Improving Fault Tolerance using Execution Guarantees

Nikolas Nehmer
University of Kaiserslautern
P.O. Box 3049
67653 Kaiserslautern, Germany
nnehmer@informatik.uni-kl.de

ABSTRACT
Dependability in today's software systems is hard to achieve. Missing physical boundaries like the laws of nature in other engineering disciplines and a lack of understanding how to control the software runtime behavior to enforce consistency are two major reasons. This paper introduces a novel approach to improving software systems' dependability by the concept of execution guarantees.

1. MOTIVATION
Consider a workflow application for planning and scheduling a meeting where different tasks like specifying the topic, time and participants, reserving a meeting room and a beamer, inviting the participants etc. are involved. The isolated examination of the single tasks results in a mass of integrity constraints that have to be fulfilled in order to guarantee a consistent processing of these tasks - e.g. the number of participants has to be larger than zero or only a vacant meeting room can be booked. Combining these tasks within a workflow complex dependencies and cross references among those tasks arise - e.g. the meeting room and the beamer have to be successfully booked before the participants can be invited or cancellation of a mandatory meeting participant leads to the cancellation of the meeting and notification of all participants and release of the meeting room. Specifying, guarding and monitoring these diverse consistency constraints is the primary goal of the concept of execution guarantees.

The idea of execution guarantees is closely related to the concept of "design by contract"[10] but goes beyond it's scope. "Design by contract" is a methodology to improve software reliability (correctness and robustness) where formal expressions are used to establish a contract between a service user and a service provider. A precondition expresses requirements that any call must satisfy if it is to be correct, a postcondition expresses properties that are ensured in return by the execution of the call and class invariants apply to all instances of a class, transcending particular routines. Runtime support in many programming languages assures that any violation of the contract between caller and provider can be detected immediately. “Design by contract” is typically limited to static boolean expressions over system variables that can be evaluated statically during compilation and dynamically monitored on predefined guarding points (before entry to a method and before return form a method). Execution guarantees expand and generalize these concepts and allow to maintain properties on the dynamical system behavior like atomic and mutual completion of multiple actions.

Accordingly, execution guarantees are formal expressions of system properties maintained during system execution unless they are explicitly waived by the application. Execution guarantees specify integrity constraints of software systems and dependencies among software components. The main scope of execution guarantees is the enforcement of system consistency. Execution guarantee specifications have to be supported by system runtime mechanisms. The adherence to execution guarantees has to be monitored and guarded by appropriate generic system mechanisms to control the runtime behavior. Whenever the guarding system has detected a violation of an execution guarantee appropriate actions have to be initiated in order to handle the violation thereby enforcing the execution guarantees. This is achieved by violation specific exception handling [12]. Consequently, execution guarantees are either fulfilled or violated and thus expressed by boolean expressions. Fault tolerance is one means of increasing dependability. While software faults can’t be ruled out handling guarantee violations is a crucial element in the concept of execution guarantees. Execution guarantees define the conditions on which software elements will work and how they will behave in situations where the guarantees are violated. It stands to reason that any unhandled actual or potential violation of an execution guarantee will (sooner or later) result in an execution failure. Generating the runtime support for execution guarantees is a challenging task.

Furthermore, the basic idea of execution guarantees is based on developing a holistic dependability approach for software engineering processes. It provides the opportunity to regard every development step and aspect of a software engineering process from requirements engineering where execution guarantees (consistency constraints and dependencies...
among components) are specified to testing where formal consistency specifications may lead to the generation of test cases. In contrast to other engineering disciplines like architecture natural boundaries like the laws of nature can’t be applied to software systems. Missing natural boundaries and corresponding missing control in today’s software systems leads to failures [9]. As already discussed above execution guarantees define integrity constraints and dependencies among components at system modeling level. Thereby execution guarantees introduce guarded control points as artificial boundaries within the software system and accordingly can be used to control the software runtime behavior.

The reminder of this paper is structured as follows. In Section 2 basic execution guarantee principles and concepts are introduced. Section 3 deals with the guarding and monitoring runtime support for execution guarantees. Section 4 presents approaches to handle guarantee violations. Section 5 briefly summarizes the paper and gives a short outlook on future work.

