Tran, Quang Minh; Winkler, Jonas Paul; Dziobek, Christian

SLRefactor: Ein Refactoring-Ansatz für Simulink-Modelle

Tran, Quang Minh; Winkler, Jonas Paul; Dziobek, Christian (2015): SLRefactor. Ein Refactoring-Ansatz für Simulink-Modelle. - In: Informatik 2015. Bonn: Gesellschaft für Informatik.
URI: https://dl.gi.de/handle/20.500.12116/2151.

Douglas Cunningham, Petra Hofstedt, Klaus Meer, Ingo Schmitt (Hrsg.): INFORMATIK 2015
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2015

SLRefactor: Ein Refactoring-Ansatz für Simulink-Modelle

Quang Minh Tran [1] Jonas Winkler [2] Christian Dziobek [3]

Abstract: Bei der Funktionsmodellierung ist die Veränderung und Erweiterung der Struktur eines Modells eine häufig durchgeführte Aktivität. Während es bereits Refactoring-Ansätze für textuelle Programmiersprachen wie Java, C# usw. gibt, fehlt ein vergleichbarer, integraler und durchgehender Ansatz für Simulink-Modelle. Wir haben einen automatisierten Refactoring-Ansatz (im Folgenden SLRefactor-Ansatz genannt) für Simulink-Modelle erfolgreich entwickelt, der in einem Zeitraum von ca. zwei Jahren in der Serienentwicklung bei der Daimler AG erprobt und eingesetzt wurde. In diesem Beitrag wird der SLRefactor-Ansatz anhand eines ausführlichen Beispiels erläutert und es wird über die Erfahrungen beim produktiven Einsatz des Ansatzes und über die dabei gewonnenen Erkenntnisse berichtet.

Keywords: Modellbasierte Entwicklung, Simulink, Refactoring, Transformation

1 Einführung

Zur Beherrschung der kontinuierlich zunehmenden Komplexität wird Fahrzeugsoftware in der Automobilindustrie modellbasiert entwickelt. Hierzu hat sich MATLAB/Simulink [Ma14] als Defacto-Standard bei vielen großen Automobilherstellen, wie zum Beispiel bei der Daimler AG [DRW12], etabliert. Mit MATLAB/Simulink werden Steuer-/Regelalgorithmen in Fahrzeugsoftware als Simulink-Blockdiagramme modelliert. Simulink bietet viele Vorteile. Diese bestehen insbesondere in Hinblick auf übersichtliche grafische Darstellung und hohe Freiheitsgrade beim Entwurf. Der Einsatz der modellbasierten Entwicklung hat zur Folge, dass die Erstellung von Funktionsmodellen ein wesentlicher Schritt bei der Entwicklung von Fahrzeugfunktionen in Kraftfahrzeugen geworden ist. Aus den Modellen wird durch den Einsatz von Codegeneratoren die Funktionssoftware für die Steuergeräte im Fahrzeug generiert. Daher ist sowohl die Qualität der Modelle in Bezug auf Struktur und Anordnung der Blöcke, als auch die Benennung von Blöcken, Signalen und Ports von großer Bedeutung.

Bei der Erstellung, Überarbeitung und Weiterentwicklung eines Modells werden typischerweise zahlreiche strukturelle Änderungen durchgeführt, um bestehende Modellteile mit Blick auf verbesserte Struktur neu zu organisieren, überflüssige Modellteile zu entfernen oder das Modell strukturell auf Erweiterungen vorzubereiten. Mit den aktuellen Mitteln des Simulink-Editors (aktuell Version: R2014b) lassen sich diese Ziele nur mit einem großen manuellen Aufwand erreichen, weil dieser lediglich elementare Bearbeitungsoperationen bereitstellt. Der Editor bietet Operationen zum Platzieren, Verbinden und Grup-

