CubedOS: A Verified CubeSat Operating System

Carl Brandon, Peter Chapin, Chris Farnsworth, Sean Klink
Vermont Technical College, 201 Lawrence Place, Williston VT 05495, PO Box 500, Randolph Center, VT 05061; email: {cbrandon, pchapin}@vtc.edu

Abstract

In this paper we present CubedOS, a lightweight application framework for CubeSat flight software. CubedOS is written in SPARK*and proved free of certain classes of runtime errors. It consists of a collection of interacting, concurrent modules that communicate via message passing over a microkernel based on Ada's Ravenscar tasking model. It provides core services such as, for example, communication protocol processing and publish/subscribe message handling. Application-specific modules can be added to provide both high level functions such as navigation and power management, as well as low level device drivers for mission-specific hardware.*

Keywords: SPARK, student project, CubeSat

1 Introduction

CubedOS is being developed at Vermont Technical College's CubeSat Laboratory with the purpose of providing a robust software platform for CubeSat missions and of easing the development of CubeSat flight software. In many respects the goals of CubedOS are similar to those of the Core Flight Executive (cFE) written by NASA Goddard Space Flight Center [1]. However, unlike cFE, CubedOS is written in SPARK and verified to be free of the possibility of runtime error. SPARK has also been used to provide some other correctness properties in certain cases. We compare CubedOS and cFE in more detail in Section 4.

The intent is for CubedOS to be general enough and modular enough for many groups to profitably employ the system. Since every mission uses different hardware and has different software requirements, CubedOS is designed as a framework into which *modules* can be plugged to implement whatever mission functionality is required. CubedOS provides inter-module communication and other common services needed by many missions. CubedOS thus serves both as a kind of operating environment and as a library of useful tools and functions.

Some of the module functionality useful for complex CubeSat missions would include interfaces to attitude determination and control systems (ADACS), electrical power systems (EPS), photovoltaic panel orientation gimbals, navigation and data radio, data collection instruments, thermal control radiators, ion engine with gimbals, and cameras. We also plan on including a specific module for spiral thrusting which allows for three axis angular momentum control with a two axis thruster.

It is our intention that all CubedOS modules also be written in SPARK and at least proved free of runtime error. However, CubedOS allows modules, or parts of modules, to be written in full Ada or C. This allows CubedOS to take advantage of third party C libraries or to integrate with an existing C code base.

CubedOS runs on top of the Ada runtime system and thus works with any underlying platform supported by the available Ada compiler. For example, CubedOS makes use of Ada tasking without directly invoking the underlying system's support for threads. This simplifies the implementation of CubedOS while improving its portability. However, CubedOS does require that a rich Ada runtime system be available for all envisioned targets. Specifically, CubedOS requires a runtime system that supports the Ravenscar profile.

For resources that are not accessible through the Ada runtime system, CubedOS driver modules can be written that interact with the underlying operating system or hardware more directly. Although these modules would not be widely portable, they could, in some cases, be written to provide a kind of low level abstraction layer (LLAL) with a portable interface. We have not yet attempted to standardize the LLAL interface. However, we see that as an area for future work.

CubedOS applications are organized as a collection of active and passive modules, where each active module contains one or more Ada tasks. Passive modules do not contain any tasks but are used as containers for shared, reusable code. Although CubedOS is written in SPARK there need not be a one-to-one correspondence between CubedOS modules and SPARK packages. In fact, modules are routinely written as a collection of Ada packages in a package hierarchy.

Critical to the plug-and-play nature of CubedOS, each active module is self-contained and does not make direct use of any code in any other active module, although passive modules serving as library components can be used. All inter-module communication is done through the CubedOS infrastructure with no direct sharing of data or executable content. In this respect CubedOS active modules are similar to operating system processes. One consequence of this policy is that a library used by several modules must be either duplicated in each module, for example as private child packages, or provided as an independent, passive module. In this respect passive modules are similar to operating system shared libraries and have similar concerns regarding task safety and global data management.

