Autonomy Software: V&V Challenges and Characteristics

Johann Schumann and Willem Visser
RIACS / NASA Ames, Moffett Field, CA,
{schumann|wvisser}@email.arc.nasa.gov

Abstract—The successful operation of unmanned air vehicles requires software with a high degree of autonomy. Only if high level functions can be carried out without human control and intervention can complex missions, in a changing and potentially unknown environment, be carried out successfully.

Autonomy software is highly mission and safety critical: failures, caused by flaws in the software cannot only jeopardize the mission, but could also endanger human life (e.g., a crash of an UAV in a densely populated area). Due to its large size, complex architecture, and use of specialized algorithms (planners, constraint-solvers, etc.), autonomy software poses specific challenges for its verification, validation, and certification.

We have carried out a survey among researchers and scientists at NASA to study these issues. In this paper, we will present major results of this study, discussing the broad spectrum of notions and characteristics of autonomy software and its challenges for design and development. A main focus of this survey was to evaluate verification and validation (V&V) issues and challenges, compared to the development of "traditional" safety-critical software. We will discuss important issues in V&V of autonomous software and advanced V&V tools which can help to mitigate software risks. Results of this survey will help to identify and understand safety concerns in autonomy software and will lead to improved strategies for mitigation of these risks.

TABLE OF CONTENTS

0-7803-9546-8/06/$20.00/©2006 IEEE
IEEEAC paper # 1261

1. INTRODUCTION

The successful operation of unmanned air vehicles (UAV) requires autonomous functions on various levels. Even in the case, where a UAV is controlled remotely via a human pilot, autonomous systems must make sure that the UAV remains safe and controllable in case of a disruption of the command and control radio link. More advanced missions, like unmanned surveillance operations need a higher level of autonomy, because the UAVs are supposed to operate for a longer time without human control and intervention. Complex missions even require that the UAV can successfully cope with changing and unknown environments and carry out its operation under changing operation profiles without human control.

In modern systems, autonomous operation is realized in software. Autonomy software is typically highly mission and safety critical: failures, caused by flaws in the software cannot only jeopardize the mission, but could also endanger human life. For example, a damaged UAV must—without human help and control—be able to avoid densely populated areas for an emergency landing or crash.

Since autonomous operation has become important and desirable in a multitude of areas, like robotics, space missions, underwater exploration, and such, many approaches toward this topic can be found. However, there is no comprehensive and accepted notion of the risks of autonomy (or even, what autonomy actually is), and mature technology to provide good guarantees for the safe and reliable operation is not yet available.

In order to address these issues, the authors surveyed NASA experts in autonomous systems and experts in software engineering about autonomy software. Our questionnaire contained 24 questions, to be answered numerically in the range from 1 to 5 (disagree, partially disagree, neutral, somewhat agree, fully agree), as well as a number of questions for which textual answers were solicited. In this paper, we present the numerical results as pairs: the mean values for the autonomy experts and for the software engineering experts).

In the rest of this paper, we will present results from this survey and discuss characteristics of autonomy software, issues in engineering and verification/validation of those systems. Finally, we present some techniques and advanced V&V tool that can help to mitigate the software risks inherent in autonomy software.

2. Autonomy Software

What is Autonomy Software?

Subjectively, autonomy software is concerned with the automatic control of a system (e.g., UAV, spacecraft, robot, rover) without the need of human intervention or control. A more detailed look at the attributes, usually associated with autonomy software [7] reveals a broad range from self-diagnosing to self-managing and self-adapting. The main point here is that the software actually contains some components for executing actions and making decisions. In principle, the autonomy software must be able to reason about the environment and the system itself. Such a notion of autonomy software, or autonomic software, as coined by IBM[1], is very broad and on a high level. Although a variety of work on the topic of autonomy exists, especially in the area of agent-based systems (e.g., [4]) or autonomic computing, the authors tried to obtain a more practical definition of what autonomy software actually is.

