Submitted for publication.

Bridging the Gap between Systems Design and Space Systems Software

David Garlan[*], William K. Reinholtz[**], Bradley Schmerl[*], Nicholas D. Sherman[*], Tony Tseng[*]

* *School of Computer Science*
Carnegie Mellon University
Pittsburgh, PA 15213 USA
{garlan,schmerl,nds,tttseng}@cs.cmu.edu

** *Jet Propulsion Laboratory*
California Institute of Technology
Pasadena, CA 91109 USA
William.K.Reinholtz@jpl.nasa.gov

Abstract

A challenging problem for software engineering practitioners is moving from high-level system architectures produced by system engineers to deployable software produced by software engineers. In this paper we describe our experience working with NASA engineers to develop an approach and toolset for automating the generation of space systems software from architectural specifications. Our experience shows that it is possible to leverage the space systems domain, formal architectural specifications, and component technology to provide retargetable code generators for this class of software.

1. Introduction

A long-term goal of software engineering has been to establish systematic techniques for developing systems from high-level specifications or models. This line of research has led to a long stream of results in areas of formal refinement, automatic code generation, and, perhaps most recently, model-driven architecture (MDA) [8]. The attraction of the later stems from the observation that considerable leverage can be obtained by separating software design into two levels. At the top level one describes a system in terms of a "platform-independent" model (PIM). Then one reifies that model into a lower-level "platform-specific" model (PSM) that binds abstract components and connectors to concrete mechanisms and code. This separation allows one to focus initially on abstract structure and functionality of a system, binding implementation issues later, and potentially allowing the same abstract model to be targeted to different platforms.

While a great idea in principle, it is currently unclear how one should instantiate MDA in practice. In particular, what exactly is meant by "platform independence?" Which details should go in a PIM and which in the PSM? Can one apply a component-based approach to a PIM? If so, how should one describe components at that level so they can be "refined" into working code in more detailed models? What is the nature of code generation in such a scheme? How automated can it be? Where does the concrete code come from, and to what extent can one really target the same PIM to different PSMs?

The answers to such questions are likely to be domain-specific. At the very least, the nature of the reification process is likely to vary considerably depending on whether we are talking about a large scale-distributed information system (with platforms such as CORBA and J2EE), as compared to a resource-constrained embedded system (with real-time OS platforms concerned with scheduling, memory footprint, power consumption, etc.)

In this paper we describe our experience of developing an approach and supporting tool set to support MDA-like approach to NASA Space Systems Software. The key elements of the approach are (a) the use of formal architectural modeling to capture the abstract system description; (b) the clear separation of essential component functionality (described using pure functions) from incidental (or platform-specific) code; and (c) a retargetable code generator and reuse repository to translate architectural designs to one of many possible deployment platforms.

2. The Challenge for Space Systems

Space systems are a natural candidate for an MDA-like approach. In current practice at NASA, systems engineers typically develop the high-level design for a space mission based on knowledge of the mission goals, the target environment, and system resources. This design is specified in part as a high-level architecture in terms of components such as sensors actuators, estimators, controllers, etc., and their pathways of interaction (including shared variables). Later that architecture is translated into working code by software engineers who take into consideration details such as scheduling priorities, communication mechanisms, storage policies, etc.

Unfortunately, bridging the gap between the system design and working code is manual, brittle, and error prone. Programmers may not fully understand the intentions of system designers, and there is no verifiable relationship between system design and code.

This situation is compounded by the fact that for most space missions multiple versions of the software must be

created. Some are full-featured, used during development, simulation, and testing on-ground; other leaner versions are needed for resource-constrained flight platforms.

An MDA-like approach could in principle have a strong impact on this kind of development by providing a more rigorous connection between abstract designs and deployed code, by helping to automate the production of code from those designs, and by reducing the effort to target the same abstract system design to multiple deployments. Such a solution, however, would need to

1. support existing systems engineering models and methods, including the ability for systems engineers to specify detailed algorithms for such things as estimation, mission planning, and actuation;
2. provide a formal enough representation of the system design to support analyses and check conformance to design constraints;
3. allow software engineers to produce code preferably automatically) targeted to radically different deployments;

Note that the desired separation of concerns for NASA space systems has a very different flavor from those in many other domains. For most systems the distinction between a PIM and a PSM is that the former excludes details about the physical deployment environment (e.g., the location and number of servers). For space systems, however, the nature of the physical platform (e.g., sensors, actuators) and its detailed characteristics (failure rates, jitter, power consumption, etc.) are central to the systems design process and the resulting high-level design, and must appear in the PIM. Lower-level variability for a particular deployment would include things like quality of monitoring and debugging code, target programming language, and storage policies.

3. Related Work

The problem of moving from abstract designs to code has a long history of research and development. Most researchers have examined this from a theoretical perspective, providing theories of correctness-preserving refinement in languages like CSP [6], Z [12], algebraic specifications [5], and many others. However, these have had limited impact on industrial practice, since they tend to require levels of formal training and large investments in up-front system specification.

