

A Model-Based Design Tool for Systems-Level Spacecraft Design

Brandon Eames, Allan McInnes, Jared Crace, Joe Graham
 Department of Electrical and Computer Engineering
 Utah State University
 4120 Old Main Hill, Logan, UT 84322-4120; 435-797-2841
 {beames, amcinnes, jcrace, jogr}@cc.usu.edu

ABSTRACT: It is standard practice to mathematically model and analyze the various subsystems that make up a spacecraft, to ensure that they will function correctly when built. However, the system-level behavior of the spacecraft is generally understood in much less rigorous terms. This leaves the spacecraft system far more vulnerable than the subsystems to unforeseen design errors which may not manifest themselves until the integration and test phase, when design changes are most expensive in terms of cost and schedule. In this paper, we present Spacecraft Design Workbench, an extensible graphical design tool built upon the Generic Modeling Environment (GME) tool infrastructure, and intended to allow spacecraft systems engineers to model and analyze proposed spacecraft system designs in a rigorous manner. The graphical models defined by our tool have an underlying formal behavior semantics rooted in the Communicating Sequential Processes process algebra, which permits these models to be analyzed using off-the-shelf tools. As a proof-of-concept, we provide a small example that illustrates the application of our tool to the specification of a simple scientific spacecraft.

INTRODUCTION

System-level design of modern spacecraft is highly complex. Designers must manage broad tradeoff analyses across multiple engineering disciplines in order to architect a balanced design which will perform the required system functions. Unfortunately, system-level spacecraft behavior does not receive the same level of rigorous definition and analysis as the subsystems. Consequently, the spacecraft system is far more vulnerable than the subsystems to unforeseen design errors which may not manifest themselves until the integration and test phase, when design changes are most expensive in terms of cost and schedule.

While this lack of rigor and formalism at the system level has not been a major problem so far, the increasing level of spacecraft autonomy, coupled with a shift towards building even smallsats out of decentralized networks of components, makes understanding the system-level behavior of a spacecraft more important than ever before.

In an effort to address this need, we are developing a graphical design tool called Spacecraft Design Workbench (SDW), which supports a visual modeling language based on domain-specific abstractions (functional flow block diagrams, functional dataflow diagrams, mode transition diagrams, etc.) appropriate for system-level spacecraft design. SDW is intended to

aid engineers in managing the complexity of a spacecraft design, by allowing them to precisely express the function and structure of a design, and to use different model views to individually examine different aspects of the design.

A myriad of visual languages and tools are currently available, both commercially and through academia. We contrast our approach with that taken by the vast majority of available tools by basing our tool design on a formal semantics defined using the Communicating Sequential Processes (CSP) process algebra. Formal semantics means that our models have a precise and unambiguous meaning. In addition, these models are suitable for analysis using existing tools developed for formal software analysis.

The graphical language is designed to make modeling and analyzing a proposed spacecraft design using our abstractions relatively straightforward, even for spacecraft systems engineers not well-versed in the underlying mathematical model.

The remainder of this paper is structured as follows. We first provide some background for our tool, including a brief review of related work. Subsequently, we describe the visual formalisms for modeling spacecraft behavior, followed by a brief discussion of the underlying CSP-based formal semantics. Finally, we provide short example of a simple spacecraft

behavior to illustrate the expressiveness and utility of our tool.

BACKGROUND

Related Work

A wide variety of tools and formalisms have previously been applied to the problem of specifying and understanding various aspects of spacecraft behavior. Several other existing tools provide visual languages that could potentially be applied to modeling spacecraft behavior. In this section we give a brief overview of these tools and formalisms, and discuss how our approach differs from these efforts.

A number of different groups within NASA have experimented with formal behavior modeling and analysis during the spacecraft development process. These experiments include using the Prototype Verification System theorem prover to verify fault protection software¹, and applying the *SPIN* model-checker to the verification of fault protection software^{2,3}, flight software concurrency^{4,5}, and domain models for autonomous planners⁶. However, all of these experiments focus on the behavior of spacecraft software, while the present work considers spacecraft system behavior in general. In addition, all of the experiments involve custom behavior models specific to their project and application, instead of working within an already defined modeling framework.

CSP has previously been used for modeling and verifying aspects of spacecraft behavior on at least two different projects^{7,8}. However, the use of CSP on both projects was limited to verifying specific aspects of single subsystems. Neither project sought to examine general system-level interactions.

Weiss⁹ and Ong¹⁰ explore general techniques for rigorous, spacecraft software specifications. While their work explicitly addresses behavior as well as function and performance, their focus is on techniques for generating component-based, reusable software requirements documentation, rather than on system modeling and analysis.

Several commercial tools are available for modeling, simulating and synthesizing spacecraft applications. Simulink¹¹ is a visual drag-and-drop design and simulation language which is widely used to model embedded control and signal processing systems. It offers an extensive library of components, together with tools for simulation and visualization of results. Stateflow extends Simulink with the ability to model hierarchical parallel finite state machines. The Real-

Time Workshop tool facilitates C code generation directly from models. Simulink is primarily focused on simulation-based verification of continuous dynamics, while our focus is on formal modeling and verification of event-driven system behavior.

The CORE¹² tool from Vitech allows users to capture requirements, together with architectural and behavioral views of a system. CORE offers the ability to capture and simulate enhanced functional flow block diagram (FFBDs). However, CORE does not offer the ability to model check or formally verify FFBDs.

The Statemate tool by Telelogic allows users to capture system behavior as a statechart, which can be verified through simulation and can then be automatically translated into C or Ada code. While the semantics of Statemate's formalisms are well defined¹³, there is no support for domain-specific modeling formalisms familiar to the spacecraft systems engineering community.

Ptolemy¹⁴ is a tool which allows the modeling, simulation and synthesis of systems. Ptolemy supports a variety of formalisms, called models of computation, allowing systems to be modeled heterogeneously. While Ptolemy does support a CSP-like model of computation, it offers simulation-based verification only.

As the preceding discussion indicates, the primary differences between our approach and other existing approaches are our consideration of system behavior, rather than just software behavior, and our focus on formal semantics and formal verification. Our tool supports domain-specific formalisms (mode transition diagrams, dataflow diagrams, and FFBDs) familiar to spacecraft designers, while offering a strong underlying formal semantics. The benefit of the formal, CSP-based semantics is the ability to perform refinement checking on system behavior, rather than simulation-based testing, to prove that a system will adhere to its specifications. Further, our tool infrastructure, which is based on the Generic Modeling Environment, is extensible, allowing the integration of other tools, formalisms and analyses.

