Keine Angst vor Software-Varianten
Wiederverwendung und Vererbung von Testfällen

Michael Wittner, Razorcat Development

Die Herausforderung beim Testen von Software-Varianten besteht darin, dass jede Variante vollständig getestet werden muss. Nachfolgend wird eine Methode zur Wiederverwendung und Vererbung von Variantentests vorgestellt. Über die Definition von Basistests, die an Variantentests vererbt werden, kann redundante Arbeit vermieden werden. Bei jeder Änderung der Applikation müssen die Tests nur an einer Stelle gepflegt werden.

1 Problemstellung

Sicherheitskritische Normen der verschiedenen Industriezweige, z.B. die ISO 26262 für die Automobiltechnik oder die IEC 62304 für die Medizintechnik und übergreifend für alle Industriezweige die IEC 61511 verlangen beim Test eine vollständige Codeabdeckung (Code-Coverage). Das bedeutet, dass jede Software-Variante vollständig getestet werden muss. In der Praxis geschieht dies oft durch Kopieren der Tests einer Variante und der Anpassung der kopierten Tests an die jeweils andere Variante. Neue Anforderungen oder Änderungen der Software verteuern einen solchen Test der Varianten aufgrund redundant durchzuführender Änderungen in allen Varianten. Neben dem hohen Aufwand bei der Pflege und Erweiterung solcher Tests können sich z.B. durch Copy&Paste sehr leicht Fehler einschleichen, durch die letztendlich auch sicherheitskritische Fehler in der Applikation unentdeckt bleiben können.

1.1 Was ist eine Variante?

Es gibt verschiedene Möglichkeiten, Software-Varianten zu erstellen (Beispiel C/C++ Source Code):

- Ein-/Ausschalten von Code-Teilen durch Defines
- Generierung von Code-Varianten über Tools (z.B. aus MATLAB)
- Kopieren, Umbenennen und Ändern von Quelldateien
- Gleiche Quellen auf verschiedenen Hardwareplattformen ausführen (bei hohen Sicherheitsanforderungen)
- Gleicher Code mit unterschiedlichen Applikationswerten

Eine Software-Variante ist durch eine bestimmte Konfiguration eines Software-Moduls (beispielsweise einer C Quelldatei) definiert. Diese Variante muss nicht notwendigerweise funktional sein, es kann sich auch um eine abstrakte Variante handeln, die in dieser Form niemals in einem fertigen Gerät seinen Dienst versehen wird. Erst durch spezifische Einstellungen (meist über Defines) werden aus einer abstrakten Basis-Variante die verschiedenen tatsächlich verwendbaren Software-Varianten.

1.2 Testziel: Code-Coverage

Für die vollständige Code-Abdeckung jeder Variante könnte das Messergebnis der Code-Coverage des variantenspezifischen Codes mit dem Messergebnis des gemeinsam genutzten Codes zusammengezählt werden. Das einfache Beispiel in Bild 1 soll verdeutlichen, warum ein solches Zusammenzählen von Code-Coverage nicht funktionieren kann und dabei Programmierfehler möglicherweise nicht erkannt werden. Die dargestellte Funktion hat in den Zeilen 19-23 einen gemeinsamen Teil und in den Zeilen 15-17 einen zusätzlichen Teil für Variante 1. Bei der Variante 1 wird vor der Prüfung der Variable „level“ ein weiterer Wert aufaddiert.

```
13 ret_t check_level(int level)
14 {
15 #ifdef VARIANT_1
16     level += supplementary_level;
17 #endif
18
19     if (level < MINIMUM) {
20         return alarm;
21     }
22
23     return  ok;
24 }
25
```

Bild 1: Funktion mit Code-Varianten

Der gemeinsame Teil in den Zeilen 19-23 könnte mit zwei Testfällen ausreichend getestet werden. Für die Variante 1 kommt kein zusätzlicher Programmzweig hinzu, so dass für eine vollständige Code-Coverage kein weiterer Testfall notwendig ist. Ein einfacher Testfall mit „supplementary_level > MINIMUM“ würde als funktionaler Test für die Variante ausreichen. Das Zusammenzählen der Coverage-Messergebnisse würde eine vollständige Codeabdeckung ergeben. Doch an dieser Stelle zeigt sich, dass ein einfacher Programmierfehler, wie das Fehlen des Additionsoperators in der Zeile 16 in Bild 2 dargestellt, nicht entdeckt werden würde.

```
13 ret_t check_level(int level)
14 {
15 #ifdef VARIANT_1
16     level = supplementary_level;
17 #endif
18
19     if (level < MINIMUM) {
20         return alarm;
21     }
22
23     return  ok;
24 }
25
```