In the survey, a number of software engineering experts and experts in the area of autonomous systems had been asked to give an informal definition of autonomy software. The spectrum of answers showed some strong commonalities, but several important aspects couldn't be more disjoint. In order to illustrate the characteristics of autonomy software more clearly, let us discuss three examples of NASA autonomy software.

The SAFM (Shuttle Abort Flight Management) system is a piece of software, which has been developed for the Shuttle update program [5]. It is a Shuttle on-board system that advises the Shuttle crew about launch abort options in case of an engine failure. In such a case, SAFM calculates the most appropriate profile of the abort flight path (e.g., which emergency landing strip to use as well as navigational aid) and displays the data to the Shuttle pilot, who then will make the final decision and carry out the required operations. In the view of NASA managers, SAFM is an autonomous software system, since it operates without ground control. Obviously, SAFM is safety critical, although the software itself has no means to actually control the Shuttle at any time.

The DART (Demonstration of Autonomous Rendezvous Technology) spacecraft[2] was intended as a prototype to demonstrate automatic (and un-guided) in space rendezvous and docking. Using its on-board sensors (image processing, radar, inertial navigation, and GPS), the autonomy software on-board the spacecraft controls the spacecraft to attempt a rendezvous with another satellite without human intervention. Due to problem(s) not yet determined, however, the approach was automatically aborted before the target had been reached. Such an autonomous software system provides a higher level of autonomy, as the software can and has to control the entire system for an extended period of time. Still, the system has a fixed "goal" and the number of external (environmental) parameters is relatively small.

An autonomous planning and scheduling system for a Mars rover (e.g., the PLEXIL [8] planning and execution system) has an even more complex task. Based upon initial high-level goals, the system has to automatically develop a plan on how to achieve this goal. This plan has to fulfill all constraints, before it can be executed. During the execution of the plan, the state of the system or the environment might change, making it necessary for the autonomous system to re-plan the entire activity, and possibly even revise achievable goals.

Although the various kinds of autonomy software work off very different requirements, they share (at least) one characteristic item: the autonomy software is mission and safety critical, which means that failure of the autonomy software can lead to mission failure and could endanger human life. Therefore, verification and validation (V&V) of autonomy software is an extremely important issue. In the following sections, we will discuss results from our survey, regarding the major characteristics of autonomy software and the software engineering task to build such a system.

Characteristics of Autonomy Software

Despite all details on what comprises an autonomous system, there was a clear agreement that "the autonomous system must be able to execute a number of basic steps without human intervention in order to achieve a given goal" $(4.14/4.25)$[3].

As to major components of a software architecture for autonomy software, answers were more disjoint. Asked if a typical autonomy system contains a planner, an executive (for execution of the plan), and a state estimation component, autonomy experts tended to agree somewhat stronger (appr. 1 of the 5 levels) than the software engineering experts. Asked if the structure and complexity of autonomy software is the same as for a comparable traditional software system, we received a slightly negative, but inconclusive answer $(2.71/2.17)$. On the other hand, there was no clear indication that an autonomy software system should contain AI-based or machine-learning algorithms $(3.14/3.17)$, is agent based $(3.00/2.17)$, is model-oriented/model-based $(2.57/3.00)$, or has non-deterministic elements $(3.29/3.00)$.

3. Engineering an Autonomous System

An autonomous software system is a complex, safety-critical piece of software of considerable size. Therefore, autonomy software (as any software) must be designed and engineered carefully. In a traditional sense, the software life-cycle phases

[1] www.research.ibm.com/autonomic

[2] www.nasa.gov/mission_pages/dart/main/

[3] In this paper, with A/B we denote that the mean value of the answers (on a range from 1 (=disagree) to 5(=fully agree) given by autonomy experts (A), those given by software engineering experts (B).

of design, implementation, testing, and deployment are distinguished. Typical software processes order these phases sequentially (e.g., waterfall model), or in spirals. A graphical representation of the phases of the software development—starting from the system requirements—is depicted in Figure 1. Solid lines show the dependencies during the actual development, dashed lines how the verification tasks relate to previous stages in a backwards manner. Finally, validation (dotted lines) indicate, how the software is tested against the appropriate requirements and specifications.

