Integration, Test und Debugging von C-Code gemeinsam mit UML-generierten Sourcen

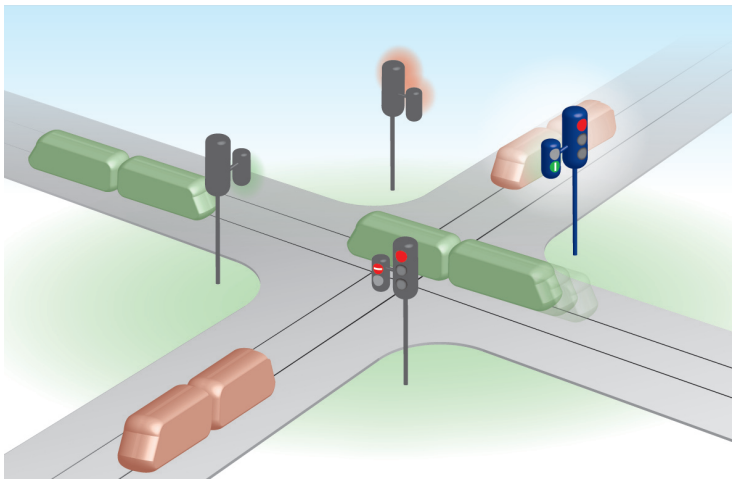

Bild 1: Eine Kreuzung für den Kfz-Verkehr soll um Straßenbahnampeln erweitert werden.

Einführung

Die wachsende Komplexität von Software wird in Zukunft ohne CASE-Tools nicht mehr zu bewältigen sein. Gerade UML ermöglicht ein durchgängiges Design von modularen Softwarekomponenten mit automatisierter Codegenerierung. Typischen Codierungsfehler werden auf diese Weise von vornherein ausgeschlossen.

Viele Unternehmen scheuen jedoch den vermeintlichen Aufwand der Umstellung auf diese neuen Techniken. Sie haben eine seit Jahrzehnten gewachsene Codebasis, bei der die Software zwar ständig erweitert und verbessert wurde, jedoch auch immer mehr „Spaghetti-Code“ entstanden ist. Es müssten nun Millionen funktionierende Codezeilen analysiert, umprogrammiert und schließlich erneut getestet werden.

Diese verständliche Angst ist meist unbegründet. Vorhandener C-Code kann (fast) ohne Änderungen in ein UML-Modell-Element portiert werden. Zukünftige Erweiterungen können dann als UML-Modell geschrieben, generiert und mit dem „alten“ C-Code integriert werden. Somit ist ein einfacher und stufenweiser Umstieg von C nach UML möglich. Im Folgenden wird anhand eines Beispiels vorgeführt:

1. Eine Überführung von „altem“ C-Code in UML.
2. Die Integration einer neuen, in UML codierten Funktionalität.
3. Das durchgängige Testen und Debuggen in UML, C++ und C.

Alter C-Code

Die Ausgangslage ist ein fertiges Projekt, das in C vorliegt. Das hier verwendete Beispiel implementiert eine Ampelschaltung (ein unter Software-Designern gern verwendetes, weil einfaches Beispiel). Wir haben eine Standardkreuzung mit vier Ampeln. Die jeweils gegenüberliegenden Ampeln sind gleich gesteuert. Die quer dazu laufenden Ampeln sind – natürlich – entgegengesetzt gesteuert. Die Phasen der Ampeln sind: rot – rotgelb – grün – gelb – rot, wobei in der Rot-Phase auf die andere Richtung umgeschaltet wird. Es gibt also eine kurze Zeitspanne, in der die Ampeln für alle Richtungen rot sind. Wir werden später sehen, dass dies wichtig ist.

Diese Schaltung ist in C codiert, als einfache Zustandssteuerung (switch-case) innerhalb einer Endlosschleife (while (1)). Die Endlosschleife befindet sich direkt in der „main()“ Routine der Applikation und wird gleich nach der Initialisierung aufgerufen (siehe Bild 2).

```
...
int main (void)
{
   horizontal = vertical = red;
   state = closed_to_h;

   while (1)
   {
      switch (state)
      {
          case closed_to_h:
             wait (1);
             horizontal = yellowred;
             state = yellowred_h;
             break;

          case yellowred_h:
             wait (3);
             horizontal = green
             state = green_h;
             break;
          ...
```

Bild 2: Bestehender C-Code für die Ampelschaltung; in der „main()“ Routine befindet sich eine Endlosschleife, welche die Schaltungslogik als Switch-Case-Konstrukt enthält.

