Dependency Algebra: A Tool for Designing Robust Real-Time Systems

Hui Ding Lui Sha
Department of Computer Science
University of Illinois at Urbana-Champaign
201 North Goodwin Avenue, Urbana, Illinois 61802
{huiding,lrs}@cs.uiuc.edu

Abstract

A robust system is one that can ensure essential services in spite of faults and failures in useful but non-essential components. Unless we can ensure that critical services can only USE but not depend on less critical components, a seemingly minor fault can propagate along complex and implicit dependency chains and bring down the system.

Modern real time systems are often developed concurrently by multiple teams. A team typically only knows the dependency relations between their components and neighboring components. In addition, dependency relations will change as software components and their interactions are being modified. Therefore, how to automatically track and analyze the system wide dependency from local information is important for the development of robust real time systems. This paper presents dependency algebra - a unified theoretical framework plus a prototype toolkit for dependency management in real-time systems.

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. Given 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 system.

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 ([14, 15, 16]) 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 real time systems, this paper presents 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. 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 that capture the impact of design changes. This allows rapid analysis and comparison of different designs and modification from the perspective of robust software designs.

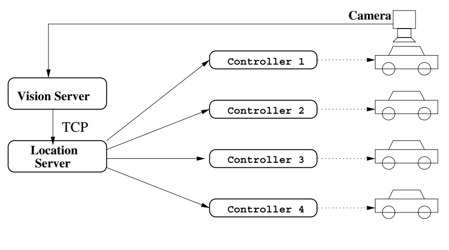
Another contribution of our approach is the extension of the generic failure semantics model to cover specific failure semantics in real time applications, e.g., parameterized timing failure and resource sharing failure. We believe that the ease of use and the attention to failure semantics in real time applications will make our theory and toolkit useful for building robust real time systems.

1.2. A conceptual framework

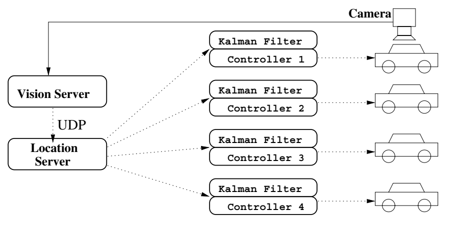
In this section, we will introduce our basic concepts and approach using the example of a vision-based distributed car control testbed [9]. It has a fleet of autonomously con-

[1]There are efforts to automatically interpret the UML design or the code and generate fault trees automatically in [16, 19]. But the challenge has been formidable.

trolled 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 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.



(a) Original Design using TCP



(b) Alternative Design using UDP and Kalman Filter

Figure 1. A vision-based car control testbed.

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(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 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(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 of UDP. The design change concerning the TCP/UDP connection and Kalman filter is shown in Figure 1.

In summary, an effective dependency management framework must be able to address the issues we have identified. But doing this is not without its own challenges. In this paper, we will tackle two main challenges. The first challenge is that currently there is a lack of formal theoretical framework for dependency management. Informal notions are confusing and can lead to errors. Specifically, we need a formal definition to measure the dependency strength as well as formally derived and proved dependency composition rules to improve the scalability. The second challenge is how to make the dependency management facilities adaptable to evolving software designs. It is important to note that a tighter binding between the dependency expressions and the actual system design is desirable for software systems whose design changes frequently. 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.

In the rest of this paper, we first present the theoretical framework of dependency algebra in Section 2. We give a brief overview and a few application examples of the prototype toolkit of dependency algebra in Section 3. We discuss related works of dependency management and fault propagation tracking in Section 4 and conclude in Section 5.

2. The theoretical framework of dependency algebra

In the following, 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 framework.

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

Cristian offers a failure classification that includes omission, timing, performance, value, and crash [5]. We adopt his classification and tailor it to better suit for real-time systems where timing is a critical consideration.

For example, in real-time systems, *omission failure* (no service delivery) and *timing failure* (service delivery misses its deadline) have the same effect for the service receiver process and we will treat them in the same failure class of *DeadlineMiss*. However, the binary fail/no-fail notion of *DeadlineMiss* does not fully capture the properties of real-time systems. As shown in Figure 1, 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. Therefore, they are user-defined 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.

As presented in the *Recovery-Oriented Computing* [18], 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*. For example, if the location server code overruns its planned real time budget (WCET) but delivers its packet as specified by the deadline, there is no corresponding failure type in the traditional model [5]. 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 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 possible failure behaviors observed by the receivers 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* $FS_{A_1} \times FS_{A_2} \times ... \times FS_{A_N}$.