[1] Daimler Center for Automotive IT Innovations, Ernst-Reuter-Platz 7, Berlin, quang.tranminh@dcaiti.com
[2] Daimler Center for Automotive IT Innovations, Ernst-Reuter-Platz 7, Berlin, jonas.winkler@dcaiti.com
[3] Daimler AG, 71059 Sindelfingen, christian.dziobek@daimler.com

pieren von Blöcken. Allerdings fehlen höherwertige funktionsorientierte und kontextabhängige Operationen, die die Modellerstellung und Modellstrukturierung als auch die Änderung bestehender Struktur effizient unterstützen. Aus diesem Grund muss derzeit beispielsweise eine Verbindung zweier Blöcke, die miteinander Informationen austauschen, vom Anwender durch sequentielle Anwendung vieler elementarer Bearbeitungsoperationen (z.B. Hinzufügen von *Inport-/Outport*-Blöcken und Linien) erstellt werden. Es fehlt eine Operation, die diese Verbindung in einem Schritt erstellt, gegebenenfalls auch über mehrere Hierarchieebenen hinweg.

In den Veröffentlichungen von den Autoren [QD13] und [TWD13] wurden die ersten Ideen eines Ansatzes zum Refactoring von Simulink-Modellen vorgestellt. Seitdem wird der Ansatz weiterentwickelt und in der Vor- und Serienentwicklung bei der Daimler AG erprobt. In diesem Beitrag wird der aktuelle Stand dieses Ansatzes beschrieben, der *SLRefactor* genannt wird. Dazu werden zunächst kurz die Basiselemente von Simulink-Modellen vorgestellt.

2 Grundlegende Simulink-Modellierungselemente

In diesem Abschnitt werden die wichtigsten Simulink-Modellierungselemente kurz beschrieben. Abb. 1 zeigt dazu ein einfaches Simulink-Modell.

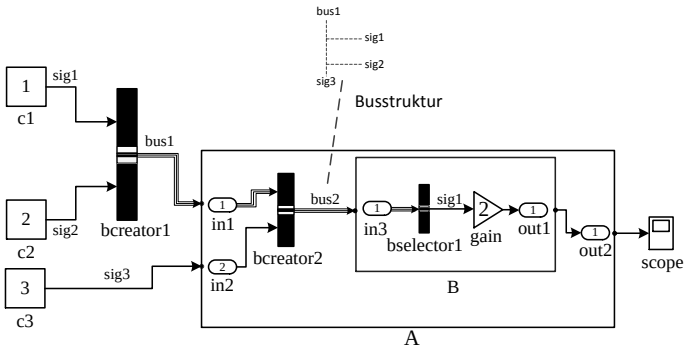


Abb. 1: Beispiel eines Simulink-Blockdiagramms

Ein Simulink-Blockdiagramm ist ein Signalflussgraph und besteht aus Blöcken und Linien. Blöcke führen in der Regel Berechnungen durch, beispielsweise Addition und Integration. Die Eingabe- und Ausgabeschnittstelle eines Blocks wird durch seine Eingangs- und Ausgangsports definiert. Blöcke werden durch Linien verbunden. Über diese Linien werden Daten in Form von Signalen zwischen den Blöcken übertragen. Ein Subsystem ist ein besonderer Block, der andere Blöcke einschließlich anderer Subsysteme enthalten kann. Subsysteme ermöglichen es, Simulink-Modelle hierarchisch zu strukturieren. In Abb. 1 enthält das Subsystem `A` die *Inport*-Blöcke `in1` und `in2`, den *BusCreator*-Block `bcreator2`, das Subsystem `B` und den *Outport*-Block `out2`. Ein Subsystem hat ebenfalls Eingangs- und Ausgangsports, die seine Eingabe- und Ausgabeschnittstelle beschreiben. Mehrere Signale können durch *BusCreator*-Blöcke zu Bussen zusammengefasst werden.

Busse können andere Busse enthalten und weisen damit eine baumartige Signalstruktur auf. Ein Signal in einem Bus kann durch einen *BusSelector*-Block ausgewählt werden.

In Abb. 1 werden die Signale `sig1` und `sig2` durch den *BusCreator*-Block `bcreator1` zu dem Bus `bus1` zusammengefasst. Dieser Bus wird wiederum zusammen mit dem Signal `sig3` durch den *BusCreator*-Block `bcreator2` zu dem größeren Bus `bus2` gruppiert. Die Struktur des Busses `bus2` wird in der Abbildung veranschaulicht. Das Signal `sig1` wird mit Hilfe des *BusSelector*-Blocks aus `bus2` selektiert.

