

# Lifecycle Verification of the NASA Ames K9 Rover Executive

Dimitra Giannakopoulou<sup>1,3</sup> Corina S. Pasareanu<sup>2,3</sup> Michael Lowry<sup>3</sup> and Rich Washington<sup>4</sup>

(1) USRA/RIACS

(2) Kestrel Technology LLC

(3) NASA Ames Research Center, Moffett Field, CA 94035-1000, USA

{dimitra, pcorina, lowry}@email.arc.nasa.gov

(4) Google Inc., 1600 Amphitheatre Parkway, Mountain View, CA 94043, USA  
rwwashington@google.com

## Abstract

Autonomy software enables complex, robust behavior in reaction to external stimuli without human intervention. It is typically based on planning and execution technology. Extensive verification is a pre-requisite for autonomy technology to be adopted in high-risk domains. This verification is challenging precisely because of the multitude of behaviors enabled by autonomy technology.

This paper describes the application of advanced verification techniques for the analysis of the Executive subsystem of the NASA Ames K9 Rover. Existing verification tools were extended in order to handle a system the size of the Executive. A divide and conquer approach was critical for scaling. Moreover, verification was performed in close collaboration with the system developers, and was applied during both design and implementation. Our study demonstrates that advanced verification techniques are crucial for real-world planning and execution systems. Moreover, it shows that when verification proceeds hand-in-hand with software development throughout the lifecycle, it can greatly improve the design decisions and the quality of the resulting plan execution system.

## Introduction

Verification is essential for planning and execution technology to be adopted in high-risk domains. This paper is a demonstration of how advanced verification techniques were used on a plan execution system in the domain of Mars rovers.

The work presented here has been performed as part of a project at NASA Ames. The objective of the project is to develop and demonstrate the use of advanced verification techniques for detecting integration problems in the design and implementation of NASA autonomy software. Traditional testing is hard for autonomous systems due to high complexity and unpredictable environments. Moreover, integration problems are very difficult to detect, and are typically checked during integration testing, i.e. *after* the entire system has been implemented. At that stage, fixing such problems may require significant time

and effort since they may involve major changes in the architecture of the system, and possible re-implementation of a large part of it. Therefore, we believe that the verification of a safety critical system should be addressed *as early as possible during its design*, and should go hand-in-hand with later phases of software development.

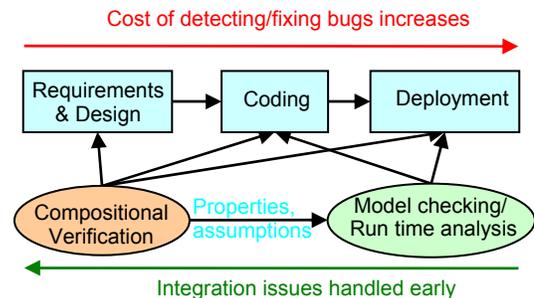


Figure 1. Compositional verification throughout the software lifecycle

Our work advocates the use of a combination of formal analysis techniques and testing to analyze autonomous systems throughout their lifecycle. Note that the size of such systems is beyond the capabilities of existing (formal) verification technologies. The combinatorics of the possible behavior paths is also beyond the capabilities of testing alone as a verification strategy. To address these issues, our work has the following goals (see Figure 1):

- Apply, extend, and integrate verification tools at different phases of software development, i.e. at design and implementation phases of the software lifecycle.
- Use divide and conquer techniques that decompose the verification of a software system into manageable verification of its components, to achieve scalability in (formal) software verification. The verification of the components can then be composed to verify the entire system, hence the name ‘compositional verification’.
- Use design level artifacts to subsequently guide the implementation of the system and to enable more efficient verification at the source-code level.

The main contribution of the work discussed here is the development of *compositional* verification and validation techniques for autonomy software, and their integration

with other verification techniques for an integrated lifecycle approach to verification. This approach to autonomy verification has been empirically validated and quantified through their application to a significant autonomy software system: the Executive of the K9 experimental Martian Rover developed at NASA Ames, a concurrent software system of 35 K lines of C++ code.

Our ‘divide and conquer’ approach achieved an order of magnitude improvement in performance, over monolithic verification techniques. The performance was measured in terms of time and memory consumption during model-checking [11]. The verification effort was performed in close collaboration with the designers and developers of the system throughout its lifecycle, and consisted of the following steps:

- **Design level modeling.** Detailed design level models were created. The models describe the overall concurrent architecture of the executive (coordinating and monitoring components) and advanced features that allow for increased autonomy (e.g. alternate plan execution, support for concurrent activities, separation of start and end constraints). A comprehensive set of requirements were also created (both English and formal descriptions). The requirements capture key concurrency and plan execution properties. We believe that both the models and the requirements could be successfully re-used for the design and analysis of future advanced executives.

- **Design level analysis.** Model checking techniques were used for the *exhaustive verification* of design models against requirements. We developed automated compositional reasoning techniques to increase the scalability of model checking. These techniques were applied to the analysis of the design models, achieving a 10x improvement – in terms of time and memory consumed - over monolithic (non-compositional) model checking.

- **Code level analysis.** Although design level verification is important, subsequent code-level verification is needed to guarantee that the implemented system indeed satisfies the properties. We developed a methodology for using the design level artifacts for the compositional verification of source code, while improving the performance of verification tools at the code level. For code-level analysis of individual components, we used two complementary techniques. We used software model checking, where we obtained a 3x improvement in terms of consumed memory. We also investigated the use of automated testing technology (a combination of run-time verification, to monitor the execution of the system, and automated test input generation, to systematically generate test inputs up to a given size).

