Verification of Autonomous Systems for Space Applications

G. Brat, E. Denney, D. Giannakopoulou, J. Frank*, and A. Jonsson
USRA/RIACS *NASA
NASA Ames Research Center
MS 269/2 *MS 269/1
Moffett Field, CA 94035-1000
{brat,edenney,dimitra,frank,jonsson}@email.arc.nasa.gov

Abstract—Autonomous[1,2] software, especially if it is based on model, can play an important role in future space applications. For example, it can help streamline ground operations, or, assist in autonomous rendezvous and docking operations, or even, help recover from problems (e.g., planners can be used to explore the space of recovery actions for a power subsystem and implement a solution without (or with minimal) human intervention). In general, the exploration capabilities of model-based systems give them great flexibility. Unfortunately, it also makes them unpredictable to our human eyes, both in terms of their execution and their verification. The traditional verification techniques are inadequate for these systems since they are mostly based on testing, which implies a very limited exploration of their behavioral space. In our work, we explore how advanced V&V techniques, such as static analysis, model checking, and compositional verification, can be used to gain trust in model-based systems. We also describe how synthesis can be used in the context of system reconfiguration and in the context of verification.

TABLE OF CONTENTS

1. INTRODUCTION

Autonomous software, especially if it is based on model, can play an important role in future space applications. For example, it can help streamline ground operations, or, assist in autonomous rendezvous and docking operations, or even, help recover from problems. Planners can be used to explore the space of recovery actions for a power subsystem and implement a solution without (or with minimal) human intervention. In general, the exploration capabilities of model-based systems give them great flexibility. Unfortunately, it also makes them unpredictable to our human eyes, both in terms of their execution and their verification. The traditional verification techniques are inadequate for these systems since they are mostly based on testing, which implies a very limited exploration of their behavioral space. In our work, we explore how advanced V&V techniques, such as static analysis, model checking, and compositional verification, can be used to gain trust in model-based systems.

Planning systems are made of two parts: the domain model describes the domain on which the planner can reason and the planning engine performs the reasoning (usually in the form of a systematic exploration of the state space induced by the planning goals and the domain model). These two parts yield different V&V challenges. On one hand, the planning engine can be verified in terms of its mechanisms, i.e., check that forward propagation is done correctly, check that constraints are elaborated correctly, and so on. We believe this can be done using automatic proving techniques. The use of these techniques comes with a high cost, but the planning engine only needs to be validated once. In some sense, it is a bit similar to validating a compiler. On the other hand, domain models change depending on the applications. This is where knowledge specific to a given problem is captured. Each new domain model needs to be validated. We focus on the verification of domain models for model-based systems and describe what type of properties can be checked on these models. We will also discuss how concepts used in static analysis, such as abstractions, can help decompose the verification process, hence, making it more scalable. Finally, we describe how model synthesis can help the verification process.

2. AUTONOMOUS SYSTEM OVERVIEW

The system under consideration follows a three-layered architecture. The top layer, also called the decision layer, consists of a domain model and a search engine. The

[1] 0-7803-9546-8/06/$20.00© 2006 IEEE
[2] IEEEAC paper #1488, Version 1, Updated November 4, 2005

domain model lists the constraints that describe the relationships between elements of the system (which may include environmental constraints) and the flight rules that need to be followed during planning. The search engine does the actual planning, i.e., it elaborates and scheduled activities to meet a given goal under the constraints described in the domain model [12]. In our project, we rely on the EUROPA planning framework. The output of the decision layer is a plan. The middle layer, also called the executive, takes a plan and "executes" it by issuing commands to the controllers in the bottom layer, also called functional layer. The executive is responsible for issuing commands, checking that commands are actually executed, and responding to exception signals generated by the controllers. The functional layer is a collection of controllers that actually command the hardware devices of the physical system. This is the lowest level of an autonomy software system. In our project, we rely on the functional layer provided by the CLARAty project [15].

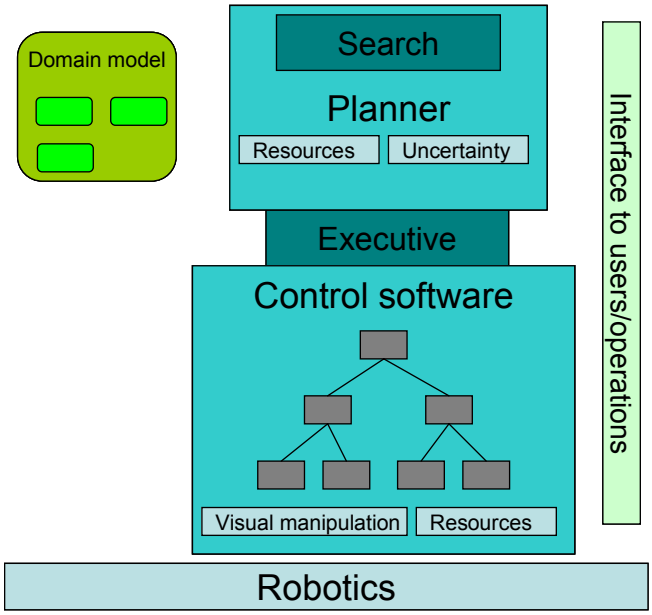

Figure 1. Architecture of autonomous systems.

