Satellite Simulations Utilizing CLIPS

by

Barbara Pauls
Mark Sherman

Rockwell International
Satellite & Space Electronics Division
P.O. Box 3644
Seal Beach, CA 90740-7644
MS: SJ62

Simulations provide necessary testbeds for system designs. Currently we are developing software whose main requirement is to produce CLIPS executable simulation code of a user prespecified system. This process minimizes the amount of engineering effort required to specify a system thereby reducing cost and providing the capability to quickly revise system definitions. Modeling satellite systems is the primary objective toward which testing has, and is, being conducted using satellite specifications. This paper describes the modeling software being developed, its formatted input and the CLIPS system simulation it produces.

Introduction

The main purpose behind our current satellite simulation efforts is to provide a testbed for autonomy research. The method currently being developed is to produce realistic and dynamic behavioral models reflecting current-state satellite systems. Future uses of the simulation method being developed may include the testing of more advanced and fault tolerant system designs.

The ability to easily add, delete, change and replace satellite subsystem definitions is required to support current research. Unfortunately, CLIPS, and expert system languages in general, are not common knowledge to most satellite engineers. To ensure efficiency, the approach used allows the specifications to be written in a 'higher-level' language. Such a modeling language has been defined and is referred to as Satellite Modeling Language (SML). The SML allows the user to specify the satellite system at any level desired. The satellite model can be defined at the system level, subsystem level or lower. Environmental affects on the satellite can also be defined using SML.

To convert SML code to CLIPS executable simulation code, a language translator was created. Consistent format of outputted code is automatically provided by the translator. The language converter can also implement necessary error checking. Currently the amount and type of error checking done by the SML translator is at a minimum. Future translator versions will include increased error checking capabilities of input modeling code. The language translator itself was written in CLIPS code. Being basically a sequential process difficulties arose forcing a language compiler to perform as an event driven process. However the experience of writing the translator in CLIPS provided understanding of CLIPS requirements needed to output simulation code.

By implementing the definition process in this manner, as shown in Figure 1, a basic structure evolved in each simulation model. This basic structure provides a certain degree of quality assurance, yet does not restrict the way in which a user defines a system. The specifications can be broken down into as many levels and/or modules as the engineer desires.

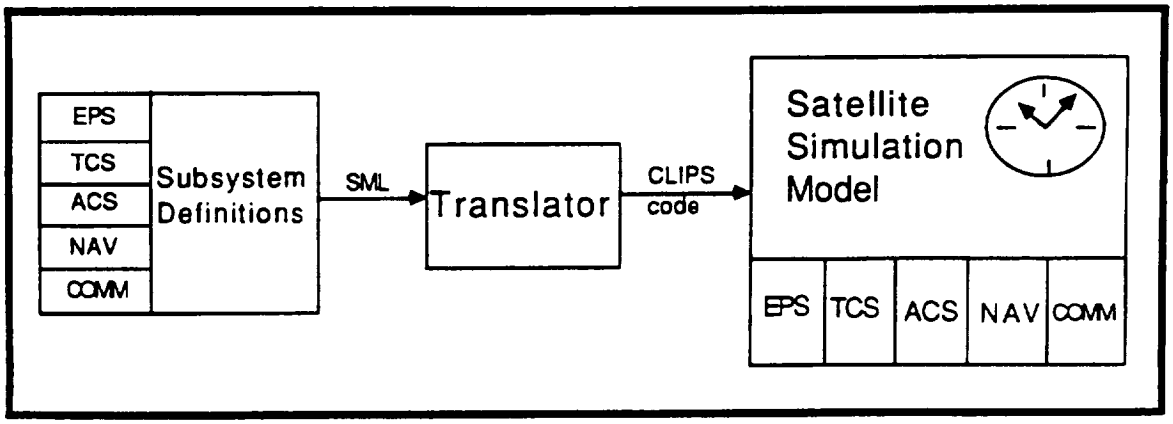

Figure 1. Simulation Definition Process

Satellite Modeling Language

SML consists of three main structures; templates, objects and rules. A template contains a generic set of attributes. The attributes are represented by simulation variables which describe the object. An object is created from a defined template and more than one object can be created from the same template. Objects can be specified at the time the template is defined or created separately. The SML rules define the simulation laws which all objects function under.

