the phases from Mission Elapsed Time (MET), into Greanwich Mean Time (GMT), there needs to be an object which can access both the phase time and the launch time. As these parameters are attributes belonging to the classes Phase and Simulation_definition, respectively, we had to attach the required

Formula to the Rendezvous class which has two attributes that point to objects belonging to the Phase and Simulation_definition classes:

```
(defschema rendezvous
 (is-a application_name)
 (first_phase_init)
 ----------
 (simulation_def)        ; this attribute points to the class Simulation_definition
 ----------
 (phase_)                ; this attribute points to the class Phase
 (text)
)

(defschema Simulation_definition
 (is-a application_component)
 (day)
 (hour)
 (min)
 (sec)
 ---
 (btime))        ; this is the launch time in seconds after midnight

(defschema Phase
 (is-a application_component)
 (day)
 (hour)
 (min)
 (sec)
 (phasnum)
 (phastit)
 (tevent))       ; this is the time of the phase in MET
```

The INTUIT system does not allow the attribute tevent of class Phase to determine any information about the value of the attribute btime of the class Simulation_definition. Therefore it is in the class Rendezvous that these two parameters will be merged in order to calculate tevent in GMT. The result is placed in the text1 attribute of the class Rendezvous. The Formula that does this calculation is therefore attached to Rendezvous. It is as follows:

```
(defschema rendezvous-formula
(instance-of formula)
(arguments ((simulation_def) (phase_) ))
(attribute text1)
(constraint-of rendezvous)
(function calc-gmt))

(def-art-fun calc-gmt (?simdef ?phases)
(bind ?stream (open-string NIL "w"))
(bind ?simtime (get-schema-value ?simdef btime))
(bind  ?ordered-phases (reorder-values ?phases phase_-ordering-function))
(bind  ?knt 1)
(for  ?p in$ ?ordered-phases do
        (bind ?tevent (get-schema-value ?p tevent))
        (bind ?gmtime (+ ?tevent ?simtime))
        (printf ?stream "%a\n%a" (get-schema-value ?p text) ?gmtime)
        (bind ?knt (+ ?knt 1)))
(bind ?string (get-stream-string ?stream))
(close ?stream)
?string)
```

5 Types of constraints that cannot be checked

During the development of the GNDSIM knowledge base, we came across a type of error that could not be prevented through good GUI techniques, nor checked by the rule-based system. This type of error involves a parameter that must be entered through a multiline free-format editor. This parameter is a sequence of events which represents the ordered list of burns that is required to accomplish a given rendezvous. This parameter has the following typical format:

```
Sequence Definition
DO EXDV AT T= 12345.5, DVLV = 0.0, 0.0, -0.5
DO EXDV AT DT= 2. , DVLV = 1.0, 0.0, 0.0
DO NC AT M= 2.0
DO NSR AT M= 9., DR = -40.0
DO NC AT DT= 20.0, LITM = -2.0
DO NH AT WT= 180.0
DO EXDV AT WT= 315.0, DVLV = 0.0, 0.0, 0.0
DO TPI AT WT= 225.0, DR = -8.0, DH= -0.2
DO TPF AT WT= 320.0,
```

We notice that it has a strict syntax and a number of reserved words. In order to make sure that the input sequence is correct, we would need to look for any typographical error in any of the reserved words used, and also to check for the syntactic correctness of each line. Moreover, some engineering rules could also be applied to check that the input plan is correct; e.g., if a given type of burn is performed, what should the next maneuver be and how much time should separate the two events?

The INTUIT system cannot currently check for this type of error because the ART-IM language does not have any input/output function that can be used to parse a line into words or characters. This is the only way we could have checked for possible typographical errors or syntax errors. In order to solve this problem, one could write a C function that would parse the sequence of events. The question that arises is how generic can this function be in order to be used for a broader type of syntax checking?

It should be pointed out that we did not search specifically for errors that cannot be checked, and therefore we cannot provide an exhaustive list of these. There are probably more types of errors that INTUIT cannot prevent nor check, and the best way to find them is to develop new applications.