Neue Anforderung

Die Straßenplaner haben entschieden, dass durch diese Kreuzung Straßenbahnen laufen müssen (Bild 1), und zwar in jeder Richtung (horizontal ⟩⟩

und vertikal). Für die Straßenbahnen ist die Kreuzung zunächst durch eine eigene Ampel gesperrt. Nach Anforderung durch den Zugführer wird die Kreuzung in der nächsten Rot-Phase für die Autos gesperrt und die anfordernde Straßenbahn erhält das Durchfahrtsrecht.

UML-Wrapper für C-Code

Natürlich könnten wir diese Anforderung in unseren C-Code „einflicken“. Aber auf diese Weise wird die Funktion immer größer und unübersichtlicher, was ja vermieden werden soll. Stattdessen gehen wir das Problem jetzt modular an und wollen es mit Hilfe der UML lösen.

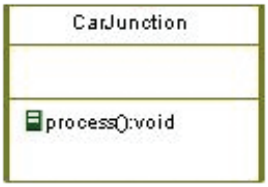

Bild 3: Die Klasse „CarJunction“ enthält die komplette „alte“ Applikation.

Zunächst müssen wir die vorhandene Applikation in unser Modell „überführen“. Das geht einfacher als gedacht: Wir erzeugen eine Klasse „CarJunction“, welche die komplette „alte“ Applikation enthält (siehe Bild 3).

Das einzige, was angepasst werden muss, sind die entsprechenden Schnittstellen. Die sonst übliche Endlosschleife in „main()“ wird schon im Framework bereitgestellt und muss somit entfallen. „main()“ wird in „processJunction()“ umbenannt und bildet den Eintrittspunkt. Als Austrittspunkt der Funktion wird der Zustand gewählt, in dem alle Ampeln auf rot sind. Nun können in der Rot-Phase, falls gewünscht, Aktionen durchgeführt werden (siehe Bild 4). — Das war’s.

```
void processJunction (void)
{
   horizontal = vertical = red;
   state = closed_to_h;

   do
   {
      switch (state)
      {
          case closed_to_h:
             wait (1);
             horizontal = yellowred
             state = yellowred_h;
             break;
  ........
          case yellow_v:
             wait (3);
             vertical = red;
             state = closed_to_h;
             break;

      } // switch (state)
   } while ((state != closed_to_h)&&(state != closed_to_v))
} // main()
```

Bild 4: Modifizierter C-Code, der in dieser Form in das UML-Werkzeug eingebunden werden kann.

UML-Design für neue Anforderung

Jetzt können wir uns schon um das Design der neuen Anforderung kümmern. Wir bauen uns eine Klasse „Junction“, welche die alte Ampelsteuerung als „CarJunction“ enthält, sowie die beiden neuen Straßenbahnampeln (Bild 5). Damit ist das Design fertig, und wir kümmern uns nun um das Verhalten der erweiterten Kreuzung. Hierzu wird ein Zustandsdiagramm verwendet.

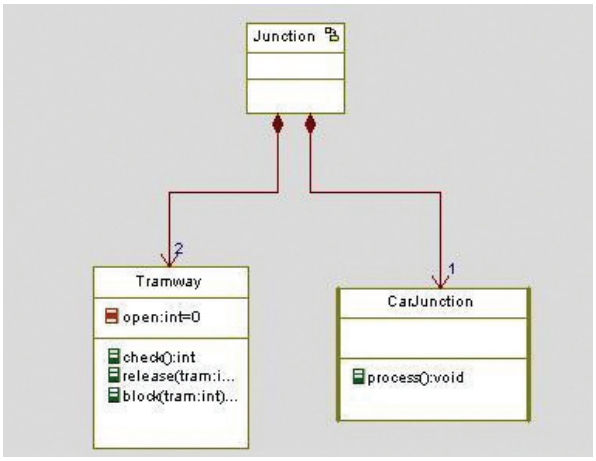

Bild 5: Die neue Klasse „Junction“ integriert die „alte“ Ampelschaltung und die beiden neuen Straßenbahnampeln.

