A Model-Based Toolset for Supporting Rapid Integration and Verification of Spacecraft Electronics

Anand S Madhusoodanan, Brandon Eames
Department of Electrical and Computer Engineering
Utah State University
4120 Old Main Hill, Logan, UT 84322-4120; 435-797-2841
anandsmadhu@gmail.com, beames@engineering.usu.edu

ABSTRACT: Recent trends in small spacecraft design seek to leverage the principles of modularity and component-level decoupling to facilitate rapid system integration. In this paper, we examine a tool-based approach to support the design and verification of rapidly integrated small spacecraft systems. Most existing approaches to rapid systems integration support device interfacing and integration at runtime, at the expense of added system hardware and software to perform configuration management. Often, rapidly integrated systems need system configuration facilities only once, when all devices are "plugged in." Once all the devices have been discovered and integrated, the logic that performs these activities is no longer needed. In addition to the overhead imposed by configuration management, self-verification is not permitted by a completely dynamic system, since there is no capability of determining dynamically what the "correct" configuration should be. We propose an approach based on the use of domain specific visual modeling to represent the system electronics, and automatic program synthesis to generate both the middleware to glue the system together, as well as a software-based self-test to validate the rapidly integrated system.

INTRODUCTION

Ongoing research efforts at Utah State University in the area of rapid systems manufacturing seek to develop modular spacecraft panels with physically embedded electronics [1]. To support rapid systems integration, the panels support "plug-and-play" of devices, based on the Universal Serial Bus protocol. The approach to plug-and-play is similar in concept to an ongoing research effort in the AFRL-based Responsive Space Initiative[2]. The goal of the manufacturing research is a modular spacecraft panel which can be used off-the-shelf for rapid system prototyping and integration.

The rapid integration and development of a system for a satellite requires the ability to interface to a variety of devices as well as to verify that the system functions correctly.

Modularity forms an important component of rapidly developed systems. The presence of a certain degree of modularity eliminates the need to manufacture components from scratch for every system that is assembled. All non-trivial modular designs tend to be hierarchical; due to this hierarchy, the presence of modularity is seen at several levels of the design. The advantage with using modular components for spacecraft design is separation of concerns. Component developers can separately design and implement the components, while system integrators must design component interfacing infrastructure. From a spacecraft systems designer's perspective, ideally, the internal details of how these components are built or fabricated are not as important as the relation and interaction between the components.

Another advantage that the rapid integration and assembling of components gives can be seen in behavioral analysis. The system behavior need not necessarily be studied at the lowest level of composition. In the case of a rapidly assembled spacecraft system, intelligent panels provide a platform to integrate pre-developed and pre-verified devices. Software executing on the panel's embedded microprocessor contains the logic to recognize and enumerate devices integrated or "plugged in" at system startup. Once devices have been integrated, application software executing on the panel implements the system level spacecraft functions, using the integrated devices.

The design approaches to software-based system recognition and integration can be distinguished, based on what knowledge the software has of the set of devices to be integrated at runtime. One approach[2] suggests the use of software-based middleware that is completely responsible for enumeration and interfacing of devices, with no knowledge of what devices to expect at runtime. In this paper, we examine an alternative approach based on rapid system configuration generation. Our approach utilizes a plug-

and-play bus, coupled with software-based middleware to manage device enumeration. We have developed a tool which generates software to manage the configuration and integration of an expected set of devices. An advantage of our approach is the ability to test at system startup, whether the expected set of devices is in fact plugged into the system, and whether those devices are functioning properly. Our approach, while not "glueless," has the advantage of being able to incorporate the capabilities of system verification in the form of an on-board self-test. The disadvantage of the presence of glue in the solution can be negated to a large extent if there is a way to automate the process of the generation of glue code. The automatic generation of glue code requires a method to specify the target system in a manner that exposes all the characteristics of the system to a process that generates the glue code. We propose a graphical, model-based approach as an efficient way to represent the salient features of the sub-systems of a spacecraft.

Our model-based tool is implemented based on the Generic Modeling Environment (GME)[3]. GME is a metaprogrammable toolkit for developing domain specific modeling tools. A visual modeling language allows the graphical representation of a system, and supports a fixed set of rules. This paper describes the visual modeling language used to support the specification of rapidly integrated spacecraft systems, as well as a code generator which translates the visual specifications into code-level middleware to support device communication and on-board self test.

