DEPENDENCY MANAGEMENT FRAMEWORK – A TOOL FOR DESIGNING ROBUST REAL-TIME SYSTEMS

BY

HUI DING

B.E., Tsinghua University, 1998
M.E., Chinese Academy of Sciences, 2001

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2006

Urbana, Illinois

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 thesis presents *Dependency Management Framework* (DMF), a unified theoretical framework and prototype toolkit for dependency management in robust real-time systems. To illustrate the usability and scalability of DMF, we also give several application examples of DMF, with case studies on a distributed car control testbed and a student-developed ION CubeSat satellite.

Acknowledgments

The completion of this dissertation and the underlying body of research over the period of five years would not have been possible without the advice and cooperation of a number of people. The most important contribution has undoubtedly been due to my adviser, Professor Lui R. Sha. The amount of time and effort that Prof. Sha has put into discussions involving everything from the smallest technical details to the big picture, has been truly extraordinary. I am also grateful to Prof. Marco Caccamo, Prof. Carl Gunter, and Prof. Grigore Rosu for serving on my thesis committee and for their kind comments.

A number of my fellow students and colleagues have been instrumental in making fruitful the last five years of my life, academically and otherwise. They are Leon Arber, Tanya L. Crenshaw, Sathish Gopalakrishnan, Sumant Kowshik, Kihwal Lee, Xue Liu, Mu Sun, Min Wu, Ajay Tirumala, Qixin Wang.

And finally, thanks to my wife, Xia Li, who endured this long process with me, always offering support and love.

Table of Contents

List of Tables

List of Figures

Chapter 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. Giving the existing technologies, industry can only afford to make critical features highly reliable. Complex and unknown dependency relations are a key contributor to software system instability. That is, a seemingly minor fault in a non-critical service can cascade along dependency chains and bring down the whole system.

There are some high-profile real-world examples where the system failures have been related to ill-formed dependencies. One of the most famous examples is in the field of avionics. On 4 June 1996, the maiden flight of the Ariane 5 launcher ended in a failure. Only about 40 seconds after initiation of the flight sequence, at an altitude of about 3700 m, the launcher veered off its flight path, broke up and exploded. Engineers from the Ariane 5 project teams of CNES and Industry immediately started to investigate the failure. The most astonishing investigation result is that *the root cause fault is within a reused Ariane 4 software component that is even not required in Ariane 5* [1]. Ariane 5 satellite had reused some software components developed for Ariane 4 for the inertial reference system. These Ariane 4 software components made the following assumption: the horizontal velocity component will never overflow a 16-bit variable. This was true for Ariane 4, but not for Ariane 5. This triggered self-destruction roughly 40 seconds after the launch of Ariane 5. A fault in a non-critical, and actually even not required, software component had cascaded along dependency chains and brought down the whole system. If the Ariane 5 system had well-formed dependency, i.e., critical components had not depended upon less critical components such as the reused Ariane 4 software component, this catastrophic failure would not have ever happened.

A robust software system is one that guarantees critical system properties and allows safe exploitation of imperfect but useful components. In safety critical systems such as flight control, the certification process mandates the verification of well formed dependency. That is, critical services will not depend on less critical services. This is typically done by the construction of hardware and software fault trees ([36–38]) to show that under the giving hazard model, faults and failures in less critical components cannot propagate to more critical ones. However, fault trees are event based logical construction. They are created by manually examining the designs and the codes [1]. 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, 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 present an alternative reasoning framework and prototype toolkit to fault trees. This new framework for software component dependency management is tightly integrated with software components development. Since the formal verification or exhaustive testing are not feasible for many large and complex software components, the potential failures of similar components in the past must be annotated and the effects to the rest of the system analyzed [2]. As long as developers annotate the dependency between their own components and the components they directly use, our system will generate the system wide software component dependency relations from local annotations and capture the impact of design changes. This allows rapid analysis and comparison of different designs and modification from the perspective of robust software designs.

1.2 A conceptual framework

Because of the limitations of what is available to us, we will introduce our basic concepts and approach using an accessible system, i.e., a vision-based distributed car control testbed [28], which is an experimental real-time control system in Prof. Panganamala R. Kumar's Information Technology Con-

[1] There are efforts to automatically interpret the UML design or the code and generate fault trees automatically in [38, 47]. But the challenge has been formidable.

[2] This is called *Pessimistic Annotation Assumption*. See Section 2.2.

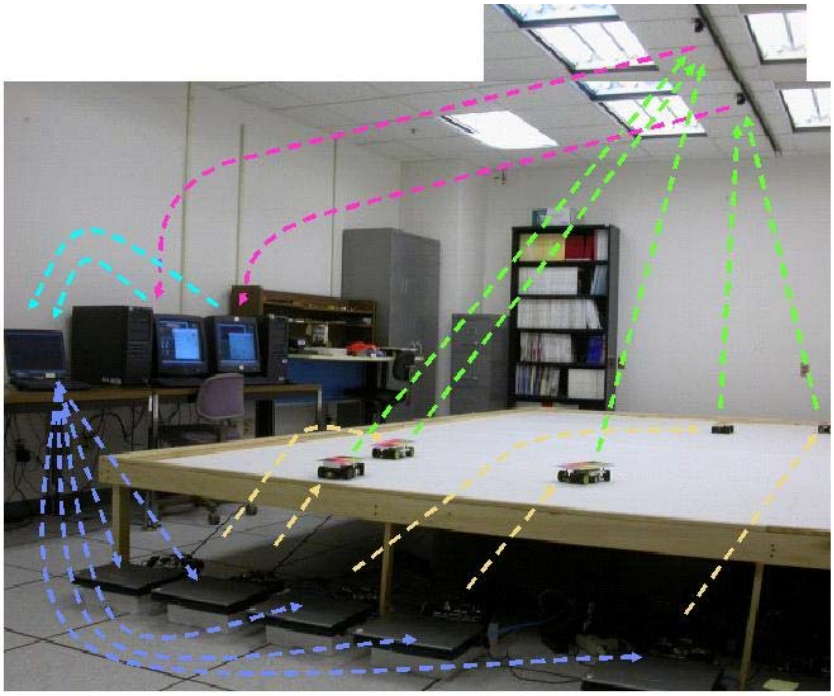


Figure 1.1: A Vision-Based Car Control Testbed

vergence Laboratory of University of Illinois at Urbana-Champaign.

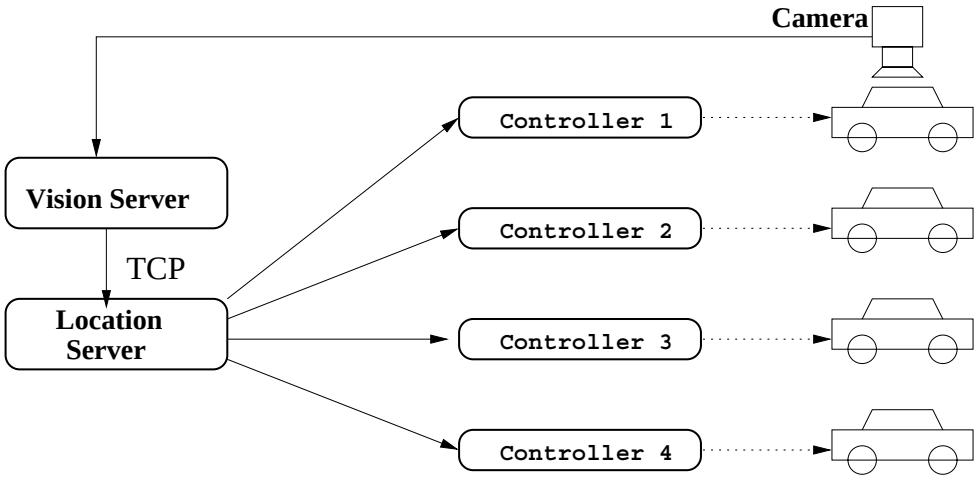
As illustrated in Figure 1.1, the testbed has a fleet of autonomously controlled cars following given trajectories. The cars do not have location sensors. Instead, the vision server periodically computes the locations of the cars using the image of the cars taken by the ceiling-mounted cameras and then sends them to the location server via Ethernet. The location server then broadcasts the location information of each car over a 802.11 network, so that each autonomously controlled car can use it to correct its trajectory following errors. Control is a periodic real-time activity and each controller expects periodic updates on the car locations.

Normally, navigation control of a car assumes periodic updates of the cars' location information. However, 802.11 network (and Ethernet) cannot guarantee the timely arrival of packets. The incompatibility between the aperiodic delivery model of 802.11 network and the periodic control of cars can be significantly lessened by adding a Kalman filter at each car as illustrated by Figure 1.2(b). The Kalman filter can generate acceptably accurate periodic location estimations for the controller, as long as there is at least one successful location update in T seconds. Let p be the desired period of

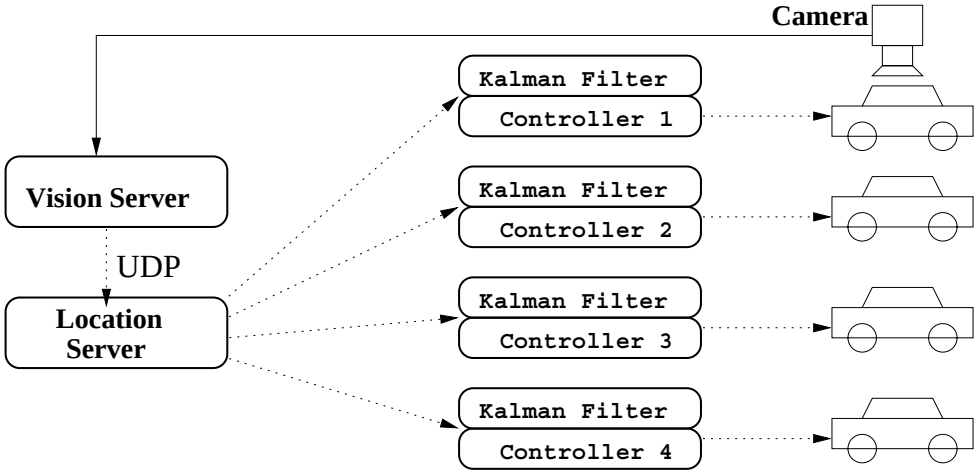
location packet delivery to the car controllers. A car can keep going without having to stop as long as there are no more than k consecutive packet loss, where $k = \lfloor T/p \rfloor$. This example illustrates that the car control *fail-safe stop* parameter k is a function of the location packet update period p and the Kalman filter delay tolerance T. Fail-safe stop parameter k must be updated if either location packet update period p or Kalman filter delay tolerance T changes. And delay tolerance T must be updated if the dynamics of the electro-mechanical system of the car is changed or the worst case road condition is changed. In short, automatically tracking and propagating the impact of changes in either hardware, software or environmental conditions are important in robust real-time systems.

We now argue for the importance of tracking fault mappings. From a criticality perspective, the most critical feature of the car control testbed is collision avoidance. As long as the the distances between cars are larger than the uncertainty in cars' locations, there will be no collision. Car controller will stop the car if the uncertainty of cars' locations grows too large (we call this *fail-safe stop*). This can be triggered by missing k location packets in a row. From this perspective, it is permissible for the location server to have a crash failure as long as it can be restarted within T seconds, but not location information errors that can lead to car collision. Thus, if the vision server gets a bad image, it should skip an update to the location server, and consequently the location server skips an update to the cars, mapping a potentially serious location-error fault into a tolerable missing-a-packet fault. Tracking the fault mapping is another important task in dependency management.

We now turn our attention to the impact of communication protocols. In a distributed system, it is important to model the failure semantics of communication protocols. For example, crash and recovery of the vision server or location server is acceptable by the design. In fact, in the testbed the upper bound of delay tolerable by Kalman filter is about 7 seconds. There is plenty of time for a vision server or location server process to restart from a failure while the cars keep going. However, if TCP is used for the communication between vision server and location server, which is the case of the original design as shown in Figure 1.2(a), the crash and then recovery of the vision server will lead to the lockup of location server, due to the loss of previous TCP handlers unless they are checkpointed. This argues the use



(a) Original Design using TCP



(b) Alternative Design using UDP and Kalman Filter

Figure 1.2: The Design Changes of the Car Control Testbed

of UDP, since TCP semantics offers no advantage except more complexity and overhead. The design change concerning the TCP/UDP connection and Kalman filter is shown in Figure 1.2.

We now argue for the necessity of a formal theoretical framework for dependency management. First, informal notions can lead to confusion and even errors. For example, some papers simply refer *A depends on B* as *A calls B*. However, *A depends on B* should actually mean that the correctness of A depends on the correctness of B, as pointed out by Cristian [16]. Under this meaning, *A calls B* does not necessarily imply *A depends on B*, e.g., if all the failure of B can be tolerated by A, then the correctness of A does not depend on the correctness of B. As another example, most dependency management methods do not discriminate the different degrees of dependencies between components. For example, Cristian [16] defined: A server A depends on a server B if the correctness of A's behavior depends on the correctness of B's behavior. Some researchers noticed the necessity of discriminating the dependency strengths. Keller et al. [33] introduced the concept of dependency strength as how strongly the dependent component depends on the antecedent resource. They classify dependency strength into *none*, *optional*, and *mandatory*. However, they did not provide a further explanation and formally defined metric of dependency strength, and such an informal classification of dependency strength did not totally clarify the confusion. For example, in the car control testbed, under the classification of [33], the dependency strengths of car controller on location server are *mandatory* in both the original and enhanced design. However, we know that actually the dependency strength is significantly weakened in the enhanced design. In the original design, the car controller requires strict periodic updates on the car location from the location server. Thus, any failure in the location server or in the communication from the sever to the car controller could lead to the failure of the controller. In the enhanced design, a Kalman filter was introduced and this allowed the car controller to be less dependent on the correctness of the location server, as explained earlier. Therefore, a formal theoretical framework is definitely needed in order to clarify the confusions and eliminate the possible errors introduced by such kind of informal notions.

What's more, any dependency management toolkit or middleware should have the capability to compose system wide dependency relations from local dependency information or annotations. Dependency composition rules are

thus in need. This calls for a unified theoretical framework where dependency and dependency strength are formally defined and dependency composition rules are formally derived and proved. The formal framework should serve as a solid theoretical foundation for the dependency management toolkit or middleware.

In summary, an effective dependency management *theory and toolkit* is desired to sufficiently address the issues we have identified. The input to the toolkit will be the annotations of potential residual faults such as packet loss and server crashes and the fault mappings. However, one may argue that: If I already know the possible faults/failures, why not eliminate them? First, certain faults cannot be eliminated, for example, packet loss in a wireless network. Second, certain residual faults are impractical to remove. For example, in the location server example, the application logic is reasonably simple and can be made reliable. But the OS has a resource leaking problem and needs to be auto-rebooted periodically. Fixing the OS would be just too large a task for the developers of the car control testbed. Besides, the crash failure is not safety critical and is recoverable due to the use of Kalman filter and fail-safe stop protocol. Third, in practice, it is too costly to prove the correctness or exhaustively test the codes except when the module is safety critical. Thus, annotating the potential residual errors and tracking their impacts is a more viable approach in practice.

1.3 Research challenges

This thesis research 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.

First of all, we claim that just the effort of annotating the criticality, failure set, and failure propagation rules of a system helps developers to be more aware of the system's failure propagation behaviors and be alerted to some failures or propagations overlooked before. Then, with the aid of DMF's *Dependency Query Commands*, 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 optimal 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 interacting 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.

But doing this is not without its own challenges. There are three main research challenges with respect to dependency management: ***formalism***, ***evolvability*** and ***scalability***. The first challenge is a theoretical framework for dependency management, as argued in last section. Currently there is a lack of formal theoretical framework for dependency management that is closely bound with software components. Specifically, we need a formal definition to measure the dependency strength of one component on the other component. Based on the formal definition, we should be able to formally define both the high-level dependency relationship (A depends on B or not) and low-level dependency relationship (fault/failure propagation properties) in a unified theoretical framework, and be able to derive their composition rules based on the definitions. We deal with this challenge by providing a theoretical framework for dependency management, i.e., Definition 1-11 and Theorem 1-6, in Chapter 2.