As a result of design level analysis, we discovered several integration problems. Based on these results, the developer *changed the design* of the Executive, resulting in a simplified architecture with increased modularity. We

analyzed both versions of the Executive. While for the first version, we created the models after coding (partly by reverse engineering), for the second version, we created the design models before coding. During this process, we re-used component models from the previous version.

This experiment convinced developers that there is considerable benefit in using verification techniques at the design level where several integration issues were identified and corrected. Models were also used to quickly experiment with design decisions. In addition, it was acknowledged that the later in the lifecycle design errors are identified, the more costly it is to fix them, especially if such errors require major design changes in the system. Our techniques are directly applicable to the analysis of other complex autonomous systems. This is particularly so for systems that make the notions of components explicit (e.g. the Mission Data Systems [7]), since our compositional techniques take advantage of the modular architecture of the system.

Our work builds on a previous effort [15], where we compared the performance of tools based on formal methods to traditional testing for the code-level analysis of the original version of the code of the K9 Executive. The study presented in [15] compares verification tools on industrial-size autonomy software (see also [15] for a presentation of related work). What differentiates the work presented here is 1) the integrated application of techniques *throughout* the lifecycle and 2) the development and application of novel *compositional* techniques as a way of addressing scalability issues.

The rest of the paper is organized as follows. In the next section we describe the architecture of the K9 Rover Executive and the design changes made as a result of our analysis. We then describe the compositional technologies that were used for design- and code-level verification. We follow with a discussion on the design-level modeling of the Executive; we also describe the properties that were checked and the results obtained from the lifecycle verification of the Executive. Finally, we close the paper with conclusions and some plans for future work.

## K9 Rover Executive

The NASA Ames K9 Rover is an experimental platform for autonomous wheeled vehicles called rovers, targeted for the exploration of a planetary surface such as Mars. K9 is specifically used to test out new autonomy software, such as the Rover Executive [16]. Previous to the development of autonomy software, planetary rovers were controlled through sequences of detailed, low level commands uploaded from Earth. The Rover Executive provides a more flexible means of commanding the rover through the use of high-level plans, which the Executive

---

interprets and executes in the context of the execution environment. The Executive monitors execution of primitive actions, and performs appropriate responses and cleanup when execution fails. The Rover executive is a software prototype written in C++ by researchers at NASA Ames (approximately 35K lines of C++ code).

Plans are programs written in a language that specifies actions and constraints on the movement, experimental apparatus, and other resources of the Rover. The operational semantics of the language takes into account the possibility of failure of atomic-level command actions. The structure of a plan is a hierarchy of actions that the Rover must perform: each plan is a *node*; a node is either a *task*, corresponding to a primitive action, a (possibly concurrent) *block*, corresponding to a logical group of nodes, or a *branch*, representing a conditional branch within the plan. In addition, *floating branches*, which are plan fragments triggered dynamically, may be inserted into the plan, allowing a limited form of run-time plan modification. The plan language allows the association of each action with a number of state or temporal start, maintenance, and end conditions, which must hold before, during, and on completion of the action execution, respectively.

In contrast to programming language interpreters, the executive is expected to be robust under many plan primitive execution failures. The operational semantics for recovery from primitive failures are extensive.

### Architecture of the K9 Executive

Figure 2 illustrates the architecture of the executive, prior to the design changes that the developer made partly as a result of our analysis. The executive has been implemented as a multi-threaded system, made up of a main coordinating component named *Executive*, components for monitoring the state conditions *ExecCondChecker*, and temporal conditions *ExecTimerChecker* - each further decomposed into two threads - and finally an *ActionExecution* component that is responsible for issuing the commands to the Rover. Synchronization between components is performed through mutexes and condition variables (implemented using the Posix libraries).

During the design level analysis of the executive, we discovered several concurrency problems with the inter-thread communication between different components of the executive. To eliminate these problems, the developer changed the architecture of the system, as illustrated in Figure 3. The main change is the use of an *Event Queue* as a communication mechanism between the *Executive* and the rest of the components. As a result, the communication between different components became much simpler and less prone to errors. E.g., our analysis of the first version revealed a concurrency problem (i.e. race condition) with a variable shared between the *ExecCondChecker* and the

*Executive*. This shared variable was eliminated in the second version, its role being replaced by the *Event Queue*.

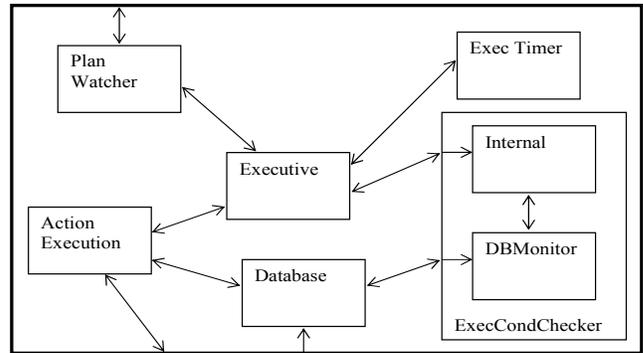


Figure 2. Original architecture of the executive

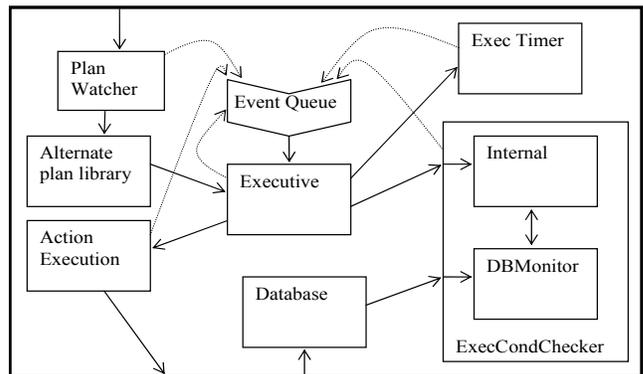


Figure 3. Updated architecture of the executive

Besides the changed architecture, the “new” executive presents several functional changes from the original one: added support for concurrent activities and “floating branches” (dynamically inserted branches and plan fragments), separation of temporal constraints from other pre- and post-conditions, and addition of relative temporal constraints to arbitrary actions within the plan. The high-level changes are summarized below:

- The Executive was changed to be event-based. An *Event Queue* was added. Both *ExecTimer* and *ExecCondChecker* were simplified to return all events, leaving the task of processing and ignoring events to the Executive. The Executive acts on an “execution context” - a data structure representing the current state of execution. This execution context was augmented to support concurrent activities and floating branches. The design documents were changed from state diagrams into event-processing loops.
- The *ActionExecution* was changed to support parallel execution threads.
- Simpler usage and removal of condition variables: there were a number of places in the first version where condition variables were used to coordinate information passing between modules (such as the *ExecCondChecker* and the *Executive*). By simplifying the information flow

(via the event queue), tight coordination is no longer necessary.

- Design simplicity over code re-use: there were a few places in the first version where code, locks, or features were re-used for conciseness. However, in some cases this made the design much more convoluted. For example, the return value of *ActionExecution* was routed through the Database and then the *ExecCondChecker* for uniformity with other conditions being checked during execution. However, this makes the information flow in the system circuitous, unclear, and error-prone.

## Design-Level Verification

At design level, we use verification techniques that *exhaustively* explore all the possible executions of a system. Although exhaustive exploration is typically intractable at the code level, designs tend to be more abstract, making them more amenable to efficient verification. Specifically, we use model checking: given some formal description of a system and of a required property, model checking [11] *automatically* determines whether the property is satisfied by the system. Since scalability can also be an issue at the design-level, we enhance model checking with compositional techniques.

In this section, we describe the LTSA verification tool for design-level software analysis. We also present the compositional techniques with which we extended the LTSA.

### The Labeled Transition System Analyzer (LTSA)

The LTSA [8] is an automated tool that supports Compositional Reachability Analysis (CRA) [9] of a software system based on its architecture. In general, the architecture of a concurrent system has a hierarchical structure. CRA incrementally computes and abstracts the behavior of composite components based on the behavior of their immediate children in the hierarchy.

The input language FSP (Finite State Processes) of the tool is a process-algebra style notation with Labeled Transition Systems (LTS) semantics. An LTS is a finite-state machine whose transitions are labeled by *actions*, representing the internal and communication events in which a component may engage. LTSs are composed by synchronization of common actions and interleaving of local, internal actions. Safety properties are expressed as LTSs with extended semantics, and are treated as ordinary components during composition. Properties are combined with the components to which they refer. They do not interfere with system behavior, unless they are violated. In the presence of violations, the properties introduced may reduce the state space of the (sub) systems analyzed.

The LTSA framework treats components as open systems that may only satisfy some requirements in specific contexts. By composing components with their properties, it postpones analysis until the system is closed, meaning that all contextual behavior that is applicable has been provided. The LTSA tool also features graphical display of LTSs, interactive simulation and graphical animation of behavior models, all helpful aids in both design and verification of system models.

### Compositional Analysis

We extended the LTSA model-checking tool with several analyses done at the component level. The aim was to address the “state-space explosion” problem; this term describes the main limitation of model checking, which is that it requires storing the entire explored system states in memory, which is impossible for most realistic systems.

Compositional verification advocates a “divide and conquer” approach to addressing state-space explosion: properties of the system are decomposed 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 [12,13,14] 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.

$$\begin{array}{l} \langle A \rangle M_1 \langle P \rangle \quad (\text{Premise 1}) \\ \langle \text{True} \rangle M_2 \langle A \rangle \quad (\text{Premise 2}) \end{array}$$

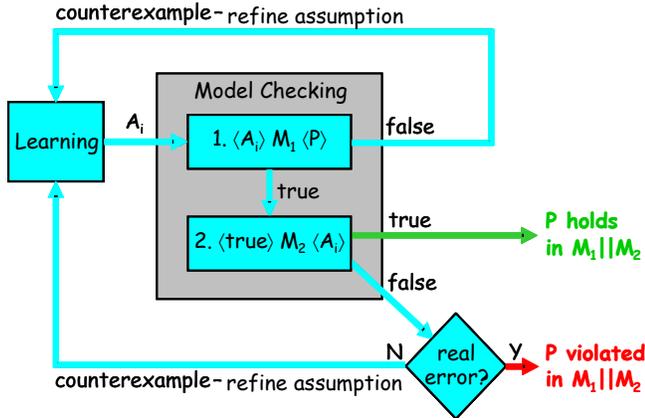

---

$$\langle \text{True} \rangle M_1 \parallel M_2 \langle P \rangle$$

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.

In previous work, we developed several novel techniques that automate assume-guarantee reasoning. We implemented these techniques in the LTSA tool and used them in the analysis of the design models of the Rover Executive. We should note that our techniques are general; they rely on standard features of model checkers and could therefore easily be introduced in any model checking tool.

In [2], we present an approach to synthesizing 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.



**Figure 4. Framework for assume-guarantee reasoning**

The technique presented in [2] does not compute partial results, meaning no assumption is obtained if the computation runs out of memory, which may happen if the state-space of the component is too large.

We address this problem in [1], where we present a model checking framework for performing assume-guarantee reasoning using the above rule in an incremental and fully automatic fashion. The framework is illustrated in Figure 4. To check that a system made up of two components  $M_1$  and  $M_2$  satisfies a property  $P$ , our framework automatically learns and refines assumptions  $A_i$  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 [17] to construct the assumptions for the compositional analysis of the models.

