

The Dependency Management Framework: A Case Study of the ION CubeSat

Hui Ding Leon Arber Lui Sha Marco Caccamo

Department of Computer Science, University of Illinois at Urbana-Champaign

201 North Goodwin Avenue, Urbana, Illinois 61802

Email: {huiding,larber,lrs,mcaccamo}@cs.uiuc.edu

Abstract

Due to the complexity and requirements of modern real-time systems, multiple teams must often work concurrently and independently to develop the various components of the system. Since a team typically only knows the dependency relations between the components they wrote and those they directly use, keeping track of system-wide dependency relations is not possible for any individual team. To further complicate matters, dependency relations often change as software components are refined or their interactions modified.

Because the robustness of any real-time system hinges on the availability of essential services in spite of faults and failures in useful but non-essential components, keeping track of the constantly evolving dependency relations between the system's components is crucial. If a system's designers cannot ensure that critical services only USE but do not DEPEND ON less critical components, a seemingly minor fault can propagate along complex and unforeseen dependency chains and bring down the entire system. Therefore, automatically tracking and analyzing system-wide dependency relations given only local dependency information is vital for the development of robust real time systems.

This paper presents DMF (Dependency Management Framework), a prototype toolkit for dependency management in designing robust real-time systems. We demonstrate the usability and scalability of DMF with a case study of ION CubeSat, the University of Illinois at Urbana-Champaign's first student-developed satellite.

1. Introduction

1.1. Motivation

In most applications, all features are not equal: some are critical, some are important, some are useful, and some are superfluous. Because of time and budget constraints,

only the most critical features are thoroughly tested and debugged. However, even if the critical components perform flawlessly, complex and unknown dependency relations can still result in system instability. A seemingly minor fault in a non-critical component can cascade along dependency chains and bring down critical systems.

A robust software system is one that guarantees critical component availability and allows for safe utilization of useful, although potentially imperfect, components. In safety critical systems, such as flight control, the certification process mandates the verification of well formed dependencies. That is, critical components will not depend on less critical components. This is typically done by the construction of hardware and software fault trees ([13, 14]) to show that under the given hazard model, faults and failures in less critical components cannot propagate to more critical ones. However, fault trees are an event based logical construction. They are created by manually examining the designs and codes¹. While the cost of manually constructing and updating fault trees is acceptable for slowly changing hardware designs, it is too high for rapidly changing software unless it is safety critical. As a result, the software industry typically does not maintain software fault trees except when mandated by a certification process.

In the context of robust real time systems, we have developed an alternative reasoning framework and prototype toolkit - DMF (*Dependency Management Framework*). This new framework for software component dependency management is tightly integrated with software component development. As long as developers annotate the dependency between their own components and the components they directly use, the framework will generate and evaluate the system wide software component dependency relations from the local annotations. This paper builds on the foundation of our prior work [6], extending it with more expressive dependency specification language, real-time domain support, and discussions on criticality allocation, scalability

¹There are efforts to automatically interpret the UML design or the code and generate fault trees automatically in [14, 16], but the challenge has been formidable.

and evolvability of DMF. We also apply this enhanced dependency tracking framework to ION, the University of Illinois at Urbana-Champaign's first student-developed satellite [1], to evaluate the usability and scalability of DMF. We believe that this component based approach, while is not as fine grained as fault trees, has significant practical value.

1.2. A conceptual framework

To ensure a system's robustness, it is important to make sure that dependencies are well formed and that critical components are protected from the failures of less critical ones. Thus, an effective dependency management toolkit must be able to understand, analyze, and track the properties of component interactions in terms of how failures are contained or propagated. The input to the toolkit will be the annotations of potential residual faults such as packet loss or server crashes and the fault/failure propagation rules.

One may argue, however, that, if the possible faults/failures are already identified as part of the process of identifying the fault propagation rules, then they should be eliminated, rather than allowed to propagate along dependency chains. There are many reasons, though, why, even if a fault is identified, it may not be fixed. First, certain faults cannot be eliminated, such as packet loss in a wireless network. Second, certain residual faults are impractical to remove. For example, consider a reliable application running on an OS that has a resource leaking problem and needs to be auto-rebooted periodically. Fixing the OS would be impractical and out of scope for the developers of the application. Third, it is often too costly to prove the correctness or exhaustively test the codes except when the module is safety critical. Thus, annotating potential faults/failures and tracking their impacts is a much more viable approach in practice. DMF is hence designed to automatically track and analyze system-wide dependency relations given only local dependency information.

DMF consists of two parts: the theoretical framework and the prototype toolkit. The theoretical framework serves as the solid foundation of the prototype toolkit, with formal definitions and theorems. The usage of DMF prototype toolkit consists of two steps: *dependency specification* and *dependency query*. The users annotate the criticality and failure types of the components as well as the fault/failure propagation rules across component boundaries using DMF's *Dependency Specification Language*. DMF will transform the annotations into an internal representation and integrate them with the underlying primitives and composition rules. The users can then perform dependency tracking and reasoning tasks using DMF's *Dependency Query Commands*, which interface with the underlying reasoning engine and derive the system wide dependency relations based on the local annotations. This paper

focuses on the prototype toolkit of DMF and its application to the ION CubeSat.

2. Background

2.1. An informal overview of the formal framework of DMF

In this section, we informally review the formal theoretical foundation of DMF. Interested readers are referred to [6] for formal definitions, theorems, and proofs.

