SSC13-VIII-3

Small Spacecraft Software Modeling: A Petri Net-Based Approach

Levi Malott, Pasha Palangpour
Missouri University of Science and Technology
400 West 13th Street, Rolla, MO 65409-0050; 573-341-7280
lmnn3@mst.edu

Faculty Advisor: Hank Pernicka
Missouri University of Science and Technology

ABSTRACT

Software design and development often presents a high-risk element during the execution of engineering projects due to devaluation of possible conflicts or identifying defects during late stages of development. Many defects identified during the late stages of small spacecraft development can be avoided by constructing interactive, dynamic models. This process is often followed for hardware fabrication/test, but often not to the same extent for software. An alternative process to the typical software development is needed that enables modeling and simulation feedback at early design stages. Petri nets allow for software visualization, simulation and verification in a cost-effective way. An alternative software modeling approach using Petri nets is presented to rapidly design, develop and verify/validate small spacecraft software. Using the presented techniques, the Missouri University of Science and Technology Satellite Research Team successfully demonstrated core functionality of their software system at the Final Concept Review (FCR) of AFRL's University Nanosat Program's Nanosat-7 Competition.

INTRODUCTION

The key attributes differentiating hardware and software development include identifying defects, visualizing software achitecture, and testing. Identifying software defects usually occurs during the implementation phases of development causing large budget and schedule overruns. Defective hardware designs are prevented and identified during design phases by evaluating compatibility of integrated components and the designer's comprehension of the implications of connecting components. Visualizing software architecture is normally realized by creating functional block, timing, class, and numerous other diagrams. This process attempts to capture the structure and interactions of a dynamic system through static methods, analogous to analyzing a three-dimensional structure CAD model via two-dimensional printouts. Additionally, understanding the meaning of these diagrams requires previous software modeling experience that is not typically found in those individuals with technical management positions. Software testing is approached by attempting to identify and test all failure cases, subsequently rectifying found defects. Hardware testing involves the creation of virtual model simulations that are used to identify design flaws and subsequently improve designs prior to fabrication and physical tests. Similar models should be incorporated in software development to identify and remove defects, provide useful graphical realizations, and encourage interactive design.

Model-driven software engineering methods in spacecraft applications have recently emerged to solve these problems. Using this methodology, software models are built and subsequently autocoded or act as a blueprint for manual implementation. Most notably, the SPIN model [1], Simulink [2], and UML Statecharts [3] are used in industry to create model diagrams and perform model-checking. SPIN models are specified using Promela [4], which natively lacks basic constructs like floating-point data types. Simulink models are nonformal, synchronous representations that cannot simulate the nondeterminism inherent in concurrent systems. Statecharts create formal models, but place restrictions on the designs and cannot fully represent real software concurrency. Additional tools have been developed to augment the downsides of the aforementioned tools, but involve converting models to different specifications, are tool specific, and requires management of more artifacts. In this paper, a new method for modeling small spacecraft software using the expressive power of Petri nets to accurately design software is presented. The diagrams produced during this process provide explicit definitions of software interfaces and interactions, while the formal mathematic definition verifies software correctness.

Motivation

The Missouri University of Science and Technology Satellite Research Team (M-SAT) has participated in four of the eight UNP Nanosat competitions, most recently placing second in Nanosat-7. The spacecraft mission proposed and developed consisted of one microsatellite and one smaller microsatellite in a mated configuration, as shown in Figure 1. The larger satellite, MR SAT, possesses two visual imaging cameras that provide the capability to perform stereoscopic imaging. In orbit, the mated spacecrafts separate with the intention of MR SAT tracking the smaller satellite, called MRS SAT. MR SAT uses stereoscopic imaging to determine the relative position vector to MRS SAT, and nominally maintains a ten meter separation.

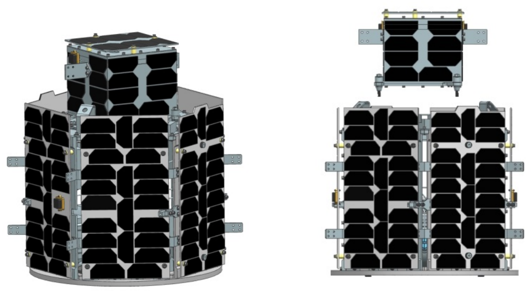


Figure 1: Mated (left) and Deployed (right) Configurations

The development of MR SAT and MRS SAT has required the involvement of many students from diverse academic backgrounds. Introducing concepts such as software architecture and development can prove to be a nontrivial task, as students tend to avoid areas with which they are not familiar. Along with the high-turnover rate of student researchers, various aspects of the spacecraft development progress more quickly than others. This also makes effective, interdisciplinary communication both stressful and critically important. Specifically regarding software, diagrams are much easier to understand than reading source code, while animations have proven to be highly effective communication and explanatory tools for transitioning members into software development roles.

The aim of the authors of this article is to not only present a decompositional method of software system modeling for small spacecraft, but also to encourage small satellite developers to consider adopting descriptive software architecture visualization. Advanced graphics and animations are instructive and foster interdisciplinary communication, which begins with the ability to present the material. With the high-turnover rate of students intimatly tied with university research teams, quickly educating younger team members is imperative for success. It is hoped that this paper will encourage other small spacecraft development teams to consider the use of Petri net-based software models as an integral role of development.