A number of researchers have looked at the more constrained problem of moving from architectural models to code. [7] proposed a form of "correct" architectural refinement, based on the use of transformational patterns. This work focused on moving from high-level architectures to lower-level ones, as opposed to code. Aldrich has proposed the ArchJava language [1] as a staging point for transforming architectures to code. This is a promising avenue, but requires the use of a specialized programming language extension to work. Moreover, it does not directly address the problem of targeting multiple code deployments. [11] addresses code generation from architectures, but also does not provide alternative deployments, and is limited to very specific forms of connection.

The closest branch of related work is the recent flurry of activity in the area of "Model-Driven Architecture" (MDA), proposed by the OMG as a method and set of notations for moving between high-level and low-level designs [8]. As noted earlier, MDA proscribes a two-level process, in which deployment details are added at the low-level so that the same abstract design can be used in different concrete settings.

While attractive in principle, as noted, there remain many details about the MDA approach that remain unanswered. This paper can be viewed as shedding light on some of those answers for the specific domain of space systems software. This domain has some distinct characteristics that make it both challenging and tractable. One of the distinguishing aspects is the need to have precise representations of component functions at an abstract level. Another is the need to model the physical setting (actuators, sensors, environmental and system state) at the high level. This is in contrast to most MDA approaches which leave such details to the lower-level model (PSM).

The work reported here also builds on previous work by the authors, unifying two separate streams of research to produce a new synthetic approach. The first stream is formal representation of software architecture and tools to analyze those descriptions [2][4]. The second is a proposal to define control systems from components specified as pure functions [3]. In this work we show how to use architectural descriptions combined with pure functions to create a tool that generates deployable code.

4. Current Practice

At the start of our collaboration, NASA was well aware of the problems outlined above, and had developed a number of processes and technologies to ameliorate some of the problems: (a) a well-defined system design process and repository to store the results; (b) an architectural style well-matched to space systems development; (c) a large body of reusable code for creating deployments of a system. We consider each, together with comments about their limitations.

Design Process: The success of most NASA missions depends critically on up-front design by system engineers who consider (a) the goals of the mission (b) scenarios of use (c) resource concerns (d) failure modes to produce a systems design. This process, called *state effects analysis*, determines the relevant state variables for the system and

their dependencies, as well as the algorithms used by the various system components that examine and change that state. Results of this process are stored in a *state database*, which records both the resulting design and the rationale behind the design decisions.

While state effects analysis and the state database help space systems domain experts to design effective systems, balancing complex requirements for functionality, resource usage, and failure handling, there are a number of limitations. First, the design decisions are largely informal. For example, the database may note a dependency between two states, but that dependency is not represented in a way that can be automatically checked against a resulting design. Second, and more importantly, components in the design must be represented using the concrete notations of a programming language: C++ in this case. While programming languages allow engineers to be concrete about the algorithms to be used, they tend to overconstrain the implementations. In particular, they force premature decisions about things such as order of processing of input variables, synchronization mechanisms, communication polices, and data representation decisions.

Software Architecture: Over the past few years engineers at NASA's Jet Propulsion Lab (JPL) had developed a new architectural style for space systems, called Mission Data Systems (MDS) [9]. MDS adopts a product-line approach to space software, by providing a generic architectural framework for space systems design, providing a vocabulary of design (sensors, actuators, state variables, etc.) together with rules for how these elements can be combined. This is coupled with a reusable code base (described below) for instantiating the framework for specific missions.

While the creation of an architectural style for space systems is an important step towards regularizing development, and providing opportunities for analysis and reuse, at the start of our project this style was largely described informally. As we illustrate later, rules for composition were expressed in English, and there was no way to either represent a full design in the MDS style formally, or to check for conformance to that style.

Reuse: Taking advantage of commonalities in space systems (as characterized by the MDS style), over the past few years NASA engineers had developed a large body of reusable code for creating specific deployments. This code covered areas like data structures for state representation, communications infrastructure, event logging, timing services, units of measurement, and visualization. In fact, in its current state there are over 250K lines of (potentially) reusable framework code.

While providing excellent opportunities for reuse, the existing body of framework code had several limitations. First was the sheer complexity of it. For a given target deployment, knowing which packages to use, in which combinations, was not a trivial matter. Second, since framework code had to be combined with mission-specific code manually, there were many opportunities for error, and very little that one could do to check that the resulting system continued to respect the abstract design. Third, as noted, components written in C++ could potentially conflict with the use of particular framework code, by prematurely binding implementation decisions.

