

APGEN scheduling: 15 years of Experience in Planning Automation

Pierre F. Maldague,¹ Steve Wissler,² Matthew Lenda³ and Daniel Finnerty⁴
Jet Propulsion Laboratory, California Institute of Technology, Pasadena, California, 91109

In this paper, we discuss the scheduling capability of APGEN (Activity Plan Generator), a multi-mission planning application that is part of the NASA AMMOS (Advanced Multi-Mission Operations System), and how APGEN scheduling evolved over its applications to specific Space Missions. Our analysis identifies two major reasons for the successful application of APGEN scheduling to real problems: an expressive DSL (Domain-Specific Language) for formulating scheduling algorithms, and a well-defined process for enlisting the help of auxiliary modeling tools in providing high-fidelity, system-level simulations of the combined spacecraft and ground support system.

Nomenclature

<i>ACS</i>	=	Attitude Control System
<i>AI</i>	=	Artificial Intelligence
<i>AL</i>	=	APGEN Language
<i>AMMOS</i>	=	Advanced Multi-Mission Operations System
<i>APGEN</i>	=	Activity Plan GENERator
<i>DI</i>	=	Deep Impact
<i>DPT</i>	=	Data Priority Table
<i>DSL</i>	=	Domain-Specific Language
<i>DSN</i>	=	Deep-Space Network
<i>FPT</i>	=	Frame Priority Table
<i>INSIGHT</i>	=	Interior Exploration using Seismic Investigations, Geodesy and Heat Transport
<i>MER</i>	=	Mars Exploration Rover
<i>MRO</i>	=	Mars Reconnaissance Orbiter
<i>MSL</i>	=	Mars Science Laboratory
<i>PEF</i>	=	Predicted Event File
<i>PEL</i>	=	Power Equipment List
<i>S/C</i>	=	Spacecraft
<i>SASF</i>	=	Spacecraft Activity Sequence File
<i>SEQGEN</i>	=	SEquence GENERator
<i>TOL</i>	=	Time-Ordered Listing

I. Introduction

THIS paper is mainly concerned with the application of APGEN (Activity Plan GENERator), a multi-mission planning application that is part of the NASA AMMOS (Advanced Multi-Mission Operations System), to a variety of scheduling problems in the context of recent NASA Space Missions. According to its initial design, APGEN was to be used interactively by mission planners. The simulation capabilities of APGEN were intended to support a planning paradigm in which the user comes up with a high-level activity plan, the software works out the consequences in terms of critical resource usage, and the user decides whether or not to iterate or refine the search for a plan that satisfies his or her criteria. The simulation capabilities of APGEN grew over time, to the point where

¹ Senior Staff, Mission Planning and Execution Section

² Chief Engineer, Mission Systems and Operations Division

³ Mission Operations Engineer, Mission Planning and Execution Section

⁴ Multi-Mission Planning and Sequencing, Group Supervisor, Mission Planning and Execution Section

APGEN can accurately predict the impact of any S/C (spacecraft) sequence on key resources and states such as data volume, battery charge and S/C attitude.¹

One of the requirements on APGEN was that it should present the user with automatically generated activities that would satisfy user-specified criteria. Although such a capability was not included in the initial design, two capabilities were subsequently added to APGEN:

- a two-pass scheduling algorithm was implemented and delivered as part of the APGEN software
- in collaboration with the MER (Mars Exploration Rover) mission, the Europa search-based planner from ARC (NASA Ames Research Center) was merged with APGEN into a mixed-initiative planning tool.

The collaboration with MER was discussed elsewhere. Here, we concentrate on the two-pass scheduling capability of APGEN.

The basic idea behind the two-pass scheduling algorithm used by APGEN is simple. In a first pass, APGEN predicts the effect of the current activity plan on S/C (spacecraft) states and ground resources. In a second pass, APGEN examines the possibility of adding one or more activity to the plan, based on time intervals during which a certain "scheduling condition" is true. If the condition is true for a given length of time, APGEN executes the activity generation part of the scheduling algorithm, which adds new activities to the plan. If the condition is false, APGEN keeps looking for other windows, always proceeding forward in time.

The scheduling condition must be provided by the APGEN user prior to running the algorithm. As an example, such a condition might yield all intervals of time during which at least one DSN station is available for telecommunications.

Note that availability of a DSN station implies not only that the DSN has made it available to the mission for a given period of time, but also that the S/C is visible from Earth. Thus, evaluating the condition requires external information from the DSN as well as computations of a geometric nature.

In making decisions, the activity generation algorithm of APGEN only has limited information, namely the resource predictions that were made during the first pass and the predictions that were made for the second pass up until the time at which a new activity could be added to the plan. In contrast with search-based planners, APGEN does not allow the algorithm designer to iterate the search for optimal placement of a new activity in the plan. As a result, APGEN's scheduling algorithm does not have the generality found in other, search-based planners.

The lack of generality of the APGEN scheduling algorithm is largely compensated by its scalability to complex plans in which many resources need to be evaluated in a high-fidelity simulation. In our paper, we review the successful application of APGEN scheduling to six NASA missions: EPOXI, Europa, INSIGHT, Juno, MRO, and MSL. In each case, we review the operational challenges that led to the need for automation, and we show how these challenges were met by the APGEN scheduling algorithm. In our conclusion, we will explain how our operations experience has allowed us to turn APGEN into the generic, multi-mission scheduling tool it is today, and we will outline our plans for providing continued support for space missions in early development as well as those that have been in operations for several years.

II. Resource modeling in early versions of APGEN and SEQGEN

The APGEN DSL has undergone continuous changes over the 15-plus years of APGEN's history, as APGEN developers sharpened their understanding of space mission needs over time. To avoid forking APGEN into multiple, competing versions, changes were introduced in a backward-compatible manner: the DSL always expanded and never shrank. As a result, the DSL in its current state is somewhat of a hybrid; recent additions have turned it into a reasonably modern-looking programming environment, while early features still present in the language make the DSL arcane and non-intuitive. After 15 years of evolution, the time has probably come to take stock of lessons learned and to undertake the development of a planning system that would be easier to learn and maintain than APGEN. Besides the complexity of its DSL, there are good systems-level reasons to upgrade APGEN: modern adaptations of APGEN demonstrate the need for integrating external S/C models into the planning model. Although APGEN supports such an integration mechanism, there is a need to make integration easier and more systematic.

In spite of its blemishes, the APGEN DSL has been remarkably adept at modeling complex systems in support of mission operations. In redesigning APGEN to better support future missions, one should take care not to throw the baby with the bath water. In particular, a drastic redesign of the APGEN DSL might inadvertently throw away arcane features that turned out to be essential in providing APGEN with the flexibility and scalability it enjoys today. To help future designers avoid this problem, a significant portion of our paper has been devoted to a description of these arcane features and to the role these features have played in actual APGEN adaptations in use today. In this Section, we summarize the early history of APGEN and comment on the DSL as designed by its original creators. In Section III we will discuss enhancements that were introduced into the APGEN DSL in order to

support the six space missions mentioned in the Introduction. In Section IV we will review the application of these capabilities to actual space missions.

A. Early history of APGEN

APGEN handles two basic types of objects: activities and resources. Activities are blocks of time devoted to a given purpose, such as a S/C maneuver or a science observation. Resources are numeric or discrete quantities that say something about the state of the S/C and how that state is evolving as a function of time. The types of activities that can be present in the plan, the resources that characterize the S/C state, and the interaction between the two are specified by the APGEN *adapter* using an APGEN-specific DSL (Domain-Specific Language) which for lack of a better name will refer to as the *APGEN DSL*. Users can then load the adapter's code (referred to as the *adaptation code*) into APGEN and create plans for the mission they are working on.

In the early phases of its development, APGEN was strongly influenced by its predecessor, a command sequence simulation and validation tool called SEQGEN. Both tools expect their users to provide S/C activities as inputs to the program, while the time histories of the resources were output by the program. The main difference between the two tools was that APGEN was a lightweight version of SEQGEN, making it easier to use by planning engineers while a mission was in its early phases. In particular, the APGEN DSL did not require that S/C commands be defined, while in SEQGEN the use of commands was essential in establishing a link between S/C activities and usage patterns for the various resources.

B. Expansion and modeling stages in early SEQGEN

As mentioned earlier, the early development of APGEN was strongly influenced by SEQGEN; in particular, the fundamental way in which APGEN computes the effect of S/C activities on resources was directly patterned on the way SEQGEN accomplished the same task. Because this still forms the foundation of the APGEN modeling algorithm, we discuss the early SEQGEN algorithm in some detail. For simplicity, we refer to the various components of a SEQGEN adaptation in informal terms; we refer the reader to the SEQGEN Users' Guide for more detailed information. We also use the present tense to indicate the way SEQGEN processes its input, even though we are referring to an ancient version of SEQGEN; modern sequence processing requires considerably more sophistication.

The basic ingredients of an early SEQGEN adaptation are the following:

- A model definition file, where *model* means a collection of *elements* identified each by a unique name and, for each element, a collection of one or more *element attributes* declared as a variable of a given type (Boolean, integer, string etc.) together with a description and an optional range within which the attribute is to be confined.
- A command definition file, which contains one entry per S/C command. Each command is identified by its name, which must be unique. Each command may have an arbitrary number of parameters, each one of which must have a type, an optional description and, if applicable, an allowed range. Finally, each command is followed by a (usually short) program written in the SEQGEN DSL called a *results section*. In that section, the adapter can use DSL programming statements to express the effect of a command on the various elements of the models, using knowledge about the command's execution time and the values of its parameters.
- An activity definition file, which contains any number of activity types. Each activity type is similar to a command in that it must have a unique name and may have an arbitrary number of parameters. However, an activity does not have a results section; instead, it contains an *expansion* section which indicates how a given activity decomposes or expands into a mix of lower-level activities and commands.

In order to simulate a sequence, SEQGEN has to be given two things: first, an adaptation in the form of the three files described above; and second, one or more sequence files describing the sequence to be simulated. A sequence is an ordered set of time-tagged objects arranged in order of increasing time; each object is either an activity or a command. Any activity in the sequence must be of a type that is included in the activity definition file, and any command in the sequence must match an entry in the command definition file. The number of parameters of the two types of objects in the sequence file must match the types listed in the definition files, and parameter values must be within the indicated range if applicable.

Once loaded with adaptation and sequence files as indicated above, SEQGEN uses a two-step algorithm to process sequences. The algorithm is described in Table 1 below.

Table 1. Stages used by early versions of SEQGEN in processing a command sequence.

<i>Stage</i>	<i>Description</i>	<i>Final State</i>
Expansion	Activities in the sequence are expanded into lower-level activities and commands, using the activity type definitions to determine activity and command timing and parameters. Any resulting activities are likewise expanded until all activities have been fully decomposed into commands.	All commands – those initially present in the sequence and those that result from activity expansion – are collected in a single time-ordered list, called the <i>event queue</i> .
Modeling	Commands in the event queue are scanned in time order, and the effect of each command on the model element attributes is determined by executing the code in its results section as specified in the command definition file.	All state changes occurring in each model element attribute are recorded in that attribute's history. All activities, commands and model element attribute changes are collected in time order in a file called the <i>PEF (Predicted Event File)</i> .

One important aspect of the expansion stage is that the resulting commands are placed in a single time-ordered list, the event queue, which is then scanned in time order during the modeling stage. The time ordering of the event queue is an essential aspect of SEQGEN and, by extension, of APGEN: the modeling code in the adaptation frequently contains logic which changes the values of S/C states based on their current values. Such logic would fall apart if commands were not processed in time order. There is of course a solid rationale for organizing commands in time order while running the simulation, since that is the order in which they are executed onboard the S/C.

C. Decomposition and Modeling Phases in early APGEN

The designers of APGEN wanted to provide a capability similar to SEQGEN but easier to use and better adapted to the needs of planning engineers. To this end, they made three decisions:

1. Allow users to represent the same plan at several levels of fidelity. This was done by introducing a flexible scheme for letting high-level activities decompose into lower-level activities. Decomposition is *exclusive* in the sense that only one level is reflected in the model. If the user chooses to display high-level activities, their impact on the model is evaluated in a coarse, high-level manner. If the user chooses instead to display lower-level activities, the impact on the model is evaluated at a higher-fidelity level.
2. Eliminate the overhead associated with the need to define S/C commands. They accomplished this by offering only a fixed set of *modeling commands*. Furthermore the effect of these commands was hard-coded so there was no need for codifying the effect of commands in the adaptation.
3. Make the adaptation language more intuitive. The hard-coded modeling commands were given intuitive names such as *use*, *set* and *reset*. Instead of SEQGEN's model element attributes, the APGEN model is made up of *resources*. Numeric resources are divided into *consumable* resources, which are depleted with each usage, and *non-consumable* resources, which are restored to their default values when they are no longer used. To represent discrete-valued quantities, adapters can use state resources, which featured an adjustable, finite set of allowable states.

As a result of these decisions, APGEN does not process an activity plan the same way SEQGEN processes a sequence. Decomposing activities into lower-level activities (and abstracting lower-level activities back into higher-level ones) is a purely interactive task. Once the user has settled on a representation level – abstract or detailed – he or she can ask APGEN to *remodel* the plan, i. e., to evaluate the effect of the activity plan (as currently displayed) on the resources that make up the model.

The remodeling process is similar to SEQGEN's processing of a sequence and is illustrated in Table 2 below.

Table 2. Stages used by early versions of APGEN in remodeling an activity plan

<i>Stage</i>	<i>Description</i>	<i>Final State</i>
Event generation	Activities in the activity list are scanned in time order. Any <i>use</i> and <i>set</i> commands present in the resource usage section of each activity are extracted and inserted as <i>usage events</i> into the event queue.	All usage events extracted from the definition of activities in the plan are collected in time order in a time-ordered list called the <i>event queue</i> .

