Precise and Scalable Static Program Analysis of NASA Flight Software

G. Brat and A. Venet
Kestrel Technology
NASA Ames Research Center, MS 269/2
Moffett Field, CA 94035-1000
650-604-1105 650-604-0775
brat@email.arc.nasa.gov arnaud@email.arc.nasa.gov

Abstract—Recent NASA mission failures (e.g., Mars Polar Lander and Mars Orbiter) illustrate the importance of having an efficient verification and validation process for such systems. One software error, as simple as it may be, can cause the loss of an expensive mission, or lead to budget overruns and crunched schedules. Unfortunately, traditional verification methods cannot guarantee the absence of errors in software systems. Therefore, we have developed the CGS static program analysis tool, which can exhaustively analyze large C programs. CGS analyzes the source code and identifies statements in which arrays are accessed out of bounds, or, pointers are used outside the memory region they should address. This paper gives a high-level description of CGS and its theoretical foundations. It also reports on the use of CGS on real NASA software systems used in Mars missions (from Mars PathFinder to Mars Exploration Rover) and on the International Space Station.

TABLE OF CONTENTS

1. INTRODUCTION 1
2. STATIC ANALYSIS FOR V&V 1
3. C GLOBAL SURVEYOR 3
4. APPLICATIONS OF CGS 5
5. LESSONS LEARNED 7
6. CONCLUSION 8
REFERENCES 9
BIOGRAPHY 9

1. INTRODUCTION

Recent NASA mission failures (e.g., Mars Polar Lander and Mars Orbiter) illustrate the difficulty of building embedded software systems for space exploration and the importance of having an efficient verification and validation (V&V) process for such systems. One software error, as simple as it may be, can cause the loss of an expensive mission ($250 millions at least for a mission to Mars), or lead to budget overruns and crunched schedules. For example, the loss of both spacecrafts in the Mars Surveyor 98 (the lander and the orbiter) mission costs $328 millions to NASA; valuable scientific data could not be obtained either.

Unfortunately, traditional verification methods (such as testing) cannot guarantee the absence of errors in software systems. Therefore, it is important to build verification tools that exhaustively check for as many classes of errors as possible. Static program analysis is a verification technique that identifies faults, or certifies the absence of faults, in software without having to execute the program. Using the formal semantic of the programming language (C in our case), this technique analyses the source code of a program looking for faults of a certain type. We have developed a static program analysis tool, called C Global Surveyor (CGS), which can analyze large C programs for embedded software systems. CGS analyzes the source code of C programs and identifies statements in which arrays are accessed out of bounds, or, pointers are used outside the memory region they should address. CGS does its verification using static analysis techniques based on the theory of Abstract Interpretation. Even though the analysis predicts what will happen at runtime, it is performed at compile time. Therefore, it does not require executing the program and providing input cases. Moreover, CGS analysis is conservative in the sense that it performs all checks necessary to find all errors of the same type (out-of-bound array accesses).

This paper gives a high-level description of the architecture of CGS and the theoretical foundations (Abstract Interpretation) supporting the correctness of the analysis. More importantly, we report on the use of CGS on real NASA software systems. We have analyzed flight software used in Mars missions (from Mars PathFinder to Mars Exploration Rover) as well as software used to control experiments on the International Space Station. The sizes of the software systems we analyzed range from 40 to 600 KLOCs. The analysis times range from 5 minutes to 2 hours on PC platforms running Linux. The analyses did not require any modifications of the original source code.

2. STATIC ANALYSIS FOR V&V

The goal of static program analysis is to assess properties of a program without executing the program. Static analysis has its infancy in compiler optimization. Most compilers do not perform verification beyond type checking and superficial syntactic checks because they focus on getting quick feedback to the code developer. However, they can rely on fairly sophisticated analyses for code optimization because the user is willing to pay a penalty (in terms of compilation time) and obtain optimized code. In other words, it is fine to spend a little more time optimizing the code (which is done once) if it makes the numerous executions run faster. Static program analysis pushes the idea further by using even more sophisticated analyses to find, at compile-time, bugs that can happen at run-time. The rationale is that it is worth spending time analyzing the software if it cuts down on testing. This is what makes static analysis attractive to the verification community.

Several techniques can be used to perform static analysis. Theorem proving, data flow analysis [12], constraint solving [1], and abstract interpretation [4,5] are among the most popular. We could devote an entire article, if not several, to the comparison of these techniques. However, in this paper, we only focus on one technique, abstract interpretation, and show its applicability to real embedded software.