Auch hier wird die alte Applikation als Ausgangsbasis genommen und das komplette Verhalten dieser Applikation als ein einziger Zustand der neuen Kreuzung modelliert (Zustand: cars). Siehe Bild 6 auf der gegenüberliegenden Seite. Wenn das neue Modell sich in diesem Zustand befindet, verhält es sich genauso wie die alte Applikation. Erst wenn das neue Feature angefordert wird – der Zugführer verlangt das Durchfahrtsrecht – wird der alte Code verlassen und die neue Modellierung startet. Im Zustandsdiagramm werden nun die Zustände und die Übergänge der Zustände für beide Straßenbahnlinien spezifiziert: warten – fahren – warten – gesperrt (eventuell mit Wechsel der Straßenbahnlinie). Damit ist die Modellierung der neuen Anforderung erledigt.

Code-Generierung

Nun erzeugen wir die lauffähige Applikation. ⟩⟩

Guten UML-Tools reicht die oben genannte Modellierung aus, um eine fertige Anwendung zu generieren. Jetzt ist es allerdings an der Zeit, einige Punkte zur Integration zu beachten: Die Ausgangs-Applikation ist in C geschrieben und bleibt dies auch. UML-Tools generieren typischerweise C++ Code. D.h. „CarJunction“ ist eine Wrapper-Klasse, die in den C-Code verweist. Dabei sind die üblichen C/C++ Übergänge zu beachten.

Meist wird zur Verwaltung des Modells ein RTOS verwendet, das entsprechend mit eingebunden werden muss. Ist dies alles einmal erledigt, reicht ein Knopfdruck – und aus dem Modell wird die fertige Applikation erzeugt.

Run in Target

Im embedded Bereich unterscheidet sich die Zielhardware vom Entwicklungsrechner. Das bedeutet, wir müssen jetzt die erzeugte Applikation in das so genannte „Target“ laden und dort laufen lassen. Dies kann mit jedem gängigen externen Tool durchgeführt werden, das Code auf ein Target laden kann.

Besonders einfach funktioniert dies, wenn das UML-Tool und der Debugger über eine gemeinsame (Software-)Schnittstelle kommunizieren. Zwischen dem UML-Tool Rhapsody von Telelogic und dem TRACE32-Debugger von Lauterbach besteht eine solche Integration. Diese ermöglicht, dass auf Knopfdruck im Modellierungswerkzeug die Applikation über die Debug-Schnittstelle ins Target geladen wird und – wenn gewünscht – sofort startet. Der komplette Zyklus Design – Modellierung – Generierung – Run ist damit aus einer Oberfläche heraus möglich.

Test in C

Es wurde eingangs schon erwähnt, dass das Testen und Debuggen in einer solchen heterogenen Umgebung bestimmte Anforderungen stellt. Es muss schließlich gewährleistet sein, dass die komplette Applikation in ihrer jeweiligen Funktion und Implementierung getestet werden kann.

Wir gehen zwar davon aus, dass wir eine schon funktionierende Applikation in C hatten. Trotzdem darf die Möglichkeit nicht fehlen, diese zu debuggen. Schließlich muss der Übergang der neuen Features zum „alten“ Bestand getestet und eventuell auch noch die alte Source betreut werden. Zumindest für diesen Teil der Applikation kann jeder handelsübliche Debugger verwendet werden, da das Debugging von C-Code im embedded Bereich heute Standard ist.

Test in C++

Interessanter wird es, wenn wir die neuen Features auf Source-Ebene testen wollen. Die meisten UML-Tools erzeugen C++ Code mit allen dazugehörigen Eigenschaften, wie Templates, Polymorphismus, Exception Handling, etc. Hier ist zu beachten, dass ⟩⟩