The second challenge is how to make the prototype toolkit adaptable to evolving software designs. It is important to note that a tighter binding between the fault/failure propagation description and the actual system design is desirable for software systems whose design changes frequently. When changes are infrequent, e.g., a hardware system, the fault tree analysis is very effective. However in constantly changing software systems, lack of explicit binding between the tree structure and the actual system design makes it hard to keep track design changes. Figure 1.3 shows an example of simplified software fault tree representation of `location server crashes` event for

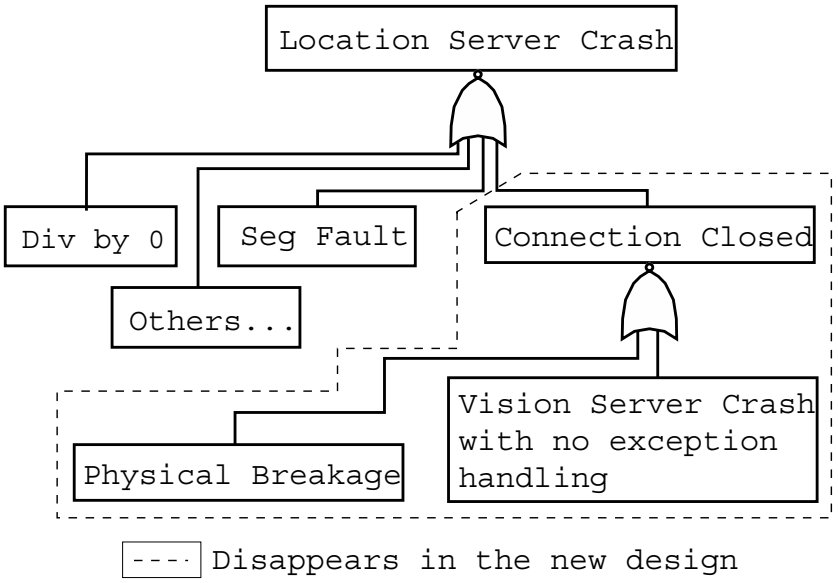


Figure 1.3: The Fault Tree for *Location Server Crashes*

the two designs. When the communication between the vision server and the location server is implemented using the TCP, a crash of the vision server process can cause the location server process to lock up even if the vision server has been recovered. However, using an alternative protocol such as UDP, the location server process need neither be restarted nor reestablish the connection. When the component interaction protocol is changed, the change in failure behavior must be understood and reflected to the fault tree manually. If we have multiple fault trees representing different top events, each of the tree may need to be modified in a different way. One of our objectives is to keep track of the dependency relations as a function of the annotated properties of components and their interaction protocols. We deal with this challenge in Chapter 4, and perform a detailed comparison with software fault tree analysis ([37, 38]) in Chapter 6.

The third challenge is the scalability of the prototype toolkit. Modern real-time systems are often developed concurrently by multiple teams. Each development team typically only knows and thus is able to annotate the potential failures and failure propagation properties between the software components within its own scope. They are also required to understand and annotate how the failures of the other teams' components that directly interact with their own components can affect their own components. But

they are not supposed to know the details of the fault/failure propagations of components exclusively developed by the other teams. Therefore, hierarchical system-wide composition of the local dependency annotations must be supported in order to guarantee the scalability of our prototype toolkit. This not only demands hierarchical composition rules in theory, but also demands the capability of the toolkit to generate system wide dependency relations from the local annotations of each development team. We deal with this challenge with Theorem 1-6 in Chapter 2 and briefly discuss the scalability of our prototype toolkit in Section 5.2.

1.4 Structure of this document

In the following of this thesis, we first present the theoretical framework of our Dependency Management Framework (DMF) in Chapter 2. We then give a brief overview and a few application examples of the DMF prototype toolkit in Chapter 3 and Chapter 4, covering the basic features and advanced features of DMF separately. We perform discussions on several important topics, such as criticality specification, scalability, and evolvability in Chapter 5. We perform a detailed comparison with the related works in the area of dependency management and fault analysis in Chapter 6. We then briefly summarize our current accomplishments and set up the future research directions in Chapter 7.

Readers who want to see the application examples of DMF early can skip Chapter 2 and read Chapter 3. After the readers grasp a general idea of the basic features of DMF, they can either come back to the theory in Chapter 2 or continue with the advanced features of DMF in Chapter 4. It is not a prerequisite to fully understand the formal definitions and theorems presented in Chapter 2 in order to try out the basic features of DMF.

Chapter 2

The Theoretical Framework of DMF

In this chapter, we present the theoretical framework of *Dependency Management Framework*. We first tailor and extend the failure semantics model developed for general purpose systems to the failure containment and propagation needs of robust real-time systems in Section 2.1, with the most important extension being from *unparameterized* failure semantics to *parameterized* failure semantics. We then come up with the formal definitions of failure semantics mapping and dependency strength, and classify dependency relations into *total dependency*, *partial dependency*, and *USE* in Section 2.2. The composition rules of dependency relations based on these definitions have been derived for low level dependency tracking (fault/failure propagation between components) in Section 2.3 and high level dependency tracking (*depend/USE* relations between components) in Section 2.4. Our objective is to provide a unified theoretical framework for dependency management in robust real-time systems. The definitions and theorems serve as the theoretical foundation of the DMF toolkit to be presented in Chapter 3 and Chapter 4.

In our framework, 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. The dependency analysis and reasoning can be performed hierarchically from the components of lower layers to components of higher layers to guarantee the scalability of the toolkit. In the logical domain, at the finest granularity, we can first investigate the fault/failure propagation properties between *functions* that are used in one *functionality module* (e.g., *class* in object-oriented programming), and derive the failure set of this specific *functionality module* based on the composition/interaction of these *functions*. Then in the execution domain, we can investigate the fault/failure propagation properties between the *functionality modules* that are used in one *process/thread*,

and derive the failure set of each execution unit, i.e., *thread*. Finally, we can investigate the fault/failure propagation properties across *process/thread* boundaries, and reason about the robustness of the whole *system*. In this hierarchical composition process, *functions/classes* and *processes/threads* are defined as component at different stage.

In this chapter, we illustrate the definitions and theorems with our motivating example, i.e., the car control testbed, and investigate the failures and failure propagation properties in the *process* layer. In Chapter 4, we will refine the component granularity and come up with the case study in detail.

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 service to component A*.

2.1 Parameterized failure semantics

In order to understand and efficiently deal with failures, failures must be categorized into failure classes. Cristian offers a failure classification that includes omission, timing, performance, value, and crash [17]. We adopt his classification and tailor it to better suit for robust real-time systems where timing is a critical consideration.

The definition of failure semantics in [16] is: *If the specification of a server s prescribes that the failure behaviors likely to be observed by the users of s should be in failure class X, it is said that s has failure semantics X.* We argue that in order to compare and differentiate the effect of the same failure in different context, it is very important for failure semantics to capture more detailed failure conditions. We extend Cristian's definition of failure semantics so that application-specific information can be encoded into the specification of failure semantics.

For example, in hard real-time systems, *omission failure* (no service delivery) and *timing failure* (service delivery misses its deadline) have the same effect for the service receiver component and we will treat them in the same failure class of *DeadlineMiss*. For example, in the distributed car control testbed, if the car controller cannot receive location packet from the location

server by its periodic deadline, there is a *DeadlineMiss* failure of the location server. It does not matter whether the packet is lost during transmission and will never arrive (omission) or the packet is delayed and will arrive later (miss deadline).

However, the binary fail/no-fail notion of *DeadlineMiss* does not fully capture the properties of robust real-time systems. As shown in Figure 1.2, a controller in (a) may fail due to one location packet deadline miss, while another one from (b) may only exhibit acceptable performance degradation even with several consecutive location packet deadline misses. This implies the system needs different kinds of remedies depending on how the system is actually affected. Therefore, we further parameterize the *DeadlineMiss* failure semantics to encode this information. We denote n consecutive deadline misses as $DeadlineMiss\langle n \rangle$.

The *performance failures* are highly application specific. For example, in the car control system, the optimal performance is that the cars strictly follow the desired trajectories, and no collision occurs. When there is big deviation between the actual trajectory and desired trajectory, there is a performance degradation. If there are consecutive deadline misses of the location packet, the cars may have to stop moving in order to avoid collision since no location information of the peer cars are available now. The performance is further degraded or we say there is a control performance failure. More seriously, if car collision occurs, there is a catastrophic control failure. Therefore, control performance degradation or failure are application specific and can also be encoded with further application-specific information. For example $TrajectoryError\langle \Delta \rangle$ denotes the accumulated deviation Δ between actual and desired trajectories of the cars.

The *value failures* are also application specific. For example, in the car control testbed, if there is deviation between the location information contained in the location packet and the cars' actual locations, it is a value error. $LocationError\langle u \rangle$ denotes a value error u in the location packet, while $LocationError\langle u, v \rangle$ denotes any location value error smaller than v and larger than u. In our framework, we still reserve *Value* as a key word for general value failure when no application-specific information is available or necessary.

As presented in the *Recovery-Oriented Computing* [43], we regard restart as a corrective action, rather than an effect. Thus we do not distinguish

crash failures based on the recovery semantics (e.g. amnesia-crash, partial-amnesia-crash, etc).

Another important failure class in real-time systems is *resource sharing failure*. Here resource includes CPU, memory, and other resources that are shared by multiple components. For example, if the location server overruns its planned real-time budget (worst-case execution time, WCET) but delivers its packet as specified by the deadline, there is no corresponding failure type in the traditional model [17]. But from the real-time computing perspective, resource consumption failure is an important issue because it can adversely affect the timing of other tasks sharing the same computing resource or communication channels.

In a word, we provide sufficient support for the general failure semantics, as well as failure semantics typical in robust real-time systems. Both the user-defined and system-defined failure semantics can be further parameterized with application-specific information during the dependency specification and tracking process.

Definition 1: *The failure semantics of component A is the set of all the observable failure behaviors of the service delivered by component A, denoted as* FS_A.

For example, suppose the location server of the distributed car control testbed has failure semantics:

$FS_{LS} = \{DeadlineMiss\langle 5\rangle\} \bigcup \{LocationError\langle 0, 20\rangle\} \bigcup \{Crash\}$.

It denotes that the location server may fail to timely broadcast periodic location information to the car controllers for up to five consecutive periods, or broadcast the wrong location information deviated from the actual car location for up to 20 distance units. The location server may crash, too.

The failure semantics of the composition of several components is defined as:

Definition 2: *The composition of components* $A_1, A_2, ..., A_N$ *is denoted as* $A_1 \bigoplus A_2 \bigoplus ... \bigoplus A_N$. *Its failure semantics is the Cartesian product* $FS_{A_1} \times FS_{A_2} \times ... \times FS_{A_N}$.

We manipulate dependency tracking and reasoning from the perspective of failure semantics mapping and propagation. The measure of dependency strength is thus based on the measure of failure semantics strength. The strength of failure semantics is defined in [17] as: The A/B failure semantics is weaker than A because A/B allows more failure behaviors than A.

Conversely, A is a stronger failure semantics than A/B. Presented in a more formal way, the relative strength of failure semantics can be defined using the inclusion relation between sets, e.g, $\{A\} \subset \{A, B\}$. This idea is extended in our parameterized failure semantics.

Definition 3: *Given two sets of failure semantics FS and $FS^{'}$, if $FS \subset FS^{'}$, we say that FS is stronger than $FS^{'}$, and conversely $FS^{'}$ is weaker than FS.*

Note that our definition is a natural extension of Cristian's definition which is based upon unparameterized failure semantics. Actually, if we do not encode application-dependent failure information into our specification of failure semantics, i.e., we only consider the general failure classes of components, then we can represent any specific failure semantics with a singleton set, e.g., $\{Value\}$, $\{Timing\}$, $\{Crash\}$, thus we can have Cristian's definition of failure semantics strength.

Our failure semantics strength definition greatly extends Cristian's work. Say $FS = \{DeadlineMiss\langle 3\rangle\} \bigcup \{Crash\}$, $FS^{'} = \{DeadlineMiss\langle 5\rangle\} \bigcup \{Crash\}$. FS is a stronger failure semantics than $FS^{'}$, because consecutive *DeadlineMiss* failure for up to 3 periods is a subset of consecutive *DeadlineMiss* failure for up to 5 periods. It is harder to ensure the former than the latter. Cristian's approach did not differentiate the strengths of these two failure semantics. The reason while ours can differentiate them is that we support the specification and comparison of *parameterized* failure semantics.

It should be pointed out that not all failure semantics pairs can be compared in terms of strength. For example, the two failure semantics $FS = \{DeadlineMiss\langle 5\rangle\}$, $FS^{'} = \{Crash\}$. In order to compare the strengths of these two failure semantics, application-specific weight should be assigned to each failure class to measure its potential impact on the robustness of the whole system. The further discussion is out of the scope of this thesis.

2.2 Dependency strength and classification of dependency relations

In this section, we only consider pairwise component dependency, i.e., only two components A and B are considered, and A receives the service directly delivered by B. In Section 2.3 and Section 2.4, we will go beyond pairwise

component dependency and derive dependency composition rules for various component interaction patterns.

Most dependency management methods do not discriminate the different degrees of dependencies between components. For example, Cristian [16] defined: *A server A depends on a server B if the correctness of A's behavior depends on the correctness of B's behavior.* Some researchers noticed the necessity of discriminating the dependency strengths. Keller et al. [33] introduced the concept of *dependency strength* as how strongly the dependent component depends on the antecedent resource. They classify dependency strength into *none*, *optional*, and *mandatory*. However, they do not provide a further explanation and formally defined metric of dependency strength, and such an informal classification of dependency strength was not be effectively useful in evaluating the dependency relations of different designs. For example, in the car control testbed, under the classification of [33], the dependency strengths of car controller on location server are *mandatory* in both the original and enhanced design. However, we know that actually the dependency strength is significantly weakened in the new design. This is because more failures of the location server can be tolerated by the car controller, thus the correctness of car controller is less dependent on the correctness of location server. As a comparison, our classification of dependency strength is based on a formally defined metric in view of failure semantics mapping and can be effectively applied in dependency relation evaluation concerning design changes.

We begin with the notion of failure semantics mapping. When component A receives the service delivered by component B, a specific failure semantics FS_B^S (a nonempty subset of the failure semantics FS_B) of component B may introduce a specific failure semantics $FS_A^{S'}$ to component A if the failure behaviors specified in FS_B^S cannot be properly masked or tolerated by component A. On the contrary, if component A can tolerate all the failure behaviors specified in FS_B^S, no failure behaviors specified in FS_B^S will introduce failure to component A. Intuitively, the dependency strength of component A on component B is stronger in the former case than in the latter case, because the correctness of component A is less influenced by the failures of component B in the latter case.

We capture this observation with the following definition:

Definition 4: *Assume that component A will function correctly if all the*

services received by A is correct. In addition, assume that all services received by A is correct except for the service of component B. If a specific failure semantics $FS_A^{S'} \subseteq FS_A$ *may be introduced to component A by the failures specified in* $FS_B^S \subseteq FS_B$ *of component B when the service of component B is delivered to component A, we say that* FS_B^S *is mapped to* $FS_A^{S'}$ *in this service delivery, denoted as* $map(FS_B^S, B, A) = FS_A^{S'}$.

Trivially, the normal (i.e., no failure) functional semantics of component B is mapped to the normal functional semantics of component A, i.e., $map(\emptyset_B, B, A) = \emptyset_A$. This is obvious if the two assumptions of Definition 4 are noticed. This observation is also based on the following general assumption in our framework.

Pessimistic Annotation Assumption: *In our framework, we assume that pessimistic fault annotations are used. If we cannot verify a component is correct, we should annotate the potential faults. Similarly, if we cannot verify whether a fault of component B will incur fault to component A through service delivery, we should annotate the potential fault propagations.*

Suppose that component B delivers service to component A. If component A will fail for any faulty service delivered by component B, 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. These concepts are defined based on the notion of failure semantics mapping as follows.

Definition 5: *The dependency strength of component A on component B is measured by the largest subset* FS_B^S *of component B's failure semantics* FS_B *such that* $map(FS_B^S, B, A) = \emptyset_A$.

The intuition is: the stronger (i.e., smaller subset) the failure semantics FS_B^S is, the larger its complement set $\overline{FS_B^S}$ with respect to FS_B, which implies more failure behaviors of component B that cannot be tolerated by component A, thus the higher probability that a failure in B leads to a failure in A, therefore the stronger the correctness of component A depends on the correctness of component B.

Based on this definition, we can categorize the dependency relations between two components into three classes from a high level perspective. For example, suppose the failure semantics of location server (LS) FS_{LS} is the same as what is given in last section's example.

$FS_{LS} = \{DeadlineMiss\langle 5\rangle\} \bigcup \{LocationError\langle 0, 20\rangle\} \bigcup \{Crash\}$.

The car controllers (CC) receives the data service (i.e., location information) delivered by the location server. If car controller directly makes use of location server's service without dealing with any of location server's possible failure behaviors as specified in FS_{LS} (as illustrated in Figure 1.2-a), then for any nonempty subset FS^S_{LS} of FS_{LS}, we have $map(FS^S_{LS}, LS, CC) \neq \emptyset$, thus the dependency of controller on location server is measured by $\emptyset$. In this situation, we say that car controller is ***totally dependent*** on location server, meaning if location server fails, car controller fails.

Definition 6: *Component A is totally dependent on component B if the dependency strength of component A on component B is measured by* $\emptyset_B$ [1]. *We denote this as* $TDep(A, B)$.