The goal of our project is to provide a robust architecture. This means that we should be able to verify every component in autonomous systems built with our architecture. Moreover, we recognize that reconfiguration plays an important role at NASA. Therefore, our architecture needs to support reconfiguration, not only in terms of adding, removing, or changing components in the functional layer, but also in terms of adding planning and execution capabilities. This guarantees us that our system can be deployed in a variety of domains and that it can evolve with technology progress. As we will show in the next sections, taking into account reconfiguration actually helps the verification process.

3. Verification Framework

From a V&V point of view, each element requires different techniques, even though some basic techniques (such as code analysis) can be applied across the board.

In terms of deployment, there are degrees of autonomy. The current practice is to use planners on the ground (MER) and rely on rough command and data handling systems. Progressively, these technologies will become more and more ambitious and the migration of complex capabilities on board will depend on their trustworthiness. Trust is directly related to the number of possible scenarios that can safely be handled by an autonomous system. These scenarios define a safe operational envelope. Obviously, the goal is to verify the system for as large an envelope as possible. Current mission practices ensure safety by testing the system for a set of nominal scenarios and some off-nominal scenarios. This process is costly and does not account much for variations to nominal scenarios. Moreover, it offers little coverage of the off-nominal scenarios. We are proposing to go beyond that by first relying on an advanced form of testing (e.g., runtime monitoring) to offer a measurable level of certification for all fault classes, and then, increasing trust by progressively replacing testing by advanced formal verification techniques (e.g., static analysis and model checking). This paper focuses on the practical application of these advanced techniques.

The functional layer, containing the control software, is the closest to traditional embedded software systems. It mostly consists of control device drivers for actuators or sensors as well as mathematical and algorithmic libraries. For these elements, we are mostly concerned with certifying that they cannot crash (regardless of their inputs) and assessing their performance in terms of execution time and memory management. This suggests that a V&V approach based on embedded system verification can be applied in a straightforward manner. Note that it is possible that some elements related to planning or execution can find their way down to this layer; in which case, they are amenable to the same techniques used for the planner and the executive.

In some high-level view, the executive and the planner are quite similar. Using a formal model (the domain model for the planner and a plan for the executive), a reasoning engine computes an output (a plan for the planner and low-level commands for the executive). The reasoning engine is usually fairly mechanical (mostly performing searches) and rarely changed (in theory, only the domain model changes for a new application). Therefore, we can afford to bring to bear some "heavy" V&V techniques (e.g., precise static analysis, theorem proving) to verify that the mechanical operation of the reasoning engine (e.g., graph expansion, constraint propagations, and rollbacks) are properly performed. For the domain model, we are concerned with

well-formedness (are constraints legal?), consistency (can constraints lead to inconsistent solutions or a deadlock?), and completeness (have we specified all necessary constraints?). Given the nature of the representation (i.e., constraints), it is natural to explore the possibility of using model checking to check these properties. However, model checking is not likely to scale better than search (planning) techniques; therefore, our emphasis will be on using proper abstractions (which have a relationship to plan graphs) and compositional techniques.

As mentioned above, the engine requires checking for "mechanical" properties while models require some functional correctness verification. As shown in Figure 2, mechanical properties include programming errors (e.g., null pointer de-references), system-level errors (e.g., deadlocks and data races), and data manipulation errors (e.g., plan manipulation and constraint propagation). These properties are organized in a hierarchy that reflects the masking power of each fault class. For example, programming errors might mask synchronization errors, which in turn might mask data manipulation errors. This imposes a natural progression in the verification process: first, eliminate programming errors, then system-level errors, and finally, data manipulation errors. Once we have achieved a satisfying level of certification of "mechanical" properties, we can study performance of the engine (e.g., execution times and memory usage), which is important for on-board execution. Note that mechanical properties are checked only when the engine changes while functional properties are checked for each application (i.e., each new model).

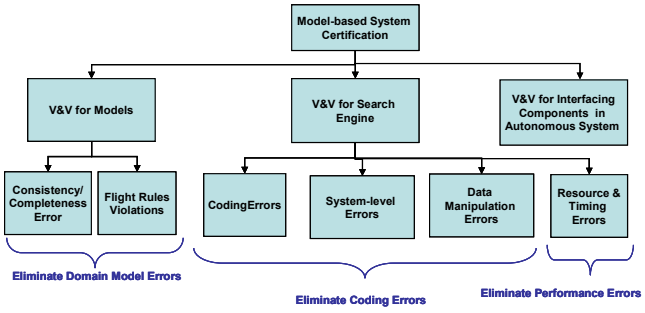

Figure 2. Fault tree analysis for autonomous system.

