Towards an Error Modeling Framework for Dependable Component-based Systems*†

Hüseyin Aysan, Radu Dobrin, Sasikumar Punnekkat
Mälardalen University, Västerås, Sweden
{huseyin.aysan, radu.dobrin, sasikumar.punnekkat}@mdh.se

Abstract

Component-Based Development (CBD) of software, with its successes in enterprise computing, has the promise of being a good development model due to its cost effectiveness and potential for achieving high quality of components by virtue of reuse. However, for systems with dependability concerns, such as real-time systems, the major challenge in using CBD will be predicting dependability attributes, or providing dependability assertions, based on the individual component properties and architectural aspects. In this paper, we propose a framework which aims to address this challenge. Specifically, we describe a revised error classification, error propagation aspects and briefly sketch how to compose error models within the context of Component-Based Systems (CBS). The ultimate goal is to perform the analysis on a given CBS, in order to find bottlenecks in achieving dependability requirements and to provide guidelines on the usage of appropriate error detection and fault tolerance mechanisms.

1. Introduction

The main advantages of CBD approach are the ability to manage complexity and the possibility to select the most suitable component among the ones that provide same functionality. However, the latter can be best achieved only if the design step incorporates rigorous analysis for this specific need. This issue becomes all the more relevant in cases where CBD is used for developing dependable systems, since one has to analyze multiple extra-functional properties as well.

Our main goal is development of a framework based on well-founded theories, while keeping industrial realities in focus, which will provide meaningful reasoning about different dependability attributes in CBS, such as reliability, availability, and timeliness based on the characteristics of the component model, specific properties of individual components and component connection scheme in a given design. Since errors are one of the main impediments for achieving dependability, this paper particularly focuses on modeling the error behavior of components and error propagation aspects in order to reason about the dependability attributes of the composed system and its failure modes.

In a recent work, Elmqvist and Nadjm-Tehrani [6] addressed formal modeling of safety interfaces and provided compositional reasoning about safety properties of composed systems. Our focus is more on reliability and timing aspects and on analytical approaches. Grunske and Neumann [8] have proposed an approach to model error behavior of composed systems by using the Failure Propagation and Transformation Notation (FPTN) for each architectural element and to construct the composed systems' Component Fault Trees (CFT) from the FPTN models to perform safety analysis. Rugina [14] proposed a framework where the Architecture Analysis and Design Language (AADL) with the features of Error Model Annex is used to create models of composed systems' error behavior. Then, these models are converted to Generalised Stochastic Petri Nets (GSPNs) or Markov Chains to be analyzed by existing tools. More recently, Joshi et.al. [10] have proposed an approach to convert error models, generated using AADL with Error Model Annex, to Fault Trees to perform further analysis.

A substantial amount of research has been conducted on reliability modeling of composed systems based on individual component reliabilities, with a recent focus on architecture based models. Most of these works assume the existence of known probabilities for error state transitions, and only a few address the error propagation aspects. On the other hand, research on dependable systems has been focussing more on fundamental system level models of errors and mechanisms for tolerating those error modes, with arguably less interest on how these models are linked to the reliability prediction models. In our view, the links between these two research directions are loosely coupled and less

*This work was partially supported by the Swedish Foundation for Strategic Research via the strategic research centre PROGRESS.

†The authors wish to thank the reviewers for their excellent comments and suggestions.

explored. Specifically in CBD, architectural decisions and specific aspects of the component model will influence the dependability evaluations. Our aim is to enable end-to-end linking from system level dependability requirements (normally specified in terms of diverse qualitative/quantitaive terms), to models for dependability evaluation and predictions of composed systems. We envision our research to provide substantial clarity and simplifications needed for CBD of applications with dependability concerns.

The rest of the paper is organized as follows: in Section 2 we state the challenges in system level modeling of error behavior, and present the principal parts of our proposed framework. Section 3 presents our revised error classification from a CBS perspective. Section 5 discusses error propagation and composition aspects and Section 6 presents conclusions and ongoing research.