In the language of operating systems, CubedOS can be said to have a microkernel architecture where task and memory management is provided by the Ada runtime system. Both low level facilities, such as device drivers, and high level facilities, such as communication protocol handlers, are all implemented as CubedOS modules. All modules are treated equally by CubedOS; any layered structuring of the modules is imposed by programmer convention.

CubedOS is currently a work in progress It is our intention to release CubedOS as open source once it is more mature and refined. We also need to review the code base to verify that it is free from International Traffic in Arms Regulations (ITAR) restrictions and possibly release both ITAR compliant and U.S non ITAR compliant versions. We anticipate this to happen in mid-2018.

2 CubedOS Architecture

To understand the context of the CubedOS architecture, it is useful to compare the architecture of a CubedOS application with that of a more traditional application. Since CubedOS is written in SPARK and must abide by the restrictions of Ravenscar, we compare CubedOS with other Ravenscar-based approaches.

Figure 1 shows an example application using Ravenscar tasking. Tasks, which must all be library level infinite loops, are shown as open circles and labeled as T_1 through T_4. Tasks communicate with each other via protected objects, shown as solid circles and labeled as PO_1 through PO_4.

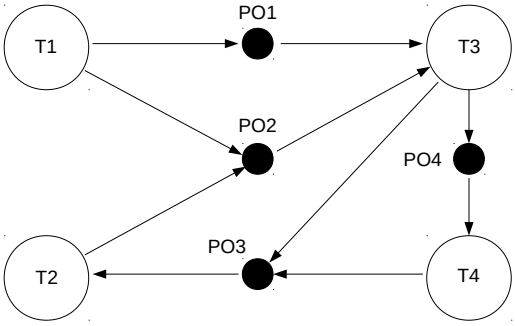

Figure 1: Traditional Ravenscar-based Architecture

Arrows from a sending task to a protected object indicate calls to a protected procedure to install information in the protected object. Arrows from a protected object to a receiving task indicate calls to an entry in the protected object used to pick up information previously stored in the object. Entry calls will block if no information is yet available but protected procedure calls do not block.

Ravenscar requires that protected objects have at most one entry and that at most one task can be queued on that entry. In CubedOS applications each protected object is serviced by exactly one task. This ensures that two tasks will never accidentally be queued on the protected object's entry. In the figure this means only one arrow can emanate from a protected object. However, multiple arrows can lead to a protected object, since it is permitted for many tasks to call the same protected procedure or for there to be multiple protected procedures in a given protected object.

In the example application of Figure 1, tasks T_1 and T_3 call protected procedures in two different protected objects. This presents no problems since protected procedures never block, allowing a task to call both procedures in a timely manner. However, task T_3 calls two entries, one in PO_1 and another in PO_2. Since entry calls can block, this means the task might get suspended on one of the calls leaving the other protected object without service for an extended time. The application needs to either be written so that will never happen or be such that it doesn't matter if it does.

There are several advantages over the traditional organization:

- The protected objects can be tuned to transmit only the information needed so the overhead can be kept minimal.
- The parameters of the protected procedures and entries specify the precise types of the data transfered so compile-time type safety is provided.
- The communication patterns of the application are known statically, facilitating analysis.

However the traditional architecture also includes some disadvantages:

- The protected objects must all be custom designed and individually implemented, creating a burden for the application developer.
- The communication patterns are relatively inflexible. Changing them requires overhauling the application.

A CubedOS application has an architecture as shown in Figure 2. In this case CubedOS provides the communication infrastructure as an array of general purpose, protected mailbox objects. CubedOS modules communicate by sending messages to the receiver module's mailbox. The messages are unstructured octet streams, and thus completely generic. Each active module has exactly one mailbox associated with it and contains a task dedicated to servicing that mailbox. That task extracts messages from the mailbox, and then decodes and acts on each message. Active CubedOS modules can also contain internal tasks as part of their implementation, but those tasks do not participate in message processing (although they can send messages) and are not important here.