By installing a Kalman filter at the sites of car controllers (as illustrated in Figure 1.2-b), the Kalman filter can produce sufficiently accurate location estimations at periodic intervals to the car controller, when up to k (suppose $k = 3$) location packets are delayed or lost. What's more, when the location server crashes, if it can be recovered within the time period of Kalman filter delay tolerance, the car controllers will not even notice such a crash failure of the location server. Now we have a nonempty proper subset $FS^S_{LS} = \{DeadlineMiss\langle 3\rangle\} \bigcup \{Crash\}$ such that $map(FS^S_{LS}, LS, CC) = \emptyset$. In other words, the dependency of car controller on location server is measured by a non-empty proper subset FS^S_{LS} of FS_{LS}. We say that car controller is ***partially dependent*** on location server.

Definition 7: *Component A is partially dependent on component B if the dependency strength of component A on component B is measured by a non-empty proper subset of* FS_B. *We denote this as* $PDep(A, B)$.

Definition 8: *Total dependency and partial dependency are collectively called dependency, denoted as* $Dep(A, B)$, *that is* $Dep(A, B) = TDep(A, B) \bigvee PDep(A, B)$.

If all the possible failures of a component B can be tolerated by component A, that is, even if component B fails, component A still works normally, we say that component A ***USE*** component B.

[1] As in Definition 5, by dependency strength measure $\emptyset_B$, we denote that no failure of component B can be tolerated by component A.

Definition 9: *Component A USE component B if the dependency strength of component A on component B is measured by FS_B. We denote this as $USE(A, B)$.*

From Definition 5-9, we see that the high level dependency relation (total dependency/partial dependency/USE) is an abstraction (zoom out) of the low level dependency relation (failure semantics mapping), while the low level dependency relation is an elaboration (zoom in) of the high level dependency relation.

As argued in Chapter 1, the most important system robustness criterion is that critical components should only USE instead of depend on less critical ones. From the fault propagation perspective, this implies that none of the failures of the less critical components should be propagated, either directly or indirectly, to critical components. We have the following definition.

Definition 10: *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.*

Definition 11: *A system has well-formed dependencies if no dependency inversion occurs in the system; otherwise, the system has ill-formed dependencies.*

2.3 Low level dependency tracking – failure semantics mapping and reasoning

Dependency tracking can be done at different levels. A high level tracking of USE, partial and total dependency allows us to quickly determine if the dependency relation is consistent with the criticality ordering between components. That is, whether the critical components only USE instead of depend on less critical components. If this observation holds, the system design has well-formed dependencies, otherwise, the dependencies of the system design is ill-formed. A low level tracking of the failure semantics propagation across component boundaries quantifies the nature of the dependency relation from a fault propagation perspective, and enables us to reason about how faults/failures of a software component may or may not affect other interacting components.

Our DMF prototype toolkit supports both low level and high level dependency tracking. The composition rules derived in this section and next section serve as the theoretical foundation for the dependency tracking facilities of the prototype toolkit.

2.3.1 The transitive failure semantics mapping

In this section. we first present the composition rules for low level dependency tracking in DMF. We will come to the composition rules for high level dependency

tracking in next section.

Theorem 1 – Failure semantics mapping rule 1:

- $map(FS_A^{S_1}, A, B) \subseteq map(FS_A^{S_2}, A, B)$ if $FS_A^{S_1} \subseteq FS_A^{S_2}$
- $map(FS_A^{S_1}, A, B) \bigcup map(FS_A^{S_2}, A, B) \subseteq map(FS_A^{S_1} \bigcup FS_A^{S_2}, A, B)$

Proof. The first rule is straightforward based on Definition 4. Since $FS_A^{S_1} \subseteq FS_A^{S_2}$, the failure semantics that may be introduced to component B by the failures specified in $FS_A^{S_1}$ is, of course, a subset of the failure semantics that may be introduced to component B by the failures specified in $FS_A^{S_2}$. Thus we have $map(FS_A^{S_1}, A, B) \subseteq map(FS_A^{S_2}, A, B)$ based on the definition of *map*.

With respect to the second rule, it should be emphasized that the relation is $\subseteq$ instead of $=$. This is because a certain failure F_A^1 in subset $FS_A^{S_1}$ combined with a certain failure F_A^2 in subset $FS_A^{S_2}$ may lead to a certain failure F_B^{new} that cannot be found in either $map(FS_A^{S_1}, A, B)$ or $map(FS_A^{S_2}, A, B)$. This is because from Definition 4, $map(FS_A^{S_1}, A, B)$ only contains the failures that may be introduced to component B assuming component A only has failures specified in $FS_A^{S_1}$, and thus assuming that no failures specified in $FS_A^{S_2}$ will ever occur in this service delivery if $FS_A^{S_1}$ and $FS_A^{S_2}$ are disjoint. The same argument applies to $map(FS_A^{S_2}, A, B)$. Thus we have $map(FS_A^{S_1}, A, B) \bigcup map(FS_A^{S_2}, A, B) \subseteq map(FS_A^{S_1} \bigcup FS_A^{S_2}, A, B)$. □

These two rules are very important although they can easily be obtained from Definition 4. If the failure semantics mapping for each of component A's basic failure types (i.e., *DeadlineMiss*$\langle n \rangle$, *LocationError*$\langle u, v \rangle$, *Crash*, etc) has been specified, and the failure semantics mapping for each *effective* combination of the basic failure types has been specified, then the failure semantics mapping for *any subset* of the failure semantics FS_A can thus be obtained by proper set union operations. Here, an *effective* combination of the basic failure types is a combination that can lead to new failures of component B that cannot be found in any of the mappings of component A's single basic failure types.

In addition to the simple pairwise components scenario that the service of component C is directly delivered to component A, a slightly more complicated interaction pattern between two components is: the service of component C is indirectly delivered to component A through a component B, i.e, the service delivery path is $A \leftarrow B \leftarrow C$. We have the following rule for this composition pattern.

Theorem 2 – Failure semantics mapping rule 2:

- $map(FS_C^S, C, B) = FS_B^{S'} \bigwedge map(FS_B^{S'}, B, A) = FS_A^{S''}$
  $\Rightarrow map(FS_C^S, C, A) = FS_A^{S''}$

Proof. This is easy to prove based on the definition of *map*. Keep in mind that in this theorem, we assume that the service delivery path is simply $A \leftarrow B \leftarrow C$, i.e., there is no direct service delivery from component C to component A (or they are not immediately interacting components). □

The most complicated interaction pattern between two components A and C is that: A directly receives the service of component C, as well as indirectly receives the service of component C through components B_1, B_2, ..., B_N, which immediately interact with component A. That is, the service delivery paths are: $A \leftarrow C$, $A \leftarrow B_i \leftarrow ... \leftarrow C$ $(i = 1, ..., N)$. We have the following composition rule:

Theorem 3 – Failure semantics mapping rule 3:

Suppose along the service delivery path $A \leftarrow B_i \leftarrow ... \leftarrow C$ (denote $i = 0$ in case $A \leftarrow C$), a specific failure semantics FS_C^S is transitively mapped to $FS_A^{S_i'}$, i.e., $map(FS_C^S, C, A) = FS_A^{S_i'}$ is obtained along the i-th service delivery path from component C to component A. We have $map(FS_C^S, C, A) = FS_A^{S_0'} \bigcup FS_A^{S_1'} \bigcup ... \bigcup FS_A^{S_N'}$.

Proof. This theorem can be proved based upon Theorem 2 and Definition 4. □

It is worthwhile to point out that the above ***Three Failure Semantics Mapping Rules***, although seemingly a little trivial, enable the scalability of the whole failure semantics mapping reasoning system. Only the failure semantics mappings between immediately interacting components need to be specified. What's more, only failure semantics mappings regarding the basic failure types and their *effective* combinations are required [2]. A *global* picture of failure semantics mappings can be composed from the *local* specifications of failure semantics mapping based on the three failure semantics mapping rules.

2.3.2 The hierarchical failure semantics mapping

When the failure semantics mapping is to be hierarchically composed, i.e., when we want to derive the failure semantics of a component (e.g., a functionality module) from the failure semantics of its lower-level subcomponents (e.g., functions), more complex composition rules should be supported. Generally, we should be able to support the following specification: *When the failure semantics of subcomponents A_1, A_2, ..., A_N make a certain boolean expression true, component A will exhibit failure semantics FA_A^S.*

[2] As stated above, a combination of the basic failure types is an *effective* combination if it can lead to new failures of the service receiving component that cannot be found in any of the mappings of the service delivering component's single basic failure types.

For example, in the commonly used majority voting method, *k out of N* implies that the service of component A will be correctly delivered only if k out of the N subcomponents A_i behave correctly. For example, suppose three different functions are implemented to realize the same algorithm, and they only have *value* failure semantics, i.e., their only possible failure is returning the wrong result. The functionality module A compares the results returned by the three functions, and performs a majority voting to deliver the final result to the component that invokes this algorithm. This is a typical *2 out 3* composition. In our failure semantics mapping terminology, they are expressed as: $(A_1.\{Value\} \wedge A_2.\{Value\}) \vee (A_2.\{Value\} \wedge A_3.\{Value\}) \vee (A_1.\{Value\} \wedge A_3.\{Value\}) \vee (A_1.\{Value\} \wedge A_2.\{Value\} \wedge A_3.\{Value\}) \Rightarrow A.\{Value\}$

Notice that any boolean expression can be transformed to a canonical disjunctive form similar to the above expression, therefore, any failure semantics mapping rule with complex boolean expression can be decomposed to several failure semantics mapping rules with only conjunctive boolean expressions.

$A_1.\{Value\} \wedge A_2.\{Value\} \Rightarrow A.\{Value\}$

$A_2.\{Value\} \wedge A_3.\{Value\} \Rightarrow A.\{Value\}$

$A_1.\{Value\} \wedge A_3.\{Value\} \Rightarrow A.\{Value\}$

$A_1.\{Value\} \wedge A_2.\{Value\} \wedge A_3.\{Value\} \Rightarrow A.\{Value\}$

We can denote them using our *map* formalism as defined in Definition 4:

$map((\{Value\}, \{Value\}, \{\}), A_1 \bigoplus A_2 \bigoplus A_3, A) = \{Value\}$

$map((\{\}, \{Value\}, \{Value\}), A_1 \bigoplus A_2 \bigoplus A_3, A) = \{Value\}$

$map((\{Value\}, \{\}, \{Value\}), A_1 \bigoplus A_2 \bigoplus A_3, A) = \{Value\}$

$map((\{Value\}, \{Value\}, \{Value\}), A_1 \bigoplus A_2 \bigoplus A_3, A) = \{Value\}$

Notice that $(\{Value\}, \{Value\}, \{\})$, etc are elements of the failure semantics of the composition of these three subcomponents $A_1 \bigoplus A_2 \bigoplus A_3$, as defined in Definition 2.

2.4 High level dependency tracking – depend/use relation propagation and reasoning

In this section. we present the high level dependency composition rules in DMF, which gives the composition rules of *total dependency*, *partial dependency*, and *USE* relations beyond pairwise component dependency.

2.4.1 The transitive dependency relation propagation

As in Section 2.3, we start with the interaction pattern that the service of component C is indirectly delivered to component A through a component B.

Theorem 4 – Dependency relation propagation rule 1:

- $TDep(A,B) \bigwedge TDep(B,C) \Rightarrow TDep(A,C)$
- $TDep(A,B) \bigwedge PDep(B,C) \Rightarrow PDep(A,C)$
- $PDep(A,B) \bigwedge TDep(B,C) \Rightarrow PDep(A,C)$
- $PDep(A,B) \bigwedge PDep(B,C) \Rightarrow PDep(A,C) \bigvee USE(A,C)$
- $Dep(A,B) \bigwedge USE(B,C) \quad \Rightarrow USE(A,C)$
- $USE(A,B) \bigwedge Dep(B,C) \quad \Rightarrow USE(A,C)$
- $USE(A,B) \bigwedge USE(B,C) \quad \Rightarrow USE(A,C)$

Proof. This can be proved based on the definition of *total dependency*, *partial dependency*, and *USE*, as presented in Definition 6-9. We only prove the fourth rule here. The other rules can be proved in the same way.

Recall that $TDep(A,B)$ res. $PDep(A,B)$ res. $USE(A,B)$ implies that none res. a nonempty proper subset res. all of the failure semantics FS_B of component B is mapped to the normal (i.e., no failure) functional semantics of component A.

Suppose the dependency strength of component B on component C is measured by $\overline{FS_C^S}$, i.e., $\overline{FS_C^S}$ is the largest subset of FS_C such that $map(\overline{FS_C^S}, C, B) = \emptyset_B$. We assume that FS_C^S is mapped to $FS_B^{S_1'}$, a nonempty subset of FS_B. That is, we have $map(FS_C^S, C, B) = FS_B^{S_1'}$. Similarly, suppose the dependency strength of component A on component B is measured by $\overline{FS_B^{S_2'}}$. We assume that $map(FS_B^{S_2'}, B, A) = FS_A^{S''}$, where $FS_A^{S''}$ is a nonempty subset of FS_A.

If $FS_B^{S_1'} \bigcap FS_B^{S_2'} \neq \emptyset$, a nonempty set $FS_C^{S^*} \subseteq FS_C^S \subset FS_C$ is mapped to a nonempty set $FS_B^{S_1'} \bigcap FS_B^{S_2'} \subseteq FS_B^{S_1'}$. And the nonempty set $FS_B^{S_1'} \bigcap FS_B^{S_2'} \subseteq FS_B^{S_2'}$ is mapped to a nonempty set $map(FS_B^{S_1'} \bigcap FS_B^{S_2'}, B, A) \subseteq map(FS_B^{S_2'}, B, A) = FS_A^{S''} \subseteq FS_A$. That is, $FS_C^{S^*}$, a nonempty proper subset of FS_C, is transitively mapped to a nonempty subset of FS_A, and $\overline{FS_C^{S^*}}$ is transitively mapped to the normal (i.e., no failure) functional semantics of component A. In other words, the dependency strength of component A on component C can be measured by $\overline{FS_C^{S^*}}$, a nonempty proper subset of FS_C. Thus we have $PDep(A,C)$.

If $FS_B^{S_1'} \bigcap FS_B^{S_2'} = \emptyset$, the above argument is invalid, and we know that all the failure semantics of component C is transitively mapped to the normal (i.e., no

failure) functional semantics of component A. That is, the dependency strength of component A on component C is measured by FS_C. Thus we have $USE(A, C)$.

Therefore we have $PDep(A, B) \bigwedge PDep(B, C) \Rightarrow PDep(A, C) \bigvee USE(A, C)$. □

Let's continue to consider the most complicated interaction pattern between two components A and C: A directly receives the service of component C, as well as indirectly receives the service of component C through components B_1, B_2, ..., B_N, which immediately interact with component A.

Theorem 5 – Dependency relation propagation rule 2:

$USE(A, C)$ is true if and only if along all the service delivery paths, $USE(A, C)$ can be derived. Otherwise, $Dep(A, C)$ is true. Further, suppose for the service delivery path $A \leftarrow B_i \leftarrow ... \leftarrow C$ (denote $i = 0$ in case $A \leftarrow C$), the dependency strength measurement of component A on component C is $FS_C^{S_i}$, i.e., $map(FS_C^{S_i}, C, A) = \emptyset_A$ is obtained along the i-th service delivery path from component C to component A. If $\bigcup \overline{FS_C^{S_i}} = FS_C$, then we have $TDep(A, C)$; otherwise we have $PDep(A, C)$.

Proof. Notice that the failure semantics of component C specified in $\overline{FS_C^{S_i}}$ will be transitively mapped to a nonempty subset of FS_A. If the union of $\overline{FS_C^{S_i}}$ equals FS_C, this implies that taking all the service delivery paths into account, none of the failure semantics FS_C is finally mapped to the normal (i.e., no failure) functional semantics of component A. In other words, the dependency strength of component A on component C is measured by $\emptyset_C$. Thus, we have $TDep(A, C)$. The partial dependency and USE scenario can be proved in the same way. □

2.4.2 The hierarchical dependency relation propagation

We now consider the depend/USE relation propagation in complex boolean compositions. As argued in Section 2.3.2, any failure semantics mapping rule with complex boolean expression can be decomposed to several failure semantics mapping rules with only conjunctive boolean expressions.

It is simple for the disjunctive scenario, i.e., the effect of subcomponent A_i's failure on component A is independent with the other subcomponents. In this scenario, the dependency relation of component A on the composition of components A_i $(i = 1, ..., N)$ is just the strongest among the dependency relations of component A on each subcomponent A_i. For example, if component A *totally depends* on subcomponent A_1, then it *totally depends* on the composition, no matter what

are the dependency relations of component A on the other subcomponents. If the dependency relation of component A on each and every subcomponent A_i is *USE*, then it *USE* the composition.