4. TRADITIONAL VERIFICATION

As we mentioned in the previous section, programming and system-level errors can be present in all elements of an autonomous system. It is particularly relevant in the functional layer where components are very similar to traditional embedded software. However, it is also important in the context of planning and scheduling, even though it might be harder to analyze than functional elements.

It is clear that our previous research on the verification of traditional embedded systems is directly relevant for programming and system-level errors. Therefore, we plan on using static analysis [4], model checking, and, some advanced form of testing. Static analysis is quite good at catching coding errors. Model checking is particularly good at catching system-level errors such as concurrency errors (deadlocks, data races, and so on). Advanced testing can offer several improvements over traditional testing. For example, we can automate the generation of input cases and test oracles using model checking techniques. In particular, we can use a form of symbolic execution to generate inputs for complex data types such as lists and graphs. This is quite interesting when it comes to testing a planner or an executive, which are all about manipulating graphs.

The use of each of these techniques deserves a full paper. In the interest of space, we only describe an experiment in which we try to determine how useful static analysis can be on code implementing a planner. We first start by a brief introduction to static analysis.

Static program analysis

The goal of static analysis is to assess code properties without executing the code. Several techniques can be used to perform static analysis, such as theorem proving, data flow analysis [13], constraint solving [1], and abstract interpretation [5,6]. For this experiment, we use a tool, called PolySpace C++ Verifier [17], which is based on abstract interpretation. The theory of Abstract Interpretation pioneered by Patrick and Radhia Cousot in the mid 70's provides a formal framework for building program analyzers which can detect runtime errors by exploring the text of the program [5,6]. The fundamental result of Abstract Interpretation is that program analyzers obtained by following the framework 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 (NIV)

(2) Access to un-initialized pointers (NIP)

(3) Out-of-bound array access (OBA)

(4) Arithmetic underflow/overflow (OVF)

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

(6) Non-terminating loops (NTL)

(7) Non-terminating calls (NTC)

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, a program analyzer can be used either as a debugger that detects runtime errors statically without executing the program or as a preprocessor that reduces the number of potentially dangerous operations that have to be checked by another validation process (code reviewing, test writing, and so on).

PolySpace C++ Verifier is the first tool implementing Abstract Interpretation techniques that is able to scale up to software systems of industrial size. This tool takes an ISO-compliant piece of C++ code and performs static analysis using sophisticated Abstract Interpretation algorithms. The result is the program in which all potentially dangerous operations have been assigned a color:

Green the operation is safe, no runtime error can occur at this point

Red a runtime error occurs whenever the operation is executed

Black the operation is unreachable (dead code)

Orange the operation is potentially dangerous (runtime error or false alarm).

The goal of our experiment is to assess the selectivity of PolySpace C++ Verifier on C++ programs, especially those implementing our architectural framework.

Example: static analysis of PLASMA's temporal network

We are analyzing parts of the PLASMA planning framework. The goal of this experiment is to assess the efficiency of static analysis techniques in verifying that planning code is free of runtime errors. This sounds like it should be a straightforward task; however, planning code uses quite a bit of dynamic data structures (graphs, constraints), which are usually hard to analyze with static analyzers. Indeed, static analysis does better on code whose behavior is known at compile time, which is not the case for a planner. We report on the lessons of this experiment.

For this experience, we focused on the code implementing temporal networks, and in particular, we concentrated on classes implementing distance graphs. These graphs are routinely used during planning. The first lesson was learned when the analyzer performed a phase similar to the semantic analysis performed by compilers. Basically, at this stage, the analyzer parses the code, checks its type correctness, and, most importantly, checks its conformance to the C++ standards. Not unexpectedly, the analyzer was much stricter than the gcc compiler used to compile the code (even though the code compiles under gcc 4.0.0, which is close to implementing the complete C++ standards). This result usually puts off developers, who dismiss it thinking that the analyzer is overly picky. However, it actually shows that the code is not highly portable since it depends on idiosyncrasies of a particular compiler. In the absence of a compiler implementing the C++ standards, this can be mitigated by using several compilers (the idea being that each compiler might compensate the shortcomings of another one).

In general, static program analyzers perform whole program analysis, meaning that they start their analysis from a main routine and they analyze the code based on the call graph rooted at the main. This simplifies the job since the main provides an environment that restricts the behavior of the code. However since we need to take into account the fact that the system may be reconfigured, we want to analyze the generic parts of the system (the planner, the executive) in a generic manner. Fortunately, the verifier offers the option of analyzing code one class at a time. The way it works is that the analyzer generates for each class a main that implement a universal environment. This allows us to obtain general results. However, general results are typically worse than contextual results (see next paragraph). This is acceptable if one can parameterize the results base don the shape of the inputs. For that, we need access to the generated main routines, which is actually not the case for the moment. So, the second big lesson is that to implement a truly compositional analysis we need to control the contextual information of the analysis.