5. Our Approach

Working with engineers at NASA JPL, we developed an approach, and a tool called MDS Studio, that is centered on three significant changes to their current practice:

1. Use a formal architectural modeling language to represent a system design and its constraints. Linked to existing NASA system design databases, a formal architectural description provides an explicit representation of a system configuration, and permits automated analyses such as conformance to architectural style, as well as system-specific constraints. (For example, if a system engineer determines that in the system under design a state A depends on state B, a rule is automatically created in the architectural modeling tool to check that the estimator for state variable A is also connected to state variable B.)
2. Define high-level components as pure functions (stateless mappings of inputs to outputs), thereby abstracting from details of timing, synchronization, communication, and data representation, while still retaining the ability to describe critical algorithms for state estimation and control.
3. Provide a retargetable compiler that can produce multiple versions of deployable code from the abstract architectural design. The compiler leverages the substantial body reusable framework code to map the "essential" computations defined in the abstract configuration to specific implementations, but does so in a way that preserves the design constraints.

This approach is applied in the following process:

1. Systems engineer conduct a state effects analysis, to determine the states required to achieve a mission.
2. Software engineers write essential code to implement these states as pure functions, which just implement the steps of the various control loops, but not how they are put together or scheduled.
3. This essential code is uploaded to a state database, which is then used to generate architectural styles, which are used to:

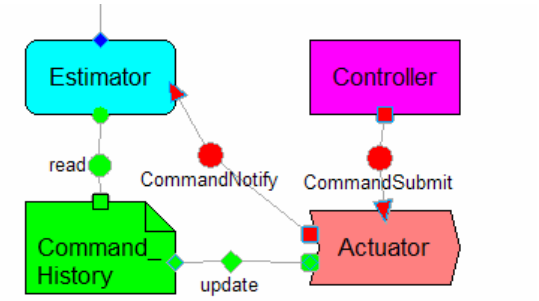

Figure 1. A *Controller/Actuator/Estimator* pattern in MDS

4. Compose the system in an architectural model that specifies instances of the essential code and how they relate.
5. This architectural model is fed into code generators that weave the essential code with framework code and other incidental code, such as connections and schedulers, to produce code that can be deployed. The aim is that code can be generated for different deployments without affecting the pure functions or the architectural model.

In the remainder of the section we elaborate on these three innovations.

5.1. Architectural Modeling and Analysis

The architectural modeling tool allows the user to create instances of components, both pure functions and state variables. The distinction between essential and incidental is valuable to understand in this process: the components are essential code, whereas the connectors are incidental. The architectural model primarily encapsulates the knowledge of the instances of components, and the communication patterns between components. The architectural model is built according to a set of architectural styles. An architectural style captures the component, connector, and interface types that can be used to compose an architecture, in addition to rules about their composition.

The base architectural style used for MDS consists of the following component types:

- *StateVariableT*: Contains the value history record of the state over time and goals associated with the state.
- *EstimatorT*: Is responsible for examining all of the available cues (other states, sensors, or goals) and updating state variables periodically to provide a current best estimate of the states value based on available evidence (command history, other states, sensor values, etc.).
- *ControllerT*: If there are goals associated with a state variable, this component is responsible for delegating the goals to other states, or for issuing commands to adaptors to achieve the state.
- *ActuatorT*: Represents the interface between a controller and the hardware. Commands are issued to actuators to get the spacecraft to do something.
- *SensorT*: Represents an interface between an estimator and hardware, for use by estimators.
- *ValueHistoryT*: Store a discrete set of data. For example, they may be used to store the history of commands sent to an actuator by a controller.

In addition to the above component types, there are connector types for communicating between components (e.g., *Command Submit*, *Measurement Request*, *State Update*), and port and role types for component and connector interfaces. Figure 1 illustrates a small segment of an architecture written in this style. This segment depicts interaction between a *Controller*, an *Actuator*, and an *Estimator*. In this interaction, the *Controller* submits a command to an *Actuator* via its *Command Submit* connector. The *Actuator* then notifies the *Estimator* that it received a command and writes that command to a *Value History*. Subsequently, the *Estimator* queries the *Value History* to find out what the command was.

In addition to a set of types, the MDS style also defines rules about the composition of an MDS system. These rules were expressed in English by JPL engineers, but needed to be translated into formal architectural rules that can be checked automatically with architectural tools. In developing the architectural style, we were given ten informal rules, which were translated into 39 architectural rules. Examples of the MDS rules:

1. If an *Estimator* can be notified of a command by an *Actuator*, then that *Estimator* must be able to read the *Value History* that the *Actuator* updates.
2. An *Actuator* must have exactly one *Controller* connected to it.
3. An *Actuator* must have the same number of *Command Submit*, *Command Notification*, and *Value History Update* ports (one for each type of command that it receives).

The first MDS rule above can be captured in Acme with the following predicate:[1]

```
invariant (forall e :! EstimatorT in self.components |
  (forall cnp :! CmdNotProvT in e.ports |
    (forall a :! ActuatorT in self.components |
      (forall cnr :! CmdNotReqrT in a.ports |
        (connected (cnp, cnr) ->
          (exists vh :! ValueHistoryT in self.components |
            reading (e, vh) and updating (a, vh))))))[2];
```

[1] In this rule, self refers to the system, *italicized* words refer to predefined Acme functions, and the clause <name> :! <type> means that <name> declares the type <type>

[2] reading and updating are defined elsewhere in the style.