The Generic Modeling Environment

The Generic Modeling Environment (GME)¹⁵, developed at Vanderbilt University, is a meta-programmable toolkit for building domain-specific visual modeling and analysis tools. A *domain* is an engineering discipline, such as signal processing, process automation, or spacecraft design. GME can be configured to support a particular *paradigm*, which

encodes the available visual constructs and relationships for a particular domain, as well as a set of well-formedness rules which are evaluated as a user creates a model. GME supports the partitioning of a system into a set of views, called *aspects*. The use of aspects facilitates the separation of concerns when modeling a system. Once GME is configured to support a particular paradigm, the user can use the drag-and-drop and interconnection features of GME to develop a visual model of a system, which conforms to that paradigm. Once a model is created, the user can invoke a translator program called an *interpreter* to perform domain-specific translations or analyses on the captured system. The paradigm, together with the corresponding interpreter, is what we refer to as a domain-specific modeling tool.

GME not only supports the use of pre-defined domain-specific languages, but also facilitates the development of new visual languages. A *metamodel* is a graphical model of a domain-specific language, capturing the entities, constructs and interrelationships available in a visual language, as well as well-formedness rules governing model construction. GME can be configured as a metamodeling tool, allowing users to develop their own domain specific metamodels. A generator tool translates the metamodel into a paradigm, which can then in turn be used to configure GME to support the domain concepts captured in the metamodel.

GME also supports the development of generators or interpreters for visual languages. It offers a software API for accessing a model database containing the domain-specific models created by the user. This API can be used to traverse, manipulate and dump these models without the tedium of dealing with traditional GUI programming. GME offers multiple language bindings for this software API (ex. C++, Java, C#), as well as a graphical modeling tool called GReAT¹⁶, allowing users to specify model-to-model transformations. The Universal Data Model (UDM)¹⁷ supports tool integration through the automatic translation between GME models and a paradigm-specific XML format. UDM also offers a software API for manipulating models.

In this paper we have developed a GME-based domain-specific visual language for modeling and verifying spacecraft behavior. In the next section, we discuss the GME-based metamodel and corresponding modeling tool. Following that, we provide an overview of the mathematical underpinnings of the modeling language.

A VISUAL MODEL FOR SPACECRAFT BEHAVIOR

Our initial tool prototype is developed using the GME infrastructure. The visual modeling language has been specified as a GME metamodel, and the analysis is based on a UDM-based C++ model translator. The GME metamodeling language¹⁸ is a variant of UML class diagrams. The presentation of the visual language definition assumes a passing familiarity with basic UML class diagram syntax.

The foundation of our visual language is based on an abstraction called Symbol. A symbol can be used to model an event, a piece of data, a control flag, a signal, or any other type of information of significance to the spacecraft behavior. Each symbol is assigned a name. Related symbols are collected into sets, referred to as a SymbolSet. Symbols form the basis of interaction between the various supported constructs of our visual language.

The Spacecraft Design Workbench currently offers three primary constructs to facilitate the modeling of system-level spacecraft behavior: Mode Transition Systems, Functional-Flow Block Diagrams (FFBDs) and dataflow descriptions. A heterogeneous modeling approach has been adopted, whereby a full system description can be composed using any or all of the supported formalisms.

Mode Transition Systems

We model spacecraft modal behavior using a classical state transition system, a description of which is shown in Figure 1. A ModeTransitionSystem is composed of a set of States, Transitions and a Start object. TransConn is an association between these objects, visualized as a directed connection between two objects. This model is equivalent to a simple finite state machine, with no parallelism or hierarchy. A Transition models an event raised either internally or externally to the system, which triggers the state transition system to transition from one state to another. Events are modeled as Symbols. A Transition object is associated with a Symbol, with the relationship being analogous to a pointer or alias. The alias to the event becomes the Transition object in the state transition system. Each State is also associated with a Symbol, the reason for which will be explained subsequently. The initial state is taken to be that state which is connected to the Start object.

Several modeling constraints (not shown) are added to this specification. These constraints require that only one Start object can be included in a

ModeTransitionSystem, and that a TransConn must associate a State with a Transition, a Transition with a State, or a Start object with a State (i.e. Transition-to-Transition connections are not allowed).

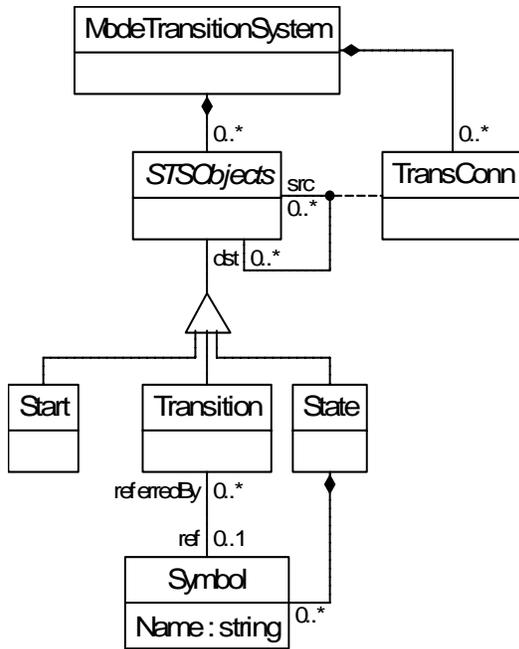


Figure 1. Description of Mode Transition System Model

A Mode Transition System can be used to model the modal behavior of a spacecraft. As will be shown later, States can be associated with operations which the spacecraft performs while in that state.

Functional Flow Block Diagrams

Functional Flow Block Diagrams¹⁹ (FFBDs) are a traditional notation for specifying spacecraft system behavior^{20,21}. FFBDs prescribe the sequence in which different functions should be executed, and allow the definition of parallel and alternative sequences of functions, as well as loops. The Spacecraft Design Workbench supports behavior modeling using hierarchical FFBDs. The FFBDFunction class models a function, which can be composed of other FFBD specifications. An FFBDFunction can contain other FFBDFunction objects, ChoiceFunctions, and Point objects. A Point object models a point of connection in the FFBD. We require all FFBDFunctions which are not leaves in the FFBD hierarchy to contain two Point objects, modeling the beginning and ending execution points for that function. AndPoints model the beginning and ending of an AND construct, which represents parallel activities in an FFBD, while OrPoints model the same for OR constructs, which

represent possible alternative activities. PointObjConn models associations between Points and Sequencable objects, which are visualized as connections. A ChoiceFunction models a binary choice, and is used as the condition in an iteration construct, as well as the selection condition in the selection construct.