We now present the results obtained during analysis. First, we should remark that the analysis did not reveal any red errors. Therefore, either the code has no errors or the analysis has precision problems. We lean towards the latter explanation. A good indication of the imprecision of the analysis is given by the rate of oranges (recall that oranges might be errors or false positives). The rate varies from 50% to 13% of oranges depending on the type of the error. A rate of 50% is an awful result; however, it was obtained on the class of overflow errors which account for less than 2% of the checks performed by the analyzer. The class of error with a 13% rate represents close to 70% of the number of checks performed by the analyzer. So, overall, we end up with a total rate of less than 20% of oranges. This is high but not uncommon in static analysis. It shows that we need to do a better job at providing contextual information for the analysis.

5. A COMPOSITIONAL APPROACH

Our goal is to design a verification framework that can support reconfiguration. Therefore, we need a compositional approach to verification. This will also help

improve the salability of advanced verification techniques. In any case, it is interesting to remark that the verification property flow is different from the flow of decision. In the architecture, the planner sits at the top and makes system-level decisions. In the verification process, the executive is at the top. As Figure 3 illustrates, the executive is a confluence point for all properties.

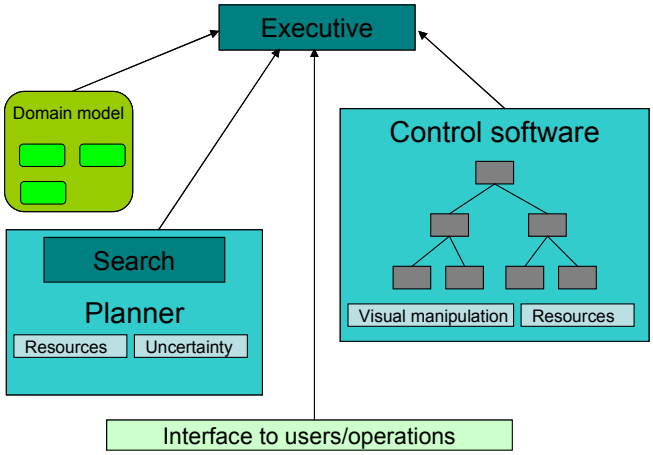

Figure 3. Flow of properties in autonomous system.

Let us consider timing properties. For example, we may wish to establish that execution times fit within the timing constraints of the physical system. From a real-time point of view, the planner is delivering a service to the executive. The planner delivers a high-level plan that needs to be expanded by the executive. So, the execution time of planner (i.e., the search time) needs to be propagated to the executive so that the overall response time can be computed. This example can be extended to other properties, especially the ones expressing safety properties pertaining to the whole system. They naturally flow to the executive. This is where we really have a system-level view of what is happening during execution.

For these reasons, we decided that the best place to demonstrate the use of our compositional framework is at the executive level. In the next two subsections, we first describe the assume-guarantee framework, which provides the theoretical basis for our compositional process, and second, an example of its application to a system for autonomous rendezvous and docking.

The assume-guarantee framework

Compositional verification decomposes the properties of a system into properties of its components, so that if each component satisfies its respective property, then so does the entire system. Components are thus model checked separately. It is often the case, however, that components only satisfy properties in specific contexts (also called environments). This has given rise to the assume-guarantee style of reasoning.

Assume-guarantee reasoning [10,11,16] first checks whether a component M guarantees a property P, when it is part of a system that satisfies an assumption A. Intuitively, A characterizes all contexts in which the component is expected to operate correctly. To complete the proof, it must also be shown that the remaining components in the system (M's environment) satisfy A. This style of reasoning is captured by the following assume-guarantee rule.

$$\frac{\langle A \rangle\ M_1\ \langle P \rangle \qquad \text{(Premise 1)} \\ \langle \mathit{True} \rangle\ M_2\ \langle A \rangle \qquad \text{(Premise 2)}}{\langle \mathit{True} \rangle\ M_1 \,||\, M_2\ \langle P \rangle}$$

Several frameworks have been proposed to support this style of reasoning. However, their practical impact has been limited because they require extensive human input in defining assumptions that are strong enough to eliminate false violations, but that also reflect appropriately the remaining system.

Previous work at NASA Ames has contributed to two main approaches for automating assume-guarantee reasoning. The first approach [8] synthesizes the assumption that a component needs to make about its environment for a given property to be satisfied. The assumption produced is the weakest, that is, it restricts the environment no more and no less than is necessary for the component to satisfy the property. The automatic generation of weakest assumptions has direct application to the assume-guarantee proof. More specifically, it removes the burden of specifying assumptions manually thus automating this type of reasoning.