<i>Stage</i>	<i>Description</i>	<i>Final State</i>
Event modeling	Usage events in the event queue are scanned in time order, and the effect of each event on the corresponding resource is calculated based on the event's timing and parameters.	All changes occurring in each resource are recorded in that resource's history. At the option of the user, all activities and resource changes are collected in a file called the <i>TOL (Time-Ordered Listing)</i> .

The Cassini cruise plan, which provided the first application of APGEN to a space mission, was modeled in 1996. It provides an example of resource and activity definitions in the APGEN DSL. The definition of the *Op(eration)Mode* resource is shown in Fig. 1 below. The resource definition includes the resource data type (string), the resource usage type (state), its parameter (*neededState*), the list of possible states (*LS*, ..., *OrbInstDeploy_Maint*), the default or profile state (*LS*), and the usage value, i. e., the function of the parameter (in this case the identity function) which the resource should take on when used.

Figure 1. Definition of the *OpMode* state resource in the Cassini cruise adaptation

```
resource OpMode : state string
begin
  parameters
    neededState : string default to "Cruise2";
  states
    "LS","Cruise1_Decon","Cruise2","ME_TCM","RCS_TCM","PCO",
    "ICO","MAGcals1","MAGcals2","MAGcals3","UVISMaint",
    "PerInstMaint","PerEngrMaint","ORS","ORSMaxTorque","DFPW",
    "INMS_FPW","RADAR_INMS","RSS2_RCS","RSS2_RWA","RSS3_RWA",
    "CatbedWarmup_OR_RWUnload","SOI","ProbeRelay",
    "OrbInstDeploy_Maint";
  profile
    "LS";
  usage
    neededState;
end resource OpMode
```

An example of an activity type definition is shown in Fig. 2 below; it illustrates the fact that the early version of the APGEN DSL does not feature any conventional programming constructs such as declarations and assignments. A form of conditional execution is provided through the *when* keyword followed by a Boolean expression. During the event generation process, the usage events are placed on an event queue along with any attached *when* clauses. When APGEN models the event as part of the second stage of the remodeling algorithm, the *when* clause is evaluated at the time of the event, and the usage statement is executed if the Boolean expression following *when* is true.

Table 3 below summarizes a few statistics about the Cassini cruise plan adaptation. These numbers should be kept in mind when compared with statistics for adaptations of APGEN discussed in later Sections of this paper.

Table 3. Statistics of the Cassini cruise plan adaptation of APGEN.

	<i>Globals</i>	<i>Functions</i>	<i>Resources</i>	<i>Activity Types</i>	<i>Constraints</i>
<i>Number of Items</i>	0	0	61	345	15
<i>Lines per Item</i>	N/A	N/A	15	21	15

Although it was not used in the Cassini cruise plan, we want to mention one more feature of the early APGEN DSL: abstract resources. Because activity decomposition in early APGEN was meant to represent alternative representations of the same plan at various levels of fidelity, there was no way to organize complex activities into a hierarchy of modular entities. To improve the scalability of the DSL, the notion of an abstract resource was introduced. An abstract resource is essentially a function call. It can be invoked by an activity (or another abstract resource) through a *use* statement, exactly like a consumable or non-consumable resource. Unlike a concrete resource, an abstract resource does not have a state variable associated with it; instead, it is allowed to invoke other (concrete or abstract) resources, much as a function can call other functions in a general programming language. By collecting recurring usage patterns into abstract resources, adapters can make their code more modular.

Figure 2. An activity type definition taken from the Cassini cruise adaptation

```

activity type RemoteSensPalletReplHtr2
begin
  attributes
    "Color" = "Orange Red";
    "Pattern" = 2;
    "Duration" = Duration;
    "Legend" = "Remote Sensing Pallet Repl. Heater 2 (T/C)";
  parameters
    Duration : local float default to 5400.0;
    PowerMode : local string default to "OFF";
  resource usage
    use RTGpower (21.32) when PowerMode == "ON";
    use RTGpower (0.0) when PowerMode == "OFF";
    use RTGpower (0.0) when PowerMode == "ENABLE";
end activity type RemoteSensPalletReplHtr2

```

The addition of abstract resources to the APGEN DSL restored the similarity between the APGEN remodeling process and SEQGEN sequence processing, which had been partially lost when APGEN designers introduced the notion of exclusive activity decomposition. Much as a SEQGEN adapter can use activity expansion to orchestrate how a complex activity expands into commands, an APGEN adapter can organize a complex resource usage pattern as a hierarchy of calls to suitable abstract resources. This is readily seen from the similarity between Table 4 below, which describes APGEN remodeling in the presence of abstract resources, and Table 1 showing the analogous SEQGEN process. For future reference, note that we refer to the modeling style in Table 4 as the *a priori* modeling style – *a priori* because the code inside the definition of abstract resources is executed in the event generation phase, before modeling commands in the event queue have had a chance to modify the history of any resources.

Table 4. Stages in remodeling an activity plan using the *a priori* modeling style of early APGEN

Stage	Description	Final State
Event generation	Activities in the activity list are scanned in time order. Any modeling commands (<i>use</i> , <i>set</i> and <i>reset</i>) found in the resource usage section of activity definitions are examined. Commands that invoke concrete resources are collected into a list. Any commands that invoke abstract resources are themselves scanned for further extraction of <i>use</i> and <i>set</i> commands, until all invocations of abstract resources have been processed.	Modeling commands extracted from the definition of activities in the plan and from the usage of abstract resources are encapsulated into <i>usage events</i> which are collected into a time-ordered list called the <i>event queue</i> .
Event modeling	Usage events in the event queue are scanned in time order, and the effect of each event on the corresponding resource is calculated based on the event's timing and parameters.	All changes occurring in each resource are recorded in that resource's history. At the option of the user, all activities and resource changes are collected in a file called the <i>TOL (Time-Ordered Listing)</i> .

D. A non trivial example: a simple science activity

Before moving on to more recent examples of APGEN adaptation, we want to make one last point regarding the early version of the APGEN DSL. In spite of the restrictions on it – no iteration, no local variables, limited availability of conditional statements – the early APGEN DSL is surprisingly expressive. As will be seen later in our paper, the current version of the APGEN DSL offers adapters a more complete set of programming constructs. Unfortunately, the flexibility that results from the new constructs easily obscures the simple (although nontrivial) infrastructure offered by early versions of the DSL. In hindsight, it might have been better to establish a clearer

distinction between the early and late APGEN modeling styles, and we hope that future developers of APGEN (or its successor) will take our remarks into account.

To illustrate our point, we consider a problem that frequently confronts APGEN adapters: the simple behavior provided by the APGEN *use* and *set* statements leads to abrupt changes in the value of a resource; can we change it so that the change in the resource becomes gradual? In the next few paragraphs, we will show that this problem can be solved using the limited capabilities of the early APGEN DSL.

The solution of our problem is to write the adaptation so that activities control the rate at which a resource changes, rather than the resource itself. To be specific, let us consider the problem of controlling the amount of data volume through the rate at which data is created. We can solve the problem by introducing four resources:

- *DataVolume*, which is the main object of our modeling effort; we will measure it in Megabits.
- *DataRate*, the rate at which data is created; we will measure it in Megabits per second.
- *LastUpdateTime*, the time at which we last updated *DataVolume* to reflect the current data creation rate.
- *AddData*, an abstract resource which acts as a controller for the other three resources and provides a simple API to activities that produce data.

We now show how these resources can be implemented in the early version of the APGEN DSL. Figure 3 below illustrates the consumable resources that implement *DataVolume* and *DataRate*.

Figure 3. The APGEN code for the *DataVolume* and *DataRate* resources

```
resource DataVolume: consumable float      resource DataRate: consumable float
begin                                     begin
attributes                                 attributes
  "Units" = "Megabits";                    "Units" = "Mbits/s";
  "Interpolation" = 1;                      parameters
parameters                                  x: float default to 0.0;
  x: float default to 0.0;                  profile
profile                                       0.0;
  0.0;                                       usage
usage                                       -x;
  -x;                                       end resource DataRate
end resource DataVolume
```

We note some of the features of the APGEN DSL which we have not mentioned before: the attributes section of the definition, in which general properties such as units can be defined. The interpolation attribute is used to control the graphical appearance of the resource; its default value is zero, in which case the resource plot shows a discontinuity at each usage event. We also note the APGEN idiom which consists in using an amount of the resource equal to minus the parameter supplied as an argument to the use statement. Without the minus sign, any use of the resource by some amount x reduces the value of the resource by x , in accordance with the basic paradigm of a critical resource being used up by the activities that consume it.

Figure 4 below shows the code for *LastTimeUpdated* and *AddData*.

Figure 4. The APGEN code for *LastTimeUpdated* and *AddData*

```
resource LastUpdateTime: consumable time  resource AddData: abstract
begin                                     begin
parameters                                  parameters
  t: time default to 2000-001T01:00:00;    rate: float default to 0.0;
profile                                       resource usage
  2014-093T20:00:00;                        use DataVolume(DataRate.currentval()
usage                                       * ((now - LastUpdateTime.currentval()) / 0:0:1))
  LastUpdateTime.currentval() - t;          when LastUpdateTime.currentval() < now;
end resource LastUpdateTime                use DataVolume(0.0) when LastUpdateTime.currentval() >= now;
                                           use DataRate(rate);
                                           use LastUpdateTime(now);
```

The code for *LastUpdateTime* exhibits an APGEN idiom: the use of the *currentval* function in a usage expression. For any concrete (non-abstract) resource R in the adaptation, the expression $R.currentval()$ can be used to refer to the current value of the resource. A usage expression of the form $R.currentval() - t$ implies that the value

of R after execution of the statement $use R(t)$ will be $C - (C - t) = t$, where C denotes the current value of R . Thus, the indicated form of the usage statement of resource R ensures that the usage statement $use R(t)$ results in R taking on the value t .

The resource usage section of the code for the abstract resource, *AddData*, contains the algorithm for updating *DataVolume*, *DataRate* and *LastUpdateTime*. Although the early APGEN DSL did not feature conditionals, it provided a *when* clause that could be appended to a usage statement; execution of the usage statement would then depend on whether the Boolean expression following *when* was true or false.

This leads us to a highly non-intuitive feature of the APGEN DSL: delayed execution of usage event logic. For the logic of a *when* clauses to work correctly, it is essential that the evaluation of the Boolean expression following it should take place while APGEN is scanning the event queue, and not while APGEN is extracting usage statements from resource usage programs inside activity type definitions. In the APGEN language of Table 4, we would say that evaluation of Boolean variables takes place at event modeling time, and not at event generation time. The distinction between event modeling time and event generation time is clear, given the context in which APGEN was first implemented. However, enhancements introduced later on into the APGEN DSL made the distinction harder to keep in mind; this remains a point of confusion for APGEN adapters. We will return to this point later on in our paper.

In order to exercise the adaptation we have just constructed, we need an activity type that uses the *AddData* resource. Such an activity type, called *science*, is shown in Figure 5 below.

Figure 5. An activity type definition called *science*

```

activity type science
begin
attributes
  "Duration" = Duration;
parameters
  Duration: duration default to 5:00;
  volume: float default to 0.5;
resource usage
  use AddData(volume/Duration) at start;
  use AddData(-volume/Duration) at finish;
end activity type science

```

To see how all these definitions work together, let us describe the modeling process APGEN goes through when given a plan containing a single *science* activity. The sequence of events is shown in Table 5 below.

Note that the events created in step 2 of Table 5 correspond to the four usage statements in the definition of the *AddData* resource, shown in Fig. 4. Because these statements do not contain an explicit time stamp (i. e., they do not end with a temporal expression of the form *at T* where T is a time expression), their time stamp defaults to the time at which the *AddData* resource is used, which is the start time of the *science* activity.

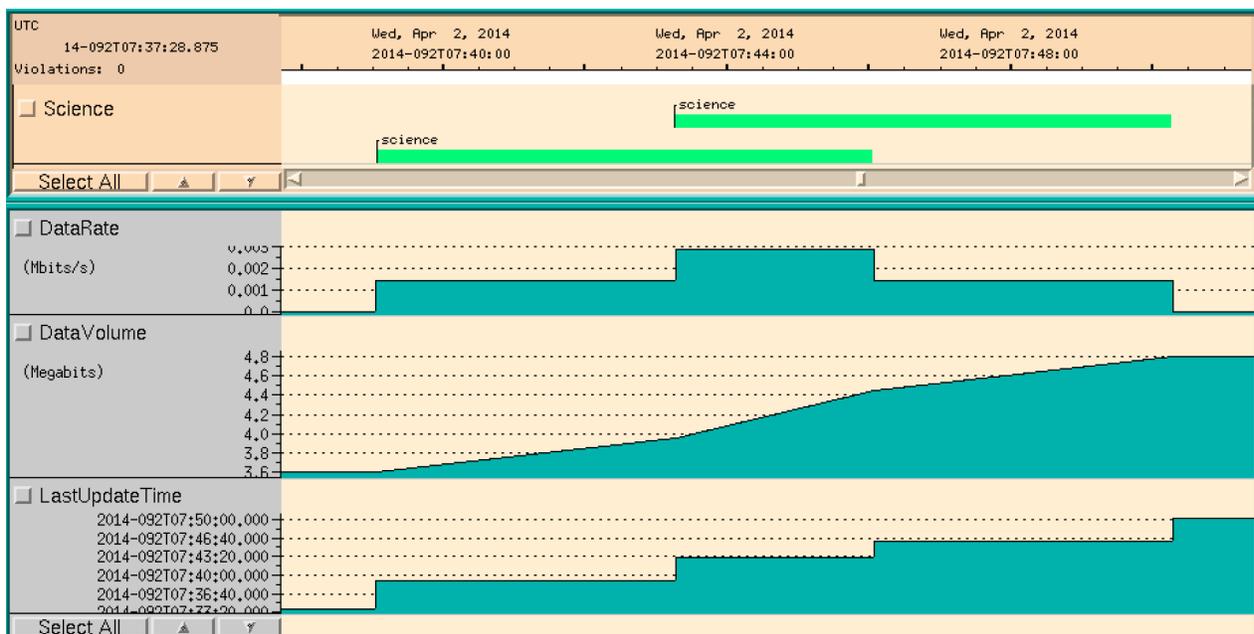
This leads us to discuss a frequent occurrence in APGEN modeling: the event queue contains several events occurring at the same time. To deal with this eventuality, the event queue is endowed with a secondary key besides the event creation time which provides the primary key. The secondary key is designed so that the resulting order is simply the order of insertion into the queue. The resulting event ordering ensures that as new events are added to the queue, the event queue iterator visits the new events once and never revisits events already processed.

