Software Construction and Analysis Tools for Future Space Missions

Michael R. Lowry

Computational Sciences Division
NASA Ames Research Center
Moffett Field, CA 94303 USA
mlowry@mail.arc.nasa.gov

Abstract. NASA and its international partners will increasingly depend on software-based systems to implement advanced functions for future space missions, such as Martian rovers that autonomously navigate long distances exploring geographic features formed by surface water early in the planet's history. The software-based functions for these missions will need to be robust and highly reliable, raising significant challenges in the context of recent Mars mission failures attributed to software faults. After reviewing these challenges, this paper describes tools that have been developed at NASA Ames that could contribute to meeting these challenges: 1) Program synthesis tools based on automated inference that generate documentation for manual review and annotations for automated certification. 2) Model-checking tools for concurrent object-oriented software that achieve scalability through synergy with program abstraction and static analysis tools.

This paper consists of five sections. The first section describes advanced capabilities needed by NASA for future missions that are expected to be implemented in software. The second section describes the risk factors associated with complex software in aerospace missions. To make these risk factors concrete, some of the recent software-related mission failures are summarized. There is a considerable gap between current technology for addressing the risk factors associated with complex software and the future needs of NASA. The third section develops a model of this gap, and suggests approaches to close this gap through software tool development. The fourth section summarizes research at NASA Ames towards program synthesis tools that generate certifiable code. The fifth section summarizes research at NASA Ames towards software model-checking tools.

1. Software: Enabling Technology for Future NASA Missions

NASA's strategic plan envisions ambitious missions in the next forty years that will project a major human presence into space. Missions being studied and planned include sample returns from comets, asteroids, and planets; detection of Earth-like planets around other stars; the search for the existence of life outside the Earth, intensive study of Earth ecosystems, and the human exploration of Mars. A major enabling factor for these missions is expected to be advanced software and computing systems. This section describes some of the requirements for these mission capabilities.

Autonomous Spacecraft and Rovers. NASA's mission of deep space exploration has provided the requirement for one of the most stressing applications facing the computer science research community — that of designing, building, and operating progressively more capable autonomous spacecraft, rovers, airplanes, and perhaps even submarines. NASA is planning to fill space with robotic craft to explore the universe beyond in ways never before possible. These surrogate explorers need to be adaptable and self-reliant in harsh and unpredictable environments. Uncertainty about hazardous terrain and the great distances from Earth will require that the rovers be able to navigate and maneuver autonomously over a wide variety of surfaces to independently perform science tasks. Robotic vehicles will need to become progressively smarter and independent as they continue to explore Mars and beyond.

In essence, robust autonomy software needs to be highly responsive to the environment of the robotic vehicle, without the constant intervention and guidance from Earth-based human controllers. In the case of Martian rovers, in the past Earth controllers would up-link commands each Martian day for limited maneuvers (e.g., roll ten meters forward northeast), which would be executed blindly by the rover. In the future, the commands will be for much more extensive maneuvers (e.g., navigate a kilometer towards a rock formation that is beyond the horizon) that require complex navigation skills to be executed autonomously by the rover, with constant adaptation to terrain and other factors. Such autonomy software, running in conjunction with an unknown environment, will have orders of magnitude more possible execution paths and behaviors than today's software.

In addition to autonomy for commanding and self- diagnosis, there is an increasing need for an autonomous or semi-autonomous on-board science capability. Deep space probes and rovers send data back to Earth at a very slow rate, limiting the ability of the space science community to fully exploit the presence of our machines on distant planets. There is a strong need for spacecraft to have the capacity to do some science processing on-board in an autonomous or semi-autonomous fashion.

Human Exploration of Space. A human mission to Mars will be qualitatively more complex than the Apollo missions to the moon. The orbital dynamics of the Mars-Earth combination means that low-energy (and hence reasonable cost) Mars missions will last two orders of magnitude longer than the Moon missions of the sixties and seventies - specifically, on the order of five hundred days. To achieve science returns commensurate with the cost of a human Mars mission, the scientist-astronauts will

need to be freed from the usual role of handyman and lab technician. They will need to have robotic assistants that support both the scientific aspects of the mission and also maintain the equipment and habitat. A particularly interesting issue that arises is that as spacecraft systems become increasingly capable of independent initiative, then the problem of how the human crew and the autonomous systems will interact in these mixed-initiative environments becomes of central importance. The emerging area of Human-Centered Computing represents a significant shift in thinking about information technology in general, and about smart machines in particular. It embodies a systems view in which the interplay between human thought and action and technological systems are understood as inextricably linked and equally important aspects of analysis, design, and evaluation.

Developing and verifying software for mixed-initiative systems is very challenging, perhaps more so than for completely autonomous software. In contrast to the current human command/software executes blindly paradigm, mixed-initiative software has far more potential execution paths that depend on a continuous stream of human inputs. In this paradigm, the human becomes a complex aspect of the environment in which the software is executing, much more complex than the terrain encountered by a Martian rover. Furthermore, from the human viewpoint, mixed-initiative software needs to be understandable and predictable to the humans interacting with it. Today's methods for developing and verifying high-assurance mixed initiative software are woefully inadequate. For example, aviation autopilot and flight-management systems behave in ways that are often bewildering and unpredictable to human pilots. Even though they decrease the manual workload of human pilots, they increase the cognitive workload. *Automation surprises* have been implicated in a number of aviation fatalities. For a mixed human/robotic mission to Mars, the robotic assistants need to be both smart and well-behaved.

