Generative Softwareentwicklung zur Optimierung der Konstruktion eingebetteter Softwaresysteme am Beispiel einer Lenkungssteuerung

[1]Christian Berger, [1]Holger Rendel, [1]Bernhard Rumpe, [2]Fabian Wolf

[1]RWTH Aachen
Lehrstuhl für Software Engineering
Ahornstraße 55
52074 Aachen
www.se-rwth.de

[2]Volkswagen AG
Entwicklung Elektronik Software
38037 Braunschweig
Brieffach 3724
www.volkswagen.de

Kurzfassung

Software-Qualität und Qualitätssicherungsmaßnahmen sind insbesondere wichtig für softwarelastige Systeme, die kritische und hochverfügbare Funktionalitäten bereitstellen. Sind diese Systeme zudem ein variierender Teil einer Produktlinie, müssen analog zu den Hardware-Bauteilen und deren Verarbeitungsprozessen auch für die Software-Bestandteile Produktvarianten-übergreifende Prozesse etabliert werden. Häufig jedoch sind diese nur unzureichend durch Werkzeuge unterstützt. Gute Werkzeuge sind jedoch essentiell, um für wiederholbare Arbeitsabläufe einen hohen Grad an Automatisierung bei der Software-Konstruktion zu erreichen und so gleichzeitig Qualität per Konstruktion ins System zu integrieren.

Am Beispiel der Business Unit Braunschweig der Volkswagen AG und des durch sie entwickelten Lenksystems wird demonstriert, wie für Steuergeräte Produktlinien-übergreifend Automatisierung erreicht werden kann. Dazu wurde ein Werkzeug eingesetzt, mit dem der bislang manuell durchgeführte Integrationsprozess einzelner Software-Bauteile vereinfacht und gemäß AUTOSAR [Aut] teilweise automatisiert werden kann.

1. Einleitung und Motivation

Produktvarianten einer Produktlinie entstehen nach [PBL05] dadurch, dass entwickelte Artefakte zu einem Produkt variabel zusammengesetzt werden. Dies wird meistens in Form von manueller Software-Erstellung durchgeführt. Häufig sind hier jedoch ähnliche Schritte in leicht abgewandelter Form und mit anderen Parametern nötig. Hier besteht Potential für Optimierung und Automatisierung dieser Abläufe. Gleichzeitig werden so halbfertige Software-Komponenten in hoher Qualität wieder verwendbar und damit per Konstruktion qualitätsgesichert.

Im Lenksystem der Business Unit Braunschweig der Volkswagen AG werden die aus MATLAB/Simulink-Modellen automatisch generierten Funktionsmodule manuell mit der zugrundeliegenden Basis-Software verbunden. Dabei wird bei dieser manuellen Codierung

zusätzlicher Code erstellt, mit dem das Zusammenspiel der einzelnen Module und die im Rahmen der IEC-Norm für sicherheitskritische Systeme [IEC61508] vorhandene Überwachung getestet werden kann. Hier wurde eine Möglichkeit der Optimierung identifiziert und ein entsprechendes Werkzeug für die Erstellung dieses Schnittstellencodes implementiert.

Der Schnittstellencode besteht in der betrachteten Version aus ungefähr 11.000 Lines of Code (LOC). Zurzeit existieren im betrachteten Projekt etwa zehn Auslieferungs-Versionen sowie weitere Versuchsstände. Die Unterschiede zwischen jeweils zwei Versionen sind teilweise sehr gering. In zukünftigen Projekten wird jeweils ein ähnlicher Umfang erwartet.

Der zu generierende Schnittstellencode hat folgende wesentliche Aufgaben (Abbildung 1):

- Initialisieren der aus MATLAB/Simulink generierten Software
- Weiterleiten des periodischen Funktionsaufrufs in das Modul
- Finalisieren des Moduls (z.B. beim Ausschalten des Fahrzeugs Werte in den nicht-flüchtigen Speicher übertragen)
- Möglichkeit des Setzens von Testwerten für den Test der Überwachung