2. BASIC PRINCIPLES

In common dependability taxonomies [1] failures, faults and errors are distinguished as threats to dependability. In the notion of execution guarantees this traditional model is slightly extended. A failure is an event that occurs when a service deviates from its expected behavior as defined at the external system interface. The deviation itself is called an error while the cause of this failure is called a fault. A fault can be internal or external relative to the system. An internal fault is necessary to enable an external fault to harm the system by causing an error and possible subsequent failures. In most cases a fault causes an error in the internal state of a system before it becomes noticeable at the system interface. For this reason it is important to note that not every error reaches the system’s external interface and results in a failure therefor. A fault is active when it causes an error, dormant otherwise. In case of execution guarantees the activation of a fault (the error) leads to a guarantee violation which can be detected by violation monitors and handled adequately before it manifests as a failure at the external interface.

\[
\text{fault} \rightarrow \text{error} \rightarrow \text{guarantee violation} \rightarrow \text{failure}
\]

In this section the basic principles of execution guarantees are introduced. Execution guarantees are classified and operators are provided. Furthermore execution guarantee scopes and levels and nested guarantees are introduced.

2.1 Execution Guarantee Classification

Today’s software systems are too large and complex to consider every potential consistency violation alternative. Defining only relevant and essential constraints focusing on avoiding hazards e.g. is one obvious solution. Using experiences from related systems within the same domain, application area or scope by focusing on domain-typical failure situations is another one. Furthermore constraints are too diverse to generate the guarding mechanisms according to every single constraint. A feasible and auspicious approach is the aggregation of execution guarantees into classes with equal characteristics. Single execution guarantee instances are mapped onto these guarantee classes (see Figure 3). Execution guarantee classes are aggregated into execution guarantee categories.

Currently three execution guarantee categories with several subcategories and 14 execution guarantee classes are distinguished. The focus is on functional guarantees which will be further detailed using a sample workflow application. The main scope of these functional guarantee classes is the enforcement of system consistency. The execution guarantee classification is shown in the following bullet list:

- functional guarantees (enforcing consistency)
  - data guarantees
    - attribute & object guarantees
    - range guarantee
    - function guarantee
    - type guarantee
    - quantity guarantee
    - existence guarantee
    - modification guarantee
    - update guarantee
    - deletion guarantee
  - dependency guarantees
    - referential integrity guarantee
    - relation guarantee
  - process guarantees
    - state guarantee
    - process dependency guarantee (may lead to atomic execution)
    - isolation guarantees
    - durability guarantees
- non-functional guarantees (enforcing service quality)
  - network (quality of service)
  - availability
  - trust, security, etc.
  - performance
  - and more...
- implicit guarantees

Execution guarantee classes are divided into two major categories – functional and non-functional guarantees – analogous to the dichotomy known from requirements engineering. Additionally implicit guarantees like division by zero or memory overflow are depicted. Non-functional guarantees enforcing service quality are not in the scope of this work. Functional guarantees, enforcing consistency, are further divided into data guarantees and process guarantees. Data guarantees focus on guarantee statements concerning data objects and corresponding attributes as well as on dependencies between data objects. Guarantees on attributes and objects split up into range guarantees that check if a data object is within a predefined range, type guarantees that checks the attribute’s or object’s type, the quantity guarantee which verifies an object’s quantity, the modification guarantee to verify if an object or attribute has been changed and existence guarantees to check the existence of an object or attribute. Dependency guarantees check for referential integrity and verify if special relations between data objects exist. On the other hand process guarantees verify dependencies to process states. State guarantees verify for example if a particular system state has been reached. Process dependency guarantees verify process oriented dependencies among process tasks/actions. The atomic com-
pletion of a set of actions or both actions A and B have to be finished before action C is finished are two examples for process dependency guarantees. Process dependency guarantees can lead to transactional system behavior and map the ideas of completion dependencies [2] to execution guarantees. The focus of process dependency guarantees is on process and data consistency instead of transactional system mechanisms. In table 1 the execution guarantee classes are illustrated by some examples.