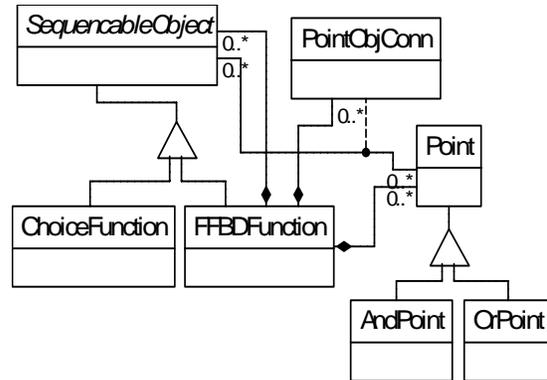


Figure 2. Description of FFBD Specification

Modeling constraints (not shown) are included in the language specification to impose well-formedness rules on the various FFBD constructs. For example, we require a ChoiceFunction to have a single input connection and two output connections (one for each outcome of the choice). A well-formed OR construct consists of two OrPoints, one modeling the beginning and one the end of the construct. The beginning OrPoint must have a single input connection, while the ending OrPoint must have a single output connection. The number of output connections of the beginning OrPoint must match the number of input connections of the ending OrPoint. A similar modeling constraint is imposed on AndPoints.

Figure 3 depicts an example of an FFBD drawn with SDW. The graph contains a loop, whose body contains an AND block with three parallel paths, Fn1, Fn2 and Fn3. The diamond block after the terminating AndPoint represents the ChoiceFunction, which models the loop termination condition. The two output connections of the ChoiceFunction (labeled “Go” and “NoGo”) model the execution paths which can be taken in response to the condition evaluation. While the condition evaluates to false, iteration continues. When the condition evaluates to true, the loop breaks and the FFBD terminates.

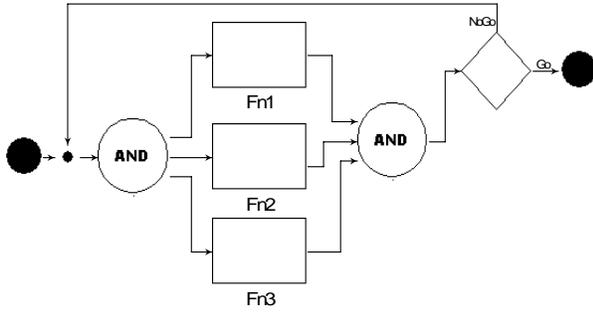


Figure 3. Example FFBD Model

The conditions under which a choice resolves to either the “Go” or “NoGo” branches is often only loosely defined in a purely graphical FFBD. However, since an SDW behavior model is intended to be an *executable* specification, we require a precise specification of the conditions for choosing which branch to follow. We now discuss how to specify the condition using the ChoiceFunction construct. The ChoiceFunction models a Boolean condition that is evaluated at a branch point in the graph. In SDW, a condition is modeled as a query of a StateObject variable. A StateObject variable is mutable location in memory which can be assigned values from a fixed, discrete domain. We model the domain of a StateObject as a set of symbols. The evaluation of a condition compares the current symbol stored in the StateObject variable against the condition’s “GoSet.” The GoSet specifies which subset of the symbols in the StateObject’s symbol set will cause the ChoiceFunction to select the “Go” branch. Symbols that are not contained in the GoSet will cause the ChoiceFunction to take the “NoGo” branch. The user specifies the GoSet visually in the model, using connections between symbols in a symbol set to indicate inclusion in the GoSet. The structure of the model for ChoiceFunction composition is shown in Figure 4.

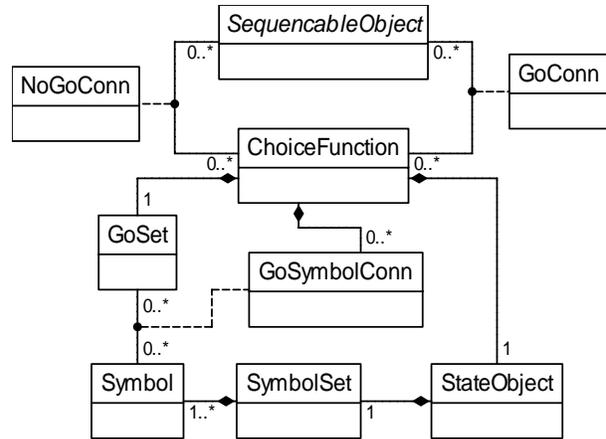


Figure 4. Model of ChoiceFunctions

A ChoiceFunction in an FFBD has two output connections, GoConn and NoGoConn, as depicted in Figure 4. The GoConn associates the ChoiceFunction with a SequencableObject (another object in the FFBD), to where execution should proceed if the condition evaluates to true. The NoGoConn associates the ChoiceFunction object with the FFBD object to where execution should proceed when the condition evaluates to false. Examples of these connections can be seen in Figure 3, corresponding to the connections labeled “Go” and “NoGo.”

Figure 5 depicts the internals of a ChoiceFunction object. SSet is a SymbolSet object, corresponding to the set of symbols associated with the ChoiceFunction’s associated StateObject. The SymbolSet contains three symbols, appearing as port objects on the edge of the SSet icon: A, B and C. In this case, the symbols A and C in SSet are connected to the GoSet object, implying that on evaluation, if the ChoiceFunction’s StateObject currently contains the symbol A or C, then the choice evaluates to true, and execution proceeds along the “Go” connection of the ChoiceFunction. Otherwise, the “NoGo” connection is followed.

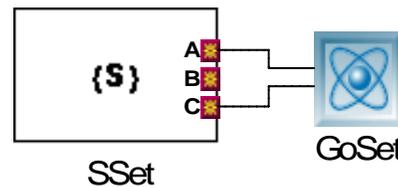


Figure 5. Example internals of a ChoiceFunction

Dataflow Diagrams

SDW also supports the definition of dataflow diagrams as a behavioral specification. A dataflow diagram is a graphical depiction of the functions of a system, together with their data dependencies. We adopt an approach similar in syntax to existing styles of dataflow specifications^{22,23}. Figure 6 depicts the metamodel description of a dataflow diagram. The basic construct of a dataflow diagram is a Function. A Function interacts with other functions via Ports. Ports of a function are associated with Ports of other functions via the DFConn association, which is visualized as a connection. Data flows across connections to ports. We abstract the data tokens passed between ports as Symbols. Each Port is associated with a SymbolSet, modeling the set of data token types which may be sent/received via that port. A data communication between two ports consists of the transmission of a single symbol from the source to the destination.