Object Definitions

Template specification contains a template name, optional object name(s), a list of attributes and their corresponding values. The SML 'define' command specifies a template and can create related objects. The syntax for the 'define' command is as follows, where optional fields are surrounded by square brackets []:

```
define template [object1 object2 ... objectN]
     ( attribute1 = value1;
       attribute2 = value2;
                :
       attributeN = valueN; )
```

Objects to be created with equivalent attributes are listed after the template name or can be specified by the SML 'create' command after the template has been defined. The syntax used to 'create' an object follows:

```
create template object1 [object2 ... objectN]
     ( [attribute1 = value1;
        attribute2 = value2;
                :
        attributeN = valueN;] )
```

Attribute values assigned when the template was defined may be changed for new objects. However, the 'create' command can not refer to any new attributes not defined in the corresponding template. If a template attribute is not listed in the 'create' command it retains the original value given in the template definition. The attribute value can be of any data type.

A one-dimensional array of attributes may also be specified. The array index is defined with square brackets and for every array element there must be a corresponding value, separated by commas. Examples of a template definition and object creations are given in Figure 2.

```
define eps_template
   (nominal_power          = 0;
    batteries_enabled      = 0;
    batteries [1, 2, 3]    = off, off, off;
    main_bus_voltage       = 0;
    power_op_command       = off;
    enable_batteries_command = off;
    battery_on_command [1, 2, 3]  = off, off, off;
    battery_off_command [1, 2, 3] = off, off, off;)


create eps_template EPS
   (nominal_power = 18;
    enable_batteries_command = on;)
```

Figure 2. SML Template and Object Examples

Rule Specifications

SML rules define and constrain simulation model behavior. Each rule is assigned a rule name in the 'behave' field and has a condition and an action section. The condition section of a rule is broken into five fields; 'priority', 'from', 'to', 'condition_start' and 'condition_end'. All five fields are optional. The action section of a rule must exist. Once the condition is met the action field is executed. The syntax for a rule is as follows:

```
 behave      rule-name
[priority    (priority-level)]
[from        (start-time)]
[to          (end-time)]
[condition_start  ( condition1
                    condition2
                        :
                    conditionN )]
[condition_end    ( condition1
                    condition2
                        :
                    conditionN )]
action      ( action1;
              action2;
                 :
              actionN; )
```

The 'behave' field identifies the name of the rule and is required. The 'priority' assigns a priority value which is applied towards the order of rule execution and is restricted by CLIPS salience values to range between 0 and 10,000. Both the 'from' and 'to' fields are time oriented and have simulation default values which are currently provided by the interfacing process that uses the simulation as a testbed. Future versions may provide the capability to allow the user to specify default simulation times. When a time is specified in the 'from' field the condition is true if the current simulation time is greater than or equal to the specified start time. When a time is specified in the 'to' field the condition is true if the current simulation time is less than or equal to the specified end time.

When a 'condition_start' field exists and all conditions are met the rules action is fired. When a 'condition_end' field exists and all the corresponding conditions are true the rules action is not fired even if all time and start conditions are met.

The logical keywords 'and' and 'or' are used to connect rule conditions. The logical keyword 'not' is used to negate a condition. Legal SML comparison symbols are =, /=, <, <=, > and >=.

The 'action' field of an SML rule must exist and is executed when the corresponding conditions are met. Each action assigns or modifies values of object attributes. Currently SML input is constrained by the translators capabilities to use prefix notation in the action fields. The envisioned final translator version will allow infix notation in SML input. The rule examples given in Figure 3 depict future versions of SML input. Legal SML arithmetic operators are +, -, *, / and **. Currently only CLIPS functions are available in the SML input. User defined functions can be added to CLIPS and then used in SML input.

Comments may be inserted throughout SML code. Code between an exclamation character, !, and an end-of-line character is interpreted as user comments.

```
behave  EPS_NOMINAL_POWER
from  (10)
to   (950)
condition_start  (eps.power_op_command = on)
action  (eps.nominal_power = 18;
         eps.power_op_command = off;)


behave  RECORDER_1_COMMANDED_ON
to   (400)
condition_start  (comm.recorder_on_command.1 = on)
action  (comm.recorder_status.1 = on;)


behave DECREASE_AREA_A_TEMP   !environmental affect
condition_end  (not (acs.gyro_heater = on))
action  (tcs.area_a_temp = tcs.area_a_temp - .3;)
```

Figure 3. SML Rule Examples

Translator Description

The translator takes input files containing SML code and generates output files containing CLIPS code and an integrator symbol table. The translator requests names from the user for the input, output and integrator symbol files. Currently the translator converts three types of SML commands into CLIPS code; behave, create and define.

The input file can contain one or more SML commands. Any combination or order of SML commands is allowed. The output file has CLIPS code translated from an input file containing the SML commands. For each SML behave name there will be a CLIPS rule with the same name. An example of an SML behave command translated to a CLIPS rule is shown in Figure 4. The integrator symbol file contains a list of SML behave names, a list of variables that have been defined, and a list of variables not defined. The list of variables not defined may be defined in another input file that is yet to be translated. It is the responsibility of the simulation integrator program to report any undefined variables.