2. Aerospace Software Risk Factors

While advances in software technology could enable future mission capabilities at substantially reduced operational cost, there are concerns with being able to design and implement such complex software systems in a reliable and cost-effective matter. Traditional space missions even without advanced software technology are already inherently risky. Charles Perrow's book [1] identifies two risk dimensions for high-risk technologies: interactions and coupling. Complex interactions are those of unfamiliar or unexpected sequences, and are not immediately comprehensible. Systems that are tightly coupled have multiple time-dependent processes that cannot be delayed or extended. Perrow identifies space missions as having both characteristics; hence space missions are in the riskiest category.

The risks that software errors pose to space missions are considerable. Peter Neumann's book [2] catalogues computer-related problems that have occurred in both manned and unmanned space missions. Given the risks already inherent with today's software technology, flight project managers are understandably reluctant to risk a science mission on new unproved information technologies, even if they promise cost savings or enhanced mission capabilities. This creates a hurdle in deploying new

technologies, since it is difficult to get them incorporated on their first flight for flight qualification. NASA is addressing this hurdle through flight qualification programs for new technology such as New Millennium. However, flight project managers also need to be convinced that any information technology can be verified and validated in the specific context of their mission. This poses a special challenge to advanced software technology, since traditional testing approaches to V&V do not scale by themselves.

This section next reviews several software errors that have had significant impact on recent space missions, in order to draw historical lessons on the difference between software failures and hardware failures.

Ariane 501. The first launch of Ariane 5 - Flight 501 - ended in a disaster that was caused by a chain of events originating in the inappropriate reuse of a component in Ariane 4's inertial reference frame software, and the lack of sufficient documentation describing the operating constraints of the software. Approximately 40 seconds after launch initiation, an error occurred when an unprotected conversion from a 64-bit floating point to a 16-bit signed integer value overflowed. This error occurred both in the active and backup system. The overflow of the value, related to horizontal velocity, was due to the much greater horizontal velocity of the Ariane 5 trajectory as compared to the Ariane 4 trajectory. This error was interpreted as flight data and led to swiveling to the extreme position of the nozzles, and shortly thereafter to self-destruction.

The full configuration of the flight control system was not analyzed or tested adequately during the Ariane 5 development program. The horizontal velocity value was actually critical only prior to launch, and hence the software was not considered flight critical after the rocket left the launch pad. However, in the case of a launch delayed near time zero, it could take a significant period for the measurements and calculations to converge if they needed to be restarted. To avoid the potential situation where a delayed launch was further delayed due to the need to recompute this value, the calculation of this value continued into the early stages of flight.

Like many accidents, what is of interest is not the particular chain of events but rather the failure to prevent this accident at the many levels the chain could have been intercepted: 1) The development organization did not perform adequate V&V. 2) Software reuse is often seen as a means of cutting costs and ensuring safety because the software has already been 'proven'. However, software which works adequately in one context can fail in another context. 3) As stated in the accident review report [3], there was a 'culture within the Ariane programme of only addressing random hardware failures', and thus duplicate back-up systems were seen as adequate failure-handling mechanisms. Software failures are due to design errors, hence failure of an active system is highly correlated with failure of a duplicate back-up system. 4) Real-time performance concerns, particularly for slower flight-qualified computers, can lead to removal of software protection mechanisms that are known to work; in this case the protection for the floating point conversion.

The board of inquiry concluded that: "software is an expression of a highly detailed design and does not fail in the same sense as a mechanical system. Software is flexible and expressive and thus encourages highly demanding requirements, which in turn lead to complex implementations which are difficult to access." The fact that

this software worked without error on Ariane 4, and was not critical after the rocket left the launch pad, contributed to overlooking this problem.

Mars Pathfinder. Today's aerospace software is increasingly complex, with many processes active concurrently. The subtle interactions of concurrent software are particularly difficult to debug, and even extensive testing can fail to expose subtle timing bugs that arise later during the mission. In the July 1997 Mars Pathfinder mission, an anomaly was manifested by infrequent, mysterious, unexplained system resets experienced by the Rover, which caused loss of science data. The problem was ultimately determined to be a priority inversion bug in simultaneously executing processes. Specifically, an interrupt to wake up the communications process could occur while the high priority bus management process was waiting for the low priority meteorological process to complete. The communication process then blocked the high priority bus management process from running for a duration exceeding the period for a watchdog timer, leading to a system reset. It was judged after-the-fact that this anomaly would be impossible to detect with black box testing. It is noteworthy that a decision had been made not to perform the proper priority inheritance algorithm in the high-priority bus management process - because it executed frequently and was time critical, and hence the engineer wanted to optimize performance. It is in such situations where correctness is particularly essential, even at the cost of additional cycles.

Mars Climate Orbiter and Mars Polar Lander. In 1998 NASA launched two Mars missions. Unfortunately, both were lost, for software-related reasons. The Mars Climate Orbiter was lost due to a navigation problem following an error in physical units, most likely resulting in the spacecraft burning up in the Martian atmosphere rather than inserting itself into an orbit around Mars. An onboard calculation measured engine thrust in foot-pounds, as specified by the engine manufacturer. This thrust was interpreted by another program on the ground in Newton-meters, as specified by the requirements document. Similar to Ariane 501, the onboard software was not given sufficient scrutiny, in part because on a previous mission the particular onboard calculations were for informational purposes only. It was not appreciated that on this mission the calculations had become critical inputs to the navigation process. The ground-based navigation team was overloaded, and an unfortunate alignment of geometry hid the accumulating navigation error until it was too late.