Table 5. Sequence of steps taken by APGEN when modeling a science activity using *a priori* modeling

Step	Description	Final State
1	APGEN scans the time-ordered list of all activity instances – there is only one in this case – and extracts the modeling commands it finds in that activity’s resource usage sections (see Fig. 5 above).	Two modeling commands are found, both involving abstract resource <i>AddData</i> ; the first occurs at the start of the activity, denoted by S , the second at the end of the activity, denoted by E .
2	The first modeling command is expanded by consulting the resource usage section of <i>AddData</i> (see the code in Fig. 4 above). This results in four new events that are inserted into the event queue at time S .	The event queue now contains four usage events with time tag S . These commands occur in a well-defined order thanks to the secondary key of the event queue, as discussed above.
3	The second modeling command involving <i>AddData</i> is also expanded, which results in four new events that are inserted into the event queue at time E .	The event queue now contains four additional usage events with time tag E .

<i>Note: this concludes the event generation stage of the remodeling process.</i>		
4	APGEN scans the event queue in time order using an iterator. The event iterator runs into the first event at time S . It sets the value of global time variable <i>now</i> to S .	The event queue iterator points at the event at time S , which contains a conditional <i>when</i> clause with argument <i>LastUpdateTime.currentval() < now</i> .
5	The Boolean argument is evaluated. Because the current value of <i>LastUpdateTime</i> is greater than <i>now</i> (which was set to the time tag S of the current event), the Boolean evaluates to false.	Since the Boolean clause in the current event was found to be false, the event is not executed and the state of the resource model is the same as before.
6	The event iterator iterates and finds the second of the four statements in the resource usage section of <i>AddData</i> , again notices the presence of a conditional, and evaluates its Boolean argument; this time the argument evaluates to true, and the usage statement saying <i>use DataVolume(0.0)</i> is executed.	A new node recording the usage of 0 Megabits is inserted in the history of the <i>DataVolume</i> resource with time tag S . This node is necessary and marks the start of a change in slope; without it, the slope would depend on the time tag of the first node in <i>DataVolume</i> 's history.
7	The event iterator iterates again, finds the event that says <i>use DataRate(rate)</i> and executes it.	The value of the <i>DataRate</i> resource is changed by the amount <i>rate</i> at time S .
8	The event iterator iterates once more, finds the event that says <i>use LastUpdateTime(now)</i> and executes it.	The value of the <i>LastUpdateTime</i> resource is changed and is set equal to S .
9	The iterator iterates and finds the first of the four events at time E , notices the presence of a conditional, and evaluates its Boolean argument; the argument evaluates to true, and the usage statement saying <i>use DataVolume(...)</i> is executed.	A new node recording the usage of an appropriate number of Megabits is inserted at time E in the history of the <i>DataVolume</i> resource.
10	The iterator finds the second of the events at E , notices the presence of a conditional, finds the condition to be false and the usage statement is not executed.	The state of the resource model is the same as previously, since the usage statement for the current event was not executed.
11	The iterator iterates, finds the event that says <i>use DataRate(rate)</i> and executes it.	The value of the <i>DataRate</i> resource is changed by the amount <i>rate</i> at time E .
12	The iterator iterates, finds the event that says <i>use LastUpdateTime(now)</i> and executes it.	The value of the <i>LastUpdateTime</i> resource is changed and is set equal to E .
<i>Note: this concludes the event-modeling phase of the remodeling process.</i>		

Figure 6. Data Volume modeling as implemented above for the case of two overlapping science activities



While we have discussed the simple case of a single science activity, it should be obvious that the logic used in the above example applies equally well to an arbitrary number of activities, overlapping or not. Figure 6 above illustrates the above adaptation handling two science activities as displayed by a recent version of APGEN.

III. Evolution of the APGEN Adaptation Language

After the initial application of APGEN to the Cruise phase of the Cassini mission, it became clear that if APGEN was to be used as a general multi-mission planning tool it would be necessary to enhance the APGEN DSL, in particular in the resource usage section where the effect of an activity on resources is specified in detail. Early APGEN offered no provision for things like declaring local variables and performing arithmetic on floating-point and integer variables.

This changed around the year 1997 when APGEN was enhanced in order to provide support to upcoming Mars missions such as Mars '98. A number of new features were introduced at that time; we list them in Table 6 below.

Table 6. New features in the APGEN DSL

<i>Item</i>	<i>Description</i>
1	An <i>array</i> data type, which was added to the list of APGEN data types: Boolean, string, integer, float, time and duration. Strictly speaking, an APGEN array is not an array in the conventional programming sense but rather a generic container which can be turned into a linked list or a string-based map, depending on how it is used in the code. APGEN arrays allow adapters to introduce data structures of arbitrary complexity in the APGEN code with essentially no overhead.
2	An <i>instance</i> data type, which can be used to store a pointer to an activity instance, for instance in the midst of a decomposition algorithm. Individual attributes, parameters and local variables of that activity instance can then be accessed (and modified) through the instance variable.
3	Global variables, which can be accessed and modified anywhere in the adaptation code.
4	Resource arrays; by referencing an existing list of many elements, a short resource declaration can be used to define a large number of distinct resources. More importantly, maintenance is greatly facilitated: if a new device was added to the S/C design, the adaptation code had to be modified in only one place.
5	Local variable declarations, which use the syntax already used for parameter declarations.
6	Conditional (<i>if</i>) and iterative (<i>while</i>) execution of a block of code. Code blocks are delimited by curly braces { and }, similar to the C language.
7	Function calls and assignments.
8	A new type of activity decomposition called <i>nonexclusive decomposition</i> .
9	A new, optional modeling style that supports <i>concurrent modeling</i> in activity type and abstract resource definitions. This option can be used instead of a resource usage section to provide a more natural programming environment to APGEN adapters. When using this option, programmers can adopt a real-time-programming style which includes the ability to <i>wait</i> and to send and catch <i>signals</i> .
10	An optional user-defined library, which can be used to extend the APGEN DSL through the addition of code external to APGEN.

Enhancements 1 through 7 basically turn the APGEN DSL into a full-fledged, if non-standard, programming language. Although they provide APGEN adapters with much more flexibility than was available in the original DSL, they do not change the basic decomposition and remodeling algorithms in Table 4. The last three features, numbers 8 through 10 in the list, *do* make significant changes to these basic algorithms, as we discuss below.

A. Improvements to the decomposition algorithm

Prior to the availability of nonexclusive decomposition, the way in which APGEN handled activity decomposition had side effects that were potentially harmful. This is because of a couple of features that were introduced in APGEN from the very beginning:

1. When a parent activity in the plan, say A, has a couple of child activities, say B1 and B2, only one set of activities can be visible at a time: either A is visible while B1 and B2 are hidden, or B1 and B2 are visible while A is hidden. The user can navigate between these two possibilities by using the *abstract* and *detail* menu options of the APGEN GUI; *abstract* replaces visible children by their parent, and *detail* replaces a visible parent by its children.

2. In the mind of the early APGEN designers, a parent activity and its child activities represent alternative views of the same underlying reality. For example, a S/C maneuver could be represented as a single activity called Maneuver, or by three child activities called First Turn, Burn, and Second Turn respectively. It is up to the user whether the maneuver activity should be represented in symbolic form by the Maneuver activity, or in more detailed form by the First Turn, Burn and Second Turn activities. Accordingly, APGEN would model the activity plan differently, depending on which level activities were presented to the user. If the Maneuver were visible, then only the resource usage section of the Maneuver activity would contribute events to the event queue; if the child activities were visible, then only their resource usage sections would be contributing.

In short, we can say that activity decomposition as implemented in early APGEN is *exclusive*. Exclusive decomposition presents difficulties for both adapters and users. The first difficulty confronts adapters, who have to maintain compatibility between the resource usage patterns used to model the parent and those used to model the children. Obviously, computing resource usage in a consistent manner for a plan that can be viewed at several levels of abstraction requires more work than computing resource usage for a plan in which the level of abstraction is fixed. The second difficulty is that the availability of exclusive levels of abstraction makes things confusing for the user when the plan contains hundreds of activities, as is the case for example for the Cassini cruise plan. The user might be looking for resource usage features that are only present at a certain level of abstraction, for example; if the user inadvertently displays the plan at another level, those features will be missing for no obvious reason.

Once non-exclusive decomposition became available, exclusive forms of decomposition all but disappeared from APGEN adaptations, except in regression tests. As a result, the semantics of activity decomposition have become simpler, and APGEN adapters have been freed from the burden of maintaining consistency between distinct versions of an activity plan.

B. Changes to the remodeling algorithm

We now turn to item 9 in Table 6, which concerns the introduction of a *concurrent modeling* option. To illustrate the impact of this change, we will again turn to the Data Volume modeling problem solved in the previous Section using the methods of early APGEN.

Before we turn to concurrent modeling per se, we discuss one of the negative impacts of items 1-7 in Table 6 when used in conjunction with the original APGEN DSL infrastructure. One could use the improvements in Table 6 to improve the appearance of the adaptation code. Looking at the code in Fig. 4, we note that the code would become more elegant, more readable and more scalable if the *when* clauses could be replaced by a more standard *if ... else* construction. We would also eliminate some duplication by using a local variable *delta* to store the quantity $now - LastUpdateTime.currentval()$. This would lead us to replacing the *AddData* code in Fig. 4 by the code in Fig. 7 below. This code is indeed more elegant than that in Fig. 4, but it is unfortunately incorrect and will not lead to the correct modeling behavior.

Figure 7. An improved, but incorrect, version of the *AddData* code

```
resource AddData: abstract
begin
parameters
rate: float default to 0.0;
resource usage
delta: float default to (now - LastUpdateTime.currentval()) / 0:0:1;
if(delta > 0.0)
use DataVolume(DataRate.currentval() * delta);
else
use DataVolume(0.0);
use DataRate(rate);
use LastUpdateTime(now);
end resource AddData
```

The reason why this code is incorrect is that it gets executed during the event generation phase, steps 2 and 3 in Table 5, before any modeling commands have been processed. At that point, resources have been initialized, but their history does not yet reflect the effect of any commands in the event queue. As a result, *LastUpdateTime* will have the value listed in its profile (see Fig. 4), and not the value expected at the start time of the *science* activity.

It is to remedy the highly non-intuitive consequences of coding patterns such as shown in Fig. 7 that the concurrent modeling option was proposed. The correct version of the code is shown in Fig. 8 below.

Figure 8. An improved and correct version of the *AddData* code based on concurrent modeling

```

global time LastUpdateTime = 2020-001T00:00:00;
resource AddData: abstract
begin
  parameters
    rate: float default to 0.0;
  modeling
    delta: float default to (now - LastUpdateTime) / 0:0:1;
    if(delta > 0.0)
      use DataVolume(DataRate.currentval() * delta);
    else
      use DataVolume(0.0);
      use DataRate(rate);
      LastUpdateTime = now;
end resource AddData

```

Two differences are evident with the previous version of *AddData*: first, the modeling code is contained in a section entitled *modeling*, not *resource usage*; this provides APGEN with a hint that we wish to use concurrent modeling instead of a *a priori* expansion. Second, *LastUpdateTime* is now a global variable and not a resource. This second change is not necessary but it demonstrates the combined benefits of having introduced global variables as well as concurrent modeling; global variables require less space and time overhead than concrete resources.

In a nutshell, the code above will execute correctly because APGEN behaves as a multi-threaded application, in which the modeling code in *AddData* and the scanning of the event queue are executed simultaneously. By postponing execution of the modeling code till the event processing stage, the APGEN adapter can write the code as if it were executed in real time.

To explain how APGEN multi-threading works, we need to use the concept of *execution thread*. Execution threads are nothing new; any application that features a DSL needs to provide something like an execution thread in order to execute an interpreted program. Loosely speaking, an execution thread provides the basic infrastructure required to run an interpreter: a program counter, searchable lists of global and local variables, stacks for function calls, etc. Suppose now that we wanted to execute a certain modeling program *P* when model time reaches a given value *T* during the event processing stage of remodeling. We can do this by creating a new type of event, called a *resumption event*, which holds a copy of the execution thread for *P*. We refer to this event as *RE*. We set the time tag of *RE* to *T*, and we insert it into the event queue. Later on, when the iterator that scans the event queue reaches *RE*, the execution thread encapsulated in *RE* will be re-enabled and execution of *P* will resume.

Table 7 below shows the operation of resumption events when *AddData* is implemented using concurrent modeling, as opposed to the *a priori* modeling style illustrated in Table 5.