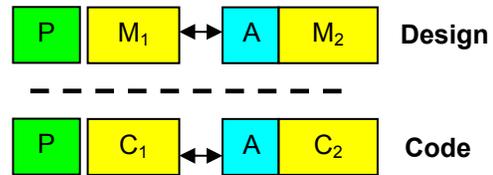
At each iteration  $i$ , the learning algorithm is used to build an approximate assumption  $A_i$ , based on querying the system and on the results of the previous iteration. The two premises of the assume-guarantee rule are then checked. Premise 1 is checked to determine whether  $M_1$  guarantees  $P$  in environments that satisfy  $A_i$ . If the result is false, it means that this assumption is too weak, and therefore needs to be refined with the help of the counterexample produced by checking premise 1. If premise 1 holds, premise 2 is checked to discharge  $A_i$  on  $M_2$ . If premise 2 holds, then according to the assume-guarantee rule  $P$  holds in  $M_1 || M_2$ . If it doesn't hold, further analysis is required to identify whether  $A_i$  is indeed violated in  $M_1 || M_2$  or whether

$A_i$  is stronger than necessary, in which case it needs to be refined. The new assumption may of course be too weak, and therefore the entire process must be repeated. For finite state systems, this process is guaranteed to terminate. In fact, it converges to an assumption that is necessary and sufficient for the property to hold in the specific system.

A useful characteristic of our 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, we have extended our frameworks to handle more assume-guarantee rules and more than two components.

## Code Level Verification

In this section, we describe our methodology for using the artifacts of the design level analysis to decompose the verification of the implementations [4]. For source code verification we investigated two technologies: software model checking (achieves exhaustive verification at the price of scalability) and run-time verification (achieves scalability at the price of exhaustiveness).



**Figure 5. Using design level assumptions for source code verification**

To address the scalability issues associated with software verification, our approach integrates assume-guarantee reasoning of concurrent systems at the design and at the implementation level (see Figure 5). At the design level, the architecture of a system is described in terms of components and their behavioral interfaces modeled as LTSs. Design models are intended to capture the design intentions of developers, and allow early verification of key integration properties. For example, consider a system that consists of two design level components  $M_1$  and  $M_2$ , and a property  $P$ , describing the sequence of events that the system is allowed to produce, or equivalently the bad behaviors that the system must avoid.

To check in a scalable way that the composition of  $M_1$  and  $M_2$  satisfies  $P$ , we use the assume-guarantee frameworks described in the previous section. We expect that, with the feedback obtained by our verification tools, the developers of the system will correct their design models until the property is achieved at the design level. At that stage, our frameworks will have automatically generated an assumption  $A$  that is strong enough for  $M_1$  to satisfy  $P$  but weak enough to be discharged by  $M_2$ .

To then establish that the property is preserved by the implementation, our approach uses the automatically generated assumption  $A$ , to perform assume-guarantee reasoning at the source code level. The implementation is decomposed as specified by the architecture at the design level (i.e. components  $C_1$  and  $C_2$  implementing  $M_1$  and  $M_2$ , respectively), and we establish that  $C_1$  composed with  $C_2$  satisfies  $P$  by checking that  $C_1$  satisfies  $P$  under assumption  $A$ , and by discharging  $A$  on  $C_2$ . If both these checks return true then the property is preserved by the implementation. Otherwise, the counterexample(s) obtained expose some incompatibility between the models and the implementations, and are used to guide the developers in correcting the implementation, the model, or both. For the actual verification of source code, we investigated two technologies: software model checking and run time verification, which are described below.

### Software Model Checking

We used the Java Pathfinder software model checker (JPF) [18] developed at NASA Ames. JPF is an explicit state model checker that analyzes programs written in Java (an implementation for C++ analysis is currently being developed). JPF checks for deadlocks and assertion violations. JPF is built around a special purpose Java Virtual Machine (JVM) that allows Java programs to be analyzed. JPF supports depth-first, breadth-first and several heuristic search strategies to search systematically explore the state spaces of the analyzed programs.

### Run Time Verification and Automated Test Input Generation

For the first version of the Executive we focused on checking implementations using compositional reasoning and software model checking tools. In the second version we experimented with methods that provide even more scalability at the price of exhaustiveness. Specifically we investigated the use of lighter-weight analysis techniques, i.e. run time verification, for the compositional analysis of the second version of the executive.

Run time verification is an advanced testing technique that provides a means for constructing oracles that examine not just the output and interfaces of a system, but the internal computational status of the system. In run time verification, a program is instrumented to emit events which are then monitored to check for conformance to

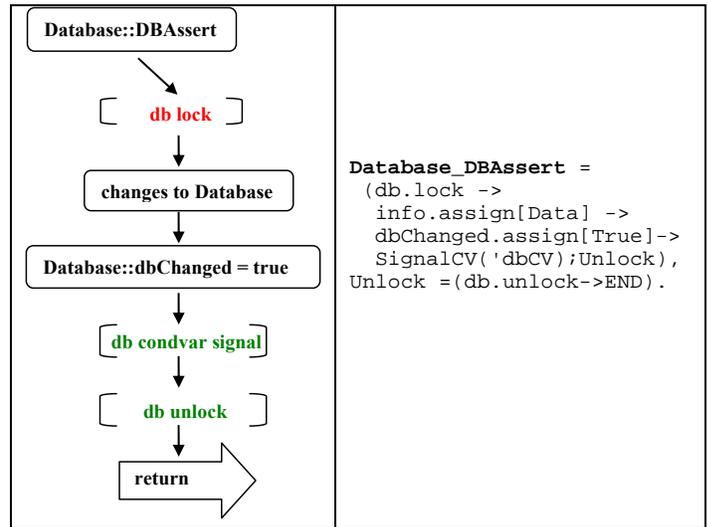
formalize requirements, either stated as temporal logic assertions, or as specialized algorithms looking for common errors, such as deadlocks and data races.