SOFTWARE MODELING

Decomposing the high-level descriptions of small spacecraft software functionality should start during the conceptual and requirement analysis phases. The Concept of Operations (CONOPS) [5] defines the *Modes of Operation* of the system, often executed in a sequential order. Each mode is further divided into ordered or cyclic execution of specific tasks. The structural and temporal organization of these tasks guarantee objectives are achieved. The organizational constraints of tasks are typically stated during the design phase where timing and performance requirements are defined. This leads to a clear separation of system responsibilities and understanding of modeled software. Petri nets provide an intuitive, graphical representation that enables modeling of complex systems. Presented below is a simple, but effective, modeling architecture that was used to create the core functionality of the Command and Data Handling (C&DH) software implemented in the M-SAT research team's spacecraft.

PETRI NETS

Petri nets (PN) are directed state-transition graphs based on traditional automata theory. Petri nets are often used to verify system properties, simulate concurrent systems, and analyze temporal attributes of systems. Many extensions have been created to enhance the expressive modeling capabilities for use in many engineering problems. Colored Petri nets (CPN) are extensions of Petri nets that allow the assignment of data types to nodes of the modeled system. CPNs are referred to throughout this document as the ability to assign data types allows software systems to be realized more effectively.

Two types of nodes exist: places and transitions, represented by an oval and rectangle respectively. Places denote the various software states, while transitions determine a change of states. Places and transitions are connected through arcs (edges), which represent data transfer. Places can only connect to transitions and are the *input places* of those transitions. Transitions connect to places and are the *output places* of those transitions. Places can contain tokens that represent data in the system. Tokens belong to *color sets* that associate data types to tokens. Places can only contain tokens of a specific type, denoted by a label located to the bottom left of a place, shown in Figure 2 on the following page. The tokens in a CPN can represent any data type one would use in implementation, whether it be a Boolean value or an object structure. The combination of tokens distributed over the net's places represent a configuration of the software, called a *marking*. The *initial marking* is defined by the system modelers and represents configuring the system to a known state. Transitions optionally contain

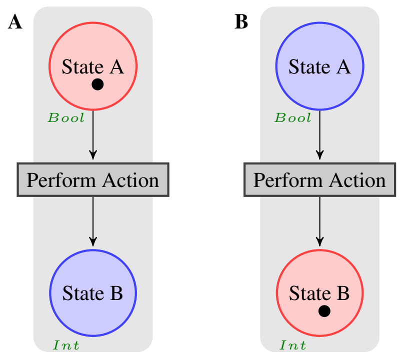


Figure 2: Simple CPN Transition Firing

guards, in the form of predicates, used to select tokens from input places. If a transition's guard evaluates to *true*, it is said to be *enabled* and can *fire*. When a transition is enabled, token(s) are consumed from its input place(s), and tokens are deposited to its output place(s). The transitions also contain logic so that they can alter consumed tokens, or simply create new tokens for their output place(s). A simple Petri net is shown in Figure 2 to illustrate the behavior of a Boolean token in *State A* enabling a transition. The ***Perform Action*** transition fires and outputs a token to its output place, *State B*.

Formalism and Benefits

The formal definition of Petri nets is shown in Definition 1 [6], which relates the previous description to the mathematical background. Corresponding graphical representations are presented on the right of the definition. The formal definition provides the ability to simulate and validate CPNs. More importantly, the developers do not need a strong understanding in the formal definition, as the CPN editors will perform syntactical and semantic checking during simulation/verification. This leaves the role of the developers to graphically lay out the structural components, create declarations, and define net inscriptions. The functional aspects of software modeling are the developer's responsibility, while the nonfunctional operations are managed by the editing tools.

A formal proof is required to demonstrate that the specification of the system is 100% correct. Given a CPN with an initial marking, every possible software state is enumerated into a *state space graph* or *reachability graph*. From an initial marking, the reachability graph contains every possible sequence of states in which the system can reside. The verification of the system comes from the reachability graph's properties. Such properties include minimum/maximum amount of tokens on a place, termination states, timing of transition firings, and whether the system always terminates in the same place or not. More complex properties, such as one state preceding another, can be verified through model checking with temporal logic [7].

Together, the graphical and mathematical representation of Petri nets assist in interdisciplinary understanding. The graphical representation provides a clear structural view of the designed system. Viewers can easily see the connections and interactions between various objects and modules. By coupling this with the mathematical definition, the dynamic interaction and state evolution can be observed. With CPN tools [8], the simulation function animates transitions firing and movement of tokens throughout the system. Conflicts in the design are recognizable; if all transitions cease firing, then the system is in a halted state. Showing the consequences of requirement or design defects in this manner is a powerful persuasion tool for immediate corrective actions.

It should also be noted that CPNs cannot formally verify systems with large number of states, due to state-space explosion. Various techniques must be incorporated to ensure an equivalent state space can be calculated. An intuitive method is to verify different sections of the model and then merge each verified section into a single place or transitions. The representative places/transitions act as a black box in place of the full subnet. Furthermore, the nature of small spacecraft software inherently reduces occurrences of state space explosion. Smaller spacecraft lead to fewer communicating modules, leading to fewer states that reduces the risk of state-space explosion. This concept has motivated using the expressive power of Petri nets to model and verify the correctness of complex small spacecraft software.

Modeling Tools

Many solutions exist for modeling Petri nets, each having their benefits and downsides. CPN Tools is widely used as it provides graphical modeling, hierarchical structures, performance-based analysis, simulation and state space verification methods [8]. Models can be quickly constructed using the easy-to-use graphical interface and subsequently simulated. Currently, no packages exist to convert CPN Tools models to procedural languages, such as C or C++, but still have the ability to act as a blueprint for software construction. SNAKES [9] is a Python library that allows rapid Petri net prototype construction, simulation, state space veri-

Definition 1. A **Colored Petri Net** is a nine-tuple, $CPN = (P, T, A, \Sigma, V, C, G, E, I)$, where:

(1) P is a finite set of **places**.

