Appears in Proceedings, 14th IEEE International Conference on Automated Software Engineering, Cocoa Beach, Florida, 12-15 October 1999. IEEE Computer Society, pp 63-72.

Automatic Generation of Test Oracles - From Pilot Studies to Application

Martin S. Feather
Jet Propulsion Laboratory,
California Institute of Technology
4800 Oak Grove Drive
Pasadena, CA 91109, USA
+1 818 354 1194
Martin.S.Feather@Jpl.Nasa.Gov

Ben Smith
Jet Propulsion Laboratory,
California Institute of Technology
4800 Oak Grove Drive
Pasadena, CA 91109, USA
+1 818 353 5371
Ben.D.Smith@Jpl.Nasa.Gov

Abstract

There is a trend towards the increased use of automation in V&V. Automation can yield savings in time and effort. For critical systems, where thorough V&V is required, these savings can be substantial.

We describe a progression from pilot studies to development and use of V&V automation. We used pilot studies to ascertain opportunities for, and suitability of, automating various analyses whose results would contribute to V&V. These studies culminated in the development of an automatic generator of automated test oracles. This was then applied and extended in the course of testing an AI planning system that is a key component of an autonomous spacecraft.

Keywords: Test Oracles, Verification and Validation, Analysis, Planning, NASA

1. Introduction

Cost, performance and functionality concerns are driving a trend towards use of self-sufficient autonomous systems in place of human-controlled mechanisms. Verification and validation (V&V) of such systems is particularly crucial given that they will operate for long periods with little or no human supervision. Furthermore, V&V must itself be done at low cost, rapidly and effectively, even as the systems to which it is applied grow in complexity and sophistication.

Spacecraft – especially deep space probes – exemplify these concerns. We have been involved in V&V of an AI planner that is a key component of a spacecraft's autonomous control system. In [8] we report our use of an automated generator of automated test oracles to support these V&V activities. The paper is organized to show the progression of steps we followed leading up to this application, and the lessons we have learnt by reflecting upon our experience:

- First pilot study: rapid automated analysis (Section 2). In this study we determined the viability of a rapid analysis approach. We did case studies of two kinds of traditional design information, yielding confirmation of the viability of the analysis method for this kind of information.
- Second pilot study: application to an autonomous planner (Section 3). We needed this second study to determine suitability of the rapid analysis approach to, specifically, checking plans generated by an AI planner. Particular concerns were scalability of the approach, and investment of domain experts' time. The pilot study produced instances of automatic test oracles.
- Development of automated generator of planner test oracles (Section 4). Based on the lessons learned from the second pilot study, we committed to developing a tool to be used in actual spacecraft testing. The tool would go beyond the capabilities of the second pilot study by both extending aspects of the analyses performed, and automating the generation of the test oracles themselves.
- Application to V&V of spacecraft planner (Section 5). We applied the tool during spacecraft planner testing. Using it, we checked thousands of test cases for adherence to hundreds of flight rules. Additionally, we extended it to perform additional validation checks of particularly complex rules.
- Lessons learned (Section 6). We describe lessons learned for both software engineering and V&V:
    - *Our experience re-iterates several well-understood virtues of pilot studies as a precursor to actual development.*
    - *When domain experts' time is a critical resource, follow an "on-demand" policy of knowledge acquisition.*
    - *V&V can make good use of redundancy and*

  rationale, to increase assurance in the V&V results, and to assist in the development of the V&V technology itself.
  - *The use of a database as the underlying analysis engine has practical applications and benefits.*
  - *Test oracles should yield results with far more content and structure than simply "passed" or "failed".*
  - *Translation between notations is a recurring need, and ideally should be done in such a way as to support understanding, specification and maintenance by domain experts.*
- Conclusions (Section 7). We summarize the relationship of our work to other efforts, and point to areas we believe are worthy of additional attention.
- Further details of the second pilot study (Appendix A).
- Further details of the development (Appendix B).

2. First pilot study: rapid automated analysis

The first stage was a pilot study that investigated analysis of simple properties of spacecraft designs. This was conducted in early 1997, primarily by the first author who, while not an expert in spacecraft, had access to spacecraft design documents and spacecraft experts. The purpose of this first study was to answer the following question:

Could simple analyses of spacecraft design information be performed rapidly by using a database as the underlying reasoning engine?

The approach under investigation was founded upon the use of a *database* as the underlying reasoning engine. We used AP5 [3], a research-quality advanced database tool developed at the University of Southern California. The architecture of this approach is shown in Figure 1. Its four main steps were:

1. Manual creation of a database schema to represent the design information.
2. Loading the design information into the database. This was made a predominantly automated operation, by constructing special-purpose programs to extract information from design documents and translate into the format of the database schema. Automation made the approach practical for handling voluminous amounts of design information.
3. Determining V&V conditions and expressing them as database queries.
4. Analysis, performed by evaluating the V&V conditions as database queries against the data. The reporting of the query results was organized into confirmations and anomaly reports