The theory of Abstract Interpretation pioneered by Patrick and Radhia Cousot in the mid 70's provides algorithms for building program analyzers which can detect all runtime errors by exploring the text of the program [4,5]. The program is not executed and no test case is needed. A program analyzer based on Abstract Interpretation is a kind of theorem prover that infers properties about the execution of the program from its text (the source code) and a formal specification of the semantics of the language (which is built in the analyzer). The fundamental result of Abstract Interpretation is that program analyzers obtained by following the formal framework defined by Patrick and Radhia Cousot are guaranteed to cover all possible execution paths.

Runtime errors are errors that cause exceptions at runtime. Typically, in C, either they result in creating core files or they cause data corruption that may cause crashes. In this study we mostly looked for the following runtime errors.

(1) Access to un-initialized variables, i.e., variables that are even though they have not been assigned a value yet.

(2) Access to un-initialized pointers, i.e., pointers that are de-referenced (i.e., attempt to read or write the memory region pointed by the pointer) without having assigned to a memory region.

(3) Out-of-bound array access, e.g., *a[10]* where *a* is an array of size less or equal to 10.

(4) Arithmetic underflow/overflow, e.g., the program does not take into account that the storage of a computed value might take more bits than is allocated for the variable holding the value.

(5) Invalid arithmetic operations, e.g., dividing by zero or taking the square root of a negative number.

(6) Non-terminating loops, e.g., the exit condition of a loop can never be evaluated to false (Note that most embedded programs contain non-terminating loops, such as '*while true do ...;*' by design).

(7) Non-terminating calls, i.e., the control flow of a program never returns from the call to a function (because this function has a non-terminating loop for example).

The price to pay for exhaustive coverage is incompleteness: the analyzer can raise false alarms on some operations that are actually safe. However, if the analyzer deems an operation safe, then this property holds for all possible execution paths. The program analyzer can also detect certain runtime errors which occur every time the execution reaches some point in the program. Therefore, there are basically two complementary uses of a program analyzer:

(1) as a debugger that detects runtime errors statically without executing the program, and

(2) as a preprocessor that reduces the number of potentially dangerous operations that have to be checked by a traditional validation process (code reviewing, test writing, and so on).

The first use is called certification; the goal is to prove the absence of errors of a certain class, thus, alleviating the need for testing for this class of errors. The second use is akin to debugging; basically, the developer tries to flush as many as bugs as he can from the code before it goes to the verification. This requires that the static analyzer achieves a good selectivity - the percentage of operations which are proven to be safe by the program analyzer. Indeed, if 50% of all operations in the program are marked as potentially dangerous by the analyzer, there are no benefits to using such techniques. In the rest of the paper, we refer to these two different types of static analysis as certification and debugging.

The question is: when should one use debugging or certification? On one hand, debugging is fast, but incomplete (since it does not find all the bugs). On the other hand, certification is complete, but it takes quite a long time. So, which one should one use? The answer is both, but not at the same stage of the software development process. Let us put it in terms of the V diagram shown in Figure 1. Black dash arrows indicate the flow of verification while blue arrows indicate what development phase is validated by what validation phase. In general, static analysis applies to the phases in the yellow zone.

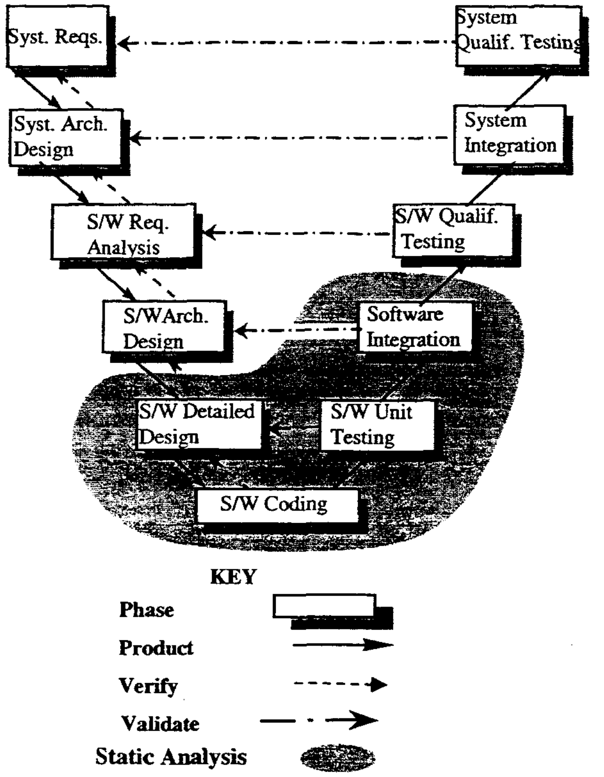


Figure 1. Place of Static Analysis in S/W Lifecycle.