Table 7. Sequence of steps taken by APGEN when modeling a science activity using concurrent modeling

<i>Step</i>	<i>Description</i>	<i>Final State</i>
1	APGEN scans the time-ordered list of all activity instances – there is only one in this case – and extracts the two modeling commands it finds in the activity’s resource usage sections (see Fig. 5 above).	Two modeling commands are found, both involving abstract resource <i>AddData</i> ; the first occurs at the start of the activity, denoted by <i>S</i> , the second at the end of the activity, denoted by <i>E</i> .
2	The definition of <i>AddData</i> contains the <i>modeling</i> keyword, indicating the need for concurrent modeling. A new execution thread is created, pointing to the modeling program in <i>AddData</i> ; this thread is encapsulated in a resumption event with time tag <i>S</i> .	The event with time tags <i>S</i> is inserted into the event queue. It contains an execution thread for running the modeling program in the definition of <i>AddData</i> .
3	The second modeling command involving <i>AddData</i> is processed similarly, resulting in a new execution thread encapsulated in a resumption event with time tag <i>E</i> .	The event with time tag <i>E</i> is inserted into the event queue. It contains an execution thread for running the modeling program in the definition of <i>AddData</i> . The event queue contains two events.

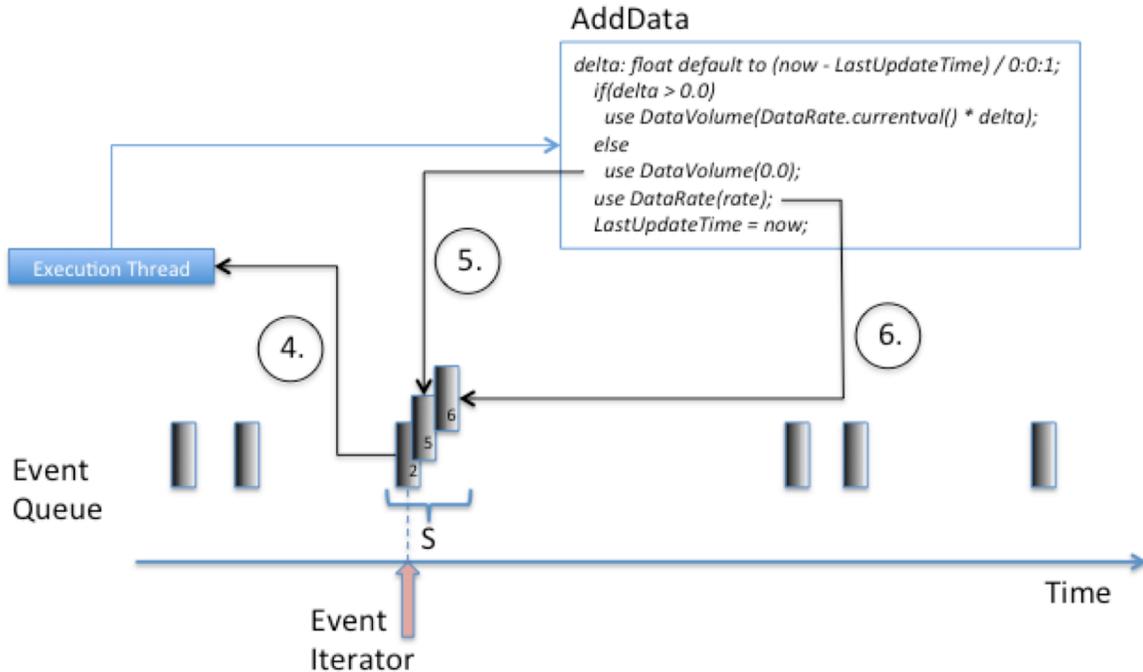
Note: this concludes the event generation stage of the remodeling process.

4	APGEN scans the event queue in time order using an iterator. The iterator runs into the resumption event previously stored at time S . It sets the value of global time variable now to S .	The event queue wakes up the execution thread attached to the event at time S and loads it in the APGEN interpreter. The APGEN interpreter starts executing the modeling section of <i>AddData</i> as shown in Fig. 8. A variable called $delta$ is initialized as shown.
5	Because the current value of <i>LastUpdateTime</i> is greater than now (which was set to the time tag S of the current event), $delta$ is found to be negative. The interpreter skips to the <i>else</i> and creates a modeling command saying <i>use DataVolume(0.0)</i> at time S .	The new event is inserted in the event queue; it has the same time tag S as the current event but since it was inserted later it will occur next in the iterator scan of the queue. This remark applies to all subsequent insertions of new events.
6	The interpreter continues executing the <i>AddData</i> code and creates a modeling command saying <i>use DataRate(rate)</i> . It encapsulates this command in a new usage event at time S .	The new event is inserted in the event queue, also at time S .
7	The interpreter continues executing the <i>AddData</i> code and finds the instruction <i>LastUpdateTime = now</i> . It executes that command immediately. The interpreter notes that this is the last statement in the <i>AddData</i> modeling program; it exits, thus returning control to the event queue iterator.	Global variable <i>LastUpdateTime</i> is now equal to S .
8	The event iterator iterates and finds the usage statement saying <i>use DataVolume(0.0)</i> . The statement is executed.	A new node recording the usage of 0 Megabits is inserted in the history of the <i>DataVolume</i> resource with time tag S . This node is necessary and marks the start of a change in slope; without it, the slope would depend on the time tag of the first node in <i>DataVolume</i> 's history.
9	The event iterator iterates again, finds the usage event that says <i>use DataRate(rate)</i> and executes it.	The value of the <i>DataRate</i> resource is changed by the amount $rate$ at time S .
10	The iterator runs into the event previously stored at time E . It sets the value of global time variable now to E .	The event queue wakes up the execution thread attached to the event at time E and loads it in the APGEN interpreter. The APGEN interpreter starts executing the modeling section of <i>AddData</i> as shown in Fig. 8. A variable called $delta$ is initialized as shown.
11	Because the current value of <i>LastUpdateTime</i> (which is S) is less than now , $delta$ is found to be positive. The interpreter creates a usage event saying <i>use DataVolume(...)</i> at time E .	The new event is inserted in the event queue at time E .
12	The interpreter continues executing the <i>AddData</i> code, skips the <i>else</i> and creates a usage event saying <i>use DataRate(rate)</i> at time E .	The new event is inserted in the event queue, also at time E .
13	The interpreter continues executing the <i>AddData</i> code and finds the instruction <i>LastUpdateTime = now</i> . It executes that command immediately. Since this is the last statement in the <i>AddData</i> modeling program, control returns to the event queue iterator.	Global variable <i>LastUpdateTime</i> is now equal to E .
14	The event iterator iterates and finds the usage event saying <i>use DataVolume(...)</i> . The statement is executed.	A new node recording the usage of an appropriate number of Megabits is inserted in the history of the <i>DataVolume</i> resource with time tag E .
15	The event iterator iterates again, finds the usage event that says <i>use DataRate(rate)</i> and executes it.	The value of the <i>DataRate</i> resource is changed by the amount $rate$ at time E .
<i>Note: this concludes the event-modeling phase of the remodeling process.</i>		

Before leaving our simple example, we want to emphasize one issue which is a source of frequent confusion among APGEN adapters: how is control transferred back and forth between the iterator which scans the event queue, and

the interpreter which executes the program pointed to by a resumption event in the queue? To answer the question, we look at the process described Table 7 in a more graphic manner. Figure 9 below illustrates steps 4, 5 and 6 of Table 7; the grey rectangles illustrate the events in the event queue. At the beginning of step 4, the only event with time tag S is the event bearing the label 2; the label reflects the fact that this event was created at step 2.

Figure 9. Graphic illustration of steps 4, 5 and 6 in Table 7



At the beginning of step 4, the execution thread pointed to is reactivated; execution of the modeling program of `AddData` resumes as indicated in Table 7, and in steps 5 and 6 the modeling commands in the program result in two new events (labeled 5 and 6 in Fig. 9) being inserted into the event queue.

At the conclusion of step 4, concrete resources `DataVolume` and `DataRate` have not yet been affected by the events at time S . It is only when the event iterator advances to events 5 and 6, after the modeling program of `AddData` has completed, that the usage statements take effect. This is illustrated in Figs. 10 and 11 below.

Figure 10. Graphic illustration of step 8 in Table 7

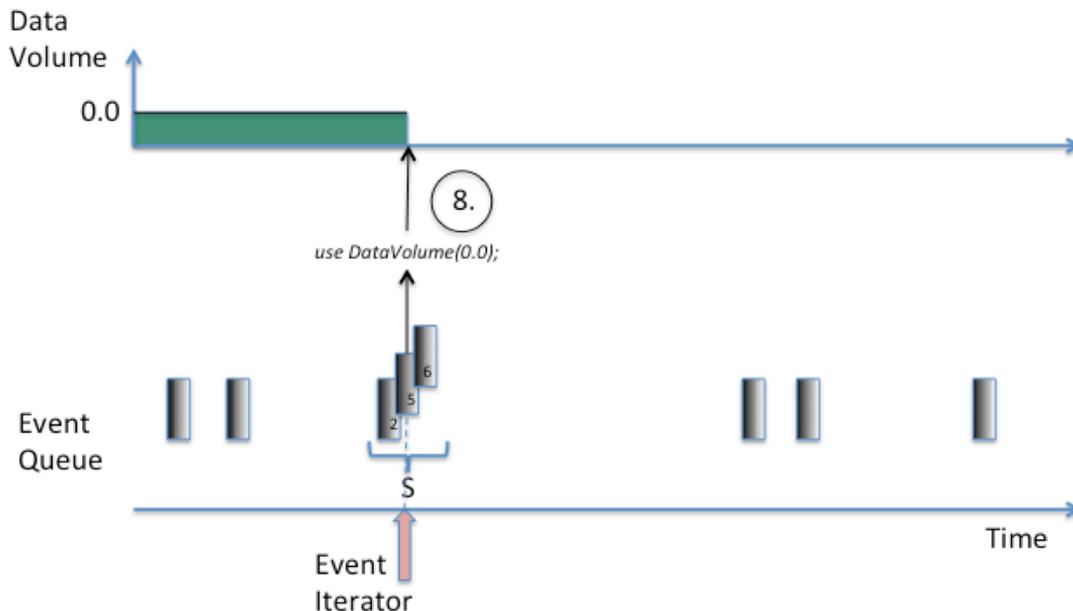
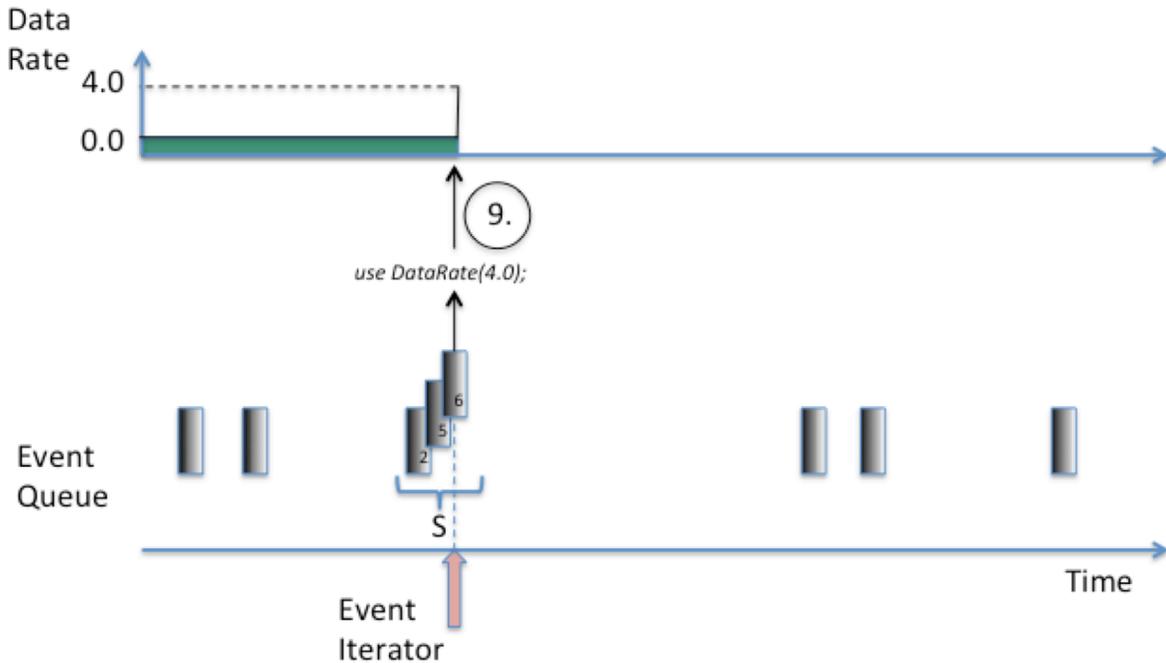


Figure 11. Graphic illustration of step 9 in Table 7



It is easy to lose track of the flow of control when writing a modeling program such as the one in *AddData*. One way to bring resource histories in synch with the execution of the modeling program is to suspend execution of the modeling program after each modeling command, as shown in Fig. 12 below. All it takes to suspend execution is to add a *wait* statement with a time duration of zero seconds. This has the effect of pausing execution of the modeling program and creating a new resumption event for it; the resumption event will have a secondary key greater than that of the usage event created just before pausing, and as a result the event iterator will execute the usage event first, then resume execution of the modeling program.

Figure 12. Modified version of the *AddData* modeling program

```

modeling
  delta: float default to (now -
LastUpdateTime) / 0:0:1;
  if(delta > 0.0)
    use DataVolume(DataRate.currentval()
      * delta);
  else
    use DataVolume(0.0);
    wait for 0:0:0;
    use DataRate(rate);
    wait for 0:0:0;
    LastUpdateTime = now;

```

Since the creation of resumption events is expensive, a new feature was added to the APGEN DSL in the form of a new temporal clause: using the keyword *immediately* in a modeling command invoking concrete resource *R* tells APGEN to update the history of resource *R* immediately, instead of inserting a usage event in the event queue. The program in Fig. 13 below has the same effect as the one above, but runs more efficiently.