The strength of failure semantics is presented in [5] 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.

For example, $FS = \{DeadlineMiss\langle 3 \rangle\} \bigcup \{Crash\}$,

$FS' = \{DeadlineMiss\langle 5\rangle\} \bigcup \{Crash\}$. 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 cannot differentiate strengths of these two failure semantics. The reason 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.

2.2. Dependency strength and classification of dependency relations

In this subsection, 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 the next two subsections, we will go beyond pairwise component dependency and derive dependency reasoning rules for various component composition patterns.

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 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 $map(FS_B^S, B, A) = FS_A^{S'}$.*

Pessimistic Annotation Assumption (PSA): In this paper, 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 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.

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

Definition 7: *Component A partially depends 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 $Dep(A, B) = TDep(A, B) \bigvee PDep(A, B)$.*

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 $USE(A, B)$.*

As argued in Section 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 whether the system has well-formed dependencies. A low level tracking of the failure semantics propagation across component boundaries enables us to reason about how faults/failures of a software component may or may not affect other interacting components.

2.3.1. The transitive failure semantics mapping. We first present the composition rules for low level dependency tracking in Section 2.3. We will come to the composition rules for high level dependency tracking in Section 2.4.

Theorem 1 – Failure semantics mapping rule 1:

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

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, the failure semantics mapping for any subset of the failure semantics FS_A can thus be obtained by proper set union operations.

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:

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

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: $A \leftarrow C$, $A \leftarrow B_i \leftarrow ... \leftarrow C$ ($i = 1, ..., N$). Notice that these service delivery paths can have common nodes (components), therefore the complete graph is not necessarily a linear chain but can be an arbitrary graph. 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 $map(FS_C^S, C, A) = FS_A^{S_0'} \bigcup FS_A^{S_1'} \bigcup ... \bigcup FS_A^{S_N'}$.

All the above three theorems can be proved based on Definition 4. Due to page limitations, we omit the proof.

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, the dependency management framework should support the following specification: *When the failure semantics of subcomponents A_1, A_2, ..., A_N truthify a certain boolean expression, component A will exhibit failure semantics FS_A^S.*

For example, in the N-version programming, *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. Take the above expression for example, 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

2.4.1. The transitive dependency relation propagation. As in last subsection, 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:

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

Proof: We prove the fourth rule here. The other rules can be proved in the same way.

Suppose the dependency strength of component B on component C is measured by $\overline{FS_C^S}$. We assume that $map(FS_C^S, C, B) = FS_B^{S_1'}$, where $FS_B^{S_1'}$ is a nonempty subset of FS_B. 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, $FS_C^{S^*}$ is transitively mapped to a nonempty subset of FS_A, and 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 $USE(A,C)$. ■

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. The following theorem is given with proof being omitted.

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 metric 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 $PDep(A,C)$.*

2.4.2. The hierarchical dependency relation propagation. We now consider the depend/USE relation propagation in complex boolean compositions. As argued in last subsection, 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.

But for conjunctive scenario, things are much more complicated. Take *k out of N* 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:

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

The sixth rule is trivial, since it is always true that $Dep(A, A_1 \bigoplus A_2) \bigvee USE(A, A_1 \bigoplus A_2)$. 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.

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 truthify $map(FS_{A_1} \times FS_{A_2}, A_1 \bigoplus A_2, A) = \emptyset_A$, which is the condition in order to truthify $USE(A, A_1 \bigoplus A_2)$.

Finally, it is worthwhile to point out that Theorem 1-6 enable the scalability of the prototype toolkit for dependency algebra to be presented in Section 3. 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 and evaluated from the *local* information of failure semantics mapping annotated by each development team, based on Theorem 1-6.

3. The prototype toolkit of dependency algebra

We have implemented a prototype toolkit for dependency algebra. Due to page limitations, we only give a brief overview and a few application examples here. A detailed tutorial of the toolkit is too long for this paper.