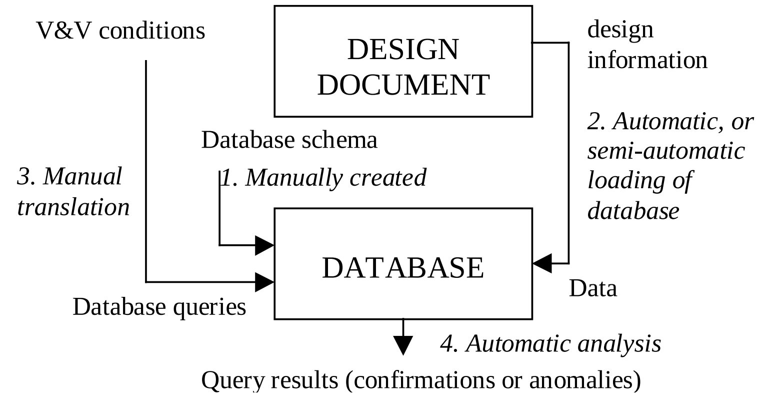


Figure 1 – Architecture of First Pilot Study

The pilot study examined two sets of design documents – interface diagrams (i.e., summaries of incoming and outgoing connections of software modules) and test logs (i.e., traces of behaviors generated in testing of the software components in simulations). Modest verification conditions were rapidly and successfully analyzed in this manner.

2.1. Conclusions drawn from first pilot study

Overall, the pilot study answered its original question affirmatively.

- The database could readily be used to represent existing design information, and populating the database with that information could be automated with little effort.
- Database queries could be used to perform simple analyses. The creation of these queries was a relatively straightforward, albeit manual, task.
- The efficiency of the database was sufficient for the volume of information dealt with in these pilot studies. However, questions remained about the scalability of the approach. In particular, checking properties of very large log files was anticipated to require a more efficient encoding of those properties. A state-machine based approach, e.g., [2] or [5] would perhaps be more appropriate in such circumstances.

For further details see [7].

3. Second pilot study: V&V of an autonomous planner

The need arose to perform V&V of autonomous spacecraft control systems. The rapid analysis approach of the first pilot study was identified as having *potential* application here. A second pilot study was conducted to investigate this potential. This section provides some background on the autonomous spacecraft, and then summarizes the study.

3.1 An Autonomous Spacecraft

NASA’s “New Millennium” series of spacecraft is intended to evaluate promising new technologies and instruments. The first of these, “Deep Space 1” (DS1) [6], was launched in 1998. Increased autonomy is one of several innovative goals that DS-1 will demonstrate [12]. The “Remote Agent” [10, 11] will be the first artificial intelligence-based autonomy architecture to reside in the flight processor of a spacecraft and control it for 6 days without ground intervention. The Remote Agent achieves its high level of autonomy by using an architecture with three key modules:

- an integrated planning and scheduling system that generates sequences of actions (plans) from high-level goals,
- a intelligent executive that carries out those actions and can respond to execution time anomalies, and
- a model-based identification and recovery system that identifies faults and suggests repair strategies.

The planner is a critical component of the autonomy architecture. The command sequences generated by the planner direct navigation, attitude control, power allocation, etc. The entire mission could be jeopardized by an error in a command sequence pertaining to any of these areas. For example, the June 1998 loss of contact with the Solar and Heliospheric Observatory (SOHO) spacecraft is believed to have involved "errors in preprogrammed command sequences" [15] (fortunately, contact has since been re-established).

3.2. Automated Verification of Plans’ Temporal Constraints

The rapid analysis approach of the first pilot study was identified as having *potential* application to V&V of DS-1’s planner. However, the first pilot study had examined traditional design information (interface diagrams and test logs), so there was uncertainty as to whether the same approach would work for the planner’s output (i.e., plans).

A second concern was motivated by the critical resource of planner experts’ time. The first author, who was not a planner expert, had done the V&V research. Development of an automated plan checker would clearly require some investment of time by the planner experts - but how much?

A pilot study to investigate this potential was conducted. It sought to answer two questions:

Could the database-based analysis approach be rapidly applied to automate checking the planner’s generated plans against its temporal constraints?

Could this be done without a large investment of time by planner experts?

We entered into this study with a reasonable expectation of success. The planner has to be able to generate plans; its constraint language is crafted to simultaneously ease the expression of certain constraints, and limit the form of expression to those that it can readily handle. Conversely, the database only has to be able to evaluate queries about a specific set of data, a far easier task than the search-intensive task of planning. The database query language is an extensible, general-purpose language and so should be capable of straightforwardly expressing the planner’s constraints. The relative computational simplicity of checking vs. planning (an instance of Blum's notion of "simple checker" [16]) also suggested that the development of a sufficiently efficient checker would not itself become a large development effort.

Figure 2 shows the architecture of the approach followed in this second pilot study.