(2) T is a finite set of **transitions** T such that $P \cap T = \emptyset$.

(3) $A \subset P \times T \cup T \times P$ is a set of directed **arcs**.

(4) Σ is a finite set of non-empty **color sets**.

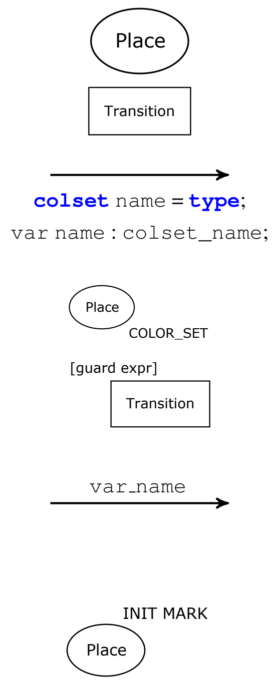
(5) V is a finite set of **typed variables** such that $Type[v] \in \Sigma \; \forall$ variables $v \in V$.

(6) $C : P \to \Sigma$ is a **color set function** that assigns a color set to each place.

(7) $G : T \to EXPR_V$ is a **guard function** that assigns a guard to each transition t such that $Type[G(t)] = Bool$.

(8) $E : A \to EXPR_V$ is an **arc expression function** that assigns an arc expression to each arc a such that $Type[E(a)] = C(p)$, where p is the place connected to arc a.

(9) $I : P \to EXPR_\emptyset$ is an **initialization function** that assigns an initialization expression to each place p such that $Type[I(p)] = C(p)$.



fication methods, visualization through GraphViz [10], user-defined extensions and import/export capabilities using the standard Petri Net Markup Language (PNML) [11].

Recently, the power of PNML-formatted Petri nets has been extended through the SNAKES-based Neco high-level Petri net compiler. Neco provides the capability to compile models into libraries callable from any C-compatible language [12]. The process of compiling software models directly to code is a move toward hardware-like development, where schematics or CAD models are used to create components. Removing the implementation step reduces the risk of developers' misinterpretation of design models or defects introduced through typographical mistakes.

System Description

Figure 3 illustrates an example spacecraft consisting of physical subsystems communicating via a data bus. The C&DH subsystem can be viewed as the master of the system: delegating tasks and responsible for high-level actions in the system. Each subsystem has their own microprocessor and manages a small number of peripherals. The main function of these subsystems is to interpret C&DH commands to interact with peripherals and respond with data. The methods presented will pertain to modeling of the software system developed for a C&DH system. The remaining subsystems' software are left as representative transitions in the overall CPN. Table 1 defines the various color sets used throughout this article. Table 2 outlines the functions used in the examples presented. For brevity, only the color sets and functions used in examples are presented.

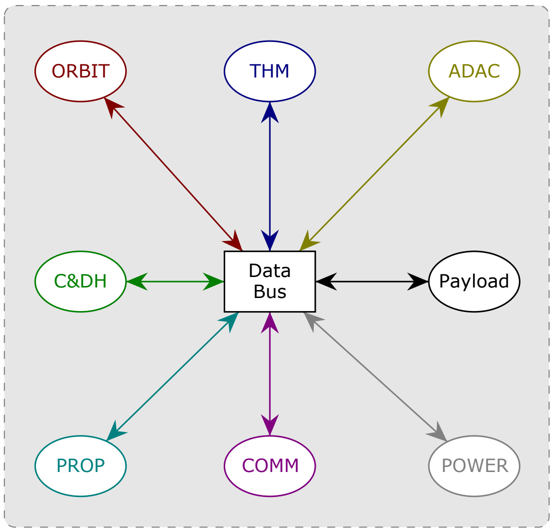


Figure 3: Example High-Level Computational Structure

Table 1: Model Color Sets

Name	Definition	Description
BOOL	`bool`	Boolean data type that can be either *true* or *false*.
BYTE	`int with 0..255`	Defines a single byte data type.
JOB_ID	`with detumble | mission_mode_one | ↩ mission_mode_n;`	Enumeration of unique Job IDs defined by the model.
Task	`record jobId:JOB_IDS * taskId:↩ TASK_ID * priority:INT;`	A structure containing the Task's owner Job ID, the Task's ID, and priority.
TASK_ID	`with task_a | task_b;`	Enumeration of unique Task IDs defined by the model.
Task	`record jobId:JOB_IDS * taskId:↩ TASK_ID * priority:INT;`	A structure containing the Task's owner Job ID, the Task's ID, and priority.
BusCmd	`record owner:Task * addr:BYTE * data:↩ BYTE;`	Structure used for sending data to subsystems. Identifies the owner Task, recipient address and data to transfer.
BusResp	`record owner:Task * from:BYTE * data:↩ BYTE;`	Structure used for receiving data to subsystems. Identifies the owner Task, sender address and data received

Table 2: Model Functions

Name	Definition	Description

bus_guard	`fun bus_guard( task:Task, granted_task:Task ) =`
	`  if (#jobID granted_task)=(#jobID task_rec) ↩`
	`      andalso`
	`    (#taskID granted_task)=(#taskID task_rec)`
	`  then`
	`    true`
	`  else`
	`    false`
		Ensures *task* and *granted_task* are equivalent by comparing their owner Job IDs and Task IDs.