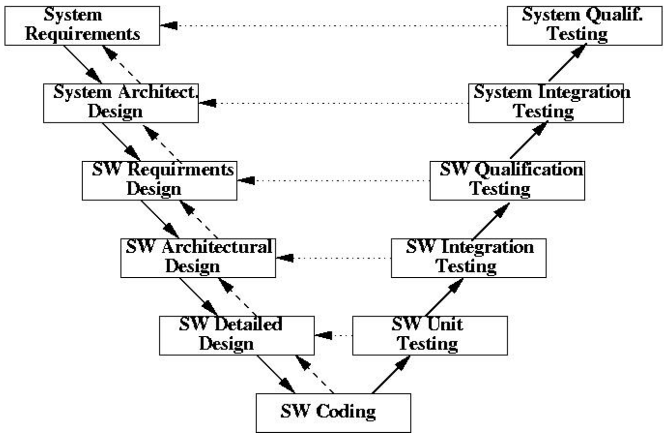


Figure 1. V-shaped software development and V&V process. Dependencies between the development phases are indicated by solid arrows. Dashed arrows concern verification activities; dotted lines validation activities.

In our survey, we asked, how the special characteristics of autonomous software systems are and need to be reflected in the way, how such software is designed and implemented. Most software engineering experts expressed the opinion that autonomy software cannot be developed using the same software development process as traditional software (2.00). For autonomy experts, using the same process seemed to be a reasonable option (3.29). Likewise, there was less special expertise/experience required from the development team ($2.86/3.50$). The overall productivity of the software development (in lines of code per person-month) was rated roughly the same as for traditional software ($2.43/3.33$).

Members of both areas expressed the specific and highly important role of system requirements ($4.00/3.67$). These requirements must not only capture the underlying autonomy execution machinery, but must also express the characteristics and requirements of the model. Since model accuracy and consistency is important, such high rating was expected.

Much less clear were the trends in the design and programming paradigm. The survey revealed that neither an object-oriented paradigm ($2.71/2.50$) nor model-based/model-oriented design principles ($2.57/3.00$) are favored in any sense. Also a distributed or multi-threaded implementation (which might be suitable for an agent-based design) seemed to be even less important ($2.00/1.33$).

Whereas it seems that for the design and implementation of an autonomy system pretty much the same process can be used to achieve roughly the same productivity, things are very different for the verification and validation activities. It was a consensus that current best practices for V&V are not quite sufficient for autonomy software ($2.14/1.83$). Whereas the autonomy experts are fairly neutral regarding the suitability of a traditional V&V process, V&V experts call for a different or augmented V&V process ($3.29/1.83$). In their opinion, also the required effort of V&V for an autonomy system compared to the V&V of a traditional system (in order to achieve the same level of reliability) is somewhat higher (1.83 compared to 2.57 for autonomy experts). However, the range of answers was very wide, ranging from "almost none" and "not a lot" to "50% of our work" to "all of my time".

According to the survey, additional effort spent on V&V of autonomous software is mainly caused by three issues:

1. the larger size and higher complexity of the valid input space, which can contain system status, environmental information, intended goals, and constraints,[4]
2. the complexity of the program logic (often highly complicated search-based, iterative algorithms like planners or reasoning engines) required to derive the answer in the autonomous system, and
3. the size and complexity of the domain model and description of the environment. This domain model has to be validated in addition to the execution software itself.

4. ERRORS AND RISKS IN AUTONOMOUS SYSTEMS

Risks can come in from many areas/sources and can show up during the entire software and system life-cycle. Especially for a safety-critical system, like autonomy software, the risk can be considerable, leading to costly mission failures or potential loss of human life (e.g., for the SAFM advisory system, or UAVs). With each potential risk, a probability is attached, indicating how likely such a failure event is. There is substantial work in the literature on this topic (e.g., [2] for software risk identification). In this survey and paper, we focus on coding errors and risks. Errors which are introduced during implementation can pose a substantial risk for the entire software system, as many incidents show. In traditional safety-critical software, there is a fairly stable list of "usual culprits", i.e., errors (or error classes), which tend to be particularly harmful. Typical examples include buffer overrun errors, uninitialized variables, or synchronization problems in distributed system. Table 1 shows a list of important coding error classes for safety-critical software (from [6]). For each error class, the importance to find such bugs (proportional to the risk), as well as the difficulty to locate such errors in the text is given. This list is the result of a survey; similar error ratings were obtained in most of the application domains

[4]Often, histories of these values also form part of the valid input space.