2. Outline of the proposed framework

The major challenges in realization of a generalized framework for dependability evaluation of CBS are:

- diversity of dependability requirements specification
- different dependability attributes require different analysis techniques and approaches
- limited information on component properties
- lack of techniques for performing analysis with partial or evolving information
- relating usage profiles of components to target system contexts
- non-scalability of most of the formal analysis techniques to industrial-size systems

In order to enable modeling and analysis of system-level dependability behavior, the framework must include dependability requirements specification, component-level error modeling, and system-level dependability analysis, which are briefly mentioned in following subsections.

2.1 Dependability requirements specification

At this step, the system designer has to specify the dependability requirements for the target system. Due to the diversity of the dependability attributes as well as the varied industrial priorities and practices, this step is critical as it has a considerable impact on the subsequent analysis (including the choice of techniques). For instance, the reliability requirements of systems are usually defined in diverse terms ranging from qualitative to quantitative ones. A typical requirement specification can be 'System reliability should exceed 0.99999' or 'System should not have any timing failures even under a hardware node failure'. The framework must have means to accurately capture and formally specify a wide variety of such requirements, and the subsequent analysis techniques need to address these diversity.

While designing a dependable system, the goal is typically to achieve fail-controllability [2], i.e., to introduce a certain degree of restrictions on how the system can fail. The level and type of such restrictions are usually dependent on the application domain, criticality of the system, and the dependability attributes that are considered. Typical failure modes include fail-operational, fail-safe, fail-soft, fail-silent, fail-stop, crash and Byzantine (arbitrary) failures [2, 11]. Failure mode requirements can effectively be used for generating subsystem-level requirements in a hierarchical way and can help in performing localized analysis.

2.2 Component-level error modeling

Typically, this step involves modeling error behavior of individual software components as well as that of other system elements, such as component connectors, hardware nodes, middleware, and communication media. Our plan is to use probabilistic automata with timing where nodes of the automata represent error states and edges denote transition probabilities. An approach based on AADL [14] can be suitable for this step with proper extensions on the error modeling aspects. Our integrated development environment for CBS is being designed to specify and include information about component error behaviors with varying levels of details based on the available specifications. The level of details in component-level error models, as well as the dependability requirement specifications of the system, will decide the choice of analysis technique to be performed.

2.3 System-level dependability analysis

The analysis to be performed at this step depends on the dependability specifications and the component-level error models. Our aim is to get the basic structure in place so that multiple analysis techniques can be easily integrated to our framework. A challenging issue is how to compose error models to obtain a system-level error behavior. Error propagation can occur between two components, between a component and another system element or between two system elements. The architecture of the system will serve as an input to this step, where both impact and criticality analysis will be performed. Ideally, by looking at the error model of the composed system, one should be able to observe whether the system can possibly fail in a mode that is not allowed. If this is the case, the framework should further enable the identification of the critical paths in the architecture, and provide guidelines for efficient detection/recovery/correction strategies along with appropriate location for incorporating them, so that the resulting system meets the original dependability requirement as specified by the system designer.

Figure 1 illustrates the skeleton of a methodology for composing error information to perform a system-level error analysis. The methodology consists of critical path identification followed by propagation analysis performed on each identified path where the type of analysis depends on the specific failure mode requirement. Though components are usually considered as black boxes, we assume traceability of a critical parameter evaluation through the component chain. Otherwise, we have to consider all possible scenarios.

3. Error classification - revised

Characteristics of errors presented in this section are based on a synthesized view of several works [2, 13, 3, 9, 5]. It follows the basic classification of Avizienis et. al. [2] while extending it into details with the other works most of which address narrower areas but with finer details. It also presents various aspects of errors in two categories based on their influence on the error handling mechanisms. These categories essentially determine 'which mechanisms' and 'how much' are needed for adequate error handling. The various aspects considered are domain, consistency, detectability, impact, criticality and persistence of errors. The domain and consistency determine what kind of error handling mechanisms are appropriate while the rest determine the amount of error handling needed. The former is more relevant for design of the system and for providing qualitative guarantees, whereas the latter is important for quantitative predictions.