2.2 Guarantee Operators and Functions
Execution guarantees are built, controlled and embodied by invariants or pre- and postconditions [7] defining integrity constraints and dependencies among components according to the ideas of “design by contract”. In the concept of execution guarantees simple and complex execution guarantees are distinguished.

2.2.1 Simple Guarantees
Simple execution guarantees are expressed by boolean expressions consisting of basic operators and simple operands which either evaluate to true or false. The well-known logic operators are supplemented by non-truth functional modal operators. Additionally more operators like arithmetic operators are possible. A choice of execution guarantee operators building single execution guarantees are illustrated and classified in the following list. A formal execution guarantee specification and grammar is work in progress.

- comparison operators
  - equals
  - greater
  - less
  - greater or equal
  - less or equal
- temporal operators
  - after
  - before
  - concurrent
- modal operators
  - quantifiers
    - exists
    - forall
  - implication
  - implies

Comparison operators compare values of variables and temporal operators relate different actions according to their relative execution. Modal operators are helpers in expressing more complex execution guarantees. Especially in postconditions current and old values of data objects have to be distinguished according to the ideas provided by “design by contract” [10]. By using old.attribute the old value of the attribute on entry to the guarded function is referenced. If an old value is used, the object is cloned at the beginning of the method. In the postcondition the current value is compared to its counterpart in the clone.

2.2.2 Complex Guarantees
More complex execution guarantees can be expressed by combining and concatenating several simple execution guarantee instances. Concatenation operators known from boolean algebra are used to combine single guarantee statements. Currently two categories of complex execution guarantee operators are distinguished – concatenation operators and dependency operators. Concatenation operators concatenate single boolean execution guarantee statements and form larger ones by logical “and”, “or” “xor” and “not” operators. Dependency operators provide the possibility relating several simple execution guarantees according to different characteristics like time or implication. At present the complex execution guarantee operators are structured as shown below:

- concatenation operators
  - AND
  - OR
  - XOR
  - NOT
- dependency operators
  - temporal operators
    * after
    * before
    * concurrent
  - implication
    * implies

2.3 Guarantee Scopes
Execution guarantees are not always valid for a whole software system. Mostly execution guarantees are restricted to a single object or are dependent on some condition. Consequently execution guarantees can be restricted to application scopes whereas the type of scope can differ according to various characteristics. Currently four different restriction scopes are distinguished as shown below:

- conditional scope
- temporal scope
- spatial scope
- state scope

A conditional scope validates execution guarantees depending on some condition. Validating execution guarantees depending on some time frame is called temporal scope. A spatial scopes restricts execution guarantees on some code area and the state scope validates execution guarantees depending on some system/workflow state.

2.4 Guarantee Levels, Mapping and Nesting
According to the layered structure of software systems execution guarantees are organized in levels of abstraction (see Figure 1). The higher the abstraction level is, the more the execution guarantees specified are focused on the dependencies among components and algorithmic details become irrelevant on that level. Furthermore on lower levels of abstraction dependencies among components are not expressive but detailed functional guarantees on algorithmic details are. On the abstraction level of workflows for example execution guarantees especially focusing on the interdependency between services can be specified which wouldn’t be meaningful on lower levels. Execution guarantees on one...
### Guarantee Class Example

<table>
<thead>
<tr>
<th>Guarantee Class</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>range guarantee</td>
<td>budget &lt; 1000</td>
</tr>
<tr>
<td>function guarantee</td>
<td>roomBudget + beamerBudget + cateringBudget &lt; budget</td>
</tr>
<tr>
<td>type guarantee</td>
<td>type(date) == Date</td>
</tr>
<tr>
<td>quantity guarantee</td>
<td>participants &lt;= 10</td>
</tr>
<tr>
<td>existence guarantee</td>
<td>meeting != null</td>
</tr>
<tr>
<td>modification guarantee</td>
<td>modified(date) == true</td>
</tr>
<tr>
<td>update guarantee</td>
<td>update(budget) &lt; 1000</td>
</tr>
<tr>
<td>deletion guarantee</td>
<td>meeting != null</td>
</tr>
<tr>
<td>referential integrity guarantee</td>
<td>meeting references participants</td>
</tr>
<tr>
<td>relation guarantee</td>
<td>forall Ressource r in bookRessource()</td>
</tr>
<tr>
<td>state guarantee</td>
<td>inviteParticipants() after specifyParticipants()</td>
</tr>
<tr>
<td>process dependency guarantee</td>
<td>atomic(bookBeamer(),bookRoom())</td>
</tr>
<tr>
<td>isolation guarantee</td>
<td>isolated(bookRoom())</td>
</tr>
<tr>
<td>persistence guarantee</td>
<td>persistent(bookRoom())</td>
</tr>
</tbody>
</table>

