A Robust Compositional Architecture for Autonomous Systems

Guillaume Brat, Ewen Denney, Kimberley Farrell, Dimitra Giannakopoulos, Ari Jónsson
Research Institute for Advanced Computer Science
NASA Ames Research Center, Mailstop 269-2
Moffett Field, CA 94035
{brat, edenney, Farrell, dimitra, jonsson}@email.arc.nasa.gov

Jeremy Frank
NASA Ames Research Center, Mailstop 269-2
Moffett Field, CA 94035
frank@email.arc.nasa.gov

Mark Boddy, Todd Carpenter
Adventium Labs
100 Mill Place
111 Third Avenue South
Minneapolis, MN 55401 USA
{mark.boddy, todd.carpenter}@adventiumlabs.org

Tara Estlin
Jet Propulsion Laboratory M/S 126-347,
4800 Oak Grove Drive
Pasadena CA 9110, USA ,
tara.estlin@jpl.nasa.gov

Abstract— Space exploration applications can benefit greatly from autonomous systems. Great distances, limited communications and high costs make direct operations impossible while mandating operations reliability and efficiency beyond what traditional commanding can provide. Autonomous systems can improve reliability and enhance spacecraft capability significantly. However, there is reluctance to utilizing autonomous systems. In part this is due to general hesitation about new technologies, but a more tangible concern is that of reliability of predictability of autonomous software.

In this paper, we describe ongoing work aimed at increasing robustness and predictability of autonomous software, with the ultimate goal of building trust in such systems. The work combines state-of-the-art technologies and capabilities in autonomous systems with advanced validation and synthesis techniques. The focus of this paper is on the autonomous system architecture that has been defined, and on how it enables the application of validation techniques for resulting autonomous systems.

TABLE OF CONTENTS

1. INTRODUCTION

Space exploration applications offer a unique opportunity for the development and deployment of autonomous systems, due to limited communications, great distances, and high cost of direct operation. At the same time, the risk and cost of space missions leads to reluctance to taking on new, complex and difficult-to-understand technology. Consequently, there is a pressing need to address the issue of designing robust architecture for autonomous systems and demonstrate a design process that can provide the trust and reliability that is required for manned and unmanned space applications.

In this paper, we describe an ongoing effort to develop a new approach to defining, implementing and maintaining compositional autonomous systems. There are two key elements to the approach. One is a modular compositional autonomy architecture where adaptation to different applications is done in an incremental manner. The other is a testing and validation methodology that allows the certification of new adaptations to be limited to the components and relations that are modified. Together, the two elements will make future autonomy applications more easily constructed and modified, while increasing reliability and reducing cost of reconfiguration and maintenance.

1

0-7803-9546-8/06/$20.00© 2006 IEEE

The work will be grounded in a specific autonomy architecture that integrates the EUROPA planning framework and the functional layer of the CLARAty control architecture. EUROPA supports incremental compositional specification of the states, commands and associated operations rules that define how it may control a given system. CLARAty provides a compositional approach to defining the functional control software that interfaces with the underlying system.

Compositional verification techniques will be used to limit the efforts required to validate and certify a new adaptation. These methods use known properties of unchanged modules to limit validation and certification efforts to changes made. The validation of core system and individual component properties is done with both formal and empirical analysis.

Our approach will enable the increased use of autonomy in future space explorations, thus reducing operations costs and increasing reliability. In addition, the methodology of composable components and assóciated incremental testing and verification, will reduce the cost of system development, maintenance, and reconfiguration.

2. An Architecture for Autonomy

Autonomous systems vary greatly in the representation and reasoning techniques utilized in such systems. Furthermore, the interface between autonomous control and underlying systems can be radically different between architectures. Both of these aspects impact the application of validation techniques to autonomous systems instantiations. Consequently, we define and use a general autonomous systems architecture that uses specific representation and reasoning approaches, combined with a structured well-defined interface to the underlying system. While the architecture provides a basis for defining validation processes and techniques, many of the general notions of how to validate autonomous systems will be applicable to other architectures.