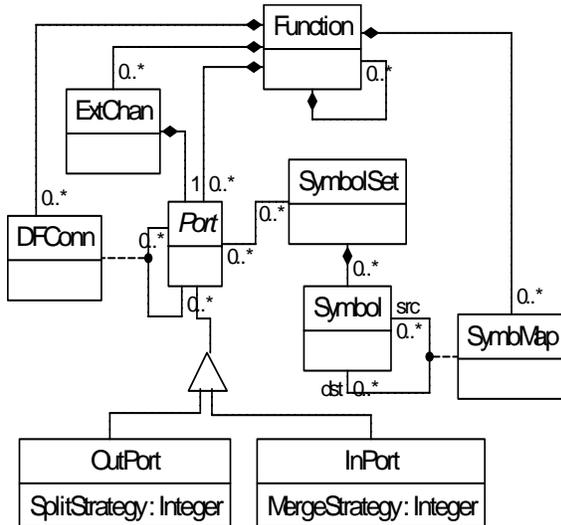


Figure 6. Dataflow diagram description

SDW supports both fan-in and fan-out dataflow connections; however a strategy for managing multiple symbols in each case must be selected. For an input port which acts as the destination of multiple connections, the user can select a MergeStrategy for collecting and merging data from each or any input line into a block. The input strategies supported are All, Any, and Custom. The All strategy implies that the system must collect a single symbol from each input connection of the input port, and build a symbol tuple from each of the collected symbols. This tuple is then passed as a single symbol to the owner of the input port. The Any strategy simply takes any input symbol on any input line and transmits it. The Custom strategy is a user-defined strategy, wherein the user provides a

custom block of CSP code to define how symbols are collected and transmitted. The semantics of Output Ports and SplitStrategy are similarly defined, except that for the All output strategy, we assume that the function passes to the output port a tuple containing one symbol for each output connection incident on the OutputPort.

Our dataflow model is hierarchical: simple functions can be composed to create more complex functions. Leaf-level functions in the hierarchy have a user-defined functionality, specified through the SymbMap associations. As mentioned previously, every Port of a function is associated with a SymbolSet, specifying the set of data tokens which are allowed to be sent or received by that port. A leaf-level function in the dataflow hierarchy is constrained to contain exactly one input port and one output port. Multiple inputs to a leaf function are supported through the use of a MergeStrategy, while multiple outputs are accommodated via a SplitStrategy. A leaf function is simply a mapping of the SymbolSet associated with its input port to the SymbolSet associated with its output port. We specify this mapping graphically as a set of associations between the symbols of each respective symbol set. Figure 7 provides an example depiction of the mapping between the input (Cmd) and output (Attitude) symbol sets of a leaf-level function.

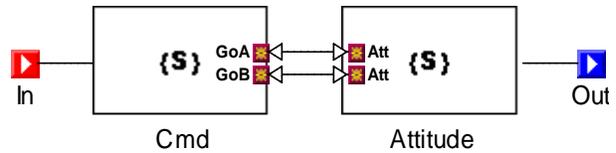


Figure 7. Example Leaf Function Mapping between Input and Output Symbol Sets

In addition to allowing dataflow connections between different functions to be defined, SDW supports the concept of an external channel, modeling the interaction of the system with its environment. The ExtChan class in Figure 6 models the external channel. An external channel is allowed to contain a single port: either an input or an output port (modeling a unidirectional external communication interface).

An example dataflow diagram specification is provided in Figure 8. The example shows a Guidance, Navigation and Control (GNC) function, which stores the current spacecraft attitude into a StateObject variable called SpacecraftAttitude. The ProcessImage function reads the current attitude as it receives an image from the external port ImagePort. This function performs some on-board image processing, then issues the processed image to a CompressImage function,

which issues the compressed image to the telemetry sub-system, modeled here as an external port called ImageDownlink.

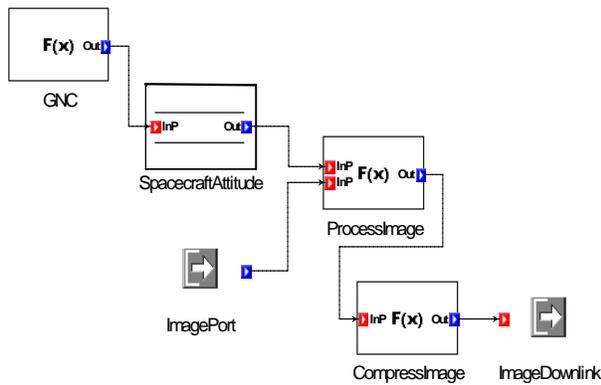


Figure 8. Example Dataflow Diagram Specification

Heterogeneous Modeling

As mentioned previously, SDW supports compositional modeling using multiple modeling disciplines. We have illustrated our visual models for capturing dataflow descriptions, mode transition systems and FFBDs. In this section we describe how to compose models conforming to each of these sub-domains when creating a full system model. Figure 9 models how the interactions between sub-domains are captured.

A key part of defining the relationships between different sub-domains is the use of *references*. A reference acts as a pointer or alias to another block in a model. To support heterogeneous models, we create two new types of objects, both of which are reference objects. FFBDReference is an object which refers to an FFBDFunction object. FunctionReference is a reference object which refers to a Function in a dataflow diagram. We allow States in our Mode Transition System to contain references to FFBD functions. We say that a reference to a dataflow function can be used in an FFBD, as if it were an FFBDFunction (via the inheritance relationship between FFBDFunction and FunctionReference).

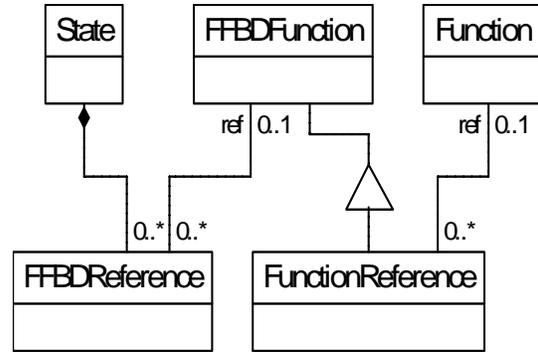


Figure 9. Integration of Sub-Domain Models to Allow Heterogeneous Modeling

The semantics allowed by these constructs are as follows. The containment of an FFBDReference within a State implies the triggering of the FFBD referred to by the FFBDReference on entrance to the state. The contained FFBD must terminate before the system can transition from that state. In the case where a transition event has been received by the mode transition system requiring the transition from the current state, the system enters an implicit wait state, pending the termination of the contained FFBD. Due to the implicit waiting for termination, the FFBD must be carefully designed so as to not block the system from transitioning between modes. Although not currently supported, we plan to introduce semantics allowing a termination signal to be issued to contained FFBDs when a mode transition signal has been received.

The integration of dataflow functions into the FFBD specification facilitates the explicit modeling of sequencing and control between functions, separate from the dataflow specification. The FFBD construct imposes an execution constraint on the corresponding dataflow function. A dataflow function can only execute when the execution of the containing FFBD indicates that it is appropriate for the dataflow function to do so. For example, an FFBD modeling a sequence of functions imposes a sequencing constraint on the execution of those functions, and their execution must adhere to the specified order in the FFBD, regardless of the data dependencies captured in the dataflow diagram. Consistency between the dependencies in the dataflow diagram and the sequencing constraints imposed by an FFBD is enforced through the model checking and refinement checking of the generated CSP model of the system.