6 Constraint checking : Lessons learned.

Let us now summarize the lessons learned regarding constraint-checking from the first INTUIT knowledge base development.

The application chosen, GNDSIM, was probably a good choice for developing the first knowledge base: we learned about the process and about the capabilities of INTUIT, as well as about the constraint-checking mechanisms. Nevertheless, GNDSIM was probably too simple an application in terms of errors to be prevented or to be checked because most of the problems faced by the end-users could be solved simply by using GUI technology. In order to use the rule-based system of INTUIT at its full potential, a more complex type of application is needed, one in which the dependencies between the variables are more numerous and more complex. Applying this technology to a more difficult system is certainly the next step we need to take, in order to thoroughly test its power and its eventual shortcomings. This might also help us address the performance issue, when the number of constraints to be checked is over one hundred.

One of the advantages of INTUIT is that the target system is completely modeled with objects. This provided a lot of flexibility for developing the constraints and for extending them, as our understanding of GNDSIM's end-user errors improved.

Introducing the constraints in the knowledge base using the available mechanisms proved to be simple enough. However, it should be noted that one of INTUIT's declared advantages, which is the possibility of easily entering everything as objects so that the knowledge engineer need not know the ART-IM language, proved partially erroneous. In fact, entering the rules as objects does not isolate the knowledge engineer from ART-IM, it just moves the problem one step into the background, because the formulas' or predicates' associated functions have to be written using ART-IM. Therefore, the internal complexity added to INTUIT in order to translate the rule objects into actual rules at compile time, is questionable. Writing the rules directly would probably have significantly simplified the INTUIT system without adding too much complexity to the knowledge engineer's tasks.

One of the questions that we still need to answer is whether or not the available mechanisms can be adapted to any type of constraints. We have found that some errors cannot be checked, using formulas or predicates. For instance, the syntax of free-format text cannot be parsed. The eventual solution to such a problem is to write a new C function that will allow us to retrieve the text and to check both its spelling and its syntax. How difficult this task may be and how difficult it is to integrate the new function with the available constraint-checking mechanisms is still to be assessed.

7 Conclusions

INTUIT, the INTelligent User Interface development Tool, has been used to develop an intelligent front-end (IFE) to a complex space flight simulation program. The errors that can be prevented or reported, and the constraint-checking capabilities of the system have been analyzed. The different functionalities of the GUI part and of the rule-based part of the system have been identified. By combining these two technologies, we proved that it is possible to significantly help the end-user of a complex program in the preparation of the input data. Introducing into the system the rules that check for the errors or the constraints appeared to be relatively easy, even though it required a fairly good knowledge of the ART-IM language. We also found out during this first IFE development, that there is a class of errors (free-format text entry) that the system cannot currently detect. The next phases of our research in this domain should include the following steps:

- Develop an IFE for a target application in which the constraint-checking is more numerous and more complex to implement than those developed for GNDSIM. We need to find new types of constraints and assess how INTUIT can handle them.
- Develop an IFE (perhaps the same application mentioned above) with a high number of constraints (above one hundred), in order to assess the performance of the system.
- In parallel, more effort should be devoted to evaluating the capabilities of a different type of GUI, such as GPIP, in order to precisely evaluate the added value provided by a rule-based system over a regular GUI system that has been enhanced with some IF-THEN type of logical constructs.
- Assess ways of automating the input of the functions associated with the formula and predicate objects, as part of a general effort to study how to ease the knowledge engineer's work.
- Finally, possibilities of deriving an intelligent assistant from the current system should be investigated. This would help the end-user by providing him a way to ask the system about pre- and post-constraints associated with a given parameter.

Appendix. Details of the INTUIT Rule System.

The INTUIT Rule System, allows the constraint-checking to be input through simple schemas, that the system transforms into rules. The process of this transformation is explained here. It follows a five steps process shown on Figure 1.