RELATED WORK

In rapidly integrated systems, modularity plays a crucial role. In scenarios where a system needs to be assembled and fully functional in a very short time, developers cannot afford the time and effort required for detailed component design and implementation. This leads to the preference for commercial off-the-shelf (COTS) components that can be used directly in a system. COTS-based system development differs from traditional development approaches through its focus on modularity. Moreover, it is observed that in the case of COTS-based development, there is a tradeoff between the system requirements and the products available and the architecture of the system[4]. A common challenge faced when using modular components in projects is that not all the components in the system are designed to interoperate. A common problem in projects that use COTS components is hasty commitment to a product or the use of a very tightly coupled subset of products. The end users have no control over the functionality and the performance of a COTS product. Moreover, the designers have no control over the evolution of a COTS product. The behavior of a COTS product is largely determined by the vendor. The evolution of a COTS product is something that is governed by the market and not the needs of the project where it is being used. The software development process when using COTS components is quite different from the process followed when the components are custom built[5], however, the time for development of the system as a whole is reduced greatly if the correct components and methods are chosen. The concept of combining components into electronic systems is an approach advocated by Air Force Research Laboratories (AFRL); the term Responsivonics[2] is a description of the same.

Another instance of COTS components being used to build a functional embedded system is provided by Cotterell et al[6]. The goal of their study was to come up with a approach whereby an end user with little or no knowledge of programming or engineering can build simple yet useful control and monitoring systems out of composable components called eBlocks. An eBlock is a rudimentary hardware assembly that performs simple tasks. eBlocks are also used to facilitate inter-block communication.

Satellite Data Model

The Satellite Data Model (SDM) is a part of the Responsive Space Test bed for the Air Force Research Laboratories (AFRL)[7]. The SDM is a middleware application that provides the capabilities of self-configuration and self discovery in networks of applications, sensors and actuators. The SDM allows plug-and-play-based hardware communications interfaces to devices. It also allows intelligent devices to describe their controls and data formats to the network using an electronic data sheet. SDM allows application software to query a device to discover its requirements and capabilities for both control and data. By facilitating the query of device capabilities and needs, SDM offers the capability of a glueless rapid system integration, which is capable of dynamic device interfacing without a-priori knowledge of an expected system configuration.

However, while SDM has the facilities to detect device failure at runtime, it does not directly support an on-board self-test to determine whether an expected set of devices has indeed been plugged into the system. Further, it offers support for true dynamic system integration, with the capability of interfacing any device that can be plugged into the system. While this level of configurability offers the benefits of dynamic configuration, it can lead to unnecessary middleware bloat. A rapidly generated customized middleware layer offers the same level of configurability when

composing a static system configuration, without the extra layers of configurability needed to support devices which are never actually plugged into the system.

Java-based Plug-and-Play

Orogo et al[8] offer an approach that specifies a Java-based implementation of the SDM. This approach makes use of the Jini[9], a dynamic distributed architecture for providing spontaneous networking of devices. Jini allows for dynamic deployment and configuration of distributed systems. While Jini offers the dynamic capabilities required to support rapid system integration, the extra layers of middleware required to implement the distributed protocols may not be appropriate for computationally constrained spacecraft platforms.

Middleware Generation

The use of middleware to support system integration has been reduced to practice in large scale software systems. Some of the common middleware frameworks are Enterprise Java Beans[10] (EJB) and Common Object Request Broker Architecture (CORBA)[11]. EJB provides a specification for the implementation of business logic. It is used mainly in enterprise applications. CORBA allows multiple pieces of software to interact with each other. These pieces of software may be in different languages executing on different physical platforms. The interfaces of each object in the system are exposed to other objects in the system using an Interface Definition Language (IDL). While EJB and CORBA have been applied in a variety of applications, their focus is more towards large scale distributed systems, while our focus is on rapid system and device integration into electronics platforms consisting of small numbers of CPUS (ex. 1-5 CPUs).

BACKGROUND

Universal Serial Bus

The Universal Serial Bus (USB)[12] protocol offers a plug-n-play communications infrastructure appropriate for rapid systems integration. USB employs a tiered-star topology and is a host-centric protocol. The USB host controller is responsible for initiating all transactions that take place on the USB bus. A device that is connected to the host is called a function. A device or a function may provide many capabilities such as a web camera, printer, thumb drive, etc. All communication between the host controller and the device takes place using endpoints on the device. Data transfer occurs through virtual pipes connecting the endpoints and the host controller. A virtual pipe is used to transfer data between the host and the endpoint on a USB device. The endpoint is either the source or the destination of the transfer. Since the USB is a host-centric protocol the endpoints marked as "IN" are used for sending data from the device to the host, and those marked as "OUT" are used to send data from the host to the device.