The Mars Polar Lander was most probably lost due to premature shutdown of the descent engine, following an unanticipated premature signal from the touchdown sensors. The spacecraft has three different sequential control modes leading up to landing on the Martian surface: entry, descent and landing. The entry phase is driven by timing: rockets firings and other actions are performed at specific time intervals to get the spacecraft into the atmosphere. The descent phase is driven by a radar altimeter: the spacecraft descends under parachute and rocket control. At thirty meters above the surface the altimeter is no longer reliable, so the spacecraft transitions to the landing phase, in which the spacecraft awaits the jolt of the ground on one of its three legs; that jolt sets off a sensor which signals the engines to turn off. Unfortunately, the spacecraft designers did not realize that the legs bounce when they are unfolded at an altitude of 1.5km, and this jolt can set off the touchdown sensors which latch a

software variable. When the spacecraft enters the landing phase at 30 m, and the software starts polling the flag, it will find it already set, and shut off the engines at that point. The resulting fall would be enough to fatally damage the spacecraft.

Lessons from Software Failures during Space Missions.

1) Software failures are latent design errors, and hence are very different from hardware failures. Strategies for mitigating hardware failures, such as duplicative redundancy, are unlikely to work for software.
2) The complexity of aerospace software today precludes anything approaching 'complete' testing coverage of a software system. Especially difficult to test are the subtle interactions between multiple processes and different subsystems.
3) Performance optimizations resulting in removal of mechanisms for runtime protection from software faults (e.g., removal of Ariane 5 arithmetic overflow handler for horizontal velocity variable), even when done very carefully, have often led to failures when the fault arises in unanticipated ways.
4) Reuse of 'qualified' software components in slightly different contexts is not necessarily safe. The safe performance of mechanical components can be predicted based on a well-defined envelope encompassing the parameters in which the component successfully operated in previous space missions. Software components do not behave linearly, nor even as a convex function, so the notion of a safe operating envelope is fundamentally mistaken.

Although the missions beyond the next ten years are still conceptual, plans for the next ten years are reasonably well defined. Sometime in the next decade, most likely 2009, NASA plans to launch a robot mission that will capture a sample of Martian soil, rocks, and atmosphere and return it to Earth. The software for this mission could be 100 times more complex than for the Mars Climate Orbiter. The software for missions beyond this 2009 Mars sample return, requiring the capabilities described in the first section of this paper, will be even more complex. The next section of this paper presents a framework for assessing the likelihood of success for these missions if current trends continue, and the potential for software construction and analysis tools to reverse these trends.

3. A Model for Software Reliability versus Software Complexity

The aerospace industry, like most other industries, is seeing an increasing importance in the role played by software: the amount of software in a mission is steadily increasing over time. This has delivered substantial benefits in mission capabilities. Software is also comparatively easy to change to adapt to changing requirements, and software can even be changed after launch, making it an especially versatile means of achieving mission goals.

The following table provides historical data from a small number of space missions, and gives flight software in thousands of lines of source code. Note that while Cassini (a mission to Saturn that will be in orbit around Saturn in 2004) and Mars Pathfinder launched in the same year, development of Cassini started many years earlier. The data clearly indicates an exponential growth over time in the size of flight software. This exponential growth is consistent with other sectors of aerospace including civilian aviation and military aerospace. In a subsequent graph we will use a log scale for thousands of line of source code versus a log scale for expected number of mission-critical software errors to extrapolate a model for expected software reliability, and the potential impact of various kinds of tools.

Mission	Launch Year	Thousands SLOC
Voyager	1977	3
Galileo	1989	8
Cassini	1997	32
Mars Path Finder	1997	160
Shuttle	2000	430
ISS	2000	1700

Although qualitative data on software reliability, or lack thereof, is abundant, empirical quantitative data is difficult to find. Our graph will take advantage of 'iso-level' tradeoffs between reliability, cost, and schedule. Fortunately, the empirical data on software development cost and schedule has been well analyzed, providing a basis for extrapolating reliability from cost and schedule. At any stage of maturity of software engineering tools and process the multiple criteria of reliability, cost, and schedule can be traded off against each other (within limits). For example a software manager might choose to compress development schedule by increasing overall man-power; the empirical data indicates that this increases total man-years and hence cost. As another example a manager might limit the number of design and code reviews and incur greater risk of overlooking a mission-critical software error.

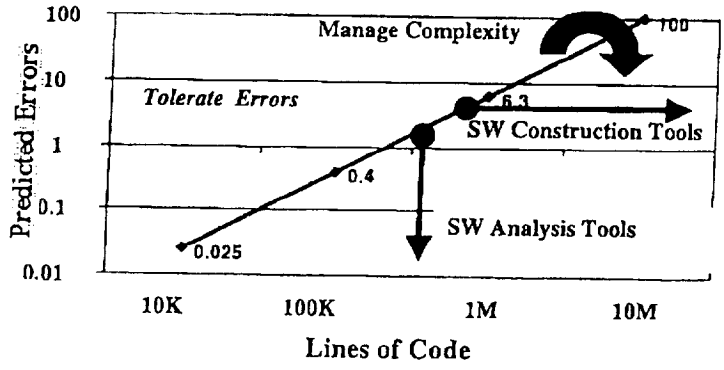
The empirical data on software development cost and schedule as it relates to increasing size and complexity has been extensively studied by Barry Boehm, who has developed mathematical models of cost and schedule drivers that have been statistically validated and calibrated. The models indicate a super-linear growth in cost and schedule with the increasing size of software, hence we should expect an accelerated exponential growth in cost and schedule for mission software in future years without changes in technology and methods. In Boehm's model [4], a primary factor contributing to this super-linear growth is the cost and time to fix unintended non-local interactions: that is, unintended interactions between separate software components and unintended interactions between software and systems.