Model Interpretation

The previous sections detail the metamodel supported by the SDW visual language for modeling spacecraft

behavior. While systems can be captured using the GUI front-end, the true power of SDW is seen in the automatic translation of a user-specified model into the equivalent underlying mathematical representation, based on CSP, coupled with the use of commercial tools to perform various checks and analyses on the modeled system. The metamodel has been designed to facilitate the automatic generation of CSP, and we anticipate the completion of the automatic translator tool shortly. However, manual translations of the visual language into CSP have illustrated the feasibility of our approach. The specification of the translation algorithm is beyond the scope of this paper. However, in the next section, we provide a brief discussion of the underlying mathematical model of SDW, and in the Example section, we illustrate how CSP code corresponding to the visual model can be used to verify the behavior of the system.

MATHEMATICAL MODEL

The mathematical basis of SDW is provided by the process algebra Communicating Sequential Processes (CSP)^{24,25}. To make the translation from graphical notations to CSP easier, we have developed a modeling framework that provides structured abstractions for defining various kinds of spacecraft behavior models. Models expressed within this framework can be analyzed using commercially available tools to ensure that they possess various desirable properties.

Communicating Sequential Processes

We elected to use CSP as the mathematical basis for SDW for several reasons. First, most spacecraft are inherently concurrent, and CSP was expressly designed to describe the behavior of concurrent systems. Second, CSP has a well-developed theory which provides a powerful set of tools for understanding and analyzing spacecraft behavior. Finally, CSP has a long history of practical application in the software industry, and commercially available analysis tools. The specific dialect of CSP that we use is CSP_M ²⁶. This is a machine-readable (ASCII) dialect supported by most modern CSP tools. It also has the advantage of having a fully-defined semantics which adds a language of functions, sets, and sequences to the traditional CSP process language.

The fundamental building-blocks of the CSP language are *events*. Events may be abstract names representing some shared phenomena, or communications events on named channels. A CSP process expression specifies the sequences in which some set of events may be observed to occur. Process expressions can be

combined and manipulated using CSP's process operators to form new process expressions, allowing complex behavior descriptions to be built up as a composition of simpler parts. The following example is a small taste of the CSP process language:

$$\begin{aligned} P &= a \rightarrow b \rightarrow P \\ Q &= (c \rightarrow Q) [] (d \rightarrow a \rightarrow Q) \\ R &= P [] \{a\} [] Q \end{aligned}$$

Process P can engage in the event a , and then the event b , following which it repeats itself. Q initially offers a choice of either event c or event d . If event c occurs, Q simply recurses and again offers a choice of c or d . If event d occurs, Q engages in a , and then recurses. R is a parallel composition of P and Q , and requires the two processes to synchronize on the occurrence of a . That is, event a cannot occur unless both processes are prepared to engage in a .

A key part of the theory of CSP is the idea of *refinement*. A process P' is said to refine a process P if the behavior represented by P' is, within some semantic model, a subset of the behavior represented by P . CSP includes several different semantic models that capture different aspects of the behavior of a process. The simplest of these models is the *traces* model, in which a process is characterized by the set of all possible traces (i.e. sequences of events) it can be observed to perform. The process

$$Q' = d \rightarrow a \rightarrow Q'$$

has traces $\{\langle \rangle, \langle d \rangle, \langle d, a \rangle, \langle d, a, d \rangle, \langle d, a, d, a \rangle, \dots\}$. Because the traces of Q' are a subset of the traces of the process Q defined above, we can say that Q' trace-refines Q :

$$Q \ [T= \ Q'$$

However, it is not the case that

$$Q' \ [T= \ Q$$

because the traces of Q include sequences involving the event c , while the traces of Q' do not include this event.

A wide variety of behavior properties can be expressed in terms of assertions about process refinements in different semantic models. Tools such as the FDR2 model-checker²⁷ can be used check the validity of these assertions.

Modeling Framework

CSP is a flexible, but relatively unstructured language with which to describe spacecraft behavior. To impose some structure upon the problem of spacecraft behavior specification, we have developed a CSP-based modeling framework that provides a set of predefined higher-order specification components²⁸. These components form the semantic foundation of the visual modeling language discussed previously.

The modeling framework can be understood in terms of three basic categories of processes: behavior components, shared resources, and constraints.

Behavior components represent different pieces of the overall behavior of a spacecraft system. These may be instantiations of predefined generic behaviors, such as simple finite state machines or functional data transformers, or more complex units of behavior built up from other components, states, and constraints. For example, the generic process

```
LiftF(in, out, f) = in?x -> out!f(x)
-> LiftF(in, out, f)
```

takes an input on a channel *in*, outputs on a channel *out* the value of applying the function *f* to the input value, and then awaits a new input. This process is instantiated by supplying it with input and output channels, and a function definition which may be either an explicit set-based mapping from input values to output values, or an expression in CSP_M's functional language. Each leaf function in an SDW dataflow diagram can be translated into a `LiftF` process instantiated with a function derived from the symbol-to-symbol mapping that is part of the definition of the leaf function.

Although it is possible to allow behavior components to interact directly with each other through parallel composition and synchronization, such direct interactions are not always desirable or realistic. *Indirect* interactions between behavior components can be captured using shared resources, and constraints.

Shared resources are state-bearing processes that may be affected by, or influence, multiple behavior components. For example, the attitude state of a spacecraft may be affected by both attitude-command behaviors and fault responses, and may influence the available power produced by a power-generation behavior. A simple state resource can be represented by the process

```
State(set, get, trans, val) =
```

```
(get!val
  -> State(set, get, trans, val))
[]
(set?val'
  -> if (val' != val)
      then (trans!val' ->
            State(set, get,
                  trans, val'))
      else
        State(set, get,
              trans, val))
```

This process permits multiple behavior components to read the value of the currently stored value, or to set a new value. The channel *trans* permits the implementation of an observer pattern, in which interested behavior components can be notified of state value changes. The `StateObjects` that appear in SDW dataflow and mode transition diagrams can be translated into instantiations of the `State` process. Other, more complex types of shared resources can also be defined within our CSP framework. These resources include buffers, which permit various kinds of asynchronous communication between behavior components, and quantitative resources, which can be used to represent consumables such as propellant or battery charge. However, these more complex resources are not yet supported by the SDW modeling language.

Constraints are processes which constrain the relative ordering of events from different behavior components and shared resources. For example, an FFBD can be understood as a constraint on the sequencing of different function components. Similarly, a restriction on the behavior components that are allowed to be active in a particular system mode can be understood as a constraint on the initiation of different behaviors.

A complete system behavior specification is structured as a parallel composition of behavior components, shared resources, and constraints.

```
SystemSpec =
  BehaviorComponents
  [|IFset1|]
  SharedResources
  [|IFset2|]
  Constraints
```

where the processes `BehaviorComponents`, `SharedResources`, and `Constraints` represent aggregations of the system behavior components, resources, and constraints respectively, and the `IFsets` are the interfaces between these different elements.