In the rest of this paper, the term *component* can be defined in different layers or with different granularities. For example, it can be defined in the *function/procedure* layer, *class/module* layer, or *process/thread* layer. To make the following presentation clear, when component A synchronously calls the functions of component B, or asynchronously receives data from component B through message passing or shared memory, we say *component A receives the service of component B*, and *component B delivers the service to component A*. Any component in the system is assigned a criticality and its failure set is specified. The *failure set* (formally, *failure semantics*) of component A is the set of all the possible failure behaviors observed by the receivers of the service delivered by component A.

Suppose that component B delivers a service to component A. If any fault of component B results in a fault in component A, we say that component A *totally depends on* component B. If component A can function correctly in spite of all possible faults in component B, we say that component A *USE* component B. Note that the capitalized *USE* is a key word in our formal model. What if component A can only tolerate a subset of component B's faults? We measure the strength of the dependency by the size of the subset of B's faults that A can tolerate. The larger the subset, the weaker is the dependency. The dependency strength between *total dependency* and *USE* is called *partial dependency* with different degrees.

If a critical component A totally or partially depends on a less critical component B, we say that a *dependency inversion* occurs in the system. A system has *well-formed dependencies* if no dependency inversion occurs in the system; otherwise, the system has *ill-formed dependencies*.

Based on these definitions, composition rules for both the high-level dependency relationship (whether A depends on B or not) and the low-level dependency relationship (the quantitative degree of dependency from a fault/failure propagation perspective) have been derived. The definitions and composition theorems are the theoretical foundation of DMF.

2.2. A brief introduction to ION CubeSat

The fast pace of progress in the areas of integrated circuits, microprocessors, and electronics in general in recent years, coupled with unprecedented access to computing, information, and other technological resources, has brought the development of amateur satellites within the reach of ordinary people. The biggest obstacle to satellite development is no longer the technical challenge involved, but rather the prohibitive costs and bureaucracy involved in organizing a launch.

It was this problem that the CubeSat standard, developed jointly by the California Polytechnic State University and Stanford University, sought to solve. The CubeSat standard sets out guidelines and specifications for interfacing an amateur satellite with a standard launcher, provided by CalPoly. This allows satellite developers to focus on building the satellite, leaving all of the logistics and unrelated technical challenges of launching the satellite to someone else. Of all of the CubeSat specifications, the most notable is that which governs the dimensions and mass of the satellite. All CubeSats measure 10cm x 10cm x 10cm and can weigh no more than 1kg, with the option of building double or triple CubeSats (measuring 20cm and 30cm in height and weighing 2kg and 3kg, respectively).

The CubeSat community has over forty universities registered with plans to develop CubeSats with scientific, private, or government payloads. To date, there have been two launches under the auspices of the CubeSat program with two more scheduled to take place in the near future. The University of Illinois at Urbana-Champaign has recently completed development of ION, a double CubeSat, and is anxiously awaiting its launch. ION was built entirely by students, who were fully responsible for all project leadership, design, development, and testing. Over the course of the past four years, over 80 students across 7 engineering disciplines including Electrical, Aerospace, Computer Science, Mechanical, Theoretical and Applied Mechanics, and General Engineering have been involved in the development of ION.

ION's missions include: 1) Measuring molecular oxygen airglow emissions in the Earth's mesosphere using a photomultiplier tube; 2) Performing space testing of Alameda Applied Sciences Corp. micro-vacuum arc thrusters; 3) Performing space testing of Tether Application's Small Integrated Datalogger processor board.; 4) Performing earth imaging using a CMOS camera.; 5) Demonstrating active attitude stabilization on a CubeSat.

To fulfill these mission objectives, ION has an above-average set of system components. Aside from the standard batteries, solar panels, processor, memory, antenna, radio, modem, temperature sensors, and voltage and current sensors, ION also has a CMOS camera, photomulti-

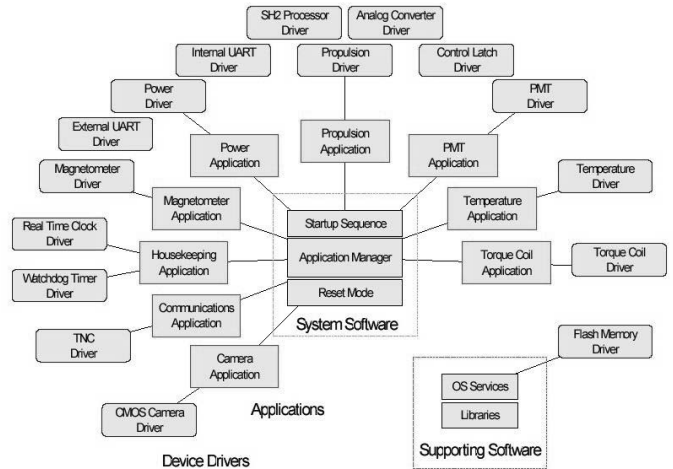


Figure 1. ION CubeSat software system.

plier tube (PMT), thrusters, torque coils, and a 3 axis magnetic field sensor. Due to the complexity of ION's mission objectives, a simple software system was not sufficient to fully control all of the components onboard. As a result, a complete operating system was written almost entirely from scratch, including a system scheduler, filesystem, applications, drivers, and libraries to run ION. A custom communication protocol was also developed for ION that allows for arbitrary scheduling of tasks, configuration of the devices onboard, and downloading of newly created data.

A diagram of the software system, with most of the components and their relationships, is shown in Figure 1. Due to page limitations, it is impossible to fully elaborate on ION's architecture and design issues. Interested readers can find more information from CubeSat's homepage [1] and [5].