The endpoints in a USB device are bundled into interfaces. The interfaces in turn are bundled into configurations. A device can have multiple configurations, and devices are allowed to switch between configurations. Only a single configuration can be enabled at any time. Whenever there is a change in the configuration of the device, the device notifies the host of this change. The information regarding the configurations is supplied to the host during the process of enumeration. Enumeration is the process whereby devices which are plugged into the bus are identified and acknowledged by the host controller. Host-based software is responsible for keeping track of endpoints, interfaces and configurations. The USB protocol supports plug-and-play of devices with drivers that can be loaded and unloaded dynamically.

libusb

USB-based I/O operations between the host controller and a device are normally managed with a device driver. Typically, device drivers must execute in privileged memory mode in the operating system. The *libusb*[13] project offers the capability of writing a non-privileged device driver for USB devices. *libusb* offers an O/S-independent API for communicating with USB devices. User software must link against a library containing the API calls. *libusb* offers a significant advantage over kernel-level device drivers, in that if abnormal program behavior is observed, a non-privileged program can be easily terminated, whereas killing a the execution of a kernel-space program can cause a full system crash. *libusb* includes calls to find all busses on a system and to find all devices attached to the busses on the system.

Electronic Datasheets

The IEEE has developed a family of protocols, IEEE1451[14], defining a set of standards and communication interfaces for the interconnection of transducers and microprocessors. The protocol defines a Transducer Electronic Datasheet (TED) as a means for an intelligent sensor or actuator device to communicate specification information electronically to an integration entity. The use of TEDs can facilitate system-level self-discovery. The TEDs follow a format for specifying the minimum mandatory data

(manufacturer, model number and serial number for the transducer) and hence, have a storage requirement of 256 bytes only.

In order to facilitate ease of automatic parsing of electronic datasheet information, researchers have proposed the use of a specialization of XML as a serialization format for TEDs[2]. The resulting XML-based TED, or xTED, can be stored in on-board flash memory of intelligent devices. If all xTEDs conform to a standardized schema file, then parsing of TED information can be automated.

The Generic Modeling Environment

As stated above, the aim of our research is to develop a design tool for supporting rapid system configuration and integration, as well as an approach for verification of the system post-assembly. Our tool consists of a domain-specific visual design environment, based on the Generic Modeling Environment (GME)[3,15]. GME allows the creation of visual design tools as well as interpreter/translator programs for mapping GUI-based system depictions into functioning software components.

GME is a customizable modeling framework that allows the creation of domain-specific models and modeling languages, with support for constructing tools for program synthesis and rapid code generation. While GME does not provide native support for any domain, it is extensible, facilitating the creation of new domain specific modeling languages. A *paradigm* refers to a modeling language. GME natively supports a modeling language to define new paradigms. The resulting visual model which describes a modeling language is called a *metamodel*.

A metamodel captures the entities and relationships that exist in the user's selected domain. For example, should the user whish to create a modeling tool for describing bicycles, she would capture entities such as tires, handle bars, chains, fames, seats, etc., and relationships such as "connects to" (as in "the seat connects to the frame"). The metamodel supplies a complete set of rules that configures the GME for a specific domain. GME offers a translator tool which converts a metamodel into a paradigm, which can then be used to configure a new instance of GME.

The paradigm generated in during the configuration process can be likened to the grammar of a programming language that specifies the syntax rules of the language. The paradigm captures the rules for the domain-specific modeling language. Once GME is configured to support the concepts and rules corresponding to a particular domain, users are free to create system models. GME enforces the rules captured in the metamodel, notifying the user when he has done something illegal.

Visual system models amount to little more than documentation, unless users are allowed to extract programmatically the information captured in the GUI. GME supports the extraction of model information through a process called interpretation. A model interpreter utilizes an API to access and traverse the objects captured in the model. GME supports several language bindings for interpreter creation, such as C++, Java, Python, etc. A model interpreter is analogous to a compiler of a programming language, which translates high level constructs into low-level executable statements (i.e. assembly language). An interpreter parses a model and, depending on the application domain, generates an implementation of the modeled system, or maps the model onto some analysis tool.

The Universal Data Model (UDM)[16] is a framework for creating, among other things, GME-based model interpreters. It offers the ability to translate GME-specified models into a domain-specific XML format, allowing for chaining of multiple domain specific design and analysis tools. A UDM-provided translator tool can be applied to a user-specified metamodel, resulting in the generation of a paradigm-specific C++ API. This API can be used to create, access, traverse and manipulate a GME-based model.

MODELING PLUG-N-PLAY SPACECRAFT

In order to support the synthesis and verification of rapidly integrated plug-n-play-based spacecraft systems, we have developed a visual modeling tool to allow the user to describe the system under construction. We have developed a metamodel for modeling such systems. The focus of our approach is on the automatic generation of an on-board self-test. The self-test software executes after the system has booted and discovered all devices which have been plugged in. The verification software then determines whether an expected set of devices has been plugged in, and whether each device is responding properly. The modeling tool supports the specification both of what devices should be expected in the system configuration, and of how to determine, through an exchange of messages, whether a device is functioning as expected.

In the following subsections, we describe the design of our modeling environment, by documenting the metamodel, together with the rationale behind why the metamodel was developed as it was. Subsequently, we describe the model interpreter.