In addition to this base MDS style, our approach takes advantage of the use of specializations of the MDS style that are tailored to both the particular mission and code generation. Specializations of this style must satisfy all of the rules of the original MDS style, but may add rules and structure. For example, if the mission requires a wheel state variable, then there will likely be specializations of the state variable type tailored to this wheel, in addition to specializations of wheel estimators, wheel controllers, etc. For example, a *WheelVelEstT* would be a subtype of *EstimatorT*, and might have additional required ports or properties. Furthermore, if state analysis specifies a dependency between a wheel state variable and a power state variable, then the corresponding mission style would check that the estimator associated with one state variable queries the state of the other state variable. A rule of this type would look like the following:

```
invariant (forall e :! WheelVelEstT in self.components |
  (exists s :! PowerStateVar in self.components |
    connected (e, s));
```

5.2. Component Specification

Whereas in MDS as implemented by NASA JPL the components of the system (controllers, estimators, and hardware adaptors) are C++ objects, in MDS Studio the components are implemented as pure stateless functions. This means that everything that is needed for a function is provided through the interface to the function, and the result is returned by the function, i.e., the function does not read or change any global variables.

Consider for example a Controller function. The goal of a controller is to update the state of any pending or current goals, as needed, and to issue commands to actuators to achieve goals. The generic header for a controller pure function would look like:

```
ControllerFunc (Goals, StateVar, Config) → messages
```

The algorithm for the controller would calculate what it needs to do to achieve the goals, based on the value of the state variable, as well as when the goals need to be achieved. It may use the configuration parameter to help in its calculations. The configuration parameter will contain information that is needed for the controller to accurately calculate what it needs to do, such as the diameter of the wheel being controlled. If it needs to actuate the hardware, then it returns the required messages that need to be sent to an actuator. Where it receives the information from, and the particular actuators to which its messages are sent, are completely independent of the controller code. The author of the controller code needs to know the type of actuator that is being controlled so that it know what message need to be sent, as well as to calculate how to achieve goals.

The suggested benefits of using pure stateless functions include making the components easier to write, test, and integrate in a code generation system. For example, while it is possible to write an OO component with a race condition depending on the order of invocation of the methods on the object, that category of defect is not possible in a pure function. Additionally, a testing suite no longer has to be savvy of the internal state of the object; the return value of the pure function is a function of its inputs alone. Finally, it is easier to capture the knowledge of how to interact with a pure function in the state database; by adopting certain constraints on the signature of the pure function, the process of generating the code to invoke the pure function becomes much simpler and easier to inspect for correctness.

Essential and Incidental Code

A critical aspect of our approach is the separation of the software into essential and incidental code. Essential code is broadly defined as software that is specific to the physical platform, the specific mission, or the physical environment in which the rover is operating. For example, essential code associated with a *Power* state on the spacecraft would include the code for recording the *Power* state variable, and pure functions for the *Power* estimator, controller, sensor, and actuator.

Incidental code is everything else. Essential code is strictly defined as the state variables with the important system state, and the pure functions representing components. Incidental code is subdivided into several types:

- A space system's reuse library; for example, classes for time reckoning and memory management are in this category.
- Platform-specific code for interacting with particular hardware on a particular space craft.
- Code for communicating between components, which is weaved in by the code generator.
- Classes and types to support the essential code, such as message and command types specific to a particular mission.

In order to have a clean division between the essential and incidental code, essential code is implemented using pure functions as described above, but may use classes and types in the incidental code. Furthermore essential code is freed from having to be concerned with scheduling or thread-safety (e.g., locking a mutex before accessing a variable). In fact, essential code is not permitted to handle this – but rather, it is the responsibility of the incidental code to handle these issues.

Generating Mission-specific Architectural Styles

Once the essential code elements have been written for a particular mission, they are added to the state database for

the mission. The architectural style specializations corresponding to a particular mission are generated from the state database. An engineer can then assemble the architectural model for the mission using these styles. Consider the Power example introduced earlier. The architectural style for the mission will include a *PowerStateVariableT* and a *PowerEstimatorT* type (in addition to other component types). The software engineer can then assemble the architectural model from these types, and may include multiple instances of these types (one for each battery on the spacecraft, for example).

5.3. Compilation

The code generation system is responsible for translating the component instances and communication patterns in the architectural model to the appropriate representations in code. For every component, it generates a wrapper around the associated pure function that handles the selection of correct objects to pass to the pure function, and where to pass messages. The code generator also produces code to ensure that the components are scheduled, as well as any code for logging or debugging.