But for conjunctive scenario, things are much more complicated. Take the k *out of* N majority voting method for example. In this scenario, we can not say that A depends on any subcomponent A_i, since even if this particular A_i fails, the service can still be properly delivered by component A provided that k out of the remaining $N-1$ subcomponents do not fail. But can we say that A USE A_i? That is, A USE A_1, A USE A_2, ..., A USE A_N, and thus conclude that A USE all the subcomponents A_1 through A_N? No, because in order to ensure A working correctly, the N subcomponents cannot fail simultaneously. Otherwise, A will fail. Thus it is not appropriate to say that A USE all the subcomponents, either. Actually, we have the following theorem.

Theorem 6 – Dependency relation propagation rule 3:

- $TDep(A, A_1) \bigwedge TDep(A, A_2) \Rightarrow TDep(A, A_1 \bigoplus A_2)$
- $TDep(A, A_1) \bigwedge PDep(A, A_2) \Rightarrow TDep(A, A_1 \bigoplus A_2)$
- $PDep(A, A_1) \bigwedge PDep(A, A_2) \Rightarrow \quad Dep(A, A_1 \bigoplus A_2)$
- $TDep(A, A_1) \bigwedge USE(A, A_2) \ \Rightarrow TDep(A, A_1 \bigoplus A_2)$
- $PDep(A, A_1) \bigwedge USE(A, A_2) \ \Rightarrow \quad Dep(A, A_1 \bigoplus A_2)$
- $\quad USE(A, A_1) \bigwedge USE(A, A_2) \Rightarrow \quad Dep(A, A_1 \bigoplus A_2) \bigvee USE(A, A_1 \bigoplus A_2)$

Proof. This can be proved following the same idea as demonstrated in the proof of Theorem 5, based on the definition of *total dependency*, *partial dependency*, and *USE*, as presented in Definition 6-9. We address the sixth rule here a little bit.

The sixth rule is trivial, since $Dep(A, A_1 \bigoplus A_2) \bigvee USE(A, A_1 \bigoplus A_2)$ is always true. It is listed here because we want to emphasize that even though $USE(A, A_1) \bigwedge USE(A, A_2)$ is true, it is still possible $Dep(A, A_1 \bigoplus A_2)$ is true. That is, a component A may (totally or partially) depend on a composition of two components, even if A only USE each component individually.

First recall that $FS_{A_1 \bigoplus A_2} = FS_{A_1} \times FS_{A_2}$. The key point is that in case of the composition of two components A_1 and A_2, $USE(A, A_1)$ implies $map(FS_{A_1} \times \emptyset_{A_2}, A_1 \bigoplus A_2, A) = \emptyset_A$, from Definition 4. Similarly, $USE(A, A_2)$ implies $map(\emptyset_{A_1} \times FS_{A_2}, A_1 \bigoplus A_2, A) = \emptyset_A$. However, the two conditions do not make $map(FS_{A_1} \times FS_{A_2}, A_1 \bigoplus A_2, A) = \emptyset_A$ true, which is the condition to make $USE(A, A_1 \bigoplus A_2)$ true. Now the paradox is clearly explained. Although component A only uses

each individual component A_i, the relation of component A on the component composed by these sub-components is *depend* instead of *USE*.

A more intuitive explanation is that when we have $Use(A, A_1)$, we really mean if all the other services (from A_2,...,A_N) are delivered correctly to component A, then the failure semantics of component A_1 is mapped to the normal failure semantics of component A. But when A_2 can neither correctly deliver its service to A, the failure semantics of component A_1 is not necessarily mapped to the normal failure semantics of component A. □

From the ***Three Dependency Relation Propagation Rules***, we can then derive the dependency relation between any two components in the system transitively (rule 1 and 2) and hierarchically (rule 3). Thus we are able to track the system-wide USE, total and partial dependency relations between components from local annotations. The three dependency relation propagation rules can help to quickly determine if the dependency relation is consistent with the criticality ordering between components, i.e., whether critical component only USE instead of depends on less critical components. This is a very important principle in designing robust real-time systems. A violation of this principle is called *Dependency Inversion*, and such a software system design is said to have *ill-formed dependencies*, as stated in Definition 10 and 11.

Finally, it is worthwhile to point out that Theorem 1-6 enable the scalability of our DMF prototype toolkit to be presented in the next two chapters from the theoretical perspective. Only the failure semantics mappings between immediately interacting components need to be specified. What's more, only failure semantics mappings regarding the conjunctive composition of basic failure types are required. *Global* system-wide failure propagation properties and dependency relations can be composed from the *local* annotations of failure semantics mapping based on Theorem 1-6. A more detailed discussion on scalability of our framework is presented in Section 5.2.

Chapter 3

The Prototype Toolkit of DMF: Basic Features

3.1 The architecture of DMF toolkit

In the following sections and Figure 3.1, we use the term *OS-layer* to denote real-time operating system domain rules, and *App-layer* to denote application-specific dependency rules. Figure 3.1 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 semantics of the components as well as the fault/failure propagation rules across component boundaries 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 failure types for these components, and *condition* is an optional expression governing under what circumstances the failure 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 dependency, partial dependency, and USE) and composition theorems in Chapter 2. The dependency query result is then displayed to the user in text format [1].

We will highlight the main features of DMF's dependency specification language in Section 3.3 and Section 4.2, the OS-layer dependency rules in Section 4.4, and demonstrate the usage of DMF's dependency query commands in Section 3.4 and Section 4.3.

We choose Maude [2, 14] as the backbone logic reasoning system. The reason

[1] It is the real-time systems laboratory's planned work to generate graphical format query result. A new Ph.D. student will work toward this direction in addition to many other future research directions as pointed out in Section 7.2.

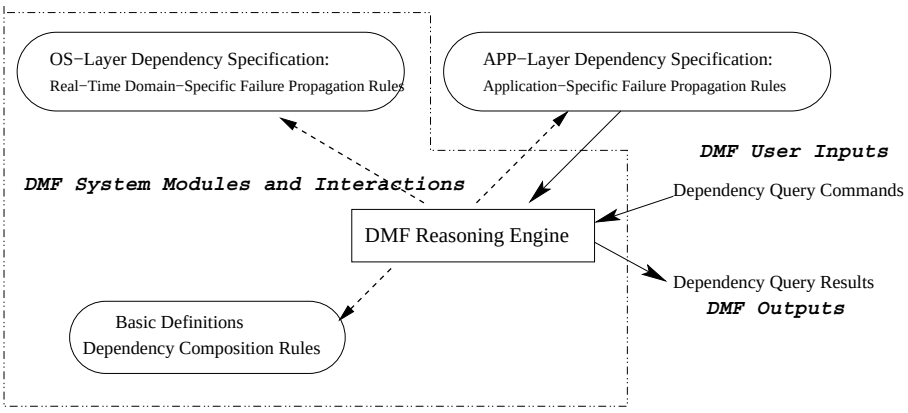


Figure 3.1: The Architecture of DMF Prototype Toolkit

why Maude is chosen is one can very naturally and easily define new logics in Maude, together with their operational semantics. The rewriting engine of Maude is very efficient. The current version of Maude can do up to 3 million rewritings per second on standard PCs, and its compiled version is intended to support 15 million rewritings per second [44]. Hence, we have decided to use Maude as the backbone logic specification and reasoning engine for DMF. However, the front-end users need no knowledge of Maude at all.

Logically, the dependency specification in our toolkit contains three dependency specification layers:

1. Layer I is the underlying reasoning rules for failure semantics mapping and *depend/USE* relation propagation (i.e., Theorem 1-6), domain-specific failure semantics mapping rules as presented in Section 4.4, and primitive definitions (e.g., definition for subset relation between failure semantics, definition for total depend, partial depend, and USE).

2. Layer II is the application-specific failure semantics mapping rules (the internal representation is in the form $map(FS_B^S, B, A) = FS_A^{S'}$ *if condition* .), where FS_B^S, $FS_A^{S'}$, and *condition* are functions of annotated component properties and their interaction protocols.

3. Layer III consists of the actual values of the annotated properties of components and their interaction protocols in the current system design. Some are explicit, such as communication protocol and period. While others are implicit, such as Kalman filter delay tolerance.

The dependency query in our toolkit assists the developers to: 1) improve the robustness of the system design; 2) compare the robustness of different designs.

The dependency query commands can be categorized into two general classes: *1) high level dependency tracking, i.e., dependency inversion checking; 2) low-level dependency tracking, i.e., impact analysis and root cause analysis.* **Alternatively**, they can also be viewed as four big classes of dependency query facilities, from the perspectives of: *1) low level fault propagation, 2) high level depend/USE tracking, 3) well-formed dependency checking, and 4) robustness metric comparison.*

For illustrative purpose, we perform the dependency query in view of the **two** general categories for the ION CubeSat case study in this chapter, and in view of the **four** dependency query facilities for the car control testbed case study in next chapter. But it should be noticed that these two different perspectives are just different views of the DMF dependency tracking facilities, nothing more. And we illustrate these facilities in different ways so that readers can get a deeper and more versatile understanding of the tool itself.

3.2 A brief description of the ION CubeSat

Historically, space has been the purview of government agencies with multi-billion dollar budgets. Typical satellites took hundreds of millions of dollars and many years to develop. However, 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 (Illinois Observing Nanosatellite), a double CubeSat, which has been launched on July 26, 2006. 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, Computer, 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, photomultiplier 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 3.2.

It is out of the scope of this thesis to fully elaborate on ION's architecture and design issues. Interested readers can find full information in [3] and [18]. The homepage of ION CubeSat also has a lot of useful information of this project [2].

[2]ION CubeSat Project: http://cubesat.ece.uiuc.edu/

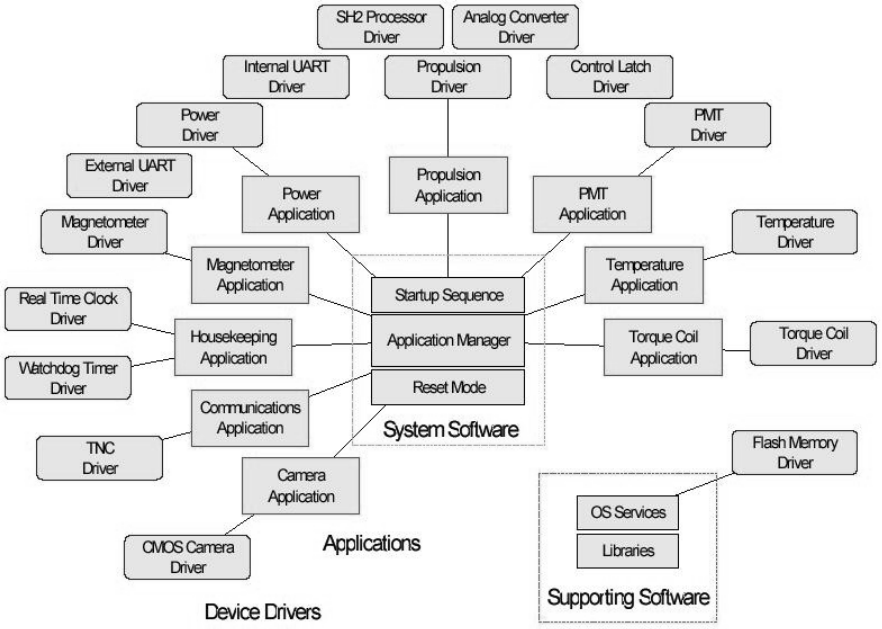


Figure 3.2: The ION CubeSAT Software System

3.3 Dependency specification language of DMF

The following section details the syntax and features of the dependency specification language, utilizing examples taken from the fault propagation rules for the ION CubeSat. ION CubeSat was developed entirely at the University of Illinois, with around **50,000** lines of code. It controls every aspect of the satellite. The system consists of quite a lot of applications and drivers, each responsible for a specific hardware component of the satellite.

The case study presented in this chapter is a cooperation achievement of this author and a master student, Leon Arber, who is one of the primary software developers of ION CubeSat. Arber is responsible for writing down the dependency specifications following DMF's dependency specification language syntax. This author is responsible for assisting him to design and perform the dependency queries to detect dependency inversions and track dependency propagations in the satellite system. It is also an iterative process to further improve the DMF toolkit's specification and query features to make it more usable and scalable by doing such a serious case study on a real world, complex software system.

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* (a failure that occurs when a file is written in the filesystem) and *fCreateFile* (a failure that occurs when a file is created in the filesystem) failures.

Notice that the *sw* prefix for components and the *f* and *s* prefix for failures are just the annotation styles that Leon Arber prefers. It's not a requirement of DMF. The DMF users are free to choose whatever annotation styles they like provided that they follow the dependency specification language syntax.

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 [3]

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 meaning of each failure term of *swPowerApp* can be found in Table 3.1. 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;

The meaning of each failure term of *swPowerApp* can be found in Table 3.2.

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)* $\bigcup$ *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.

3.3.3 Boolean expressions of failure 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

[3]Notice that the examples listed in this chapter are only for illustrative purposes - to illustrate the dependency specification language syntax and query facilities. We only create a table to explain the meaning of each term in the following two specification examples. A full explanation of the meaning of each failure term and failure propagation rule and a full description of the case study are out of the scope of this dissertation, and the interested readers are referred to Leon Arber's master thesis [11].

Failure Term	Meaning of This Failure Term
fDeadlineMiss	The power application misses its deadline.
fPowerSampleWrite	The power application fails to write out the latest voltage/current readings to the file system.
fPowerSampleData	The power application fails to sample the latest voltage/current readings.
fPowerSampleUpdateBeacon	The power application fails to update the beacon with the latest voltage/current readings.
fAutoPowerExec	The work unit dealing with autonomous (i.e., not-ground controlled) power management fails to execute.
fAutoPowerSetState	The power application fails to properly set the current power state as directed by the autonomous power work unit.
fSetPowerStateExec	The power application fails to execute the work unit responsible for setting the power state (as controlled from the ground).
fSetPowerStateInvalid	The power application sets the power state of the system incorrectly.

Table 3.1: The Failure Terms of Component *swPowerApp*

Failure Term	Meaning of This Failure Term
fRehash	An application fails to update its settings as commanded by the ground.
fInvalidConfigData	An application finds that its configuration file was corrupt.
fGiveUpCPU	An application fails to give up the CPU to another application.
fStart	An application fails to start up.
fOpenOutputDatafile	An application fails to open an output data file.
fGetWorkUnit	An application cannot get the next work unit for itself.
fInitialRehash	An application fails to perform its initial read of default settings upon satellite startup.

Table 3.2: The Failure Terms of Component *swApplication*

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 rules 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 semantics 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*$\langle 11 \rangle$*;*

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

Another thing to be mentioned here is that this failure propagation rule can be combined with the *(swApplication.fGiveUpCPU* $\wedge$ *swAppManStartup.fCPU_WDTInit)* $\vee$ *...* $\mapsto$ *swSatellite.fSatelliteReset;* rule above. Actually, they can either be combined through a $\vee$ in one single rule or separated into two rules. DMF treat them identically.

Here, we call *fSatelliteReset*$\langle 11 \rangle$ 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*$\langle 5 \rangle$. Actually, we have already addressed the need for parameterized failure semantics in Section 2.1.

What's more, the user should also be able to specify that only when the system is designed in a certain manner (e.g., the connection between two components is TCP instead of UDP), can such a failure propagation rule be validate. That is, the failure propagation rules can be guarded by OS or application design properties. We will illustrate this feature together with the real-time domain support in Chapter 4.

3.4 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 interacting 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.

3.4.1 High-level dependency tracking – dependency inversion checking

Please refer the definitions of *dependency strength* and *dependency inversion* to Section 2.2. 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 3.4.2, we discuss the **impact** and **rootCause** dependency queries, which are specifically designed to track down the cause of dependency inversions.

3.4.2 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*} $\mapsto$ {*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. On 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*} $\mapsto$ *... ...* $\mapsto$
{*swPowerApp , fPowerSampleData*} $\mapsto$ {*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 optimal place to cut off the failure propagation is to 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.

In addition, the case study of ION CubeSAT does demonstrate that design flaws in system design become evident when writing dependency specification, for example, large number of flaws per component, large number of propagation rules per component, tendency to label all components as *critical*, and large number of dependency inversions due to criticality misspecification. DMF also helps the developer to locate some root causes of satellite failure which otherwise will be overlooked, such as the failures in the file systems and bootloader. A detailed description of this case study can be found in Leon Arber's master thesis [11].

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 be revealed 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.

Chapter 4

The Prototype Toolkit of DMF: Advanced Features

In this chapter, we present the advanced features of DMF toolkit, i.e., parameterized failure semantics and dependency expressions, and real-time systems domain support. We also explain the dependency tracking facilities from an **alternative** view by exploiting the case study on the distributed car control testbed. The implementation issues of each dependency tracking facility are also briefly discussed in this chapter.