Tool Overview

Figure 1 describes the architecture of our toolflow. The tool consists of a metamodel, which specifies a GME-based modeling paradigm. GME is configured with this paradigm and can be used to create system models. A model interpreter has been developed, which performs translations on the GME model. The artifacts of translation are 1) a suite of generated middleware to facilitate access to the integrated set of devices, and 2) an on-board self test software sequence which can be employed at runtime to verify whether the integrated system contains the proper devices, and whether those devices function as expected.

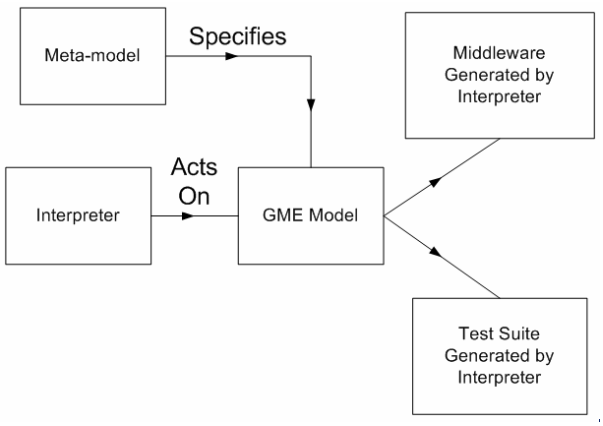

Figure 1. GME-based Toolflow

Metamodel Overview

The metamodel specifies a domain specific modeling language for describing modular spacecraft panels, and the devices which are plugged into the panels. GME utilizes a variant of UML class diagrams as the means for specifying a metamodel.

Metamodel: Panels and Boards

In our tool, the entire system is modeled as a panel. A panel is an entity into which single-board computers and other electronics are physically embedded. USB-based electronic components can be interfaced to a panel through the integrated USB bus.

A board represents a Single Board Computer (SBC). It contains at least one host controller that that drives the USB bus on the SBC. The host controller is responsible for enumerating the USB devices and assigning them an address on the USB bus. Embedded into a panel with the SBC is a set of ports for interfacing to external devices. A host controller drives the USB bus and can support up to 127 USB devices on that bus.

Figure 2 provides the metamodel description of Panel objects. A Panel has a string attribute called panelName, and contains one or more Boards. A Board object represents a single board computer embedded in the panel. A SBC may offer multiple host controllers; we capture this relationship by allowing a Board object to contain one or more Host objects. Each Host object is assigned a unique integer identifier.

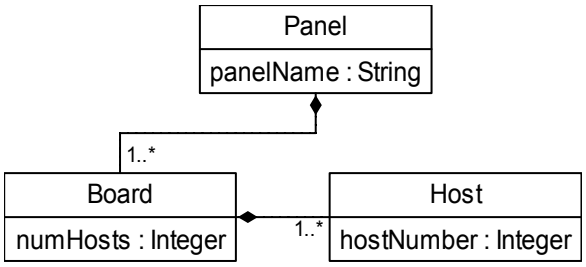

Figure 2. Metamodel describing a Panel and its relation to a Board and Hosts

Devices

A USB device (sensor, accelerometer etc.) can be plugged into a USB hub or port connecting it to the host controller. For the devices we consider, this connection is the only medium of communication that the device has with the host computer. We model intelligent USB devices, which are packaged with an xTED that contains a description of the relevant device specifications and behavior. The relevant characteristics of a device include, but are not necessarily limited to, manufacturer name, device name, vendor ID, product ID, serial number and device class. The vendor ID and product ID form a globally unique key for the device, and are assigned by the device manufacturer. This key is used in the process of enumeration.

Figure 3 depicts the metamodel allowing the representation of devices. Device objects are required to have a DevicePort object and an XTED object. The DevicePort object is used to support the representation of connectivity with the host, as will be demonstrated below. The DevicePort object is assigned an integral deviceNumber attribute, an ID which is unique within the scope of devices targeting a particular host. The Device object has a list of attributes as described above. The association between an XTED and a Device is captured as a containment relationship in the metamodel, specifying that every device is associated with a particular XTED.

