Insegnamento

# Qualità del Software (blended)

2324-1-F1801Q115

| | |
|---|---|
| **Iniziato** | domenica, 24 marzo 2024, 18:05 |
| **Stato** | Completato |
| **Terminato** | domenica, 24 marzo 2024, 22:40 |
| **Tempo impiegato** | 4 ore 34 min. |
| **Valutazione** | Non ancora valutato |

Domanda **1**

Completo

Punteggio max.: 1,00

In the Company, the marketing division is worried about the start-up time of the new version of the operating system that the Company produces and distributes. The marketing division representative suggests a software requirement stating that the start-up time shall not be annoying to users. Explain why this simple requirement is not verifiable and try to reformulate the requirement to make it verifiable.

Summarize your answer in one slide (or a text that resembles a slide).

This simple requirement, stating that the start-up time shall not be annoying to users, is **not verifiable** because it is **subjective**.

Different users could interpret "annoying" in different ways, so since it varies from one user to another **it doesn't provide a measurable requirement**.

The specification is **ill-formed**: it is **ambiguous**, since the required properties depend on the interpretation.

Validation against actual requirements depends on human assessment, which can introduce **ambiguity**, misunderstanding and disagreement.
On the other hand, a specification must be **clear to avoid ambiguity**.

To reformulate this requirement to make it verifiable, we can define "annoying" in a **measurable** way.

For example, we can reformulate the requirement as follows: "The system shall start up **within 30 seconds** when running on the recommended hardware configuration.".

The requirement is now specific, measurable and testable, making it verifiable since it removes subjectivity.

Domanda **2**

Completo

Punteggio max.: 1,00

A calendar program should provide timely reminders; for example, it should remind the user of an upcoming event early enough for the user to take action, but not too early. Unfortunately, “early enough” and “too early” are qualities that can only be validated with actual users. How might you derive verifiable dependability properties from the timeliness requirement?

To derive verifiable dependability properties, we can utilize **verification** techniques.

Verification techniques check the consistency of an implementation (the **calendar program**) with a specification (which contains the **timeliness requirement**). These checks include self-consistency and well-formedness, which allow us to definitively determine that a program is "incorrect" if it is ill-formed.

**Dependability properties** cover correctness, reliability, robustness, and safety.

Firstly, we can define "timely" in a **measurable** way, for instance "reminders should be sent 30 minutes before an event takes place.".
We can adjust this based on **user feedback**; while conducting user testing, we verify that the calendar program works as intended through test results.

Since the qualities "early enough" and "too early" can only be validated with actual users, users can set their **preferred reminder times** in the settings.

Therefore, to ensure the system is both **useful** (it meets its actual goals) and **dependable** (it is consistent with its specification), we consider **user preferences and behaviors**.

Domanda **3**

Completo

Punteggio max.: 1,00

It is sometimes important in multi-threaded applications to ensure that a sequence of accesses by one thread to an aggregate data structure (e.g., some kind of table) appears to other threads as an atomic transaction. When the shared data structure is maintained by a database system, the database system typically uses concurrency control protocols to ensure the atomicity of the transactions it manages. No such automatic support is typically available for data structures maintained by a program in main memory.

Among the options available to programmers to ensure serializability (the illusion of atomic access) are the following:

- The programmer could maintain very coarse-grain locking, preventing any interleaving of accesses to the shared data structure, even when such interleaving would be harmless. (For example, each transaction could be encapsulated in a single synchronized Java method.) This approach can cause a great deal of unnecessary blocking between threads, hurting performance, but it is almost trivial to verify either automatically or manually
- Automated static analysis techniques can sometimes verify serializability with finer-grain locking, even when some methods do not use locks at all. This approach can still reject some sets of methods that would ensure serializability
- The programmer could be required to use a particular concurrency control protocol in his or her code, and we could build a static analysis tool that checks for conformance with that protocol. For example, adherence to the common two-phase-locking protocol, with a few restrictions, can be checked in this way
- We might augment the data accesses to build a serializability graph structure representing the "happens before" relation among transactions in testing. It can be shown that the transactions executed in serializable manner if and only if the serializability graph is acyclic.

Compare the relative positions of these approaches on the three axes of verification techniques: pessimistic inaccuracy, optimistic inaccuracy, and simplified properties.

Comparing the options available to programmers to ensure serializability in multi-threaded applications, based on the verification trade-off dimensions we can observe the following:

- **Coarse-grain locking**: since we are preventing every interleaving, even when it would be harmless, the **pessimistic** inaccuracy is **high**, while the **optimistic** inaccuracy is **low** because it almost guarantees serializability.
  On the other hand, since we are considering simplicity, the "**simplified properties**" dimension is **high**.
- **Automated static analysis**: it can reject some methods that ensure serializability, so the pessimistic inaccuracy is not low; also, the optimistic dimension is not low since it may not detect all violations. The "simplified properties" dimension is for sure not as high as the coarse-grain locking's, since it requires a more complex analysis. Overall, we could say that **all the dimensions** are **medium**.
- **Concurrency control protocol**: since the protocol is well-known, the **pessimistic** dimension is **low**, just **like the optimistic** one, assuming it can catch violations. Since we are substituting a property with a simpler sufficient one, the "**simplified properties**" dimension is **at least medium**.
- **Serializability graph**: the **pessimistic** inaccuracy is **low**, **like the optimistic** one, because it can be shown that the transactions executed in serializable manner iff the graph is acyclic. We can say that the "**simplified properties**" dimension is not low because we are using a property that is easier to check, so **at least medium** (it depends on how complex the graph is).