We first list the components of the distributed car control testbed in Section 4.1. We then demonstrate how our toolkit tackles challenge II (desired tight binding due to constantly changing software designs, referring to Section 1.3) in Section 4.2. We will demonstrate how to make use of the information of dependency tracking in improving robustness of the system design and how to compare the robustness of different designs in Section 4.3, and implementation issues are briefly discussed with respect to each dependency tracking feature.

4.1 A brief description of the distributed car control testbed

In the examples presented in this chapter, we investigate the dependency relations in the execution/scheduling unit layer, i.e., between threads. That is, in the following discussion, a *component* of the distributed car control testbed denotes a thread. Suppose there are N cars to be controlled in the system, and they are numbered as car 1, car i, ..., car N.

Vision server only has 1 thread, denoted as *VS*.

In addition to the main thread, the threads of location server include:

- *LSvs:* receive location packet from *VS*
- *LStm:* send location packet to trajectory manager
- *LScc[i]*: send location packet to the i-th car controller

In addition to the main thread, the threads of trajectory manager include:

- *TMls:* receive location packet from *LStm*
- *TMcc[i]:* send desired trajectory information to the i-th car controller

In addition to the main thread, the threads of the i-th car controller include:

- *CCls[i]:* receive location packet from *LScc[i]*, and perform model-based state estimation in case of packet loss if Kalman filter is installed
- *CCtm[i]:* receive desired trajectory information from *TMcc[i]*
- *CCstop[i]:* detect the scenario when another car is too close to itself, and may directly send out STOP command to the actuator in emergency (this is called *fail-safe stop*)
- *CCcal[i]*: calculate control command based on the desired trajectory, and send control command to the actuator. It may also plan a temporary trajectory deviation to avoid collision.

It should be pointed out that *CCstop[i]* is conservatively designed to tolerate the uncertainty in cars' locations, e.g., 5 distance units. As long as the distance between cars are larger than the uncertainty in cars' locations, there will be no collision, because *CCstop[i]* will stop the car i once the distance between cars are close to 5 distance units. In other words, *LocationError⟨0,5⟩* fault can be tolerated by the system. Therefore, in the original design of the car control testbed as illustrated in Figure 1.2(a), *LocationError⟨5,Inf⟩* cannot be tolerated and may lead to car collision.

But in the enhanced design as illustrated in Figure 1.2(b), a Kalman filter is installed at the site of each car controller, and a too large location value error (i.e., larger than 10 distance units) can be detected by *CCls[i]* based on state estimation. The *CCstop[i]* will thus perform a fail-safe stop either when another car is within 5 distance units from the i-th car or when a location value error is detected. That is, in the enhanced design, only *LocationError⟨5,10⟩* cannot be tolerated and may lead to car collision.

There are other components who take care of the friendly user interface and other functionalities. To make the following presentation clear, we do not go into further details.

We define three system-level failures. The catastrophic system failure is *Collision*, i.e., car collision occurs. The less severe system failure is *FailSafeStop*, i.e., cars are stopped to avoid possible collision in case of serious communication block

or emergency. The third is *TrajectoryError*, that is, there is no collision or fail-safe stop, but the trajectory following errors are unacceptable.

An important objective of dependency management is to make sure that no *dependency inversion* exists. That is, faults/failures of the less critical components (components to realize the less critical requirements) will not be propagated to the more critical components (components to realize the more critical requirements). That is, critical components only *USE* instead of *depends on* less critical components. A violation of this principle indicates that the system design has ill-formed dependencies.

From the criticality of the system failures, we can see that the most critical requirement of the car control system is *collision free*. That is, it should be guaranteed that the cars will not collide with each other. Based on this, the secondary critical requirement is that the cars should follow their desired trajectory as much as possible, i.e., the error between the actual trajectory and the desired trajectory for each car, including the error between the actual time and desired time to reach a certain position, should be as small as possible. The car control system also has optional requirements, such as an easy-to-use user interface.

In the distributed car control testbed, the most critical components, which are essential to avoid car collision, are *VS*, *LSvs*, *LScc*, *CCls*, and *CCstop*. The other less critical components, which work together with most critical components to realize the trajectory following objective, are *LStm*, *TMls*, *TMcc*, *CCtm* and *CCcal*. The components responsible for friendly user interface, which are not listed here, are useful but non-critical components.

4.2 Tight binding between dependency expressions and software system designs

In this section, we present a few examples of the dependency specification language, emphasizing the tight binding between dependency expressions and software designs. The tight binding is realized by keeping track of the failure semantics mapping rules automatically as a function of annotated properties of components and their interaction protocols.

Example 1: With respect to the use of TCP or UDP connection between *VS* and *LSvs*, we have the following failure propagation rules.

$VS.Crash \mapsto LSvs.Lockup$
    if *commProtocol (VS, LSvs) == TCP;*
$VS.Crash \mapsto LSvs.Suspend\langle recoveryTime(VS)\rangle$

if *commProtocol (VS, LSvs) == UDP;*

That is, if TCP is used directly and if *VS* crashes, *LSvs* locks up and cannot continue even if *VS* is restarted. When UDP is used, *LSvs* will resume its service from suspension, when *VS* is restarted. The suspension duration is equal to the time needed by *VS* to recover from the crash.

Example 2: With Kalman filter installed at car controllers, the fail-safe stop parameter k is a function of the location packet update period p and the Kalman filter delay tolerance T. The following annotation concerns with the mapping of the *DeadlineMiss* failure of the location server to the car controllers.

LScc[i].DeadlineMiss⟨n⟩ ↦ CCls[i].Normal
if n <= floor (CCls[i].T / period (LScc[i], CCls[i]));
LScc[i].DeadlineMiss⟨n⟩ ↦ CCls[i].FailToPredict
if n > floor (CCls[i].T / period (LScc[i], CCls[i]));

That is, with Kalman filter installed at car controllers, up to k consecutive location packets from the location server can miss their deadlines, where $k = \lfloor \frac{T}{p} \rfloor$. But if the consecutive deadline misses of location packets exceed k, *CCls[i]* exhibits a *FailToPredict* failure, and *CCstop[i]* will be notified and immediately issue a STOP command to the actuator, i.e., the i-th car has to perform fail-safe stop. Kalman filter delay tolerance T is a design property of the component *CCls[i]*, while packet update period p is a design property of the interaction between component *LScc[i]* and *CCls[i]*. A too big location value error in the location packet can also be detected based on the model-based state estimation. The above annotation only concerns the *DeadlineMiss* failure.

In the TCP/UDP example 1, the dependency expression (failure semantics mapping specification) is a function of the interaction protocol between components. In the Kalman filter example 2, the dependency expression is a function of the annotated property of the components and their interaction properties. Generally, we define dependency based on the failure semantics mapping notion, and keep track of the failure dependency relations automatically as a function of the annotated properties of components, interaction protocols, and the environment. Our prototype toolkit automatically updates the affected failure propagation rules given the system design changes.

For example, the original design of the communication protocol between *VS* and *LSvs* is TCP, thus the underlying reasoning engine automatically applies the following rules in dependency tracking:

VS.Crash ↦ LSvs.Lockup;

Later, the communication protocol between *VS* and *LSvs* is changed from TCP to UDP as illustrated in Figure 1(b). Suppose *recoveryTime(VS) = 4*, i.e., the time

for the vision server to recovery from a crash is 4 seconds by design. The underlying reasoning engine will then automatically change the mapping rule to:

$$VS.Crash \mapsto LSvs.Suspend\langle 4\rangle;$$

Note that when *LSvs* locks up, the system will lock up and all the cars will be stuck in the *failure-safe stop* state even if *VS* is restarted. When the UDP is used between *VS* and *LSvs*, the car may continue to run smoothly as long as *VS* is successfully restarted before missing k packets.

Example 3: Suppose that the performance of Kalman filter is improved by new design and the delay tolerance T becomes 11 seconds from the original 7 seconds. Suppose also that the desired location packet update period is 2.5 seconds, then the above failure semantics mapping rules are automatically updated. That is,

$$LScc[i].DeadlineMiss\langle n\rangle \mapsto CCls[i].Normal \quad if\ n <= 2;$$
$$LScc[i].DeadlineMiss\langle n\rangle \mapsto CCls[i].FailToPredict \quad if\ n > 2;$$

Become

$$LScc[i].DeadlineMiss\langle n\rangle \mapsto CCls[i].Normal \quad if\ n <= 4;$$
$$LScc[i].DeadlineMiss\langle n\rangle \mapsto CCls[i].FailToPredict \quad if\ n > 4;$$

Recall that the dependency specification in our toolkit contains three dependency specification layers, as presented in Section 3.1. When the software design changes, the underlying dependency reasoning engine tracks the system design annotation changes in layer III, and automatically update the affected failure propagation rules in layer II. That is, the developer is not burdened to change the failure semantics mapping specification in layer II whenever the actual component property or interaction protocol is changed. The only necessary update is the current values of the annotated design properties in layer III.

One important point should be mentioned here. Including the component properties and interaction protocols, the specification of failure semantics mapping rules are local to the components, since they only concern the failure propagation between immediately interacting components. In other words, all the annotations above are local views. But a global view of the failure propagation along the dependency chain can be obtained via the composition of these local views by transitively or hierarchically applying Theorem 1-6. Thus the scalability of our approach is guaranteed.

4.3 Improving the robustness and comparing the robustness of different designs

We now turn our focus on how our reasoning framework and toolkit will assist us to 1) improve the robustness of the system design; 2) compare the robustness of different designs. Our prototype toolkit supports four dependency management functionalities to achieve this objective, from the perspectives of low level fault propagation, high level depend/USE tracking, well-formed dependency checking, and robustness metric comparison separately [1].

In robust real-time systems, we want key properties not affected by the potential faults in the services delivered by other components. In the distributed car control testbed, first of all, we want to ensure the correct functioning of the *fail-safe stop* operation. Second, we want to keep the cars running as much as possible. This requires us to have the *fail-safe stop* parameter k updated whenever location packet update period p or Kalman filter delay tolerance T is changed. We have already demonstrated this in the examples above. We have also illustrated that the use of UDP avoids the lockup of *LSvs* when *VS* crashes and reboots due to resource leaking (or preventive reboot). That is, UDP improves performance. The following examples illustrate the comparisons of the robustness of different designs when a low quality (but the quality is not low enough to be discovered as bad) image taken by the ceiling-mounted camera is received by the vision server, and hence a value error in the location packet.

4.3.1 Low level dependency tracking

The prototype toolkit supports two basic low level dependency tracking and reasoning functionalities. The reasoning engine tracks and reasons about the fault/failure propagation based on the dependency specification and Theorem 1-3.

- USAGE: **impactPP (Component, Failure)**

  PERFORMS: forward analysis, i.e., impact analysis

  RETURNS: the components and corresponding failures incurred by *Failure* of *Component* through direct or indirect service delivery, and the forward failure propagation path ($\{Component\ ,\ Failure\} \mapsto ...$)

[1] Again, this is an **alternative** view of the DMF dependency management facilities as illustrated in Section 3.4. We illustrate these facilities in different views so that readers can get a deeper and more versatile understanding of the tool itself.

- USAGE: **rootCausePP (Component, Failure)**

  PERFORMS: backward analysis, i.e., root-cause analysis

  RETURNS: the components and corresponding failures leading to *Failure* of *Component* through direct or indirect service delivery, and the backward failure propagation path (... $\leftarrow$ {*Component , Failure*})

Our toolkit also supports the forward analysis when two or more failures occur at the same time, for example:

- USAGE: **impactPP (Component1, Failure1, Component2, Failure2)**

We now show how to compare the robustness of different designs from the perspective of low level dependency tracking with two utility examples. Recall that (Section 4.1) in the original design of the distributed car control testbed, any value error larger than 5 will lead to car collisions. While in the enhanced design, only value error larger than 5 and smaller than 10 will lead to car collisions.

Example 4: In the original design, **impactPP(VS, LocationError$\langle$15$\rangle$)** will return the following:

- {*LSvs , LocationError*$\langle$*15*$\rangle$}

  PATH: {*VS , LocationError*$\langle$*15*$\rangle$} $\mapsto$ {*LSvs , LocationError*$\langle$*15*$\rangle$}

- {*LScc[i] , LocationError*$\langle$*15*$\rangle$}

  PATH: {*VS , LocationError*$\langle$*15*$\rangle$} $\mapsto$ {*LSvs , LocationError*$\langle$*15*$\rangle$}
  $\mapsto$ {*LScc[i] , LocationError*$\langle$*15*$\rangle$}

- {*CCls[i] , LocationError*$\langle$*15*$\rangle$}

  PATH: {*VS , LocationError*$\langle$*15*$\rangle$} $\mapsto$ {*LSvs , LocationError*$\langle$*15*$\rangle$}
  $\mapsto$ {*LScc[i] , LocationError*$\langle$*15*$\rangle$} $\mapsto$ {*CCls[i] , LocationError*$\langle$*15*$\rangle$}

- {SYSTEM *, Collision*}

  PATH: {*VS , LocationError*$\langle$*15*$\rangle$} $\mapsto$ {*LSvs , LocationError*$\langle$*15*$\rangle$}
  $\mapsto$ {*LScc[i] , LocationError*$\langle$*15*$\rangle$} $\mapsto$ {*CCls[i] , LocationError*$\langle$*15*$\rangle$}
  $\mapsto$ {SYSTEM *, Collision*}

where SYSTEM is a key word of our toolkit, exclusively used to associate with system-level failures.

The above information of failure propagation path is very useful in assisting the developers to improve the robustness of the system design. It helps the developers

to discover the multiple possible choices, and even the best choice, of places where fault masking or tolerance mechanism should be enforced. For example, in the enhanced design, the failure propagation path from {*LScc[i] , LocationError⟨15⟩*} to {*CCls[i] , LocationError⟨15⟩*} can be blocked by installing a Kalman filter at the site of car controller i. Any location error larger than 10 distance units can be detected by *CCls[i]*, and *CCstop[i]* will perform *fail-safe stop* to avoid collision.

Example 5: In the enhanced design, **impactPP(VS, LocationError⟨15⟩)** will return the following:

- {*LSvs , LocationError⟨15⟩*}

  PATH: {*VS , LocationError⟨15⟩*} $\mapsto$ {*LSvs , LocationError⟨15⟩*}

- {*LScc[i] , LocationError⟨15⟩*}

  PATH: {*VS , LocationError⟨15⟩*} $\mapsto$ {*LSvs , LocationError⟨15⟩*}
  $\mapsto$ {*LScc[i] , LocationError⟨15⟩*}

- {*CCls[i] , LocationErrorDetected*}

  PATH: {*VS , LocationError⟨15⟩*} $\mapsto$ {*LSvs , LocationError⟨15⟩*}
  $\mapsto$ {*LScc[i] , LocationError⟨15⟩*} $\mapsto$ {*CCls[i] , LocationErrorDetected*}

- {SYSTEM , *FailSafeStop*}

  PATH: {*VS , LocationError⟨15⟩*} $\mapsto$ {*LSvs , LocationError⟨15⟩*}
  $\mapsto$ {*LScc[i] , LocationError⟨15⟩*} $\mapsto$ {*CCls[i] , LocationErrorDetected*}
  $\mapsto$ {SYSTEM , *FailSafeStop*}

where *LocationErrorDetected* is a component state concerning fault detection or masking, not a fault, of *CCls[i]*. The introduction of component state of fault detection or masking enables more flexible fault propagation rules.

Comparing the two query results of Example 4 and Example 5, we can see that the enhanced design is more robust than the original design when there is value error in the location packet.

We now have a brief discussion of several **implementation issues**. For the dependency query *impact (Component1, Failure1, Component2, Failure2)*, the query result is the union of the query result of *impact (Component1, Failure1)*, and *impact (Component2, Failure2)*, as well as the possible failures that will occur when these two failures happen at the same time (Notice that some fault/failure mapping rule may be in the form $Component1.Failure1 \wedge Component2.Failure2 \mapsto Component3.Failure3$). Recall that $Component1.\ Failure1 \vee Component2.$

$Failure2 \mapsto Component3.\ Failure3$ will be decomposed to $Component1.\ Failure1 \mapsto Component3.\ Failure3$ and $Component2.\ Failure2 \mapsto Component3.\ Failure3$, therefore, in the underlying internal representation, there is no $\vee$ composition rules.

Actually, even for the basic dependency query **impact (Component, Failure)**, the $\wedge$ composition rules may still be encountered during the reasoning process. Because $Component\ .\ Failure$ may lead to $Component_i\ .\ Failure_i$ and $Component_j\ .\ Failure_j$ through different service delivery paths, and these two failures may conjunctively lead to $Component_k\ .\ Failure_k$. Therefore, we need to iteratively explore the possible conjunctive combinations of all the impacted failures to see whether new failures will be incurred due to a certain combination. The iteration terminates and the final query result is returned when no new failure is introduced for any conjunctive combination of the failures that have already been discovered.