Both the quantitative cost/schedule data and the qualitative record on software-induced failures are readily understandable: as the size of software systems increase (and the number of elements which interact increases proportionally) the number of potential interactions between elements grows quadratically. Tracking down these interactions is complex and difficult, fixing bad interactions without introducing new errors takes time and money, and the consequences of not fixing unintended interactions can be fatal. We thus extrapolate Barry Boehm's schedule/cost model to

an analogous model for software reliability. The model is based on proportional factors of expected interactions between components as software size increases. If every one of S components interacted with every other one there would be S^2 interactions. Fortunately, the interactions are sparser; the best calibration over many projects gives an exponent of 1.2 as indicated by growth in cost and schedule. The data also indicates that improvements in software process not only reduce the total number of errors but also the growth in errors as software size increases. For software projects with high levels of process maturity, the exponent is 1.1. This makes sense: better engineering management gives a handle on unintended interactions through better communication and coordination across the development organization, as well as better documentation.

In the figure below we show the number of predicted mission-critical errors versus size of mission software (*LOC* - lines of source code), on a log-log scale. We assume that the number of errors is proportional to $(S/M)^N$, where S/M is the number of components (modules), computed as the number of source lines of code divided by the lines per module. For the baseline model, we take the number of lines of code per module, M, to be 100. For this baseline model the exponent N is assumed to be 1.2. The model is calibrated with an assumption of a 40% probability of a critical software error at 100K SLOC, based on recent deep space missions. (More specifically, the vertical axis is interpreted as the mean number of expected critical software errors.) This is a conservative estimate based on recent missions including Mars Polar Lander, Mars Climate Orbiter, and Mars PathFinder.

This model indicates that the probability of critical errors is small with systems under 10K SLOC, but grows super-linearly as the size grows towards what is expected of future missions incorporating advanced software technology. Without improvements in software tools and methods, this model predicts a low chance of a space mission being free of critical software errors beyond 200K SLOC level. Of course, there are many examples of *commercially viable* software systems that are much larger than 200K SLOC. However, commercial viability is a much lower standard of reliability, and in fact the first deployment of a commercial system seldom has fewer critical errors (that can crash the system) than predicted in this graph.



Although this graph is based on a model that has not been validated, it provides a conceptual basis for understanding past trends and making future predictions. It also enables us to visualize the potential impact of tools for software construction and analysis. This graph is annotated to indicate potential strategies for achieving complex yet highly reliable software. *Managing Complexity* means reducing the slope of the line by reducing the factor *N*. Scaling-up SE through *software construction tools* means shifting the line over to the right by enabling developers to produce more software than the mostly manual process prevalent today. Detecting Errors means shifting the line down through the use of *software analysis tools* that detect more errors than is possible through testing alone. *Tolerate errors* means being able to detect and recover from errors that occur at runtime, so that errors that would otherwise lead to mission failure are recognized and handled. A simple example of this is runtime overflow exception handling. Clearly all these strategies will need to be combined synergistically to achieve complex yet reliable software. Managing complexity is the focus of the software process community. Tolerate errors is the focus of the fault-tolerant computing community. The rest of this paper will describe work at NASA Ames towards the other two strategies that are the topics of TACAS: software construction tools and software analysis tools.

4. Software Construction Technology for Certifiable Software

Software construction technology such as autocoders (e.g., MatrixX) or rapid development environments (e.g., Matlab) have the potential for magnifying the effort of individual developers by raising the level of software development. Studies have shown that the number of lines of code generated by each individual per day remains roughly constant no matter at which level they program. For example, the same team of developers for the Cassini software, coding at the level of conventional programming languages, could in theory generate the software for the Mars Pathfinder mission in the same amount of time using autocoding technology. Thus the graphs for software cost and schedule versus size of software system are shifted to the right, by the log of the expansion factor from the size of the spec given to the autocoder to the size of the generated code. Furthermore, in theory, the organizational factors and non-local interactions that lead to the superlinear growth in software errors with increased software size are held constant no matter at which level the software coding is done. Thus the graph for software errors versus size of software system is also shifted to the right by the same factor.

This simple analysis breaks down for mission-critical or safety-critical software, particularly for software that needs to run efficiently on limited computational platforms (as is typical for space applications, where the computer needs to be radiation hardened). For mission-critical software, certification costs dominate development costs. Unless the certification can also be done at a higher level and then translated into a certification at a lower level, cost and schedule savings will not be realized. Similarly, for autocoders that produce efficient code, non-local interactions are introduced by the autocoder itself, just as optimizing compilers introduce non-local interactions at the object code level. The non-local interactions that are

introduced must be guaranteed not to lead to software faults. Thus within the aerospace domain, the automated generation of code needs to be done with technology that ensures reliability and addresses certification issues to achieve the potential benefits of shifting the graph to the right.

The current generation of aerospace autocoding tools are based on traditional compiler technology, and are subject to the same limitations as compilers. A major limitation in today's aerospace autocoders is that they perform their process of code generation in a black-box manner. This leaves a major gap for certification, and means that with few exceptions any verification that is done at the source specification level for the autocoder does not count towards certification.

For the past several years, the Automated Software Engineering group at NASA Ames has been developing program synthesis technology that could address the certification problem. Specifically, the technology generates code through a process of *iterated, fine-grained refinement* – with each step justified through automated reasoning; we use a combination of deductive synthesis [5] and program transformations. Many sources of error are precluded through this method of generating software. The record of the justifications for each step provides documentation and other artifacts needed for certification. The code generation process is no longer opaque, in fact, the process can potentially be better documented than with manual code development. Part of the research at NASA Ames has been to realize the potential of this methodology with respect to certification, as described below.

Demonstrations of the automated construction of verifiably correct software in the 1980s focused on small examples relevant to computer science, such as programs that sorted lists. In the 1990s more ambitious demonstrations showed how more sophisticated programs could be generated. This involved representations of knowledge beyond low-level programming knowledge, including algorithm knowledge [6], design knowledge, and domain knowledge. These demonstrations included Amphion/NAIF [7], which demonstrated the generation of programs for calculating space observation geometries, and KIDS/Planware [8], which generated planning and scheduling programs for military logistics support as well as other applications.