3 Problemstellung

Wie eingangs erwähnt ist die Erstellung und Weiterentwicklung von Simulink-Modellen ein sehr aufwändiger und zeitintensiver Entwicklungsschritt. Simulink-Modelle mit mehr als 20 000 Blöcken sind in der Praxis keine Seltenheit. Zur Beherrschung der hohen Komplexität wird ein Simulink-Modell typischerweise hierarchisch unter Verwendung von Subsystemen organisiert. Auf oberster Modellebene wird die Gesamtfunktionalität in mehrere Module unterteilt. Die durch Subsysteme realisierten Module kapseln zusammenhängende Funktionsanteile und tauschen untereinander Informationen aus. Da zwischen Modulen auf oberster Ebene sehr viele Signale übertragen werden, werden häufig Signale zu Bussen zusammengefasst. Innerhalb eines Moduls wird die Funktionsaufteilung hierarchisch weiter verfeinert. Abb. 2 zeigt die oberste Ebene eines solchen Modells. In diesem Modell kommunizieren die Module `F1`, `F2` und `F3` untereinander über Busse. Eingangs- und Ausgangssignale sind ebenfalls Busse.

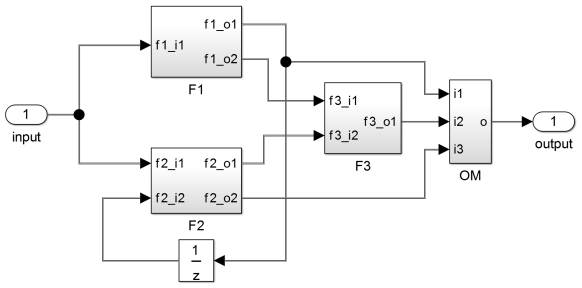


Abb. 2: Eine typische Modularchitektur des Modells

Strukturänderung von Modellen ist eine häufige Aktivität bei der Modellierung. Insbesondere sollen häufig die Schnittstellen zwischen den Modulen modelliert werden. Als nächstes sollen Module restrukturiert werden, wie zum Beispiel durch das Zerlegen eines Moduls oder Zusammenfassen mehrerer Module. Wenn das Modell eine gute Struktur erhält, sollen häufig bestimmte Funktionalitäten im Modell erweitert oder geändert werden. Dazu müssen unter anderem Signale von einem Modul zu einem anderen Modul weitergeleitet oder unnötige Signale entfernt werden. Bei jeder Strukturänderung muss darauf geachtet werden, die für das Modell vereinbarten Muster beizubehalten.

Um alle diese Aufgaben zu erledigen, fehlt allerdings integrierte Unterstützung für höherwertige Operationen in der heutigen Simulink-Umgebung. Vielmehr muss der Entwick-

ler heute auf die elementaren Editierungsoperationen wie beispielsweise Hinzufügen von Blöcken und Linien zurückgreifen.

4 Katalog von Refactoring-Operationen für Simulink

Ausgehend von einer detaillierten Analyse von typischen Anwendungsfällen bei der Bearbeitung von Simulink-Modellen ist ein Katalog von Refactoring-Operationen für Simulink entstanden. Dieser Katalog ist an einen vergleichbaren Katalog für objektorientierte Programmiersprachen [Fo99] angelehnt[3]. Abb. 3 zeigt den Katalog in Form einer Mind-Map. Der Katalog enthält alle Refactoring-Operationen und gruppiert diese anhand der Art der am Simulink-Modell durchgeführten Änderung. Im Folgenden werden die Hauptkategorien dieses Katalogs vorgestellt.