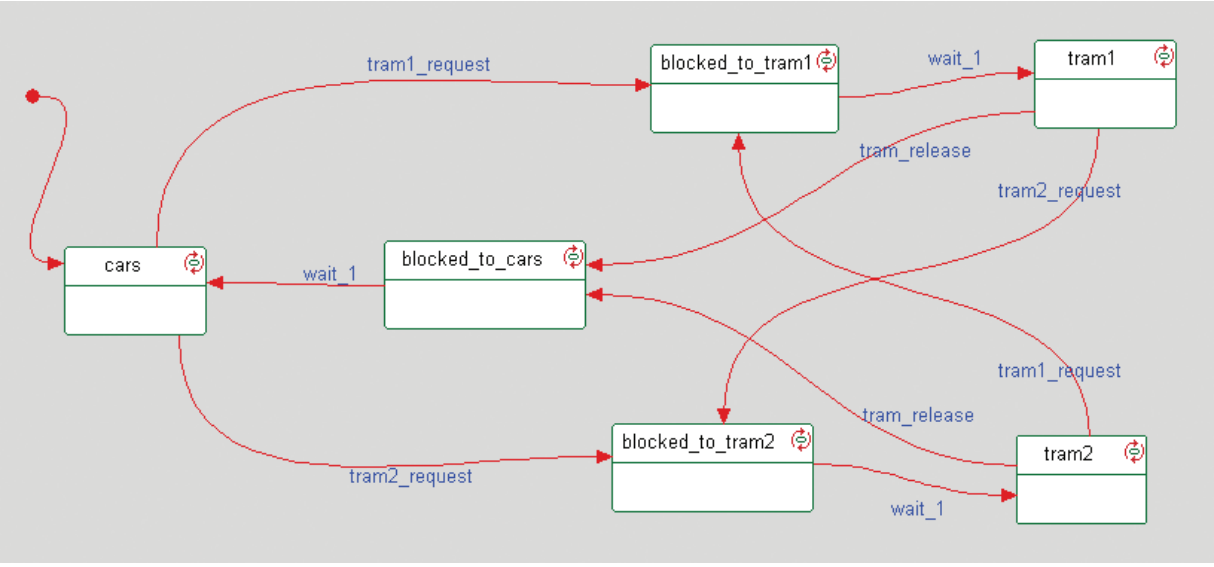

Bild 6: Das Zustandsdiagramm der neuen Ampelschaltung.

der eingesetzte Debugger den jeweiligen C++ Dialekt des Compilers voll beherrscht. Der TRACE32-Debugger von Lauterbach arbeitet mit allen gängigen C++ Compilern zusammen und gewährleistet so ein komfortables Debugging im objektorientierten C++ Code.

Test in UML

Aber eigentlich haben wir die Applikation ja gar nicht in C++ geschrieben, sondern in UML. Was liegt da näher, als auch auf der Modellierungsebene zu debuggen?

Rhapsody von Telelogic beispielsweise bietet hierzu verschiedene Möglichkeiten an.

Wenn das Verhalten der Applikation komplett in UML modelliert ist, kann aus dem Modell heraus der Ablauf bereits simuliert werden. Es ist somit für die reine Verhaltensanalyse keine aktuelle Zielhardware notwendig. Natürlich wird sich das Target zeitlich anders verhalten als die Simulation, daher kann die Applikation auch als so genannte „Animation“ auf dem echten Target laufen. Über einen Kommunikationskanal (seriell oder Ethernet) kann dabei das UML-Tool den Ablauf im Target steuern und visualisieren. So kann man beispielsweise „Single Steps“ durch *State Charts* durchführen oder Events einschleusen.

In Kombination mit dem TRACE32-Debugger kann diese Kommunikation über die Debug-Leitung erfolgen, wenn z.B. keine anderen freien Schnittstellen verfügbar sind. Über „Animation“ wird ein direktes Debugging im UML-Modell ermöglicht.

Integriertes Debugging UML -> C++ -> C

In dieser Situation arbeiten wir auf drei Ebenen: UML, C++ und C. Besonders im C++ Code möchte man, wenn man einen Fehler findet, diesen im Modell beheben und nicht im generierten C++ Code.