The translator was written in CLIPS to better understand the requirements of translation into CLIPS code. The translator is more of a sequential process than an event driven process. Many challenges were presented when a sequential process was coded in an event driven environment. Sequential coding was accomplished by using control flags. The translator was written to take advantage of event driven processes as much as possible.

The CLIPS translator code is stored in eight different files. The behave, create, and define files parse the SML commands and build the related CLIPS code. The read, and write files deal with input and output files. The index and field files parse a line from the SML file. The main file of the translator obtains user inputs, starts the translator and terminates the translator.

The translator relies on CLIPS being case sensitive. By converting the SML code into upper case and using lower case for the translator variables, duplicate fact names are reduced. The only exception to this rule is when a CLIPS function is used by an SML command thus requiring conversion to lower case.

SML

```
behave tcs_nominal_power_on
priority (  2)
from     (  0)
to       (250)
condition_start (tcs.power_op_command = on)
condition_end   (tcs.power = off)
action ((tcs.nominal_power = 5);
        (tcs.power_op_command = off);)
```

CLIPS

```
(deffacts TCS_NOMINAL_POWER_ON-time
   (TCS_NOMINAL_POWER_ON-from-time 0)
   (TCS_NOMINAL_POWER_ON-to-time 250))

(defrule TCS_NOMINAL_POWER_ON
   (declare (salience 2))
   ?a_toc <- toc TCS_NOMINAL_POWER_ON)
   (time ?time)
   (TCS_NOMINAL_POWER_ON-from-time ?from-time)
   (TCS_NOMINAL_POWER_ON-to-time   ?to-time)
   (TCS.POWER ?TCS.POWER)
   ?a_TCS.POWER_OP_COMMAND <- (variable-data
      TCS.POWER_OP_COMMAND ?TCS.POWER_OP_COMMAND)
   ?a_TCS.NOMINAL_POWER <- (variable-data
      TCS.NOMINAL_POWER ?TCS.NOMINAL_POWER)
=>
   (retract ?a_toc)
   (if (and
       (>= ?from-time ?time)
       (<= ?to-time   ?time)
       ( eq ?TCS.POWER_OP_COMMAND ON )
       (not
       ( eq ?TCS.POWER OFF )
       )) then
      (retract ?a_TCS.NOMINAL_POWER)
      (retract ?a_TCS.POWER_OP_COMMAND)
      (assert (variable-data TCS.NOMINAL_POWER 5))
      (assert (variable-data TCS.POWER_OP_COMMAND OFF)
   )
)
```

Figure 4. Sample SML Behave Translation

Translation of SML Define and Create Commands

Figure 5 shows the translation of the SML define and create commands into CLIPS code. Each part of the define and create command is broken up into individual pieces (i.e. template, object, attributes) during the reading of the command. Each piece is tagged with the template name for latter use in generating CLIPS code. The generation of CLIPS code from the define command is delayed until after all the create command CLIPS code has been generated. This is because the create and define command can come in any order and the create translation needs the pieces of the define command. After all the create commands have generated their CLIPS code, the define command can then generate CLIPS code. Once the define command has generated the CLIPS code all the pieces related to the define command can be deleted.

SML

```
define tcs_template
   (power_op_command     = on;
    power                = off;
    nominal_power        = 5;)

create tcs_template tcs
   (power_op_command = on;)
```

CLIPS

```
(deffacts TCS_TEMPLATE
   (variable-data TCS.POWER_OP_COMMAND OFF)
   (variable-data TCS.POWER_OFF)
   (variable-data TCS.NOMINAL_POWER 5))

(deffacts TCS
   (variable-data TCS.POWER_OP_COMMAND ON)
   (variable-data TCS.POWER_OFF)
   (variable-data TCS.NOMINAL_POWER 5))
```

Figure 5. Sample SML Define and Create Translation

Translation of SML Behave Command

Figure 4 shows the translation of the SML behave command. For every SML behave command the translator produces a maximum of one CLIPS deffacts statement and one CLIPS simulation rule. If any time conditions are specified in the SML rule 'from' and 'to' fields, a deffacts statement is created which asserts minimum and/or maximum time values specifically corresponding to the simulation rule. These are then tested in the CLIPS rule against the simulation time.

In order to assure that all CLIPS rules are executed once per simulation second, the left hand side (LHS) conditions of the CLIPS rule must always be true. Therefore only necessary facts are referenced on the LHS using binding variables whenever possible. The SML specified conditions are then tested on the right hand side (RHS) of the CLIPS rule using an 'if...then' structure.