The communication connections shown in Figure 2 are the same as those shown in Figure 1 except that the two communication paths from T_1 to T_3 are combined into a single path going through one mailbox.

CubedOS relieves the application developer of the problem of creating the communications infrastructure manually. Adding new message types is simplified with the help of a tool, XDR2OS3, that we describe in Section 3. In addition to providing basic, bounded mailboxes, CubedOS also provides other services such as message priorities and multiple sending modalities (for example, best effort versus guaranteed delivery). Many of these additional services would be tedious

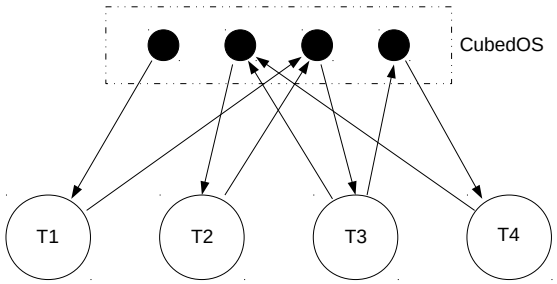

Figure 2: CubedOS-based Architecture

to provide on a case-by-case basis following the traditional architecture. CubedOS also allows any module to potentially send a message to any other module. Thus the communication paths in the running application are very flexible and dynamic.

Although the CubedOS architecture supports only point-to-point message passing, the CubedOS system provides an active module supporting a publish/subscribe discipline. The module allows multiple channels to be created to which other modules can subscribe. Publisher modules can then send messages to one or more channels, allowing for message broadcast and multicast. Since the messages themselves are unstructured octet streams, the publish/subscribe module can handle them generically without being modified to account for new message types.

Every CubedOS module has a statically assigned ID number. Messages sent from a module include the ID number of the sender. This allows a server module to return reply messages without statically knowing its clients. Thus server modules can be written as part of a general purpose "module library" and used without modification in a variety of applications. We have started compiling a registry of "well known" module IDs for common services, such as file handling and timer services. This allows CubedOS module libraries to make use of well known services and remain reusable. Here active CubedOS modules resemble network clients and servers where the module IDs play the role of a network address. Extending the architecture across different physical machines, or between different operating system processes on the same machine, is an interesting area for future work.

We are also defining standard message interfaces to certain services, such as file handling, that third party modules could implement. This allows modules to use a service without knowing which specific implementation backs that service.

However, CubedOS's architecture also carries some significant disadvantages as well:

- All mailboxes must have the same size since they are stored in an array. Some mailboxes will be larger than necessary, wasting space.
- All messages must have the same type and thus the same size. Some messages will be larger than necessary and slower to copy than necessary.

  The common message type also requires that typed information sent from one module to another be encoded into a raw octet format when sent, and decoded back into specifically typed data when received. This encoding and decoding increases the runtime overhead of message passing and reduces static type safety. Modules must defend themselves, at runtime, from malformed or inappropriate messages, causing certain errors that were compile-time errors in the traditional architecture to now be runtime errors. This is exactly counter to the general goals of high integrity system development.
- In order to return reply messages, the mailboxes must be addressable at runtime using module ID numbers. Accessing a statically named mailbox isn't general enough. As a result, the precise communication paths used by the system cannot easily be determined statically.

  In particular, since SPARK does not attempt to track information flow through individual array elements, it is necessary for us to manually justify certain SPARK flow messages. The architecture of CubedOS ensures that there is a one-to-one correspondence between a module and its mailbox. The tools don't know this, and the spurious flow messages they produce must be suppressed.