Let us ignore the software detailed design phase (because it requires a special type of static analysis). Debugging is the most useful in the software coding phase. The developer can quickly find a range of bugs and gain some confidence that the software might not crash at run-time. Debugging could also be applied in the unit testing and software integration phase. However, debugging does not give you any coverage information (as opposed to sophisticated testing techniques). Therefore, it cannot measure how well you have tested the units or the system. Now, certification gives you that coverage (actually, it guarantees 100% coverage of all the control and data paths). In general, its application for unit testing requires writing (or generating) drivers for each of the units. This pre-required step might take some time, but the analyses should be fairly fast and precise, especially if the size of each unit is kept to a few thousands of lines of code. Certification can be very useful in the phase of system integration. Since the whole system is put together, the analysis only considers a coherent set of inputs to each function and module. This can potentially yield good precision. The question is: can it really be done for realistic software systems, especially those used in aerospace? This is what this work tries to answer.

3. C Global Surveyor

C Global Surveyor (CGS) is a scalable, precise static analyzer that detects memory errors in C programs. Simply stated, CGS takes the source code of a software system written in C, builds an abstract model of it, and analyzes it to detect errors such as out-of-bound array accesses and de-referencing of null pointers. CGS analysis is exhaustive (all possible execution paths are explored), conservative (all errors, including potential ones, are flagged), and does not require test cases or even executing the program. What differentiate CGS from other static analyzers is its ability to scale to large systems (more than 250 KLOC) and its precision (less than 15% false positives). Scalability is a minimal requirement to be useful to any NASA mission. Precision is critical to user acceptance, since engineers tend to get discouraged by the high number of warnings produced by static analyzers.

Abstract interpretation

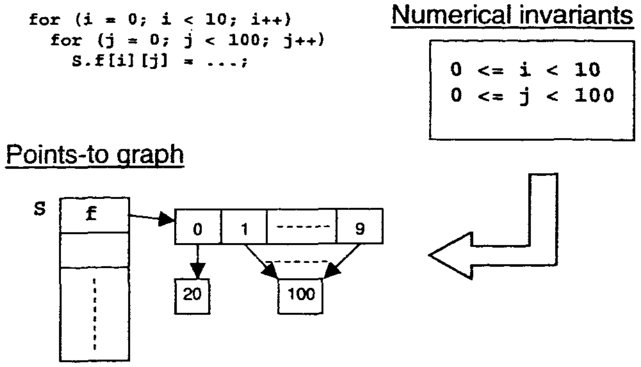
Abstract Interpretation [5,8,9] is a theoretical framework developed by Patrick and Radhia Cousot that gives a methodology for constructing static analyses. The major feature of a static analyzer based on Abstract Interpretation consists of the mathematical guarantee that all properties hereby computed hold for all possible execution paths of the program. The core idea behind this theory is the careful use of the notion of *approximation*: all possible values a variable can take at a certain program point are approximated by a set that can be compactly represented like an interval in the case of scalar variables for example. All possible values of the variable are guaranteed to lie within this set, thus ensuring the soundness of the analysis. However, infeasible value assignments of the variable can be introduced because of the approximation process. This results into a number of *false alarms*, where the analyzer detects a potential problem at some program statement because of spurious information whereas the program is safe in reality. The main point though is that a statement deemed as safe can never cause an error. This is the backbone of abstract-interpretation-based program certification.

The choice of the approximation guides the construction of the static analyzer. Once an approximation scheme has been designed for all objects manipulated by the family of programs considered, we construct a translator from the program source into a system of *semantic equations*. These equations model the flow of information between the statements of the program. It is similar in its structure to the code generation phase of a compiler. This phase is called the *build* in CGS. Any solution of the semantic equations is a sound approximation of all possible values of the program variables. Since we want to limit the number of false alarms caused by the approximation we are interested in the smallest solution of these equations which is guaranteed to exist (see [5,7] for more details). Unfortunately this smallest solution is not always computable or can take too much time to compute. Therefore we have to use heuristics that can lead us to a solution that is "as good as possible" (by using *widening* and *narrowing* operators [5,6,13]) with reasonable execution times. This phase is called the *solve* in CGS. Because of the sub-optimality of the solution computed during this phase, CGS allows the user to iterate the solve in a feedback loop, thus enabling a stepwise refinement of the results. Intuitively, the process of abstract interpretation is very similar to that of designing a system in control theory: a physical system is first modeled using a system of partial differential equations which is not directly solvable in general and for which approximate numerical resolution schemes are employed.