Fortunately, most failure semantics mapping rules of software systems do not involve too many component failures on the left side, or they can be hierarchically decomposed to failure semantics mapping rules with simpler conjunctive boolean expression on the left side. So we needn't worry too much about the efficiency of the query reasoning engine.

4.3.2 High level dependency tracking

The prototype toolkit supports the high level depend/USE relation tracking and reasoning based on Theorem 4-6 and Definition 6-9.

- USAGE: **depUseRelation (Component1, Component2)**

  RETURNS: total dependency, partial dependency, or USE relation of *Component1* on *Component2*

We now show how to compare the robustness of different designs from the perspective of high level dependency tracking with two utility examples.

Example 6: In the original design, **depUseRelation(CCls[i], LScc[i])** will return the following:

- *The dependency relation of CCls[i] on LScc[i] is:* ***total dependency****.*

This is because in the original design, any failure of *LScc[i]*, such as location packet deadline miss or invalid location information (e.g., *LocationError⟨5,Inf⟩*), will lead to the failure of *CCls[i]*. The component *CCls[i]* may miss its deadline to provide location update to the other threads of the car controllers (e.g., *CCcal[i]*, *CCstop[i]*), or provide location information with value error to the other

threads who may then send out the wrong control command and finally lead to SYSTEM.*Collision*.

Example 7: In the enhanced design, **depUseRelation(CCls[i], LScc[i])** will return the following:

- *The dependency relation of CCls[i] on LScc[i] is:* ***partial dependency***.

The reason is that the Kalman filter installed at the site of car controller i can tolerate a certain number of consecutive deadline misses of *LScc[i]*, as well as detect a too big value error (e.g., *LocationError⟨10,Inf⟩*) based on the model-based state estimation. Therefore, the dependency strength measure of *CCls[i]* on *LScc[i]* is only a nonempty proper subset of $FS_{LScc[i]}$.

Comparing the two query results of Example 6 and Example 7, we can see that the dependency strength of component *CCls[i]* on component *LScc[i]* has been weakened in the enhanced design.

Example 8: In both original design and enhanced design, **depUseRelation(CCstop[i], TMls[i])** will return the following:

- *The dependency relation of CCstop[i] on TMls[i] is:* ***USE***.

Because *CCstop[i]* is in charge of detecting possible collision and directly issuing a STOP command in emergency to avoid collisions. No failure of *TMls[i]* (and actually all other threads of trajectory manager) will lead to the malfunctioning of *CCstop[i]*. But the component *CCcal[i]* depends upon *TMls[i]* in order to minimize the trajectory following error.

We now have a brief discussion of the **implementation issues**. With respect to the high-level dependency queries, the depend/USE relation between immediately interacting components have already been derived based on Definition 6-9 when the dependency specification is loaded. Once the dependency reasoning engine receives a *depUseRelation* query, it first applies Theorem 4-6 to derive the dependency of *Component1* on *Component2*. If their dependency relation cannot be uniquely determined through these composition rules, then the dependency reasoning engine applies Definition 6-9 to determine the dependency relation, where Theorem 1-3 are used to compose the system wide fault/failure propagation properties. In this case, the **impactAllFailures(Component2, Component1)** query (refer to Section 3.4.2) will be applied to determine the dependency strength measure of *Component1* on *Component2* by exploring the impact of each failure of *Component2* on *Component1*.

4.3.3 Checking of well-formed dependencies

As argued in Section 1.1, the most important system robustness criterion is that critical components should only USE instead of depend on less critical ones. From the fault propagation perspective, this implies that none of the failures of the less critical components should be propagated, either directly or indirectly, to critical components. If this is ensured in the system design, we say that the system design has *well-formed* dependencies, otherwise, it has *ill-formed* dependencies (refer to Definition 10 and Definition 11 in Chapter 2).

In the dependency specification, each component has an annotated property called *Criticality*. The higher the value of *Criticality* is, the more critical the component is. If the dependency relation is ill-formed, i.e., some critical components depend on less critical components, we say that there is *dependency inversion*. The checking of dependency inversion is performed using the following query command.

- USAGE: **depInversion**

  RETURNS: all pairs of components in the system where a dependency inversion occurs.

Now let's demonstrate the utility of **depInversion**.

Example 9: We purposely change the criticality of *VS* to be lower than the criticalities of the other most critical components. For example, the criticality of the other most critical components to guarantee collision-free is set to 3, while the criticality of *VS* is set to 2. The query command **depInversion** returns [2]:

{*PDEP : LSvs[3] , VS[2]*};
{*PDEP : LScc[3] , VS[2]*};
{*PDEP : CCls[3] , VS[2]*};
{*PDEP : CCstop[3] , VS[2]*};

That is, according to the dependency management principle, the more critical components *LSvs*, *LScc*, *CCls* and *CCstop* should not depend on the less critical component *VS*. But in the actual design, such a dependency inversion exists. We should have set the criticality of *VS* the same as the other most critical components and afford sufficient efforts to improve its robustness.

When the users of our toolkit are alerted of the *dependency inversion*, they should first check whether there is a criticality specification mistake, i.e., the criticality of some component is mistakenly specified, which is the case of Example 9. If there is no criticality specification mistake, then fault masking or fault tolerance

[2]Note that the integer in the brackets is the criticality of the corresponding component, not the index of the cars as illustrated in previous examples.

mechanism should be enforced to cut off the failure propagation path from the less critical component to the more critical component, and transform the *dependency* relation to *USE* relation. This information is thus very instructive and helpful for the developers to discover the potential dependency inversions and make the right effort to improve the robustness of the system. The lower-level dependency queries of Section 4.3.1 will help to reveal fault/failure propagation path details.

We now have a brief discussion of the **implementation issues**. With respect to the *depInversion* query, the underlying implementation simply explores all the component pairs where *Criticality(Component1) > Criticality(Component2)*, performs the dependency query *depUseRelation(Component1, Component2)*, and includes the pair in the returning set if the query result is *total dependency* or *partial dependency*.

4.3.4 Comparison of robustness metrics

The software architects and developers need a robustness metric to compare the robustness of two different software designs for the same mission. First, we argue that simply counting the edges of the call graph (in some papers, call graph is simply referred as dependency graph) is not appropriate to serve the purpose of robustness metric for a software system.

For a mission-critical software system, when we are talking about the robustness of the system, we are always discussing it in the context of a certain mission-critical requirement. That is, we are more concerned about what kind of failure will bring down the whole system. In the DMF terminology, what kind of failures will be transitively or hierarchically mapped to the system failure. Here we use the term system failure, which is defined as the failures of the union of all the components that implements the mission critical requirements. For example, in the car control testbed, the developers may want to investigate the system's robustness defined as collision free, or as collision free and no fail-safe stop, or as collision free, no fail-safe stop, and no trajectory following error. Obviously, with respect to robustness concerns of different criticalities, there are different robustness metrics. Therefore, we have the following definition:

Definition 12: *The robustness requirement of a robust real-time system is denoted as:* $robustness = noFailure(Failure1, Failure2, etc)$.

The robustness metric is a subset of the union of all the components' failure semantics $\bigcup (Comp_i, FS_{Comp_i})$ *that can transitively or hierarchically lead to the system-level failure* $Failure1$*, or* $Failure2$*, etc. We denote robustness metric of the robust real-time system with respect to a certain design* A *as* RM_A.

Given two designs A and B of the same robust real-time system, if $RM_A \subset RM_B$, we say that the robustness of the system in design A is higher than the robustness of the system in design B.

We can get the robustness metric using the following query expression:

- USAGE: **robustnessMetric**

  RETURNS: robustness metric of the current system design with respect to the specified robustness definition.

It should be pointed out that we measure and compare robustness based on set theory instead of probability analysis. Therefore, our approach serves as a complement to the traditionally used statistical robustness metric.

We now show how to compare the robustness of different designs from the perspective of robustness metric comparison with two utility examples. In the distributed car control testbed, suppose we define *robustness = noFailure(Collision).* That is, only if no collision occurs, we say that the car control system is robust.

Recall that the *CCstop* is designed conservatively to tolerate a value difference (e.g., 5 distance units) between the location information contained in the location packet and the actual locations of the cars. That is, if *CCstop* discovers that another car is within 5 distance units (+ the length of two cars and other similar considerations) away from itself, a STOP command will be sent out to the actuator to avoid potential collisions. Therefore, in the old design of the car control testbed, any location value error larger than 5 distance units may lead to car collision. But in the new design, a Kalman filter is installed at the site of each car controller, and a location value error larger than 10 distance units can be detected. The *CCstop* can perform a fail-safe stop either when another car is within 5 distance units from itself or when a location value error (larger than 10 distance units) is detected.

Example 10: In the original design, **robustnessMetric** will return the query result RM_{old} as:

$\{VS\ ,\ LocationError\langle 5,\ Inf\rangle\}$

$\{LSvs\ ,\ LocationError\langle 5,\ Inf\rangle\}$

...

$\{CCstop\ ,\ UnableToDetect\}$

$\{CCstop\ ,\ Crash\}$

Example 11: In the enhanced design, **robustnessMetric** will return the query result RM_{new} as:

$\{VS\ ,\ LocationError\langle 5,\ 10\rangle\}$

$\{LSvs\ ,\ LocationError\langle 5,\ 10\rangle\}$

...

{*CCstop , UnableToDetect*}

{*CCstop , Crash*}

In the above two robustness metrics, *UnableToDetect* and *Crash* are failures of *CCstop* that can lead to car collision. The former is user-defined and implies that the thread *CCstop*'s collision detection algorithm fails to discover the situation that another car is already too close to itself even when no value error exists in the location packet. The latter is system-pre-defined and implies that the thread *CCstop* crashes (e.g., spin in infinite loop, deadlock, thread crash) and is unable to issue STOP command to the cars even when possible collision has been detected.

Comparing the query results of Example 10 and Example 11, it is obvious to see that $RM_{new} \subset RM_{old}$, which implies that the new design has a higher robustness than the old design.

When we define *robustness = noFailure(Collision, FailSafeStop)* or *robustness = noFailure(Collision, FailSafeStop, TrajectoryError)*, different robustness metrics will be returned by the query, and we can thus compare the two designs with respect to these different robustness concerns.

4.4 Real-time domain support of DMF

We target the application domain of DMF at robust 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 3.1.

4.4.1 Process/thread 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 [39]). But only the schedulability analysis in theory is not sufficient for the timing considerations in a robust real-time 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.

For example, consider a situation where a more critical component thread A and a less critical component thread B share the same CPU, and the schedulability analysis tells us that they are schedulable under EDF scheduling with respect to their worst-case execution time. The problem is that it is highly possible that some programming error, such as an infinite loop or even a denial of service attack, make

the component thread B overrun its budget. Then, under EDF, it is possible that component thread A misses its deadline because it cannot get hold of the CPU timely enough, unless there is OS support for budget monitoring. In this way, the less critical component B indirectly causes a failure of the more critical component A. The result is a dependency inversion because the critical component A now depends on the less critical component B through resource sharing.

We have embedded in DMF the following domain dependency 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 [29] is implemented in the real-time operating system.

- *A:Thread.BudgetOverrun* $\mapsto$ *B:Thread.DeadlineMiss*

  if *execTimeClock(OS)* $==$ *false* $\wedge$ *schedPolicy(APP)* $==$ *SCHED_FIFO* $\wedge$ *schedPriority(A)* $<=$ *schedPriority(B);*

- *A:Thread.BudgetOverrun* $\mapsto$ *B:Thread.DeadlineMiss*

  if *execTimeClock(OS)* $==$ *false* $\wedge$ *schedPolicy(APP)* $==$ *SCHED_RR* $\wedge$ *schedPriority(A)* $<$ *schedPriority(B);*

- *A:Thread.BudgetOverrun* $\mapsto$ *B:Thread.DeadlineMiss*

  if *execTimeClock(OS)* $==$ *false* $\wedge$ *schedPolicy(APP)* $==$ *SCHED_EDF;*

Here, *Thread* is a DMF pre-defined abstract component, and any application-specific component corresponding to a thread can be declared as a subcomponent of *Thread* using the ***extends*** key word (refer to Section 3.3.2). *A:Thread* and *B:Thread* represent two variables of component type *Thread*, and will be matched and replaced with any concrete subcomponent of *Thread* during dependency tracking process. *execTimeClock(OS)* is an OS-layer design feature and indicates whether the execution time clock mechanism, which is a RT-POSIX standard [29], is implemented in the underlying real-time operating system. *schedPolicy(APP)* is an APP-layer design feature. And *int schedPriority (Thread 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. We also include *SCHED_EDF* in case earliest deadline first scheduling is used in the application, even though it is not yet part of the standard.

Once the user of DMF specifies the features of the RTOS they are using, the underlying DMF reasoning engine will automatically choose the appropriate *BudgetOverrun* failure propagation rules which lead to *DeadlineMiss* failures of the corresponding components. The reasoning process will continue from the *DeadlineMiss* failure of each affected component based on the fault propagation rules specified in the APP-layer by the user.

4.4.2 Process synchronization

Another example can be given concerning process synchronization. RT-POSIX defins 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 dependency rules embedded in DMF:

- *priorityInversion(A:Thread, B:Thread) = true*
  if *synPolicy(OS) == NO_PRIO_INHERIT* $\wedge$
  sharedMutex(A, B) =/= emptySet;
- *failureSet(A:Thread) = addToFailureSet(failureSet(A), Lockup)*
  if *synPolicy(OS) == PRIO_INHERIT* $\wedge$
  sharedMutex(A, B) == {M1:Mutex M2:Mutex} $\wedge$ *M1 =/= 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 [39], and a *Lockup* failure is added to their failure sets. The user will be alerted of this newly added failure, if it has not been foreseen and has not been already included in the component's failure set. The user should thus go ahead to annotate how the *Lockup* failure of thread A will be propagated to the neighboring components.

4.4.3 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 one process is 35 ms. However,

since the system cannot support this interval, 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.

More generally, we have the following rule embedded in DMF:

*period(A:Thread) = ceiling(period(A) / clockResolution(OS)) * clockResolution(OS);*

In summary, we have embedded the OS-layer domain dependency rules in DMF based on the RT-POSIX standards. These general OS-layer dependency rules and the specific APP-layer dependency rules are integrated, and the DMF reasoning engine will perform dependency tracking on these rules, as illustrated in Figure 3.1. However, the real-time domain support of the current DMF version is still preliminary. To cover the full spectrum of RT-POSIX standard, the examination and specification of the following features are also desirable, including timeout mechanisms, real-time signals, interrupt control, device driver control, inter-process communication and shared resources. This has been pointed out as one of the future research directions in Section 7.2.

DMF provides OS-layer dependency rules constrained by OS design features (e.g., *execTimeClock(OS)*, *synPolicy(OS)*, *clockResolution(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*, *clockResolution(OS) = 10*), and the DMF reasoning engine will then choose the corresponding fault propagation rules to apply in the reasoning process.

Chapter 5

Further Discussions of Several Important Topics

5.1 A brief discussion of criticality specification of DMF

To make sure that dependencies are well formed and that critical services are protected from the faults and failures from 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 U.S. Federal Aviation Administration (FAA) established DO-178B [4] as the accepted means of certifying all new aviation software. The targeted DO-178B certification level is either A, B, C, D, or E. Correspondingly, these DO-178B levels describe the consequences of a potential failure of the software: catastrophic, hazardous-severe, major, minor, or no-effect. The criticality of a software components 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 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 higher 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 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 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 such criticality mix-ups, and a violation of the principle that requirements of different criticalities should be assigned 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 realized by this component, if requirements of multiple criticality levels are not separated and have been assigned 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 assigning 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. 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. 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. The Precautionary Principle adopted by FAA provides good guidance. In 1999, in response to inquiries about the necessity of the ban of various electronic devices during takeoff and landing and neither should cellular telephones any time during flight, the U.S. Federal Aviation Administration (FAA) commissioned a study to gather stronger evidence for or against the hypothesis that consumer electronic devices interfere with aircraft functions. The study failed to find any

evidence of this interference. Nevertheless, the FAA ruled that, in the absence of strong evidence of safety, the ban would continue in effect. Most people agree that the inconvenience of not being able to talk on the phone in flight is offset by even a small risk of an airplane crash. Similarly, the users of DMF can also decide the trade-off when annotating the fault propagation rules, between the inconvenience of annotating all potential uncertain fault propagations and the risk in case of false negative result.

5.2 A brief discussion of the scalability of DMF

As argued in Chapter 1, modern real-time systems are often developed concurrently by multiple teams. Each development team typically only knows and thus is able to annotate the potential residual failures and failure propagation properties among the software components within its own scope. They are also required to understand and annotate how the failures of other teams' components that directly interact with their own components affects them. But they are not supposed to know the details of the fault/failure propagations of components exclusively developed by other teams which they do not interact with. Therefore, system-wide composition of the local dependency annotations must be supported in order to guarantee the scalability of DMF.