For the analysis of the Rover Executive, we used the Eagle temporal logic runtime verification framework [5]. In order to generate different executions for thorough testing, we used automated test input generation techniques developed at NASA Ames to create all (non-isomorphic) input plans up to a pre-defined size [5]. Given a formal description of the inputs to a system, the test input generation techniques combine symbolic execution, model checking and heuristic search to systematically search and generate the input state space and to achieve full coverage of the input specification.

## Modeling and Analysis of the Rover Executive

### Initial Modeling

We produced abstract models of the Rover Executive that contained enough information – but at a higher level – to allow us to study architectural properties of the system and detect potential integration problems. The developer of the executive initially described the architecture of the system as a hierarchy of threads as illustrated in Figure 2. Moreover, he provided some design documents in his own, ad-hoc flowchart-style notation, describing the main functionality of the threads in the Rover Executive.



**Figure 6. Original design (left) and corresponding FSP model (right) produced for a method in the database**

These documents were produced “after the fact”, meaning after a first implementation of the Rover was available. It took the developer *only a few hours* to produce these documents. Moreover, he found the diagrams that we produced of the architecture of the system helpful, and

subsequently maintained it for communicating the structure of the system to his collaborators.

Figure 6 illustrates the original design provided to us and the corresponding (FSP) model that we produced. In the model, *Data* is the domain of values for variable *info*. *SignalCV* is the method that needs to be called to signal a specific condition variable, in this case *dbCV*. *Unlock* is simply a state alias – mutex *db* must get unlocked after signaling *dbCV* and before returning.

We made a systematic effort to keep the architecture explicit in the model. Each thread has a unique instance name – the name of the thread in the architecture – which prefixes all the actions in its behavior, thus clearly differentiating its behavior from that of other threads in the system. This was achieved by the instantiation operator that the LTSA tool provides. Moreover, communication points were modeled by binding the associated actions, captured by the renaming operator of the LTSA. The resulting model was approximately 600 lines of FSP code that had a very close correspondence to the design documents provided by the developer.

### Modeling the New Executive

As mentioned, the developer changed the design of the Executive, partly as a result of our analysis. We created two new models for the design level analysis of the new executive. Model 1 (~800 lines of FSP code) captures the new architecture of the executive, the queuing mechanism and the detailed event handling for block and task nodes. Model 2 (~900 lines of FSP code), which captures synchronous and asynchronous execution of floating branches.

#### Model 1: Queuing and Event Handling

We reused from our previous model the FSP encoding of the functionality of mutexes and condition variables. We added a model for the FIFO *Queue* and the event handling mechanism in the *Executive* and we updated the *ExecCondChecker* and *ActionExecution* models according to the new design, as illustrated in Figure 3.

#### Model 2: Floating Branch Execution

We extended Model 1 to handle the execution of floating branches. This execution is triggered by pre-defined conditions. Floating branches can be synchronous (i.e. triggered at action transitions within the plan) or asynchronous (i.e. monitored continuously in parallel with execution). The execution of floating branches involves suspending execution of the principal plan, executing the floating branch, and resuming execution in the principal plan. In the case of asynchronous floating branches, the currently executing action is suspended, and then it resumes after completion of the floating branch. We extended the Executive's main loop to deal with new events (e.g. *Task Suspension/Suspended*, *Task Resume*,

*Floating Branch Expand* and *Floating Branch Terminate*). We also extended the event handling mechanism in node (i.e. block or task) execution, to deal with suspension/resuming of the execution of the current node when a floating plan is activated. We added functionality for event handling in nodes of synchronous and asynchronous floating branches.

### Properties

Our analysis focused on properties related to the correct execution of the plans, according to the plan semantics, and to the synchronization issues between threads. Specifically, we analyzed the following properties:

P1: *If the last task in the plan terminates successfully, then the only possible outcome for the plan is successful termination.*

P2: *When a task fails, the continue-on-failure flag on the block will always be checked before any outcome is produced; moreover, if continue-on-failure is true, the outcome is success, otherwise it is failed.*

P3: *The Executive only receives ExecCondChecker events if it has registered for them.*

P4: *The ExecCondChecker only puts events in the queue if the Executive registered for them.*

P5: *When a task fails, it will always check its continue-on-failure flag; moreover, if the continue-on-failure flag is false, no subsequent task in the block will be started; new tasks can be started after the parent block reports the results (i.e. other block is expanded).*

P6: *If a task fails, then the parent block's continue-on-failure flag will be checked: if it is true, then the block succeeds, otherwise it fails.*

P7: *If the Executive thread reads the value of the shared variable savedWakeupStruct, then the ExecCondChecker thread should not read it until the Executive clears it first.*

P8: *(Race condition) All accesses to shared structure conditionSetChanged by the Executive and the ExecCondChecker threads will be protected by locks.*

P9: *(Race condition) All accesses to shared structure existConditions by the Executive and the ExecCondChecker threads will be protected by locks.*

P10: *Absence of local and global deadlocks.*

P11: *No irrelevant action execution events can happen.*

P12: *No irrelevant condition checker events can happen.*

P13: *Floating branches and principal plans cannot execute concurrently.*

## Design Level Verification

Our initial analysis uncovered a number of synchronization problems such as deadlocks and data races. Moreover, the design models were used for quick experimentation with alternative solutions to existing defects, leading eventually to the re-design of the software.

As mentioned, safety properties are expressed as LTSs. For example, Figure 7 illustrates property P7 that was formulated by the developer. The property is represented as two states, corresponding to the shared variable *savedWakeupStruct* being cleared or not cleared, and with a third state representing the error state. The developer expected the property to be satisfied. We applied assume-guarantee reasoning as supported by our tools, where assumptions were generated for the *ExecCondChecker* thread (module  $M_1$ ) and discharged on the *Executive* thread (module  $M_2$ ).