The details of CubedOS mitigate, to some degree, the problems above. For example, the mailbox array is actually instantiated from a generic unit by the application developer. This allows the developer to tune the sizes of the mailboxes, and the messages they contain, to the application's needs. CubedOS does not attempt to provide a one-size-fits-all mailbox array that will be satisfactory to all applications.

Also every well behaved CubedOS module should contain an |API| package with subprograms for encoding and decoding messages. This package is generated by the XDR2OS3 tool that we describe in Section 3. The parameters to these subprograms correspond to the parameters of the protected procedures and entries in the traditional architecture, and provide much of the same type safety. However, using the API subprograms is not enforced by the compiler. It is also possible to accidentally send a message to the wrong mailbox. Thus modules still need to include runtime error checking to detect and handle these problems.

So far we have described two extremes: a traditional approach that does not use CubedOS at all, and an approach that entirely relies on CubedOS. However, hybrid approaches are also possible. Figure 3 shows a combination of several CubedOS mailboxes and a hand-made, optimized protected object to mediate communication from T_3 to T_4.

This provides the best of both worlds. The simplicity and flexibility of CubedOS can be used where it makes sense to do so, and yet critical communications can still be optimized if the results of profiling indicate a need. In Figure 3 task T_4 can't be reached by CubedOS messages. The hand-made

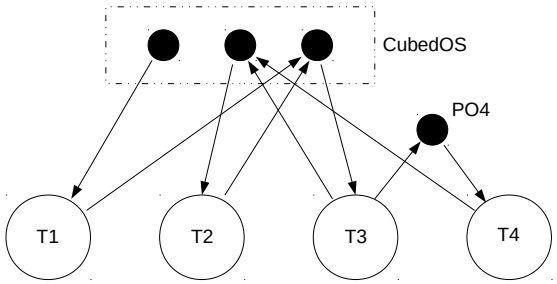

Figure 3: Hybrid Architecture

protected object creates a degree of isolation that can also simplify analysis as compared to a pure CubedOS system.

It is also possible to instantiate the CubedOS message manager multiple times in the same application, effectively creating multiple communication domains using separate mailbox arrays. Figure 4 shows an example of where T_4 is in a separate domain from the other modules (because it receives from a mailbox that is separate from the others).

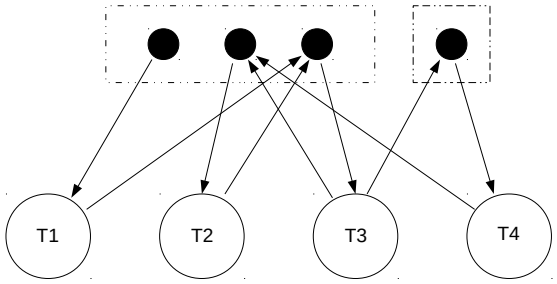

Figure 4: Multiple Communication Domains

This approach allows the CubedOS infrastructure to be used for easy development while still partitioning the system into semi-independent sections. For example, the sizes of the mailboxes and of the messages used in each communication domain need not be the same. The parts of the application that require large messages could be grouped into a domain separate from the parts that only require small messages.

Notice in Figure 4 tasks (modules) T_3 and T_4 send messages into multiple domains. This is, of course, sometimes necessary if the domains are going to interact. Modules that do this will need multiple module ID values scoped to different domains. At the moment the handling of this is largely a matter of manual configuration, which is reasonable for the relatively small programs typical of CubeSat missions. Creating a more comprehensive solution for module and domain addresses would be necessary as part of extending the architecture to multiple processes or machines as mentioned earlier. It is also likely that a naming service of some kind would need to be added to the module library provided by CubedOS. This is also an area for future work.

3 Message Encoding

CubedOS mailboxes store messages as unstructured octet arrays. This allows a general purpose mailbox package to store and manipulate messages of any type. Unfortunately this also requires that well structured, well typed message information be encoded to raw octets before being placed in a mailbox and then decoded after being retrieved from a mailbox.