1 In access.c, in the main program, during the initialization there is the following instruction:
a_send(a_art_symbol("compile"),a_art_symbol("compile_rule"),NULL)
This C function,a_send, executes the ART function which is pointed to by the attribute "compile" of the object "compile_rule". The schema "compile_rule", shown below, has an attribute "compile" inherited from its parent class "rule", which value is "compile_rule_method".

```
(defschema rule
(is-a Object)
(compile compile_rule_method)
(name "Rule"))
```

```
(defschema compile_rule
(instance-of rule)
(text "   (defrule compile_rule
          (declare (salience -100))
          (schema ?rule&~compile_rule
          (instance-of rule) (text ?))
          =>
          (send compile ?rule))"
))
```

In the file artfuns.art there are user defined functions (def-user-fun) that act as pointers to C functions. In particular, there is a def-user-fun compile_rule_method that points to a C function called also compile_rule_method which is in the file accessfns.c. The a_send statement calls this function and passes the schema name ("compile_rule") as an argument.
The C function compile_rule_method extracts the value of the text slot from the schema "compile_rule". The value of this slot, "defrule compile_rule...." is a rule that is put in the active knowledge base.
2 Once in the knowledge base this rule is firing on each schema that is an instance-of rule. For each of these schemas, this rule sends a compile message (the right-hand side of the rule is :send compile ?rule) which means that the function compile_rule_method is executed on each of them. So, for example, the schema generate_class_restriction_rule, which is an instance of rule, is compiled into a rule that is the value of its text attribute: defrule generate_class_restriction_rule.

```
(defschema generate_class_restriction_rule
 (linstance-of rule))
(compile compile_rule_method)
 (text "(defrule generate_class_restriction_rule\n\t
        (logical\n\t(schema ?attribute-restriction\n\t\t
        (instance-of attribute-restriction)\n\t\t
        (allowable-classes $?x &: (> (length$ ?x) 0))\n\t\t
        (attribute ?attribute)))\n\t\n\t
        (test (not (instance-of-p ?attribute-restriction mandatory-attribute-restriction)))
        =>\n\t
        (bind ?rule-name (sprintf \n\t\t\"%a-VIOLATION-RULE\" ?attribute-restriction))\n\t
        (bind ?violation-name (sprintf \n\t\t\"%a-VIOLATION\" ?attribute-restriction))\n\t
        (bind ?string (sprintf \n\t\t\"
        (defrule %s \n\t(logical (schema ?s \n\t\t(instance-of object)\n\t\t(%a ?val)))
                =>\n\t(if (not-instance-of-allowable-class ?val %a)
        then\n\t(assert (schema ?s \n\t\t(violates_constraint %s)))\n\t\t(add-warning-item ?s %a
                %a %a ?val ALLOWABLE-CLASSES)))
        \" \n\t\t?rule-name \n\t\t?attribute  \n\t\t?attribute-restriction \n\t\t?violation-name
        \n\t\t?violation-name \n\t\t?attribute-restriction \n\t\t?attribute))\n\t
        (bind ?temp
        (read-from-string \n\t\t(sprintf  \"%a-GEN-RULE\" ?attribute-restriction)))\n\t
        (assert (schema?temp\n\t\t(instance-of generated-rule)\n\t\t(TEXT ?string))))")
(VIOLATES_CONSTRAINT))
```

3 Once in the knowledge base, this rule fires for each schema which is an instance-of attribute-restriction. It generates a schema, instance-of generated_rule, which includes the attribute compile that has the value compile_rule_method. For example, the slot "vehicle_init" is being restricted to the class "vehicle_initialization". This is the way, the knowledge engineer input this restriction:

```
(defschema vehicle_init-restriction
  (instance-of attribute-restriction)
  (allowable-classes vehicle_initialization))
  (attribute vehicle_initialization))
```

This schema being an "instance-of attribute-restriction", the rule "generate_class-restriction-rule" fires and creates out of it, the following schema:

```
(defschema vehicle_init-restriction-gen-rule
  (instance-of  generated-rule object rule)
  (compile compile_rule_method)
  (TEXT "(defrule VEHICLE_INIT-RESTRICTION-VIOLATION-RULE \n\t(logical (schema ?s
\n\t\t(instance-of object)\n\t\t(VEHICLE_INIT ?val)))=>\n\t(if (not-instance-of-allowable-class ?val
VEHICLE_INIT-RESTRICTION) then\n\t(assert (schema ?s \n\t\t(violates_constraint
VEHICLE_INIT-RESTRICTION-VIOLATION)))\n\t\t(add-warning-item ?s VEHICLE_INIT-
RESTRICTION-VIOLATION  VEHICLE_INIT-RESTRICTION VEHICLE_INIT ?val ALLOWABLE-
CLASSES)))")
  (VIOLATES_CONSTRAINT))
```