The results obtained from the design-level analysis are summarized in the first row of Table 1. The design level model is an order of magnitude smaller than the corresponding Java implementation. The largest state space that our modular verification techniques compute consists of 541 states, as opposed to 4672 states computed by monolithic model checking. We therefore achieved an order of magnitude savings in terms of space.

**Table 1. Analysis results at design & code level**

Analysis	Tool	LOC	Monolithic model checking	Modular verification
Design level	LTSA	700 FSP	4672 states	541 states
Code level	JPF	7.2K Java	183K states	53K states

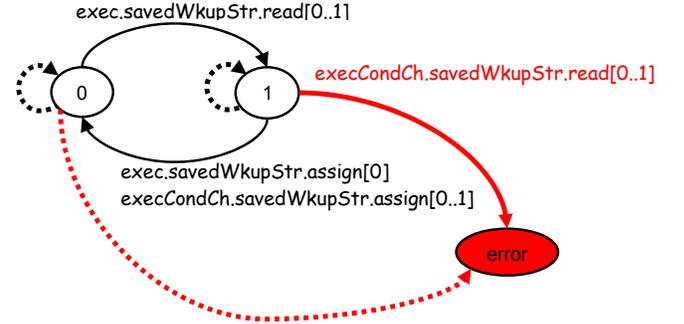
The generated assumption consists of 5 states. It describes an environment where the *Executive* thread reads the *savedWakeupStruct* variable after acquiring the *exec* mutex and holds the mutex until it clears (assigns value 0) the variable. The assumption is illustrated below (in FSP).

```

Assumption = Q0,
Q0 = ( executive.exec.lock -> Q2),
Q2 = (executive.exec.unlock -> Q0
      | executive.savedWakeupStruct.read[1] -> Q3
      | executive.savedWakeupStruct.assign[0] -> Q4
      | executive.savedWakeupStruct.read[0] -> Q5),
Q3 = ( executive.savedWakeupStruct.read[1] -> Q3
      | executive.savedWakeupStruct.assign[0] -> Q4),
Q4 = ( executive.exec.unlock -> Q0
      | executive.savedWakeupStruct.assign[0] -> Q4
      | executive.savedWakeupStruct.read[0] -> Q5),
Q5 = ( executive.savedWakeupStruct.assign[0] -> Q4
      | executive.savedWakeupStruct.read[0] -> Q5).

```

This assumption could not be discharged on the *Executive* thread. The counterexample obtained describes a scenario where the *Executive* thread reads *savedWakeupStruct* and then it performs *wait* on a condition variable associated with the *exec* lock (a *wait* operation automatically releases the lock). The problem was temporarily fixed by adding to the *Executive* thread a statement that clears the shared variable. Note that the variable *savedWakeupStruct* was eliminated altogether when the *Executive* was re-designed.



**Figure 7. Example property**

**Stage I** In the first stage, we checked several simple properties (P10, P11, P12). To do this, we decomposed the system into two modules,  $M_1$  that consists of the *Executive*, the *ActionExecution* and the *EventQueue*, and  $M_2$  that consists of the *ExecCondChecker* and the remaining threads in the system. The results of our analysis are summarized in Tables 2a-2c.

**Table 2a. Analysis results - stage I**

Property	Subsystem	#States, #Trans	A	Result
P10	$M_1$	3805, 10450	n/a	false
P11	$M_1$	8478, 22875	n/a	true
P12	$M_1$	8478, 22875	37 4	false

**Table 2b. Analysis results – property P12**

Subsystem	#States
$M_1$	8478
$M_2$ (discharge)	18080
$M_1 \parallel M_2$ (CRA)	74649
$M_1 \parallel M_2$ (monolithic)	84690

**Table 2c. Reachable state space computation**

Subsystem	#States
$M_1$	8478
$M_2$	14448
$M_1 \parallel M_2$ (monolithic)	> 10 Million

We first checked local and global deadlocks (P10) by incrementally putting components of  $M_1$  and  $M_2$  together.

Note that, in the LTSA, the assumption is that environment inputs are always available. This is a significant benefit for modeling partially specified systems (or verification of modules of systems), because one does not need to explicitly model drivers for the component. Moreover, uninteresting cases where the Executive is deadlocked because no plans are available at the input are ignored.

A local deadlock was detected in  $M_1$ . Two threads, *Executive* and *ActionExecution*, synchronize on shared transitions (in order to start and stop the execution of actions) and they also synchronize via the *EventQueue* (i.e., the *ActionExecution* sends events when the execution of the action is completed). The counterexample represents a behavior where the *Executive* tries to stop the current action, without knowing that the current action was completed (i.e., before processing the respective event), while the *ActionExecution* is waiting to start a new action. This was a problem in our design, which we fixed (by adding self-loops for “unconsumed” stops from the previous actions).

Property P11 was checked on  $M_1$ . This property holds in any environment. Property P12 was checked on the same subsystem. This property does not hold in any environment, since it depends on the behavior of the *ExecCondChecker*, which is in  $M_2$ . We generated automatically the assumption that  $M_1$  needs to make about the *ExecCondChecker* for the property to hold. We obtained an assumption of 374 states. By minimizing  $M_1$  using compositional reachability analysis as supported by the LTSA, we obtain a subsystem of 1493 states; the assumption is therefore more concise to use for analysis.

When we tried to discharge this assumption on the *ExecCondChecker*, after exploring 18080 states we obtained a counterexample describing the following scenario: the *ExecCondChecker* detects the fact that a maintenance condition has been broken, sends an event to the *EventQueue*, but the action terminates before this event gets handled. As a result, the event remains unconsumed in the *EventQueue* and gets handled in the context of the next node, at which time it is irrelevant. The counterexample exhibited the fact that the system is highly asynchronous, as a result of which it is possible for the *EventQueue* to hold “obsolete” events that are no longer relevant to the execution of the current node.