As before, it is organized into four main stages:

1. Creation of database schema to represent the plan’s activities. This was confirmed to be a straightforward, manual task.
2. Loading the database with plan activities. This was made a completely automatic step in this pilot study. The amount of effort to do this was small, in part because both planner and database happened to be implemented in the same programming language (Common Lisp). Had there not been this fortuitous coincidence of a common implementation language, it would have been necessary to develop code to parse and translate between linguistic forms. At worst, this would have been a modest standard programming

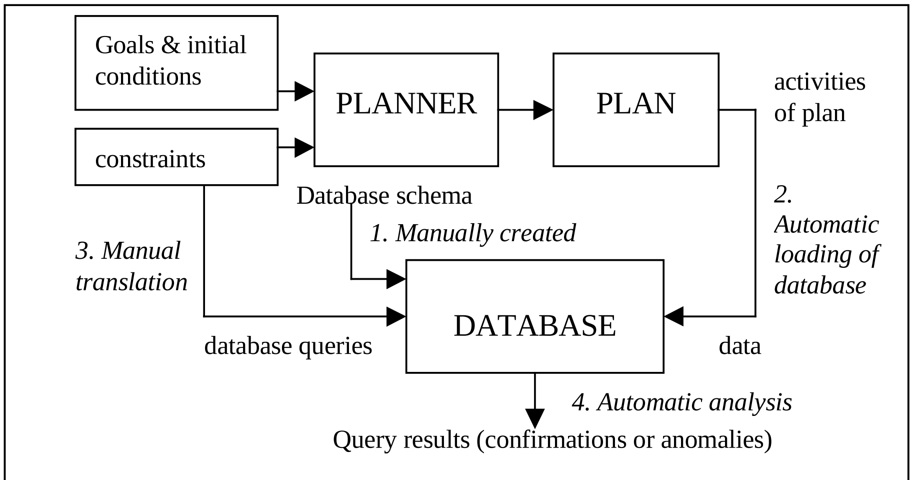


Figure 2 – Architecture of Second Pilot Study

task.

3. Translation of constraints. Representative planner constraints were selected for hand-translation into the equivalent database queries. The study revealed translation to be feasible, although a somewhat detailed process (see Appendix A).
4. Analysis. As before, analysis was automatic, yielding reports of confirmations and anomalies. Importantly, this study confirmed that the database approach scaled sufficiently well to efficiently analyze representative plans. (The study used actual plans produced during test runs of the DS-1 planner.)

3.3. Conclusions drawn from second pilot study

The study answered affirmatively its first question. It demonstrated the feasibility of automating checking of plans. This was recognized to be an onerous task to perform manually, and yet thorough checking of plans dictated that it be done (for more discussion of the rationale, see [8]).

The second question was also answered affirmatively. Interestingly, while the amount of time expended by planner experts on this task remained well below that expended by V&V expert, it was noticeably higher than had been the case for the first pilot study. Generally, we attributed this to the need to delve into more application-specific details, resulting in the need for more coaching of the V&V tool expert by the spacecraft planner experts.

Illustrations and further discussion are presented in Appendix A.

4. Development of analysis tool

The success of the second pilot study led to the next phase – a commitment to develop an analysis tool that would be used during testing of the planner by the planner experts themselves. While this might appear to be just a small extension of the previous phase, there were several important ramifications of this transition from pilot study to actual development:

- **Reliance upon the result:** The pilot shadowed the actual spacecraft development effort, but did not promise to yield results upon which that development effort would rely. Indeed, a valid result of the pilot study could have been that the approach was infeasible. In contrast, this phase committed to the development of a tool that the project would rely upon during testing.
  The positive results of the pilot studies were necessary precursors to this commitment. Additionally, our realization that the analyzer employed an extensible, general-purpose language gave us a justification of why we could extrapolate those positive results to the entire planner constraint language.
- **Developer and end-user different people:** The pilot study tools were developed primarily by the V&V expert, and used by that same person. In contrast, this phase committed to the development of a tool that would be applied by the planner experts with little, if any, involvement of the V&V expert during use.
  This motivated two extensions to the approach demonstrated in the second pilot study: (i) automating the translation from planner constraints into database queries, and (ii) rendering the outputs of the analysis step in terms understandable by the planning experts.
- **End-user agenda:** the DS-1 planner experts constructed an agenda of capabilities they desired of the to-be-developed tool. This featured a prioritized list of capabilities, such that the capabilities to be developed sooner would be the ones they predicted would be of more value to them.
  The preceding pilot studies had helped by providing illustrations of the kinds of analyses that could be accomplished employing this approach. The fact that those illustrations were in terms of DS-1 specific information contributed to their (the planner experts) ability to see its potential. They were thus able to formulate an agenda at this stage, supplanting what was previously the V&V tool expert's *guess* as to what analyses might be interesting and/or valuable.

The architecture of the system developed in this phase is shown in Figure 3. For the remainder of this paper we

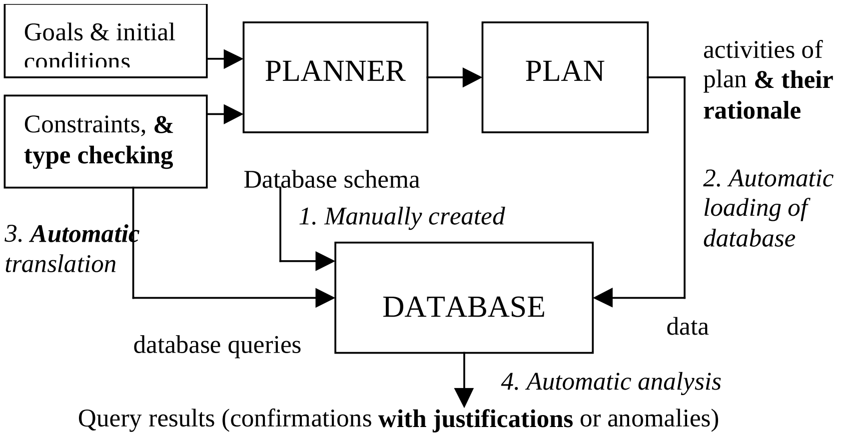


Figure 3 – Planchecker Architecture (changes from 2nd Pilot Study shown in **bold**

will refer to this system as the “planchecker”. It has the same stages as the second pilot study, but with some additional capabilities:

- **Additional analyses:** the planner experts asked for further analyses beyond temporal constraints, notably typechecking of plan elements, and cross-checking of plan activities against their rationale (information on which is included in the generated plans). These required loading additional information from plans into the database, and development of additional database queries.
- **Automatic translation:** there were over 200 temporal planner constraints (counting each lowest-level clause as one constraint). Based on the observations of the second pilot study, we recognized that manual translation of the whole set would be a tedious task. Worse yet, we expected the set of planner constraints to grow and change over time. In keeping with our overall goal of judicious use of automation, it was decided build an automatic translator that would take *any* constraint expressible in the planner language and generate the equivalent database query.
- **Extended output:** the planner experts wanted the query results to report more than simply “OK” when a plan passed the checks. In essence, they wanted a justification for *why* a temporal constraint was satisfied. For example, a constraint that says every SEP-thrusting interval is followed by an SEP-idle interval would be justified by listing, for each SEP-thrusting interval, the specific SEP-idle interval found to satisfy the constraint.
- **Coverage analysis:** the planner experts also wanted to know *which* of the planner constraints had been exercised in the plan. For example, only plans that involved intervals of SEP thrusting would exercise a constraint of the form “every thrusting interval must …”.

4.1. Insights gained from development experience

The development effort did indeed culminate in the planchecker tool (use of which is discussed in the next section). We therefore confirmed the validity of the conclusions drawn from the second pilot study. We also gained some further insights. These fell into two key areas:

- The second pilot study had suggested that the translation from planner constraints to database queries would be straightforward. In practice, automating the translation of the full planner language turned out to be more complex than the pilot study had indicated (see Appendix B for examples). While a procedural approach to programming the planchecker’s translator sufficed to meet the development goals, we concluded that translation warrants further attention. We will return to this in Section 6, Lessons Learned.
- In practice, testers need analysis results with more content and structure than simply “pass” or “fail”. Again, details can be found in Appendix B, and discussion is deferred to Section 6. Lessons Learned.

5. Use of analysis tool

The planchecker was used by the second author (a planning expert) during testing. Interaction with the V&V expert was not required during this phase.

The planchecker was applied to check each plan generated. Its results were accumulated alongside other statistics about the plan generation, e.g., how long it took to generate the plan, how much memory was required to do so. It was easy to apply in “batch mode” to a whole series of plans. It was tolerably efficient, taking on the order of 2 minutes to complete the checking of a typical plan.

Over the course of use, several sets of changes were made to the planner constraints. Re-translating the entire set of constraints, to generate a new instance of the test oracles, easily accommodated these changes. On these occasions the V&V tool expert was on hand. The re-translations went smoothly, with only one instance of the need to step in and make a corrective modification. There

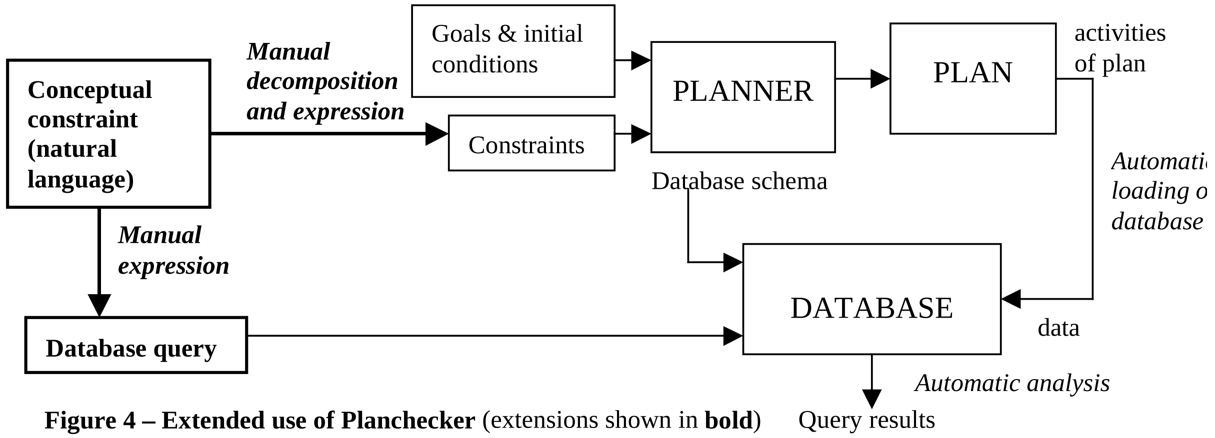


Figure 4 – Extended use of Planchecker (extensions shown in **bold**)

were even changes to the plan format, in response to which the V&V tool expert had to (manually) adjust the corresponding portions of the planchecker system.