Different code generators can be plugged in to generate code for different platforms, without having to change the architectural model. For example, a code generator for a simulation may choose to log all messages that are sent, to aid in debugging.

It is possible to write incidental code generators with a high degree of variability in the generated software. Attributes that can be modified include, but are not necessarily limited to, the following list.

- *Threading model*: single-threaded or multi-threaded systems are possible. The incidental code is responsible for taking any precautions (e.g. locking mutexes) necessary in a multithreaded environment.
- *Debugging code*: The incidental code may include debugging software to examine messages or state while the system is executing.
- *Scheduler implementation*: the implementation of the scheduler is not dictated by the architectural model.
- *Style of implementation*. This is the example difference implementation constructed over the course of this project. In one incidental code generator, procedural code is produced, with globally accessible data and wrapper functions; in another incidental code generator, wrappers are classes, and connectors between the components are also full-fledged classes.

Figure 2 shows a pseudo-code example of the wrapper that may be generated by a code generator for calling a controller. The wrapper handles thread-safety and logging, making sure that it acquires and releases the appropriate locks, and sends the appropriate logging messages.

```
void controller17_wrapper () {
   pthread_mutex_lock (stateVar17_mutex);
   pthread_mutex_lock (stateVar17_goals_mutex);
   pthread_mutex_lock (actuator17_config_mutex);
   pthread_mutex_lock (actuator17_inputQ_mutex);
   log.out ("Calling controller17 function");
   message =
      controllerFunc (stateVar17_goals, stateVar17,
                      actuator17_config);
   log.out (messages);
   actuator17_inputQ.put (message);
   pthread_mutex_unlock ()...
}
```

Figure 2. Example of Generated Code.

6. Example

To illustrate how this approach works in practice, consider a simple example of a wheel motor "control loop." The wheel motor control loop controls an actuator that adjusts the current sent to a wheel motor, and receives data from a sensor that reports encoder values, which count how far the wheel has turned. The goal is to construct a system that controls the motion of a wheel. It should be possible to set a goal (for example, for a particular velocity) on the state variable associated with the wheel; it should also be possible to read data such as the current velocity from the wheel's state variable. (For the purposes of this example, we will not discuss the interaction between this control loop and other control loops that may exist on the space craft.)

6.1. Writing the Essential Code

The essential code is written based on the results of the state effects analysis. For each important state that is identified during state analysis, essential code components associated with it need to be written.

In the wheel control loop, the following mission-specific components are created:

- Wheel State Variable: This state variable contains the value history of the wheel position and its derivatives, such as the wheel velocity and the wheel acceleration.
- Wheel Controller: This pure function component is responsible for handling goals placed on the Wheel State Variable. It achieves the goals by issuing commands to the wheel actuator, such as increasing the amount of current sent to the hardware by the actuator to affect the speed at which the wheel turns.
- Wheel Motor Actuator: This pure function component is responsible for accepting commands from the Wheel Controller, relaying the commands to the actuator, and storing them in a command history.