Figure 13. Version of the AddData modeling program using the *immediately* temporal clause

```

modeling
  delta: float default to (now -
LastUpdateTime) / 0:0:1;
  if(delta > 0.0)
    use DataVolume(DataRate.currentval()
    * delta) immediately;
  else
    use DataVolume(0.0) immediately;
  use DataRate(rate) immediately;
  LastUpdateTime = now;

```

C. Extending the APGEN DSL through external libraries

Here we turn to item 10 in Table 6, the optional user-defined library. APGEN and SEQGEN are not the only applications at JPL that provide high-fidelity simulations of complex systems such as S/C and their ground support equipment. Many specialized tools are available from domain experts at JPL and elsewhere:

- The SPICE toolkit from NASA⁴
- The MMPAT Multi-Mission Power Analysis Tool⁵
- Sleuth, a simulation and verification tool for the DAWN Attitude Control System⁶
- Telecom Forecast Predictor, a tool for predicting the performance of a telecommunications link⁷

APGEN offers adapters a mechanism for attaching such tools to their adaptation. This mechanism is known as the user-defined library; it requires that the external simulation tool be available as a shared C or C++ library. The main components of the user-defined library mechanism are

- The simulation tool as a shared library written in C or C++
- “Glueware” for linking the APGEN parser to the simulation library’s API

The simulation library and the glueware are linked together into a shared library that can be directly attached to the APGEN executable. In the presence of this library, additional functions not built into APGEN become available to the adapter; typically, such functions are available for

- initializing the external library
- setting parameters of the external model
- initiating the simulation at a given time T_1
- propagating the state of the external model from T_1 to T_2
- ending the simulation

As long as the external simulation tool has an API which supports these basic functions, it can be integrated with APGEN, in effect providing the adapter with an enhanced modeling tool at the cost of a small amount of programming.

D. The current remodeling algorithm in APGEN

To facilitate the description of the current remodeling algorithm in APGEN, we first note that the APGEN DSL supports four distinct types of programs:

1. Functions, which can be invoked from anywhere in the adaptation. APGEN offers a number of built-in functions such as $\sin(x)$ and $\text{random}()$; custom functions can be declared and implemented in the adaptation.
2. Decomposition programs, which appear inside activity type definitions under the heading *decomposition* or *nonexclusive decomposition* depending on whether or not the decomposition is exclusive. In addition to standard instructions, these programs support statements of the form


```
ActType(arg1, arg2, ...) at T;
```

 meaning that a child activity of type *ActType* should be created with start time T and with the arguments supplied in the activity call.
3. Resource usage programs, which can appear inside activity type or abstract resource definitions. Besides standard programming instructions, such programs typically contain modeling commands which affect the state of model resources or invoke abstract resources. These programs, identified by

the keywords *resource usage* at the beginning of the program, will be interpreted in the *a priori* modeling style by APGEN.

4. Modeling programs, which also appear inside activity type or abstract resource definitions. In addition to standard instructions and modeling commands, these programs support instructions with a real-time flavor such as *wait for duration*, *wait until condition* and *wait until "signal-name"*. Such programs are identified by the *modeling* keyword and they will be interpreted in the *concurrent* modeling style by APGEN.

For convenience, we will refer to abstract resources containing a resource usage program as *a priori* abstract resource, and to those containing a modeling program as *concurrent* abstract resources.

After these preliminaries, we can describe the current remodeling algorithm in APGEN as shown in Table 8 below.

Table 8. The current remodeling algorithm in APGEN

<i>Step</i>	<i>Description</i>
1	The event queue and resource histories are cleared; resources are initialized to their default value.
2	<p>The activities in the Activity List are scanned in time order.</p> <ol style="list-style-type: none"> a. If an activity in the list has a resource usage program, that program is executed immediately. Any modeling command referencing a concrete resource results in the creation of a usage event in the event queue. Any modeling command referencing an <i>a priori</i> abstract resource is executed immediately, as if it were a function call. Any modeling command referencing a <i>concurrent</i> abstract resource results in the creation of a resumption event pointing to the modeling program of that abstract resource. b. If an activity in the list has a modeling program, that program is paused at the very beginning and a resumption event pointing to it is added to the event queue. <p style="text-align: center;"><i>This concludes the event creation stage of the remodeling process.</i></p>
3	<p>The events in the event queue are scanned in time order.</p> <ol style="list-style-type: none"> a. Any usage event referencing a concrete resource is executed, resulting in an update of that resource's history. b. Any resumption event causes the program it points to to be reactivated; execution of that program resumes at the point where it was paused. Any modeling commands encountered during execution are executed exactly the same way as in the event generation stage of the remodeling process. <p style="text-align: center;"><i>This concludes the event modeling stage of the remodeling process.</i></p>

E. The Introduction of Scheduling Capabilities in APGEN

One of the requirements that had been levied onto APGEN was that it should assist its users by suggesting where certain types of activities could be added to the plan. The vision behind this requirement was that a scientist could start an APGEN session with a skeleton plan, i. e., a plan containing only engineering activities necessary for maintaining the health of the S/C. The scientist could then choose a type of science activities and ask APGEN to place it as judiciously as possible into the activity plan. For this to be possible, APGEN would have to be given rules and constraints relative to the placement of such activities. Given such rules and constraints, some type of a reasoning engine inside APGEN would come up with suggestions to the user, telling him or her where the activities could be placed while satisfying all rules and constraints.

The requirement that APGEN should feature a reasoning engine went unmet for a number of years, until an opportunity presented itself thanks to the MER (Mars Exploration Rover) mission. MER was concerned that without such an engine, it was going to be impossible to conduct tactical planning sessions within the short turnaround time allocated to surface operations. MER enlisted the help of the AI group at ARC (the NASA Ames Research Center), which had developed a search-based automatic planner called Europa. In a couple of years' development time, APGEN and Europa were merged into a single mixed-initiative tool called MAPGEN; the MAPGEN tool is still in use today as part of MER operations. MAPGEN was discussed in Ref. 2 and will not be discussed further here.

Here, we concentrate on another attempt to endow APGEN with automated scheduling capabilities, which was initiated just before the start of the joint effort with ARC to develop MAPGEN. This earlier attempt did not have the generality typically found in search-based approaches to planning, such as those discussed in Ref. 3. The APGEN approach to scheduling is incremental in nature, and takes advantage of the forward-propagation character of the simulation engine inside APGEN.

In a nutshell, the APGEN scheduling algorithm consists of a remodeling pass as in Table 4, followed by a second pass during which scheduling actions – i. e., the creation of new activities - can take place. The second pass resembles a remodeling pass, with the differences highlighted in Table 9 below.

Table 9. The scheduling pass in APGEN

<i>Step</i>	<i>Description</i>
1	The event queue is cleared; resource histories are not, so that resource values reported by $R.value(T)$ reflect the state of resource R at the end of the remodeling pass.
2	<p>The activities in the Activity List are scanned in time order.</p> <ul style="list-style-type: none"> a. If an activity in the list has a resource usage program or a modeling program, that program is processed exactly as it was in the remodeling pass. b. If an activity in the list has a scheduling program, that program is paused at the very beginning and a resumption event pointing to it is added to the event queue. In addition to the types of statements available in modeling programs, scheduling programs can also contain special <i>scheduling commands</i> of the form <i>wait until OKtoSchedule(Condition) for Duration.</i> Such a command interrupts the execution of the scheduling program. Execution will be resumed in the event modeling stage only when a <i>scheduling window</i> is found. By definition, a scheduling window is an interval during which <i>Condition</i> is found to be true continuously for a duration at least equal to <i>Duration</i>. <p style="text-align: center;"><i>This concludes the event creation stage of the scheduling pass.</i></p>
3	<p>The events in the event queue are scanned in time order.</p> <ul style="list-style-type: none"> a. Any usage event referencing a concrete resource is executed, resulting in an update of that resource's history. b. Any resumption event causes the program it points to to be reactivated; execution of that program resumes at the point where it was paused. Any modeling commands encountered during execution are executed exactly the same way as in the event generation stage of the remodeling process. Scheduling commands are executed as indicated in step 2b. Typical scheduling conditions refer to present and future values of one or several resources; what makes the computation of scheduling windows possible is the availability of complete resource histories, as opposed to histories truncated at the value of <i>now</i> as in a remodeling pass. Scheduling programs can also create (schedule) new activities, using a syntax similar to that of activity creation statements in decomposition programs. As soon as an activity is created in this manner, it is processed the same way as activities in the Activity List during the event generation stage, resulting in new events being added to the event queue. c. Whenever the event iterator advances and sets the global variable <i>now</i> from its current value T_1 to a new time value T_2, it deletes from resource histories any nodes with a time tag greater than T_1 and less than or equal to T_2. The rationale for doing this is that these values were computed during the remodeling pass; they are stale because they do not reflect the impact of any new activities created by the schedulers. They are about to be replaced by up-to-date values resulting from execution of the modeling events added to the queue during the scheduling pass. <p style="text-align: center;"><i>This concludes the event modeling stage of the scheduling pass.</i></p>

IV. The application of APGEN scheduling to space missions

Here we discuss the application of APGEN scheduling to six NASA missions: DI, Europa, INSIGHT, Juno, MRO and MSL. The application of APGEN to DI (and its successor, EPOXI) has been discussed at length in Ref. 1; the emphasis in that paper was put on the functional aspects of the DI planning tool, which consisted of APGEN linked to five external libraries using the user-defined library mechanism. Here, we do not focus on the functionality as much as on the *manageability* of the adaptation, as expressed by following questions:

1. How much work does it take to adapt APGEN to a new mission?
2. How difficult is it to reuse an APGEN adaptation?
3. How difficult is it to train operations personnel in the art of adapting APGEN?
4. Could APGEN be replaced by one or more standard modeling tools?

While we do not have sufficient INSIGHT to answer all of these questions, we want to pave the way for future discussions and investigations of these issues. To this end, we will focus our questions on the *morphology* of an APGEN adaptation in the following sense:

1. The form and structure of the adaptation: what are the main parts of the adaptation, how do these parts interact?
2. Common features the adaptation may share with other adaptations we have studied: are there patterns that make at least some of the adaptation code reusable?
3. APGEN-isms: does the adaptation make essential use of them? If so, is there a way to replace the non-standard code by an external capability based on standard tools?
4. APGEN glue: an apparent strength of APGEN is its ability to let adapters build an integrated system out of subsystems that were not meant to work together. What are the basic ingredients of this “glue”? Could a similar integration capability be provided in a more standard way?

A. Generic morphology of an APGEN adaptation

We start with a preview of the four morphology questions introduced above.

1. Main parts of an adaptation

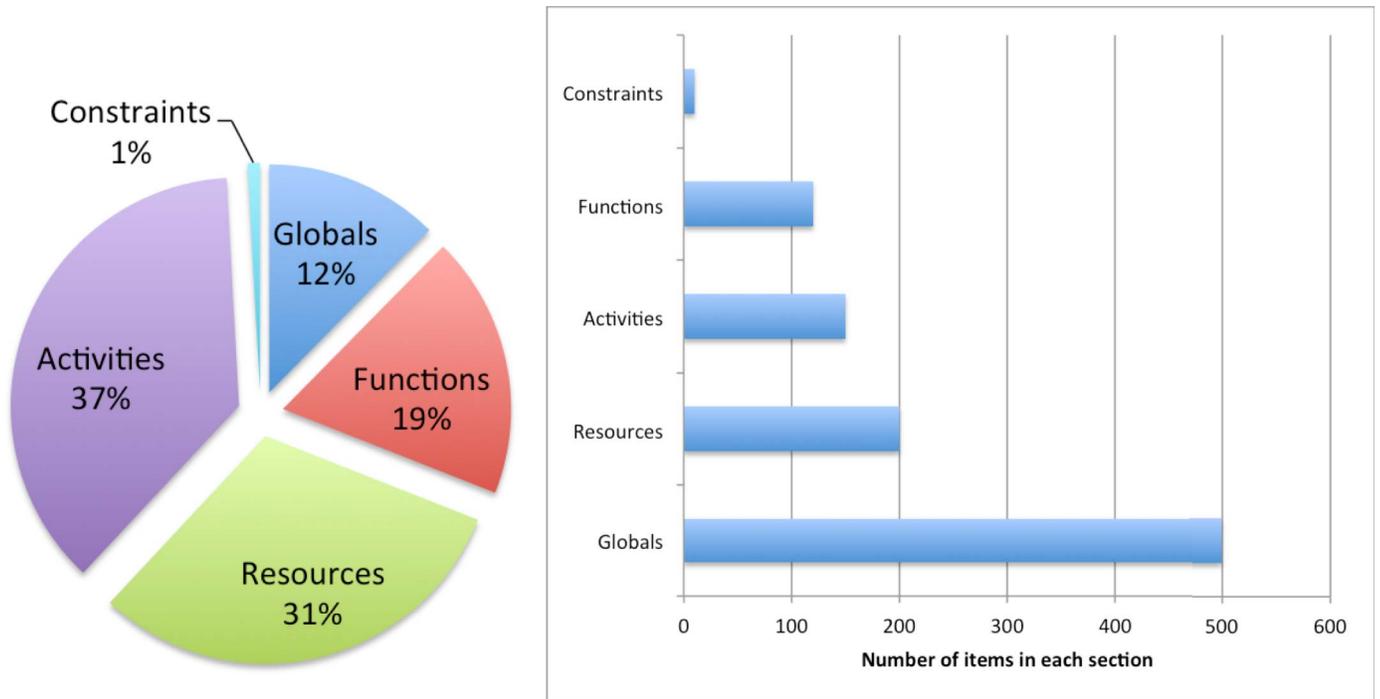
In general, an adaptation file written in the APGEN DSL contains seven distinct components as listed in Table 10 below.