Our architecture uses a constraint-based planning framework called EUROPA (Extendible Uniform Remote Operations Planning Architecture) for the core representation and reasoning. This provides the ability to make decisions about what actions to take so as to achieve mission goals, while ensuring that flight rules and constraints are satisfied. The actions are implemented in the CLARAty framework (Coupled-Layer Architecture for Rover Autonomy), which also provides structured access to system states and sensory information. The architecture is shown in Figure 1.

The decision-making component, implemented in EUROPA, uses the domain model to generate safe plans and decisions that respect flight rules and other constraints on operations. The domain model is a declarative specification of actions and states implemented in the functional layer, along with rules on how these actions can be used. The domain model is compositional, meaning that new actions and rules can be added without changing existing content.

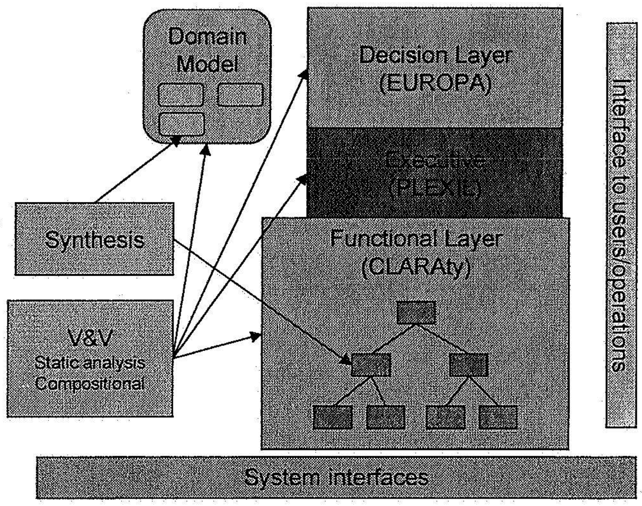

Figure1: Autonomy architecture outline

The executive, implemented in an execution framework called PLEXIL, executes the plans and actions specified by the decision layer. It also monitors the execution, ensuring that the constraints and assumptions in the given plan are satisfied in the system during execution. When deviations occur, the executive can either recover or call on the decision layer to decide how to proceed. . The functional layer is part of the CLARAty framework. It is a set of functional components, arranged in a hierarchy where higher-level components utilize capabilities and services of lower-level components. The functional layer thus provides a compositional approach to implementing interfaces to system functions.

The verification and validation methods are applied to all levels of the architecture. The details are further described here below in the section on validation. The automated synthesis techniques are then used to generate, from high-level specification, both CLARAty functional layer components and the associated specifications in the domain model. This approach, combined with the compositional nature of the overall architecture, enables rapid adaptation and reconfiguration of the autonomy system.

EUROPA – constraint-based planning

The Extensible Universal Remote Operations Planning Architecture (EUROPA) is a model-based planning and scheduling architecture descended from the Remote Agent Planner [Jonsson, et al., 2000]. Users of EUROPA can specify the rules of planning domains using a rich domain description language that supports time, resources, disjunctive preconditions and conditional effects. EUROPA makes extensive use of constraint based representation and reasoning, which allows for more concise representation of planning models, and more efficient reasoning during planning [Frank and Jonsson, 2003]. EUROPA provides support for "foreign function" calls implementing complex constraints such as power consumption and generation.

EUROPA consists of a hierarchy of highly configurable components, supporting the building many types of planners and plan representations. The modeling language, NDDL, contains a small number of elementary entity types, providing ease in modeling. These types can be extended to provide more specialized components, leading to a rich set of modeling primitives. The *plan database* contains the current plan and information about its state. The database provides mechanisms to efficiently query the plan state and modify the plan. Modification leads to inference, which is performed by a *rules engine* module and a *constraint reasoning engine* module. The rules engine determines which rules in the domain description apply after each modification of the plan, and updates the state accordingly. The constraint reasoning module is further broken down into specialized modules that efficiently handle particular classes of constraints, such as temporal constraints. Finally, EUROPA provides interfaces to specialized *heuristics* modules that provide search control to planners.