The second author (a spacecraft planner expert) extended the planchecker in a particularly interesting manner. On occasion, the writers of planner constraints had found it necessary to manually decompose a fairly obvious constraint that they want the plans to exhibit into a *set* of constraints that the planner would accept, and that in combination would achieve the original constraint. The need to do this stemmed from the limited forms of expression allowed in the planner constraint language. Because the database query language was not so tightly constrained, it was often possible to hand-express the original constraint into a *single* database query. This could then be applied to automatically check plans. Doing so gives increased confidence in the validity of their manual translation of the original constraint into multiple planner constraints. Figure 4 shows the architecture of this extended use of the planchecker.

The implications of this are twofold: (1) a planner expert was able to master the use of the database language and the special-purpose constructs added to represent and reason about plans. Seeing familiar examples (translations of the standard constraints) helped in achieving this level of understanding. (2) the planchecker architecture facilitates such extensions – specifically, automatic loading of plans into the database, and automatic evaluation of database queries, can both be reused. (Of course, the translator from planner constraint language could not be reused, because the original constraints were not expressible in that language.) The net result is extra validation at the cost of very little extra time and effort.

6. Lessons learned

The lessons we draw from this experience are presented next, beginning with those related to general software engineering principles, followed by those specific to V&V.

Software Engineering Lesson 1: Pilot Studies

Our experience re-iterates several well-understood virtues of pilot studies as a precursor to actual development.

Pilot studies provide evidence of feasibility, serve as prototypes and yield examples, which inspire suggestions for extensions, further applications, etc.

In addition, we found it useful to formulate a justification of why the pilot study approach would extend to the full problem. Such a justification nicely complemented the evidence provided by the pilot studies' specific cases.

Software Engineering Lesson 2: "On-Demand" Knowledge Acquisition

When domain experts' time is a critical resource, follow an "on-demand" policy of knowledge acquisition.

At the start of the project the V&V expert lacked a complete and fully documented specification of the task (i.e., plans and the planner language). Furthermore, the domain experts' time was very limited. In response, we followed an "on demand" approach to knowledge acquisition, where the V&V expert would proceed as far as possible before making the next enquiry of the planner experts. This made good use of the planner experts' limited time and availability, since it kept the sum total of their time small, consumed it in small chunks, and could be done asynchronously (e.g., via email exchanges, supplemented by brief telephone calls).

We benefited from the existence of numerous sample inputs (plans and planner constraints). Also, the nature of the task clearly circumscribed the areas that the analysis expert would have to master.

We found it useful to work from an example plan that a planner expert had already vetted as being correct. If the planchecker reported faults with such a plan, the V&V expert would know that most likely there was an error in his own understanding, or his coding of the planchecker itself. Any remaining anomaly that the V&V expert could not resolve would then be a plausible candidate for a genuine plan anomaly, something the plan expert was very interested in!

V&V Lesson 1: Encourage and Use Redundancy and Rationale

V&V can make good use of redundancy and rationale, to increase assurance in the V&V results, and to assist in the development of the V&V technology itself.

Each plan generated by the spacecraft planner contains both a schedule of activities, and a rationale relating those activities to the constraints taken into account in their planning. Checking both of these might appear redundant – surely what really matters is whether or not a plan satisfies all the constraints. Nevertheless, we found this redundancy to be useful in two ways:

1. The planner experts gained additional assurance that their generated plans were correct, in particular, that they generated the "right" results "for the right reasons."
2. The V&V tool expert made use of the redundancy to extend (and debug) his understanding of the task. Every constraint that the planchecker identified as being involved had to be identified in the plan's rationale, thus forcing the planchecker to be complete and correct in its treatment of rationales. Likewise, every constraint mentioned in the rationale had to be seen to be involved by the planchecker, thus forcing the planchecker to be complete and correct in its treatment of constraints. This helps assure that the planchecker is not reporting "false positives" (plans judged as correct which are actually incorrect). [2] describes false positives as more serious than false negatives. He suggests "…a thorough system of document reviews …can mitigate the risk of these

false positives." Our experience indicates that machine-generated rationale can provide a basis for automating some of this review process.

V&V Lesson 2: Database-based Analysis

The use of a database as the underlying analysis engine has practical applications and benefits.

Based on the first of our pilot studies we had made the argument that database-based analysis was suited to "lightweight" V&V [7]. The success of this whole effort strengthens our belief in this position, and highlights some further benefits.

The database approach suggests a natural decomposition of the problem into: translating the V&V conditions into database queries, loading the data into the database, performing the analyses, and generating the reports. This simple architecture nicely separates the key steps. For example, in response to a change in format of plan structures it sufficed to modify the planchecker's database loading portion. Also, this architecture facilitated the planner experts' extended use of the planchecker (i.e., their checking of complex conceptual constraints by manually expressing them as database queries).

The database itself is used as intermediary between analysis and report generation steps. The planchecker places analysis results back into the database, alongside the original data (plans) from which those results are derived. Thus the report generation phase has uniform and simultaneous access to both kinds of data regardless of source, considerably facilitating the report generation task.

V&V Lesson 3: Analysis Results Need Structure

Test oracles should yield results with far more content and structure than simply "passed" or "failed".

During the pilot studies it had sufficed to yield analysis results with trivial structure – they reported either that the object had "passed" the analysis test, or had "failed due to…." (with some simple distinctions among failure cases).

The planchecker development entailed the generation of analysis results and reports with considerably more structure to both the "passed" and "failed" cases. For example, reports that identified *which* constraints had been exercised by a plan, and that distinguished *how* constraints had been satisfied: those that were wholly satisfied by the plan, those that deferred some condition to activities beyond the plan's horizons, etc.