Table 10. The main components of an APGEN adaptation

<i>Component</i>	<i>Description</i>
Header	The header consists of a line stating whether the file contains adaptation or scripting data and some comments regarding the original files from which the adaptation information was taken.
Preliminaries	Preliminaries contain the definition of custom attributes and custom data types (typedefs), directives to APGEN such as the desired size and placement of the APGEN GUI on the screen, the definition of important epochs, and the definition of any non-Earth-based time systems such as Mars Time.
Globals	Globals contain the definitions of all global variables needed by the adaptation code. APGEN does not distinguish between variables whose value is never changed (constants) and true variables whose values can be changed by statements in the adaptation.
Functions	This part of the adaptation file contains the definitions of custom functions invoked by other sections in the adaptation.
Resources	This part of the file contains the definitions of all APGEN resources. A resource can be concrete or abstract. A concrete resource can be scalar or an array; it has a data type and a usage pattern type that must be one of consumable, non-consumable or state. An abstract resource can be written in the a priori modeling style, in which case it contains a section entitled <i>resource usage</i> , or in the concurrent modeling style, in which case it contains a section entitled <i>modeling</i> .
Activity	This part of the file contains the definitions of all activity types except for the <i>generic</i> activity type which is available by default. Activities can contain attribute definitions, parameter definitions, an exclusive or non-exclusive decomposition section, a resource usage or modeling section, and creation and destruction sections.
Constraints	Constraints are basically conditions that are expected to hold true throughout the activity plan. A constraint must conform to one of a few specific types: logical condition, forbidden overlap etc. Constraints are purely passive; if violated, the violation is reported but no attempt is made to eliminate it.

The first two components are not an issue since they exhibit minimal complexity and do not present difficult issues. The remaining five components constitute the bulk of a typical APGEN adaptation; the following figure was compiled based on the 6 APGEN adaptations we have studied.

Figure 14. Size of the components of an APGEN adaptation, as a percentage of total and as an item count



2. Recurring patterns in APGEN adaptations

The following patterns are found in the APGEN adaptations we have studied:

- The global section is usually divided into a large number of small global definitions and a couple of very large global arrays. The large arrays are used to represent things like DSN station configuration codes (of which there is a very large number) or the individual stars held in the memory of the Star Tracker (as was done for Deep Impact).
- Functions likewise follow a bimodal distribution: there is a significant number of short utility functions and a small number (fewer than 10) of complex functions whose implementation takes several hundred lines of code.
- Concrete resource definitions are generally short; note that a single resource array definition referencing a global array contains in effect as many individual resources as there are elements in the array.
- Abstract resources can be divided into two groups: those whose resource usage or modeling section is moderate in length (of the order of 20 lines), and those with complex modeling algorithms. The first group is often used to act as an API in front of simple resources arranged as one or more resource arrays; the second group is used to model complicated subsystems such as the Attitude Control System or the sophisticated API (Applications Programming Interface) of a flight instrument.
- Activity type definitions likewise fall into two groups: a hundred or so activity types of moderate complexity (30 lines or so) and a dozen complex activity types, usually schedulers, with algorithms several hundred lines in length.

Figures 15 and 16 below illustrate the general tradeoff we have observed between item number and item size.

Figure 15. Tradeoff between number of items and item size for key APGEN adaptation components

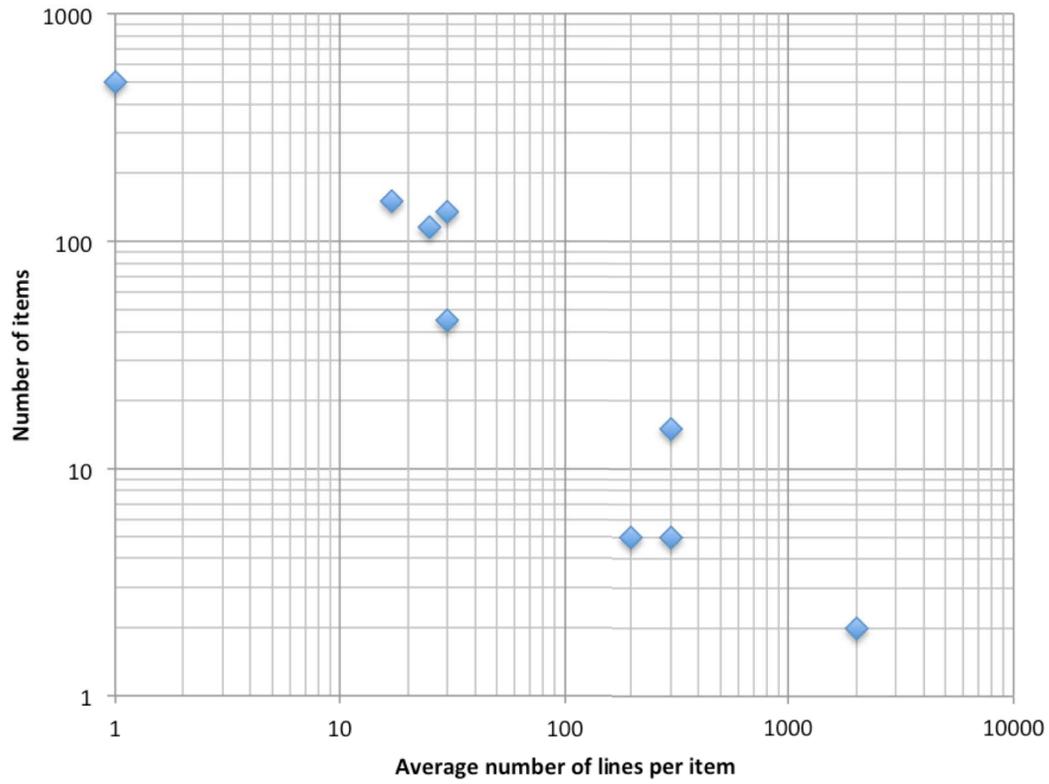
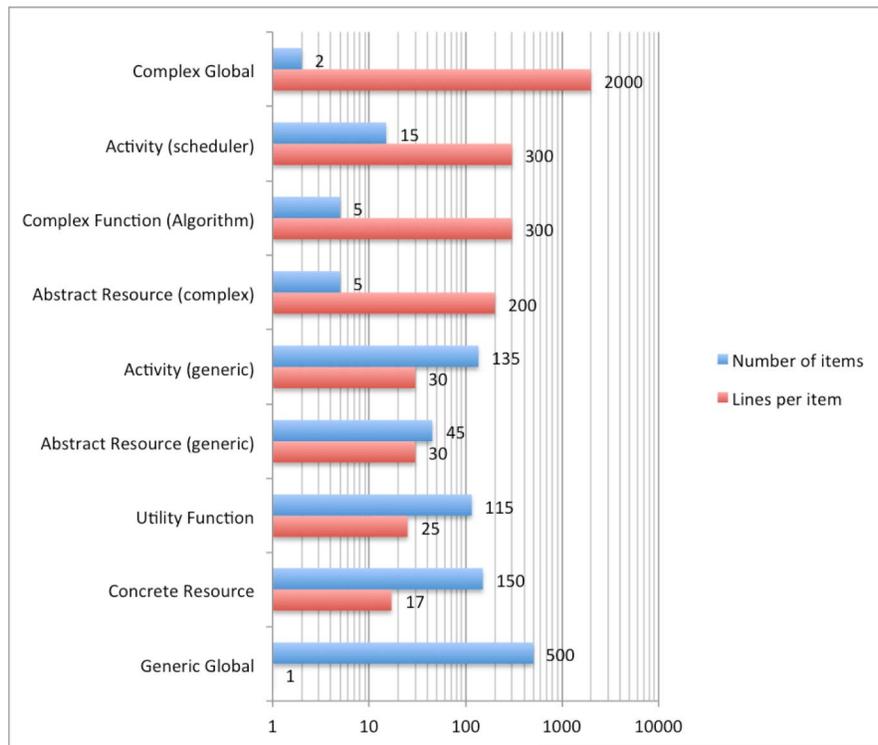


Figure 16. The data for the items plotted in Fig. 15.



3. *Opportunities for improvement and automation*

The dream multi-mission application is one that can be adapted to a new mission with virtually no effort. Most real applications fall short of that dream. Because of the ever increasing complexity of space mission and the continued need to reduce costs, it is therefore important to understand where the adaptation effort goes, and how the level of effort could be reduced.

The charts shown above suggest some possible answers to these questions. Here are our observations:

- The most labor-intensive parts of the adaptation are the complex functions, abstract resources and schedulers. Fortunately:
 - there are relatively few of these
 - many of them are common to more than one adaptation, indicating that they are generic in nature
 - in many cases, good documentation is available to guide future adapters
- Items found in large numbers, such as simple global variables, are good candidates for automatic generation from Systems Engineering databases
- Items of moderate complexity such as utility functions are candidates for integration with other modeling tools and libraries. Example: vector math libraries in matlab, ACS quaternion manipulation utilities...
- Items of moderate complexity found in large numbers, such as short activity and concrete resource definitions, might be extracted from SE databases or from external repositories such as VML programs or SEQGEN files. Alternatively, modeling such activities could be delegated to external tools, such as a VML simulation program or a SEQGEN modeling server.

B. The DI/EPOXI mission

The DI (Deep Impact) mission was a NASA Discovery mission. During a flyby of comet Tempel 1, it released a small impactor spacecraft which collided with the comet in an explosion that was observed around the world and led to a wealth of science data. The EPOXI follow-on mission (the acronym combines EPOCH, Extrasolar Planet Observations and Characterization, and DIXI, Deep Impact Extended Investigation) featured both an encounter with comet Hartley 2 and a series of extrasolar planet observations. DI/EPOXI provides the most ambitious example of an APGEN adaptation in terms of comprehensive integration with high-fidelity models.

On the DI part of the mission, the primary benefit of the high-fidelity APGEN model was to have all the subsystems integrated into the planning model, which allowed the Encounter System Lead (Steve Wissler) to build and validate many versions of encounter sequences quickly and limit testbed use to sequences that were likely to succeed. Note that it took 36 hours to simulate an encounter sequence on the DI testbed.

The EPOXI part of the mission was run on a very tight budget; as a result, the small team of Systems Engineers running the mission had little Subsystem support. As a result, the Systems Engineers enhanced the APGEN model to allow the small team to quickly build and validate sequences without the usual interaction with Subsystems Engineers. In the past, such interactions were the source of many iterations through building and validating S/C sequences prior to uplink. Since more automation was needed than in the DI phase of the mission, it was decided to take advantage of APGEN scheduling and enhance the DI adaptation of APGEN.

1. *Main parts of the DI adaptation*

APGEN scheduling was used in two areas:

- To build observations and downlink sequences in EPOCH
- To build background sequences during the Hartley 2 encounter

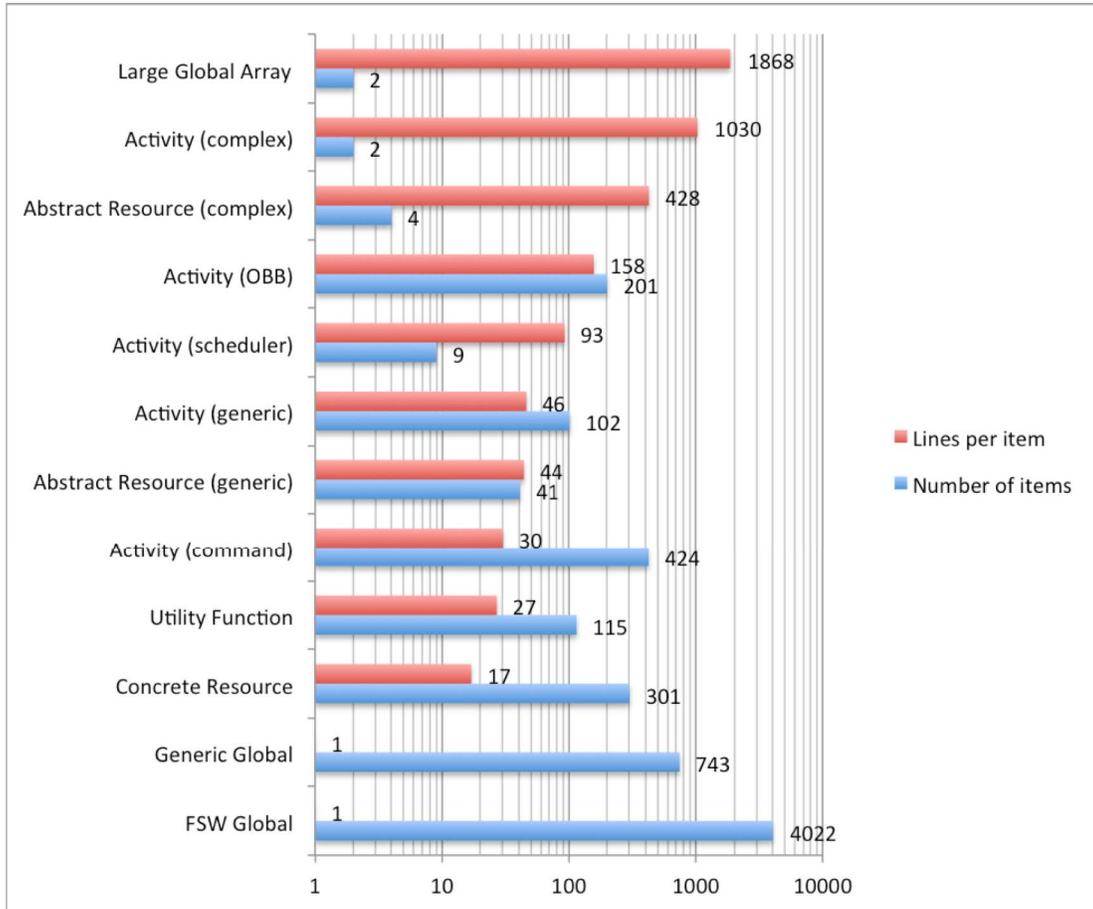
Figure 17 below shows complexity and size data for the individual sections of the DI/EPOXI adaptation. Adaptation items fall into the following categories:

- Highest complexity items:
 - Complex Activities
 - Complex Abstract Resources
 - Large Global Arrays
- Moderate complexity items:
 - Onboard Blocks
 - Schedulers
 - Generic activities
 - Generic abstract resources
- Low complexity items:

- Commands
- Concrete resources
- Utility functions
- Simple items:
 - FSW globals
 - Generic globals

We can use the above information to comment on an efficient strategy for developing a comprehensive, system-wide adaptation of APGEN. Note that for this to be reasonably complete, we also need to comment on the complexity of the glueware, and on any correlations between glueware and adaptation complexity.