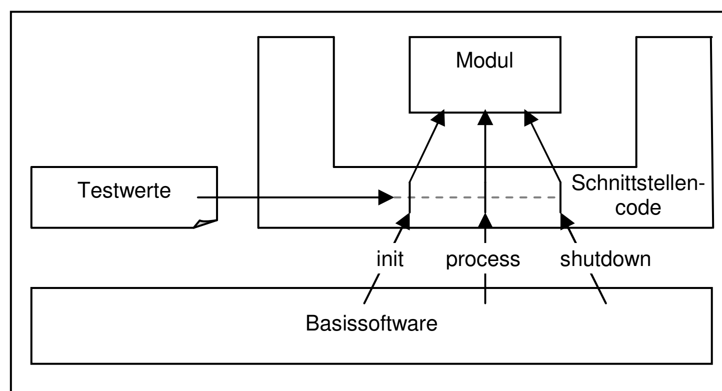

Abbildung 1: Aufgaben des Schnittstellencodes

Der Code, der das jeweilige Modul umschließt, ist für jedes Modul in einer einzelnen Datei abgelegt, deren Aufbau immer einer ähnlichen Struktur folgt.

2. Modellbasierter Entwicklungsprozess

Das eingesetzte Automatisierungswerkzeug erstellt aus einer Komponentenspezifikation den zugehörigen Programmcode. In Abbildung 2 ist dargestellt, welcher Schritt im Entwicklungsprozess des V-Modells dabei betrachtet wird.

Im Entwicklungsprozess werden Modelle eingesetzt, um die Entwicklung zu unterstützen und zu optimieren. Starke Verwendung finden dabei MATLAB/Simulink-Modelle, in der die Funktionalitäten der Lenkfunktionen in Form von unabhängigen und in sich abgeschlossenen Modulen modelliert werden. Die Erstellung des Schnittstellencodes wird bisher jedoch nicht

von Modellen unterstützt. Das liegt vor allem daran, dass keine geeigneten Beschreibungsformen und Werkzeuge dafür zur Verfügung stehen.