In our framework, the term *component* can be defined in different layers or with different granularities. For example, it can be defined in the function/procedure layer, the class/module layer, or process/thread layer. The dependency analysis and reasoning can be performed hierarchically from the components of lower layers to components of higher layers to guarantee the scalability of the toolkit. In the logical domain, at the finest granularity, we can first investigate the fault/failure propagation properties between functions that are used in one functionality module (e.g., class in object-oriented programming), and derive the failure set of this specific functionality module based on the composition/interaction of these functions. Then in the execution domain, we can investigate the fault/failure propagation properties between the functionality modules that are used in one process/thread, and derive the failure set of each execution unit (e.g., thread). Finally, we can investigate the fault/failure propagation properties across process/thread boundaries, and reason about the robustness of the whole system. In this hierarchical composition process, functions/classes and processes/threads are defined as component at different stages.

The ION CubeSat case study presented in Chapter 3 tracks the dependency relations between different *classes*, while the distributed car control testbed case study presented in Chapter 4 tracks the dependency relations between different *threads*. Another case study on Etherware middleware [15] tracks the dependency relations between different*functional modules*.

In detail, the hierarchical dependency management process can be performed as follows:

- Each development team specifies the fault/failure propagation and composition rules among the components (e.g., in the function/procedure layer) developed by themselves, and derive the potential failure set of the functionality module (e.g., in the class/module layer) within its scope based on the composition rules.

- Each development team sends the potential failure set of the modules within its scope and the specification of fault/failure propagation rules where the modules within its scope are destination components (e.g., in the class/module layer) to a centralized higher-level development manager or development team leader.

- The higher-level development manager or development team leader reasons about the potential failures and robustness of the system in the next higher layer (e.g., in the process/thread layer) based on the information sent by all the lower-level teams and annotates the appropriate fault propagation rules for this layer.

- This dependency tracking process can be done iteratively until the system layer is reached and a complete set of fault propagation rules for each layer exists.

For example, the car control system has been developed by four people, each one in charge of vision server, location server, trajectory manager, and car controller separately. According to the hierarchical dependency management process presented above, each person can first perform dependency tracking and reasoning among the C functions within his own scope, and derive the failure sets for each functionality module within his scope (e.g., the model-based state estimation module, the control command calculation module, and the collision detection and avoidance module in car controllers). Then each person sends the derived potential failure set and the specified fault/failure propagation rules where the module within his scope is the destination component to the development team leader.

The development team leader then reasons about the system robustness based on the information collected from all the four developers.

The formally derived and implemented dependency composition rules of DMF, i.e., Theorem 1-6 of Chapter 2, 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 from the local annotations of failure propagation based on the fault/failure propagation theorems and depend/USE relation propagation theorems. In addition, since most updates and fixes occur at the lowest layer, once the fault propagation rules for the process/thread and system layers are defined, they will rarely change.

However, it has to be pointed out that it is not an easy job to keep the dependency specifications consistent among all development teams. For example, even though DMF has a system-defined key word *Crash* for crash failure, and *DeadlineMiss* for deadline miss failure etc, some users may prefer to use *fCrash* and *fDeadlineMiss* instead. This is fine provided that *fCrash* and *fDeadlineMiss* are used consistently among all the development teams. However, if team A chooses to use *fCrash* and team B chooses to use *Crash* (or self-defined *failCrash*), and they are not informed of this inconsistency by each other, problem may occur. The *A.fCrash* failure will not be matched to the left side of the failure propagation rule $A.Crash \mapsto B.***$ even though semantically they should have been matched. Thus an existing failure propagation path is overlooked due to inconsistency of dependency specifications among the teams.

In general, this is a multiple model consistency problem, which is very hard to tackle with in reality.

In Chapter 14 *Maintaining Model Integrity* of his book [27], Eric Evans writes: *We need ways of keeping crucial parts of the model tightly unified. None of this happens by itself or through good intentions. It happens only through conscious design decisions and institution of specific processes. Total unification of the domain model for a large system will not be feasible or cost-effective...*

Even a single team can end up with multiple models. Communication can lapse, leading to subtly conflicting interpretations of the model. Older code often reflects an earlier conception of the model that is subtly different from the current code...

When a number of people are working in the same bounded context, there is a strong tendency for the model to fragment. The bigger the team, the bigger the problem, but as few as three or four people can encounter serious problems. Yet breaking down the system into even smaller contexts eventually loses a valuable

level of integration and coherency.

The solution to this unavoidable tendency of inconsistency among multiple specifications is more of an organizational matter than technical matter, as Eric Evans writes in his book [27]: *Institute a process of merging all code and other implementation artifacts frequently, with automated tests to flag fragmentation quickly. Relentlessly exercise the ubiquitous language to hammer out a shared view of model as the concepts evolve in different people's heads.*

5.3 A brief discussion of the evolvability of DMF

When a new component is added to the system, the DMF user only needs to specify this component's criticality, failure set, and fault propagation rules concerning immediately interacting components. 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 the corresponding criticality, failure set, and fault propagation rules concerning immediately interacting components. In short, 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 burden its users.

However, there is another issue that is more critical and difficult to tackle with, i.e., the *software artifact correspondence* problem. It is easy to specify various properties or invariants, such as using consistent units, in architecture and design documents. How to ensure source codes compliance with architecture specification is a major scientific and technological challenge in software producibility.

In the context of DMF toolkit, it would be perfect if the dependency specification can be synchronously updated as the system implementation (source code) is changed. Put it in another way, it would be ideal if the component interaction and failure propagation rules specified in the dependency specification file (or annotations if the dependency specification is embedded in the source code) exactly correspond to the ever changing software system implementations.

However, this ideal goal is too ambitious to achieve without manual effort involved. The majority of software system design properties cannot be directly extracted by statically analyzing the source code, e.g., the metric system units implicitly used, the Kalman filter delay tolerance in the distributed car control

testbed. These design properties either have to be explicitly specified or can only be obtained through off-line experimentation. Similarly, most failure propagation rules can only be explicitly specified by the domain expert or the developer of the specific software modules. There are techniques trying to automatically generate these fault propagation rule by data mining or fault injection [12, 13, 25, 32]. But they can only generate a small subset of possible failure propagation rules.

Another cause of the inconsistency between dependency specifications and system implementations is the reluctance of the developers to go back and revise the dependency specification after they make changes to the source files. This leads to the fact that the dependency specification will sometimes be out-of-dated when the implementation is constantly modified and improved.

In general, the solution to the notoriously difficult software artifacts correspondence problem is rather more organizational than technical. As Eric Evans writes [27]: *It is essential to have some process of* CONTINUOUS INTEGRATION. CONTINUOUS INTEGRATION *means that all work within the context is being merged and made consistent frequently enough that when splinters happen they are caught and corrected quickly.* CONTINUOUS INTEGRATION*, like everything else in domain-driven design, operates at two levels: (1) the integration of model concepts and (2) the integration of the implementation.*

Concepts are integrated by constant communication among team members. The team must cultivate a shared understanding of the ever-changing model...Meanwhile, the implementation artifacts are being integrated by a systematic merge/build/test process that exposes model splinters early.

Even though it is impossible to totally automatically generate the dependency specifications out of the implementation, even though manual effort has to be involved, we argue that it is a worthwhile effort to manually keep consistency between dependency specification and system implementations. First of all, we claim that just the effort of annotating the criticality, failure set, and failure propagation rules of a system helps developers to be more aware of the system's failure propagation behaviors and be alerted to some failures or propagations overlooked before. Then, with the aid of DMF's *Dependency Query Commands*, 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 placed 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 interacting 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.

Even though it is impractical to totally solve this *software artifact correspondence* problem, or *tight binding between specification and implementation* problem in our terminology, DMF proposes a partial solution trying to urge the user to enforce the consistency between dependency specification and system implementations.

Our solution consists of two main strategies. First, DMF allows the user to specify the source files associated with one or more failure propagation rules. Each time the dependency specification is reloaded, the underlying DMF parser will check whether the latest modification date/time of the associated source files are more recent than that of the dependency specification file. If it is, a warning will be shown to the user, and the user is alerted to look into the specific failure propagation rules to see whether they are still valid after the modification of the source files.

Second, DMF enables parameterized failure semantics and conditional failure propagation rules where the condition is a boolean expression of software design properties, failure semantics parameters, etc. In this way, when the software design changes, the DMF user only needs to update the values of the corresponding software design properties in the COMPONENT PROPERTY VALUES section or the INTERACTION PROPERTY VALUES section of the dependency specification. The tool will then automatically apply the correct failure propagation rules in the dependency reasoning. This has been demonstrated in Section 4.2.

Notice that this is one form of *tight binding* in design layer: (1) DMF realizes the tight binding between failure and component, i.e., each failure is bound with a component, and each failure propagation rule is bound with a destination component; (2) DMF realizes the tight binding between dependency specification and software system design properties, i.e., each failure propagation rule can be specified as conditional upon a boolean expression of system design properties.

The *tight binding* discussed in this specific section is the *software artifact correspondence* problem, i.e., the binding between system **design** properties and system **implementation** properties. That is, the dependency specifications, the failure propagation rules, and the system design properties are desired to be up-to-date with the constantly changing system implementations (source files). As we said, this has to involve manual effort and is more of an organizational matter than technical matter as pointed out in [27].

In a word, there are two layers of *tight binding* here. One is the binding between **system design** and **system implementation**, for which we have a partial solution. The other is the binding between **dependency specification** and **system design** (e.g., components, interactions, design properties), for which

DMF has a sufficiently strong support.

Chapter 6

Related Works

6.1 Related works in dependency management

In addition to Cristian's work([16,17]) on dependency management in distributed systems, there are a lot of researches conducted in this area. In this chapter, we list several representative works that target different goals and aspects of dependency management and fault analysis.

Kon et al. [34] classify dependencies between software components into two categories: prerequisite and dynamic dependencies between loaded components in a running system. Their goal is to implement software components that can configure themselves and adapt to the highly dynamic environments. Our focus is the fault/failure containment and propagation analysis and dependency tracking, thus the dependency classification and management mechanisms are fundamentally different between Kon's work and ours.

Keller et al. [33] introduce the concept of dependency strength, which is defined as how strongly the dependent component depends on the antecedent resource. They classify dependency strength into none, optional, and mandatory. While *none* simply states that there is no interaction between the two components at all, *optional* and *mandatory* seem similar with our *USE* and *depend*. However, they did not provide further explanation nor a formally defined measure of dependency strength. Our classification of dependency strength extends their informal notions by providing a formally defined strength measure and the notion of failure semantics mapping.

The problem how to acquire the dependency information is another crucial issue in dependency management, although this is not fully within the scope of this thesis. Brown et al. [12] apply active perturbation, i.e., injecting faults in a controller manner and observing the behavior of the components, to identify and characterize dynamic dependencies between system components in distributed application environments such as e-commerce systems where their dependency graph

is unknown or partially known. Kar et al. [32] present a pragmatic repository-based approach to generate appropriate service dependency information from the system configuration repositories. Ensel [25] presents an approach applying artificial neural networks to automatically determine whether two real world objects have a relationship or not over time. In our work, we view dependency from the perspective of robustness criterion, i.e., fault/failure propagation across component boundaries and the well-formed dependency checking. Our tool can evaluate system-wide dependency relations from local annotations. However, with respect to the local annotations, it is the developer's responsibility to annotate the fault propagation properties within his scope, i.e., among the components developed by his own development team and the immediately interacting components developed by other teams. Currently, our tool is not able to automatically acquire this kind of information without the user's annotations.

Ensel et al. [26] describes an approach for applying XML, XPath, and RDF to the problem of describing, querying, and computing the dependencies among services in a distributed computing system. A key contribution of this paper is a web-based architecture for retrieving and handling dependency information from various managed resources. Alda et al. [10] present a component architecture FREEVOLVE. Hasselmeyer [30] provides a dependency management architecture based on Jini technology. Their application domain is application service over service provider networks or peer-to-peer networks and they try to provide a dependency management middleware. While these works focus on implementing a dependency management middleware to integrate with the running system in software operation phase, we are more concerned about developing a unified theoretical framework and implementing a prototype toolkit to aid the developers to improve the system robustness primarily in software design phase.

Nett et al. [42] identify managing dependencies as a basic problem for the design of fault-tolerant algorithms, and cope with this problem by the specification and realization of a distributed dependency management systems. Realized as a generic software tool, it can be used as a customizable component that eases the design and implementation of existing and future algorithms. Their work is concerned about the fault-tolerant algorithms, while our work is concerned about the fault-tolerant systems.

The concept of operational profiles [41] in software reliability engineering can be properly integrated into our framework, helping us have a clear picture of the possible failures and failure handling mechanisms. Since operational profile is concerning a set of disjoint operational alternatives with the probability that each will occur, they are more useful in statistical analysis of the system dependability.

Our theoretical framework is analytical instead of statistical.

6.2 Related works in fault analysis

Many different types of fault analysis have been proposed and are in use. Some differ primarily in their names, whereas others truly have unique and important characteristics. In Chapter 14 of *Safeware: System Safety and Computers* [37], Nancy Leveson presents a detailed overview of the basic features, the life-cycle phase to which it applies, and a brief evaluation of each major fault analysis techniques. Below is a list of the fault analysis techniques discussed in [37]:

- Checklists
- Hazard indices
- Management oversight and risk tree analysis
- Event tree analysis
- (Software) fault tree analysis
- Cause-consequence analysis
- Hazards and operability analysis
- Interface analysis
- Failure modes and effects analysis
- Failure modes, effects, and criticality analysis
- Fault hazard analysis
- State machine hazard analysis
- Task and human error analysis techniques

Among them, the technique most related to our research is software fault tree analysis. Fault tree analysis [36–38, 47] has been widely used in system reliability studies, offering the ability to focus on an event of importance and work to minimize its occurrence or consequence. When changes are infrequent, e.g., a hardware system, the fault tree analysis is very effective in analyzing system reliability or safety issues. However in constantly changing software systems, lack of explicit binding between the tree structure and the actual system design makes it hard to keep track design changes, as illustrated in Figure 1.3. While in our work, the

importance of tight binding between the dependency specification and software system designs has been adequately addressed in the following way:

- Each failure semantics is tightly bound to the component it belongs to (in the general form $B.FS_B^S \mapsto A.FS_A^{S'}$, i.e., $map(FS_B^S, B, A) = FS_A^{S'}$), thus we are able to reason about the fault/failure propagation across component boundaries. In software fault tree analysis, however, the notion of component boundary is blurred, because the fault/event is more of a high level abstraction in view of the whole system than of a specific failure explicitly attached to a certain component.

- The dependency expression (i.e., failure propagation rules) is a function of the annotated properties of components and their interaction protocols. That is, there is a tight binding between the actual design of software component interactions and the failure propagation descriptions. Therefore, once the software design parameters change, the underlying DMF reasoning engine will automatically update the failure propagation rules that should be applied in dependency tracking, and automatically generate the fault/failure propagation or dependency tracking query results upon query submissions.

The International Society for Automotive Engineers (SAE) has produced *Architecture Analysis and Design Language* (AADL) [5], which is a textual and graphical language supporting model-based engineering of embedded real-time systems and has been approved and published as SAE Standard AS-5506 by SAE in November 2004. AADL has an *Error Model Annex* that extends the core language to support reliability modeling. The MetaH toolset [6] of Honeywell, the starting point for the AADL standard, has shown that AADL is a very useful architecture description language for error modeling. To the best of our knowledge, AADL/MetaH is the best toolkit for error modeling and analysis up to date, considering its standard modeling language and analysis capabilities.

Malcolm Wallace describes a modular representation and compositional analysis of a system's hardware and software components, called *Fault Propagation and Transformation Calculus* (FPTC) [48]. He shows, given an architectural description of how components are combined into a whole system, together with an FPTC expression of each component's failure behavior, how the failure properties of the whole system can be computed automatically from the individual FPTC expressions. From a safety point of view, this provides some idea of robustness: the system's capability to withstand certain types of failures in individual components. It also provides a way to understand how and where to develop fault accommodation within an architecture.

The objective and accomplishments of [48] are quite similar with our *Dependency Management Framework* from the first glance. However, the building block of FPTC is the statically schedulable code unit, which represents a single sequential thread of control. The basic connection between units are therefore the communication protocols between these threads. While with DMF, the building block is much more flexible, as discussed in Section 5.2 and demonstrated by the ION CubeSat case study (refer to Chapter 3), where the building block of the dependency specification is a *class* in object-oriented programming paradigm.