The second approach provides a model checking framework for performing assume-guarantee reasoning using the above rule in an incremental and fully automatic fashion. To check that a system made up of two components M_1 and M_2 satisfies a property P, the framework automatically learns and refines assumptions Ai for component M_1 to satisfy the property, which it then tries to discharge on component M_2. The framework uses an automata learning algorithm [2] to construct the assumptions for the compositional analysis of the models.

A useful characteristic of this framework is that the generated assumptions are minimal; they strictly increase in size as the learning algorithm progresses, and grow no larger than the weakest assumption for M_1 to satisfy P. Moreover, in our experience, the interfaces between components are small for well designed software. Therefore, assumptions are expected to be significantly smaller than the environment that they represent in the compositional rules, and the cost of assume-guarantee reasoning will be significantly smaller than monolithic (non-modular) model checking, both in terms of time and consumed memory. Recently, this framework has been extended to handle more assume-guarantee rules and more than two components [3].

The above techniques have been implemented in the LTSA model checking tool [14] and have been used in the

example described in the next section among other case studies [9]. These techniques are general; they rely on standard features of model checkers and could therefore easily be introduced in any model checking tool.

Example: an autonomous rendezvous and docking system

A safety critical capability for spacecrafts, also relevant to the CEV, is that of autonomous rendezvous and docking (ARD). Over the last few months, we have been studying ARD from a design perspective in order to understand the characteristics of such a system and to identify verification challenges that it poses. Based on (limited due to export control issues) input from existing ARD systems (for example, ISS) and from publications on such systems, we have created an LTSA model of an example such system.

The model is essentially made up of two types of components: sensors that are used to estimate the absolute or relative position, velocity etc. of the spacecraft, and the modes that constitute a typical ARD system. The ARD software goes through the modes in a sequential manner, and exhibits different behavior in each of these, as its goals are different: for example, the spacecraft uses different sensors (GPS, StarPlanetTracker, InertialNavigation) before it approaches its docking target, but once it is in close proximity, it uses a docking sensor that provides information relative to the target. We have also modeled a component named OrbitalState, which takes readings from the sensors and returns to the ARD mode-related software whether there are good readings from enough sensors to have a good state estimation of the spacecraft for ARD. Our current version of the model is untimed, which means that our verification effort will not concentrate on timing issues. However, we have modeled the requirement that you may have limitations in the amount of time you may be in a specific mode, or within the ARD system. We performed this by including timers that may non-deterministically timeout, in which case the ARD software will need to take the appropriate measures.

We have expressed a number of properties that are typically required from an ARD system. We have verified both local properties of modes and system level properties. We report here the results we obtained by applying learning-based compositional verification to check a property which states that:

"if within CaptureApproachMode two of my sensors fail, it will not be possible to proceed to the next mode"

We have expressed two variants of this property in the LTSA tool. We then decomposed the system in the following way. On the one hand, we included all the mode-related components. Since these are mostly sequential, the composition is fairly small. On the other hand, we included the OrbitalState component with stubbed out sensors to make the model smaller. In checking these properties, monolithic model checking ran out of memory whereas compositional verification succeeded in proving the properties in less than 2 minutes with an assumption of 6 states.

6. RECONFIGURATION AND SYNTHESIS

One of the original goals of this project is to provide an autonomy architecture that fits NASA's vision for exploration. Autonomous systems might be deployed first on the Crew Exploration Vehicle, the Cargo Launch Vehicle, and then on the lunar, and eventually Mars, surface. Re-inventing autonomous systems for each deployment would be prohibitively costly to NASA. Therefore, we recognize from the start that we need to design not only for verification but also for reconfiguration. This will allow us to use the same design and verification techniques and to re-use components across deployments. Moreover, it gives us a convenient means to evolve these systems and keep pace with new technologies.

Figure 4 illustrates the possibilities for reconfiguration in our architecture. First, the most obvious place for reconfiguration is the functional layer where the control software sits. For example, adding a camera to a rover will trigger a reconfiguration in which control software for the camera needs to be added to the functional layer. Generally, such changes trigger changes in both the executive and the planner. The domain model needs to be updated with the constraints specific to the new camera. We can also imagine that new reasoning capabilities (e.g., geometric reasoning) can be added to the planner.