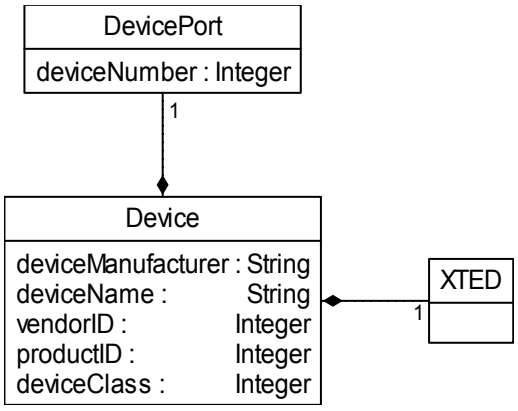

Figure 3. Metamodel describing a Device

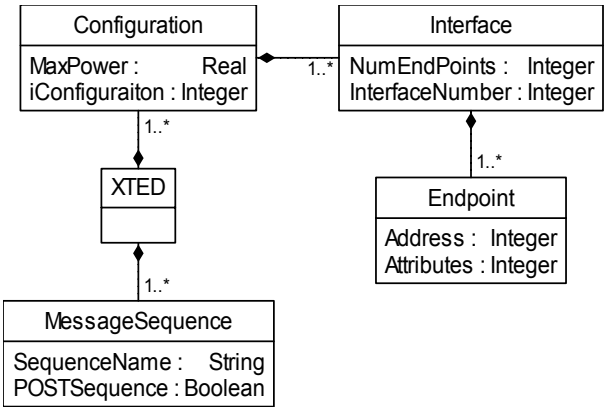

Figure 4. Metamodel description of an xTED

XTED

While the IEEE 1451 protocols specify information about the TED, the design community has not completely specified all the necessary format and content information for an xTED. However, for the purposes of this project, we offer the following as potential content for an xTED.

An xTED should contain protocol-specific information about the device. This includes all the endpoints, interfaces and configurations that are supported in the device. The endpoints are uniquely addressable sources or sinks on the bus. They are bundled into interfaces. The interfaces handle one type of logical connection. The interfaces represent the basic functionality of the device. The interfaces are bundled into configurations. An xTED also contains one or more message sequences, which will be discussed below.

Figure 4 depicts the metamodel for an XTED object. In our system, we assume an XTED is composed of a set of Configuration objects, and a set of MessageSequence objects. Each Configuration object is attributed with information on the maximum power consumption of the configuration, and an integer ID assigned to the configuration. A Configuration object is composed of one or more Interface objects. Each Interface is characterized with two attributes, one capturing the number of end points associated with it, and another assigning to the interface an integral ID. An Interface contains EndPoint objects. Each EndPoint object is assigned an address and a set of attributes. This information is communicated to the host during enumeration of the device at system configuration time. The MessageSequence objects will be described in more detail in the following section.

On-Board Self-Test

Our model-based approach facilitates not only the representation of rapidly assembled systems, but also facilitates the verification of such systems. The concept of an on-board self-test is important for rapidly constructed systems, since the time available for testing the integrated system is limited. While plug-n-play hardware and software interfaces facilitate the dynamic integration of devices, they contain no direct notion of correctness. Such systems lack, beyond only simple "heartbeat" communications, the ability to determine whether a device is functioning as expected. Further, plug-n-play networks typically have no notion of what devices and configurations to expect at runtime, and therefore cannot notify the user prior to deployment of integration errors.

Given the cost constraints on spacecraft design and deployment, we propose with our model-based tool framework an approach which directly integrates device and system verification into the development of rapidly integrated and deployed spacecraft systems. Our approach is two-fold. First, our tool allows the user to specify the expected communication behavior for a device. From this specified behavior, the tool derives a message-based testing sequence for each device in the system, which can be applied at any time post enumeration to ascertain the health of a device. Second, the tool facilitates the specification of the expected system configuration(s). The specification of what devices should be present in the system configuration allows the tool to generate an integration test, which checks the dynamically discovered device set against the pre-specified configuration, and notifies the user of any discrepancies.

The rationale behind our approach to system verification is based on the deployment scenario for

spacecraft, especially small satellites. Small satellites tend to have limited mission lifetimes, and are not serviced post-deployment. Their configuration tends to be static in nature, in the sense that the set of devices used in the system are not changed after the spacecraft is deployed. However, current approaches to static system integration involve the manual configuration of system "glue", typically software-based, to manage both device interfacing and application usage of devices. As discussed above, research efforts into rapid systems integration seek to reduce or remove the level of "glue" needed to manage device interfacing through the use of dynamic, plug-n-play bus structures. The runtime integration of devices in most small spacecraft are actually only used at system configuration time, not post-deployment.

The model based tool framework facilitates the specification of a static configuration for a spacecraft system, together with software infrastructure to verify that the deployed configuration matches that which is expected. Should the user determine a change in the configuration is needed, she simply changes the system model and re-generates the software infrastructure.

Message Sequences

A message sequence is used to specify the correct communication behavior of a device. A message sequence specifies a stimulus and response sequence for a given device, as an interaction between the device and the host. With a pre-specified message sequence, a host can initiate a message sequence, and then observe whether the actual device communication conforms to the pre-specified sequence. Lack of conformance indicates device failure. While no single test can cover all potential device faults, message sequencing can aid in uncovering device communication and interaction faults.