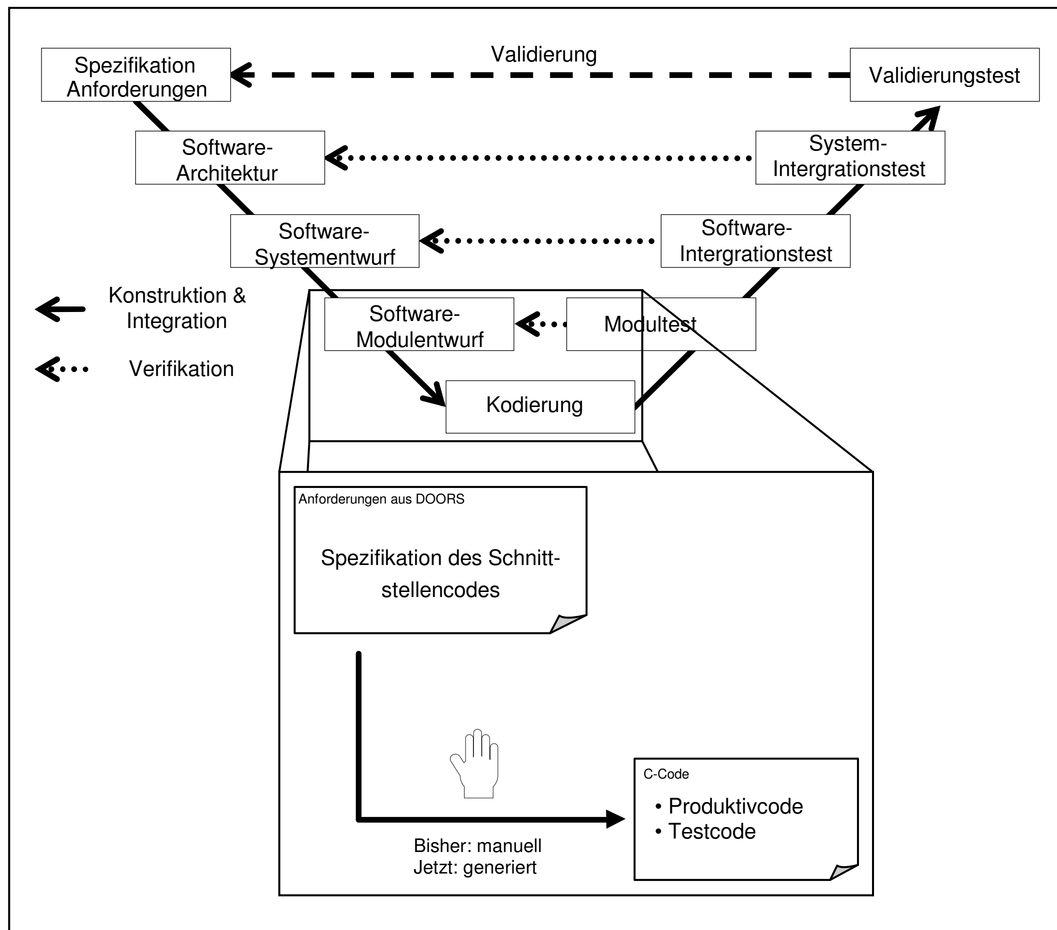

Abbildung 2: Einordnung in den VW-Entwicklungsprozess

Damit das Automatisierungswerkzeug den Prozess unterstützen kann, müssen folgende Anforderungen erfüllt werden:

1.) *Sprachverarbeitungsfähigkeit*: Das Werkzeug muss in der Lage sein, abstrakte Schnittstellen-Beschreibungen einzulesen und zu verarbeiten. Dabei von einer domänenspezifischen Sprache ausgegangen, die eine auf einen bestimmten Sachverhalt zugeschnittene Informationen beschreibt.

2.) *Trennung von unveränderlichen und variablen Anteilen*: Die Artefakte, die in dem betrachteten Prozess erstellt werden, haben sowohl unveränderliche als auch variable Anteile. Unveränderliche Teile dürfen nur einmal definiert und um den jeweils variablen Anteil bei der Generierung ergänzt werden.

3.) *Austauschbare Anbindung an das Anforderungsmanagement*: Das Automatisierungswerkzeug soll vom Anforderungswerkzeug DOORS [IBM] unabhängig sein und gegebenenfalls sollten nur Schnittstellen anzupassen sein.

4.) *Kompilierfähige Artefakte*: Die Artefakte, die das Automatisierungswerkzeug erzeugt müssen kompilierfähig sein. Eine Anpassung der Artefakte soll nicht stattfinden. Das heißt, dass die Artefakte vollständig sind und keine Fehler aufweisen dürfen.

5.) Reproduzierbare Generierung: Die generierten Artefakte können jederzeit wieder generiert werden (statt nur „One-Shot“) und dürfen auch daher nicht manuell angepasst werden.

6.) *Misra-Kompatibilität*: Sämtlicher Code, der im Prozess erstellt wird, muss Misra-kompatibel sein. Somit müssen die erstellten Artefakte auch Misra-Kompatibilität aufweisen.

7.) *Werkzeug soll qualitätsgesichert sein*: Da das Automatisierungswerkzeug zur Erstellung eines SIL3-Systems (Safety Integrity Level 3) [IEC61508] eingesetzt werden soll, muss eine hohe Qualität des Werkzeuges gewährleistet sein. Aus diesem Grund sind das Werkzeug und die generierten Artefakte entsprechend abzusichern.

3. Werkzeuge

Als System zur Verwaltung von Anforderungen kommt die Software DOORS zum Einsatz. In DOORS lassen sich Anforderungen erfassen und nachverfolgen. Zu jeder Anforderung können verschiedene Attribute erstellt werden, um weitere Informationen zu einzelnen Anforderungen anzulegen. So lassen sich beispielsweise Zugehörigkeiten zu verschiedenen Produkten spezifizieren. Abhängigkeiten oder Beziehungen zwischen den Anforderungen lassen sich über sogenannte Links festhalten. Zudem lassen sich DOORS-Artefakte durch die integrierte Skriptsprache DXL (DOORS Extension Language), die an C angelehnt ist, verarbeiten.