(NASA, aerospace industry).

Obviously, such errors can show up in autonomy software. A question, which we addressed in our survey was, how different such a list would look like in the area of autonomy software. Results of the survey showed that the coding errors, found in autonomy software are not specific to autonomy software; most of these errors could be found in traditional software as well.

On one hand, this result is reassuring in the sense, that no fundamentally different V&V tools must be designed. On the other hand, subtleties in autonomy-specific errors (e.g., modeling errors) are not yet fully understood.

Imp	Diff	Property
5	3	Divide by zero
5	3	Array index overrun
5	5	Mathematical functions sin, cos, tanh
5	1	Use of un-initialized variables or constants
3	3	No unused variables or constants
4	2	All variables explicitly declared
5	5	Proper synchronization in multi-threaded execution
4	4	Incorrect computation sequence
5	3	Loops are executed the correct number of times
5	3	Each loop terminates
3	2	All possible loop fall-throughs correct
4	3	Priority rules and brackets in arithmetic expression evaluation used as required to achieve desired results
5	5	Resource contention
5	2	Exception handling
5	5	The design implemented completely and correctly
4	2	No missing or extraneous functions.
5	1	Error messages and return codes used
5	1	Good code comments

Table 1. Sample Questionnaire for traditional safety-critical code

5. Risk Mitigation

In order to mitigate the risks of using autonomy one can improve the verification and validation process for such systems. From our survey it emerged that the surveyed autonomy system developers didn't use any special/custom tools for V&V, nor did they consider the current best practices for software V&V adequate to ensure reliable autonomous systems will be developed. Current best practice for software V&V is testing, hence we here consider three additional approaches to mitigate risk: static analysis, model checking and runtime analysis. For each of these techniques we relate experimental results to show what the expected risk mitigation for each technique and error-class combination is. For brevity we only consider a few relevant software error-classes, that are not autonomy specific. Note that our survey results indicate that errors in autonomy software are similar to errors commonly found in any software system - the complexity of V&V is therefore not in the type of errors, but rather the complexity of the software and the environment it operates in.

Methods and Approaches

We consider the following three advanced V&V techniques that can augment testing for autonomy V&V:

Static analysis This approach allows a program to be analyzed without having to execute it, and typically checks for the potential of run-time errors such as null pointer dereferences, array out of bounds accesses, divide-by-zero and uninitialized variable usage. The strength of this technique is that it evaluates the program operations for all possible execution environments, and the weakness is that it might produce false warnings.
Model Checking This technique allows the analysis of all possible program behaviors for essentially any behavioral property violation - although it performs best on properties such as deadlocks and race violations that traditional testing cannot easily detect. Its major weakness is that it doesn't scale to large programs.
Run-time analysis This is an advanced form of testing where the execution of the program is monitored to check for some common errors (such as potential for deadlocks, data races, etc.) as well as functional properties of the program under test. This approach requires program instrumentation, that can often be achieved through aspect-oriented programming.

Tools and Techniques for V&V

For static analysis we considered two commercial tools, namely PolySpace[5] and Coverity[6]. PolySpace does a dataflow analysis based on abstract interpretation. Coverity does a path sensitive analysis and has a lower error detection rate, i.e., it might miss to flag errors. However, Coverity rarely produces false warnings, whereas PolySpace produces large numbers of false warnings that the user need to evaluate to determine whether they are real errors or not. They also behave quite differently in setup and running times: Coverity is easy to configure and runs in a matter of minutes and PolySpace is very complicated to get running and typically runs for days on programs of the order of $30,000$ lines of code. Both tools analyze C and C++ programs and they were evaluated on NASA flight code: one unmanned autonomous mission and a relevant portion of SAFM code.