EUROPA can also be customized to support both long-range deliberative planners as well as short-horizon continuous planners that may operate on the same model. This approach partially resolves problems due to building multiple models in different languages for the same autonomy system (e.g. [Muscettola et al, 1998]). For example, one planner may have a time horizon limited to 5 minutes into the future, and can delay subgoals. Another planner may only plan activities for a hazard avoidance system, leaving other goals to other planners. EUROPA supports customizations of this form by limiting a planners' "view" to a subset of the model. EUROPA also allows multiple planners to modify the same plan concurrently, by providing authority mechanisms indicating what planners may modify.

Automated planning technology such as EUROPA has been utilized as part of on-board autonomy architectures for deep space probes [Jonsson, et al., 2000], robotic rovers [Dias et al. 2003] and free-flying robots [Muscettola et al, 2002].

CLARAty – layered architecture for robotics

Most robotic control systems employ a variant of the Three-Layer Architecture pioneered by Brooks in 1987. CLARAty is an evolution of the three-layer architecture that provides a wide-range of robotic functionality and simplifies the integration of new technologies on robotic platforms. CLARAty is a joint project between the NASA Jet Propulsion Laboratory, NASA Ames Research Center, Carnegie Mellon University and a number of other universities and has been designed specifically for space-based robotic control applications. CLARAty features a Functional Layer of robotic primitives, coupled with a Decision Layer of planning and execution functionality; each of these layers contains a hierarchy of components ranging from the most elementary to the most "intelligent".

The Functional Layer (FL) provides a set of standard, generic robot capabilities that interface to system hardware. These capabilities are organized as a software class hierarchy of robotic components; for example, wheeled-mobility is a subclass of mobility, and individual rover wheel assemblies are child classes. As is natural in object-oriented systems, the interface is separated from implementation. Physical limitations of devices are distinguished from algorithmic limitations. Finally, runtime

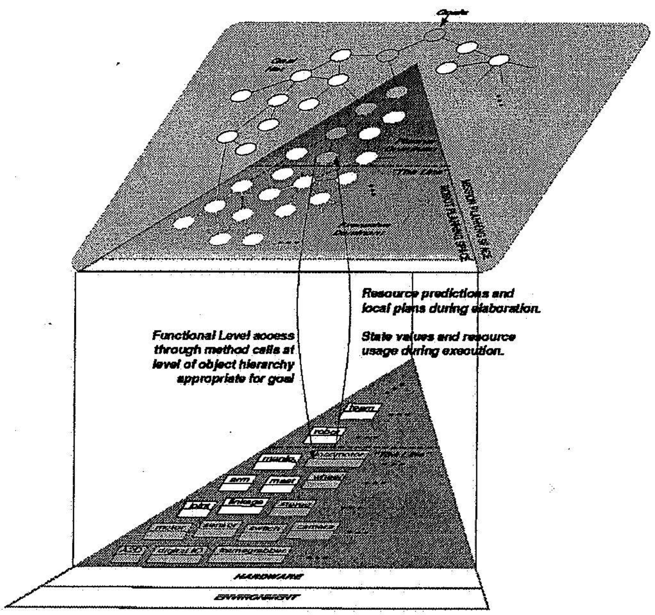
Goals
Goal Net
"The Line"
Functional Level access through method calls at level of object hierarchy appropriate for goal
Resource predictions and local plans during elaboration.
State values and resource usage during execution.
team
robot
manip.
locomotor
arm
mast
wheel
joint
linkage
stereo
motor
sensor
switch
camera
A/D
digital IO
framegrabber
HARDWARE
ENVIRONMENT

Figure 2: CLARAty framework organization

models of devices are incorporated in the Functional Layer.

3. Validation of Autonomous Systems

In the autonomy architecture outlined here, an instantiation of an autonomous system consists of the following elements:

(1) The core EUROPA planning and decision-making framework, which will largely be unchanged between applications and thus can be validated without reference to the specific instantiation.

(2) The CLARAty instantiation for the system in question, which consists of a set of core CLARAty components and the specific components used to operate the system. The core CLARAty components can be

validated separately, while the specific components are validated as part of the instantiation process. Some components may be synthesized, which offers an opportunity for easier validation of those components.