4 As this schema is an "instance-of rule" which "compile" attribute has a value of "compile-rule-method", it is then compiled using the same procedure specified above. Its text attribute is compiled into a defrule named attribute-restriction-violation-rule:

```
(defrule VEHICLE_INIT-RESTRICTION-VIOLATION-RULE
  (logical
     (schema ?s
        (instance-of object)
          (VEHICLE_INIT ?val)))
=>
   (IF
      (NOT-INSTANCE-OF-ALLOWABLE-CLASS ?VAL VEHICLE_INIT-RESTRICTION) THEN
     (ASSERT
       (SCHEMA ?S
          (VIOLATES_CONSTRAINT VEHICLE_INIT-RESTRICTION-VIOLATION)))
     (ADD-WARNING-ITEM ?S VEHICLE_INIT-RESTRICTION-VIOLATION VEHICLE_INIT-
RESTRICTION VEHICLE_INIT ?VAL ALLOWABLE-CLASSES)))
```

5 Finally this rule fires whenever violation occurs on the restricted attribute, thus modifying the value of the constraint-violation attribute of the corresponding object.

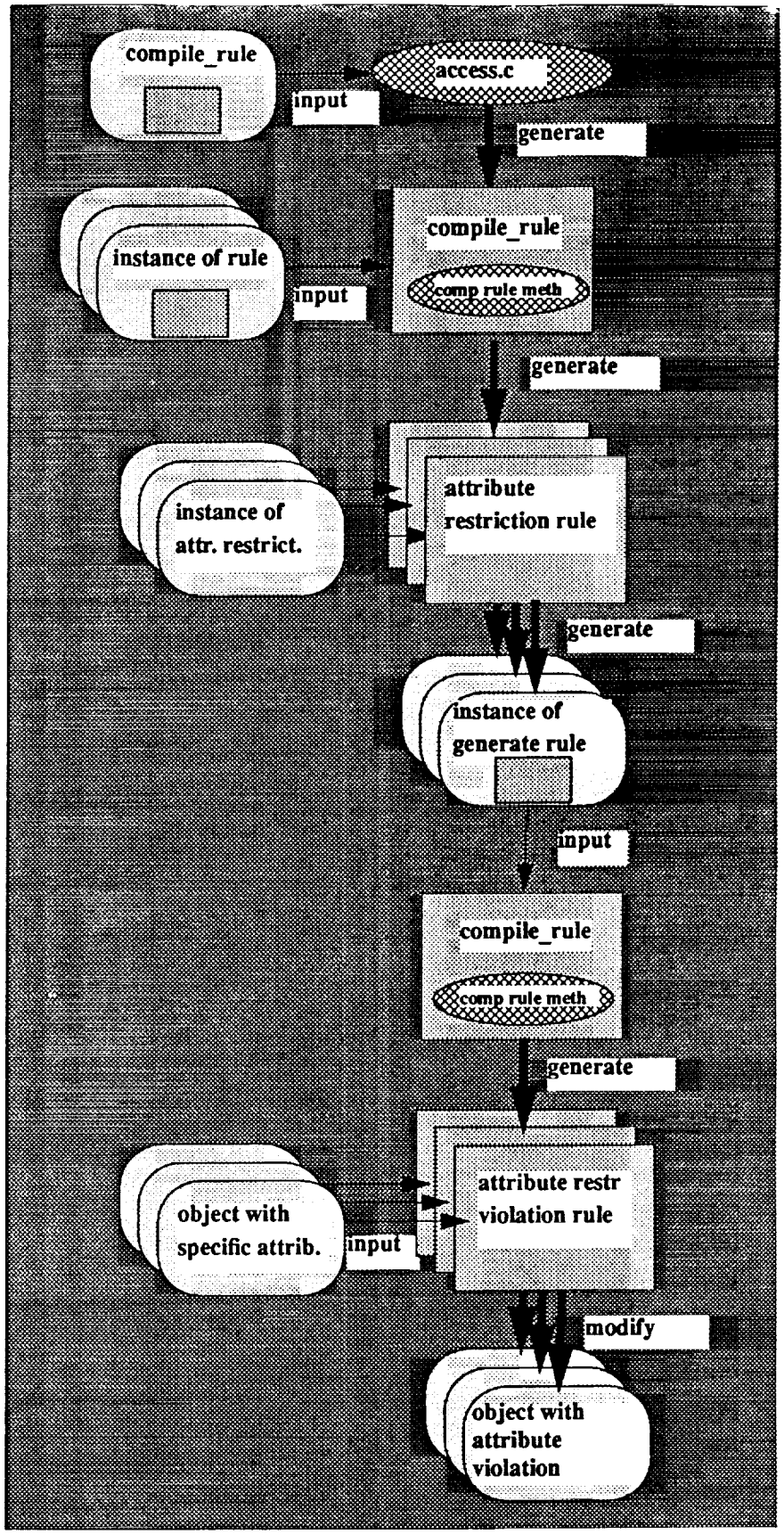
U.S. Gov't

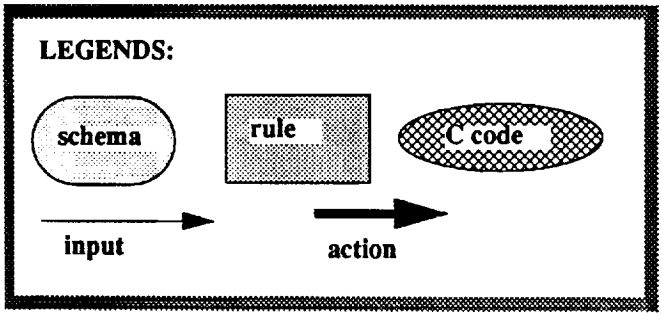
ART-IM SCHEMA TO RULE GENERATION FLOW FOR:

- CLASS_RESTRICTION_RULE
- MANDATORY_CLASS-RESTRICTION_RULE
- ENUMERATED_SET_RULE
- MANDATORY_ENUMERATED_SET_RULE
- TYPE_RESTRICTION_RULE
- MANDATORY_TYPE_RESTRICTION_RULE

The following actions occur for each of the rules listed above:

1) at start up, main routine(access.c) builds the compile_rule rule from the compile_rule schema.
2) compile_rule rule generates restriction rules for all instances of rule exist in the knowledge base. A compile_rule_method C routine is called to generate a new rule.