Recently, the Automated Software Engineering group at NASA Ames has demonstrated scaling this technology to the avionics domain [9], and also demonstrated how this technology could be used to generate artifacts for certification. Guidance and navigation are primary control functions of the avionics in aerospace vehicles. Unfortunately, as documented in section 2, faulty geometric state estimation software within GN&C systems has been a factor in numerous aerospace disasters, including the Mars Polar lander, the Mars climate orbiter, and Ariane 501; as well as contributing to many near-misses. Thus in this domain verifiably correct software is critical.

Amphion/NAV [10] is a program synthesis system that generates geometric state estimation programs through iterative refinement. The generated programs iteratively estimate the values of *state variables* -such as position, velocity, and attitude- based on noisy data from multiple sensor sources. The standard technique for integrating multiple sensor data is to use a Kalman filter. A Kalman filter estimates the state of a linear dynamic system perturbed by Gaussian white noise using measurements

linearly related to the state but also corrupted by Gaussian white noise. The Kalman filter algorithm is essentially a recursive version of linear least squares with incremental updates. More specifically, a Kalman filter is an iterative algorithm that returns a time sequence of estimates of the state vector, by fusing the measurements with estimates of the state variables based on the process model in an optimal fashion. The estimates minimize the mean-square estimation error.

Amphion/NAV takes as input a specification of the process model (a typical model is a description of the drift of an INS system over time), a specification of sensor characteristics, and a specification of the geometric constraints between an aerospace vehicle and any physical locations associated with the sensors - such as the position of radio navigation aids. The input specification also provides architectural constraints, such as whether the target program should have one integrated Kalman filter or a federation of separate Kalman filters. Amphion/NAV produces as output code that instantiates one or more Kalman filters. Like rapid prototyping environments, Amphion/NAV supports an iterative design cycle: the user can simulate this generated code, determine that it is lacking (e.g., that the simulated estimate for altitude is not sufficiently accurate), reiterate the design (e.g., by adding a radio altimeter sensor to the specification), and then rerun the simulation. However, Amphion/NAV also produces artifacts that support certification.

Within the Amphion/NAV system, the Snark theorem prover [11] is given the (negated) specification and the axioms of the domain theory and then generates a refutation proof and a vector of witness terms for the output variables, in our case these are applicative terms comprising the synthesized program. These terms are then subsequently transformed through a set of program transformations to produce code in the target programming language, which is C++ with subroutine calls to the Octave library (a Matlab clone).

Amphion/NAV produces extensive documentation both as a printed document and an HTML document indexed by links in the generated program. These are both derived from a common XML document that is generated through the algorithm described below. This documentation can be used by system developers for test and code review purposes, and also for system integration and maintenance. It supports code reviews in the certification process, as well as providing a trace from the code back to the specification. The process by which the code is generated is documented by mapping the derivation tree to English language text. The technology for generating this documentation consists of several components: an algorithm for generation of *explanation equalities* from the proof traces [12] together with instantiation of templates associated with axioms in the domain theory [10], and XLST translators. We focus here on the first two components.

Intuitively, an explanation of a statement in a generated program is a description of connections between the variables, functions and subroutines in that statement and the objects, relations, and functions in the problem specification or domain theory. In other words, an explanation traces back a statement in a program to the parts of the specification and domain theory from which it originated. The explanations are constructed out of templates and explanation equalities.

The algorithm for generating explanation equalities works on the proof derivation of the synthesized program. The proof derivation is a tree whose nodes are sets of formulas together with substitutions of the existentially quantified variables, and

whose arcs are steps in the proof (i.e., they encode the derived-from relation). Thus, an abstract syntax tree (AST) of the synthesized program and the empty clause is the root of this derivation tree (recall that Amphion/NAV generates resolution refutation proofs). The leaves are domain theory axioms and the problem specification. Since the AST and all formulas are represented as tree-structured terms, the derivation tree is essentially a tree of trees.

For each position of the abstract syntax tree, the explanation equality generation procedure traces back through the derivation tree extracting explanation equalities along the way. These equalities record the links between positions of different terms in the derivation. Each explanation equality is a logical consequence of the semantics of the inference rule applied at that point in the derivation tree. For example, a resolution rule will induce a set of equalities from disjuncts in the parent clauses to disjuncts in the child clause. The transitive closure of these equalities, the *goal explanation equalities*, are derived which relate positions of the generated program with terms in the specification and formulas in the domain theory.

The second ingredient for constructing explanations that are understandable to engineers in the domain are explanation templates. Domain theory axioms are annotated with explanation templates, which consist of strings of words, and variables linked to the variables in the axiom. There can be multiple templates for each axiom, where each template is indexed to a particular position in the axiom.

For each position in the generated program, an explanation is constructed through instantiation of templates linked together through the explanation equalities. First, the chain of equalities is constructed linking each position in the generated program back through the derivation tree to the specification and domain theory. Second, the corresponding chain of templates is constructed by extracting the template (if any) associated with each node in this chain (that is, the template associated with the position in the axiom from which the term was derived). Third, the templates are instantiated by replacing the variables (corresponding to variables in the axioms) with the corresponding terms in the problem specification - this correspondence defined by the goal explanation equalities originating at the variable position. Finally, the templates are concatenated together in the order defined by the derivation tree. A more extensive mathematical description of this algorithm in terms of equivalence classes of terms can be found in [10].

Our current research on certifiable program synthesis focuses on generating other artifacts besides documentation that support the certification process. We have developed prototype algorithms that generate test cases which exercise the code. We are also developing the capability to generate formal annotations in the synthesized code that support independent certification algorithms based on extended static checking. These algorithms provide an independent check that the code conforms to safety properties, such as the consistent use of physical units and co-ordinate frames. Early results from this research can be found in [13].