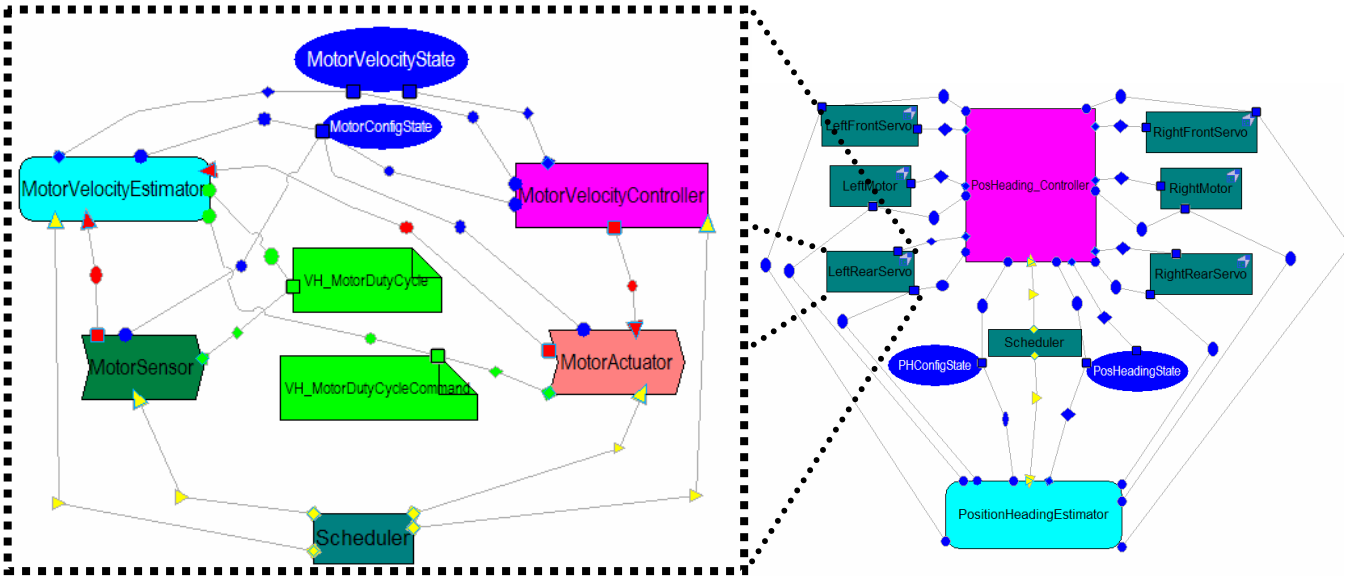

Figure 3. The Architectural Model of the Wheel Controller, and its Context in the Larger Robot Architecture.

- Wheel Encoder Sensor: This pure function component is responsible for reading the data sent by the encoders (encoders measure the number of times the wheel has turned) and storing these measurements in a measurement history.
- Wheel Estimator: This pure function component is responsible for analyzing the available cues (current state of the Wheel State Variable, command history, measurement history, and goal record) and updating the state function in the Wheel State Variable with a new, best estimate of the value history of the wheel. This estimate is represented as a state function; this continuous function returns the state at any time in the past or future; different times will have different levels of the uncertainty for that time. Entities that use the state need to be savvy of the uncertainty associated with any state they read.
- Wheel Configuration State Variable: This state variable is responsible for holding information about the wheel. For example, the diameter of the wheel would be a good candidate for capturing in a configuration state variable, as opposed to hard-coding the value; this allows for easy reuse of the code for different size wheels.

6.2. Putting Essential Code in the State DB

The next step is to describe the essential code in the State Database. Every essential function is named; the arguments and their associated order and types are described; and the return values are described. In addition, every state variable is described, along with its associated member variables.

For the wheel motor control loop example, all of the pure functions and state variables are entered into the state database. Our tool then generates architectural styles to represent the pure functions as components in an architectural model. The types generated for the wheel example would be: MotorVelocityStateT, MotorConfigurationStateT, MotorVelocityEstT, MotorSensorT, MotorActuatorT, and MotorVelocityCntrlT.

6.3. Developing the Architectural Model

The architectural model is then developed, allowing the software engineer to connect the various pure function components. In addition to the component types generated as part of the previous step, the software engineer uses the following types of generic MDS connectors to connect instances of these components:

- Connectors between Pure Functions and State Variable members: these connectors represent that member of the state variable being passed into the pure function as an argument. There are two subtypes of this connector, representing read-only and read-write semantics, respectively.
- Connectors between Pure Functions and Standalone Value Histories: these connectors represent that standalone value history being passed into the pure function as an argument. There are two subtypes of this connector, representing read-only and read-write semantics, respectively.
- Connectors between Pure Functions and Pure Functions: these connectors represent a message being passed from one Pure Function to another.

Furthermore, the software engineer needs to introduce information to aid the code generator in scheduling the components. The software engineer can instantiate a Scheduled Rate Group component and an appropriate connector to connect those components that must be scheduled. This is used to specify the rate at which components are scheduled, and which components should be scheduled in that rate group.

Once the architectural model has been developed, it can be checked for conformance to architectural rules prior to code generation. For example, the model can be checked to ensure that ports on either side of a connection are compatible, and that every pure function is connected to a scheduler.

The left of Figure 3 shows the architectural model for the wheel controller example. It shows that the MotorVelocityEstimator, MotorVelocityController, MotorActuator, and MotorSensor are all in the same rate group, that the MotorVelocityEstimator uses information from the MotorVehicleState, the MotorSensor, and the MotorActuator to calculate the new states for the MotorVelocity. If the estimator also required information from other states (such as a power state), then this would also be indicated in the architectural model.

The wheel motor controller is actually part of a larger system for a robot with four wheels. The full architectural model is presented to the right of Figure 3. This allows for scalability of the architectural model, and also allows for reuse. For example, the RightMotor component is decomposed with a similar control loop for the LeftMotor.

6.4. Generating the Code

Once the architectural model is developed, a code generator is chosen by the software engineer to produce the code for a particular deployment. For example, the code could be generated from this model for deployment on a Personal Exploration Robot (PER) hardware, or for simulation in a software environment.

7. Implementation and Evaluation

7.1. Implementation

We have implemented the above approach in a toolset that consists of the following components:

- A cleanroom MDS Framework implementation in C++. Because of security restrictions, we were at the time unable to view actual MDS code. We implemented our own version of the framework that is a subset of the existing MDS framework, but doesn't include some of the utility packages (such as units of measurement).
- API interfaces for the pure functions.
- MDS Studio, an MDS architectural modeling tool that extends AcmeStudio [10], an architecture development environment. It is built on top of Eclipse, an open extensible integrated development environment.[3] The extensions provide access to the code generators, as well as actions that are specific to the MDS architectural style.
- Two types of code generators to generate code in different platform styles:
  - The procedural incidental code generator wraps each pure function (corresponding to the component type) with a wrapper function (corresponding to the component instance) that is responsible for accessing a shared data structure and selecting the correct data structures to interact with each other in the pure function. In other words, there is a global data structure containing state, and wrapper functions that maintain the structure and communication patterns in the network.
  - The object-oriented incidental code uses C++ classes to wrap each pure function. These classes are responsible for supporting the communication patterns described in the architectural model.