We suspect that there may be general principles by which test oracles can be built to yield such structured analysis results, an area we think is worthy of further attention.

V&V Lesson 4: Translation is the key

Translation between notations is a recurring need, and ideally should be done in such a way as to support understanding, specification and maintenance by domain experts.

The planchecker, and the pilot studies that preceded it, made extensive use of translation between notations. For example, the loading of a plan into the database was a simple translation from plan format into database schema format.

In the pilot studies, it sufficed to perform these translations manually, or to develop procedural-style code to automate the translation. In development of the planchecker, translation from planner constraint language to database query language was also programmed procedurally, but, because of the complexity of this translation, this had some untoward consequences. Notably, the procedural code was hard to understand and maintain.

We believe that for translation of this complexity, a more declarative style would be superior. In one such approach, translation would be expressed as a set of translation rules, executed by a general-purpose rule engine (e.g., POPART [17]). We would hope that such translation rules are readily created, understood and maintained.

A desirable objective is that planner experts, guided by the translations of their planner constraint language, would readily see how to use and write additional translations. Perhaps they could even go on to use the same approach to extend the planner constraint language itself, i.e., to automatically translate the formal expression of a conceptual constraint into the set of simpler constraints that the planner language currently accepts.

We are currently pursuing approaches to development and use of translators. The planchecker's translation will serve as a challenge problem for this effort.

7. Conclusions

Our work follows the trend towards the use of automation for generation of test automation. For example, [14] presents an approach to generating test oracles from specifications. [9] present an industrial application feasibility study on automatically constructing testing software for safety properties. Efficiency (and therefore scalability) of the test oracles themselves is a dominant concern in much of the related work. Commonly, safety properties (typically expressed in some form of temporal logic) are turned into finite state machines whose construction ensures their efficiency of execution (e.g., [5]). For our particular application, the efficiency of the test oracles did not turn out to be a driving concern. Our database-based approach to analysis sufficed. More important to us was the investment of effort that would be required of our domain experts, whose time was in short supply. This led us to automate the generation of test oracles from a domain-specific representation. Thus the domain experts' effort it would take to construct that generator became out dominant concern. Approaches that could reduce this kind of effort include the parameterized tableaus [4], or the algebraicsignature based mappings of [13]. We found, however, the need to yield needed test results with finer distinctions than simply "passed" or "failed." Information about "passed" cases was useful to for test coverage analysis, and for ascertaining that the test had been passed "for the right reasons". Information about "failed" cases was useful to locate the relevant portions of the plan contributing to those failures, and so speed the domain expert in debugging what was going wrong in the planner.

We are not aware of work on automatic generation of test oracles that supports this capability. Based on our practical experience of application of test oracle generation, we see the need for further investigation of this area.

8. Acknowledgements

The research described in this paper was carried out by the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space administration. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not constitute or imply its endorsement by the United States Government or the Jet Propulsion Laboratory, California Institute of Technology.

The authors thank the other members of the DS-1 planner team, Nicola Muscettola and Kanna Rajan, for their help.

9. References

[1] J.F. Allen. Maintaining Knowledge about Temporal Intervals. *Communications of the ACM*, 26(11):832-843, 1983.

[2] J.H. Andrews. Testing using Log File Analysis: Tools, Methods, and Issues. *Proceedings of the 13th IEEE International Conference on Automated Software Engineering* (Honolulu, Hawaii, October 1998), IEEE Computer Society, 157-166.

[3] D. Cohen. Compiling Complex Database Transition Triggers. *Proceedings of the ACM SIGMOD International Conference on the Management of Data* (Portland, Oregon, 1989), ACM Press, 225-234.

[4] L.K. Dillon & Y.S. Ramakrishna. Generating Oracles from Your Favorite Temporal Logic Specifications. *Proceedings 4th ACM SIGSOFT Symposium Foundations of Software Engineering* (San Francisco, October 1996), ACM Press, 106-117.

[5] L. Dillon & Q. Yu. Oracles for checking temporal properties of concurrent systems. *Proceedings 2nd ACM SIGSOFT Symposium Foundations of Software Engineering* (New Orleans, December 1994), ACM Press, 140-153.

[6] http://nmp.jpl.nasa.gov/ds1/

[7] M.S. Feather. Rapid Application of Lightweight Formal Methods for Consistency Analyses. *IEEE Transactions on Software Engineering*, 24(11): 949-959, Nov 1998.

[8]. M.S. Feather & B. Smith. V&V of a Spacecraft's Autonomous Planner through Extended Automation. *Proceedings of the 23rd Annual Software Engineering Workshop* (NASA Goddard, MD, December 1998).

[9] L.J. Jagadeesan, A. Proter, C, Puchol, J.C. Ramming & L.G.Votta. Specification-based Testing of Reactive Software: Tools and Experiments. *Proceedings of the 19th International Conference on Software Engineering* (Boston, MA, May 1997), 525-535.

[10] B. Pell, D.E. Bernard, S.A. Chien, E. Gat, N. Muscettola, P.P. Nayak, M.D. Wagner & B.C. Williams. A Remote Agent Protoype for Spacecraft Autonomy. *Proceedings of the SPIE conference on Optical Science, Engineering and Instrumentation*, 1996.