bus_resp_guard	`fun bus_resp_guard( bus_cmd:BusCmd, bus_resp:↩`
	`     BusResp ) =`
	`  if (#owner bus_cmd)=(#owner bus_resp) andalso`
	`    (#addr bus_cmd)=(#from bus_cmd)`
	`  then`
	`    true`
	`  else`
	`    false`
		Evaluates to *true* if and only if the bus_cmd and bus_resp owner Tasks are equivalent and recipient address matches the received address.

priority_sort	`fun priority_sort(t1:Task, t2:Task) =`
	`    (#priority t1) < (#priority t2);`
		Used as a function handle for the *sort* function in CPN Tools. *priority_sort* is used to determine how Task precedence should determined.

METHODS

Mission Lifecycle Representation

Modeling the high level software states coincides with construction of the CONOPS. Conceptualizing a spacecraft mission usually starts as a series of bullet points in a notepad. After deliberations and iterations, the first level bullet points represent mission modes. The order of the mission modes may not be defined yet, but can be represented as a place named respectively. Adding transitions between the places identifies the precedence of mission modes. Figure 4 displays how the sequence of operation modes are connected in CPN Tools.

The double-outline rectangles are known as *substitution transitions*, which are fundamental hierarchical blocks or abstractions. Essentially, every substitution transition is represented by another Petri subnet, that is separate from the current net, as shown in Figure 5. Subnets defined as substitution transitions have a single entry point, called the *input socket*, where the tokens from connected places are deposited. Similarly, they contain a place denoted as the *output socket*, which acts as the single point of exit from that subnet. Internally, the substitution transitions represent another CPN that contains the operations performed. An important characteristic of these types of transitions is that they are *instances*. They can be reused in other parts of the CPN and independently behave in the exact same manner.

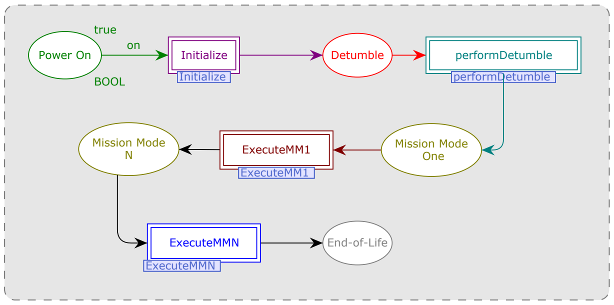


Figure 4: C&DH Software Lifecycle Petri Net

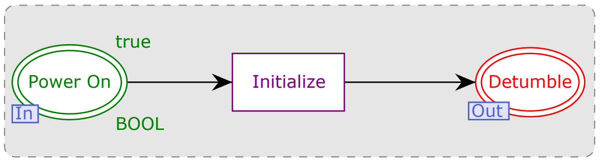


Figure 5: Default *Initialize* Substitution Transition

Using CPN Tools to simulate the diagram in Figure 4 would start with a single *Boolean*-type token located in the **Power On** place. This is identified by the *true* label located above **Power On**; this is the initial marking. The place-type of **Power On** must match the type of the initial marking, otherwise an error occurs and a message is displayed. ***Initialize*** is immediately enabled and fires, placing a token on the *input socket* place of the ***Initialize*** subnet. By default, the subpage representing the substitution transition is simply the input and output socket connected by a transition. Simulation continues by firing enabled transitions in ***Initialize***. Once a token is placed onto the *output socket* place of ***Initialize***, the token is placed on **Detumble**. Simulation continues following the above process until there are no enabled transitions, which is when ***ExecuteMMN*** finishes and a token is located on **End-of-Life**.

Mission Mode Operations as Jobs

The substitution transitions connecting mission modes are defined as *Jobs*. Jobs represent a complex organization of computations composed from smaller, indivisible units. The organization of these units make the Job unique and define its operational behavior. The basic building-block units are defined as Tasks. Tasks are analagous to integrated circuits; they are used in a system design to achieve a specific purpose with clearly defined inputs and outputs. Knowing the internal structure of the Task is not necessary for its use in a system, as long as there exists a clear interface and an expected operation. Figure 6 depicts a simple Job subnet where two Tasks are sequentially executed. More intricate schemes can be represented, like requesting multiple Task execution simultaneously, by adding additional Task places on the Job net connected to ***Input Actions*** and ***Output Action***.

Functional Requirements as Tasks

The primary design principle behind a Task is that there should only exist one functional C&DH requirement traced to a Task. This significantly reduces the complexity of a given Task and increases Task reuse. Tasks should be considered as the fundamentals units, where some configuration of them creates a composite structure. When verifying composite structures, the process can be broken by verifying each constituent. Upon a constituent unit passing, a representative structure replaces it in the composed CPN. Once all units are verified, then the overall CPN with representative unit nets can be verified through state space analysis. This is a powerful optimization technique for verifying a system represented by a large number of states.