Zur Erstellung des Generators EVFGen wird das am Lehrstuhl entwickelte Generatorframework MontiCore [GKR+06, KRV07, GKR+08, KRV08] eingesetzt. MontiCore stellt wesentliche Funktionen für die Erstellung des Generators bereit:

- Definition der EVFGen DSL
- Erstellung eines Parsers für EVFGen-Modelle
- Einlesen der EVFGen-Modelle
- Verarbeitung der eingelesenen Daten
- Generierung von Artefakten

DSLs erlauben die effiziente Definition von Zusammenhängen oder Teilaspekten der realen Welt (Domäne) [DKV00, MHS05]. MontiCore stellt für diesen Zweck ein Framework bereit, welches einen Parser um die in der DSL beschriebenen Eingabedateien einzulesen, enthält. In diesem Framework lässt sich die eingelesene Instanz effektiv weiterverarbeiten.

Da beim Generieren unveränderte Anteile (z.B. Methodenköpfe) mit variablen Informationen (z.B. Methodennamen und -rümpfe) verbunden werden, ist die Benutzung von Vorlagen sinnvoll. Als eine bewährte Software-Umgebung zur Verarbeitung von Vorlagen ist die weit verbreitete Template-Engine Velocity [Vel] in MontiCore integriert.

Mit diesen Werkzeugen wurde der im folgenden Kapitel vorgestellte Generator entworfen und realisiert.

4. Der Generator EVFGen

Ansatzpunkt für die Optimierung und Automatisierung ist die Erstellung des Schnittstellencodes, welcher den aus MATLAB/Simulink-Modellen generierten Code für einzelne Teilfunktionalitäten (Module) mit der Basis-Software verbindet. Bei der Analyse wurde Optimierungspotential identifiziert.

Die Erstellung des Schnittstellencodes wird bisher manuell durchgeführt. Das in C-Dateien abgelegte Ergebnis beinhaltet außer dem eigentlichen Code auch noch eine Reihe von strukturierten Kommentaren, die Code-Teile mit Anforderungen aus der zugehörigen Spezifikation verknüpfen. Zusätzlich gibt es auch eine Integration von Code-Teilen, die für den Test nötig sind. Dieser wird über Compiler-Flags ein- oder ausgeschaltet. Dadurch ist die resultierende Datei relativ unübersichtlich.

Problematisch ist bei dieser Codeform insbesondere die Anfälligkeit gegenüber Fehlern bei Änderungen. Durch die wiederholte Nutzung in verschiedenen Kontexten (Produktlinie) und durch relativ viele notwendige Ergänzungen und Umbauten der Schnittstelle entsteht so ein QS-Risiko.

Der Aufbau des Schnittstellencodes folgt im Wesentlichen dem gleichen Schema. So ist immer eine Methode zum Initialisieren und zum Finalisieren der enthaltenen Module vorhanden. Ebenso gibt es eine Methode, die zyklisch ausgeführt wird und jeweils für einen Takt die Eingabedaten setzt, eine mögliche Abschaltung des Moduls prüft, den Aufruf an das enthaltene Funktions-Modul weiterreicht und nach dessen Ausführung die Ergebnisse ausliest. Zudem gibt es eine eingebettete Test-Methode, die nur im Rahmen des Überwachungstests ausgeführt wird.

Durch vorhandene Konventionen kann viel von diesem Inhalt sehr kompakt aufgeschrieben werden. Hierzu ist eine domänenspezifische Sprache sinnvoll. Für den Schnittstellencode ist diese auszugsweise im MontiCore-Format in Abbildung 3 gezeigt. Zur begrifflichen Unterscheidung wird diese Sprache als Envelopecode bezeichnet. Die dargestellte Grammatik beschreibt den Schnittstellencode, der einen bestimmten Namen (Zeile 3) und verschiedene Methoden hat (Zeile 4-7). Diese selbst werden jeweils nach einem bestimmten Schema definiert (Zeile 10-20). Die Verknüpfungen zu den im Anforderungswerkzeug hinterlegten Anforderungen werden über den Wert „rq“ in der Grammatik hergestellt.

```
01 grammar Envelope {
02
03   Envelopefunction = "envelope" name:IDENT "{"
04     Initialize
05     | Shutdown
06     | Process
07     | ...
08   "}";
09
10   Initialize = "initialize" rq:RQ "{"
11     ...
12   "}";
13
14   Shutdown = "shutdown" rq:RQ "{"
15     ...
16   "}";
17
18   Process = "process" rq:RQ "{"
19     ...
20   "}";
21
22   ...
23 }
```