Bild 2: Code-Variante mit Fehler

Es reicht daher nicht, einzelne Teile eines mit Hilfe von Defines zusammengesetzten Varianten-Codes zu testen und die Messergebnisse der Code Coverage zusammenzuzählen: Jede Code-Variante ist wie ein eigenständiges Programm zu sehen, da ausgeblendete oder dazugekommene Teile die gemeinsamen Teile beeinflussen können.

1.3 Hoher Pflegeaufwand bei Änderungen

Es liegt in der Natur von Software-Varianten, dass sie ähnliche Funktionalitäten besitzen. Daher werden auch die notwendigen Tests relativ ähnlich sein. Es lohnt sich also, über Variantentests nachzudenken, gerade wenn die Unterschiede der Varianten relativ gering sind. Der Testaufwand lässt sich durch gezielte Wiederverwendung von Testfällen deutlich reduzieren.

Der einfachste Ansatz wäre, zunächst eine Menge von Basistests zu entwickeln, die mehr oder weniger für alle Varianten gelten. Diese Tests werden dann für jede Variante kopiert und nach Bedarf abgeändert. Man erspart sich dadurch die Neuerstellung von Tests, allerdings entsteht gleichzeitig eine Unmenge fast identischer Tests, die bei Änderungen der Software mit hohem Aufwand gepflegt werden müssen.

2 Lösungsansatz

Anhand eines Beispiels wird im Folgenden eine Methode zur Erstellung und Pflege von Variantentests vorgestellt. Das Beispiel behandelt eine Funktion zur Statusanzeige des Tankinhalts verschiedener Fahrzeugvarianten (PKW und LKW). Erschwerend kommt hinzu, dass die Fahrzeugvarianten "LKW" mit einem Zusatztank ausgestattet sein können, dessen Pegelstand ebenfalls berücksichtigt werden soll.

2.1 Beispielfunktion

Als Beispiel soll eine Funktion dienen, die bezogen auf den Füllstand des Tanks eines Fahrzeugs einen Status zurückliefert. Die Spezifikation ist in Bild 3 in grafischer Form gegeben: Die Funktion soll eine Warnung oder einen Alarm zurückgeben, wenn jeweils ein definierter Pegelstand unterschritten wird. Ansonsten liefert die Funktion den Wert "Normal".

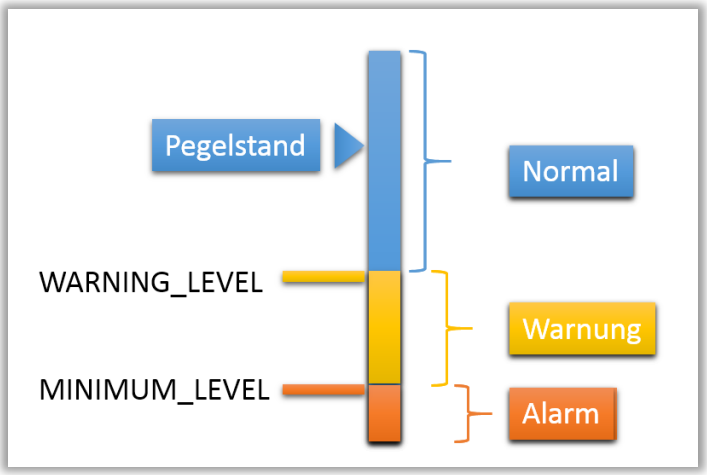

Bild 3: Definition des Pegelstandwertes mit Berechnungsschwellen

Eine einfache Implementierung dieser Funktion könnte wie in Bild 4 dargestellt aussehen. Über die Varianten-Konfiguration (#define TRUCK) wird der Pegelstand eines Zusatztanks ggf. in die Berechnung mit einbezogen.