For model checking we used the Java PathFinder (JPF)[7] model checker for Java code. It is an explicit-state model checker that can handle programs up to $10,000$ lines of code. Run-time analysis was done with the commercial Temporal Rover[8] tool as well as the special purpose Java tool called JPaX [3]. These tools were all evaluated on a Java version of code for a prototype Mars Rover and the results of the experiments conducted were first reported in [1].

As for the classes of errors that we consider here, we focus on the errors that was present (or is a concern) in the three

[5]`www.polyspace.com`
[6]`www.coverity.com`
[7]`javapathfinder.sourceforge.net`
[8]`www.time-rover.com`

systems we mention above (unmanned and manned NASA flight software and the Rover code). These are typical errors that one would anticipate in code and since the three systems are all autonomy related therefor also in autonomy code.

After analyzing the three systems with the given techniques we formed a qualitative view of the risk mitigation obtained by each tool - the results are shown in Table 2. Note that although the experiments reported in [1] produced quantitative results, the analyzes done here were not done in a controlled fashion and hence the results are not as precise.

Only in two cases did our experiments validate instances where we believed our tools will perform well and it actually did perform well: Coverity on finding uninitialized variables and model checking with JPF for detecting deadlocks/dataraces. The worst cases are when we believed the tools can perform a good job and then they are either not applicable at all or they perform very badly. For example, we believed Coverity can detect divide-by-zero errors, but in fact it cannot and similarly, although PolySpace can detect these it flags too many false warnings for the results to be useful. Model checking can in theory be very good at finding some of the error classes, but our current set of experiments were not capable of determining its strengths.

Note that runtime analysis, which is just an advanced form of traditional testing, performs the best in our experiments. This is mostly because some of the more behavioral types of errors (such as faults in error handling code) can only reliably be detected by running the programs. This is somewhat worrying for risk mitigation for autonomy in general, since many of the subtle errors in autonomy is more likely to be in aspects of the code where static analysis for example is not likely to perform well - finding errors in models for example. Clearly more research will be required to develop new V&V methods to mitigate some of these risks. Model checking for example has been shown to be valuable to analyze models used in vehicle health management, and, testing where the tests are designed to give coverage of the models seem like a useful avenue for investigation.

In summary it seems that the current state-of-the-art in V&V tools can find errors that are present in autonomy software, but not as reliably well as one would have hoped. In addition, tools for detecting behavioral errors, which one can easily argue will be the most complex to find in autonomy systems, are not as developed as the ones for finding (simple) runtime errors.

6. CONCLUSIONS

In this paper, we have presented results on a survey about autonomy software, its characteristics and V&V issues that was carried out at NASA in summer 2005. We had asked software engineering experts and experts in autonomy software. Although most of the projects originated from a NASA background (Shuttle Autonomy, Rovers, Robotics, etc.), the main characteristics of a safety- or mission-critical autonomy software and associated verification and validation challenges seem to be the same across the board, indicating that results of our survey can be carried over to other application domains like UAV.

The main findings of the survey were:

- NASA autonomy experts considered there to be no meaningful difference between autonomy software and non-autonomy software in its structure, development process, and V&V process. Whereas software engineering experts believed that there would be a difference in development and V&V process.
- There was however consensus that current best practices in V&V is not suitable for autonomy software. A previous experiment [1], where it was shown that testing (i.e. current best practice) does not find as many defects as more advanced V&V techniques, also supports this view.

However as the qualitative results in Table 2 indicate even advanced V&V techniques don't do an adequate job of finding defects from important error classes. Furthermore, our survey indicated that some autonomy experts (and most of the software engineering experts) felt that certain behavioral errors in autonomy systems, especially errors in models, can be very hard to detect and therefore can seriously undermine reliability. Currently, only preliminary results for tools detecting such errors are available, and no commercial tools exist th detect such defects. It is therefore clear that more investment in the research and development of V&V techniques and tools specifically geared towards these behavioral and model errors is needed.