3. The DMF architecture and specification language

3.1. The architecture of DMF

In the following sections and Figure 2, we use the term *OS-layer* to denote real-time domain dependency relations, and *APP-layer* to denote application-specific dependency relations. Figure 2 illustrates the DMF architecture. The usage of DMF consists of two steps: *dependency specification* and *dependency query*. First, the users annotate the criticality and failure types of the components as well as the application-specific fault/failure propagation rules using DMF's *Dependency Specification Language*. The application-specific failure propagation rules are of the form $A.fType_A \mapsto B.fType_B$ if condition, where A and B represent system components, $fType_A$ and $fType_B$ represent

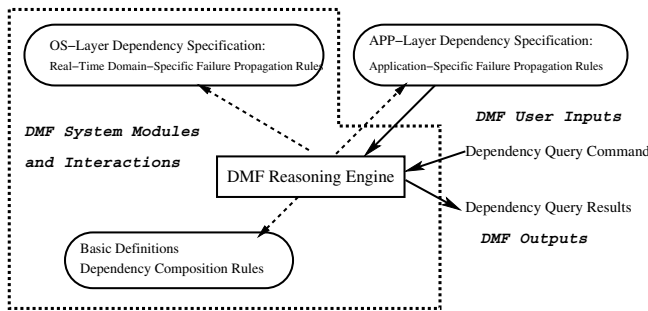


Figure 2. The DMF architecture.

failure types for these components, and *condition* is an optional expression governing under what circumstances the failure propagation can occur.

When loaded into DMF, these APP-layer dependency specifications will be parsed and transformed to an internal representation within DMF. The users can then perform dependency tracking by submitting query commands to DMF. The dependency query command is passed to the DMF reasoning engine, which reasons on both the DMF-provided OS-layer dependency rules and the user-provided APP-layer dependency rules, based on the primitive dependency definitions (e.g., definitions for subset relations between failure semantics and definitions for *total depend*, *partial depend*, and *USE*) and dependency composition theorems (details can be found in [6]). The dependency query result is then shown back to the user in text format².

We will highlight the main features of DMF's dependency specification language in Section 3.3, the OS-layer dependency rules in Section 3.4, and demonstrate the usage of DMF's dependency query commands in Section 4.

3.2. Criticality specification and allocation, and pessimistic annotation assumption

To make sure that dependencies are well formed and that critical components are protected from the faults and failures of less critical ones, the first step is to separate requirements into different criticality levels. There are various methodologies to assign criticalities to different system requirements and software components. For example, the DO-178B standard [2] of U.S. Federal Aviation Administration (FAA) classifies the consequences of a potential software failure to catastrophic, hazardous-severe, major, minor, or no-effect. The criticality of a software component is then identified based on the criticality of the system requirement fulfilled by the component to the overall safety of the system. Another more intuitive classification uses

²It is our future work to generate graphical format query result

four criticality levels: 1) Safety critical, 2) Mission critical, 3) Performance/feature enhancement, and 4) Optional. In the ION CubeSat for example, the highest criticality level would be mission critical, i.e., the satellite is able to communicate with the ground station. The event logging capability, however, is just an optional feature. In DMF, we actually allow more fine-grained criticality levels. The criticality of each component is specified by an integer number. The larger this integer number is, the more critical the corresponding component is. Therefore, it is up to the users of DMF to decide how many criticality levels they want to use when annotating the component criticalities.

Next, we make sure that requirements with different criticality levels are allocated to different protected modules and to make sure that dependency relations are well formed. We also need to track timing, functional and resource sharing dependencies. This does not seem difficult until we look at a real system, either experimental or production. To complicate matters further, we sometimes find that the definition of safety critical depends on the mode of operation. For example, in the ION CubeSat, the bootloader is only critical during system bootup. After the system has been successfully started, it becomes inactive. We also find that sometimes the functionalities of a component will simultaneously belong to multiple criticality levels, which is the case with the current ION CubeSat. This occurs in components that fulfill both critical and non-critical functions. Although being able to assign criticalities at the failure as opposed to component level would help to alleviate this problem, the presence of such a situation really reveals a flaw in the underlying design and a lack of criticality separation.

The FAA's DO-178B standard prohibits criticality mix-ups, and a violation of the principle that requirements of different criticalities should be allocated to different software components would not pass the FAA certification process. For a less critical software system such as ION, however, the users of DMF can temporarily specify the criticality of a software component equal to the highest criticality of the requirements allocated to this component, if requirements of multiple criticality levels are not separated and have been allocated to the same component. But this indicates poor system design and requires architectural changes to make dependency relations well formed. In ION, for example, there is dependency inversion in the current design due to either allocating requirements of multiple criticality levels to the same component or unforeseen global fault propagation paths. With the aid of DMF, developers will be alerted of the potential dependency inversions and all of the possible global fault propagation paths. Based on this information, the developers should consider restructuring the design until the dependency relations are well-formed. As a result of following this process, system robustness is improved incrementally.

There is another point to be mentioned before we move on to DMF's specification language and query commands. DMF has a *Pessimistic Annotation Assumption*: In DMF, we assume that pessimistic fault annotations are used. If a developer cannot verify that a component is correct or if he does not know the impact of a failure of another component providing a service to the current component, he should annotate the most pessimistic potential faults. This style of annotation may result in some false positive warnings to the user, which will be shown in the following sections. However, this also greatly reduces the possibility of false negative warnings. The extent to which this should be done depends on the criticality of the system and the awareness of the users.

3.3. Overview of DMF's dependency specification language

The following sections detail the syntax and features of the dependency specification language, utilizing examples taken from the fault propagation rules for the ION CubeSat.