Abbildung 3: Auszug aus der Envelope-Grammatik

Diese Grammatik definiert Beschreibungen für die Schnittstelle wie auszugsweise in Abbildung 4 dargestellt. Links ist der C-Code des Schnittstellencodes und rechts der Envelopecode dazu abgebildet. Es ist zu erkennen, dass der Name der entstehenden Datei, der an mehreren Stellen im C-Code auftaucht, nun nur noch einmal und damit redundanzfrei bereitgestellt wird. Generell wurde bei der Erstellung der Grammatik darauf geachtet, dass keine Redundanzen spezifiziert werden müssen und so eine wesentliche Quelle von fehlerhaften Inkonsistenzen vermieden wird.

001 /* file: Exx.c
... ...
230 void Exx_Process(void){
... ...
328 void Exx_Init(void){
... ...
356 void Exx_Shutdown(void){
... ...
370 end of file Exx.c*/
	001 envelope Exx {
	... ...
	011 init {
	... ...
	015 shutdown {
	... ...
	019 process {
	... ...
	072 }

Abbildung 4: Vergleich von C-Code und Envelope-Code

Durch den Einsatz des Envelopecodes (evf-Codes) kann das in Abbildung 2 skizzierte Vorgehen vereinfacht werden. Abbildung 5 zeigt den veränderten Erstellungsweg für den C-Code. Aus der Spezifikation wird zunächst manuell der Envelopecode erstellt. Dieser ist in dem

oben abgebildeten Format verfasst. Anschließend wird mit dem erstellten Automatisierungswerkzeug der C-Code erstellt.

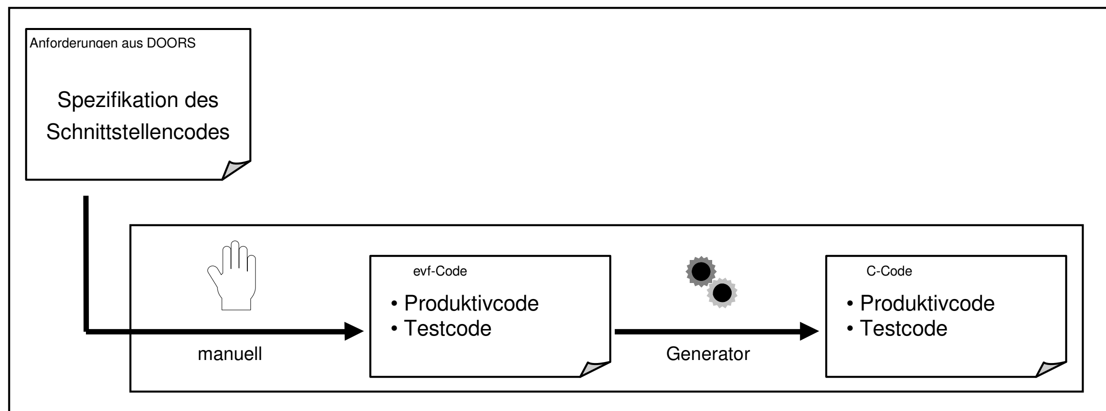

Abbildung 5: Erstellungsvorgang mit Generator

In einem zweiten Schritt wurde die Erstellung des C-Codes aus der Spezifikation weiter automatisiert. Der C-Code gliedert sich in einen Teil für das Produktivsystem und einen Teil, der ausschließlich zum Testen benötigt wird. Im Testteil gibt es die Möglichkeit, Ein- und Ausgangssignale zu manipulieren. Diese Manipulationen sind in den Anforderungen sehr detailliert und nach einem festen Schema in DOORS abgelegt.

Die Umsetzung in Testcode aus dieser Beschreibung erfolgt ebenfalls nach einem festen Schema. Diese Transformation ist in Form eines DXL-Skriptes implementiert worden. Da dieser Testcode nicht im ausgelieferten System vorhanden ist, müssen für diesen Teil nicht die strengen Vorschriften für die SIL3-Zertifizierung vorhanden sein. Jedoch kann durch Eingabefehler eine kritische Situation im Test entstehen. Diese Risiken werden durch ein schon vorhandenes Review über das Anforderungsdokument und eine Plausibilisierung bei der Transformation minimiert. In Verbindung mit dem erstellten Generator ergibt sich der in Abbildung 6 dargestellte Erstellungsvorgang.