Architecture

Large programs as those developed for the Mars Exploration Program pose a number of challenges in designing an efficient static analyzer. Determining how objects and variables may be connected one to each in memory via pointers is a problem known as pointer analysis. This is a very active research area which has produced several good algorithms [14,10,11] over the years that are able to scale to million lines of code. However these algorithms cannot be directly applied in our case because they abstract away all information about positions in arrays and objects which would cause an unacceptable level of false alarms. CGS computes numerical relationships between the scalar program variables that are used for indexing arrays, controlling loop iterations and performing pointer arithmetic. These numerical relationships are hereafter used to perform an array-sensitive pointer analysis. This is illustrated in Figure 2 where all elements of the array S.f are memory blocks of size 100 except the first one, thus causing a memory error during the execution of the loop nest.

Classical abstract interpretation algorithms which can discover numerical relationships between program variables do not scale to the large programs we were considering. We tackled the time complexity problem in two different directions. First, we improved the scalability of existing algorithms using adaptive variable clustering as described below. Second, we designed a distributed architecture for CGS that enables the distribution of the static analysis algorithms over a cluster of machines.



```
for (i = 0; i < 10; i++)
  for (j = 0; j < 100; j++)
    S.f[i][j] = ...;
```

```
0 <= i < 10
0 <= j < 100
```

Figure 2. Combining numerical invariants and pointer analysis

The second major problem in the design of a scalable static analyzer is the management of the memory. The number of artifacts produced by a static analyzer is tremendous. For example, the semantic equations of the smallest flight software we have analyzed (140 KLOC) cannot even fit all in memory. This requires a smart memory management in which we can dynamically load and unload artifacts. We chose a database-centric architecture in which a relational database plays the role of a persistent network-transparent memory. This also makes the implementation of distributed analysis algorithms simpler since there is no stream of data between two processes on two different hosts, everything being centralized within the database. Moreover we do not have to handle mutual exclusion since this is already part of the database management system.

Innovations

To achieve the main goals of CGS, i.e., be scalable and very precise, we had to go beyond the state-of-the-art in research in static analysis. We now describe the research innovations that had to take place to achieve our goals. The first two innovations work towards improving scalability while the last two target precision. The other two contribute to both scalability and precision.

CGS relies on scalable abstract numerical domains, switching from one to the other in an adaptive manner. We rely on two main abstract domains. First, numerical intervals allow us to keep track of information about integer variables. Second, we can also express numerical constraints between pairs of variables (e.g., $x-y \leq c$) using the difference-bound matrices domain. It allows for example to track constraints between loop indices and variables appearing in the loop body. Unfortunately, this domain does not scale, and therefore, it applies only to small sets of variables. In [2], these sets are computed syntactically. We use an adaptive method to keep the size of those sets small. Future iteration of CGS should include a generalization of this technique that will allow us to apply even more powerful abstract domains (e.g., polyhedra [6]) to even smaller sets of constraints. This means that an improvement in scalability will yield greater precision.

CGS uses distributed algorithms. We are not aware of any static analyzer that can distribute the analysis over several processors; CGS can. This gives CGS a speed advantage since PC machines now come with dual processors and it also reduces our memory requirements (note that all intermediary results are stored in a common database residing on a disk), thus reducing the risk of time-consuming swapping cycles.

CGS mixes flow sensitive and flow insensitive analyses. Other static analyzers allow using one or the other. CGS takes a different approach since it depends on the manners variables are allocated. Thus, top-level variables (i.e., local variables whose addresses are not taken) are handled in a flow-sensitive manner while heap data are treated in a flow-insensitive manner.

CGS performs a mutual incremental refinement of the points-to (which determines memory locations, or addresses) and numerical information (which computes possible offsets). Traditionally, the points-to analysis is done before any analysis (especially before the numerical analysis). In CGS, both analyses feed off each other in an iterative process; each iteration performs an incremental refinement on the precision of the analyses until a global fixed point is reached.

CGS relies on a precise representation of pointer arithmetic in complex data structures such as multi-dimensional arrays. This allows us to compute precise offsets for array elements, even in the case of multidimensional arrays [15].