Our prototype toolkit consists of two main parts: *Dependency Specification* and *Dependency Query*. The users annotate the criticality and failure semantics of the components and the fault/failure propagation rules across component boundaries using our *Dependency Specification Language*. Our toolkit will transform the annotations into the internal representation and integrate them with the underlying primitives and composition rules, such as Theorem 1-6 and domain-specific rules presented in Section 3.3. The users can then perform dependency tracking and reasoning using our *Dependency Query Commands*, and the underlying reasoning engine will reason about the system wide dependency relations based on the user annotations, composition rules as presented in Theorem 1 to Theorem 6, and domain-specific rules as presented in Section 3.3, and return the results to the users.

3.1. A brief description of the case study – the distributed car control testbed

In the examples presented in this section, 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.

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

In addition to the main thread, the threads of location server include: 1) *LSvs*: receive location packet from *VS*; 2) *LStm*: send location packet to trajectory manager; 3) *LScc[i]*: send location packet to the i-th car controller.

In addition to the main thread, the threads of trajectory manager include: 1) *TMls*: receive location packet from *LStm*; 2) *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: 1) *CCls[i]*: receive location packet from *LScc[i]*, and perform model-based state estimation in case of packet loss if Kalman filter is installed; 2) *CCtm[i]*: receive desired trajectory information from *TMcc[i]*; 3) *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*); 4) *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. Therefore, in the original design of the car control testbed as illustrated in Figure 1(a), *LocationError*$\langle 5, Inf \rangle$ cannot be tolerated and may lead to car collision.

But in the enhanced design as illustrated in Figure 1(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*$\langle 5, 10 \rangle$ cannot be tolerated and may lead to car collision.

3.2. Tight binding between dependency expressions and software system designs

In this subsection, 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. To ensure local description and tight binding with software components, we create an associated annotation file with each source file. In the annotation file associated with location server, we have

```
DESTINATION: LSvs
FAULT-MAPPING:
{
   VS.{Crash} ↦ LSvs.{Lockup}
      if  commProtocol(VS, LSvs) = TCP;
   VS.{Crash} ↦ LSvs.{Suspend}
      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.

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.

```
DESTINATION: CCls[i]
FAULT-MAPPING:
{
   LScc[i].{DeadlineMiss⟨k⟩} ↦ CCls[i].{ }
      if  k <= floor (CCls[i].T /  period(LScc[i], CCls[i]));
   LScc[i].{DeadlineMiss⟨k⟩} ↦ CCls[i].{FailToPredict}
      if  k >  floor (CCls[i].T /  period(LScc[i], CCls[i]));
}
```

Note that 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.

Our prototype toolkit can track the system design changes and automatically update the affected failure propagation rules. 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). The underlying reasoning engine will then automatically change the mapping rule to: *VS.*$\{$*Crash*$\}$ ↦ *LSvs.*$\{$*Suspend*$\}$. Note that when *LSvs* locks up, the system will lock up and all the cars will be stuck in the *fail-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.

In Example 2, the period p is an explicit design parameter and its value is explicitly annotated. So should be the delay tolerance value T of Kalman filter. There is an important difference, however. The change of location packet update period is explicit, and we can directly annotate the p's dependency on the period variable of the location server. We cannot do that in the Kalman filter code because T is not part of Kalman filter code. To solve this problem, we will create an annotation function that computes T. Furthermore, whenever the Kalman filter file is changed, our toolkit will automatically call this function to recompute T.

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⟨k⟩} ↦ CCls[i].{ }
      if  k <= 2;
   LScc[i].{DeadlineMiss⟨k⟩} ↦ CCls[i].{FailToPredict}
      if  k >   2;
}
Become
{
   LScc[i].{DeadlineMiss⟨k⟩} ↦ CCls[i].{ }
      if  k <= 4;
   LScc[i].{DeadlineMiss⟨k⟩} ↦ CCls[i].{FailToPredict}
      if  k >   4;
}
```

3.3. A brief discussion of the timing and resource sharing issues

Besides the application-specific fault/failure propagation rules annotated by the users, there are also domain-specific rules that are common for all the applications in the same domain. These general domain rules are part of the underlying reasoning engine. We target our application domain at real-time systems. In Section 2.1, we have tailored the traditionally used failure classification into our target domain. In Section 3.2, we have presented examples of different effects of *DeadlineMiss* failure with respect to two different designs. In this section, we continue to present a brief discussion of the fault propagation concerning the timing and resource sharing issues in real-time systems.

Schedulability analysis theoretically verifies whether the tasks are schedulable under a certain scheduling method, e.g., RM or EDF. 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?* We have two basic general failure mapping rules with respect to budget overrun in real-time systems (a smaller *schedPriority* value implies a higher scheduling priority).