3) each attribute restriction rules query for attribute restriction instances and builds a generate rule schema for the instance.
4) compile_rule rule generates attribute restriction violation rule from each of the generate rule schemas.
5) step 2 is repeated
6) each attribute restriction violation rules query for their designated attributes from instances of object containing that attribute; every time the designated attribute changes, the specific attribute restriction violation rule checks for attribute restriction violation; the violation is recorded in the violates_constraint slot of the object and the warning is posted.





Bibliography

M. E. Izygon, C. L. Pitman, "INTUIT - An INTelligent User Interface Development Tool", *Proceedings of the Ninth TAE User Conference,* New-Carollton, Md, November 5-7, 1991.

M. E. Izygon, C. L. Pitman, "Applying Expert System Technology to Existing Simulation Programs", *International Conference on Simulation Technology, SIMTEC 91*, Orlando, Florida, October 21-23, 1991.

M. E. Izygon, C. L. Pitman, "A Knowledge Based Front-End for a Complex Spaceflight Simulation Program", *Proceedings of the 1991 Summer Computer Simulation Conference*, Baltimore, Md, July 22-24, 1991

C. L. Pitman, M. E. Izygon, E. W. Ralston, E. M. FridgeIII, and B. P. Allen, "Intelligent Interfaces for Complex Software", *Proceedings of the 4th International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems,* Hawaï June 2-5 1991.