CGS also performs a points-to analysis that is an extension of Das' algorithm [10]. Das noticed that being precise about the first level (or depth) in graphs of pointers increases drastically the precision of points-to analysis for most C programs. It turns out that for MPS-based software we need to be precise to the second or third level. We therefore implemented a multi-level flow points-to analysis that generalizes Das' idea.

Some of these innovations formed the basis for developing CGS; others were the results of constantly testing our ideas, and their implementation, with real NASA software systems such as the MPF and DS1 flight software.

Results interpretation

By the end of the analysis CGS has assigned a set of possible addresses together with an index range to each memory access operation of the program. The array-bound checking process simply scans these data and checks the indices against the size of the objects being accessed. We use a four-color code (green, orange, red, black) to present the results to the user, If the set of indices accessed during the operation lies within the range of memory cells spanned by the object being dereferenced, the operation is deemed safe and colored in green. If the set of indices being accessed is completely disjoint from the memory area covered by the object, this is a definite memory violation that will occur whenever the operation is performed. The operation is flagged in red. If the set of indices being accessed is empty, this simply means that no execution path ever leads to this operation, in other words this is dead code. We color the operation in black. In all other cases we color the operation in orange. An operation flagged in orange has two possible meanings: it is either an intermittent error that occurs for certain paths of execution but not for others, or it is a false alarm due to spurious values for the index caused by the approximation scheme. The tables stored in the database containing the numerical relationships and the points-to information can then be browsed during an interactive SQL session in order to bring out the causes of a red or orange error. This turned out to be quite useful in practice and this task could be easily performed by newly trained users from Marshall Space Center on software running on the International Space Station.

4. APPLICATIONS OF CGS

The MPF software family

What we call the MPF software family consists of flight software systems that were developed based on the flight software system for the Mars PathFinder mission. The first to "re-use" the MPF flight software was the Deep Space One (DS1) mission. DS1 was not a Mars mission; it was a technology demonstration mission. For example, DS1 flew the Remote Agent experiment, which demonstrated the first use of planning and scheduling technology to control a spacecraft. We analyzed the conventional part of the flight software (i.e., the one directly inherited from MPF). Since the goals of DS1 were different from the goals of MPF, the flight software was slightly different. For example, since DS1 did not land on any planet, the Entry/Descent/Landing module was not used in DS1. The second re-use of MPF was done for the Mars Exploration Rover mission. Actually, the core of the development team for MER was the same as the development team for MPF. So, in some sense, the heritage from MPF was more direct than for DS1. However, the flight software (more than) quadrupled because of increased functionalities and changes in the overall design of the spacecraft. For example, while both the rover and the spacecraft had their own software on MPF, MER went a different route and had the rover controlled the whole spacecraft, even during cruise and landing.

From a static analysis point of view, the three systems are quite similar since they use the same (object-oriented) software architecture as well as some modules (such as the quaternion library). For example, all systems are multi-threaded and they use the threading package of VxWorks. Communication between threads is done using message queues. Even though messages are quite complex (e.g., they contain not only data but also references to callback replies), they are serialized into arrays of integers. Thus, in some cases, the analysis loses information about for example the call flow, or the sizes of matrices passed from one module to the next. This was a major source of imprecision in our analyses. Another important factor is the size of these applications. Overall, the increased complexity of the missions was reflected in the size of each application. As Table 1 shows, the size ranges from 140 KLOCs to 540 KLOCs and the number of threads increased from 23 to more than one hundred in MER.

Table 1. Software complexity for MPF family.

	MPF	DS1	MER
Size (in KLOCs)	140	280	540
#threads	23	40	100+

The results for the MPF family were very good. This is not surprising since we design CGS to work well for this family. In fact, we used the MPF and DS1 software as testbeds during the development of CGS. It made for uneasy debugging since they are quite large software, but it gave us a very realistic "tuning" base. Overall, we obtained about 85% of precision (remember that precision is the percentage of checks that are classified with certainty as correct, incorrect, or unreachable). The average running times were about 1.5 hours for MPF and about 3 hours for DS1. The analysis of MER took much longer (between 1 and 2 days). There are two major reasons for that. First, the sheer size of MER (540 KLOCs) is a big factor. This translated into storing tables (especially for alias information) holding two or three millions artifacts in the database. Loading and populating such big tables take a lot of time. Moreover, their storage in memory (which is limited to 1 GB per process on our machines) needs to be optimized. Unfortunately, it is mostly done at the expense of the analysis time. Overall, we were disappointed in the performance of the database we used (PostgreSQL). Manipulations of large tables were slow even with the use of indices. Actually, we are still actively searching for ways to improve the database response time. The second reason for the slow response times lies in the imprecision of our alias analysis. As mentioned above, callback replies are cast as integer when they are placed in messages. This causes the analysis to lose track of them and therefore to make some conservative approximations about the binding of these replies. This resulted in creating big strongly connected components (SCC) in the call graph (in other words, recursive calls involving lots of functions). Our firt run of CGS on MER showed an SCC of more than 10000 functions (almost all the functions in MER since there are about 11588 functions in our version of MER). By making the analysis ignore some of the low level functions we were able to cut this set to 1000 functions. This is still a huge drain on the response time since the analysis needs to perform a fix-point iteration over every SCC.