As illustrated in Table 2b, our assume-guarantee framework enables a significant reduction in the state space that needs to be explored (18080 states) as compared both to CRA (74649 states) and to monolithic model checking. Note that, as illustrated in Table 2c, if we disable error detection and simply compute the reachable state space of the model, monolithic model checking runs out of memory after exploring 10 million states.

**Stage II** After we enriched our models with advanced autonomy features (i.e. detailed task and block execution,

floating branch execution, etc.) we checked the remaining properties (P1-P9, P13). We should first note that we could not compute the reachable states of the whole system (the computation runs out of memory when using 1GB of memory); this means that checking any property on the whole system would not complete.

We therefore used compositional techniques. Again, we decomposed the system into components:  $M_1$  (the *Executive* thread, the *EventQueue* and the *ActionExecution* thread) and  $M_2$  (the *ExecCondChecker* thread).  $M_1$  has 47906 states,  $M_2$  has 14496 states. We analyzed the properties using assume guarantee reasoning. Checking properties P3, P4, P8, P9 required small assumptions (the largest obtained assumption has 7 states). Interestingly, properties P1, P2, P5, P6 were checked locally (no environment assumption was necessary). This reflects the modularized architecture of the new executive.

During our analysis we discovered a problem with the design (reflected in the implementation) due to the asynchronous communication between components through the queue. Specifically, property P1 did not hold because of the order of events arriving in the queue: if a task terminates successfully and at the same time a timeout occurs or a condition fails for the parent block, then, the outcome for the parent block can be non-deterministically success or failure, depending on the order in which the corresponding events are put in the queue. Similar problems were found in relation to the execution of synchronous floating branches. The problems were corrected according to the developer’s suggestions, by adding an extra test for cases when events signaling timeouts or failed conditions are received by the *Executive* thread.

It is interesting to note that compositional reachability analysis fails for this large case study. E.g. for  $M_1$  composed with property P3 which has 47918 states, compositional reachability analysis runs out of memory, while the generated assumption has only 5 states computed in 16.165 seconds.

## Code Level Verification

**Model Checking** We used JPF for the analysis of the software components of the first version of the Executive code (which was manually translated in Java). To check each component in isolation, we used the assumptions that were generated during design level analysis to build appropriate environments. Techniques for automated generation of environments from user supplied assumptions are presented in [6]. These environments provide stubs for the methods called by the component that are implemented by other components, or test drivers that execute a component by calling methods that the component provides to its environment. Moreover, these

environments are constrained by the design level assumptions.

Some of the results of this analysis are reported in the second row of Table 1. Compositional verification yields a 3x improvement (in terms of memory used) over monolithic verification. E.g., when we checked property P7 on the corrected system, monolithic (non-compositional) model checking explored 183K states and it consumed 952 Mb of memory in 12 minutes and 12 seconds. In contrast, compositional verification explored at most 60K states, and it consumed 315 Mb in 6 minutes and 55 seconds.

**Run Time Verification** We used Eagle for the run time verification of the C++ code of the Executive. The design-level artifacts (properties and assumptions) were automatically translated into Eagle monitors. We instrumented (by hand) the code of the Executive, to emit events that appear in these assumptions and properties. To generate test input plans, we encoded the plan language grammar as a non-deterministic input specification. Running model checking on this specification generates hundreds of input plans in a few seconds.

We developed a tool that integrates run time verification and test input generation to perform assume guarantee style reasoning about the run-time behavior of the executive. The tool is fully automated after setup. It generates a set of test input plans, a script runs the Executive on each plan and it calls Eagle to monitor the generated run-time traces. The user can choose to perform a whole program (monolithic) analysis or to perform assume-guarantee reasoning. In the latter case, the Executive is broken in two parts:  $M_1$  consists of the *Executive* thread, the *Event Queue* and the *ActionExecution* thread, and  $M_2$  consists of the *ExecCondChecker* thread and the remaining threads.

We ran several experiments for different input plan configurations. For Property P3, we found a discrepancy between the implementation and the models, due to the fact that nodes can send *null* conditions. Instead of putting these in the condition list (and altering the values of variables *conditionSetChanged* and *existConditions*), the *ExecCondChecker* code immediately pushes an event to the queue. We corrected this in the model.

One benefit of our approach is that the use of design-level assumptions in the verification of software implementations enables the detection of costly integration problems well prior to system integration. In fact, assume-guarantee verification can detect such problems as soon as one component of a software system becomes “code complete” (while the remaining software components may not be even implemented yet). Whenever the complete implementation of one component becomes available, we can check it against the required properties, under environments that are suitable restricted by the design-

level assumptions. When the rest of the components become available, we check the assumptions on these components. As a result, we guarantee that the whole system behaves correctly, without being necessary to perform verification/testing on the integrated components.

Also note that assume-guarantee verification provides better unit testing. We only test components (i.e. units) in the environments in which it can be expected that the units will be integrated. Moreover, assume-guarantee reasoning provides increased behavioral coverage of the integrated system. We have found that, in some cases, assume-guarantee verification uncovers errors that escape integration testing. The reason is that by generating and analyzing traces for each component in isolation, we can predict, by mathematical inference, properties about all the possible inter-leavings of these traces, while at system integration, one could generate only a subset of these interleavings.

## Conclusions and Future Work

We described the development and application of compositional verification techniques to a significant autonomous system throughout its lifecycle. Subtle errors in the design and implementation of a rover executive have been detected. Compared to testing by itself, these techniques are particularly good for two challenging aspects of autonomy verification:

- 1) Assuring correct execution of plans, particularly in environments that cause multiple failures of plan primitives. Robust execution of plans, especially plans with contingencies, is a significant advantage of autonomy software compared to traditional sequence execution. Verifying this advantage is expected to be a significant factor in the acceptance of autonomy software for NASA.