5. Software Analysis Technology

Mathematical verification technology has had a profound effect on commercial digital hardware engineering, where finding errors prior to initial fabrication runs is orders of magnitude more cost-effective than discovering these errors afterwards. This technology includes equivalence checkers for combinatorial aspects of digital circuits and model checkers for sequential and concurrent aspects of digital hardware systems.

Software verification technology could have a similar effect on software systems, where finding an error prior to system deployment can be orders of magnitude more cost-effective than finding the error after it has caused a mission-critical failure. From the viewpoint of the graphical model of software errors versus software size presented in section 3, software analysis technology shifts the line downward by finding a substantial fraction of the software errors, which are then subsequently remedied. The downward shift on this log-log graph is the log of the fraction of errors that are not found by the analysis tools, e.g., if one percent of the errors are not found, then the graph is shifted down by a constant factor of negative two.

Software analysis technology faces greater technical challenges than digital hardware analysis technology: lack of regularity, non-local interactions, and scale. Digital hardware is highly regular and repetitive in its layout, a feature that is implicitly exploited by analysis technology such as BDD-based model checkers. The regularity of digital hardware appears to keep BDD representations of sets of states tractable in size for such model-checkers. In contrast, software compactly encapsulates regularity through constructs such as iteration and recursion. Hence on a line-by-line basis, software is denser than digital hardware circuits; it is a more compact representation of the state-space of a system. Partially because of this lack of regularity, digital hardware analysis techniques often hit combinatorial explosions rather quickly when applied to software. Similarly, physical constraints make non-local interactions in digital hardware costly in power and wiring layout. However, non-local interactions dominate software, from exception handling to interrupts to asynchronous interleaving of multiple threads. Finally, the reachable state space that needs to be analyzed for a software system is usually much larger than for even complex digital hardware systems such as a microprocessor. Even though the software executes on the hardware, its state space is an exponential function of not just a microprocessor, but also the memory in which the program and the data is stored.

This section will provide an overview of research at NASA Ames to develop model-checking technology suitable for software. Because model-checking thoroughly explores the graph of reachable states, it has the precision needed to find even subtle errors arising from concurrent software, and hence shift the graph downward substantially. The automated nature of model-checking makes it potentially attractive for use outside of the formal methods research community. However, because of the factors described in the paragraph above, the combinatorics for model-checking software is much worse than the combinatorics for similarly analyzing hardware. In addition, the semantics of object-oriented software fundamentally mismatches the assumptions of previous model-checking algorithms.

To meet these challenges and to increase the size of software systems that could be analyzed, our research has evolved from case studies using previous model-checking

technology (e.g., SPIN [14]), to prototypes that translated from software artifacts to the modeling language for previously existing model checkers, (i.e. Promela), and finally to a new model-checking technology built from the ground-up for the semantics of object-oriented software. To meet the challenge of scale, we now use a synergy of technologies to cut down on the combinatorics of the reachable state space. In fact, our Java Pathfinder system incorporates static analysis algorithms [15], predicate abstraction algorithms [16] based on automated theorem proving, data abstraction algorithms based on abstract interpretation [17], and guided search techniques. The result over the last five years has been a steady increase in the number of source lines of code that can be analyzed, measured both by the size of the programs that can be analyzed before running out of main memory (the limiting factor in explicit state model-checking) and as measured by human productivity when using the technology. The source lines of code analyzed per person per day has gone up from 30 lines of code in 1997 to 1,000 lines of code in 2002.

Java PathFinder 2 (henceforth JPF) [18] is a Java model checker built on a custom-made Java virtual machine (JVM) that takes as input Java bytecode. The Java language was chosen as our initial research target because it is a streamlined modern object-oriented language without complicating factors such as arbitrary pointer arithmetic. Developing a custom JVM solved the problems of handling the semantics of an object-oriented language without awkward translations to a model-checker with mismatched semantics. JPF introduced a number of novel features for model checking:

- Support for dynamic object creation and class loading
- Support for garbage collection
- Support for floating point numbers

JPF is an explicit-state model checker : it enumerates each reachable system state from the initial state. In order to not redo work (and therefore be able to terminate) it is required to store each state reached in the graph of states. When analyzing a Java program each state can be very large and thus require significant memory to store, hence reducing the size of systems that can be checked. In JPF state-compression techniques [19] reduce the memory requirements of the model checker by an order of magnitude. Another novel feature of JPF is the use of symmetry reduction techniques to allow states that are the same modulo where an object is stored in memory to be considered equal [19]. Since object-oriented programs typically make use of many objects, this symmetry reduction often allows an order of magnitude less states to be analyzed in a typical program. JPF also supports distributed memory model checking, where the memory required for model checking is distributed over a number of workstations [19], hence enlarging the size of the state space that can be explored by the number of workstations. Experiments on partitioning the state space over different workstations showed that dynamic partitioning works best, where partitions change during a model checking run rather than be statically fixed at initialization.

When using JPF to analyze a program for adherence to properties, including properties specified in temporal logic, a user works with a collection of tools that analyze and manipulate the program in synergy with the core model-checking algorithm. Some of these tools, such as program slicing and abstraction, are invoked by the user prior to submitting the program to the model checker. Others, such as

heuristics to guide the search of the model checker, are selected by the user as parameters to the model checker. Below a brief summary is presented of some of these tools and synergistic algorithms, the interested reader can find more detail in the cited papers.