5. Qualitätssicherung

Der Generator erfüllt wie folgt die in Abschnitt 2 festgelegten Kriterien:

1.) *Sprachverarbeitungsfähigkeit*: Die Sprachverarbeitung wird durch das Werkzeug MontiCore bereitgestellt. Die benutzte domänenspezifische Sprache wird durch Beispielinstanzen getestet. Dazu wurden für mehrere Releases Instanzen der evf-Codes erstellt, aus denen der C-Code zu dem jeweiligen Release erzeugt werden kann.

2.) *Trennung von statischen und variablen Anteilen*: Die verwendete Template-Engine erlaubt die Definition eines statischen Rahmens, der gezielt um variable Informationen angereichert werden kann. Der statische Anteil ist in einem übersichtlichen Template festgehalten, wodurch leicht Änderungen durchgeführt werden können.

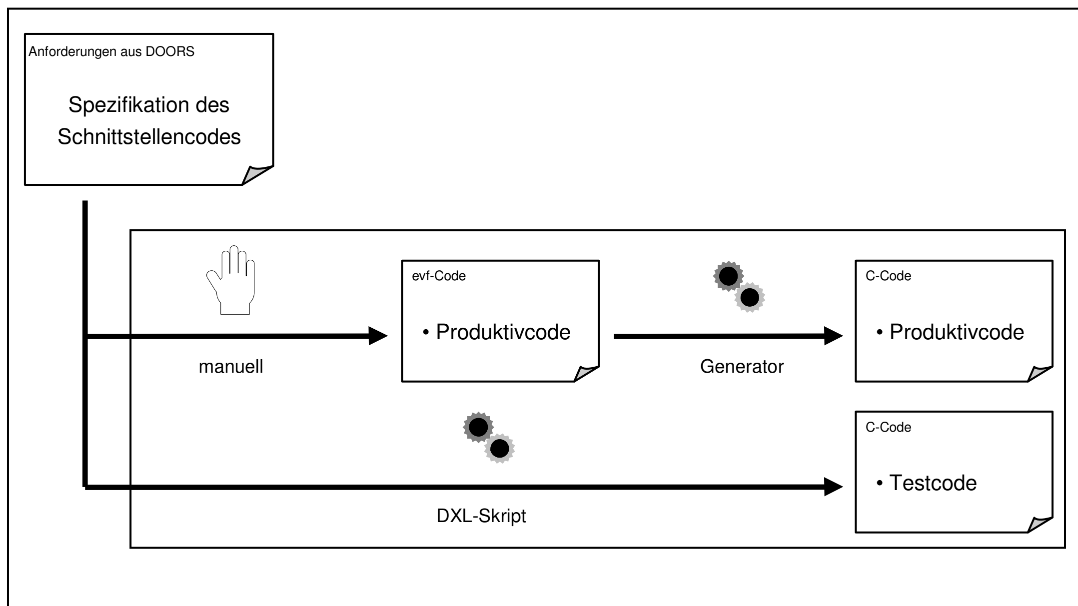

Abbildung 6: Erstellungsvorgang mit Generator und DXL-Skript

3.) *Austauschbare Anbindung an das Anforderungsmanagement*: Durch die Nutzung einer Zwischensprache ist der Generator unabhängig von einem Anforderungsmanagement-Werkzeug. Die Zwischensprache liegt als ASCII-Datei vor und ist sowohl durch MontiCore einlesbar als auch für den Entwickler einfach verständlich.

4.) *Kompilierfähige Artefakte*: Der Generator erzeugt syntaktisch korrekte C-Dateien. Syntaktische und semantische Fehler können lediglich durch falsche Eingaben im Schnittstellencode erzeugt werden. Die Sprache erlaubt nur die definierte Angabe von Namen, Operationen und Anforderungs-IDs, die automatisiert in C-Code umgesetzt und entsprechend getestet werden. An einigen Stellen ist explizit die Möglichkeit vorgesehen, direkt C-Code einzubetten. Diese Teile werden unverändert in den generierten C-Code übernommen, so dass hier Fehler in der Ausgabe durch Fehler bei der Eingabe ausgelöst werden könnten.