(3) The execution system that links CLARAty and EUROPA and provides monitoring capabilities to ensure that execution does not continue when the assumptions supporting the plan no longer hold. The core execution system is validated once, but online validation and checking techniques can be used to validate specific executable plans.

(4) The domain model describing possible actions and the flight rules governing those actions and the related system states. Model validation is a key element of ensuring that the autonomy system instantiation is robust and safe.

(5) The properties that should hold for the system and various components. These define the criteria for validation of the system.

The validation of an instantiation thus involves validating core architecture systems, instantiation-specific CLARAty components, the domain model used by the planner and executive, and finally, the overall properties for the instantiated system. To tackle this, we apply three kinds of techniques. *Model-based validation*, using compositional verification, can be applied to core software as well as special-purpose components. In addition, compositional techniques allow us to verify system-level properties from component properties. *Static analysis* is a powerful technique to directly analyze software code, without requiring a formal modeling of the software components and properties. Finally, *automated synthesis* techniques allow us to generate instance-specific elements from high-level specifications. In addition to simplifying the process of implementing instantiations, synthesis offers an additional level of validation by generating provably correct code.

Compositional verification

Model-based verification techniques use exhaustive search through possible execution trajectories to verify desired system properties. While these techniques can provide the formal validation desired for our autonomous systems, they suffer from state-space explosion, making them impractical to use. In addition, they do not lend themselves to incremental validation, as we desire to do for instantiations of autonomous systems.

To address these issues, we turn to compositional verification techniques. The basic idea is as follows: Consider a system consisting of two components X and Y. The desire is to prove that a property P is satisfied by the overall system X|Y. In compositional verification, this is done by identifying an intermediate property A, called an assumption, and using that to split the validation into two smaller problems. The first part is to prove that X satisfies A, and the second part is to prove that given A, Y satisfied P.

This notion can be utilized in different ways. If the X and Y components are already validated, it is likely that the assumption A is already known and the compositional verification techniques can be applied directly. This is likely to be the case in situations such as where new CLARAty components are being added on top of existing ones. The properties of the core components are known and validated, so the new components can be validated against these proven assumptions.

A more interesting case is when the assumption A is not known. To address that, we turn to techniques for automatically generating such assumptions from the components X and Y. Recently developed techniques make this possible, and allow both the generation of a weakest valid assumption for a given component, and an easier-to-find stronger but also valid assumption that is still satisfied and suffices to prove the overall property P.

As outlined above, we will rely heavily on these compositional verification techniques in our work, both to address computational cost issues, and to enable incremental validation of autonomy system instantiations.

Static analysis

The goal of static program analysis is to assess properties of a program without executing the program. Several techniques can be used to perform static analysis. Theorem proving, data flow analysis, constraint solving, and abstract interpretation are among the most popular. Generally speaking, a static program analyzer infers properties about the execution of the program from its text (the source code) and a formal specification of the semantics of the language (which is built in the analyzer). Static program analyzers are in general excellent to detect runtime errors.

Runtime errors are errors that cause exceptions at runtime. Typically, in C, either they result in creating a core dump or they cause data corruption that may cause crashes. The main classes of runtime errors are accesses to un-initialized variables, accesses to un-initialized pointers, out-of-bound array accesses arithmetic underflow/overflow, invalid arithmetic operations, non-terminating loops, and non-terminating calls.

In general, static program analyzers aim at checking all execution paths, sometimes at the cost of incompleteness (i.e., impossibility of determining the safety of all operations with exact precision). In other words, the analyzer can raise false alarms on some operations that are actually safe. However, if the analyzer deems an operation safe, then errors cannot occur on any execution path. The program analyzer can also detect certain runtime errors which occur every time the execution reaches some point in the program.

Traditionally, there are two complementary uses of a program analyzer:

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

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

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

Automated 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.

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 [Denney et al, BigSky Boondoggle 2005] 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

4. Concluding Remarks

The work described in this paper is an ongoing effort. The architecture has been defined and an initial version has been implemented. The process for validating new instantiations has been defined and initial efforts are underway to apply static analysis, compositional verification and automated synthesis to parts and aspects of the autonomy architecture.