[11] B. Pell, D.E. Bernard, S.A. Chien, E. Gat, N. Muscettola, P.P. Nayak, M.D. Wagner & B.C. Williams. An Autonomous Spacecraft Agent Prototype. *Proceedings First International Conference on Autonomous Agents*. ACM Press, 1997.

[12] http://nmp.jpl.nasa.gov/ds1/tech/autora.html

[13] A.A. Reyes & D.J. Richardson. Specification-Based Testing of Ada Units with Low Encapsulation. *Proceedings of the 13th IEEE International Conference on Automated Software Engineering* (Honolulu, Hawaii, October 1998), IEEE Computer Society, 22-31.

[14] D.J. Richardson, S.L. Aha & T.). O'Malley. Specification-based Test Oracles for Reactive Systems. *Proceedings of the 14th International Conference on Software Engineering* (Melbourne, Australia, May 1992), 105-118.

[15] *SOHO Mission Interruption Preliminary Status and Background Report – July 15, 1998* http://umbra.nascom.nasa.gov/soho/prelim_and_background_rept.html

[16] H. Wasserman & M. Blum. Software Reliability via Run-Time Result-Checking. JACM 44(6): 826-845, 1997.

[17] D. Wile. Abstract Syntax from Concrete Syntax. *Proceedings of the 19th International Conference on Software Engineering* (Boston, MA, May 1997), 472-480.

Appendix A - Details of the second pilot study

A.1. Example of planner constraint

The following example of one of the simpler plan constraints, as expressed in the planner's special purpose language, will convey a feel for the challenges faced in this pilot study:

```
(Define_Compatibility
   ;; Idle_Segment
   (SINGLE ((SEP_Schedule SEP_Schedule_SV))
         (Idle_Segment))
      :duration_bounds [1 _plus_infinity_]
   :compatibility_spec
   (AND
    ;; Thrust and Idle segments must all
meet--no gaps
    (meets
     (SINGLE
     ((SEP_Schedule SEP_Schedule_SV))
    (Thrust_Segment (?_any_value_
                         ?_any_value_)))))
```

```
    (met_by (SINGLE
      ((SEP_Schedule SEP_Schedule_SV))
      ((Thrust_Segment (?_any_value_
                        ?_any_value_)))))))
```

This illustrates several areas where knowledge held by the planner experts had to be acquired by the V&V expert:

- **Overall application domain knowledge:** "`SEP`" is an acronym for "Solar Electric Propulsion," the innovative engine that provides thrust to DS-1. "`Thrust`" and "`Idle`" are the two main states this engine can be in. Knowledge such as this of the spacecraft domain provided useful intuition to the V&V expert, and this second pilot study warranted a deeper level of understanding than had been necessary for the first pilot study.
- **Problem-specific terminology:** "`SINGLE`" has a connotation specific to DS-1's planner. It introduces a description that matches a single interval. (One alternatives is "`MULTIPLE`," introducing a description that matches a contiguous sequence of intervals).
- **Terminological variants:** The overall definition is of a "compatibility." The V&V expert preferred to think of this as a "constraint," in keeping with the terminology of the database tool. Another example is the "`?_any_value`" term, which serves as a wildcard, indicating any acceptable parameter value may occur in the corresponding parameter position. Again, the V&V expert had the exact same concept, but preferred a different syntax.
- **Confirmation of shared understanding:** there were some areas of shared understanding, but these had to be confirmed, not taken for granted. A trivial example is "`AND`", which in the above is used to indicate that the constraint [compatibility] holds if all of the clauses of this `AND` hold. More interesting are the terms "`meets`" and "`met-by`," which are binary temporal relations between intervals, drawn from the work by Allen [1].

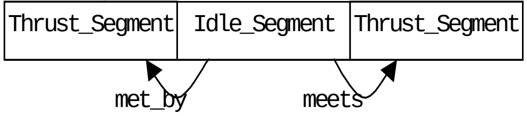
The net result was that the V&V expert required an intensive session of coaching on the meaning of the planner notations (plans and constraint language) at the start of this pilot study, and incremental assistance at various points throughout. Overall this did not amount to an undue consumption of planner experts' time.

A.2. Example of Translation from Planner Constraint to Database Query

Consider the `Idle_Segment` constraint given earlier. Its essential core is the following:

```
  (SINGLE ((SEP_Schedule ...(Idle_Segment))
  :compatibility_spec
  (AND
   (meets (SINGLE ((SEP_Schedule ...
                (Thrust_Segment (?,?)))
   (met_by (SINGLE ((SEP_Schedule ...
                (Thrust_Segment (?,?))))
```

The fragments `(SINGLE ((SEP_Schedule ...` introduce descriptions that are to match to activities of the SEP scheduled in the plan. The first such description is of an `Idle_Segment` activity. For every instance of an activity in the plan matching that description, the constraint requires that the logical condition `(AND ...)` is true. The logical condition is the conjunct of two clauses. The first says that the matching instance `meets` a `Thrust_Segment` activity, i.e., the end-point of the `Idle_Segment` activity exactly coincides with the start point of some `Thrust_Segment` also in the plan. The second says that the matching instance is `met_by` a `Thrust_Segment` activity, i.e., the start point of the former exactly coincides with the end point of the latter Pictorially,