The CubedOS convention is to use External Data Representation (XDR) encoded messages. XDR is a well known standard [2] that is also simple and has low overhead. We have defined an extension to XDR that allows SPARK's constrained scalar subtypes to be represented. We are currently working on a tool, XDR2OS3, that will compile a high level description of a message into message encoding and decoding subprograms. Our tool is written in Scala and is not verified, but its output is subject to the same SPARK analysis as the rest of the application. It is easier to prove the output of XDR2OS3 than it is to prove the correctness of XDR2OS3 itself.

The use of XDR2OS3 mitigates some of CubedOS's disadvantages. The developer need not manually write the tedious and repetitive encoding and decoding subprograms. Furthermore, those subprograms have well-typed parameters thus shielding the application programer from the inherent lack of type safety in the mailboxes themselves.

The use of XDR encoding may seem like an odd choice since XDR was originally defined for use in networking applications where data must be sent between potentially heterogenous systems. Since we envision current CubedOS applications being written entirely in SPARK and executing in a single process, XDR seems like a needless complication. However, as described in Section 2, we anticipate extending CubedOS to work in exactly the kind of potentially heterogenous environment XDR was developed to support. Thus we aim to provide a single standard for message encoding that will work in both the near and long term.

To illustrate CubedOS message handling, consider the following short example of a message definition file that is acceptable to XDR2OS3.

```
enum Series_Type { One_Shot, Periodic };

typedef unsigned int Module_ID
                       range 1 .. 16;
typedef unsigned int Series_ID_Type
                       range 1 .. 10000;

message struct {
    Module_ID      Sender;
    Time_Span      Tick_Interval;
    Series_Type    Request_Type;
    Series_ID_Type Series_ID;
} Relative_Request_Message;
```

This file introduces several types following the usual syntax of XDR interface definitions. The syntax is extended, however, to allow the programmer to include constrained ranges on the scalar type definitions in a style that is normal for Ada. The message itself is described as a structure containing various components in the usual way. The reserved word message prefixed to the structure definition, another XDR extension, alerts XDR2OS3 that it needs to generate encoding and decoding subprograms for that structure. Other structures serving only as message components (parameters) can also be defined.

XDR2OS3 has built-in knowledge of certain Ada private types such as Time_Span (from the Ada.Real_Time package). Private types need special handling since their internal structure can't be accessed directly from the encoding and decoding subprograms. There is currently no mechanism in XDR2OS3 to solve this problem in the general case.

Each message type has an ID number that is scoped by the module that defines the message. Upon receiving a message, the first step in message handling is to verify that the module ID of the receiver in the message header agrees with the ID of the module that is processing that message. This ensures that the message was actually sent to the intended module. Once that is done, the module is free to interpret the message ID value locally. Message ID values are never directly visible to module clients since the client calls a named encoding procedure to build each message. Thus the value and meaning of the message IDs defined by a module is entirely an implementation matter for the module. XDR2OS3 defines an enumeration type that specifies a module's messages as easy to read enumerators. It then uses the position value of a message enumerator as the message ID value.

XDR2OS3 is a work in progress. We intend to ultimately support as much of the XDR standard as we can including, for example, variable length arrays and discriminated unions. The development of XDR2OS3 is guided by our immediate needs with our currently envisioned missions, but we intend to extend and generalize the functionality of XDR2OS3 as the tool matures.

There are other possible encoding and decoding schemes that could have been used. For example, ASN.1 [3] is another standard with approximately similar goals as XDR. However, ASN.1 is much more complicated and entails more overhead both in space and time. ASN.1 includes type information in the encoded message itself, however, which may have advantages for error detection and handling. Since the application developer invokes tool-generated encoding and decoding procedures, and does not directly deal with message encoding, it would be possible to switch the message encoding method without significantly impacting applications. A future version of XDR2OS3 could potentially provide an ASN.1 mode (as one possiblity), perhaps for reasons of error handling or interoperability with legacy systems. This is also an area for future work.