**Table 1: execution guarantee examples**

Level of abstraction have a great impact on the other levels and therefore they have to be transparently traceable. This can be achieved by a two-way mapping mechanism which maps execution guarantees from higher levels to lower ones and vice versa.

Additionally, the structuring of software systems into software layers mostly builds up a structure of nested components constituting the overall system – e.g. workflows consist of services, services consist of objects or other services, objects consist of attributes and methods and so on. Accordingly, a nested structure of execution guarantees arises. Combined/Expanded by the idea of guarantee scopes a structure of nested guarantee spheres emerges (see Figure 4). Considering a bottom up approach, execution guarantees on higher levels base on guarantees on lower levels. Consequently execution guarantees on higher levels have to be at most as strict as the underlying guarantees. Regarding a top down approach execution guarantees on higher levels enforce adequate guarantees on lower levels. The guarantees on the lower levels have to be at least as strict as the corresponding guarantee on the higher levels. These correlations are depicted by $\text{Strictness}(eg_h) \leq \text{Strictness}(eg_l)$ and $\text{EG}_h \supseteq \text{EG}_l$. Furthermore the mapping relations are illustrated in Figure 2.

#### 3. GUARANTEE GUARDS AND MONITORS

Execution guarantee specifications have to be supported by corresponding system runtime mechanisms to control the system runtime behavior. The adherence to execution guarantees has to be monitored and guarded by appropriate generic system mechanisms corresponding to the different guarantee classes. Monitoring execution guarantees has to reliably detect every execution guarantee violation to enable a proper handling of the violations. Execution guarantees can either be satisfied or violated, nothing in between due to their design as boolean expressions. Execution guarantees provide a duality between satisfied guarantees and guarantee violations. Execution guarantee violations have to be handled appropriately. Guarding execution guarantees supports and enhances the adherence to the given guarantee. As stated before a mapping from execution guarantee instances specified at system modeling level to guarantee classes is required. Furthermore guarantee classes are mapped to adequate generic system mechanisms guarding and monitoring guarantees associated with the guarantee class (see Figure 3). Currently it is intended to guard and monitor execution guarantees by using assertions [7, 3]. Generating the runtime support for execution guarantees is a challenging task. In contrast to the current practice of relying on snapshot data we propose to base failure analysis on extensive logging of execution guarantee violations and subsequent log data analysis[11].

#### 4. RECOVERY

In todays complex software systems the occurrence of faults is unavoidable. Means of increasing the fault tolerance of
such systems is necessary to increase the overall system dependability. In traditional fault tolerance taxonomies [1] error detection and recovery are distinguished. Error detection is already covered by monitors detecting guarantee violations described earlier. Recovery is further divided into fault handling and error handling. As guarantee violations detect activations of faults (errors), error handling will be emphasized here.

However if an execution guarantee violation is detected an appropriate guarantee specific violation handler has to be triggered. It is intended do define guarantee violation handling patterns according to the different execution guarantee classes discussed before. Furthermore when exceptions are raised concurrently they cannot be simply handled sequentially in some order or priority-based [12]. As execution guarantees are nested and organized in different levels it is intended to design the execution guarantee violation handling accordingly in a hierarchical structure. Concurrent low-level violations are aggregated into a higher-level violation which covers all raised exceptions. Termination of the execution guarantee violation handling process is a crucial requirement to avoid infinite nested violation handler calls (violation triggers a new violation and so on). Concerning concurrent execution guarantee violations and complex guarantees (like a+b+c < 100) interesting questions arise. Which system instance is responsible for handling the violation? Who has got the correct information on handling the violation? Who was responsible for violating the complex guarantee (a, b or c) and how should the violation be handled appropriately?