Jobs control when Tasks should execute, based on system parameters and the global system state. To initiate this sequence, Jobs place token(s) on **Start Task** enabling ***Add Task Information*** where supplementary information, such as resources needed, is added. Figure 7 displays a generic Task structure used as a modeling

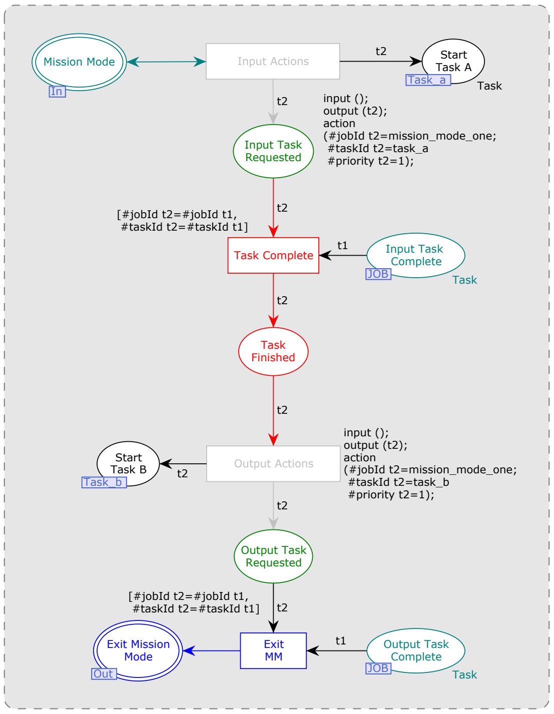

Mission Mode
In
Input Actions
t2
Start Task A
Task_a
Task
input (); output (t2); action (#jobId t2=mission_mode_one; #taskId t2=task_a #priority t2=1);
Input Task Requested
[#jobId t2=#jobId t1, #taskId t2=#taskId t1]
Task Complete
t1
Input Task Complete
JOB
Task Finished
Start Task B
Task_b
Output Actions
input (); output (t2); action (#jobId t2=mission_mode_one; #taskId t2=task_b #priority t2=1);
Output Task Requested
Exit Mission Mode
Out
Exit MM
Output Task Complete

Figure 6: Basic Job Subnet

template. Every Task must contain the Microkernel interface as that module controls Task execution and resource management. The colored boxes on the left

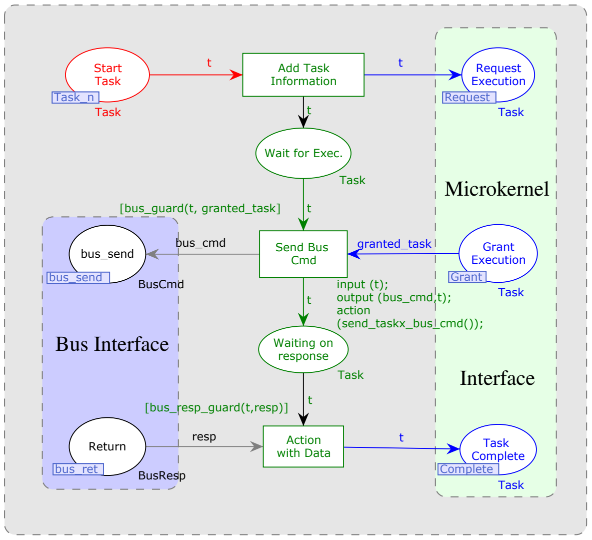


Figure 7: Generic Task Structure

and right sides represent interfaces to the data bus and Microkernel, respectively. Interface places belong to a *fusion set*, denoted by the purple *fusion set tag* labels. The name on the *fusion set tag* indicates which *fusion set* a place belongs to. Fundamentally, *fusion sets* contain any number of places that act as a single state. This means that the marking of a place is shared between all places in a *fusion set*. Conceptually, this capability can be thought of as any global entity whose state must be known throughout the system. Commonly this includes data buffers, singleton modules, module interfaces, and data buses throughout the system. To clarify, singleton modules are any module restricted to one instance in the system. A Microkernel is referred to as a singleton, because only one Microkernel is allowed to schedule and execute Tasks. Note that Tasks are not required to use the communication bus, but any global module or resource interface can exist here. The only requirement is that if a Task is using a global module or resource, it must explicitly request those resources from the Microkernel.

Once a Task is allowed to execute, `granted_task` contains the information of the Task that is able to execute. The Microkernel passes this information through **Granted Execution**. If `t` matches `granted_task`, then ***Send Bus Cmd*** is enabled and fires. The bracket-enclosed statement above ***Send Bus Cmd*** is the *guard* that enforces this rule. Comparing the tokens is necessary since all Tasks connected to the Microkernel will receive the token, but only a specific Task should execute. Once ***Send Bus Cmd*** fires, a `bus_cmd` token is placed on **bus_send**, which enables the first transition within the Bus Handler, shown in Figure 8.