A specification of the type just described can be considered a "black-box" specification of the spacecraft behavior, since it contains specification constructs such as constraint processes. It is also possible to assemble a "white-box" behavior specification. This can be done by developing a collection of behavior components that represent the behavior of different spacecraft subsystems, and then connecting these behavior components together such that the resulting process network resembles a spacecraft functional block diagram. The spacecraft system behavior defined by a "white-box" specification is an emergent property of the interactions between the behavior components representing each subsystem. It should be the case that

BlackBoxSpec [= WhiteBoxSpec

Verifiable Properties

Given a visual specification that has been transformed into CSP, there are a variety of specification properties that we can verify. Perhaps the simplest type of verification is to simply step through an execution of the specification, using a tool such as the *ProBE* process animator²⁹. This allows specifiers to check that the specified behavior matches their intentions.

More rigorous checks can be carried out using *FDR2*, which can automatically explore all possible executions of the behavior specification. The most basic check we might carry out with *FDR2* is a verification of specification consistency. Such verification ensures that none of the specified constraints in a black-box specification conflict with each other, and that the interfaces between different subsystems in a white-box specification are consistent on each side of the interface. Consistency verification can be performed as a check for deadlock.

Other properties that might be verified using *FDR2* include:

- Refinement of a black box specification by a white box specification
- Parsimony: all specified inputs and outputs are actually used
- Operational correctness: the specified design is capable of performing various mission scenarios
- Robustness: unexpected inputs, or the failure to receive an expected input, does not cause mission-ending failure

EXAMPLE

We now consider a simple specification example that illustrates the use of most of the graphical constructs defined in the current version of SDW, and demonstrates how the formal semantics underlying the graphical language permit a specification to be analyzed in a rigorous manner. This example is intended to be clear, rather than detailed or highly realistic, and focuses on just a few spacecraft functions.

Visual Specification

Our example is a very simple behavior definition for a notional scientific spacecraft. The spacecraft is assumed to have two modes of operation: a "standby" mode, and a "science" mode. Transitions between these modes are triggered by commands. The mode transition diagram in Figure 10 defines this behavior. Commands are assumed to originate from an external source, such as a ground station.

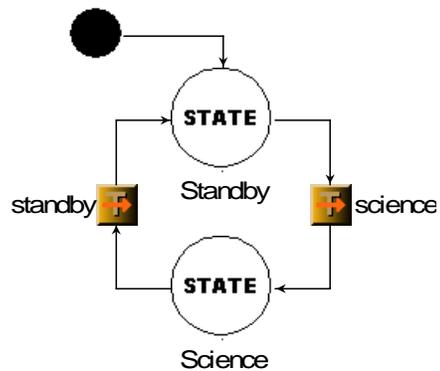


Figure 10. Mode Transition Diagram with Standby and Science modes

When in the "Standby" mode, the spacecraft simply repeatedly processes attitude commands from the ground station. An FFBD that precisely defines the spacecraft behavior in "Standby" mode is shown in Figure 11.

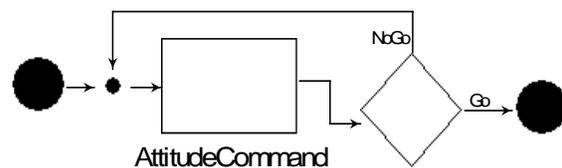


Figure 11. FFBD executed in Standby Mode

When in "Science" mode, the spacecraft may at any time either process an attitude command, or take science measurements. After executing one of these

functions, the spacecraft is again ready to execute either function. The FFBD that describes this behavior is shown in Figure 12.

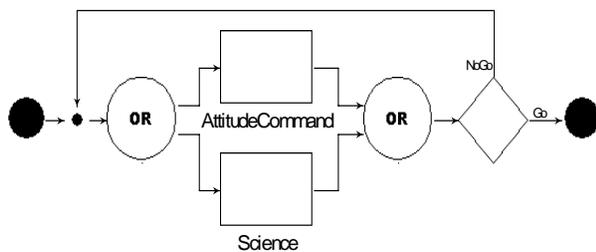


Figure 12. FFBD executed in Science Mode

The dataflow view provides a way to specify the interactions between the different functions that appear in the FFBDs. As the dataflow diagram in Figure 13 indicates, attitude commands are received from a source external to the spacecraft (AttitudeCmdPort), as are science measurements (via ScienceDataPort). The AttitudeCommand function alters the attitude state, which in turn forms an input to the Science function. The Science function combines measurements and attitude data, and outputs the result through a downlink channel.

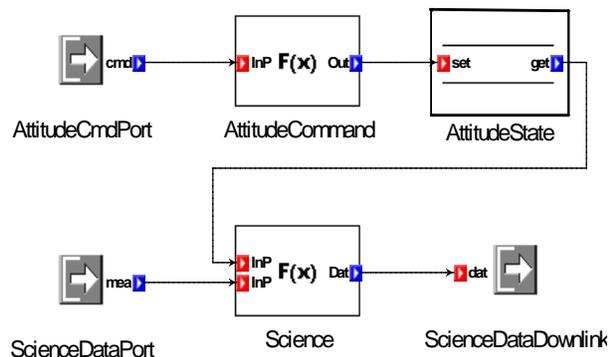


Figure 13. Dataflow diagram and interaction with external ports

The combined behavior defined by the mode transition, FFBD, and dataflow views constitutes an executable specification of the behavior of the scientific spacecraft. Using the ProBE process animator, we can explore a CSP_M translation of the graphical model. By stepping through different sequences of behavior execution we can gain a better understanding of how the fragments of behavior defined in the different model views interact with one another to produce the complete spacecraft behavior, and check that the behavior implied by the combined specification is actually the behavior we intended to specify. A sample session with ProBE is shown in Figure 14.

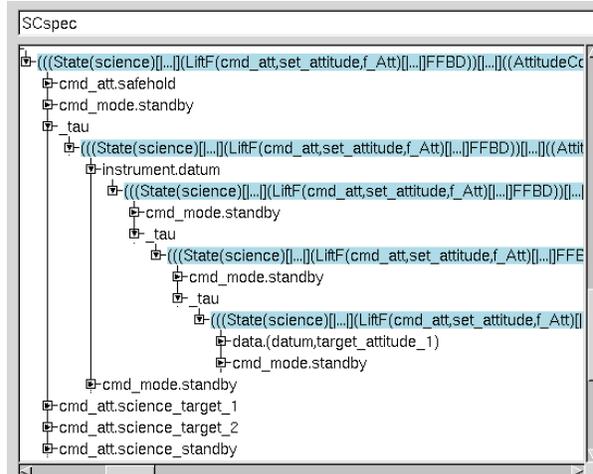


Figure 14. Exploring Different Execution Sequences using ProBE

Verification

Suppose that one requirement for the simple scientific spacecraft is that all downlinked science data must have been sampled when the spacecraft is in an attitude in which it is directed at a scientific target of interest. Although we can perform an *ad hoc* check for satisfaction of this requirement by stepping through a CSP process description derived from the visual specification with a tool such as ProBE, stepping through only a few of the many possible execution sequences will not necessarily uncover an execution sequence that fails to meet the requirement. A more rigorous and complete way to check that our behavior specification satisfies the requirement is to formalize the requirement as a refinement check that can be analyzed using FDR2. At present, these refinement checks must be formulated manually.