As was mentioned above, for the purpose of studying the dynamical behavior of software systems, we have to distinguish the different levels of abstraction of the pieces constituting the overall system. As an illustration, consider the table 2. Just focusing on the average execution times of typical execution units at different levels of abstraction, we can distinguish 14 orders of magnitude from low-level routines (10\(^{-6}\) seconds) up to workflows or large software engineering projects (10\(^3\) seconds). For the sake of simplicity we will – in the following – not consider all those different levels, but restrict discussion to the module level (programming in the small) and the system level (programming in the large). In short- and medium-lived executions atomicity is a powerful mechanism to handle errors by recovering to a state before the action had been taken. Regarding long-lived executions restoring a state before a action had started and undo all actions taken in between and thus losing all work is unacceptable in most cases. Traditional transactional [4, 6] error handling models are rarely applicable in long-lived scenarios such as large workflow applications. Some ideas to tackle these problems are proposed below.

Two different types of execution guarantee violation handling are distinguished – intra-system violation handling and inter-system violation handling. In an ideal case an error can be handled within the system while the service provision (intra-system error handling) and service users are not affected. One idea to reach this goal is to encapsulate every access which potentially could cause a guarantee violation within an atomic sphere. The state before entering the atomic sphere can be regarded as a recovery point. If a guarantee violation is detected within an atomic sphere the system state is reseted to the last recovery point. This yields the advantage that if a guarantee violation occurs it does not reset the whole service, but only the part that caused the violation. Code blocks where a guarantee violation is detected are not always the origin of the problem. The root cause could have been executed much earlier.

To cover this problem, nesting atomic spheres (see Figure 4) is considered as described before. If a guarantee violation is detected the inner atomic sphere (e.g. inner method of a cascading method call) is reset first. If the recovery process was unsuccessful the system is reset one level further, e.g. on the level of the enclosing method. A flexible backward recovery mechanisms according to the concept of “Spheres of Control” [8, 4, 5] will be investigated. Reconsider the meeting scheduler workflow example. One Service is used to specify the meeting date, the topic, the participants etc. Within the main method parameters like the topic are set directly while participants and the meeting room e.g. are set via method calls from the main method to sub-methods. Imagine a guarantee violation within the sub-method for specifying the meeting room. In a first attempt the system will be recovered to the state before the sub-method-call. If this recovery process is unsuccessful the system state is reset one level further to the state before the main-method-call.

If this fine grained intra-system violation handling is unsuccessful and service provision is affected a violation handling on a higher level of abstraction (inter-system violation handling) is initiated. If the system is regarded as the service provider and the invoking system as service user, inter-system violation handling mechanism takes place between service provider and service user. Predefined fixed handling patterns lends themselves to inter-service violation handling.
Certainly, the applicability of such patterns depend on the violated guarantees and guarantee classes. Possible patterns are:

- kill client – resetting the service user’s state to a consistent state before the service call
- recovery – resetting the service provider’s state to a consistent state before the service call
- negotiation – negotiating between service user and provider with possible compensation
- ...

5. CONCLUSION
Fault tolerance is an important means to increase software systems’ dependability. Proper error detection and recovery mechanisms are essential for a fault tolerance concept. By generalizing and extending the ideas of “design by contract” execution guarantees provide a novel approach to fault tolerance introducing reliable error detection and recovery mechanisms. Execution guarantees are formal expressions of system properties specifying integrity constraints of software systems and dependencies among software components. Execution guarantees focus on enforcing system consistency by maintaining complex consistency constraints during system runtime. Execution guarantees can either be satisfied or violated, nothing in between. They provide a duality between satisfied guarantees on the one hand and guarantee violations on the other hand. Execution guarantee violations are reliably detectable by monitors and can be handled appropriately. Execution guarantees provide a promising concept to control the software runtime behavior to enforce system consistency.

Work in progress and futures steps are shown below:

- Building a sample workflow application
- Developing a formal grammar for execution guarantees with a formal representation of dependencies between the different levels of abstraction
- Developing a framework of generic system mechanisms to support execution guarantees
- Integrating the sample application and the framework
- Refinement and implementation of violation handling concepts

6. REFERENCES