The model-based approach provides an intuitive, graphical means of specifying message sequences. Figure 5 depicts the metamodel description of message sequences. A message sequence is modeled as part of an xTED, as shown in Figure 4. The fundamental block of a message sequence is a message. A message can either be in-bound (from device to host controller) or out-bound (from host controller to device). The set of message sequences associated with a device specifies the communication protocol between the host and device. A message is modeled as with a Message object. The metamodel supports two types of messages, an InMessage, modeling an in-bound message, and an OutMessage, modeling an out-bound message. A MessageSequence also contains a Start object and a Stop object. These objects do not actually represent communicated information. They are for visual context, and are used in the model interpreter to determine the beginning and ending points of a sequence.

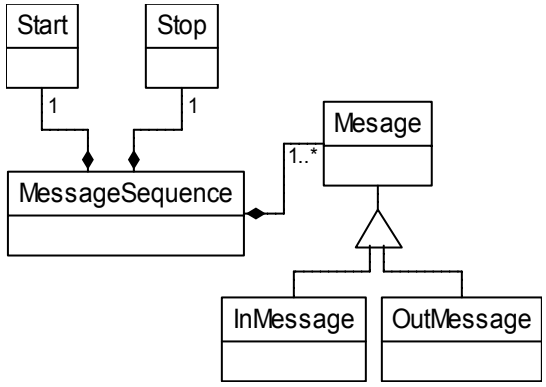

Figure 5. Metamodel description of message sequences

Model Interpreter

The metamodel described above facilitates the automatic generation of a GME-based modeling environment, which allows the user to graphically describe a spacecraft system. The modeling environment supports and enforces the specification rules captured in the metamodel. However, visual models and diagrams are of little use beyond documentation if no translations can be applied. We have developed a model interpreter to extract and translate the information captured in the graphical models into useful artifacts for the spacecraft systems designer. As depicted in Figure 1, the model interpreter acts on a GME model that conforms to the metamodel outlined above, and produces two artifacts:

- A device-specific middleware API for supporting application-device communications
- A test suite, consisting of software to query device health, and a verification of the device configuration set.

Platform-Specific Middleware

The first artifact is related to the second, and comes as an artifact of the representation of message sequences. Since the models capture device communication protocols with the message sequences, it is possible to generate a device-specific API, wrapping the libusb device-independent API, to support host-device communications. The test suite software can then be composed in terms of this device specific API.

The generated software API takes the form of a set of C header and source files, one pair for each device in the system. The interpreter traverses the system model hierarchically, starting with the Panel object. On encountering a Device object, the interpreter creates a header file corresponding to the device. The interpreter writes to the device header files the pertinent device details such as endpoint addresses and configuration IDs, together with messages specific to the device. The interpreter generates for each device methods for supporting I/O. These methods encapsulate the API calls provided by the libusb library. By wrapping the low-level device details, the user is presented with a simple API for device reading and writing. This generated API is referred to as platform-specific middleware.

The platform-specific middleware provides the end user with an API for reading from and writing to the devices connected to the single board computer in the panel. The middleware assumes that the libusb library is available as a shared object (.so) file on the target platform.

Test Suite

The test suite offers the capability of ensuring that the device configuration for the system is as expected, and that all devices are communicating as expected. The test suite takes the form of a set of generated C functions which implement the verification checks based on the generated API.

As the interpreter parses the system models, it collects a list of Device objects which are present in the system. From this list, the interpreter produces a table of devices, each identified by a globally unique VedorID+ProductID key. This table is dumped as an array in the output C file. A C function is generated which queries the currently loaded set of devices against this table, to determine whether each device that is specified in the model is present in the system. If a mismatch occurs, the host computer enters into an error state, and the user is notified. In future revisions of the tool, we will examine the generation of a deployment scheme for dealing with device failure at configuration time, post-deployment. Currently, our implementation assumes that the user is testing a panel, and has a debug terminal attached.

Each device captured in the modeling tool is associated with one or more message sequences. The interpreter parses the message sequences associated with a device, and translates them into a sequence of I/O function calls. These function calls make use of the generated platform-dependent middleware API for reading from and writing to a device. Figure 6 provides an example message sequence for a device. The sequence begins by the host sending to the device a message, consisting of the ASCII character ‘A’. The device responds by sending the ASCII character ‘B’.

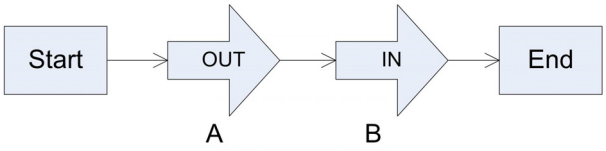

Figure 6. Example message sequence

Figure 7 provides a simplified code snippet reflective of the code the interpreter generates to implement the self-test from the message sequence depicted in Figure 6. The function calls TxDev and RxDev represent the I/O functions that are generated in the platform-specific middleware.

```
(1)     char c;
(2)     TxDev(Dev, "A", 1);
(3)     RxDev(Dev, &c, 1);
(4)     if(c != 'B') {
(5)         Error("Expected B");
```

Figure 7. Code snippet implementing message sequence of Figure 6.