For translation, this is split into two separate constraints, one for each clause of the conjunct. This allows the checking to be conducted separately for each conjunct, so that any anomaly in a plan can be narrowed down as much as possible. The translated form of the first such conjunct looks close to the following (it has been tidied up slightly for presentation purposes):

```
(A (x) (IMPLIES
    (activity-in-plan x Idle_Segment
            SINGLE SEP_Schedule)
    (E (y) (AND (activity-in-plan
       Thrust_Segment SINGLE SEP_Schedule)
                (meets x y)))))
```

`A` and `E` are the database's notations for the logical concepts for-all and exists. `IMPLIES` and `AND` have the standard logical meaning. `activity-in-plan` is a ternary relation (defined for plan checking) that relates an activity name (e.g., `Thrust_Segment`) to a keyword (e.g., `SINGLE`) and schedule (e.g., `SEP_Schedule`). `meets` is a binary relation (again, defined for plan checking) that relates two activities if and only if the end point of the first coincides exactly with the start point of the second.

For this pilot study, some of the more complex planner constraints were also selected for hand-translation. Their additional complexity stemmed from references to activities' parameter values. For example, a constraint that says that every `Max_Thrust_Time` interval whose 1st parameter is `100` must end an `Accumulated_Thrust_Time` interval whose parameters are respectively `100`, `0`, the same value as `Max_Thrust_Time` interval's 2nd parameter, and `WHILE_NOT_THRUSTING`.

Appendix B - Details of the planchecker development

B.1. Automating the translation from planner constraints to database queries

The hallmark of this task was the need to deal with many small (and to the V&V tool expert often surprising) details. Most commonly, these were details of the plan constraint language that the V&V tool expert had not encountered earlier. The representative sample of constraints hand-translated in the second pilot study did not cover the full range of constraint language constructs. The discovery of these came to light when the partially developed planchecker was applied to increasingly more of the entire set of DS-1 constraints, and to increasingly many of the plans that had been generated. They manifested themselves in one of three ways:

- **Error (break) during translation, loading or analysis.** For example, if the constraint translator encountered a variable in a location where it expected a constant. Generally, these were easy to find and understand. A break in the middle of analysis required some simple debugging-like activity to trace back to the underlying discrepancy. Since the database was implemented on top of Common Lisp, the power run-time environment available in the middle of a break made this task fairly simple.

  All these cases resulted in a simple question that the V&V expert would ask of the spacecraft planning experts (e.g., "what does it mean to use a variable name as a range value where normally there is an explicit integer?")
- **False alarms - spurious anomalies detected by analysis**. Often the automated steps would complete, but would report a whole host of (as it turned out, spurious) anomalies. The V&V tool expert generally interpreted a large number of anomalies to be indicative of a flaw in his understanding, rather than a grossly incorrect plan. Indeed, genuine plan anomalies were so few and far between that this was an effective working hypothesis.

  The crucial issue in these cases was finding the underlying cause of the spurious anomalies. The V&V expert would spend time to narrow down the likely cause of a reported anomaly. This culminated in a question to ask of the spacecraft planning experts. For example, suppose this was the first analysis of a plan that exercised default interval range values for one of the temporal relationships. An "anomaly" that could be traced back to one of these defaults would be indicative of a misinterpretation of what the default should be. The V&V expert would then know to ask a specific question about that default value.

  This was a somewhat labor-intensive process for the V&V tool expert. Its benefit was that it ensured that the planner experts' (very limited) time was not squandered unnecessarily.
- **False positives – failure to detect anomalies.** The surprises that were hardest to recognize and understand were those concerning failure to detect anomalies.

  The redundancy of the information in plans was especially useful to help detect these cases. See V&V lesson 1 (in section 5) for discussion of this issue.

  Additionally, the V&V tool expert followed the traditional approach of seeding genuine plans with deliberate errors, and observing whether the analysis caught them.

B.2. Structure analysis results

The need to structure analysis results to be more than simply "pass" / "fail" was a strong theme of the planchecker development. Some examples of the need for this are as follows:

- All the DS-1 planner constraints take the overall form: for every activity-1 that matches description-1 there exists an activity-2 that matches description-2. A constraint of this form is *trivially* satisfied if the plan contains no activities matching description-1. The planchecker separates trivial and non-trivial cases in its reports of constraint satisfaction.
- The DS-1 planner generates plans for a segment of the entire mission (e.g., one week). Thus a plan is bounded within some "horizon"– it has a start and an end. Yet, the constraints may extend across this planning horizon. Such an instance is reported as a special kind of constraint satisfaction in which the plan satisfies the constraint within its horizon, but defers some residual checking for the next plan. The details of all such deferred checks are included within the planchecker's report.
- In an early version of the planner, a few of the constraints referenced information that is not stored in plans. In essence, this external information directed which one of several constraints is to apply. The planchecker's constraint translations handle these circumstances by checking each alternative. If all fail, it is an anomaly. If the plan is found to satisfy one of the alternatives, again, a special kind of constraint satisfaction is reported, which included the deduction of what the external information must be to direct the choice of the satisfied constraint.

The details are domain-specific, but we see a recurring need to make distinctions among classes of "pass" reports, and structure the analysis results accordingly.