Program Abstraction. Program abstraction supports the simplification and reduction of programs to enable more focused and tractable verification, often resulting in dramatic reductions of the state space. Although our tools support abstraction for both under- and over-approximations, in practice we mostly use over-approximations that preserve concrete errors under the abstraction mapping, but potentially introduce additional spurious errors. The abstractions are done through data type abstraction (abstract interpretation) and through predicate abstraction [16]. The data type abstractions are calculated offline with PVS [20]. The predicate abstraction technology invokes the Stanford Validity Checker (SVC) [21] to calculate abstract program statements given a predicate definition. Object-oriented programs are particularly challenging for predicate abstraction, since predicates can relate variables in different classes that have multiple dynamic instantiations at run-time.

Static Program Analysis. Static program analysis technology consists of several classes of algorithms that construct and analyze graphs that represent static dependencies within programs. Applications of this technology are in program slicing [22], control flow analysis, concurrency analysis, points to and alias analysis. Static analysis information can be useful in optimizing and refining model checking and program abstraction techniques. All these applications of static analysis are incorporated in JPF and its associated tools.

Environment Modeling and Generation. One of the steps in behavioral verification is constructing a model of the environment to which the software reacts. Model checking applies to a closed system. In order to check a reactive system such as an autonomous controller, that system must be completed with a simulated *environment* with which it will interact — in much the same way as testing requires a test harness and suitable test cases. The environment must reproduce the different possible stimuli that the system will possibly meet when in operation, as alternative choices that the model checker can explore. Technology has been developed to support modeling of complex non-deterministic environments. Environment models are constructed using a combination of special object-oriented methods to support non-deterministic choice, generic reusable environment components, and environmental constraints specified in linear temporal logic [23].

During the course of the research leading to the JPF software model-checking technology, we have done a series of case studies that both demonstrate the increasing power of the technology to address NASA's needs for reliable software, and also provided us feedback to inform the direction of our research. Below we highlight some of these case studies related to the strategic NASA requirements described in the first section. The case studies address verification of autonomy software, verification of a next-generation aerospace operating systems, and verification of mixed-initiative human-machine systems.

Autonomy Software. Starting in 1997 the ASE group analyzed parts of the Remote Agent [24] that formed part of the Deep-Space 1 mission, the first New Millennium mission dedicated to flight validating new technology in space for future missions. The Remote Agent is an integrated set of AI-based autonomy software - planning/scheduling, robust execution, and model-based diagnosis - that is a prototype for future autonomy software that will control spacecraft and rovers with minimal ground intervention. The Remote Agent software was tested in space during the Deep-Space 1 mission. The Remote Agent team asked the ASE group to analyze portions of the Lisp code prior to going operational in May 1999. The analysis was performed through manual development of a model in PROMELA of parts of the code. The SPIN model checker [14], developed at Bell Labs, was used to analyze the model. The model checking was very successful in that it uncovered five previously undiscovered bugs in the source code [24]. The bugs were concurrency errors and data race conditions, many of which could have led to deadlock.

After launch, the Remote Agent was run as an experiment for a week in May 1999. Soon after the experiment began, the Remote Agent software deadlocked due to a missing critical section in a part of the code that was not analyzed prior to launch through model checking. After a dump of the program trace was downloaded to earth, the remote agent team was able to find the error. As a challenge to the ASE group, without telling us the specific nature of the error, they asked us to analyze the subsystem in which the deadlock occurred (10,000 lines of Lisp code), and gave us a weekend to find the error. We were able to find the error through a combination of inspection and model checking [25]; it turned out to be nearly identical to one that we previously found through model checking (on a different subsystem) prior to launch.

Next-Generation Aerospace Operating System. In 1998 Honeywell Technology Center approached the ASE group with a request to investigate techniques that would be able to uncover errors that testing is not well suited to find [26]. The next generation of avionics platforms will shift from federated system architectures to Integrated Modular Avionics (IMA) where the software runs on a single computer with an operating system ensuring time and space partitioning between the different processes. For certification of critical flight software the FAA requires that software testing achieve 100% coverage with a structural coverage measure called Modified Condition/Decision Coverage (MC/DC). Honeywell was concerned that even 100% structural coverage would not be able to ensure that behavioral properties like time-partitioning will be satisfied. In particular, Honeywell's real-time operating system, called DEOS, had an error in the time partitioning of the O/S that was not uncovered by testing.

Similar to the challenge from the Remote Agent team, Honeywell asked us to determine if model-checking could uncover the error without us knowing any specifics of the error. At this point we had developed a translator into SPIN, so the modeling was done directly in a programming language (Java). This considerably speeded up developing the abstracted model of DEOS. With this technology, we were able to find a subtle error in the algorithm DEOS used to manage time budgets of threads. The analysis of the DEOS system was well received by Honeywell and led to

Honeywell creating their own model checking team to analyze future DEOS enhancements as well as the applications to run on top of DEOS.

Human-Computer Interactions. The environment generation technology is a critical part of our ongoing research in Human/Computer System Analysis. As described in section one, human/machine interactions are a common source of critical-software related errors. The technology for environment modeling has been extended for modeling the incorporation of human actors into system models containing actual software. This technology consists of specialized static analysis and program abstraction techniques. This was applied in the summer of 2001 by a summer student to model the interactions of an autopilot and a pilot, successfully uncovering automation surprise scenarios. In one scenario the autopilot fails to level out at the altitude specified by the pilot and continues to climb or descend, resulting in a potentially hazardous situation.

Summary

Complex software will be essential for enabling the mission capabilities required by NASA in the next forty years. Unfortunately, as NASA software systems evolved from the tens of thousands of source lines of code typical of the nineteen eighties deep-space missions to the hundreds of thousands of source lines of code typical of the nineteen nineties and beyond, software faults have led to a number of serious mission failures. A model of software errors versus size of software systems indicates that without new software development technology this trend will accelerate. This model is based on a reasonable extrapolation of models for software cost and schedule that have been calibrated, and an analysis of underlying causes.