The refinement check will necessarily involve spacecraft attitude. For simplicity, the behavior model abstracts the possible attitudes to which the spacecraft can be commanded into four qualitatively distinct states: sun-pointing, earth-pointing, and two attitudes representing scientific targets. In CSP_M, we represent these attitude states as the set of symbols

```
{sun_pointing, earth_pointing,
target_attitude_1, target_attitude_2}
```

The CSP_M description of the science function is a process that outputs 2-tuples which represent a combination of an instrument measurement and an attitude. We can thus define the set of acceptable science outputs, using CSP_M's set comprehension notation, as

```
AcceptableScienceOutput =
  {data.(meas,a) |
   meas <- Measurement,
   a <- {target_attitude_1,
         target_attitude_2}}
```

That is, an acceptable science output can consist of any measurement value drawn from the set `Measurement`, but can only contain one of the two attitude values `target_attitude_1` and `target_attitude_2`, which represent scientific targets.

Given this definition of acceptable outputs, the refinement check to verify the requirement we are interested can be expressed in *FDR2*'s input language as the assertion

```
assert RUN(AcceptableScienceOutput)
  [T=
   SCspec\diff(Events, {|data|})
```

where `SCspec` is the process description produced by the SDW interpreter, `RUN(A)` is a process that can always engage in any event from the set `A`, and the notation `\diff(Events, {|data|})` "hides" all events except those involving the data channel. Hidden events are ignored for the purposes of the refinement check. In English, the assertion of refinement is effectively a claim that every possible sequence of science outputs (i.e., events involving the channel data) that can be produced by the `SCspec` process will consist only of values from the set `AcceptableScienceOutput`.

Checking an assertion of refinement is a simple matter of loading a `CSPM` script containing the `SCspec` process description and the refinement assertion into *FDR2*, and instructing *FDR2* to check the assertion. *FDR2* indicates whether a particular assertion is *true* or *false* by marking it with either a tick (*true*) or a cross (*false*). As Figure 15 shows, in the case of the refinement assertion defined above *FDR2* finds that in fact the assertion is *false*: the requirement on downlinked science data is not met.

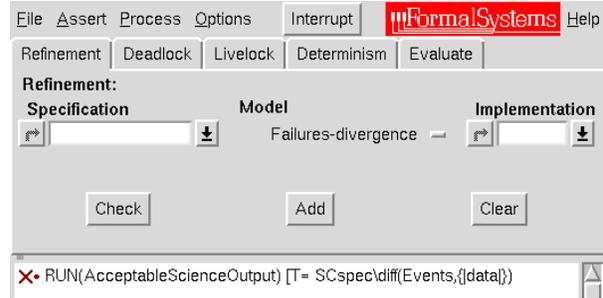


Figure 15. Failed Refinement Check in *FDR2*

The problem is that in some of the execution sequences permitted by the specification the spacecraft may not be in an acceptable attitude before it enters science mode, or it may be commanded into an unacceptable attitude during science mode. A simulation of the behavior specification might never execute one of these problematic sequences, and thus never reveal that the requirement is not met. In contrast, *FDR2* considers *every* possible execution of the specified spacecraft behavior, and for any failed assertion about those execution sequences generates "counterexample" traces that can be used to diagnose exactly what is wrong with the behavior specification.

Figure 16 shows *FDR2*'s process debugger displaying one of the 12 counterexamples found during the failed refinement check. The left pane of the debugger contains the tree of processes that make up the specification, and can be used to select an individual process for inspection. The right pane contains the portion of the counterexample trace generated by the currently selected process (the entire `SCspec` process in this case). The counterexample indicates that the sequence of commands shown in the trace can lead to the science function output

```
data.(datum,earth_pointing)
```

This output contains an attitude value that is not one of the two science target attitudes, and thus does not meet our hypothetical requirement.

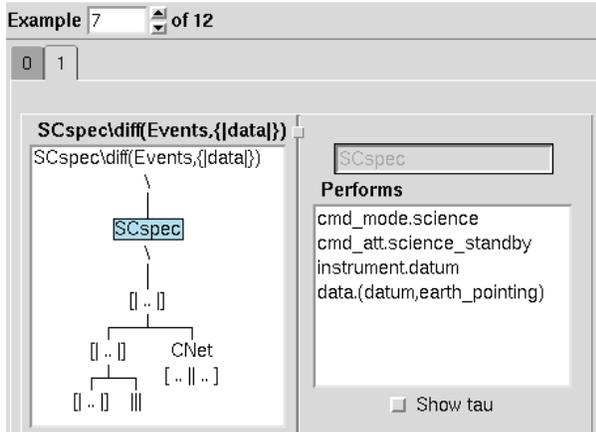


Figure 16. FDR2 Process Debugger Displaying a Counterexample Trace

Having identified that a problem with the specification exists, and explored the cause of the problem using *FDR2*'s process debugger, we are now in a position to consider possible solutions. One possibility is to define an operational solution. That is, we can assume that the ground station will only issue sequences of commands that produce the required behavior. An alternative approach is to revise the spacecraft behavior specification such that it meets the requirement. For example, we could specify that some autonomy function places the spacecraft in an appropriate attitude before every execution of the science function.

For the current example, we choose to implement an operational solution. We can encode our assumption about the behavior of the ground station as the process

```
GS = (|~| a:AttitudeCommand @
      cmd_att!a -> GS)
      |~|
      (cmd_att!science_target_1
       -> cmd_mode!science
       -> GS')
```

```
GS' = (|~| a:{science_target_1,
             Science_target_2} @
      cmd_att!a -> GS')
```

This process essentially states that the ground station is free to command any possible attitude while the spacecraft is in standby mode, but will always command the spacecraft into a science attitude prior to commanding a transition to the science mode, and will only command the spacecraft into acceptable science attitudes from then on.