5.) Reproduzierbare Generierung: Durch die Verwendung von Modellen kann die Basis für die Generierung versioniert und so jederzeit der Generator wieder mit dieser ausgeführt werden.

6.) *Misra-Kompatibilität*: Der generierte C-Code entspricht dem bisher handgeschriebenen C-Code. Da für diesen auch schon Misra-Kompatibilität gefordert und diese durch Werkzeuge nachgewiesen wurde, ist auch der generierte C-Code Misra-kompatibel. Zur Absicherung wurden die Misra-Regeln durch das vorhandene Werkzeug PC-Lint nochmals überprüft. Hierbei traten keine Fehler auf. Eine direkte Einbettung in die Generierung ist durch das Tool in [BRV09] möglich, welches ebenfalls auf MontiCore aufsetzt.

7.) *Werkzeug soll qualitätsgesichert sein*: Das Gesamtwerkzeug besteht aus drei Komponenten. Die Komponenten Sprachverarbeitung sowie Template-Engine sind von den jeweiligen

Entwicklern entsprechend gestestet und qualitätsgesichert. Die erstellte dritte Komponente „Generator (EFVGen)“, die die anderen beiden nutzt, wird mit Unit Tests getestet. Als weitere Maßnahme wurde für mehrere Releases von Zieldateien der Generator durch einen automatisierten Test validiert. Dieser Test vergleicht den MD5 Hashwert des generierten Object-Codes der von EVFGen erzeugten C-Dateien mit dem der originalen C-Dateien. Hier traten keine Unterschiede auf. Eine weitere Qualitätssicherung findet bei der Anbindung der Template-Engine statt. Hier wird durch einen Mechanismus sichergestellt, dass die Template-Daten korrekt angegeben werden. Dieser Mechanismus sieht vor, dass in den Template-Klassen angegeben wird, welche Attribute zwingend belegt werden müssen und welche nur optional sind. Sollte dies für ein Attribut nicht erfüllt sein oder ein Attribut nicht in dieser Weise definiert sein, wird eine Warnmeldung ausgegeben.

6. Anwendbarkeit im Projekt

Im Rahmen Qualitätssicherung wurde auch der Nutzen des erstellten Automatisierungswerkzeuges evaluiert. Es wurde auf Basis der Eingabe- und Zieldateien für ein Release ermittelt, wie hoch die Einsparungen durch den Einsatz von generativer Softwareentwicklung sind. Das Ergebnis ist eine Einsparung von 82% der zu erstellenden Dateiinhalte, der zu einem deutlichen Teil auf redundanzfreie Darstellungen zurückzuführen ist. Diese fördert daher die Übersichtlichkeit und damit die Fehlersuche in der Datei, führt aber insbesondere zu weniger Inkonsistenzen und ist daher per-se qualitativ hochwertiger und weniger aufwändig zu warten.

Eine weitere Einsparung ergibt sich bei der Erstellung des Schnittstellencodes. Die automatische Erstellung des C-Codes aus dem kompakt formulierten Envelopecode nimmt nur wenige Sekunden in Anspruch. Review-Zeit kann ebenfalls eingespart werden. Das Review des C-Codes ist durch die Sicherheitsanforderungen und den Entwicklungsprozess zwar weiterhin nötig, jedoch wird der Testcode vom entwickelten DXL-Skript in eine separate Datei geschrieben, für die ein weniger umfangreiches Review wie für den Produktivcode nötig ist.

Durch Einsatz des Generators wurden zudem Tippfehler in den Kommentaren des bisher manuell erstellten Schnittstellencodes aufgedeckt. Diese Kommentare realisieren die Nachverfolgung von Anforderungen zu DOORS. Die Sicherheit des Systems ist damit nicht gefährdet, jedoch sind eventuell Änderungen der Anforderungen im Quellcode nicht nachzuvollziehen gewesen.