Obviously, more work is needed to refine the precision and the response time as flight software systems are getting larger and larger as well as more and more complex. For example, the flight software system for MSL is expected to reach 1 MLOCs. Yet, we are quite happy with the current results, especially when we compare them with the results we obtain using a commercial static analyzer as described in [3]. We can now analyze the whole system without having to cut it in pieces. Moreover, our processing time for the whole system (540 KLOC) is of the same order as the average processing time of the commercial analyzer for a 40 KLOC-size slice.

Shuttle and Space Station Flight Software

The application of CGS to flight software for the shuttle and the International Space Station (ISS) is part of a technology infusion effort. Our goal is to teach NASA developers to use CGS and, from then on, have them use CGS on a regular basis in their projects. In this particular case, three developers from the Marshall Flight Space Center (MSFC) came to NASA Ames, got train in using CGS, and used CGS on flight software systems they had developed at MSFC. Overall, we analyzed five modules.

(1) The Application Processor (AP) module is part of the flight software for the Advanced Video Guidance Sensor (AVGS), which flew as experiments on two Space Shuttle missions and will be the primary sensor for the close-proximity operation in the DART mission. The DART mission seeks to advance the state of the art in safe and reliable autonomous rendezvous capabilities at NASA. The AP module represents about 12 KLOCs of C code.

(2) The IO Processor (IOP) module is also part of the AVGS. It represents 7 KLOCs of C code.

(3) The goal of the Materials Science Research Rack (MSRR) aboard the ISS is to offer capabilities to facilitate a wide range of materials science investigations. For example, the facility will provide the common subsystems and interfaces required for the operation of experiment hardware, accommodate telescience capabilities, and provide the capability for simultaneous on-orbit processing. This application consists of 55 KLOCs of C code.

(4) The Urine Processor Assembly (UPA) is part of the life support in the ISS. The UPA controller consists of 47 KLOCs of C code.

(5) Finally, the last module is the boot loader (BOOTLDR) for the shuttle engine controller. It consis of 7 KLOC of C code. The MSFC development team is also in the process of using static analyzers (including CGS) to analyze the whole controller. However, we do not have results for this experiment.

The results for these modules were neither good nor bad. First, the response times of these analyses are very satisfying. Each analysis was only a matter of minutes on laptop (i.e., machines that are slower and have less memory than the desktops we use for the analysis of the MPF family). Second, the precision was quite good (around 85%), but it revealed some flaws in CGS. For example, structures with bitfields were not treated properly. Moreover, we had problems with pointers to physical hardware devices. The analysis cannot find any size information and it therefore assumes that the size is zero. This deficiency points out the need for user information. We are in the process of implementing an interface that a user could use to give such information. This experiment was also a good opportunity to get feedback from CGS users who are not part of the development team. It gave us some usefully usability data. For example, it is clear that CGS needs to provide type information when the results are scanned by the tool user. It was very cumbersome for users to track the type information across code and header files. We are therefore implementing a function that dumps type information in the database, thereby making it directly available to a user.

Space Station Biological Experiment Software

This experiment was the opportunity for us to try CGS on a different type of software. The Habitat Holding Rack (HHR) software is not a flight software system in the sense that it is not controlling a spacecraft or a rover; it controls biological experiments done on the International Space Station. The HHR is the central part of the biological and scientific experiments to be conducted on-orbit and on the ground for the UF-3 and future missions. The Biological Research Project Rack Interface Controller (B-RIC) and Centrifuge Rack Interface Controller (C-RIC) are the command and control components of the HHR. The C-RIC being unavailable at the time, CGS was applied only to the B-RIC. The B-RIC formats telemetry data received from Payloads for download to the ground, and creates HHR and Payload Health and Status (H&S) data for transmission to the ISS.