fuel_sensor.c

```
// Fuel level example
//
// Returns warning/alarm depending on the fuel level

#include "fuel_sensor.h"

tank_control_t tank;

#ifdef TRUCK
tank_control_t supplementary_tank;
#endif

//
//
fuel_level_t calculate_fuel_level() {
    short current_level = tank.level;

#ifdef TRUCK
    if (supplementary_tank.is_active) {
        current_level += supplementary_tank.level;
    }
#endif

    if (current_level < MINIMUM_LEVEL) {
        return alarm;
    }
    if (current_level < WARNING_LEVEL) {
        return warning;
    }
    return normal;
}
```

Bild 4: Implementation der Füllstandfunktion

2.2 Analyse der Software-Hierarchie

Die Menge der zu testenden Software-Varianten ergibt sich automatisch aus allen möglichen Konfigurationen einer Software. Für den Test ist dabei zu berücksichtigen, ob eine Variante (beispielsweise die Basis-Konfiguration) tatsächlich getestet werden kann: Ob es sich also um eine ausführbare Software handelt oder nur um eine Basis-Konfiguration, die alleine für sich noch keine funktionale Einheit bildet. Für solche abstrakten Varianten können unter Umständen auch schon abstrakte Tests definiert werden. Allerdings entstehen erst durch die weitere Implementierung solcher Tests für eine bestimmte funktionale Variante tatsächlich ausführbare Tests.

Auf der anderen Seite dient die Definition einer Varianten-Hierarchie vor allem der besseren Übersichtlichkeit bei sehr verschachtelten Software-Varianten. In der vorgestellten Methode werden die Varianten in einem Variantenbaum angelegt, der durchaus mehrstufig sein kann und die Variantenstruktur der Software abbildet. Dieser Baum dient der Orientierung, welche Tests auf welchem Varianten-Level erstellt werden sollen.

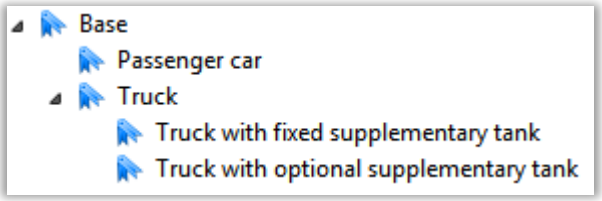

Bild 5: Variantenhierarchie

Das obige Beispiel zeigt eine Variantenhierarchie verschiedener Fahrzeuge, wobei es für LKW die Varianten mit fest verbundenem oder optional zuschaltbarem Zusatztank gibt. Die Variante "Truck" könnte in diesem Fall eine abstrakte Konfiguration oder auch eine konkrete Ausprägung eines Fahrzeugs sein.

2.3 Definition von Varianten-Tests

Tests können in zwei unterschiedliche Arten unterteilt werden:

- Basistests
- Variantentests

Die Basistests beziehen sich auf abstrakte Funktionalitäten, also beispielsweise auf die Grundkonfiguration eines Software-Moduls. Auf dieser Ebene können bereits alle potentiellen Testfälle der aus diesem Basismodul abgeleiteten Varianten definiert werden. Auch erste Testdaten können für Basistests bereits festgelegt werden, sie müssen aber nicht vollständig sein (weil man sie sowieso noch nicht ausführen möchte). Auf der obersten Ebene eines Variantenbaums lassen sich die möglichen Testfälle beispielsweise mit Hilfe eines Klassifikationsbaums definieren. Die untenstehende Testspezifikation berücksichtigt alle möglichen Variantenkonfigurationen unseres Beispiels.