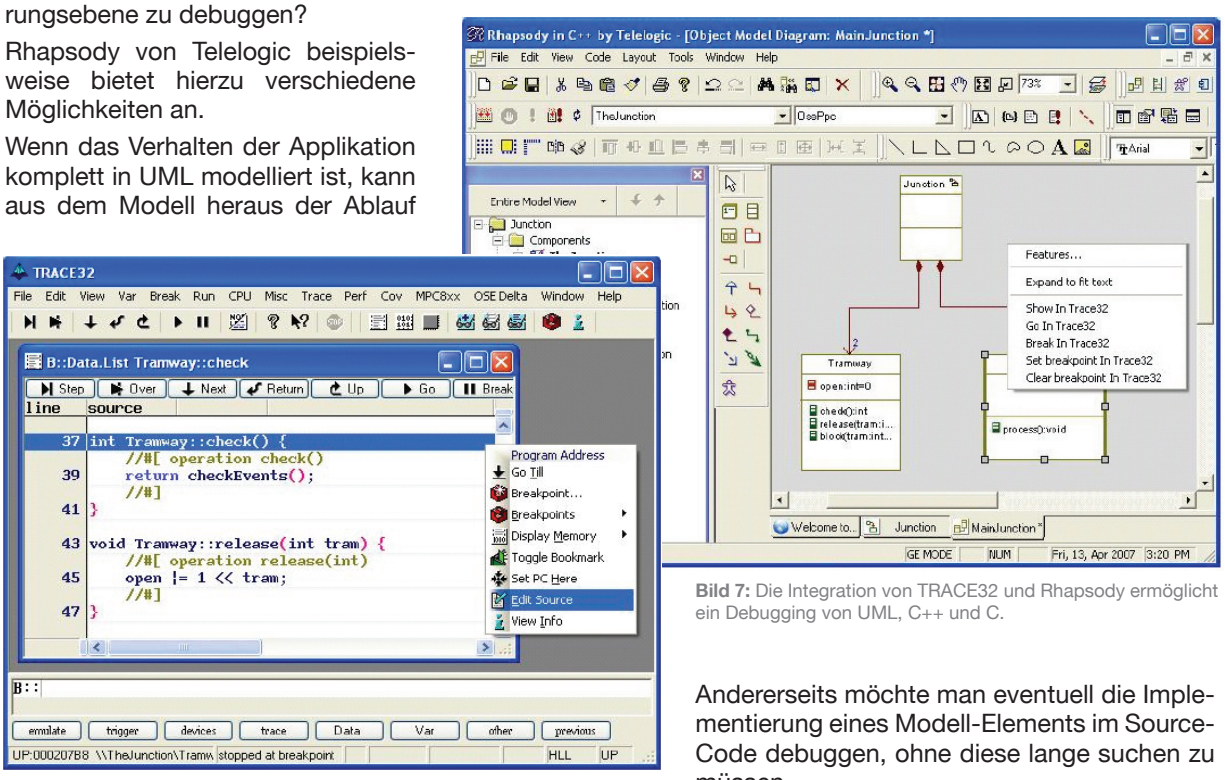

Bild 7: Die Integration von TRACE32 und Rhapsody ermöglicht ein Debugging von UML, C++ und C.

Andererseits möchte man eventuell die Implementierung eines Modell-Elements im Source-Code debuggen, ohne diese lange suchen zu müssen.

Die Integration von TRACE32 und Rhapsody bietet daher weitreichende Funktionalitäten, um diese Ebenen zu verbinden (Bild 7). So ist eine einfache wechselseitige Navigation implementiert. Ein Mausklick auf ein Modell-Element in Rhapsody genügt, um in TRACE32 den dazugehörigen Source-Code anzuzeigen. Es kann dann eine weitergehende Untersuchung von Variablen, Funktionsaufrufen etc. folgen. Umgekehrt ist es auch möglich, im Debugger auf eine Source-Zeile zu klicken und automatisch im UML-Tool das zugehörige Modell-Element zu markieren. Das ist besonders hilfreich, wenn man im ⟩⟩

Debugger einen Fehler findet, den man dann im Modell beheben will. Ein langwieriges Suchen und Navigieren zum entsprechenden Element entfällt so.

Es können im UML-Modell auch Debug-Breakpoints gesetzt und das Target in den Go- oder Stop-Status versetzt werden. Somit kann man einen Breakpoint auf eine Klasse setzen und das Target starten, alles innerhalb des UML-Tools. Der Debugger hält dann die Applikation an, sobald das zu untersuchende Element angesprungen wird.

Fazit

Es muss nicht immer ein komplettes Redesign sein. Gerade bei Projekten, die aus Zeit- oder Organisationsgründen kein komplettes Re-Engineering erlauben, ist oft ein stufenweiser Ansatz richtig. Die Umstellungsmaßnahmen halten sich dabei in Grenzen. Dafür eröffnet sich die Möglichkeit, die bestehende Software schrittweise in ein modernes und zukunftsweisendes CASE-Tool zu portieren.

Voraussetzung ist natürlich, dass die dabei eingesetzten Tools diesen Schritt unterstützen anstatt ihn zu behindern. Die Auswahl der Werkzeuge ist deshalb ein wichtiger Punkt. Schließlich müssen diese einen solchen Ansatz nicht nur zulassen, sondern auch fördern.

Lauterbach GmbH und Telelogic haben sich mit der Integration ihrer Tools TRACE32 und Rhapsody genau diesem Anforderungsprofil angenommen. Beide Firmen wollen ihren Kunden dabei helfen, die Qualität der Software trotz steigender Komplexität und immer mehr Anforderungen auf hohem Niveau zu halten. Einer Modernisierung alter Code-Basen in Richtung UML steht so nichts mehr im Wege.