3.3.1. Components, criticalities, and failure sets. Before one can begin writing fault propagation rules, DMF must have some notion of the components that will be involved, their criticalities, and the failure sets of these components.

In the case of ION, 35 components were identified, along with over 160 unique failure types. As an example, consider the filesystem onboard ION. This component's criticality and failure set is written as:

```
swFS(3): fCreateFile, fReadFile, fWriteFile, fInit, fOpenFile,
fCorruption, sFSFull;
```

This line defines a component called *swFS*, which will represent the filesystem onboard ION. We use the *sw* shorthand throughout to represent a software component. The criticality of *swFS* is 3, with small numbers representing relatively non-critical components and large number representing more critical components. Note that criticality is not absolute and is defined relative to the other components in the system. Furthermore, since the criticality is specified as an integer, there is an effectively infinite number of different criticalities available, allowing for very fine-grained criticality specification.

Following the component definition and its criticality is the failure set of this component. The failures listed are standard for any filesystem, with the *f* prefix being notational shorthand for failure and the *s* prefix notational shorthand for a state that is not a failure in and of itself but which may lead to other failures. For example, the *sFSFull* state (a state representing a full filesystem) is not considered a failure by itself. However, the *sFSFull* state will lead to the *fWriteFile* and *fCreateFile* failures.

3.3.2. Component inheritance. Aside from the basic

component criticality and failure set specifications shown above, DMF also supports the concept of inheritance, in a style similar to that of superclass and subclass of Object-Oriented Programming. This feature is useful since systems often have many components that share a common set of failures. An example of component inheritance is:

```
swPowerApp(5) extends swApplication: fDeadlineMiss, fPowerSampleWrite,
fPowerSampleData, fPowerSampleUpdateBeacon, fAutoPowerExec,
fAutoPowerSetState, fSetPowerStateExec, fSetPowerStateInvalid;
```

This statement defines a new component called *swPowerApp*, which derives from *swApplication*. The supercomponent itself, in this case *swApplication*, is defined just like any other component (the only exception being that the criticality of abstract components must be 0):

```
swApplication(0): fRehash, fInvalidConfigData, fGiveUpCPU,
fStart, fOpenOutputDatafile, fGetWorkUnit, fInitialRehash;
```

As a result of component inheritance, the failure set of *swApplication* will be appended to the failure set of *swPowerApp*. In general, component inheritance works by setting the failure set of component *A* as $failureSet(A) \cup failureSet(superComponent(A))$. What's more, during the dependency tracking process, if the fault propagation rule $A.F_A \mapsto B.?$ is not explicitly specified, then the fault propagation rules $superComponent(A).F_A \mapsto B.?$, $A.F_A \mapsto superComponent(B).?$, and $superComponent(A).F_A \mapsto superComponent(B).?$ are examined one by one.

This OOP class inheritance style allows for expressive and flexible specifications. For example, in ION's dependency specification, all of the high-level components (e.g., *swPowerApp*, *swCameraApp*, *swHousekeepingApp*, *swTorqueApp*, *swTempApp*, *swCommApp*) are subcomponents of the abstract component *swApplication*. This is identical to the way the classes are defined in the source code (e.g., `class PowerApp : public Application`, and `Application` is an abstract class). Since all of the application classes derive from a single abstract class, it is natural for some failures to be identical across all applications, and component inheritance provides a natural way of representing this relationship.

One should note that only abstract components have no criticality of their own (hence the specification of 0 as the component criticality). The instance of an abstract component will inherit the criticality of its subclass. For instance, in the *swPowerApp* example above, the criticality of that instance of *swApplication* will be 5. However, in situations where a component with some criticality C_1 extends another component with non-zero criticality C_2 , a dependency inversion occurs if $C_1 > C_2$.

3.3.3. Boolean expressions of fault propagation rules. After all of the components, criticalities, and failure sets have been annotated, the next step is to define the fault propagation rules. Fault propagation rules specify precisely what the causes and effects of any given failure are. The

most basic fault propagation rules take the following form:

$$swFS.fOpenFile \mapsto swApplication.fRehash;$$

The fault propagation rule states that, if the *swFS* component experiences the *fOpenFile* failure, then the *swApplication* component will experience the *fRehash* failure. Note that this fault propagation rule only lists interactions between neighboring components. The cause of the *fOpenFile* failure and the effect of the *fRehash* failure would be listed in separate fault propagation rules. Furthermore, since *swApplication* is supercomponent of many other components, this rule would result in every subcomponent of *swApplication* also experiencing the *fRehash* failure.

Since most failures result in a complex series of fault propagations, boolean operators are often required to fully describe a fault propagation rule. Boolean operators can be used on both the source (left-hand) and target (right-hand) side of a fault propagation rule. This is an example of a boolean operator being used to specify the effect of a failure:

$$swTNCDriver.fPowerUp \mapsto swCommApp.fPowerUp \wedge swTNCDriver.fReadWrite;$$

This rule states that the *fPowerUp* failure of the *swTNCDriver* component results in the *fPowerUp* failure of the *swCommApp* component **and** the *fReadWrite* failures of the *swTNCDriver* component.

Boolean operator can also be used on the source side of a fault propagation rule, as shown in this example:

$$(swApplication.fGiveUpCPU \wedge swAppManagerStartup.fCPU_WDTInit) \vee swSysInit.fConfigFileInvalid \mapsto swSatellite.fSatelliteReset;$$

This example shows that, if any subcomponent of the *swApplication* component suffers a *fGiveUpCPU* failure **and** the *swAppManagerStartup* component suffers the *fCPU_WDTInit* failure **or** the *swSysInit* component suffers the *fConfigFileInvalid* failure, then the *swSatellite* component will fail with *fSatelliteReset*.