The B-RIC software is about 50 KLOCs of C code. The software is divided into five modules that were analyzed separately:

(1) Video Digitalization Compression Card (VDCC). The software module for the VDCC provides the main entry points for all modules running on the Card. The main module initializes message queues, spawns all tasks, and monitors H&S on the Card. It contains 4 KLOCs.

(2) High Rate Link Card (HRLC). The HRLC software module is the main entry point for all other modules running on the Card. Like the VDCC, it initializes message queues, spawns tasks, and monitors H&S. It contains 16 KLOCs.

(3) Serial Card 1553 (SC1553). The software module on this Card is the 1553 communications link to the HHR. It contains 9 KLOCs.

(4) Serial Card (SERC). The SERC software module is the serial communications link to the HHR. It contains a bit less than 1 KLOCs.

(5) Main Controller Card (MCC). This software module is the main controller for the HHR. It contains 19 KLOCs.

The results of the analyses are quite surprising for us and show that we need to adapt CGS to this type of software. For example, the precision is 30%, which is quite disappointing. Moreover, about 35% of the checks classified as certain errors are not errors. After further (manual) investigation, we found that this is due to hardware pointers. The isolation of the source code verification with CGS without attachments to external devices and/or interfaces shows up as repeated errors for all instances where there is no connectivity. For example, in the SERC Module, the Payload Manager Function calls the Payload Table. Because that Function depends on an external link for execution, the CGS Tool tagged all instances of calls as Red errors. Anyway, the analysis times are also quite disappointing given the sizes of the modules being analyzed. In general, the analyses took from 30 minutes to 2 hours, except for the VDCC modules which took 14 hours. We are still in the process of analyzing these modules to find out what causes such long analysis times. Given that these analyses were performed on a laptop with poor (memory and processing speed) performances, our current guess is that CGS spent most of its processing time doing garbage collection. This assumption needs to be verified.

5. LESSONS LEARNED

The first lesson is that scaling up to large programs requires a fine-grained control of the dynamic allocation of data in the analyzer. In our case, the use of a garbage collector frees us from having to manage all de-allocations, but it forces us to be smart in our use of memory allocation. Indeed, the garbage collector used in CGS has a limit of 1 GB, which cannot be changed. Therefore, we need to ensure that we are not allocating more than 1 GB of memory, knowing that the garbage collector might allocate larger chunk of memory than is needed by the data we manipulate. Of course, the size of the memory blocks allocated by the garbage collector can be set to (almost) any arbitrary size. However, reducing the size of the blocks also triggers more frequent calls to the garbage collector, thus impacting the response time of the analysis.

The second lesson concerns the use of a database to manage permanent artifacts. In our original mindset, the SQL database operations could be used to efficiently compute functions on the artifacts (e.g., alias tables, call table, and so on) in the database, rather than having to pull the data out of the database, compute the function, and dump the results in the database again. This turned out to be an unrealistic expectation. Database operations are too slow for that. So, the biggest gain in using a database is that it can automatically handle distributed requests. Finally, it might be possible to optimize the database accesses by breaking large tables into hierarchy of small tables. However, this is possible only if the keys used to access data in the tables are consistent throughout the analysis.

The third lesson is that distributing the analysis does not always pay off, unless you can run the analysis on a truly parallel machine. Indeed, in many phases of the analysis the network access times (for loading or storing analysis artifacts or synchronizing with other processes) outweigh the processing time required by the analysis. Therefore, unless processes can communicate without going through a network, the analysis might be slowed down. Typically, we observed that we are not gaining anything in terms of time when we use more than four processors (which represents two machines since each of our machines have dual processors). Finally, using PVM to distribute the analysis really hampered the debugging of CGS. It was quite hard to pinpoint crashes because most analysis processes died without giving any information back to the master process.

6. CONCLUSION

In this paper, we have given a short introduction to C Global Surveyor (CGS), a static analyzer based on Abstract Interpretation. CGS can find array-out-of-bound and null pointer de-reference errors in embedded C programs. We also have reported on the use of CGS on real NASA software systems ranging from the flight control software for three JPL missions (MPF, DS1, and MER) to software controlling experiments on the International Space Station. We observed that CGS scales without producing many false positives to the large JPL applications (over 500 KLOCs). We expected this result since we design our analysis algorithms to work well with the software following the MPF family. Similarly, the low precision and response times of CGS in the analysis of the Habitat Holding Rack software are not surprising; we did not specialize CGS for this type of software. Anyway, we conjecture that the slow response times may be due to running the analysis on under-performing hardware (laptop with low processing speed and little memory capacity). However, as of now, we can not explain what coding practices caused the precision problems. Still we are confident that we can tune CGS to work well for all these applications, and therefore, be applicable to lots of NASA missions.