4 Related Work

The most closely related work to CubedOS is NASA's Core Flight Executive [1]. Like CubedOS, cFE endeavors to be a general purpose framework for building flight software. Also like CubedOS, cFE is associated with a collection of modules, called the Core Flight System (CFS), that support common functionality needed by many missions. In addition, cFE makes use of a message passing discipline using a publish/-subscribe model of message handling. CubedOS can provide support for publish/subscribe message handling by way of a library module.

The main difference between the systems, aside from maturity level (cFE is a long established project with a history of actual use), is that CubedOS is written in SPARK and verified free of runtime error. In contrast, cFE is written in C with no particular static verification goals.

The cFE architecture is layered, whereas CubedOS modules operate as peers of each other. The cFE architecture makes use of a separate Operating System Abstraction Layer (OSAL) that presents a common interface across all the platforms supported by cFE. In contrast, CubedOS relies on the Ada runtime system for this purpose, and is thus Ada specific. CubedOS also uses a module library, the CubedOS LLAL, to provide hardware and OS independence in areas not covered by the Ada runtime system and standard library, but the interface to these modules is not yet standardized.

Kubos [4] is a project with roughly similar goals as cFE and CubedOS. It is not as mature as cFE. Like cFE, Kubos is written in C without static verification goals in the sense that we mean here.

Some CubeSat flight software is written directly on top of conventional embedded operating systems such as Linux, VxWorks [5], or RTEMS [6]. These systems allow application software to potentially be written with a variety of tools and methods, although C is most often used in practice. They provide flexibility by imposing few restrictions, but they also don't, by themselves, provide support for common flight software needs. Also they are themselves not statically verified as CubedOS is, although the Wind River VxWorks Cert platform [7] does provide a means by which VxWorks can be used in safety critical avionics applications conforming to the DO-178B standard.

5 Conclusion

CubedOS is an application framework based on message passing that is intended to support the flight software of space missions, particularly CubeSat missions. Our early experience with CubedOS is favorable. The architecture seems to provide an effective way to organize flight software.

Unlike similar projects such as cFE and Kubos, CubedOS is written entirely in SPARK and proved free of runtime errors in the sense meant by SPARK. It is necessary to manully suppress certain SPARK messages related to information flow through the CubedOS mailbox array. However, we feel the danger of doing this is minimal since the easy to understand architecture of the system ensures no flow problems will actually arise in practice.

CubedOS provides a great deal of concurrency and runtime flexibility, but sacrifices some static type safety to achieve

this. We mitigate the danger using a tool, XDR2OS3, that generates message encoding and decoding subprograms based on strongly typed message descriptions. The output of the tool is verified by SPARK.

We intend to release CubedOS to the open source CubeSat community once we have completed an ITAR review of our code base and possibly release both ITAR compliant and U.S non ITAR compliant versions. We anticipate that release to be some time in mid-2018.

References

[1] "Core flight executive." http://opensource.gsfc.nasa.gov/projects/cfe/. Accessed 2017-01-22.

[2] M. Eisler, *RFC-4506: XDR: External Data Representation Standard*. Internet Engineering Task Force, May 2006. http://tools.ietf.org/html/rfc4506.html.

[3] ITU, *Information technology – Abstract Syntax Notation One (ASN.1): Specification of basic notation*. International Telecommunications Union, November 2008. http://www.itu.int/rec/T-REC-X.680/en.

[4] "Kubos." http://www.kubos.co/. Accessed 2017-01-22.

[5] "Vxworks." http://www.windriver.com/products/vxworks/. Accessed 2017-01-22.

[6] "Real time executive for multiprocessor systems." https://www.rtems.org/. Accessed 2017-01-22.

[7] "Wind river vxworks cert platform." http://www.windriver.com/products/product-overviews/vxworks-cert-product-overview/. Accessed 2017-01-22.