3.3.4. Parameterized failure types and dependency expressions. All of the fault propagation rules so far have listed qualitative relationships between the different failures. However, there are circumstances in which failures have quantitative values associated with them. In the case of ION, for example, the watchdog timers will reset the satellite in 11 minutes if they are not kicked. This sort of quantitative failure can be represented like this:

$$swHousekeepingApp.fKickWDTExec \mapsto swSatellite.fSatelliteReset(11);$$

This fault propagation rule states that, if the *swHousekeepingApp* component has the *fKickWDTExec* failure, then the satellite will reset in 11 minutes.

Here, we call *fSatelliteReset(11)* a parameterized failure type. Generally, parameterized failure types and dependency expressions enable users of DMF to encode application-specific information into the failures and their

propagation rules. For example, the user should not only be able to specify that a component has a *fDeadlineMiss* failure but should also be able to specify an upper limit on the number of consecutive deadline misses the service receiver component can tolerate. This can be denoted as, for example, *fDeadlineMiss(5)*.

Although ION does not make extensive use of this feature, more details on it can be found in [6], whose case study makes extensive use of parametrized failure types and dependency expressions.

3.4. Real-time domain support of DMF

We target the application domain of DMF at real-time systems. In this section, we will illustrate how DMF builds a bridge between the OS-layer dependency relations and the APP-layer dependency relations, as shown in Figure 2.

3.4.1. Process scheduling. We first take schedulability analysis as an example. Schedulability analysis theoretically verifies whether the tasks are schedulable under a certain scheduling method (e.g., RM or EDF [15]). But only the schedulability analysis in theory is not sufficient for the timing considerations in a fault-tolerant system. A question that has to be asked is: *What happens if a certain job overruns its budget (i.e., estimated worst-case execution time)?* Budget overrun (caused by infinite loop, deadlock, denial of service, etc) is highly possible if the execution time of the job is not being monitored by the OS.

We have embedded in DMF the following domain fault propagation rules concerning the budget overrun fault in real-time systems (a smaller *schedPriority* value implies a higher scheduling priority) if no execution-time clock mechanism [9] is implemented in the real-time operating system.

- $A:swThread.fBudgetOverrun \mapsto B:swThread.fDeadlineMiss$
if $execTimeClock(OS) = false \wedge schedPolicy(APP) = SCHED_FIFO \wedge schedPriority(A) \leq schedPriority(B);$
- $A:swThread.fBudgetOverrun \mapsto B:swThread.fDeadlineMiss$
if $execTimeClock(OS) = false \wedge schedPolicy(APP) = SCHED_RR \wedge schedPriority(A) < schedPriority(B);$
- $A:swThread.fBudgetOverrun \mapsto B:swThread.fDeadlineMiss$
if $execTimeClock(OS) = false \wedge schedPolicy(APP) = SCHED_EDF;$

Here, *swThread* is a DMF pre-defined abstract component, and any application-specific component (e.g., *swPowerApp*) corresponding to a run-time thread can be declared as a subcomponent of *swThread* using the **extends** key word. $A:swThread$ and $B:swThread$ represent two variables of component type *swThread*, and will be matched and replaced with any concrete subcomponent of *swThread* during dependency tracking process. *execTimeClock(OS)* is an

OS-layer design feature, and *schedPolicy(APP)* is an APP-layer design feature. And *int schedPriority (swThread A)* is a function corresponding to an application-specific design parameter, i.e., the scheduling priority of a thread *A*.

SCHED_FIFO is a fixed priority preemptive scheduling policy, in which processes with the same priority are treated in first-in-first-out (FIFO) order. *SCHED_RR* is similar to *SCHED_FIFO* but uses a time-sliced (round robin) method to schedule processes with the same priorities. Both of them are standard RT-POSIX specifications [9]. We also include *SCHED_EDF* in case earliest deadline first scheduling is used in the application. The OS design parameter *execTimeClock(OS)* indicates whether the execution time clock mechanism, which is also a RT-POSIX standard, is implemented in the underlying RTOS. Here, *execTimeClock(OS)* is of domain-knowledge and *schedPolicy(APP)* is of application knowledge.

Once the user of DMF specifies the features of the RTOS they are using, the underlying DMF reasoning engine will automatically choose the appropriate *fBudgetOverrun* failure propagation rules which lead to *fDeadlineMiss* failures of the corresponding components (e.g., *fDeadlineMiss* failure of *swPowerApp*, refer to Section 3.3.2). The reasoning process will continue from the *fDeadlineMiss* failure of each affected component based on the fault propagation rules specified in the APP-layer by the user.

Note that the above OS-layer fault propagation rules are natively supported by DMF.