Within the Bus Handler, ***bus_send*** passes the `bus_cmd`

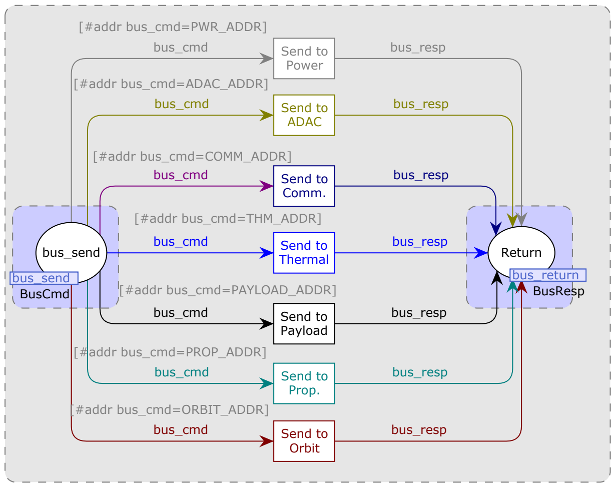


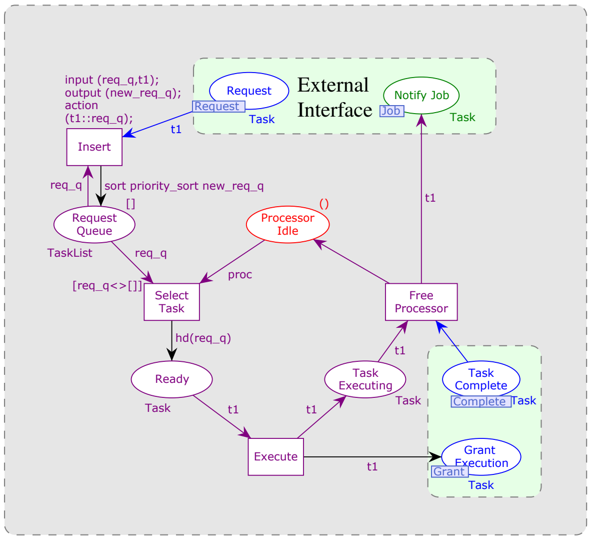
Figure 8: Generic Bus Handler

token to all connected subsystems. For illustrative purposes, representative transitions are connected to **bus_send** that can be replaced by substitution transitions for a complete model. In this example, there is an assumption that the bus communication protocol requires connected components to be uniquely addressed followed by the data. Each subsystem corresponding transition contains a [#addr bus_cmd=SUBSYSTEM_ADDR↩] guard that evaluates whether the destination of bus_cmd is equivalent to a pre-defined constant. If it does not match, then the subsystem does not respond to the sent message. For example, if a token addressed to the Power subsystem, only ***Send to Power*** would be enabled. Upon the firing of ***Send to*** . . . transitions, the bus_resp token is placed on **Return** and enables the corresponding Task's transition ***Action with Data***. Here the Task is allowed to modify the returned data. A common action here is to store the received data in a global buffer, which requires the Task to have the buffer in its resource list. Once a Task has completed modifying and storing data, t is placed on **Task Complete**. This notifies the Microkernel that the Task has completed and can allow another Task to execute. The Task token t is passed to the Microkernel so it can subsequently notify the parent Job that one of its Tasks has completed.

Global Resources

Modeling global resources is very similar to the above techniques and is necessary for accurate software representations. Figure 9 shows a very simple Microkernel that inserts Tasks into a priority queue and executes the head of queue. Examples of preemptive, resource-managing Microkernels can be found in [13]. **Request Queue** maintains a prioritized queue of Tasks to execute and is initialized to an empty list. When a Task t1 is placed on **Request**, the current queue req_q and t1 are provided as inputs to the ***Insert*** transition function. t is appended to the front of req_q, then sorted based on priority. The result of these actions are stored into a new request queue, new_req_q, and placed on **Request Queue**. Once a Task has been placed into the request queue, ***Select Task*** fires and the Task executes. The guard, [req_q<>[]], of ***Select Task*** compares the req_q to an empty list and evaluates to *true* if the req_q is not empty. The second component that enables ***Select Task*** is an available token on **Processor Idle**. A single UNIT token is initially placed on **Processor Idle**, denoting that there is a single processor available in the system. Additional initial tokens may be placed here to indicate multi-core systems. ***Select Task*** will stay enabled as long as there are Tasks in req_q and processors available. If the system is modeled this way, simulation continues by the nondeterministic firing of enabled transitions. There are ways to enforce transition firing precedence, but is out of the scope of this article.

The head of req_q is chosen to execute, which subsequently produces a token on **Grant Execution**. The selected Task is notified by the use of the *Grant* fusion set and the token t1 is renamed as granted_task inside of the Task subnet. The model simulation pro-

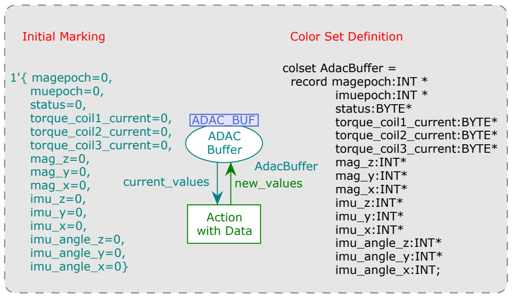


```
input (req_q,t1);
output (new_req_q);
action
(t1::req_q);
```

Figure 9: Simplified Priority-based, Non-Preemptive Microkernel

gresses along the steps described with Figure 7. Care should be taken when naming variables located on arcs connected to fusion sets. If the fusion sets are used as module interfaces, the variables used to transfer data among them are in the scope of all connected modules. Renaming variables on one side of the interface, as in Figure 7, ensures that variables do not bind to new tokens and corrupt the needed data.