- *A.*$\{$*BudgetOverrun*$\}$ ↦ *B.*$\{$*DeadlineMiss*$\}$
  if *schedPolicy = RM* $\wedge$
  schedPriority(A) <= schedPriority(B);

- $A.\{BudgetOverrun\} \mapsto B.\{\ \}$
  if *schedPolicy = RM* $\wedge$ *schedPriority(A)* $>$ *schedPriority(B);*
- $A.\{BudgetOverrun\} \mapsto B.\{DeadlineMiss\}$
  if *schedPolicy = EDF;*

There are other aspects concerning the predictability, schedulability, and fault propagation in the domain of real-time systems, e.g, impacts of virtual memory locking, process synchronization (BIP/PCP), timeout mechanism, real-time signal, clocks and timers, interrupt control, special I/O device drivers, resource sharing faults (resource hogging, resource corruption, resource exhaustion), etc. Due to page limitations, we cannot present a comprehensive discussion here. Another paper will be published to address these domain-specific issues in detail.

3.4. Improving the robustness and comparing the robustness of different designs

We now turn our focus on how our dependency algebra will assist the developers 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. Due to page limitations, we only present the low level dependency tracking facilities with application examples. With respect to the other three perspectives, we simply list the dependency tracking facilities supported by our toolkit, with no application examples being given.

In real time systems, we want key properties not affected by the potential faults in the services delivered by less critical components. For example, 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 comparison 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.

3.4.1. 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: **impact (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* $\Rightarrow$ ...)
- USAGE: **root-cause (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 (... $\Rightarrow$ *Component . Failure*)

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

- USAGE: **impact (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.

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

- $LSvs.\{LocationError\langle 15\rangle\}$
  Path: $VS.\{LocationError\langle 15\rangle\} \Rightarrow$
  $LSvs.\{LocationError\langle 15\rangle\}$
- $LScc[i].\{LocationError\langle 15\rangle\}$
  Path: $VS.\{LocationError\langle 15\rangle\} \Rightarrow$
  $LSvs.\{LocationError\langle 15\rangle\} \Rightarrow$
  $LScc[i].\{LocationError\langle 15\rangle\}$
- $CCls[i].\{LocationError\langle 15\rangle\}$
  Path: $VS.\{LocationError\langle 15\rangle\} \Rightarrow$
  $LSvs.\{LocationError\langle 15\rangle\} \Rightarrow$
  $LScc[i].\{LocationError\langle 15\rangle\} \Rightarrow$
  $CCls[i].\{LocationError\langle 15\rangle\}$
- SYSTEM.$\{Collision\}$
  Path: $VS.\{LocationError\langle 15\rangle\} \Rightarrow$
  $LSvs.\{LocationError\langle 15\rangle\} \Rightarrow$
  $LScc[i].\{LocationError\langle 15\rangle\} \Rightarrow$
  $CCls[i].\{LocationError\langle 15\rangle\} \Rightarrow$
  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*$\langle 15 \rangle$} to *CCls[i]* . {*LocationError*$\langle 15 \rangle$} 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 *CC-stop[i]* will perform *fail-safe stop* to avoid collision.

Example 5: In the enhanced design, *impact(VS,* {*LocationError*$\langle 15 \rangle$}*)* will return the following:

- *LSvs.*{*LocationError*$\langle 15 \rangle$}
  Path: *VS.*{*LocationError*$\langle 15 \rangle$} $\Rightarrow$
  LSvs.{*LocationError*$\langle 15 \rangle$}
- *LScc[i].*{*LocationError*$\langle 15 \rangle$}
  Path: *VS.*{*LocationError*$\langle 15 \rangle$} $\Rightarrow$
  LSvs.{*LocationError*$\langle 15 \rangle$} $\Rightarrow$
  LScc[i].{*LocationError*$\langle 15 \rangle$}
- *CCls[i].*{*LocationErrorDetected*}
  Path: *VS.*{*LocationError*$\langle 15 \rangle$} $\Rightarrow$
  LSvs.{*LocationError*$\langle 15 \rangle$} $\Rightarrow$
  LScc[i].{*LocationError*$\langle 15 \rangle$} $\Rightarrow$
  CCls[i].{*LocationErrorDetected*}
- SYSTEM.{*FailSafeStop*}
  Path: *VS.*{*LocationError*$\langle 15 \rangle$} $\Rightarrow$
  LSvs.{*LocationError*$\langle 15 \rangle$} $\Rightarrow$
  LScc[i].{*LocationError*$\langle 15 \rangle$} $\Rightarrow$
  CCls[i].{*LocationErrorDetected*} $\Rightarrow$
  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 expressive 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.