3.4.2. Process synchronization. Another example can be given concerning process synchronization. RT-POSIX defines three basic synchronization protocols: 1) *NO_PRIO_INHERIT*: the priority of the thread does not depend on its ownership of mutexes (a mutex is owned by the thread that locked it). 2) *PRIO_INHERIT*: the thread owning a mutex inherits the priorities of the threads waiting to acquire that mutex. This is the priority inheritance protocol. 3) *PRIO_PROTECT*: when a thread locks a mutex it inherits the priority ceiling of the mutex, which is defined by the application as a mutex attribute. This is also known as the priority ceiling protocol. We have the following domain failure propagation rules embedded in DMF:

- $priorityInversion(A:swThread, B:swThread) = true$
 $if\ synPolicy(OS) = NO_PRIO_INHERIT \wedge$
 $shareMutex(A, B) \neq NULL;$
- $failureSet(A:swThread).add(fLockup)$
 $if\ synPolicy(OS) = PRIO_INHERIT \wedge$
 $shareMutex(A, B) = (M1\ M2) \wedge M1 \neq M2;$

The first rule says that if no priority inheritance protocol is implemented, unbounded priority inversion may occur. The second rule says that if thread *A* and thread *B* share two different mutexes under the basic priority inheritance protocol, deadlock may occur [15], and a *fLockup* failure is

added to their failure sets. The user will be alerted of this newly added failure, if it has not been foreseen and already included in the component's failure set specification as exemplified in Section 3.3.1 and 3.3.2, and should thus annotate how the *fLockup* failure of thread *A* will be propagated to the neighboring components.

3.4.3. Clock resolution. To give another example, suppose the clock resolution for the RTOS is 10 *ms*. A user specifies that the sensing-control loop period of process *A* is 35 *ms*. However, since the system cannot support a clock resolution of 5 *ms*, it would, in reality, result in a loop period of 40 *ms* instead. In such a situation, DMF will warn the user that the actual period for component *A* will be 40 *ms* and will substitute *period(A)* with 40 *ms* wherever it occurs.

In summary, we have embedded the OS-layer domain fault propagation rules in DMF based on the RT-POSIX standards. These general OS-layer fault propagation rules and the specific APP-layer fault propagation rules are integrated, and the DMF reasoning engine will perform dependency tracking on these rules. Due to page limitations, we are not able to give a comprehensive description and explanation of all the OS-layer real-time domain fault propagation rules corresponding to RT-POSIX standard, such as clock resolution, timeout mechanisms, real-time signals, interrupt control, device driver control, inter-process communication and shared resources.

DMF provides OS-layer fault propagation rules constrained by OS design features (e.g., *execTimeClock(OS)*, *synPolicy(OS)*). The users of DMF provide the actual value of the OS features of the RTOS they are using (e.g., *execTimeClock(OS) = false*, *synPolicy(OS) = PRIO_INHERIT*), and the DMF reasoning engine will then choose the corresponding fault propagation rules to apply in the reasoning process.

4. The dependency tracking facilities of DMF

In this section, we demonstrate the dependency tracking facilities of DMF as applied to the ION CubeSat. First of all, it is worth noting that just the effort of annotating the criticality, failure set, and failure propagation rules of a system helps developers to become more aware of the system's failure propagation behaviors and alerts them to failures or propagations that may have been overlooked. Then, with the aid of dependency tracking facilities, DMF will further assist the developers in checking whether the system design has well-formed dependencies. DMF will also help the users to discover the best places to enforce fault tolerance and masking mechanisms to improve the robustness of the system.

In addition, DMF allows developers to examine their system as a whole and determine which components are in-

teracting and which components can result in critical failures. This knowledge can then be used to focus debugging and testing efforts on components in the critical path.

Dependency tracking can be done at different levels. High-level dependency tracking checks whether the system design has well-formed dependencies. Low-level dependency tracking quantifies the nature of the dependency relation from a fault propagation perspective, i.e., to what extent does the correctness of A depend on the correctness of B; or, put another way, what is the set of faults of B that will cause A to become faulty.

4.1. High-level dependency tracking – dependency inversion checking

DMF tracks the high level *total depend/partial depend/USE* relations with the following query commands:

- QUERY: **depUseRelation (Component1, Component2)**
RETURNS: total dependency, partial dependency, or USE relation of *Component1* on *Component2*
- QUERY: **depInversion**
RETURNS: all pairs of components in the system where a dependency inversion occurs.

For example, in ION, a dependency inversion check returns the following results (some have been omitted for the sake of brevity). *TDEP* stands for total dependency and *PDEP* stands for partial dependency. The integer in the brackets is the criticality of the corresponding component.

```
{PDEP : swResetMode[9], swBootloaderHighLevel[8]};
{PDEP : swResetMode[9], swSysInit[7]};
{PDEP : swResetMode[9], swFS[3]};
{PDEP : swPowerApp[5], swFS[3]};
{TDEP : swPowerApp[5], swTime[2]};
{TDEP : swPowerApp[5], swRTCDriver[2]};
{TDEP : swPowerApp[5], swAnalogConverter[4]};
{PDEP : swCameraApp[4], swFS[3]};
{TDEP : swCameraApp[4], swTime[2]};
{TDEP : swCameraApp[4], swRTCDriver[2]};
```

There are a total of over 40 dependency inversion pairs. Among them, one quarter are between high level application modules and low level device drivers, such as $\{TDEP: swPowerApp[5], swRTCDriver[2]\}$. In reality, though, this should be regarded not as a serious dependency inversion but as a criticality specification mistake. The criticality of a device driver should be specified as the maximum of the criticalities of the application modules that receive service from the device driver. When there is a criticality specification mistake, the DMF users should not only correct the criticality annotation mistake but also test and verify the well-formed dependencies once again according to the raised criticality level.

However, more than half of the dependency inversion pairs in ION are between normal application modules and

seemingly unrelated system components, such as the dependency inversion between *swPowerApp* and *swFS*. Intuitively, any failure of the file system should not affect the normal operation of the power management application. So, the cause of this dependency inversion should clearly be investigated, and, if possible, eliminated. In Section 4.2, we discuss the **impact** and **rootCause** dependency queries, which are specifically designed to track down the cause of dependency inversions.

4.2. Low-level dependency tracking – impact analysis and root cause analysis

DMF supports two basic low-level dependency tracking and reasoning queries.