Global data buffers are represented as a single place within a net. For an accurate realization, a buffer needs a name, unique type, and initial marking. The name uniquely identifies the buffer, the type describes the data fields, and the initial marking initializes the buffer to a known starting state. Figure 10 shows an example global data buffer connected to the ***Action with Data*** transition in Figure 7. The Attitude Determination and Control (ADAC) subsystem global data buffer is represented as the place **ADAC Buffer**.



```
1'{ magepoch=0,
    muepoch=0,
    status=0,
    torque_coil1_current=0,
    torque_coil2_current=0,
    torque_coil3_current=0,
    mag_z=0,
    mag_y=0,
    mag_x=0,
    imu_z=0,
    imu_y=0,
    imu_x=0,
    imu_angle_z=0,
    imu_angle_y=0,
    imu_angle_x=0}
```

```
colset AdacBuffer =
  record magepoch:INT *
         imuepoch:INT *
         status:BYTE*
         torque_coil1_current:BYTE*
         torque_coil2_current:BYTE*
         torque_coil3_current:BYTE*
         mag_z:INT*
         mag_y:INT*
         mag_x:INT*
         imu_z:INT*
         imu_y:INT*
         imu_x:INT*
         imu_angle_z:INT*
         imu_angle_y:INT*
         imu_angle_x:INT;
```

Figure 10: Global ADAC Buffer

The ADAC subsystem consists of a microcontroller connected to an inertial measurement unit (IMU), threeaxis magnetometer, and three separate torque coils. Update requests with the ADAC subsystem provides measurement epochs, a status register, all three torque coil currents, magnetometer-measured magnetic field vector, IMU-measured magnetic field vector, and IMU-provided angle rates. Each field is represented as an entry in a record color set, shown on the right side of Figure 10. Attaching **ADAC Buffer** to ***Action with Data*** requires that token must be initially placed on **ADAC Buffer**. If a designer forgets to place an initial marking on a data buffer, CPN Tools would show a dead marking when a Task places a token on the place connected to ***Action with Data***. To enable a transition, all input places must have a necessary amount of tokens. Because **ADAC Buffer** is an input place to ***Action with Data***, it must always contain a token. When ***Action with Data*** is enabled, `current_values` contains the values of the most recently deposited token. In the case of the first firing, `current_values` would have every record field set to zero; as determined by the initial marking. ***Action with Data*** can use `current_values` in calculations and subsequently saves the new values by binding values to `new_values`. The token bound to `new_values` is deposited into **ADAC Buffer**, which saves the values for future use.

FORMAL VERIFICATION

Conducting a formal analysis of the modeled system is the integral part of using CPNs as a software modeling solution. CPN Tools contains an option for calculating the state space of a CPN and generating a report detailing statistics, liveness, boundness, and fairness properties. Table 3 shows the *Statistic* and *Liveness* property sections of a generated report from the modeled C&DH subsystem shown at Nanosat-7 FCR. The dead marking of state 60850 required further investigation to determine the details of its occurrence. Expanding the state space revealed that this marking represented the state where **End-of-Life** contained a single token. This was the desired outcome of the modeling process as there should be no enabled transitions at **End-of-Life** and denotes the end of the mission.

Table 3: State Space Analysis Results

```
-------------------------
 Statistics
-------------------------

  State Space
     Nodes:  60850
     Arcs:   197955
     Secs:   111
     Status: Full

  Scc Graph
     Nodes:  60850
     Arcs:   197955
     Secs:   3

-------------------------
 Liveness Properties
-------------------------

  Dead Markings
     [60850]

-------------------------
```

The number of possible C&DH software states totaled $60,850$, which further demonstrates the feasibility of implementing the proposed modeling procedures. CPN Tools is capable of calculating state spaces with hundreds of thousands of nodes; it is only limited by the memory of the computer. If the number of nodes in the state space graph become increasingly large (state space explosion), CPN Tools will only calculate partial state spaces. Repeating the process will obtain more nodes in the state space graph. Formally verifying system correctness at this point is not feasible, but useful characteristics can still be derived. Once a (partial) state space is calculated, users can make queries to answer specific questions about system functionality. Some properties that can be identified from queries include minimum/maximum tokens in a state, recurring cycles, reachable markings, report dead markings, and fairness properties [6].

CONTRIBUTIONS

The presented process decomposes complex software systems into manageable portions which facilitate software designers' ability to effectively conceptualize and diagram concurrent, distributed systems. The article provides readers with the details and benefits of incorporating Petri net software modeling to their respective systems. Additionally, it was suggested that readers could develop more intricate systems through the use of abstractions and modeling other software systems within a spacecraft. Current software modeling techniques often either lack abstraction or lack formal mathematical definitions. Proprietary or application specific tools have been constructed to create a beneficial synergy between various tools. Petri nets integrate the usefulness of abstractions and formal mathematics, which is apparent in CPN Tools.

The simplified examples provided were taken from the actual model of the C&DH subsystem of MR SAT. From these examples, small spacecraft missions are provided a framework for software modeling, formal verification, and constructing instructional, dynamic software architecture visualizations. These attributes benefit the software development process by removing ambiguity through explicit declarations of nets, instantaneous feedback through simulations, understandable diagrams, partitioned system models/diagrams, and assurance through formal verification. The only tool involved for completing modeling was CPN Tools, which is an open-source, freely-available tool. This study demonstrated the applicability of CPN Tools to accomplish tasks of equivalent commercial software, but without the high costs involved when using commercial software.

ADVANTAGES

Abstraction Composing larger CPNs from smaller subnets allows for diagram segmentation, subnet reuse, definition of global resources, explicitly defined external interfaces, and plug-and-play testing of modeled software components. For example, to see the effects of a different scheduling scheme, a different Microkernel with the same external interface can replace an existing one by modifying fusion sets.

Visualization The graphical differences between places and transitions allow viewers to easily see how the software states change. Structural dependencies and interactions are clearly defined through arcs that define the flow of data. The use of subpages allow large diagrams to be separated into equivalent subnets, that retain the integrity of the model, in an intuitive way.

Simulation The ability to simulate a models execution allows for instant feedback to designers that removes ambiguity. Proposed design changes are modeled and affected functionality is visually shown.