- 2) Concurrency is an inherent feature of autonomy software but is difficult to debug through testing alone. Concurrency is inherent to autonomy: first, because responding robustly to asynchronous environmental changes introduces concurrency – and difficulties such as race conditions – independent of implementation. Second, in contrast to sequence execution or simple sequential plans, plans for rovers consist of multiple concurrent tasks overlapping in time. Third, modern programming practices for complex software favors encapsulating control into multiple threads, introducing concurrency at the implementation level. Concurrency errors are typically manifested only as transient faults in black-box testing, and are often masked by thread scheduling and computational resource profiles that differ subtly and uncontrollably between the testing environment and actual field conditions. Our techniques provide high assurance that autonomy software is free of concurrency errors.

---

The techniques presented have several benefits. First, they can be applied early in the software development life cycle, when it is cheaper to detect and fix bugs. Second, our compositional techniques provide a way to automatically decompose global (system-level) requirements into local properties, which are cheaper – in terms of time and consumed memory – to check, with an increased level of confidence. Third, assumptions allow checking global properties (that are usually checked at integration testing) at unit testing level, thereby increasing the chances of detecting costly integration errors early. Fourth, our results show that compositional reasoning can enhance integration testing at the source-code level (by exploring multiple inter-leavings in concurrent programs).

We assume a top-down software development process where one first creates and debugs design models that are subsequently used to guide the implementation. It goes without saying that it is not a straightforward task to obtain a correct model. However, verification tools provide several features such as interactive simulation, which facilitate the debugging of models. Moreover, as our results show, it is essential to make connections between verification performed at the design level with the actual implemented system. Note that we are currently investigating a complementary approach that uses abstraction techniques to automatically extract models from source code [3].

In the future, we plan to leverage our work for the analysis of other executives (and autonomy software), with minimal modifications (e.g. for plan generation, we could simply modify the plan language grammar; for properties and assumptions, we expect to build upon the existing specifications). For example, we plan to participate in the development and analysis of next generation executives, built within the CLARATy decision layer distribution [10].

## References

- [1] J. M. Cobleigh, D. Giannakopoulou, C. S. Pasareanu, Learning Assumptions for Compositional Verification, in Proc. 9th International Conf. on Tools and Algorithms for the Construction and Analysis of Systems, 2003.
- [2] D. Giannakopoulou, C. S. Pasareanu, H. Barringer, Component Verification with Automatically Generated Assumptions, J. of Automated Software Engineering, 2005.
- [3] S. Chaki, E. Clarke, D. Giannakopoulou, and C. Pasareanu. Abstraction and assume-guarantee reasoning for automated software verification. *RIACS TR 05.02*, October 2004.
- [4] D. Giannakopoulou, C. S. Pasareanu, J. M. Cobleigh, Assume-guarantee Verification of Source Code with Design-level Assumptions, Proc. of the 26<sup>th</sup> International Conf. on Software Engineering, 2004.
- [5] C. Artho, H. Barringer, A. Goldberg, K. Havelund, S. Khurshid, M. Lowry, C. S. Pasareanu, G. Rosu, K. Sen, W. Visser, R. Washington, Combining Test Case Generation and Runtime Verification, in Theoretical Computer Science, 2004.
- [6] O. Tkachuk, M. Dwyer, C. S. Pasareanu, Automated Environment Generation for Software Model Checking, in Proc. of the 18<sup>th</sup> IEEE International Conf. on Automated Software Engineering, 2003.
- [7] D. Dvorak, R. Rasmussen, G. Reeves, A. Sacks. Software Architecture Themes in JPL’s Mission Data System. In Proceedings 2000 IEEE Aerospace Conference.
- [8] J. Magee, and J. Kramer, Concurrency: State Models & Java Programs: John Wiley & Sons, 1999.
- [9] S. C. Cheung, J. Kramer, Checking Safety Properties using Compositional Reachability Analysis, ACM Transactions on Software Engineering and Methodology, 8(1):49-78, 1999.
- [10] I.A. Nesnas, A. Wright, M. Bajracharya, R. Simmons, T. Estlin, W.S. Kim, CLARATy: An Architecture for Reusable Robotic Software, SPIE Aerosense Conf., 2003.
- [11] E. M. Clarke, O. Grumberg, and D. A. Peled, Model Checking: The MIT press, 1999.
- [12] 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.
- [13] C. B. Jones, Specification and design of parallel programs. Information Processing 83: Proceedings of the IFIP 9th World Congress, 1983: pp. 321--332.
- [14] A. Pnueli, In Transition for Global to Modular Temporal Reasoning about Programs, in Proceedings of the Logic and Models of Concurrent Systems. 1985.
- [15] G. Brat, D. Giannakopoulou, A. Goldberg, K. Havelund, M. Lowry, C. S. Pasareanu, A. Venet, W. Visser, R. Washington, Experimental Evaluation of verification and validation Tools on Martian Rover Software, in International Journal on Formal Methods in System Design, September – Nov. 2004, 25(2-3), 167-198.
- [16] R. Washington, K. Golden, J. Bresina. Plan Execution, Monitoring, and Adaptation for Planetary Rovers. Electronic Transactions on AI, 4(A):3-21,2000.
- [17] D. Angluin, Learning Regular Sets from Queries and Counterexamples, Information and Computation, 75(2).
- [18] W. Visser, K. Havelund, G. Brat, S. Park, F. Lerda. Model Checking Programs. Automated Software Engineering Journal. Volume 10, Number 2, April 2003.