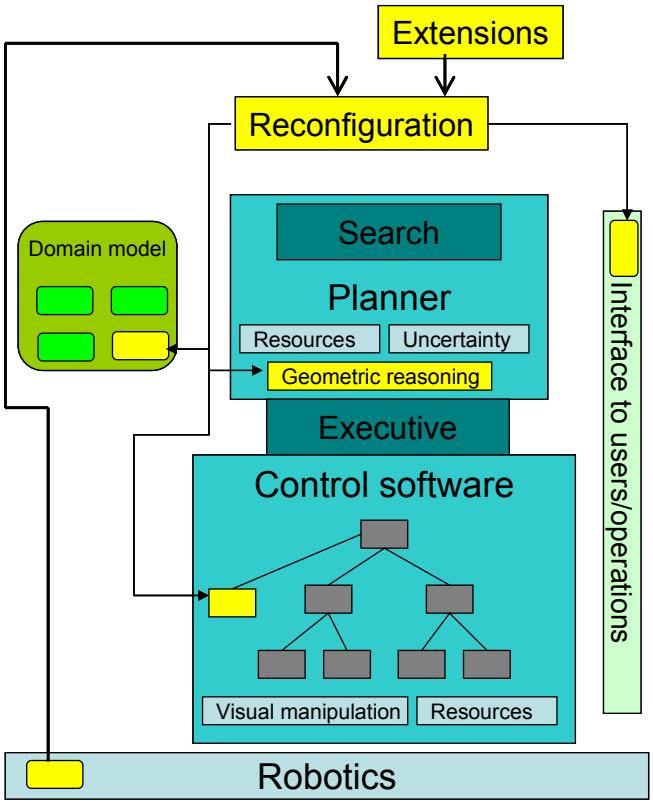

Figure 4. Reconfiguration of autonomy systems.

Reconfiguring autonomous systems is a complicated exercise. If the architecture is not adapted, it can trigger massive changes and result in high cost for the project. Therefore, the architecture needs to be designed for reconfiguration, which means that all aspects of the design need to take into account the possibility of reconfiguration. This is what we have done for verification; we have placed compositional verification at the heart of our verification process. Composition helps re-use verification artifacts in connection with new verification artifacts (for the added components) and efficiently put together the verification of the reconfigured system. In so doing we created a scalable verification mechanism which can deal with reconfiguration.

We actually go one step further in improving the verification process by using synthesis. Synthesis is a formal form of automatic code generation. Generating code automatically allows us to control the shape of the code, and in particular, giving priority to language constructs that facilitates the verification process (described in previous sections). Moreover, our synthesis process itself generates code in a trustworthy manner, which can completely eliminate some error classes.

Introduction to synthesis

The aim is to have certified components. One way of achieving this is to verify that components are free of bugs. Another is to generate the components in an inherently trustworthy manner. We are developing the use of automated code generation (also known as program synthesis) for this. Control software is particularly appropriate for code generation since it can be modeled concisely at a high-level, while the code which implements it tends to be idiomatic.

We now give a brief overview of automated code generation, and the AutoFilter system, which generates Kalman filter-based state estimation code [18]. We then describe the specific adaptations which have been carried out so that AutoFilter generates CLARAty functional layer components.

A code generator takes as input a domain-specific high-level description of a task (e.g., a set of differential equations) and produces optimized and documented low-level code (e.g., C or C++) that is based on algorithms appropriate for the task (e.g., the extended Kalman filter). This automation increases developer productivity and, in principle, prevents the introduction of coding errors.

AutoFilter [7] is a domain-specific program synthesis system that generates customized Kalman filters for state estimation tasks specified in a high-level notation. AutoFilter's specification language uses differential equations for the process and measurement models and statistical distributions to describe the noise characteristics. It can generate code with a range of algorithmic characteristics and for several target platforms. The tool has been designed with reliability of the generated code in mind and is able to automatically certify that the code it generates is free from various error classes (most are programming error, some address functional concerns). Since documentation is an important part of software assurance, AutoFilter can also automatically generate various human-readable documents, containing both design and safety related information.

Due to the modularity of the CLARAty functional layer, we can gradually develop our synthesis capabilities, one module at a time. In the first part of the project, therefore, we have concentrated on modifying AutoFilter's existing synthesis capabilities in order to generate a CLARAty estimator. There are a number of compelling reasons for this.

(1) We already have substantial expertise in synthesizing Kalman filters, so it will enable us to more easily gain experience for the synthesis of other components.

(2) They form an integral part of not just rover software (along with navigators and locomotors), but all vehicles.

(3) They are directly affected by hardware reconfiguration, in particular, sensor configurations

(4) There is a clear notion of model for estimators (namely, the sensor and system models, which are already part of the CLARAty framework).

(5) There is substantial potential variability at multiple levels (different algorithms, different sensor models, mathematical approximations) so it offers scope for reconfiguration.

In CLARAty, estimators are part of the functional layer. The estimator class offers functionality for generic estimators, from which the kalman_filter class is one specialization.

Example: synthesis of a state estimator

The following extensions have been carried out to the AutoFilter program synthesis system in order to let it generate code in the CLARAty framework.