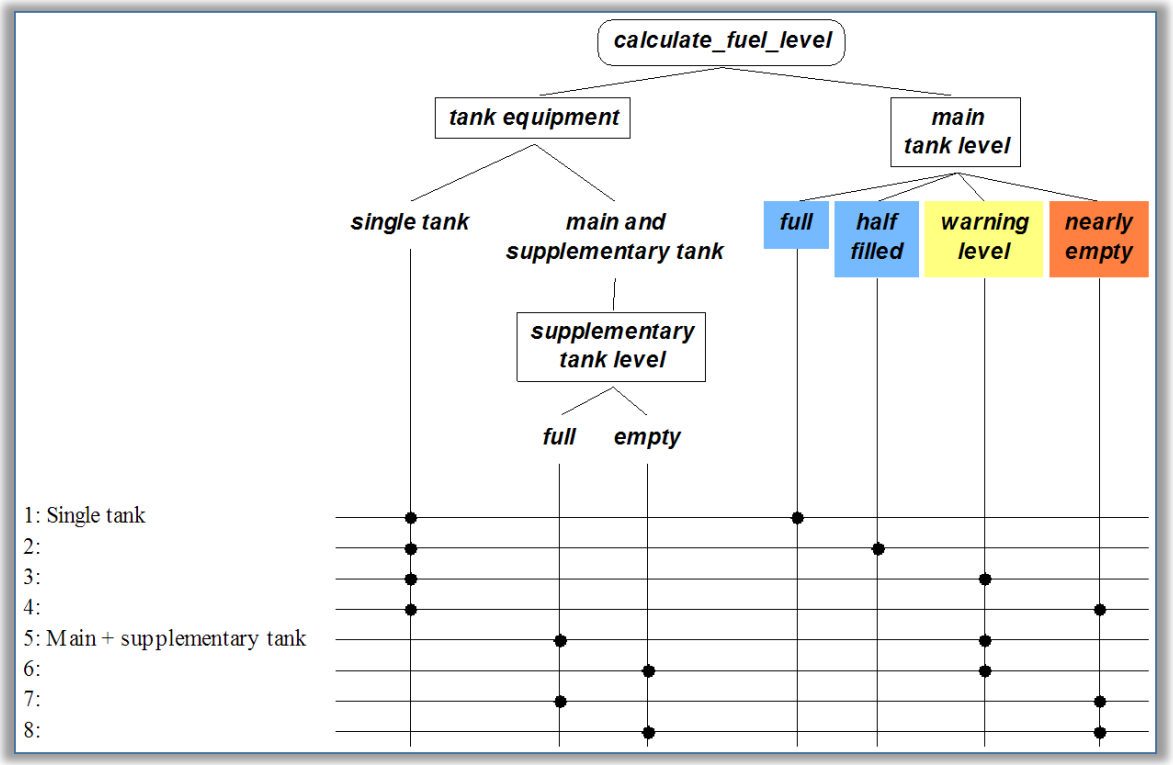

Bild 6: Testspezifikation für alle Varianten

Die Testspezifikation beschreibt die notwendigen Tests, die nun im Variantenbaum weitervererbt werden: Die geerbten Tests können in jeder Variante geändert, ausgeblendet oder mit spezifischen Tests ergänzt werden. In Bild 7 sind beispielsweise alle Tests ausgeblendet, die sich nicht auf die Variante „PKW“ beziehen.

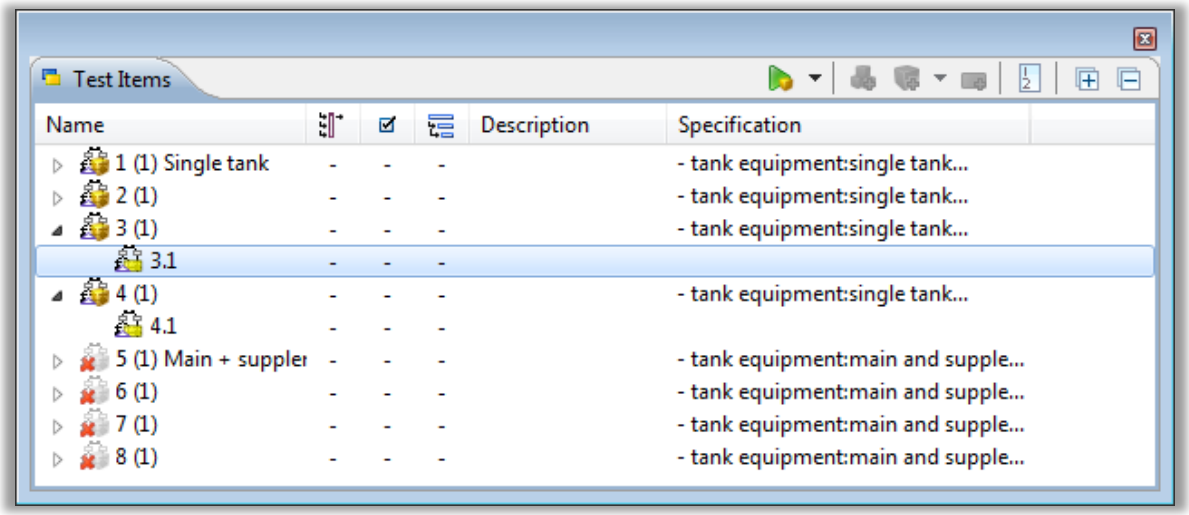

Bild 7: Testfälle für die Variante „PKW“

Jeder übergeordnete Variantentest muss somit nur einmalig angelegt werden und bei einer neuen Anforderung und Änderung der Applikation nur an einer Stelle gepflegt werden.