Comparing to AADL/MetaH and FPTC, our toolkit has several distinctive characteristics. Like our work, AADL/MetaH and FPTC also addresses the binding between failures and the associated components, more or less. Yet the tight binding between dependency expressions (i.e., fault/failure propagation rules) and the software system designs has not been fully addressed. Therefore, like software fault tree analysis, AADL/MetaH and FPTC is also not capable to keep track of the software design changes when fault analysis is performed.

In addition to the tight binding, DMF also enables parameterized failure semantics, which AADL/MetaH, FPTC or software fault tree analysis do not offer. As shown by the application examples in Chapter 4, parameterized failure semantics significantly enriches the fault propagation model. For example, in AADL/MetaH, FPTC or software fault tree analysis, it is difficult to express the following fault propagation property: *When the number of consecutive location packet deadline misses exceeds k (k is not a constant but a variable that should be updated when software design or environment changes), there is a fail-safe stop failure of car controllers.*

We now perform a comparison of the toolkit functionalities. In the query *rootCause(Component, Failure)*, the *Component.Failure* is like the *top event* of the fault tree, and the automatically generated root-cause failure set together with the propagation paths can be looked as a textual representation of the graphical representation in fault tree. That is, we can do what software fault tree does in backward analysis. We can also perform forward analysis, as AADL/MetaH and FPTC can do. Besides these low-level dependency tracking features, our toolkit also supports high-level dependency (total depend/partial depend/USE) tracking, which is very important to make a mission-critical system robust, i.e, to ensure essential services in spite of faults and failures in useful but non-essential components. High-level dependency tracking and well-formed dependency checking are not supported in software fault tree analysis, FPTC or AADL/MetaH.

With respect to the quantitative analysis, i.e., the quantification of the probability of occurrence of a failure, our toolkit does not support the statistical analysis,

which is an important feature in fault tree analysis and AADL/MetaH. However, we argue that numbers for probability of software faults are not yet as available as hardware in practice. But it should be pointed out that we measure and compare robustness based on set theory instead of probability analysis. Therefore, our approach serves as a complement to the traditionally used statistical dependability metric, as used in fault tree analysis and AADL/MetaH.

Currently, DMF does not support dependency composition expressions similar to dynamic fault tree gates, such as *Priority-AND*, *Sequence-Enforcing*, and *Cold-Spare*, which we think are common in hardware systems but not quite typical in software systems.

Notice that in addition to the DMF toolkit, we also formalize a unified theoretical framework for dependency management, where concepts such as failure semantics and dependency strength and composition rules for both high-level and low-level dependency relations have been formally defined or derived. The definitions and theorems in the theoretical framework serves as the solid theoretical foundation of the underlying dependency reasoning engine. To the best of our knowledge, this is the first time that such a unified theoretical framework for dependency management is provided.

Finally, it must be pointed out that dependency tracking, like fault tree analysis, is limited to the tracking and analysis of the propagation of potential faults and failures under a given fault model. They cannot be used to detect if a component has internal defects, which is the task of proof of correctness, model checking [24] and testing.

Chapter 7

Conclusions and Future Research Directions

7.1 A brief summary of accomplishments

In this thesis, we have provided a theoretical framework as well as a many-featured prototype toolkit *dedicated to dependency tracking and reasoning* in robust real-time systems. The current accomplishments are summarized as below:

Theoretical Framework of DMF [20, 21]:

1. We extend the traditional failure semantics to parameterized failure semantics, which enables application-specific software failure semantics specifications.

2. We provide the formal definition of dependency strength based on the notion of failure semantics mappings, and classify the dependency relation into three categories: Total Depend, Partial Depend, and USE.

3. We formally derive transitive and hierarchical composition rules for both high level dependency relation tracking and low level failure propagation reasoning based on the definitions.

Prototype Toolkit of DMF [19, 21]:

1. The dependency specification language of DMF is simple and expressive, e.g., it supports component inheritance, Boolean expressions of failure propagation rules, and parameterized failure semantics and dependency expressions.

2. The dependency query commands of DMF support both high-level dependency queries (dependency inversion check) and low-level dependency queries (the detailed failure propagation path resulting in dependency inversion).

3. We sufficiently address the challenges of scalability and evolvability by only asking the users to specify the failure propagation properties between immediately interacting component.

We have done several case studies to test the usability and scalability of DMF, i.e., the ION CubeSat [19], the convergence lab's distributed car control testbed [21], the Etherware middleware [15], and eSimplex testbed [20].

In addition to these papers on dependency management, this author has other three papers on fault-tolerance software architectures [22, 23, 40]. The first two papers deal with software reliability in view of formal specification and validation based upon Actor model [9] and Real-Time Maude [7,31]. The third paper focuses on co-design of control and schedulability. The case studies in these three papers are performed on the eSimplex system [45,46].

7.2 Future research directions

There are a bunch of interesting and challenging research topics to continue this work. We summarize them into 10 future research directions.

1. The real-time domain support of the current version of DMF is rather preliminary. It only covers very basic scheduling and resource sharing failures and failure propagation rules. However, there are plenty of challenging research directions concerning real-time domain-specific dependency management. For example: (1) Timing dependency. The late arrival of expected message may cause the failure of a loop and cascade into other loops. Modern software frequently uses indirect communication methods such as publication and subscription service offered by middleware. The ability to automatically track and see the timing dependency of a complex distributed real-time system is important. (2) Resource sharing dependencies. This includes the shared use of (2.1) Hardware resources such as CPU, memory, I/O channels and specialized devices. This should be handled by dependency tracking and integrate it with schedulability analysis; (2.2) Software library, OS and middleware services. This requires static analysis to track and to ensure rules of usages are enforced. It also requires fault injection tests for the efficacy of fault containment mechanisms; (2.3) Run time checks for faults that cannot be eliminated by testing and static analysis. This requires detailed fault modeling and analysis so that we know which one must be checked at runtime.

2. Dependency management for distributed systems. This will bring up a lot of interesting research issues specific to distributed and networked systems. The support for dependency management in distributed systems of the current version of DMF is still pretty naive, even though we have done case

studies on the dependency management of the distributed car control testbed and Etherware middleware.

3. Further improvement of the user-friendliness of DMF - such as a more user-friendly graphical user interface, and a better syntax parser for the dependency specification language. A viable platform to implement this is Eclipse [8], which has formed an independent open eco-system around royalty-free technology and a universal platform for tools integration.

4. Integration with other tools to, at least partially, automatically generate fault propagation paths. 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, for example, by exploiting the fault injection techniques.

5. Support for more complex Boolean operators, including dynamic Boolean expressions such as priority AND (the output occurs if and only if all input events occur in a particular order, but the input events are not constrained to occur in this particular order), and sequence-enforcing (the output occurs if and only if all input events occur in a particular order, and the input events are constrained to occur in this particular order). These dynamic Boolean operators, in addition to the traditional AND/OR Boolean operators, will be very useful for dynamic dependency tracking and management in run-time, which is listed as the next research direction.

6. Run-time dependency tracking and management. The current version of DMF is designed for dependency tracking and management at design and development time. A run-time dependency tracking and management framework would be very helpful for debugging, fault diagnostics, and fault recovery. A middleware may be implemented to integrate with well-know fault tolerance techniques. Take fault recovery for example, given a component failure that has been observed, the DMF user can perform root cause analysis to determine which components possibly lead to this failure, and then only the identified components are recovered, say by restarting the process. A recovery decision and management middleware will be implemented to interact with DMF and identify the minimal subset of components that should be recovered in order to minimize MTTR (mean time to recovery). This can be completed on top of *Process Resurrection* [35], work done by another RTSL group member, Kihwal Lee.

7. Further theoretical framework - the most ambitious goal may be deriving the domain-specific dependency relation composition and propagation theorems in a similar fashion of our six general theorems as presented in Chapter 2. Notice that a solid theoretical foundation is an indispensable part of the research in order to make the prototype toolkit theoretically sound.

8. Dependency annotations embedded in the source code, or even extracted from the source code. This part is very challenging and need further investigation on its feasibility because we have to then set up a programming paradigm for the DMF users if they want to make use of such a feature. But then users may be reluctant to change their programming habit to adapt to our specific tool.

9. Security and dependency management. Robustness cannot be totally detached from security concerns. This is a very promising and interesting future research topic.

10. Statistical analysis or even simulation of failure propagation behaviors. That is, given the failure rate of each component, how to estimate the probability for a certain system-level failure to occur within a certain period of time? Or what's the availability and reliability of the system? Can we do a simulation purely on the fault propagation behaviors?

Bibliography

[1] http://sunnyday.mit.edu/accidents/Ariane5accidentreport.html.

[2] http://maude.cs.uiuc.edu.

[3] http://cubesat.ece.uiuc.edu/.

[4] http://www.stsc.hill.af.mil/crosstalk/1998/10/schad.asp.

[5] http://www.aadl.info/.

[6] http://www.htc.honeywell.com/metah/.

[7] http://www.ifi.uio.no/RealTimeMaude/.

[8] http://www.eclipse.org/.

[9] Agha, G., Mason, I., Smith, S., and Talcott, C. A foundation for actor computation. *Journal of Functional Programming 7* (1997), 1–72.

[10] Alda, S., Won, M., and Cremers, A. B. Managing dependencies in component-based distributed applications. In *Scientific Engineering for Distributed Java Applications: International Workshop, FIDJI 2002. Vol. 2604 / 2003 of LNCS* (2003), Springer-Verlag, pp. 143–154.

[11] Arber, L. A dependency analysis of the ion cubesat. Master's thesis, University of Illinois at Urbana-Champaign, 2006.

[12] Brown, A., Kar, G., and Keller, A. 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).

[13] Candea, G., Delgado, M., Chen, M., and Fox, A. Automatic failure-path inference: A generic introspection technique for internet applications. In *Proceedings of the 3rd IEEE Workshop on Internet Applications (WIAPP)* (San Jose, CA, USA, June 2003).

[14] Clavel, M., Duran, F., Eker, S., Lincoln, P., Marti-Oliet, N., Meseguer, J., and Quesada, J. Maude: Specification and programming in rewriting logic. *Theoretical Computer Science* (2001). http://maude.cs.uiuc.edu/.

[15] Crenshaw, T., Robinson, C. L., Ding, H., and Kumar, P. R. A pattern for adaptive behavior in safety-critical, real-time middleware. In *Proceedings of the 27th IEEE Real-Time Systems Symposium (RTSS'06)* (Rio de Janeiro, Brazil, December 2006).

[16] Cristian, F. Understanding fault-tolerant distributed systems. *Communications of the ACM 34*, 2 (1991), 56–78.

[17] Cristian, F. Abstractions for fault-tolerance. In *Proceedings of the IFIP 13th World Computer Congress. Volume 3 : Linkage and Developing Countries* (Amsterdam, The Netherlands, 1994), K. Duncan and K. Krueger, Eds., Elsevier Science Publishers, pp. 278–286.

[18] Dabrowski, M. The design of a software system for a small space satellite. Master's thesis, University of Illinois at Urbana-Champaign, 2005. http://www.interave.net/projects/ion_thesis_2005/.

[19] Ding, H., Arber, L., Sha, L., and Caccamo, M. The dependency management framework: A case study of the ion cubesat. In *Proceedings of the 18th Euromicro Conference on Real-Time Systems (ECRTS'06)* (Dresden, Germany, July 2006).

[20] Ding, H., Lee, K., and Sha, L. Dependency algebra: A theoretical framework for dependency management in real-time control systems. In *Proceedings of the 12th IEEE International Conference on Engineering of Computer Based Systems (ECBS'05)* (Greenbelt, Maryland, USA, April 2005).

[21] Ding, H., and Sha, L. 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).

[22] Ding, H., Zheng, C., Agha, G., and Sha, L. Automated verification of the dependability of object-oriented real-time systems. In *Proceedings of the 9th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS'03F)* (Capri Island, Italy, October 2003).

[23] Ding, H., Zheng, C., Sha, L., and Agha, G. Specification and validation of fault-tolerant software architectures based on actor model. In *Proceedings of the 15th International Conference on Software Engineering and Knowledge Engineering (SEKE'03)* (San Francisco, California, USA, July 2003).

[24] Edmund M. Clarke, J., Grumberg, O., and Peled, D. A. *Model Checking*. MIT Press, 2000.

[25] Ensel, C. Automated generation of dependency models for service management. In *Proceedings of Workshop of the OpenView University Association (OVUA'99)* (June 1999).

[26] Ensel, C., and Keller, A. 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).

[27] Evans, E. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley Professional, 2003.

[28] Graham, S., Baliga, G., and Kumar, P. R. Issues in the convergence of control with communication and computing: Proliferation, architecture, design, services, and middleware. In *Proceedings of the 43rd IEEE Conference on Decision and Control* (December 2004).

[29] Harbour, M. G. Real-time posix: An overview.

[30] Hasselmeyer, P. Managing dynamic service dependencies. In *Proceedings of the 12th International Workshop on Distributed Systems: Operations and Management - DSOM 2001* (Nancy, France, October 2001).

[31] Ölveczky, P. C., and Meseguer, J. Real-time maude: A tool for simulating and analyzing real-time and hybrid systems. In *Proceedings of 3rd International Workshop on Rewriting Logic and its Applications (WRLA'00), volume 36 of Electronic Notes in Theoretical Computer Science* (2000), pp. 361–383.

[32] Kar, G., Keller, A., and Calo, S. 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)* (Honolulu, HI, USA, April 2000), pp. 61–75.

[33] Keller, A., and Kar, G. Dynamic dependencies in application service management. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications* (Las Vegas, NV, USA, June 2000).

[34] Kon, F., and Campbell, R. H. Dependence management in component-based distributed systems. *IEEE Concurrency 8*, 1 (January 2000), 26–36.

[35] Lee, K., and Sha, L. Process resurrection: A fast recovery mechanism for real-time embedded systems. In *Proceedings of the 11th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'05)* (San Francisco, California, USA, March 2005).

[36] Lee, W. S., Grosh, D. L., Tillman, F. A., and Lie, C. H. Fault tree analysis, methods, and applications: A review. *IEEE Transactions on Reliability R-34* (August 1985), 194–302.

[37] Leveson, N. G. *Safeware: System Safety and Computers*. Addison-Wesley, 1995.

[38] Leveson, N. G., Cha, S. S., and Shimeall, T. J. Safety verification of ada programs using software fault trees. *IEEE Software 8*, 4 (July 1991), 48–59.

[39] Liu, J. W. S. *Real-Time Systems*. Prentice Hall, 2000.

[40] Liu, X., Ding, H., Lee, K., Sha, L., and Caccamo, M. Feedback based real-time fault tolerance – issues and possible solutions. *ACM SIGBED Review, Special Issue on Feedback Control Implementation and Design in Computing Systems and Networks 3*, 2 (2006).

[41] Musa, J. D. Operational profiles in software reliability engineering. *IEEE Software 10*, 2 (March 1993), 14–32.

[42] Nett, E., Mock, M., and Theisohn, P. Managing dependencies: A key problem in fault-tolerant distributed algorithms. In *Proceedings of 27th International Symposium on Fault-Tolerant Computing (FTCS'97)* (1997), pp. 2–10.

[43] Patterson, D. A., Brown, A., Broadwell, P., Candea, G., Chen, M., Cutler, J., Enriquez, P., Fox, A., Kiciman, E., Merzbacher, M., Oppenheimer, D., Sastry, N., Tetzlaff, W., Traupman, J., and Treuhaft, N. Recovery-oriented computing (roc): Motivation, definition, techniques, and case studies. Tech. rep., UC Berkeley Computer Science Technical Report UCB//CSD-02-1175, March 2002.

[44] Rosu, G., and Havelund, K. Rewriting-based techniques for runtime verification. *Automated Software Engineering 12*, 2 (2005), 151–197.

[45] Sha, L. Dependable system upgrade. In *Proceedings of 19th IEEE Real-Time Systems Symposium (RTSS'98)* (1998), pp. 440–448.

[46] Sha, L. Using simplicity to control complexity. *IEEE Software 18*, 4 (July/August 2001), 20–28.

[47] Wallace, D., Towhidnejad, M., and Coleman, C. Software fault tree analysis. Tech. rep., Software Assurance Technology Center, NASA, Decemeber 2003.

[48] Wallace, M. Modular architectural representation and analysis of fault propagation and transformation. *Electronic Notes in Theoretical Computer Science 141*, 3 (2005), 53–71.

Author's Biography

Hui Ding was born in Rizhao City, Shandong Province, China. He received his two B.E. degrees (with honors of excellent student and best thesis) in 1998, one from Civil Engineering Department and the other from Automation Department of Tsinghua University, China. He earned his M.E. degree in 2001, from Institute of Automation, Chinese Academy of Sciences. He was an intern of Microsoft Research, China during the summer of 1999. Since 2001, he has been with the Department of Computer Science, University of Illinois at Urbana-Champaign. He will be joining UBS Investment Bank upon graduation.