7. Zusammenfassung und Ausblick

In diesem Beitrag wurde die Optimierung und Automatisierung im Bereich der Erstellung von Schnittstellencode zwischen der Basis-Software und aus Modellen generierten Modulen dargestellt. Basierend auf verschiedenen Anforderungen sowie unter Berücksichtigung der [IEC61508] wurde ein Automatisierungswerkzeug in zwei Iterationen entwickelt. Im ersten

Schritt wurde eine Zwischensprache mit einem Generator für die Erstellung des Schnittstellencodes eingeführt. Danach wurde darauf aufbauend eine vollautomatische Generierung für den Testcode aus dem Anforderungsdokument implementiert. Durch diese Schritte ist der manuell zu erstellende Anteil deutlich redundanzfreier und kompakter, die den Erstellungsprozess verkürzen und optimieren.

Die vorgestellte Automatisierungslösung ist auch eine essentielle Voraussetzung für die Erweiterung auf Softwareproduktlinien. Dazu lässt sich ein Feature-Modell integrieren, mit dem eine produktspezifische Zusammenstellung von Modulen ausgewählt und der dazugehörige Code entsprechend generiert werden kann. Hierzu können ebenfalls domänenspezifische Sprachen eingesetzt werden [BKRR09].

Eine derzeit angedachte Erweiterungsmöglichkeit besteht im Einsatz von des Requirements Interchange Format (RIF) [Rif]. Hier könnte vor allem bei der Testcode-Erstellung eine Unabhängigkeit von DOORS erreicht werden.

Literatur

[Aut] Automotive Open System Architecture (AUTOSAR) Website `http://www.autosar.org`, besucht am 22.10.2009.

[BKRR09] C. Berger, H. Krahn, H. Rendel, B. Rumpe: Feature-basierte Modellierung und Verarbeitung von Produktlinien am Beispiel eingebetteter Software. In: Tagungsband des Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter Systeme V. Informatik-Bericht 2009-01, Institut für Software Systems Engineering, TU Braunschweig, 2009

[BRV09] C. Berger, B. Rumpe, S. Völkel: Extensible Validation Framework for DSLs using MontiCore on the Example of Coding Guidelines. In: Symposium on Automotive/Avionics Systems Engineering 2009 (SAASE 09), San Diego, California, USA, Oktober 2009

[DKV00] A. van Deursen, P. Klint, J. Visser: Domain-Specific Languages: An Annotated Bibliography. SIGPLAN Not., ACM, 2000, 35, 26-36.

[GKR+06] Grönniger, H.; Krahn, H.; Rumpe, B.; Schindler, M.; Völkel, S.: MontiCore 1.0 – Ein Framework zur Erstellung und Verarbeitung domänenspezifischer Sprachen. Technischer Report, Informatik-Bericht 2006-04, Institut für Software Systems Engineering, TU Braunschweig, 2006.

[GKR+08] H. Grönniger, H. Krahn, B. Rumpe, M. Schindler, S. Völkel. MontiCore: A Framework for the Development of Textual Domain Specific Languages. In 30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10–18, 2008, Companion Volume, Seiten 925–926, 2008.

[IBM] IBM Rational DOORS Website: `http://www-01.ibm.com/software/awdtools/doors/`, besucht am 22.10.2009.

[IEC61508] International Electrotechnical Commission. IEC61508: Funktionale Sicherheit sicherheitsbezogener elektrischer/elektronischer/programmierbarelektronischer Systeme, 2001.

[KRV07] Krahn, H.; Rumpe, B.; Völkel, S.: Integrated Definition of Abstract and Concrete Syntax for Textual Languages. Proceedings of Models 2007, 2007, 286-300.

[KRV08] Krahn, H.; Rumpe, B.; Völkel, S.: MontiCore: Modular Development of Textual Domain Specific Languages. Proceedings of Tools Europe, 2008.

[MHS05] M. Mernik, J. Heering, A.M. Sloane: When and how to develop domain-specific languages. ACM Comput. Surv., ACM Press, 2005, 37, 316-344.

[PBL05] K. Pohl, G. Böckle, F. v.d. Linden: Software Product Line Engineering: Foundations, Principles, and Techniques. Springer, 2005.

[Rif] Requirements Interchange Format Website der Herstellerinitiative Software: `http://www.automotive-his.de/rif`, besucht am 22.10.2009.

[Vel] Apache Velocity Website: `http://velocity.apache.org`, besucht am 22.10.2009.