3.1 Domain

In component based systems, outputs generated by components can be specified by two domain parameters, viz., value and time as in [2, 3, 13]. Our hypothesis is that tolerating value and timing errors at component-level, requires different approaches and the associated costs are significantly different. Therefore, separation of value and time domains will enable using dedicated fault tolerance mechanisms for each type as well as help in achieving better error coverage with minimum cost. In this paper, we define the specified output generated by a component as a tuple based on these domain parameters:

Specified Output (SO) = $< v^*, V, T, \Delta_1, \Delta_2 >$

where the v^* is the exact desired value, V is the set of acceptable values, T is the exact desired point in time when the output should be delivered and $[T - \Delta_1, T + \Delta_2]$ is the acceptable time range for the output delivery to be termed as correct.

The output generated by a component is denoted as:

Generated Output (GO) = $< v, t >$

where v is the value and the t is the time point when the output is actually delivered.

The GO is considered to be correct if:

$v \in V$ and $T - \Delta_1 \leq t \leq T + \Delta_2$.

Value errors: The output generated by a component is erroneous in value domain (e_v) if $v \notin V$, where V is the set of acceptable values. We first classify errors in value domain as ***subtle*** (e_v^s), and ***coarse*** (e_v^c) based on our knowledge about the set of reasonable values for the output and the syntax that should be followed as in [3, 13].

The correctness of the value of a component output depends on the expectations of the user for that output. For instance, when a component produces a CAN message, the user expects the message identifier to be exactly correct. In an other case, the value of an output can be an optimal value as well as some other acceptable distinct values, such as an output of a meta-heuristic algorithm trying to find a near optimal allocation of tasks to hardware nodes under several constraints. Furthermore, for instance, if the produced output is a control value, such as a temperature reading, the value of the output is considered correct as long as it is within a specified range. As the cost for error handling mechanisms may differ for each of these cases, we further classify value errors as follows:

- *Inexact value* errors $(e_v^{\overline{e}})$

  $v \notin V$, where $V = \{v^*\}$

- *Unacceptable distinct value* errors $(e_v^{\overline{d}})$

  $v \notin V$, where $V = \{v^*, v_1, v_2, ..., v_n\}$, v^* is the ideal value and $v_1, v_2, ..., v_n$ are the other acceptable values

- *Inaccurate value* errors $(e_v^{\overline{a}})$

  $v \notin V$, where $V = \{v^* - \Delta_1^v, ..., v^* - 1, v^*, v^* + 1, ..., v^* + \Delta_2^v\}$ and $[v^* - \Delta_1^v, v^* + \Delta_2^v]$ is the range of acceptable values

A value error is a combination of the above classifications, i.e., e_v^{xy}, where $x \in \{c, s\}$ and $y \in \{\overline{a}, \overline{d}, \overline{e}\}$.

Timing errors: In [3, 13, 5] and in our classification, errors in time domain are classified into *early, late* and *infinitely late(omission)* timing errors. There are additional classes defined in [13] related to timing, namely bounded omission and permanent omission (crash or permanent halt) errors, which are covered later in this section.