3.4.2. 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: **depend-use (Component1, Component2)**
  RETURNS: total dependency, partial dependency, or USE relation of *Component1* on *Component2*

3.4.3. Checking of well-formed dependencies. Based on the Definition 10-11, the checking of well-formed dependencies is performed using the following query command.

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

3.4.4. 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.

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* $Failure1$ *and/or* $Failure2$*, etc. We denote robustness metric of the real-time system with respect to a certain design* A *as* DM_A.

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

Based on Definition 12, the following query expression is supported by our prototype toolkit:

- USAGE: **robustness-metric**
  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.

4. Related works

Keller et al. [12] introduce the concept of dependency strength. 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*. We complement this work by providing a formally defined dependency strength metric and the notion of failure semantics mapping.

Dependency information acquisition and identification problem is tackled in [4], [7], [11], etc. Other research projects, such as [3], [8], [10], [13], and [17], attempt to cope with the dependency management problem by providing a middleware or software architecture. We emphasize our work in the fault/failure propagation aspect, and the *dependency* in these works can be categorized into the high-level dependency in our framework.

Fault tree analysis [14, 15, 16] has been widely used in system reliability studies, and is especially powerful in analyzing system reliability or safety issues for fault-tolerant hardware systems, where design changes are infrequent. Yet in constantly changing software systems, it has to be mentioned that the lack of explicit binding between the tree structure and the actual software system design sometimes makes it hard to keep track design changes. As shown in Section 3.2, we address a tight binding between dependency expressions and the annotated software designs.

AADL [1] has an Error Model Annex that extends the core language to support reliability modeling. The MetaH toolset [2], the starting point for the AADL standard, has shown that AADL is a very useful architecture description language for error modeling. Comparing to AADL and fault tree analysis, in addition to the tight binding, we have provided a unified theoretical framework, the support of parameterized dependency expressions, and both high-level and low-level dependency management facilities.

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 [6] and testing.

5. Conclusion and future work

In this paper, we have presented *dependency algebra* – a theoretical framework as well as a many-featured prototype toolkit dedicated to dependency tracking and reasoning in robust real-time systems.

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

In the future, a more friendly (graphical) user interface should be designed. Current implementation is commandline based and it is our plan to port the current implementation to Eclipse. More case studies are needed to further evaluate the usability and scalability of the toolkit.

6. Acknowledgment

This research is sponsored in part by NSF CCR-02-09202, ONR N00014-02-1-0102, and ONR N00014-05-1-0739.

References

[1] SAE Architecture Analysis and Design Language (AADL): http://www.aadl.info/.

[2] Honeywell's MetaH: http://www.htc.honeywell.com/metah/.

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

[4] A. Brown, G. Kar, and A. Keller. An active approach to characterizing dynamic dependencies for problem determination in a distributed environment. In *Proceedings of the 7th IFIP/IEEE International Symposium on Integrated Network Management*, Seattle, WA, USA, May 2001.

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

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

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

[8] C. Ensel and A. Keller. Managing application service dependencies with xml and the resource description framework. In *Proceedings of the 7th International IFIP/IEEE Symposium on Integrated Management (IM 2001)*, May 2001.

[9] S. Graham, G. Baliga, and P. R. Kumar. 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.

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

[11] G. Kar, A. Keller, and S. Calo. Managing application services over service provider networks: Architecture and dependency analysis. In *Proceedings of the 7th IEEE/IFIP Network Operations and Management Symposium (NOMS 2000)*, pages 61–75, Honolulu, HI, USA, April 2000.

[12] A. Keller and G. Kar. 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.

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

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

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

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

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

[18] D. A. Patterson and A. Brown. Recovery-oriented computing (roc): Motivation, definition, techniques, and case studies. Technical report, UC Berkeley Computer Science Technical Report UCB//CSD-02-1175, March 2002.

[19] D. Wallace, M. Towhidnejad, and C. Coleman. Software fault tree analysis. Technical report, Software Assurance Technology Center, NASA, Decemeber 2003.