Figure 17. Number of items and number of lines per item for the main components of the DI adaptation of APGEN



2. Recurring patterns in the DI adaptation

Table 11 below lists the key elements of the DI adaptation, together with information that provides a hint of how much planning-specific labor is involved in the creation of each element.

Table 11. Key elements of the DI adaptation

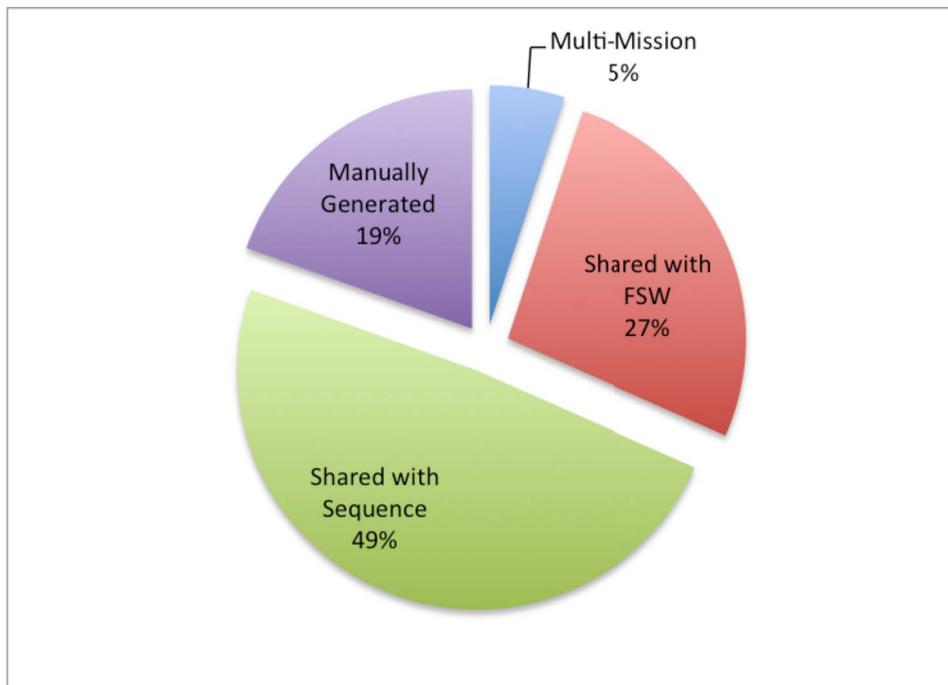
Category	Item (if itemized) Number of items (if not)	Size	Multi-Mission	Possible Template	Shared with...	Manually generated
Large Global Array	Star Tracker	2135			FSW	
	DSN Configuration	1600	Y		DSN	
Complex Activity	Tlm_fb_mode	1745			Sequence	
	InitialConditions	311			Sequence	
Complex Abs. Res.	Station_Viewperiods	265	Y			Y
	CCDRReadout	371				Y

	Geometry_Control	724		Y		Y
	ComputeSlew	352		Y		Y
Activity (OBB)	201	31713			Sequence	
Activity (scheduler)	9	834	Y*			Y
Activity (generic)	102	4800			Sequence*	Y*
Abs. Res. (generic)	41	1842				Y
Activity (Command)	424	13476			FSW	
Utility Function	115	3130	Y*		Library*	Y
Concrete Resource	301	5294				Y
Generic Global	743	743				Y
FSW Global	4022	4022			FSW	

*: Only some items in this category have the indicated property

We can now use the data in the above table to get an idea of how much manual labor is actually involved in creating the adaptation. The pie chart in Fig. 18 below shows that out of the 74,000 lines of code in the DI adaptation, only 19% or 14,450 lines needs to be manually generated.

Figure 18. Chart showing how much of the total adaptation lines needs to be generated by hand



3. Opportunities for improvement and automation

The above analysis is too crude to provide a true costing model of what it takes to generate an APGEN adaptation as sophisticated as the DI adaptation. The limited analysis carried out above indicates, however, that there is considerable potential for extracting planning adaptation data from non-planning sources. What makes it difficult today to integrate a model such as the APGEN DI model is that data from external sources are difficult to get. For example, code that simulates a given subsystem – say ACS or propulsion – is generally not available to planning personnel, for two reasons: first, the code in question may very well have been developed by a subsystem specialist for his or her personal use, and so the existence of the code is not general knowledge. Second, even if a planning person knows that code is available for simulating a given subsystem, the code is often available as a main program but not as a library, and it takes significant cooperation between the planning person and the subsystem person to end up with a library version suitable for integration with APGEN.

The key to further improvements in the ease of integration of system-wide high-fidelity simulations of a mission system is the ready availability of subsystem data from the subsystem experts who know the most about it. In a future publication we hope to return to this topic and to explore promising directions for the future.

C. The proposed Europa mission

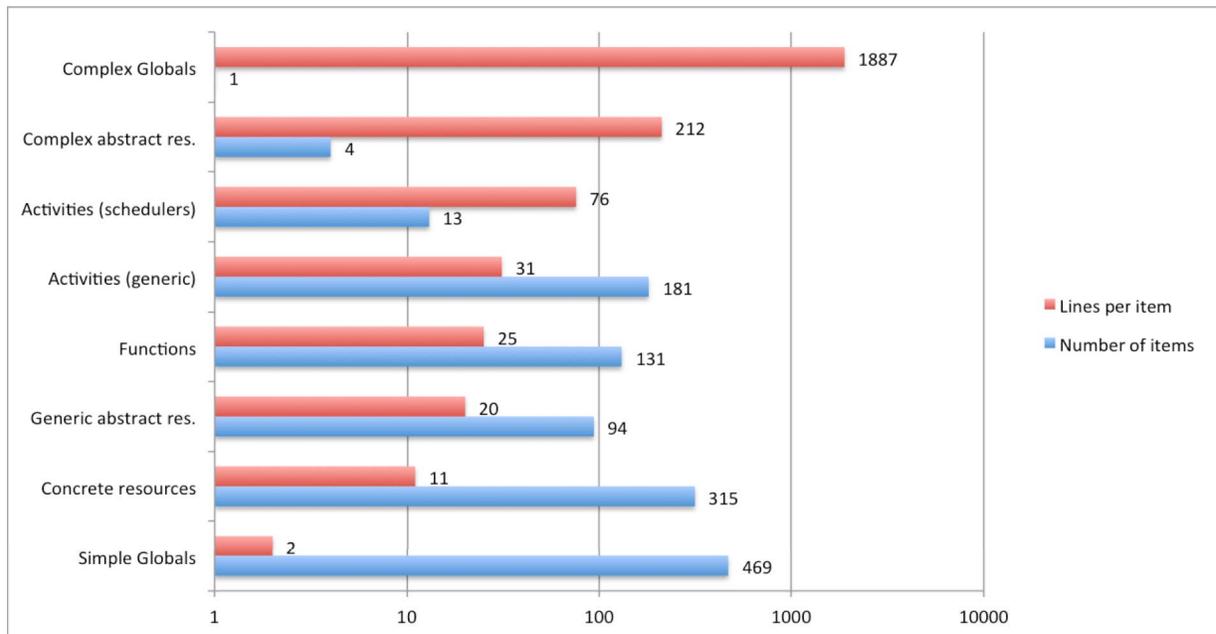
Among the six missions we have studied for this paper, Europa is exceptional not only because it is still in the proposal stage, but because the APGEN model for Europa is remarkably detailed for a mission in such an early proposal and development phase. As a result, a number of features of the Europa adaptation are unusual:

- DSN allocations are not yet available, and it was necessary to develop a scheduler to create such allocations based on realistic assumptions.
- Functionally, the Europa model is as comprehensive as the DI model. However, since detailed subsystems such as attitude control and propulsion have not been finalized yet, APGEN models for these subsystems had to be written at the functional (not command) level.
- Various subsystems, in particular power and instruments, had to be parameterized to allow different S/C design assumptions.
- APGEN simulations are generally run over long periods of time (several years), so as to generate meaningful statistics for e. g. the amount of science return given a certain S/C configuration.
- Although the set of S/C commands is not known, it was necessary to synthesize low-level activities for all low-level functions, such as turning instruments and devices on and off.
- S/C designers are using Systems Engineering tools such as SysML to build an engineering database of S/C design parameters, presenting APGEN adapters with a unique opportunity to derive parts of the APGEN adaptation directly from the engineering data.

1. Main parts of the Europa adaptation

Figure 19 below shows the elements of the Europa adaptation.

Figure 19. Elements of the Europa adaptation of APGEN



2. Recurring patterns in the Europa adaptation

The following observations can be made regarding recurring patterns in the Europa adaptation:

- There was considerable reuse of code: among the 131 functions in the adaptation, 82 are common with at least one of the DI, MRO and Juno adaptation.
- About half of the global definitions are concerned with PEL (Power Equipment List) items which were extracted from the System Engineering database by automated methods.

- The Ground model (which includes the DSN model) and much of the Telecommunications model was mostly inherited from DI also.

3. *Opportunities for improvement and automation*

Although the APGEN adapters were able to tap the System Engineering database for PEL information, other subsystems are still in development. The plan is that as more and more of the subsystem designs gets captured in engineering databases, the APGEN model will be updated through new automated pathways between these databases and the model.

D. The INSIGHT (Interior Exploration using Seismic Investigations, Geodesy and Heat Transport) mission

The INSIGHT lander will carry three science instruments to the surface of Mars. Although the payloads of the two missions are quite different, the spacecraft itself will be very similar to the Phoenix Mars Lander, which landed on the surface of Mars in 2008. As a result, the INSIGHT adaptation of APGEN will reuse a number of features originally developed for Phoenix.

Lander operations typically involve S/C-Ground interactions in a closed loop, so that telemetry from a given sol (Martian day) can be used to plan the activities for the next sol. Although this does not mean that there is no opportunity for automatic plan and sequence generation in a lander mission, the tactical (daily) operations on Phoenix were largely done by hand, with the exception of initial task scheduling which was performed by the Europa planner from NASA's Ames Research Center.

For most of INSIGHT landed operations, the instruments will be operating autonomously, and as a result the need for closed-loop operation between the S/C and ground personnel will be much reduced compared with Phoenix. As a result, there is likely to be significant opportunity for automated scheduling.

One area where some automated scheduling has already taken place is the use of a high-fidelity data model to assist in the selection of the best configuration for the various data buffers onboard the S/C. The INSIGHT data model uses the following inputs:

1. A set of realistic DSN allocations
2. Parameters that determine the sizes of the various internal data buffers

The output consists of a complete scenario spanning the entire duration of the mission. The scenario includes a prediction of how much high-resolution data has been downloaded over the course of the entire mission, assuming that the Science Team always elects to downlink as much data as can be accommodated by the onboard data storage system and telecommunications opportunities with Earth.

The current INSIGHT adaptation only produces engineering data, not S/C sequences. At about 1400 lines of adaptation code, the INSIGHT adaptation of APGEN is therefore much smaller than the other adaptations we discuss. We expect to carry out a more meaningful comparison once some of the mission operations processes have been fully automated.

E. The Juno mission

Juno is a spinning solar-powered spacecraft currently in its Cruise phase towards Jupiter; once there it will enter a highly elliptical orbit that avoids most of Jupiter's high radiation regions. The main purpose of the present Juno adaptation of APGEN is to automate a number of tasks in the Cruise phase. The initial automation effort was carried out for Mars missions such as Odyssey and MRO by Roy Gladden using his AUTOGEN program,⁸ a complex wrapper around APGEN; the result of this effort was the BGS (BackGround Sequence Generation) tool. BGS was subsequently refurbished and adapted to the Cruise phase of the Juno mission by Matthew Lenda who added a new repointing algorithm described below.