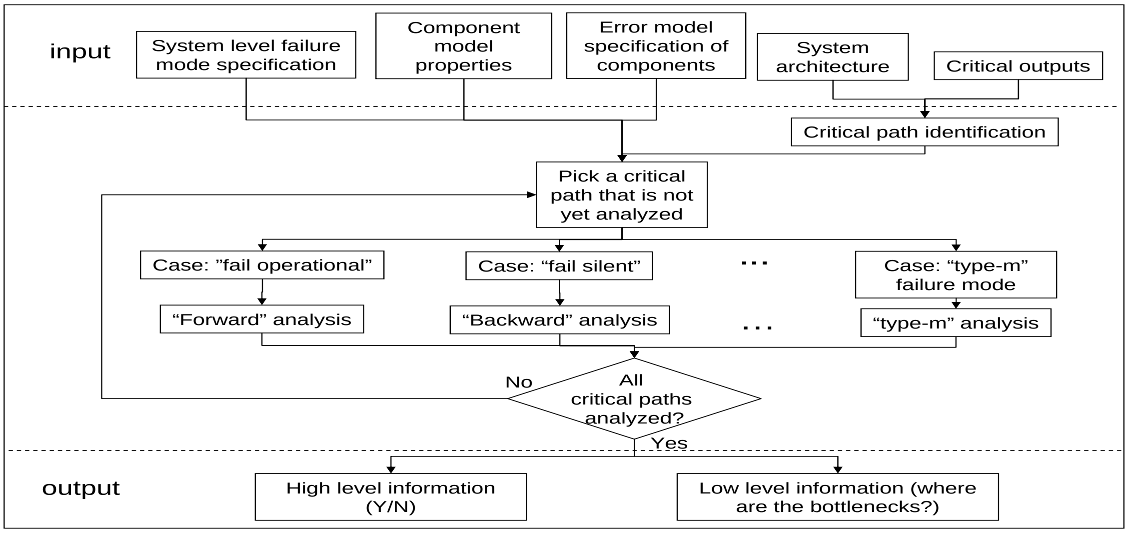

Figure 1. System-level error analysis

- *early* timing errors (e_t^e): $t < T - \Delta_1$
- *late* timing errors (e_t^l): $t > T + \Delta_2$
- *omission* timing errors (e_t^o): $t = \infty$

Errors both in time and value domain: Component outputs under this category are erroneous in both value and time domain simultaneously, i.e., $e_{v,t}^{a,b}$, where $a \in \{c\bar{e}, c\bar{d}, c\bar{a}, s\bar{e}, s\bar{d}, s\bar{a}\}$ and $b \in \{e, l, o\}$ if:

$$v \notin V \text{ and } (t < T - \Delta_1 \text{ or } t > T + \Delta_2)$$

3.2 Consistency

If a component provides replicas of an output to several consequent components, consistency issues may arise. In such a situation, the errors are considered consistent if all the receivers get identical errors. In [13], multi-user service errors are classified into consistent value errors, consistent timing errors, consistent value and timing errors and semi-consistent value errors. In semi-consistent value errors, some output replicas have unreasonable or out-of-syntax values, while the rest have identically incorrect values. In [3], non-homogeneous output replicas are defined to be erroneous. A non-homogeneous error with respect to value occurs if the values of received outputs are not close enough to each other. A non-homogeneous error with respect to time occurs if the timing of received outputs are not close enough to each other. Closeness is specified by using threshold values. In our classification, we use both consistency and homogeneity concepts.

- *Consistent* errors: Replicas of the output from a component are consistently erroneous if they belong to the same error category, e.g., both have coarse value errors or late timing errors.

  We further classify these errors as being precise or imprecise:

  - *Precise* errors: The values or generation times of replicas are consistently erroneous as well as both are within a precision range or identical.
  - *Imprecise* errors: The values or generation times of replicas are consistently erroneous, however either values or generation times (depending on the error type) are outside the specified precision range.

- *Semi-consistent* errors: Replicas of an output are defined as semi-consistently erroneous if all users receive erroneous outputs while at least one of them belongs to a different error category than the others.

- *Inconsistent* errors: Replicas of an output are defined as inconsistent if there are both correct and incorrect replicas.

The characteristics presented so far define our error classification and will be used in both propagation analysis and composition of component error models. Furthermore the classification will be used to determine which error handling mechanisms are adequate to control the error behavior during composition.

The error characteristics presented in the remaining of this section are related to the error coverage and controllability, i.e., these are the properties of errors which give hints about the coverage of existing error handling mechanisms and how much are needed to achieve the required degree of dependability.

3.3 Other error characteristics

Detectability: We characterize errors that can occur within a component in value, time or both value and time domains as *detectable* or *undetectable*, based on the available error detection mechanisms implemented within that component, or at the interfaces to other components such as reasonableness checks and watchdog timers. The complexity and the cost of implementing such mechanisms depend on the difficulty of detecting them.