Incorporating our assumption about ground station behavior into the refinement check is matter of

composing the GS process description in parallel with *SCspec*. The resulting refinement assertion is

```
assert RUN(AcceptableScienceOutput)
  [T=
    (SCspec
     [ | { | cmd_att, cmd_mode | } | ]
     GS) \ diff(Events, { | data | })
```

As shown in Figure 17, checking this assertion in *FDR2* reveals that the science output requirement is now met. Our newly defined ground station operational requirement ensures that a spacecraft with the behavior we have specified will not produce any undesirable outputs.

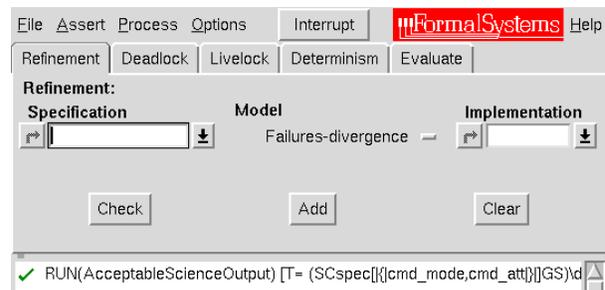


Figure 17. Result of the Refinement Check once an Operational Assumption is Included

CONCLUSIONS

We have developed Spacecraft Design Workbench, a prototype domain-specific modeling tool for spacecraft design. The modeling language used in this design tool has an underlying formal semantics that permits system-level models of proposed spacecraft behavior to be rigorously analyzed for conformance to requirements. The current version of SBW supports the specification of system-level behavior in terms of mode transition diagrams, functional flow block diagrams, and dataflow diagrams, and specification analysis via translation of these diagrams to CSP_M . At present, refinement checks and assumptions must be manually defined. We have shown an example illustrating not only the graphical front end, but the utility and power of formal analysis in catching subtle errors in system specification, even for a trivial example.

In the future, we plan to continue the development of SDW to include functional block diagrams and subsystem interactions, scenario definition using message sequence charts, more expressive modeling of state transition systems, and definition of refinement checks. We further plan to extend the system

specification tool to facilitate the capture and exploration of sets of alternative behaviors and specifications, and use a design space exploration tool based on finite domain constraints to explore and prune the design space based on user-provided system requirements. Other future directions for this tool include interfacing to Matlab/Simulink to facilitate simulation of system dynamics, and the development of a system synthesis tool based on these system specifications.

Acknowledgments

Allan McInnes is supported by a Space Dynamics Laboratory "Tomorrow" Ph.D. Fellowship.

References

1. Easterbrook, S., et al., "Experiences using lightweight formal methods for requirements modeling," *IEEE Transactions on Software Engineering*, vol. 24(1), pp. 4-14, 1998.
2. Feather, M.S., S. Fickas, and N.A. Razermary, "Model-checking for validation of a fault protection system," Proceedings of the 6th International Symposium on High Assurance System Engineering, 2001.
3. Barltrop, K.J. and P.J. Pingree, "Model checking investigations for fault protection system validation," Proceedings of the NASA SMC-IT Conference, 2003.
4. Havelund, K., M. Lowry, and J. Penix, "Formal analysis of a space craft controller using SPIN," NASA Ames Research Center TR-1770, 1998.
5. Gluck, P.R. and G.J. Holzmann, "Using SPIN model checking for flight software verification," Proceedings of the 2002 IEEE Aerospace Conference, 2002.
6. Smith, M.H., G.C. Cucullu, G.J. Holzmann, and B.D. Smith, "Model checking autonomous planners: Even the best laid plans must be verified," Proceedings of the 2005 IEEE Aerospace Conference, 2005.
7. Mota, A. and A.C.A. Sampaio, "Model-checking processes with states: An industrial case study," Proceedings of the Anais do XII Simposio Brasileiro de Engenharia de Software, 1998.
8. Schlingloff, H., O. Meyer, and T. Hulsing, "Correctness analysis of an embedded controller," Proceedings of the International Conference on Data Systems in Aerospace (DASIA 99), 1999.
9. Weiss, K.A., E.C. Ong, and N.G. Leveson, "Reusable specification components for model driven development," Proceedings of the International Conference on Systems Engineering (INCOSE 03), 2003.
10. Ong, E.C. and N.G. Leveson, "Fault protection in a component-based spacecraft architecture," Proceedings of the International Conference on Space Mission Challenges for Information Technology, 2003.
11. <http://www.mathworks.com>
12. <http://www.vtcorp.com/>
13. Harel, D. and A. Naamad, "The STATEMATE Semantics of Statecharts," *ACM Transactions on Software Engineering and Methodology*, vol. 5(4), pp. 293-333, October 1996, 1996.
14. Hylands, C., et al., "Overview of the Ptolemy Project," University of California, Berkeley, Technical Memorandum UCB/ERL M03/25, July 2 2003.
15. "GME User's Manual," Vanderbilt University March 2004.
16. Karsai, G., A. Agrawal, F. Shi, and J. Sprinkle, "On the Use of Graph Transformation in the Formal Specification of Model Interpreters," *Journal of Universal Computer Science*, vol. 9(11), pp. 1296-1321, 2003.
17. Magyari, E., et al., "UDM: An Infrastructure for Implementing Domain-Specific Modeling Languages," Proceedings of the The 3rd OOPSLA Workshop on Domain Specific Modeling, Anaheim, CA, 2003.
18. Ledeczi, A., et al., "Composing domain-specific design environments," *Computer*, vol. 34(11), pp. 44-+, NOV, 2001.
19. Oliver, D.W., T.P. Kelliher, and J.G.J. Keegan, *Engineering Complex Systems with Models and Objects*: McGraw-Hill, 1997.
20. Davis, D., et al., *SMC Systems Engineering Primer & Handbook*, 2nd ed: USAF Space and Missile Systems Center, 2004.

21. Shishko, R., et al., "NASA Systems Engineering Handbook," NASA SP-6105, 1995.
22. Yourdon, E., *Modern Systems Analysis*. Englewood Cliffs, NJ: Prentice-Hall, 1989.
23. Hatley, D.J. and I.A. Pirbhai, *Strategies for Real-Time System Specification*. New York: Dorset House, 1987.
24. Hoare, C.A.R., *Communicating Sequential Processes*. Englewood Cliffs, New Jersey: Prentice Hall, 1985.
25. Roscoe, A.W., *The Theory and Practice of Concurrency*. Englewood Cliffs, New Jersey: Prentice Hall, 1998.
26. Scattergood, J.B., "The Semantics and Implementation of Machine-Readable CSP," D.Phil. Oxford University, 1998
27. Gardiner, P., et al., "Failures-Divergence Refinement: FDR2 User Manual," Formal Systems (Europe) Ltd. May 2003.
28. McInnes, A.I., "A Formal Approach to Specifying and Verifying Spacecraft Behavior," Ph.D. Dissertation, Utah State University, to appear
29. FSEL, "Process Behaviour Explorer: ProBE User Manual," Formal Systems (Europe) Ltd. 2003.