- QUERY: **impact (Component, Failure)** and **impactPP (Component, Failure)**
RETURNS: The list of {Component, Failure} pairs that a failure *Failure* of component *Component* will cause through direct or indirect service delivery. **impactPP** will also show the forward failure propagation path that results in the failures of the resulting components.
- QUERY: **rootCause (Component, Failure)** and **rootCausePP (Component, Failure)**
RETURNS: The list of {Component, Failure} pairs that can result in failure *Failure* of component *Component*. **rootCausePP** will also show the backward failure propagation path that results in the failure of the queried component.

For example, **impactPP(swPowerDriver, fSetState)** returns the following (some results have been omitted for the sake of brevity):

```
{swPowerApp, fAutoPowerSetState}
PATH: {swPowerDriver, fSetState} ↦
      {swPowerApp, fAutoPowerSetState}
{swCameraApp, fRequestHighPower}
PATH: {swPowerDriver, fSetState} ↦
      {swPowerApp, fAutoPowerSetState} ↦
      {swCameraApp, fRequestHighPower}
{swCameraDriver, fCameraOnWithoutHighPower}
PATH: {swPowerDriver, fSetState} ↦
      {swPowerApp, fAutoPowerSetState} ↦
      {swCameraDriver, fCameraOnWithoutHighPower}
... ..
{swSatellite, fSampleData}
PATH: {swPowerDriver, fSetState} ↦
      {swPowerApp, fAutoPowerSetState} ↦
      {swCameraApp, fRequestHighPower} ↦
      {swSatellite, fSampleData}
{swSatellite, fSatelliteReset}
PATH: {swPowerDriver, fSetState} ↦
      {swPowerApp, fAutoPowerSetState} ↦
      {swCameraDriver, fCameraOnWithoutHighPower} ↦
      {swSatellite, fSatelliteReset}
```

There are several other queries based upon the **impactPP(Component1, Failure1)** query. One of these is the **impactAllFailures(Component1, Component2)** query, which returns all failures of Component2 that can result from any failure of Component1. Recall from earlier that there was a dependency inversion between *swPowerApp* and *swFS*. An **impactAllFailures(swFS, swPowerApp)** query can be used to show how this occurred (the intermediate {Component, Failure} pairs along the fault propagation paths have been omitted for the sake of brevity):

```
{swPowerApp, fPowerSampleWrite}
PATH: {swFS, fWriteFile} ↦ ... ↦ ↦
      {swPowerApp, fPowerSampleWrite}
{swPowerApp, fPowerSampleWrite}
PATH: {swFS, sFSFull} ↦ ... ↦ ↦
      {swPowerApp, fPowerSampleWrite}
```

Here, we can see the source of the dependency inversion between *swPowerApp* and *swFS*. However, since the affected failure of *swPowerApp* is an event logging failure (*fPowerSampleWrite* is shorthand for unable to write updated power data to a data file), this failure will most likely not result in any other catastrophic failure. This intuition can be confirmed by the dependency query **impact(swPowerApp, fPowerSampleWrite)**, which reveals that the system level failure affected is {*swSatellite, fSampleData*}, which does not interfere with the normal operation of the satellite. Therefore, we can regard this dependency inversion warning as a false positive.

We now turn our attention to inspecting a dependency inversion that turns out to be far more serious. The query **impactAllFailures(swAnalogConverter, swPowerApp)** returns the following:

```
{swPowerApp, fAutoPowerExec}
PATH: {swAnalogConverter, fSampleData} ↦ ... ↦ ↦
      {swPowerApp, fPowerSampleData} ↦
      {swPowerApp, fAutoPowerExec}
```

Notice that *fAutoPowerExec* is a serious failure, which results in the system potentially operating in an incorrect power state, and one in which high power requests from certain critical devices may be denied. The effect can be revealed by the query **impact(swPowerApp, fAutoPowerExec)**, which returns, among other things, {*swSatellite, fSatelliteDead*}. Therefore, we can conclude that this is a very serious dependency inversion which must be removed. From the failure propagation path, we see that the best place to cut off the failure propagation is prevent the *fSampleData* failure of the *swAnalogConverter* from being propagated to the *fPowerSampleData* failure of *swPowerApp*. One way to reduce the likelihood of this happening is to add redundancy to the analog converter by duplicating hardware.

In a similar fashion, the users of DMF can issue the **rootCausePP(swSatellite, fSatelliteDead)** query to reveal the reverse failure propagation paths which lead to a failure of the satellite.

Given the brief excerpts of propagation rules and queries shown here, it may seem that DMF is only able to state the obvious, in terms of fault propagations. For example, without writing a single propagation rule, all of the ION developers would be able to conclude that failures of the file system or power subsystem could lead to the failure of the satellite. However, aside from the insight gained by merely writing out all of the propagation rules for a system, the true value of the various DMF queries lies in unearthing the minor, often forgotten components that can lead to a system failure. Although the vast majority of failures that the DMF queries unearthed were indeed obvious, the real value of the queries lies in finding the small minority of overlooked components such as message queues that no one would have expected to lead to a catastrophic system failure.

To summarize, by combining the various query commands, the users of DMF can get a clear understanding of the failure propagation behavior of the target system, be alerted to potential dependency inversions, and reveal more detailed information through impact and root cause analysis. They can then use these results to improve the robustness of the system design by enforcing fault tolerance mechanisms. The results of the ION case study are already being applied to aid the design of the second generation ION satellite, which is currently under development.