We have successfully generated code that runs on the Personal Exploration Rover (PER),[4] that is a robot designed to be similar to the Mars Sojourner.

7.2. Evaluation

Having asserted that the tool developed in this project provides several benefits to JPL Engineers, it is appropriate to describe the benefits that have been observed.

7.2.1 Reduced Costs

In this system, there are the following opportunities for reuse of code:

- MDS Framework code (schedulers, memory management, etc) may be reused across all missions.
- Essential code may be reused if a different mission is using the same hardware, or in cases where the robots are operating in a similar physical environment (for example, estimators for time-of-day and solar radiation levels on Mars will be reusable across different robot platforms).
- Type definitions for stateful objects may be reused across systems, if system and software engineers choose to. They contain no program logic, merely data. For example, a TemperatureKelvin class definition likely will find use in many circumstances.
- Incidental code generators can be reused across different platforms.

In addition, the code generation system saves having to write a significant amount of code. It takes approxi-

[3] http://www.eclipse.org
[4] http://www.cs.cmu.edu/~personalrover/PER

mately one hour to assemble the architectural model of the PER, resulting in a system of approximately 50 components. The incidental code generated for the PER consists of approximately 542 lines of code. This is a good productivity level. However, this incidental code represents the very simplest of the incidental code generators that can be imagined. As incidental code generators become more capable, and generate more sophisticated code, the productivity gains will be even greater.

7.2.2 Increased Reliability

The architectural modeling tool provides a platform for reasoning about the correctness of the model developed by the software engineers. Whenever new rules about the desired behavior of the system are discovered, the system engineer can express this as a rule in the architectural modeling tool, ensuring that this check becomes a part of the build process of the system. One example of a potential rule would be a check that a state cannot depend on itself for estimation (this would cause instability in estimation), even if separated by many intermediate states.

Another benefit of this approach is a side effect of the reuse library; as components accumulate over time, the amount of testing that any one component might experience will increase, providing more confidence in the reliability of the components.

Finally, the functional approach to developing the essential components is amenable to the design and maintenance of an automated suite of unit tests. The simplicity of the pure function signature specification ensures that writing unit tests for these functions is a straightforward practice; each pure function can be tested in complete isolation from all other pure function components.

7.2.3 Usability

This project was developed in collaboration with engineers at the NASA Jet Propulsion Laboratory, in multiple meetings. As a part of this project, the tool was demonstrated to several engineers at JPL. Initial reactions to the tool were quite positive, indicating that such a tool for their system would be helpful in their work.

8. Technical Challenges

Having described the approach and briefly illustrated it, we now consider some of the specific technical challenges that arose in carrying out the work.

8.1. Adding implementation choices

One of the critical issues in an MDA-like approach is how to introduce implementation choices that are not directly relevant at the PIM model, but are needed to produce working code. In the case of MDS Studio there were two categories of extra detail. First was scheduling information: as real-time control systems specification and analysis of timing properties are critical for the final system, even though they are not part of the essential code produce for the high-level model.

To handle this we introduced a special architectural construct called *scheduled rate groups*. Components in the same rate group are scheduled together as periodic tasks at a specified rate defined by the group. Putting this in the architecture has the benefit that timing analysis can be performed over the architectural model.

The second class of implementation choices concerned the nature of the code produced. MDS Studio permits variability along the dimensions outlined earlier. The choice of value within each of those dimensions is determined by the choice of compiler. For example, in our prototype one compiler produces code appropriate for large-footprint testing and debugging; another produces smaller run-time code that can operate in a resource constrained environment.

8.2. Scalability

The most significant issue concerning the user interface in this project was the issue of scalability. In order to address this, the team used the concept of compositions, or logical groupings of entities that may be captured as new types in the system. These represent replicable structural patterns that can be instantiated and visualized as first class entities. For example, the wheel controller described in this paper could be captured as a composition, given a type name, and then instantiated anew at the discretion of the software engineer.

8.3. Fitting a Functional Style of Code into a State-Oriented Environment

In this project, a functional style of code for the essential code must be integrated with a state-oriented system, in which globally accessible stateful objects are the most important elements in both the architecture and implementation of the system. The functional style is important from a system point of view, but may not be enforced by the semantics of the target deployment language (e.g., C++). Therefore, while the essential functions are not themselves stateful, they manipulate stateful objects in the system. Thus persistent system state is factored out of the components that access or change that state. The benefit of this is that it becomes easy to understand the interactions between components. Pure functions have only two ways of interacting with the external world: they can return messages; or they may read or modify stateful objects that are passed in as arguments.