The autonomy software has been adapted to operating JPL's FIDO rover in simulation. As expected, the integration of EUROPA and CLARAty has been straightforward, assisted by the use of the PLEXIL execution framework. At this point, the planner domain model is constructed manually from the specification of the CLARAty interface for FIDO control. Future efforts in the use of synthesis will focus on automatically generating domain model information from functional layer specifications, but the expectation is that engineering experts will still refine the domain model to specify the desired constraints and flight rules.

Static analysis is being applied to modules of the EUROPA framework, providing initial analysis for the current implementation. In the near future, this work will be extended to other modules of EUROPA and to components of the CLARAty framework.

Compositional verification techniques are being tested on models of autonomous rendezvous and docking systems. The results of this work are very promising, having demonstrated the ability to generate small assumptions that in turn enable very fast validation of system properties. These results are in stark contrast to the large amount of time and computing resources needed to apply traditional model checking methods to the full model without the benefit of decomposition.

Automated synthesis techniques for generating Kalman filters from formal specifications have been adapted to generate Kalman filter components for CLARAty. Since such filters play a key role in interpreting sensor information for states like rover location, this is a significant step forward.

To summarize, the effort outlined here has only recently been started, but even at this early point in time, the results are promising. The goal of this work is to provide a robust verified core autonomy architecture, along with the process and tools needed to adapt it to spacecraft operations applications, such that the instantiated autonomous control system is validated and robust.

References

[1] 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.

[2] B. Fischer, J. Schumann, "AutoBayes: A System for Generating Data Analysis Programs from Statistical Models. Journal of Functional Programming, Vol. 13, No. 3, May 2003, pp. 483-508.

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

[4] P. Gluck, G. Holzmann, "Using Spin Model Checking for Flight Software Verification," Procedding of 2002 Aerospace Conference, March 2002.

[5] D. Giannakopoulou, C. Paraseanu, H. Barringer, "Component Verification with Automatically Generated Assumption," Journal of Automated Software Engineering, Vol. 11, Kluwer, 2004.

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

A. Venet "Non-uniform Alias Analysis of Recursive Data Structures and Arrays." In SAS'02, LNCS 2477, pp. 36-51, 2002.

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

[8] Venet, "A Scalable Nonuniform Pointer Analysis for Embedded Programs," Proceedings of the International Static Analysis Symposium, SAS 04, Verona, Italy. LNCS 3148, Pp. 149-164, Springer 2004.

[9] J. R. Buch, E.M. Clarke, K.L.McMillan, D.L. Dill, J. Hwang, "Symbolic Model Checking: 10E20 states and beyond," In LICS, 1990.

[10] J. Frank and A. Jonsson, "Constraint-Based Attribute and Interval Planning." In the Journal of Constraints, vol. 8, no. 4, 2003.

[11] N. Muscettola and P. Nayak and B. Pell and B. Williams, "Remote Agent: To Boldly Go Where No AI System Has Gone Before." Artificial Intelligence,103(1-2), 1998.

[12] J. Frank, "Bounding the Resource Availability of Partially Ordered Events with Constant Resource Impact", In Proceedings of the 10th International Conference on Principles and Practices of Constraint Programming, 2004.

[13] 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.

[14] V. Verma and T. Estlin and A. Jonsson and C. Pasareanu and R. Simmons. Plan Execution and Interchange Language (PLEXIL) for Executable Plans and Command Sequences. Proceedings of the 9th International Symposium on Artificial Intelligence, Robotics and Automation in Space, 2005

[15] M. Bernardine Dias and S. Lemai and N. Muscettola. A Real-Time Rover Executive Based on Model-Based Reactive Planning. Proceedings of the 7th International Symposium on Artificial Intelligence, Robotics and Automation in Space, 2003

[16] N. Muscettola and G. Dorais and C. Fry and R. Levinson and C. Plaunt. IDEA: Planning at the Core of Autonomous Reactive Agents. Proceedings of the 3d International NASA Workshop on Planning and Scheduling for Space, 2002

Biographies

To be filled in later.