The original system synthesized code from specifications in a number of stages. First, the schemas generated code in the *synt* intermediate language format. This was then transformed into the lower-level *lang*, which is still an internal abstract representation, but depends on the chosen target language (e.g., C++). Finally, the *lang* code is pretty printed in the native format of the target language.

The existing implementation, however, made a number of architectural assumptions which were inapplicable to the representation of filters in CLARAty. In particular, it interleaved the construction of architecture-specific code with the construction of mathematical entities common to all filters. For example, the previous system assumed that a filter obtained its measurements in batch form and that the propagate-update cycle was within a for-loop of predetermined size. Many of these assumptions were implicit and undocumented.

The first change, therefore, was to separate these two parts of the synthesis process and make what is constructed mathematically a more explicit intermediate stage prior to architectural specialization. The first stage now is the construction of a "filter model". This is common to all filters and consists of certain matrices, such as the state transition and measurement matrices, and initial values for the state vector and covariance matrices. It also contains code for computing the residual and propagating the state.

The second stage is the transformation of the filter model into a CLARAty (filter) model, which consists of a system model, sensor model, measurable, and controllable. This involves the construction of the appropriate C++ classes and methods (in the internal *lang* format). Expressions that involve concepts from the filter model must also be transformed. For example, measurement variables are represented as `filter_model(myRover, meas)` in the filter model, and as `measurement.get_mz ()` in CLARAty.

A very simple example of part of an input specification follows. The model describes a simple four wheeled rover where the two left where are mechanically coupled (that is, cannot move independently) as are the right wheels. The state vector, `x`, estimates the left speed, right speed, and the yaw rate of chassis. In the equations, `l` is vehicle length, and `u` represents white noise. We assume that the sensors directly measure the state variables, so the measurement vector, `z`, just reads `x`, subject to noise, `u`. The equations are given, in this case, in discrete rather than continuous form.

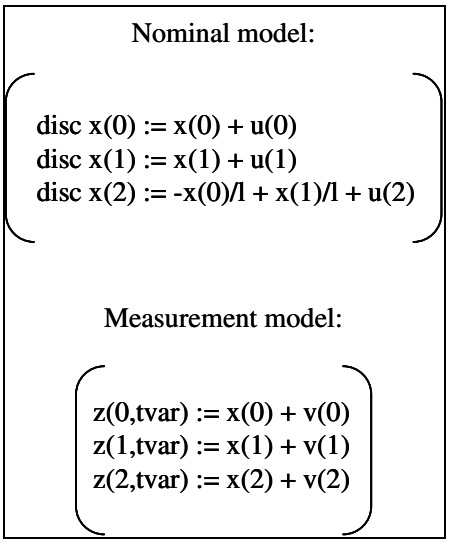

Figure 5. Example of synthesis input specification.

The code that is generated is quite simple in this case, since the filter is linear and all of the derived matrices are constant. For example, Figure 6 shows the system model. Note that some of the variables are declared elsewhere.

```
class Nominal_filter_System_Model :
      public KF_System_Model {
  Vector<double> compute_transition
         (const Vector<double> &state,
          const kf_measurement_t &control)
    {
      Vector<double> state2 = state;
      state2 = get_transition_matrix () * state;
      return state2;
    }
    Matrix<double> get_transition_matrix()
    {
      double tmp[] = {1 , 0 , 0 , 0 , 1 , 0 ,
                      -1 / 1 , 1 / 1 , 0};
      Matrix<double> m(1, 3 - 1, tmp);
      return m;
    }
    Matrix<double> get_process_noise_matrix()
    {
      double tmp[] = {_sigma(0) , 0 , 0 , 0 ,
                      _sigma(1) , 0 , 0 , 0 ,
                      _sigma(2)};
      Matrix<double> m(1, 3 - 1, tmp);
      return m;
    }
 public:
    Nominal_filter_System_Model
                 (Vector<double> &sigma)
    {
      Vector<double> _sigma = sigma;
    }
private:
  Vector<double> _sigma;
};
```

Figure 6. Example of synthesizes kalman filter class.

We are currently making a number of extensions in order to deal with more substantial examples. For example, there is not yet syntax in the specification language for specifying the sources of measurables and controllables or their uncertainty (we currently assume a fixed value). A more interesting extension allows the use of complex expressions for measurements, such as deltas, integrators, and virtual sensors (that is, using one filter as the measurement for another).

7. CONCLUSIONS

In this paper, we have described an approach to verify autonomous systems. We are especially interested in systems based on the three traditional layers (planning, execution, and functional layer) and that are subject to reconfiguration (either because of changes in the physical system they control or because of a need to take advantage of technology evolution).

Our verification approach is highly modular (hence, the need for compositional verification) and exploits the dichotomy found in autonomous systems (these systems consist of models and reasoning engines). The first result is that we target a variety of possible fault classes, from traditional coding errors (run-time errors and system-level errors) to functional errors (such as the violation of flight rules). We also rely on a testing framework which is enhanced by the use of advanced formal verification techniques such as static analysis, model checking, and theorem proving. The second result is that we use synthesis as a fast and robust implementation tool (for code and models) and as a means to augment the precision of our verification results (by favoring constructs amenable to verification).