4.3. A brief discussion on the scalability and evolvability of DMF

Due to the complexity and requirements of modern real-time systems, multiple teams must often work concurrently and independently to develop the various components of the system. Since a team typically only knows the dependency relations between the components they wrote and those they directly use, keeping track of system-wide dependency relations is not possible for any individual team. Therefore, system-wide composition of the local dependency annotations must be supported in order to guarantee the scalability of DMF. To further complicate matters, dependency relations often change as software components are refined or their interactions modified.

The formally derived and implemented dependency composition rules of DMF enable its scalability. Only the failure propagations between immediately interacting components need to be specified. Furthermore, only failure propagations regarding the conjunctive composition of basic failure types are required. The global system-wide failure propagation properties and dependency relations can be composed and evaluated from the local annotations based on the fault/failure propagation composition theorems and depend/USE relation composition theorems [6].

When a new component is added to the system, the DMF user only needs to specify this component's criticality, fail-

ure set, and fault propagation rules concerning components immediately interacting with the new component. When a component is removed from the system, the DMF user simply removes the declaration for this component and its criticality. When a component is updated or replaced, the user only needs to update or replace the corresponding criticality, failure set, and fault propagation rules concerning immediately interacting components. In a word, when the system design evolves, the scope of the re-specification is local and isolated from the other components which are not immediately interacting with the updated component. In this way, the evolvability of the dependency specification is guaranteed, and DMF does not create a burden on its users.

5. Related works

While the dependency information acquisition and identification problem is tackled in [4], [7], [11], etc, other research efforts, such as [8], [10], and [12], attempt to cope with the dependency management problem by providing a middleware or software architecture. We emphasize our work in the fault/failure propagation aspect, and the *dependency* in these works can be categorized as high-level dependency within our framework.

AADL [3] has an Error Model Annex that extends the core language to support reliability modeling. Compared to AADL, DMF has several distinctive features. DMF supports parameterized failure semantics, thus significantly enriching the fault model. DMF also allows users to perform both high-level (*total depend/partial depend/USE*), low-level (fault/failure propagation) dependency tracking, and both forward impact analysis and backward root cause analysis with respect to low-level dependency tracking. In this sense, DMF provides a stronger dependency analysis capability than AADL.

6. Conclusion and future work

In this paper, we have presented the *Dependency Management Framework*, a prototype toolkit dedicated to dependency tracking and reasoning in robust real-time systems, as well as a case study of the ION CubeSat, which demonstrates the use of many of these features. DMF emphasizes the expressiveness and simplicity of the specification language syntax as well as the scalability and evolvability of the dependency tracking facilities to help users improve the robustness of their system designs.

At this point, the dependency tracking procedure depends on the correct annotation of fault propagation relations among components by developers. However, at least some of these relations could be derived automatically.

In the future, a more friendly (graphical) user interface should be designed, to be used for both entering and dis-

playing fault propagation paths. More case studies are also needed to further evaluate the usability and scalability of the toolkit.

Acknowledgment

This research is sponsored in part by NSF CCR 02-09202, by ONR N00014-02-1-0102, and by MURI N00014-01-0576.

References

- [1] <http://cubesat.ece.uiuc.edu/>.
- [2] <http://www.stsc.hill.af.mil/crosstalk/1998/10/schad.asp>.
- [3] <http://www.aadl.info/>.
- [4] A. Brown, G. Kar, and A. Keller. An active approach to characterizing dynamic dependencies for problem determination in a distributed environment. In *Proceedings of the 7th IFIP/IEEE International Symposium on Integrated Network Management*, Seattle, WA, USA, May 2001.
- [5] M. Dabrowski. The design of a software system for a small space satellite. Master's thesis, UIUC, 2005. http://www.interave.net/projects/ion_thesis_2005/.
- [6] H. Ding and L. Sha. Dependency algebra - a tool for designing robust real-time systems. In *Proceedings of the 26th IEEE Real-Time Systems Symposium (RTSS'05)*, Miami, Florida, USA, December 2005.
- [7] C. Ensel. Automated generation of dependency models for service management. In *Proceedings of Workshop of the OpenView University Association (OVUA'99)*, June 1999.
- [8] C. Ensel and A. Keller. Managing application service dependencies with xml and the resource description framework. In *Proceedings of the 7th International IFIP/IEEE Symposium on Integrated Management (IM 2001)*, May 2001.
- [9] M. G. Harbour. Real-time posix: An overview.
- [10] P. Hasselmeyer. Managing dynamic service dependencies. In *Proceedings of the 12th International Workshop on Distributed Systems: Operations and Management - DSOM 2001*, Nancy, France, October 2001.
- [11] G. Kar, A. Keller, and S. Calo. Managing application services over service provider networks: Architecture and dependency analysis. In *Proceedings of the 7th IEEE/IFIP Network Operations and Management Symposium (NOMS 2000)*, pages 61–75, Honolulu, HI, USA, April 2000.
- [12] F. Kon and R. H. Campbell. Dependence management in component-based distributed systems. *IEEE Concurrency*, 8(1):26–36, January 2000.
- [13] N. G. Leveson. *Safeware: System Safety and Computers*. Addison-Wesley, 1995.
- [14] N. G. Leveson, S. S. Cha, and T. J. Shimeall. Safety verification of ada programs using software fault trees. *IEEE Software*, 8(4):48–59, July 1991.
- [15] J. W.-S. Liu. *Real-Time Systems*. Prentice Hall, 2000.
- [16] D. Wallace, M. Towhidnejad, and C. Coleman. Software fault tree analysis. Technical report, Software Assurance Technology Center, NASA, Decemeber 2003.