Using this graphical model, we see that software engineering tools can mitigate this trend towards super-linear error growth through two orthogonal directions: shifting the graph to the right through software construction technology, and shifting the graph downwards through software analysis technology. Research at NASA Ames towards software construction tools for certifiable software and towards software model-checking tools was overviewed in this paper.

Acknowledgements

My colleagues in the Automated Software Engineering group at NASA Ames have compiled an enviable research track record while working to meet NASA's strategic goals and challenges described in the first two sections of this paper. The last two sections of this paper are essentially overviews of recent research within our group. More detailed descriptions can be found in the cited papers. The graphical model for software reliability found in section three was developed in conjunction with Cordell Green of the Kestrel Institute, with input and encouragement from Barry Boehm and Peter Norvig.

References

1. Perrow, C.: Normal Accidents: Living with High Risk Technologies, Princton University Press (1999)
2. Neumann, P.: Computer Related Risks, Addison-Wesley Press, 1995
3. Lions, J. "Report of the Inquiry Board for Ariane 5 Flight 501 Failure", Joint Communication ESA-CNES (1996) Paris, France
4. Boehm, B. et al: Software Cost Estimation with COCOMO II, Prentice Hall PTR (2000)
5. Green, C.: Application of theorem proving to problem solving. Proceedings Intl. Joint Conf. on Artificial Intelligence (1969) 219-240
6. Smith, D., Lowry, M., Algorithm theories and design tactics. Lecture Notes in Computer Science, Vol. 375 (1989) 379-398, Springer-Verlag.
7. Stickel, M., Waldinger, R., Lowry, M., Pressburger, T., Underwood, I. : Deductive Composition of Astronomical Software from Subroutine Libraries. Lecture Notes in Computer Science, Vol. 814. Springer-Verlag (1994).
8. Smith, D.: Kids: A semiautomatic program development system. IEEE Trans. Software Engineering 16(9): 1024-1043 (1990).
9. Brat, G., Lowry, M., Oh, P., Penix, J., Pressburger, T., Robinson, P., Schumann, J., Subramaniam, M., Whittle, J.: Synthesis of Verifiably Correct Programs for Avionics. AIAA Space 2000 Conference & Exposition, (2000), Long Beach, CA
10. Brat, G., Lowry, M., Oh, P., Penix, J., Pressburger, T., Robinson, P., Schumann, J., Subramaniam, M., Van Baalen, J., Whittle, J.: Amphion/NAV: Deductive Synthesis of State Estimation Software. IEEE Automated Software Engineering Conference (2001), San Diego, CA
11. Stickel, M. The snark theorem prover, 2001. http://www.ai.sri.com/~stickel/snark.html.
12. Van Baalen, J., Robinson, P., Lowry, M., Pressburger, T.: Explaining synthesized software. IEEE Automated Software Engineering Conference (1998), Honolulu, Hawaii
13. Lowry, M., Pressburger, T., Rosu, G.: Certifying Domain-Specific Policies. IEEE Automated Software Engineering Conference (2001), San Diego, CA
14. Holzmann, G., Peled, D.: The State of SPIN. Lecture Notes in Computer Science, Vol. 1102 (1996), Springer-Verlag.
15. Corbett, J., Dwyer, M., Hatcliff, J., Pasareanu, C., Robby, Laubach, S., Zheng, H. : Bandera: Extracting Finite-state Models from Java Source Code. Proceedings of the 22nd International Conference on Software Engineering (2000), Limeric, Ireland.
16. Visser, W., Park, S., Penix, P.: Using Predicate Abstraction to Reduce Object-Oriented Programs for Model Checking. Proceedings of the 3rd ACM SIGSOFT Workshop on Formal Methods in Software Practice (2000).
17. Dwyer, M., Hatcliff, J., Joehanes, J., Laubach, S., Pasareanu, C., Robby, Visser, W., Zheng, H,: Tool-supported program abstraction for finite-state verification. Proceedings of the 23rd International Conference on Software Engineering (2001).
18. Visser, W., Havelund, K., Brat, G., Park, S.: Model checking programs. IEEE International Conference on Automated Software Engineering, (2000) Grenoble, France.
19. Lerda, F., Visser, W.: Addressing dynamic issues of program model checking. Lecture Notes Computer Science, Vol. 2057 (2001), Springer-Verlag.
20. Owre, S., Rushby, J., Shankar, N.,: PVS: A prototype verification system. Lecture Notes in Computer Science, Vol. 607 (1992), Springer-Verlag.
21. Barrett, C., Dill, D., Levitt, J.: Validity Checking for Combinations of Theories with Equality. Lecture Notes in Computer Science, Vol. 1166 (1996), Springer-Verlag.
22. Hatcliff, J., Corbett, J., Dwyer, M., Sokolowski, S., Zheng, H.: A Formal Study of Slicing for Multi-threaded Programs with JVM Concurrency Primitives. Proc. of the 1999 Int. Symposium on Static Analysis (1999).

23. Pasareanu, C.: DEOS kernel: Environment modeling using LTL assumptions. Technical Report NASA-ARC-IC-2000-196, NASA Ames, (2000).
24. Havelund, K., Lowry, M., Penix, P.: Formal Analysis of a SpaceCraft Controller using SPIN. Proceedings of the 4th SPIN workshop (1998), Paris, France.
25. Havelund, K., Lowry, M., Park, S., Pecheur, C., Penix, J., Visser, M., and White, J.: Formal Analysis of the Remote Agent Before and After Flight. Proceedings of the 5th NASA Langley Formal Methods Workshop (2000),
26. Penix, J., Visser, W., Engstrom, E., Larson, A., and Weininger, N. Verification of Time Partitioning in the DEOS Scheduler Kernel. In Proceedings of the 22nd International Conference on Software Engineering, (2000) Limeric, Ireland.