The interpreter produces a code snippet similar to the above for each message sequence in the system. Each snippet is wrapped by a C function. The interpreter generates a test suite implementation function which invokes all the generated tests for each device in the system. This generated test suite is available for invocation at system configuration time, and is available for re-invocation at any point during the system lifetime.

EVALUATION AND RESULTS

We have evaluated the model-based tool by developing a simple application targeting an ARM-based embedded platform. The platform consists of a single board computer, the TS 7260 board from Technolgic Systems, and a USB smart-sensor, the DLP-TEMP-G temperature sensor from DLP Design.

The TS 7260 is a single board computer based on the Cirrus Logic EP9302 ARM9 CPU. The EP9302 processor features a 200MHz ARM920T processor with a memory management unit (MMU). One of the key features of the 7260 is its low power consumption (<1 watt, full speed, <¼ watt at low clock speeds). Figure 8 shows a photograph of the TS-7260 board. The board

offers two RS-232 ports, two USB ports, an RJ-45 connector for Ethernet, and a PC/104 bus connector for expansion.

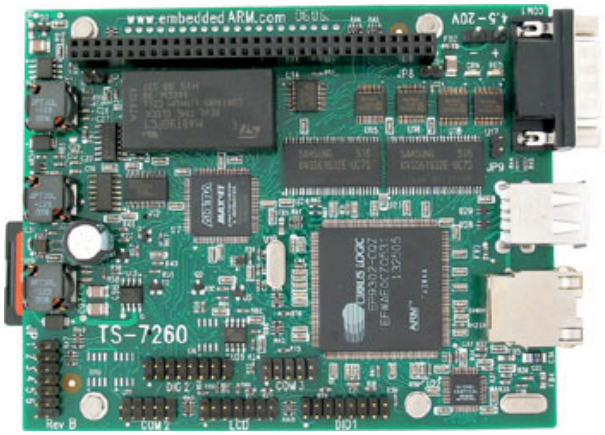

Figure 8. The Technologic Systems TS-7260 computer

The 7260 is packaged with an embedded version of the Linux operating system based on the version 2.4.26 of the Linux kernel. The 7260 uses the Redboot boot loader, and can support a variety of operating systems. The 7260's RS-232 serial port can be used to interface to a host PC. The 7260's Linux O/S supports the use of the Ethernet port as a means of host PC communication as well. Technologic systems provides a suite of GNU-based cross compiler/linker/debugger tools for supporting software development.

DLP-TEMP-G

The DLP-TEMP-G is a USB-based programmable data acquisition system. It has three digital I/O pins and it supports up to three temperature sensors. It offers a PIC12F683 programmable microcontroller, which can accept user-specified code. It has a male type A USB connector for direct connection to a USB port. Figure 9 shows a DLP-TEMP-G device.

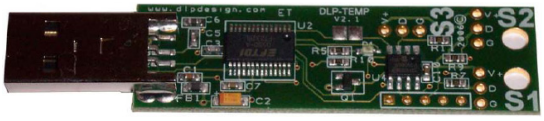

Figure 9. DLP-TEMP-G Data/Temperature acquisition board

The PIC12F683 is an 8-pin microcontroller that is pre-programmed for analog voltage measurement and for reading the on-board digital temperature sensors. The firmware for the microcontroller is written in C and can be reprogrammed using an appropriate cross-compiler and programmer. The USB-to-serial interface on the chip is implemented using the FT232RL USB-UART chip from FTDI. The device employs a simple, character-based communication protocol, implemented in software on the microcontroller. The device accepts various characters as requests and returns appropriate results for each request.

Example Model

We demonstrate the GME-based modeling environment by modeling the TS-7260 as a host computer connected to a single DLP-TEMP-G board. The model for this simple system is depicted in Figure 10. In the diagram, the icon labeled "Board" represents the 7260 board. The large box outlines illustrate the levels of hierarchy in the tool. The Device icon models the DLP-TEMP-G device. It is associated with a DevicePort object, and an xTED object. The xTED object contains a single MessageSequence object. That MessageSequence consists of an OutMessage, followed by an InMessage. The contents of these messages are not shown in the figure, but are captured as attributes associated with the icons. Similar to the example given in Figure 6, the OutMessage consists of the ASCII character 'A', and the InMessage consists of the ASCII character 'B'.