REFERENCES

[1] A. Aiken and M. Fähndrich, "Program Analysis using Mixed Term and Set Constraints". In *Proceedings of the 4th International Static Analysis Symposium* (SAS'97), 1997.

[2] Blanchet, B. *et al.* Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software. LNCS 2566, pp. 85-108, 2003.

[3] Brat, G., and, Klemm, R. Static Analysis of the Mars Exploration Rover flight software. In Proceedings of the 1st International Space Mission Challenge for Information Technology, pp. 321-326. Pasadena, California, 2003.

[4] P. Cousot and R. Cousot, "Static Determination of Dynamic Properties of Programs". In *Proceedings of 2nd International Symposium on Programming*, pages 106-130, 1976.

[5] P. Cousot and R. Cousot, "Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints". In *Proceedings of 4th Symposium on Principles of Programming Languages*, pages 238-353, 1977.

[6] P. Cousot and N. Halbwachs, "Automatic discovery of linear restraints among variables of a program", *Fifth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 84—97, 1978. ACM Press.

[7] P. Cousot and R. Cousot. Constructive versions of Tarski's fixed point theorems. *Pacific Journal of athematics*, Vol. 82, No. 1, 43—57, 1979.

[8] P.Cousot, "Semantic foundations of program analysis", *Program Flow Analysis: Theory and Applications*, Ch. 10, 303—342, Prentice-Hall, 1981.

[9] P. Cousot and R. Cousot, "Abstract interpretation frameworks", *Journal of Logic and Computation*, 2(4):511—547, 1992.

[10] M. Das, "Unification-Based Pointer Analysis with Directional Assignments", *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2000, ACM Press.

[11] N. Heintze and O. Tardieu, "Ultra-fast Aliasing Analysis using CLA: A Million Lines of C Code in a Second", *International Conference on Programming Language Design and Implementation, PLDI'01*, 254-263, ACM Press, 2001.

[12] W. Landi, "Interprocedural Aliasing in the Presence of Pointers", *Ph.D. thesis*, Rutgers University, 1992.

[13] A. Mine, "A New Numerical Abstract Domain Based on Difference-Bound Matrices", *Programs As Data Objects II*, 155—172, Lecture Notes in Computer Science 2053, Springer, 2001.

[14] B. Steensgaard, "Points-to Analysis by Type Inference of Programs with Structures and Unions", *International Conference on Compiler Construction*, number 1060 in Lecture Notes in Computer Science, 136-150, Springer-Verlag, 1996.

[15] A. Venet, G. Brat, "Precise and Efficient Static Array Bound Checking for Large Embedded C Programs," Proceedings of PLDI 2004, Washington, D.C., June 2004.

BIOGRAPHY

***Dr. Brat** received his M.Sc. and Ph.D. in Electrical & Computer Engineering in 1998 (The University of Texas at Austin, USA). His thesis defined a (max,+) algebra that can be used to model and evaluate non-stationary, periodic timed discrete event systems. Since then, he has worked on the application of static analysis to software verification. From 1997 to June 1999, he worked at MCC where he was the technical lead on a project developing static analysis tools for software verification. In June 1999, he joined the Automated Software Engineering group at the NASA Ames Research Center and focused on the application of static analysis to the verification of large software systems. For example, he developed and applied static analysis tools based on abstract interpretation to the verification of software for the Mars PathFinder, Deep Space One, and Mars Exploration Rover missions at JPL, various Space Station controllers at MSFC, and the Space Station Biological Research Project at the NASA Ames Research Center.*

Dr. Venet obtained a PhD in Abstract Interpretation in 1998 from Ecole Polytechnique (France) under the supervision of Radhia Cousot. He has worked as an associate researcher in the group of Patrick Cousot at Ecole Normale Supérieure (France). Dr. Venet architected and implemented industrial static analyzers for real-size critical software systems at PolySpace Technologies. He also worked on automated test generation and code compression for JavaCard applications at Trusted Logic. Dr. Venet has an extensive theoretical and industrial experience in static analysis. He is currently a Research Scientist at NASA Ames and is working on specialized static analyzers for verifying large mission-critical software systems developed in NASA. Dr. Venet coauthored three patents on industrial applications of static analysis.