7. ACKNOWLEDGMENTS

This work is in part sponsored by a NASA H&RT project. We would like to thank the participants of the survey for they time and effort to answer our questionnaire.

REFERENCES

[1] Guillaume Brat, Doron Drusinsky, Dimitra Giannakopoulou, Allen Goldberg, Klaus Havelund, Mike Lowry, Corina Pasareanu, Arnaud Venet, Willem Visser, and Rich Washington. Experimental evaluation of verification and validation tools on martian rover software. *Form. Methods Syst. Des.*, 25(2-3):167–198, 2004.

[2] M. Carr, S. Konda, I. Monarch, F. Ulrich, and C. Walker. Taxonomy-based risk identification. Technical report, CMU/SEI, 1993.

[3] K. Havelund and G. Rosu. Monitoring java programs with java pathexplorer.

[4] M. Huhns and M. Singh (eds). *Readings in Agents*. Morgan Kaufman, 1998.

[5] M. Jackson, T. Straube, T. Fill, T. Barrows, and S. Nerneth. Onborad determination of vehicle glide capabil-

Error	Failure	static analysis				Model		Runtime	
Class	Risk	Coverity		PolySpace		Checking		Analysis	
Divide-by-zero	25%	1	X	1	3	1	?	3	2
Uninitialized Var	30%	1	1	1	2	1	?	3	3
Deadlock/Race	20%	X	X	3	?	1	1	3	2
Array Bounds	30%	1	2	1	2	2	?	3	2
Math functions	50%	X	X	X	X	3	?	2	2
Resource contention	30%	1	2	1	2	2	?	2	1
Error Handling	60%	X	X	X	X	2	?	3	3
Return Codes	15%	1	2	1	?	2	?	2	2

Table 2. Capabilities of V&V tools to detect errors for important error classes. The left column indicates the expected usefulness of the tool, the right column the usefulness as obtained in our experiments (1=very good, 2=useful, 3=probably not useful, X=cannot be used, ?=requires more research and experiments)

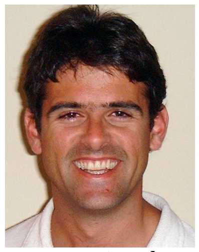
ity for the shuttle abort flight manager (safm). In *IEEE Aerospace Conference*. IEEE Aerospace, 2002.

[6] S. Nelson and J. Schumann. What makes a code review trustworthy? In *In Proceedings of the Thirty-Seventh Annual Hawaii International Conference on System Sciences (HICSS-37)*. IEEE, 2004.

[7] M. Rovatsos and G. Weiss. Autonomous software. In *Handbook of Software Engineering & Knowledge Engineering*. SEKE, 2004.

[8] V. Verma, T. Estlin, A. Jonsson, C. Pasareanu, and R. Simmons. Plan execution interchange language (plexil) for command execution. In *Proceedings iSAIRAS'05*, 2005.

Dr. Johann Schumann *(PhD 1991, habilitation degree 2000) is a Senior Scientist with RIACS and working in the Robust Software Engineering Group at NASA Ames. He is engaged in research on verification and validation of autonomy software, adaptive controllers, and learning software. He is also working on automatic generation of navigation and state estimation code. Dr. Schumann is author of a book on theorem proving in software engineering and has published more than 60 articles on automated deduction and its applications, automatic program generation, and neural network oriented topics.*



Dr. Willem Visser *Dr. Visser received his Ph.D. from the University of Manchester in 1998. After completion of his Ph.D. studies in October 1998 he started work at the Research Institute for Advanced Computer Science (RIACS) at NASA Ames. His main research focus is on the application of model checking to programming languages. He is one the main developer of the Java PathFinder model checker for Java—that won the 2003 TGIR Engineering Innovation award from the Office of Aerospace Technology at NASA. His current research focuses on using symbolic execution and model checking for test-case generation and program proofs, environment generation, feasible counter-example detection during abstraction-based model checking, belief-logics and agent verification.*