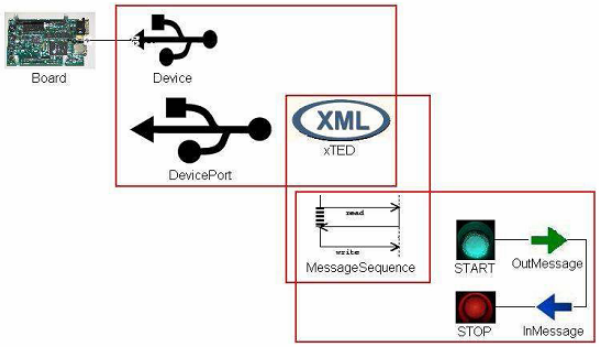

Figure 10. Example system model

On invocation, the interpreter parsed the example model, and generated a set of C source files, providing the implementation of the middleware API for the DLP device, as well as the code to implement the self-test. This generated code was bundled with a simple application program, which invokes the test software, and was cross-compiled and loaded on the board. Upon execution, the self test software reported a successful integration. A separate test involved switching the DLP sensor for a different sensor board. In this case, the execution of the on-board self test reported a failure, since an unexpected device was plugged into the system. We then re-programmed the DLP board with a different communication protocol, where in response to an 'A'

received from the host machine, it returns a 'Z' instead of a 'B'. On plugging the DLP board back into the 7260 and executing the self-test software on boot-up, the self-test software reported an error saying that the DLP device had failed a message sequence test.

CONCLUSION

The design of rapidly integrated small spacecraft systems presents several unique challenges. Designers are pressed with the goal of assembling a functioning spacecraft system from off-the-shelf subsystem modules in a very short time period. Rapid systems integration leaves no time for module design, and little time for testing. Given the cost implications of deployment failures of spacecraft systems, the need for integration of verification into the design process of rapidly integrated spacecraft systems is plainly apparent.

In this paper, we have presented a model-based toolset for supporting the specification of small scale spacecraft system electronics, as well as the automatic generation of verification software which can be executed at system configuration time. The model-based toolset had three main objectives:

- The graphical representation of spacecraft system electronics,
- The generation of device communications test software,
- The generation of system configuration test software.

We have developed a domain-specific visual modeling environment, based on the Generic Modeling Enviornment (GME). Our modeling environment encodes the rules for specifying modular spacecraft panels with embedded electronics, together with the representation of a set of devices interfaced to those electronics.

We have developed a model interpreter for our modeling environment, which is capable of rapidly generating both platform-specific middleware for supporting USB-based device communication, as well as test software suites for checking the dynamically discovered set of devices against the configuration set specified in the models, and for querying the health of each integrated device. Health determination was performed based on a set of message sequences captured in the models.

The tool framework described in this paper offers only a limited set of verifications for rapidly integrated systems. However, the concept of using a model-based toolset to drive system configuration facilitates the intuitive specification of configuration information which remains unavailable in truly dynamic system integration. Further, a generative approach can potentially trim unneeded excess in general purpose middleware by tailoring the device communications infrastructure to fit the set of devices actually employed in the system. While our tool currently attempts only to address the generation of on-board self-test software, there exists several potential benefits to a model-based approach that will be examined in the future.

Acknowledgements

This work was funded in part by the State of Utah through the Center for Advanced Satellite Manufacturing at Utah State University.

References

1. Stucker, B. and Q. Young, "Rapid Manufacturing of Reconfigurable Satellite Panels with Embedded Electronics, Embedded Thermal Devices, and Novel Structural Features," Proceedings of the 20th Annual AAIA/USU Conference on Small Satellites, Logan, UT, 2006.

2. Lyke, J., S. Cannon, D. Fronterhouse, D. Lanza, and T. Byers, "A Plug-and-Play System for Spacecraft Components Based on the USB Standard," Proceedings of the 19th Annual AIAA/USU Conference on Small Satellites, Logan, UT, 2005.

3. "GME User's Manual," Vanderbilt University March 2004.

4. Torchiano, M. and M. Morisio, "Overlooked aspects of COTS-based development," IEEE Software, vol. 21(2), pp. 88-93, March-April, 2004.

5. Vigder, M.R. and J. Dean, "An architectural approach to building systems from COTS software components," Proceedings of the Centerfor Advanced Studies Conference, 1997.

6. Cottrell, S., F. Vahid, W. Najjar, and H. Hsieh, "First Results with eBlocks: Embedded Systems Building Blocks," Proceedings of the Hardware Software Codesign and System Synthesis (CODES+ISSS), 2003.

7. Cannon, S., "The Satellite Data Model," Proceedings of the International Conference on Embedded Systems and Applications, 2006.

8. Orogo, C., M. Enoch, and D. Flaggs, "Java-based plug-n-play (flight) control systems for responsive space," Proceedings of the 4th Responsive Space Conference, 2006.

9. http://www.jini.org

10. http://java.sun.com/products/ejb/

11. http://www.corba.org/

12. http://www.usb.org

13. http://libusb.sourceforge.net/

14. Lee, K., "IEEE 1451: A standard in support of smart transducer networking," Proceedings of the IEEE Instrumentation and Measurement Technology Conference, 2000.

15. Ledeczi, A., et al., "Composing domain-specific design environments," Computer, vol. 34(11), pp. 44-+, NOV, 2001.

16. 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.