Each SML behave command consists of six specific parts. The 'priority' part translates to a declaration of rule salience. The 'from' and 'to' parts define a check on the simulation time facts done on the RHS of the CLIPS rule. The 'condition_start' and 'condition_end' part also define the 'if...then' check done on the RHS of the rule. The SML 'action' part translates to retract and assert statements in CLIPS code.

Simulation Integration

The integrator program accepts input from the integrator symbol table. The integrator symbol table is created by the translator program. The integrator symbol table, see Figure 6, contains a list of all SML rule names, a list of SML variable names, and a list of undefined SML variable names. The list of SML defined and undefined variable names have been provided for future enhancements. The output of the integrator program is the dynamic CLIPS code, see Figure 7. The dynamic CLIPS code file contains any simulation control code needed to run the simulation model.

```
                                                              INPUT
*** NEW ***
TCS_NOMINAL_POWER_ON
     :
*** NEW ***
     :
NAV_PAYLOAD_ELECTRONICS_SDTBY


                                                              OUTPUT
TCS_NOMINAL_POWER_ON
     :
NAV_PAYLOAD_ELECTRONICS_SDTBY
```

Figure 6. Sample Integration Symbol Table

```
(defrule tic
   (not (tic-done))
   ?a_tic <- (tic)
   ?a_time <- (time ?time)
   (time-max ?time-max)
=>
   (retract ?a_tic)
   (bind ?num (+ ?time 1))
   (if (<= ?num ?time-max) then
      (retract ?a_time)
      (assert (time ?num))
      (assert (toc TCS_NOMINAL_POWER_ON))
           :
      (assert (toc NAV_PAYLOAD_ELECTRONICS_SDTBY))
   else
      (assert (tic-done))
      (assert (get-tic))
   )
)
```

Figure 7. Dynamic CLIPS code

Simulation Model

All simulation code output from the SML translator is CLIPS executable. For every SML file input to the translator one corresponding CLIPS file is output. To execute the simulation all the translator outputted files and two other input files, one static and one dynamic file, are loaded into the CLIPS environment. The simulation static file contains the simulation time control rules and any other CLIPS rules needed that are not subsystem dependent. This additional code provided in the static file is user specified simulation requirements not supplied by the SML input. The dynamic file contains time rules which control simulation rule execution. This file is generated by the integrator program previously described. The remaining files contain SML translated commands. In our simulation model each subsystem was described in one SML input file and after translation each subsystems simulation code was contained in a unique output file.

As previous examples have shown, satellite simulations have been defined on the subsystem level using command and measurement attributes to describe each subsystem. Once these object attributes have been defined and created a time clock is introduced by the translator produced static file to control the simulation processing. The implementation of time restricts rule execution by allowing each CLIPS rule to fire only once per simulation second. Start and end times of the simulation clock are currently defined by a higher level process interfacing with the satellite model. The simulation can be defined as a stand-alone process if the start and end times are hard coded in the static file.

In order to simulate time the translator produces a predefined set of time rules which are based on a 'tic toc' process. A 'tic' fact serves as a timer interrupt and in our current simulations is produced by the higher level process interfacing with the satellite model. This interrupt could be produced by the CLIPS simulation model itself if it were to execute stand-alone. The CLIPS simulation always processes the timer interrupt using one rule. This rule retracts the 'tic' fact when it exists, validates that the current time is less than the maximum simulation time, increments time and asserts a 'toc' fact for every translated SML rule. Each 'toc' fact is retracted when its corresponding simulation rule is executed. When all 'toc'

facts have been retracted the simulation model is hung until another timer interrupt, a 'tic' fact, is asserted.

Currently no user interface exists to run a stand-alone CLIPS simulation model. Any information to be displayed during runtime must be added to the CLIPS simulation code and no operator interrupt capability has been provided. However our current uses do not require a stand-alone interface.

Summary

The method which evolved from the basic satellite simulation approach provides the tools needed to minimize development effort and allow the subsystem engineers to quickly revise system definitions. The input and output requirements for any simulation are independent and in our approach we left such requirements to be implemented by the simulation coordinator. The simulation interface can be coded in CLIPS and put into the static file so as not to complicate subsystem engineer development. When a function is needed which is not provided by either SML or CLIPS it can be easily defined in CLIPS and then referenced in the SML descriptions.

Utilizing the CLIPS expert system language as the simulation code was quite advantageous. Coding the SML translator in CLIPS was a challenge, however, this approach did provide insight to CLIPS capabilities and functionality. For the satellite modeling effort CLIPS provided a more than suitable event driven simulation environment. Other advantages to utilizing CLIPS included low cost, high portability and easy integration with external systems. We believe the approach described allows the definition of a wide range of satellite architectures, satellite behaviors and environmental influences with minimal effort.