1. *Main parts of the Juno adaptation*

Figure 20 below shows the main components of the Juno APGEN adaptation, which implements the automated BGS process. This process requires the following inputs:

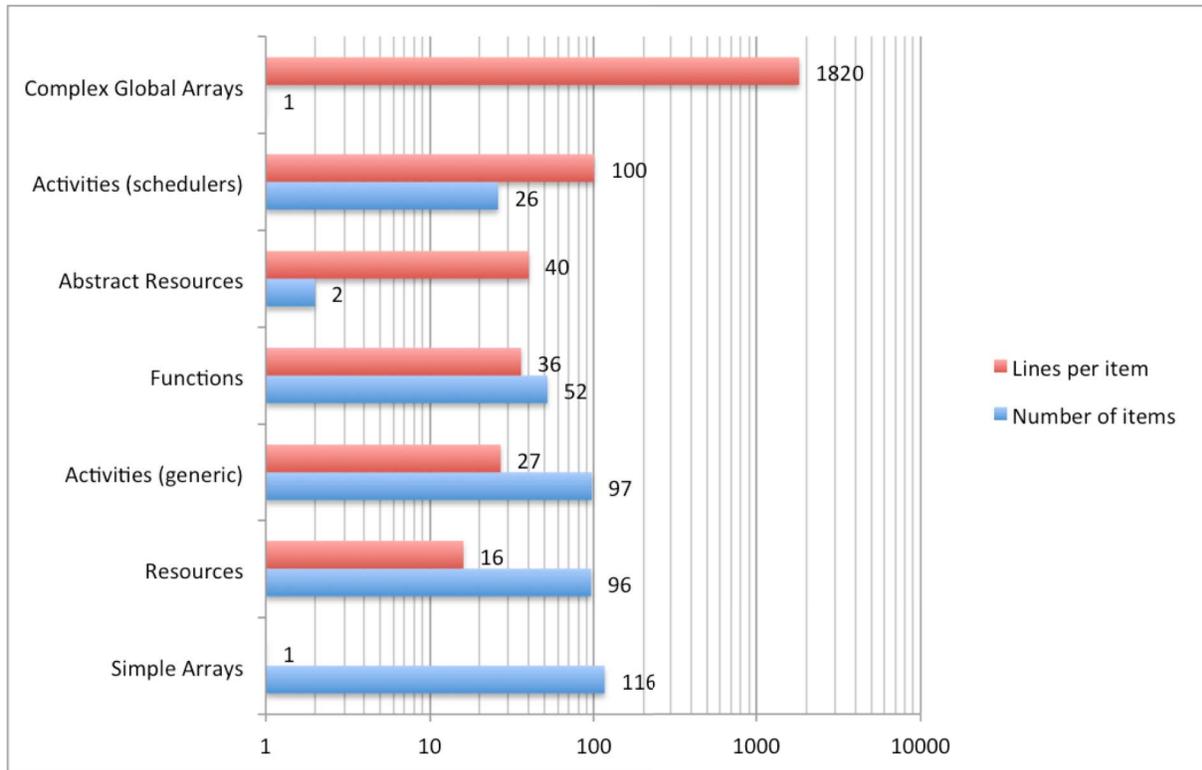
- Spice kernels for the S/C and DSN stations.
- DSN allocations and view period files.

The main outputs from the BGS process are

- SAFs (Spacecraft Activity Sequence File) to be fed to SEQGEN for sequence validation and creation of uplink products.

Note that an SAF contains, besides commands and S/C activities, a number of directives to SEQGEN. For example, one of those directives may order SEQGEN to create a FINCON (Final Conditions file) at a certain time in the simulation. This is one of the tasks automated by the BGS process.

Figure 20. Main Components of the Juno APGEN adaptation



The tasks automated by the BGS process fall into two groups: those that must be performed in all phases of the mission, and those that are specific to a particular phase.

The following tasks are common to all phases:

1. Schedule FSW diagnostics activities
2. Schedule FINCON requests in the output sequence file
3. Change the downlink rate on the S/C corresponding to the setup of the allocated dish on the ground
4. Schedule telecom setup and teardown (“comm blocks”) activities around gaps in DSN coverage, including DDOR (Delta Differential One-way Ranging) activities
5. Identify overlaps in DSN allocations and distinguish between handovers and UL (UpLink) transfers
6. Find all UL transfer times during allocation overlaps to determine additional retransmit times
7. Identify all 1-way to 2/3-way Doppler Mode transitions after the start of a comm block and manage the DPT (Data Priority Table) mode
8. Identify all downlink rate changes and determine the DPT/FPT (Frame Priority Mode) and comm_begin/end management commanding
9. Query resources and states at the sequence cutoff time and print them to an output file

Phase-specific tasks involve attitude commanding. This type of commanding is required to control the so-called “turn angle” of the Juno S/C. The turn angle is defined as the angle between the Juno S/C spin axis vector and the Juno-to-Earth vector. Nominally, the S/C spin axis is the S/C +Z axis, which is also the HGA boresight axis. There is generally a small angular difference between the spin axis and the Juno-to-Earth vector. The S/C does not continuously maintain Earth-point, and as a result these two vectors drift apart over time. To re-establish alignment, a repointing activity is required; different activities are actually used, depending on the size of the correction. The following tasks are specific to the Cruise phase:

1. Identify and define parameters of repointing activities during the Cruise phase
2. Schedule repointing activities to slew S/C’s HGA boresight at the Earth during the Cruise phase

The following tasks are specific to the Orbital phase:

1. Execute a small-angle orbital repointing activity 12 hours prior to the middle of each DSN allocation
2. Execute a large-angle orbital activity 19 hours before and 3 hours after each Peijove (Periapsis)

Each one of the above tasks was assigned to a specific scheduler in the APGEN adaptation. These schedulers do not work in isolation; they take advantage of the modeling work provided by the resource model and by non-scheduling activities.

2. *Recurring patterns in the Juno adaptation*

There is a lot in common between the Juno adaptation and previous APGEN work, especially the DI adaptation (for its Ground model) and the AUTOGEN adaptation. As a result much of the adaptation code was inherited from previous missions. On the other hand, about half of the schedulers involve features that are specific to the Juno S/C.

3. *Opportunities for improvement and automation*

As in other examples of adaptations derived from the original AUTOGEN, adaptations such as this one would benefit from true adaptation templates. Currently, re-using a previous adaptation of APGEN generally means hand-editing some of the adaptation files from that previous adaptation. Adaptation templates would remove the need for such edits, making the adaptation much easier to maintain.

F. The MRO mission

1. *Main parts of the MRO adaptation*

The MRO (Mars Reconnaissance Orbiter) S/C was launched in 2005 to provide high-resolution remote sensing observations of the surface of Mars and to provide a high-data-rate communications relay for Mars surface missions. Because it is an orbiter, its operation is highly repetitive. As a result, planning and sequencing personnel decided to modify Roy Gladden's AUTOGEN tool⁸ and adapted it to the MRO mission. Recently, there was a need to revisit the algorithms in AUTOGEN, in part because the original scripts needed maintenance and in part because the complexity of the scheduling requirements kept increasing. The current version of AUTOGEN, named AUTOGEN Mk-II, was put together by Matthew Lenda.

AUTOGEN Mk-II is a collection of scripts which have at their core an adaptation of APGEN. This adaptation has much in common the earlier DI adaptation; in particular, the Ground model used to track the state of DSN stations is identical to that used in DI (and also with the models used in Europa and MSL Cruise).

Just like Juno and other missions that use some version of AUTOGEN, a number of tasks have been automated by the adaptation. The list of tasks that were automated for the MRO mission is shown below. A special scheduler was written for each specific task.

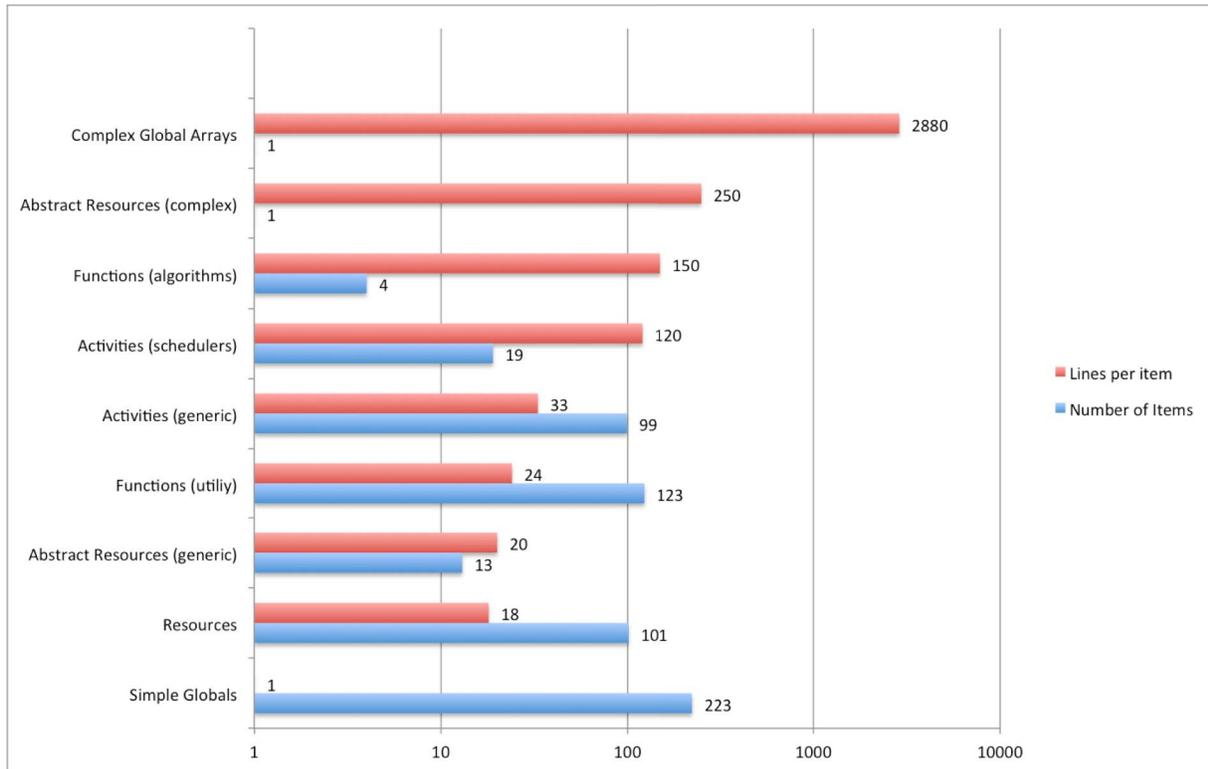
1. Orbital Geometry Events

Schedulers were written to handle the following types of events:

- a. Occultations
 - b. Eclipses
 - c. Periapsis and Apoapsis
 - d. Ascending and Descending Nodes
2. Daily activities
 3. RWA (Reaction Wheel Assembly) desaturation activities
 4. HGA (High Gain Antenna) hard stop Activities
 5. HGA management activities
 6. Ranging and radio science orbits
 7. Downlink data rate selection
 8. Low-elevation downlink suppressors
 9. Communications blocks
 10. Daily activities in contact
 11. Critical low-level activities

The main components of the MRO adaptation of APGEN are shown in Figure 21 below.

Figure 21. Main components of the MRO adaptation



2. Recurring patterns in the MRO adaptation

The logic of each one of the MRO schedulers is generic; only the specific low-level activities created by the schedulers vary from one mission to another. As a result, there is potential to turn much of the MRO adaptation into a genuine multi-mission adaptation template. Right now, APGEN does not support templates; it can only accept fully specified adaptations. Extending APGEN to adaptation templates is one of the options we are looking at for future development.

3. Opportunities for improvement and automation

Some of the algorithms in the MRO adaptation of APGEN would benefit from a more flexible way of computing windows of opportunity in the scheduling phase. Currently, a request for such windows has to pause until modeling time reaches the start time of the first window of opportunity. It would be relatively easy to provide this capability without pausing the requesting program, which would allow adapters to write their code in a more flexible and transparent manner.

G. The MSL cruise mission

At 6,200 lines of adaptation code, the MSL adaptation of APGEN is the smallest of the ones we have discussed. The MSL mission was not planning to use the APGEN tool. During the Cruise phase of the mission, it became clear that the generation of communications sequences was very repetitive; it also became clear that sequence generation was going to become more and more challenging as the S/C was nearing EDL (Entry, Descent and Landing), as communications were becoming more and more critical.

What prompted MSL sequence engineers to use an AUTOGEN-like adaptation of APGEN was simply the availability of a suitable Ground model from the DI mission and the availability of scheduling algorithms very similar to those that would be needed for MSL. As a result a prototype adaptation demonstrating the feasibility of automatic sequence generation was put together in a few days, and the mission gave its approval for the development of an APGEN adaptation for the purpose of generating background Cruise sequences.

V. Conclusion

In our paper, we have described the APGEN scheduling tool and we have provided a quick tour of the basic ingredients of an APGEN adaptation, first in terms of basic concepts and then in more detail by reviewing the use of APGEN scheduling in six different missions and proposed missions. Because of the technical depth of the subject, we have only provided basic information concerning the specifics of the scheduling algorithms involved. In our future work, we plan to explore the possibility of making APGEN more modular, so that it could be better integrated with SEQGEN for sequence validation and with other planning and scheduling tools of interest to future space missions.

Acknowledgments

This research was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

References

¹Wissler, S., Maldague, P.F., Rocca, J., and Seybold, C., "Deep Impact Sequence Planning Using Multi-Mission Adaptable Planning Tools with Integrated Spacecraft Models," *AIAA SpaceOps 2006 Conference*, 2006.

²Mitchell, A., Bresina, J., et al, MAPGEN: Mixed-Initiative Planning and Scheduling for the Mars Exploration Rover Mission, IEEE Intelligent Systems [online journal], IEEE1094-7167/04, pp8-12, URL: <http://ieeexplore.ieee.org/iel5/9670/28315/01265878.pdf?arnumber=1265878> [cited 10 May 2006].

³Chien, S., Johnston, M., Frank, J., Giuliano, M., Kavelaars, A., Lenzen, C., and Policella, N., "A Generalized Timeline Representation, Services, and Interface for Automating Space Mission Operations," *AIAA SpaceOps 2012 Conference*, 2012.

⁴Acton, C.H., "Ancillary Data Services of NASA's Navigation and Ancillary Information Facility", *Planetary and Space Science*, Vol. 44, No. 1, pp. 65-70, 1996.

⁵Wood, E., and Adamson, A., "Multi-Mission Power Analysis Tool", URL: <http://www.techbriefs.com/component/content/article/1059-gdm/tech-briefs/9446-npo-47290> [cited Apr. 8, 2014]

⁶Vanelli, A., Swenka, E., and Smith, B., "Verification of Pointing Constraints for the Dawn Spacecraft", *AIAA/AAS Astrodynamics Specialist Conference and Exhibit*, Aug. 2008.

⁷Cheung, K.-M., and Lee, C., "Link-capability driven network planning and operation", *IEEE Aerospace Conference Proceedings*, 2002.

⁸Gladden, R. E., "AUTOGEN: The Mars 2001 Odyssey and the 'Autogen' Process", *AIAA Small Satellite Conference*, Logan UT, August 2002.