2.4 Regeln zur Vererbung von Testfällen

In unserem Beispiel wurden auf oberster Ebene sämtliche möglichen Testfälle definiert. Für manche Varianten (beispielweise "PKW") sind aber nur einige dieser Testfälle sinnvoll: Die übrigen Testfälle müssen für diese Variante ausgeblendet werden. Zusätzlich könnte es sinnvoll sein, weitere Tests auf der Ebene einer untergeordneten Variante hinzuzufügen. Folgende Operationen sind daher für die Vererbung von Testfällen notwendig:

- Ändern von geerbten Testdaten
- Löschen bzw. Ausblenden von geerbten Testfällen
- Hinzufügen zusätzlicher Testfälle

Auf der Ebene der Testdaten muss ebenfalls unterschieden werden, ob ein Wert geerbt oder nur lokal definiert wurde. Bei der Synchronisation der Varianten werden alle geerbten Werte aktualisiert. Für den Wert einer Variablen in einem Variantentest ergeben sich damit folgende Zustände:

- Wert wurde geerbt
- Wert wurde geerbt und überschrieben
- Wert wurde lokal für diesen Variantentest definiert

Über eine Farbcodierung lassen sich diese Werte wie in Bild 8 gezeigt leicht unterscheiden: Die hellblauen Werte wurden geerbt, der lilafarbene Wert wurde in der Variante „Truck“ überschrieben.

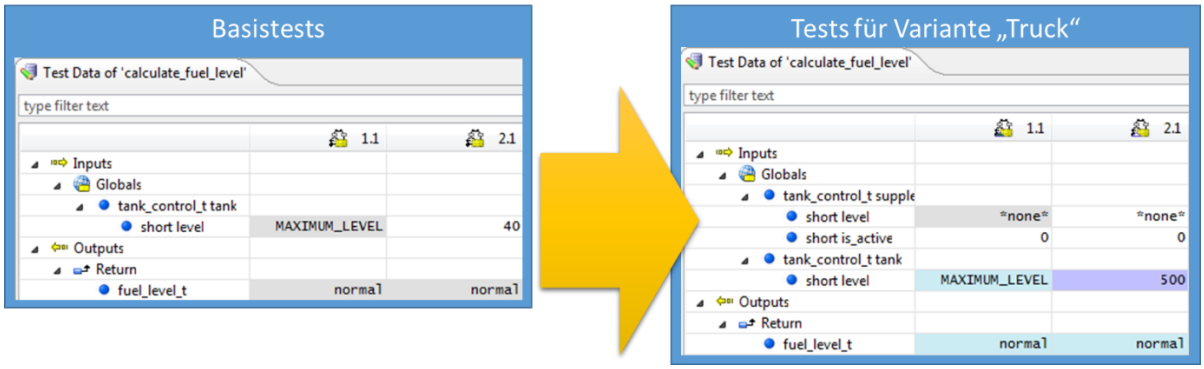

Bild 8: Farbcodierung von geerbten und überschriebenen Werten

Bei den Basistests wurden die meisten Werte bereits in der Testfallspezifikation im Klassifikationsbaum zugeordnet, daher erscheinen sie grau unterlegt. In Testfall 2.1 wurde für die Basistests bereits ein Wert von „40“ eingetragen, der für einen angenommenen 80 Liter Tank einen normalen Füllstand ergibt. Dieser Wert muss in der Variante „Truck“ deutlich erhöht werden, wenn man von einem 1000 Liter Tank ausgeht. Daher wurde der Wert von „level“ in der Variante mit „500“ überschrieben.