8.4. Interfacing to Essential Code

One of the key challenges that had to be addressed for the pure functions was finding the right interface to the pure

functions. The trade-off is that we required the interface to allow engineers full expressive power to implement the functions, while making the interfaces simple enough to understand and to reliably and simply generate code around them. If the interfaces were complex, this would require a greater number of connections in the architectural model, in addition to more complex code generation for each of these new types of connections.

We addressed this challenge in close consultation with JPL engineers, conducting interviews to simplify the interface and ensure that the engineers could still write the necessary code in the pure functions. In the end, we settled on the interfaces to pure functions essentially being value histories and messages. Once we got this right, implementing the code generators was quite straightforward.

9. Conclusions

This paper has described our experience of adapting two research threads to develop a new tool for NASA space systems engineers that allows them to move from high level system specifications to multiple deployments. Those threads were formal architectural representation within a domain-specific style for spaces systems, and pure functional representations of component capability.

Although the tool we developed addresses a particular audience and domain, we believe there are a number of important general lessons that can be applied more generally to create tools for moving from architectural designs to working systems.

1. *Augmentation (not replacement) of existing capabilities*: For this work to be successful we had to fit into an existing development process and use predetermined technologies. These included the architectural style rules, the state database, and the MDS framework code. This meant that we had to design our tools so that they worked with current practices, rather than mandating a set of new technologies that would fit seamlessly with our approach. This is something that will need to be addressed when applying the MDA approach more generally.
2. *Use of formal models*: The formal architectural style that we used is a benefit in this work because it is reasonably complex. This means that certain errors can be caught at the architectural modeling phase and not propagated to implementation. Less interesting styles would not produce the same benefits.
3. *Style specialization*: We used two related architectural styles for this work. The first style is a generic style that captures mission-independent rules about MDS composition. This was specialized to contain types representing particular specializations of the generic components for particular missions (for example, a WheelControllerT became a subtype of ControllerT), and additional rules about composition. In fact, we are currently experimenting with additional styles to factor out some additional implementation details, such as scheduling, rather than including it in the one substyle. This use of successively detailed styles should prove useful in other areas where abstract architectures need to be mapped into implementations.
4. *Separation of essential from incidental code*. A key to making the refinement successful was to find the sweet-spot between what goes into the architectural model and what goes into the code generators. We suggest that any similar refinement process will need to involve both architects and developers, in order to define this interface. However, once this work is done, the actual implementation of the process in another domain should be straightforward.

While still a prototype system, the work to date shows promise for becoming a major advance in space systems engineering. In particular, the combination of component specification clarity, automatic generation of code, and formal checking of architectural constraints provides a powerful combination of new capabilities beyond existing practices of NASA engineers.

References

[1] Aldrich, J., Chambers, C., and Notkin, D. ArchJava: Connecting Software Architecture to Implementation. Proc. ICSE 24, Orlando, Florida, 2002.

[2] Allen, R. and Garlan, D. A Formal Basis for Architectural Connection. ACM Transactions on Software Engineering and Methodology, 6(3), July 1997.

[3] Dvorak, D., and Reinholtz, K. Separating Essential from Incidentals, An Execution Architecture for Real-Time Control Systems. Proc. 7th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing. Austria, 2004.

[4] Garlan, D., Monroe, R., and Wile, D. "Acme: Architectural Description of Component-Based Systems." Foundations of Component-Based Systems, Cambridge University Press, 2000.

[5] Guttag, J.V., and Horning, J.J. (Eds) *Larch: Languages and Tools for Formal Specification*. Springer-Verlag, 1993.

[6] Hoare, C.A.R. Communicating Sequential Processes. Prentice Hall, 1985.

[7] Moriconi, M., Quian, X., and Riemenschneider, R. Correct Architecture Refinement. IEEE Trans. Soft. Eng. **21**(4), 1995.

[8] Object Management Group. MDA: The Architecture of Choice for a Changing World. http://www.omg.org/mda.

[9] Rasmussen, R. Goal-Based Fault Tolerance for Space Systems using the Mission Data Systems. Proc. 2001 IEEE Aerospace Conference, Big Sky, MT, 2001.

[10] Schmerl, B. and Garlan, D. Supporting Style-Centered Architecture Development. ICSE 26, Edinburgh, Scotland, 2004.

[11] Shaw, M., DeLine, R., Klein, D.V., Ross, T.L., Young, D.M., Zelesnik, G. Abstractions for Software Architectures and Tools to Support Them. *IEEE Transactions on Software Engineering*, 21(4):314-335, 1995.

[12] Woodcock, J. and Davies, J. Using Z: Specification, Refinement, and Proof. Prentice Hall International, 1996.