Impact: Impact is a measure of an error that can occur within a component which represents the probability (or in case of having no probability figures, possibility) that the error propagates into a system failure visible at system outputs, as defined by Hiller et. al. in [9]. If more than one outputs exist, the impact of an error is the probability/possiblity of causing a failure in any of these outputs.

Criticality: Criticality is a measure which indicates the ability of an error to cause a system failure with catastrophic consequences, visible at system outputs. Criticality of an error is directly proportional to the criticality of the system outputs which they can have impact on:

$$\text{Criticality of } e_{v,t}^{a,b} \text{ on } O = C_O.\text{Impact of } e_{v,t}^{a,b} \text{ on } O$$

where O is the system output and C_O is the criticality of that output. If the system has more than one outputs:

$$\text{Criticality of } e_{v,t}^{a,b} = 1 - \prod_j (1 - \text{ Criticality of } e_{v,t}^{a,b} \text{ on } O_j))$$

where O_j is the j-th system output and $j = 1, 2, \ldots$. Note that criticality of errors is different from criticality of system outputs, where the latter is a measure of the severity of failure consequences at system output-level.

Persistence: Errors can be classified as transient, intermittent or permanent, based on their persistence. Transient errors are assumed to occur only once, while permanent ones never leave the system. Intermittent errors are assumed to occur during a bounded period or below a bounded frequency. Bounded omission failure is an example of such errors [13].

4. Error propagation in CBS

Errors in a component based system can occur in software components, middleware or hardware platform, and can propagate up to a system interface causing a system failure with a probability. This probability is dependent on the probability of occurrences of errors, the isolation between different system elements, existing error detection and handling mechanisms as well as the type of the errors. The research effort is currently increasing for finding ways to get these probabilities, and to use them appropriately [9, 7, 12, 1, 4].

We define the set of errors E, which includes instantiations of error types discussed in the previous section. We also define the following subsets of E as follows:

- E_i^{in} is the set of errors that are propagated into component C_i
- E_i^{gen} is the set of errors that are internally generated by component C_i and propagated out without any changes
- E_i^{pass} is the set of errors propagated into C_i that are propagated out without any changes
- E_i^{mod} is the is the subset of E_i^{in} that are transformed to another error type, masked or corrected
- E_i^{trans} is the set of errors that were originally belonging to E_i^{mod} or internally generated errors and transformed into the members forming this set
- E_i^{out} is the set of errors that are propagated from component C_i

$$E_i^{in} = E_i^{mod} \cup E_i^{pass}$$

$$E_i^{out} = E_i^{gen} \cup E_i^{pass} \cup E_i^{trans}$$

Errors can be transformed into E_i^{trans} by either C_i's normal execution or error handling mechanisms. These mechanisms can be implemented within components at component design stage or at the component interfaces at the architectural design or integration stages of CBD. Various mechanisms for different types of errors and their effects on error propagation are discussed in the following paragraphs.

Transformation of value errors: The possible ways of error transformations in value domain are shown in Table 1.

One way to detect coarse value errors is using reasonableness checks. Implementing reasonableness checks necessitates having knowledge about the behavior of the producer, for example, a range checking mechanism marks the temperature reading of a room as erroneous if the value read is -200°C based on our knowledge about the reasonable boundaries for that output. Coding checks are used to detect non-code value errors which is a specific type of coarse value errors (parity-check is an example for this type of check). Obviously, if more advanced error detection mechanisms are used which can identify more complex erroneous

Cause	Initial error	Final error
Error detection	e_v	e_v (transformation in time domain)
Error masking	e_v	no error
Error correction	e_v	no error
Component operation	e_v e_v^s e_v^c	no error e_v^c e_v^s

Table 1. Transformations of value errors

behaviors, the coverage of detectable errors are increased. Detecting subtle value errors is performed by more expensive error detection mechanisms, such as replica checking at a voter element. Propagation of value errors can be blocked after detection, by simply not allowing the erroneous output to be delivered to the next component. In this case a value error is transformed into an omission timing error.