2.5 Eindeutige Identifizierung von Testfällen

Für die eindeutige Identifikation werden jedem Basistestfall ein "Universal Unique Identifier" (eine sogenannte UUID) zugeordnet. Diese UUIDs werden weltweit und zeitlich eindeutig generiert. Wenn ein Testfall nun an eine Variante vererbt wird, dann handelt es sich bei dem geerbten und für die Variante angepassten Testfall letztlich um denselben (Basis-)Testfall. Bei einem Review der Tests können die geerbten Testfälle damit eindeutig mit den Basistests oder den Tests einer anderen Variante verglichen werden.

Wichtig wird die UUID insbesondere bei der Synchronisation von Testfällen: Wenn Basistestfälle verschoben, gelöscht oder modifiziert werden, dann muss bei der Synchronisation der jeweils passende geerbte Testfall gesucht und aktualisiert werden. Nicht mehr vorhandene geerbte Testfälle müssen gelöscht werden. Die Zuordnung von UUIDS ermöglicht auch ein räumlich verteiltes Arbeiten an Variantentests, da sämtliche Testfälle eindeutig definiert sind und die Tests deshalb problemlos wieder zusammengeführt und aktualisiert werden können.

3 Ergebnisse

Für das gezeigte Variantenbeispiel wurde folgende Strategie für die Definition von Tests verwendet:

- Übernahme der Variantendefinitionen aus dem Applikationsdesign
- Definition aller möglichen Testfälle auf oberster Ebene für alle Varianten
- Ausblendung der nicht benötigten Testfälle in jeder Variante
- Ergänzung bzw. Implementierung der Tests jeweils getrennt in jeder Variante

Der Vorteil dieser Vorgehensweise liegt in der zentralen Spezifikation der Testfälle in einem einzigen Klassifikationsbaum. Das erhöht die Übersichtlichkeit und gibt einen vollständigen Überblick über alle Tests bei einem Review. Das Ausblenden von nicht benötigten Testfällen in Varianten ist relativ einfach und bewirkt gleichzeitig, dass sich der Testingenieur Gedanken darüber machen muss, welcher Testfall für seine Variante tatsächlich notwendig ist. Abweichend von dieser Strategie sind auch die nachfolgend beschriebenen alternativen Vorgehensweisen möglich.

3.1 "Natürliche" Vorgehensweise

Für einen Testingenieur ist es in der Regel leichter, Tests für eine konkrete Variante zu erstellen, als auf abstraktem Niveau über alle Varianten gleichzeitig nachzudenken. Je nach Art der zu testenden Software kann es daher sinnvoll sein, zunächst eine Variante komplett zu testen und dann die erzeugten Testfälle auf eine andere Variante zu übertragen.

Damit die Vererbung von Testfällen genutzt werden kann, ist aber eine hierarchische Definition der Varianten erforderlich. Es wäre daher notwendig, die Testfälle der fertig getesteten Variante auf eine übergeordnete (Basis-)Variante zu übertragen und ggf. zu verallgemeinern. Eine Übertragung der Testdaten setzt natürlich voraus, dass zumindest der größte Teil des Interfaces der Variantenfunktion auch auf dieser übergeordneten Ebene verfügbar ist.

3.2 Varianten im Klassifikationsbaum

Er wäre außerdem denkbar, Teile des Klassifikationsbaums jeweils variantenspezifisch zu filtern. Daraus würde dann jeweils eine eigene Testfallspezifikation pro Variante erzeugt. Der gesamte Klassifikationsbaum mit allen Varianten würde trotzdem auf der obersten Ebene der Variantenhierarchie verwaltet und bearbeitet werden. Zum Review der Tests könnte entweder der Gesamtbaum oder der jeweilige Teilbaum der Variante betrachtet werden.

Quellen

http://www.razorcat.com

Autor

Dipl.-Inform. Michael Wittner ist seit vielen Jahren im Bereich Software-Entwicklung und Test tätig. Nach dem Studium der Informatik an der TU Berlin arbeitete er als wissenschaftlicher Mitarbeiter der Daimler AG an der Entwicklung von Testmethoden und Testwerkzeugen. Seit 1997 ist er geschäftsführender Gesellschafter der Razorcat Development GmbH, dem Hersteller des Unit-Testtools TESSY und CTE sowie des Testmanagement-Tools ITE und der Testspezifikationssprache CCDL.

Kontakt

Email: mw@razorcat.com