This paper describes our overall approach and it demonstrates the use of advance verification techniques (static analysis and compositional verification) and synthesis. We illustrate the use of these three techniques with their application to different parts of our autonomy architecture.

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] D. Angluin, Learning Regular Sets from Queries and Counterexamples, Information and Computation, 75(2).

[3] Barringer, H., Giannakopoulou, D., and Pasareanu, C.S. Proof Rules for Automated Compositional Verification through Learning. ESEC/FSE'03 Workshop on Specification and Verification of Component-Based Systems (SAVCBS'03).

[4] G. Brat, and A. Venet, "Precise and scalable static program analysis of NASA flight software". In *Proceedings of the 2005 IEEE Aerospace Conference*, Big Sky, MO, March 2005.

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

[6] 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.

[7] E. Denney, B. Fischer, J. Schumann and J. Richardson, "Automatic certification of Kalman filters for reliable code generation". In *Proceedings of the 2005 IEEE Aerospace Conference*, Big Sky, MO, March 2005.

[8] D. Giannakopoulou, C. S. Pasareanu, H. Barringer, Component Verification with Automatically Generated Assumptions, J. of Automated Software Engineering, 2005.

[9] Giannakopoulou, D., Pasareanu, C.S., Lowry, M., Washington, R. "Lifecycle Verification of the NASA Ames K9 Rover Executive" , ICAPS'05 Workshop on Verification and Validation of Model-Based Planning and Scheduling Systems, Monterey, California, June 2005.

[10] T. A. Henzinger, S. Qadeer, and S. K. Rajamani, "You assume, we guarantee: methodology and case studies", in Proc. of the International Conf. on Computer-Aided Verification (CAV'98). LNCS 1427, pp. 440-451.

[11] C. B. Jones, Specification and design of parallel programs. Information Processing 83: Proceedings of the IFIP 9th World Congress, 1983: pp. 321--332.

[12] A. Jonsson and P. Morris and N. Muscettola and K. Rajan and B. Smith, "Planning in Interplanetary Space: Theory and Practice." Outstanding Application Award winner. Proceedings of the International Conference on Artificial Intelligence Planning and Scheduling, 2000.

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

[14] J. Magee, and J. Kramer, Concurrency: State Models & Java Programs: John Wiley & Sons, 1999.

[15] I.A. Nesnas, A. Wright, M. Bajracharya, R. Simmons, T. Estlin, Won Soo Kim, "CLARAty: An Architecture for Reusable Robotic Software," SPIE Aerosense Conference, April 2003.

[16] A. Pnueli, In Transition for Global to Modular Temporal Reasoning about Programs, in Procedings of the Logic and Models of Concurrent Systems. 1985.

[17] PolySpace: http://www.polyspace.com/.

[18] J. Whittle, and J. Schumann, "Automating the Implementation of Kalman Filter Algorithms," Accepted for publication in ACM Transactions on Mathematical Software (TOMS).

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 to model and evaluate non-stationary, periodic timed discrete event systems. Since then, he has specialized on the application of static analysis to software verification. From 1997 to June 1999, he worked at MCC where he led a project that developed 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 co-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 International Space Station controllers at MSFC, and the*

International Space Station Biological Research Project at the NASA Ames Research Center.

Dr Ewen Denney *(PhD University of Edinburgh, 1999) has published over 30 papers in the areas of automated code generation, software modeling, software certification, and the foundations of computer science. He has been at NASA Ames for three years, where he has mainly worked on techniques for reliable automated code generation.*

Dr Dimitra Giannakopoulou *has been a RIACS research scientist at the NASA Ames Research Center since August 2000. Her research focuses on scalable specification and verification techniques for NASA systems. In particular, she is interested in incremental and compositional model checking based on software components and architectures. She holds a Ph.D. in Distributed Computing from Imperial College, University of London. She has authored over 25 peer-reviewed technical publications. She acts as a regular reviewer for scientific journals and has been a program committee member for several international conferences. More information is available at: http://ti.arc.nasa.gov/people/dimitra/.*

Dr. Ari Jónsson *is a recognized leader in the development of advanced planning and scheduling technology for space operations, and has worked on both theoretical foundations and applications, which include two NASA missions. Currently, he is a member of the MER mission team, serving as the development lead for MAPGEN, which is an interactive activity planning tool for the Spirit and Opportunity rovers.*

Dr. Jeremy Frank *is a research scientist at NASA Ames Research Center. He has extensive experience in planning and scheduling, as well as in constraint reasoning, and has worked on applications ranging from satellite operations scheduling to flight path planning of airborne observatories.*