Certain means allow masking of value errors, such as N-modular redundancy techniques, while some others can *correct* value errors by using, e.g., error correction codes. Both masking and correction techniques enable continuation of correct functioning upon errors.

Transformations of timing errors: Errors in time domain can be transformed according to the following order:

$$e_t^e \rightarrow \textit{no error} \rightarrow e_t^l \rightarrow e_t^o$$

Timing checks and watchdog timers can be used to detect timing errors produced by components. Early timing errors can be corrected by introducing delays. Propagation of early or late timing errors can be blocked by not transmitting them, if there are no means to correct them. In such cases, these errors are transformed into omission errors. When a value error is detected and omitted as described previously, the output is actually transformed from having no timing error to an omission error.

For *errors regarding consistency*, similar checks can be used and inconsistent errors can be transformed into consistent errors in both value and timing domains.

5. Summary and Ongoing Work

In this paper, we have proposed a framework to enable compositional reasoning of error models. We have surveyed various error classifications and failure modes in the literature with the aim of identifying their relations/contrasts as well as in arriving at an 'all-encompassing compilation of classifications'. We have investigated the error propagation in CBS and discussed the effects of error handling mechanisms on error propagation. Our ongoing research aims to a) add formalizations of component error models, error propagation models and error handling mechanisms, (including probabilistic variants of them) b) provide links to architectural reliability prediction models as well as c) new theories on dependability reasoning of multi-level compositions.

References

[1] W. Abdelmoez, D. Nassar, M. Shereshevsky, N. Gradetsky, R. Gunnalan, H. Ammar, B. Yu, and A. Mili. Error propagation in software architectures. *Proceedings of the 10th International Symposium on Software Metrics*, 2004.

[2] A. Avizienis, J. Laprie, and B. Randell. Fundamental concepts of dependability. *Research Report N01145, LAAS-CNRS*, April 2001.

[3] A. Bondavalli and L. Simoncini. Failure classification with respect to detection. *Proceedings of 2nd IEEE Workshop on Future Trends in Distributed Computin*, 1990.

[4] V. Cortellessa and V. Grassi. A modeling approach to analyze the impact of error propagation on reliability of component-based systems. *Proceedings of the Component-based Software Engineering Conference*, 2007.

[5] K. Echtle and A. Masum. A fundamental failure model for fault-tolerant protocols. *Proceedings of Computer Performance and Dependability Symposium*, 2000.

[6] J. Elmqvist and S. Nadjm-Tehrani. Safety-oriented design of component assemblies using safety interfaces. *Elsevier-Electronic Notes in Theoretical Computer Science*, (182):57–72, 2007.

[7] L. Grunske. Towards an integration of standard component-based safety evaluation techniques with SaveCCM. *Proceedings of the Conference Quality of Software Architectures*, 2006.

[8] L. Grunske and R. Neumann. Quality improvement by integrating non-functional properties in software architecture specification. *Proceedings of the Second Workshop on Evaluating and Architecting System dependabilitY*, 2002.

[9] M. Hiller, A. Jhumka, and N. Suri. EPIC : Profiling the propagation and effect of data errors in software. *IEEE Transactions on Computers*, 53(5):512–530, 2004.

[10] A. Joshi, S. Vestal, and P. Binns. Automatic generation of static fault trees from aadl models. *Proceedings of the Workshop on Architecting Dependable Systems*, 2007.

[11] H. Kopetz. On the fault hypothesis for a safety-critical real-time system. *Workshop on Future Generation Software Architectures in the Automotive Domain*, January 2004.

[12] P. Popic, D. Desovski, W. Abdelmoez, and B. Cukic. Error propagation in the reliability analysis of component based systems. *Proceedings of the 16th IEEE International Symposium on Software Reliability Engineering*, 2005.

[13] D. Powell. Failure mode assumptions and assumption coverage. *Proceedings of 22nd International Symposium on Fault-Tolerant Computing*, 1992.

[14] A. E. Rugina, K. Kanoun, and M. Kaâniche. An architecture-based dependability modeling framework using AADL. *Proceedings of International Conference on Software Engineering and Applications*, November 2006.