Formal Verification Formally verifying software systems increases the confidence of software design. Small spacecraft software naturally has fewer possible states than larger spacecraft which reduces the risk of state space explosion. Without the restrictions of similar software modeling methods, CPNs allow designers to creatively model software without worrying about work-arounds during model development.

Scalable While larger spacecraft software models will need to make use of abstractions, CPNs are still applicable. Subnets can be individually verified, then represented by simple places or transitions in the overall CPN. The overall CPN is then formally verified with the abstracted subnets. This significantly reduces the number of states in the state space, while maintaining equivalent formal proof results.

Adaptable Changing requirements is an inevitable reality that all spacecraft projects face. Formal models are quicker and more cost effective to change than implemented source files.

Nondeterminism When a CPN has multiple transitions enabled as a given simulation step, they are fired in a nondeterministic order. This effectively and accurately represents true concurrent, multi-cored systems.

CONCLUSION

Modeling small spacecraft software using Petri nets has been shown to be a viable method, as small spacecraft inherently reduce the occurence of state space explosion. Petri nets allow advanced abstractions, diagram partitioning, formal verification, simulation, intuitive representations of software. A simple method for decomposing small spacecraft missions into manageable CPNs was presented. Addtionally, it was shown how the modeled architecture extends to incorporate software executing on distributed hardware subsystems. The same method was used to model and subsequently implement the core functionality of the C&DH software system onboard MR SAT. The process was met with success and demonstrated at Nanosat-7 FCR. The procedure can be summarized as the following sequence:

1. Create the CPN to represent the software lifecycle
   - 1.1 Populate with mission modes realized as places
   - 1.2 Determine mission mode ordering using transitions
2. Convert lifecycle transitions into substitution transitions, called Jobs
3. Determine all necessary system functionality
   - 3.1 Create a Task or global resource for each
   - 3.2 Using fusion sets, create external interfaces for all components
4. Structurally arrange Tasks within Jobs to achieve complex objectives

By incorporating simulation and verification along the way, errors are surfaced by feedback through CPN Tools. Defective designs result in a failure to simulate or deadlocked states during verification. Modifying the model as defects are encountered reduces the risk of errors propagating into the implementation phase, thus reducing the risk of budget and scheduling overruns.

ACKNOWLEDGEMENTS

The authors would like to thank their faculty advisor, Dr. Hank Pernicka, for the encouragement to pursue this research and advice throughout. The M-SAT team has been an integral part of completing and implementing this project. M-SAT has provided an unmatched educational experience not often found at the university level.

REFERENCES

[1] Klaus Havelund, Mike Lowry, and John Penix. Formal Analysis of a Spacecraft Controller using SPIN. *Software Engineering, IEEE Transactions on*, 27(8):749–765, 2001.

[2] K. Gundy-Burlet. V&V for Model-Based Software Development. Presented at the annual Workshop on Spacecraft Flight Software hosted by the Jet Propulsion Laboratory, 2009.

[3] E. Benowitz. UML Statechart Autocoding for the Mars Science Lab (MSL) Mission. Presented at the annual Workshop on Spacecraft Flight Software hosted by the Jet Propulsion Laboratory, 2012.

[4] R. Iosif. The Promela Language. Available: `http://www.dai-arc.polito.it/dai-arc/manual/tools/jcat/main/node168.html`, April 1998. [Online; accessed 04-April-2013].

[5] IEEE Guide for Information Technology - System Definition - Concept of Operations (ConOps) Document. *IEEE Std 1362-1998*, page i, 1998.

[6] Kurt Jensen and L.M. Kristensen. *Coloured Petri Nets: Modelling and Validation of Concurrent Systesms*. Springer Berlin Heidelberg, 2009.

[7] E. Allen Emerson. Temporal and Modal Logic. In *HANDBOOK OF THEORETICAL COMPUTER SCIENCE*, pages 995–1072. Elsevier, 1995.

[8] Wells L. Lassen H.M. Laursen M. Qvortrup J.F. Stissing M.S. Westergaard M. Christensen S. Ratzer, A.V and K. Jensen. Cpn tools for editing, simulating, and analysing coloured petri nets. In *Proceedings of the 24th international conference on Applications and theory of Petri nets*, ICATPN'03, pages 450–462, Berlin, Heidelberg, 2003. Springer-Verlag.

[9] F. Pommereau. Quickly Prototyping Petri Nets Tools with SNAKES. In *Proceedings of the 1st international conference on Simulation tools and techniques for communications, networks and systems & workshops*, Simutools '08, pages 17:1–17:10, ICST, Brussels, Belgium, Belgium, 2008. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[10] J. Ellson, E.R. Gansner, E. Koutsofios, S.C. North, and G. Woodhull. Graphviz and Dynagraph – Static and Dynamic Graph Drawing Tools. In M. Junger and P. Mutzel, editors, *Graph Drawing Software*, pages 127–148. Springer-Verlag, 2003.

[11] Michael Weber and Ekkart Kindler. The Petri Net Markup Language. *Petri Net Technology for Communication-Based Systems: Advances in Petri Nets*, 2472:124, 2003.

[12] L. Fronc and F. Pommereau. Optimising the Compilation of Petri Net Models. In *Proc. of the second International Workshop on Scalable and Usable Model Checking for Petri Net and other models of Concurrency (SUMO 2011)*, volume 626, 2011.

[13] M. Naedele. Petri Net Models for Single Processor Real-Time Scheduling. TIK-Report No. 61, November 1998.