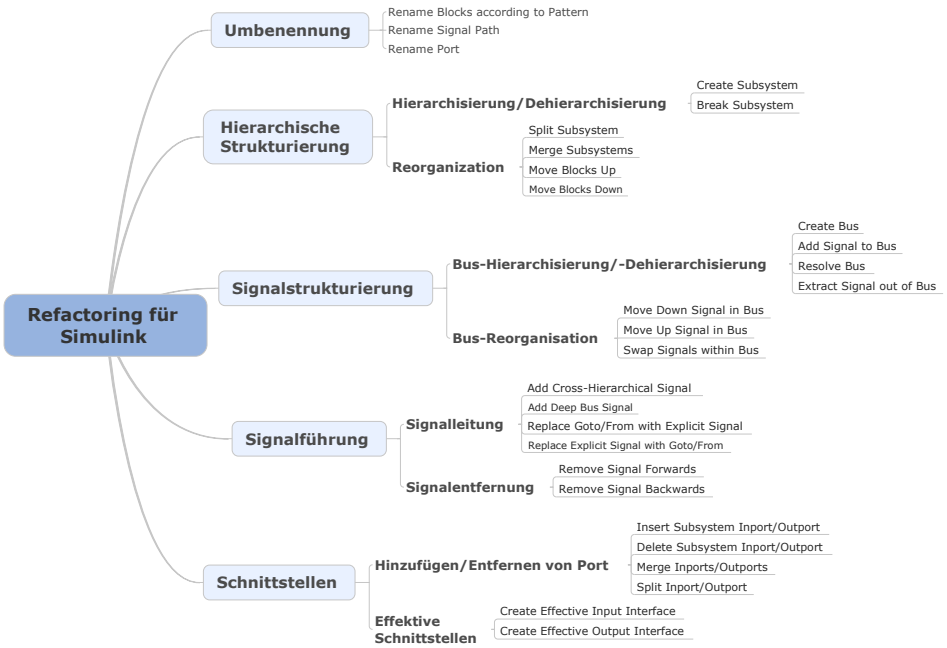


Abb. 3: Katalog von Refactoring-Operation für Simulink

Umbenennung: In einem Simulink-Modell sind die Namen von Blöcken, Subsystemen, Signalen und Ports ein wichtiges Beschreibungselement und beeinflussen unmittelbar die Verständlichkeit und Lesbarkeit des Modells. Der Simulink-Editor unterstützt bereits die Umbenennung einzelner Elemente. Allerdings sind derzeit komplexe Umbenennungen, wie zum Beispiel die Umbenennung aller Ports entlang eines Signalpfades oder die Anwendung eines Namensmusters auf alle Elemente innerhalb eines Subsystems nicht möglich. Die Refactoring-Operationen in dieser Kategorie ermöglichen eine effiziente Umbenennung von Blöcken, Signalpfaden und Ports.

[3] Martin Fowler pflegt auf seiner Webseite einen aktuellen Katalog von Refactoring-Operationen für objektorientierte Programmiersprachen. Der Katalog ist abrufbar unter der Adresse: http://refactoring.com/catalog/

Hierarchische Strukturierung: Bei der Modellerstellung spielt die Partitionierung eines Modells in hierarchisch angeordnete Subsysteme sowie die Definition der Schnittstellen und die Festlegung des Informationsflusses zwischen den Subsystemen durch Signale eine große Rolle. Diese Kategorie umfasst Refactoring-Operationen zur Bearbeitung der Modellstruktur. *Create Subsystem* fasst Blöcke zu einem Subsystem zusammen, während die inverse Operation *Break Subsystem* ein Subsystem durch seinen Inhalt ersetzt. *Move Blocks Up* verschiebt Blöcke auf die nächst höhere Subsystemebene und *Move Blocks Down* verschiebt Blöcke auf die nächst niedrigere Subsystemebene. *Split Subsystem* teilt ein Subsystem in zwei kleinere Subsysteme. Die dazu inverse Operation *Merge Subsystems* fasst zwei Subsysteme zu einem einzigen Subsystem zusammen. Bei all diesen Operationen werden die Signalbeziehungen der verschobenen Blöcke beibehalten.

Signalstrukturierung: Häufig werden Signale zu Bussen zusammengefasst, um so logische Zusammenhänge zu definieren und damit einhergehend die visuelle Komplexität zu reduzieren. Die Refactoring-Operationen dieser Kategorie dienen der Bearbeitung hierarchischer Busstrukturen. *Create Bus* erzeugt einen Bus aus mehreren Signalen. *Add Signal to Bus* fügt ein Signal einem Bus hinzu. *Resolve Bus* löst einen Bus in seine einzelnen Signale auf. *Extract Signal out of Bus* überführt ein Signal eines Busses in ein elementares Signal. *Move Down Signal in Bus* und *Move Up Signal in Bus* erlauben eine Verschiebung eines Signals entlang einer Bushierarchie.

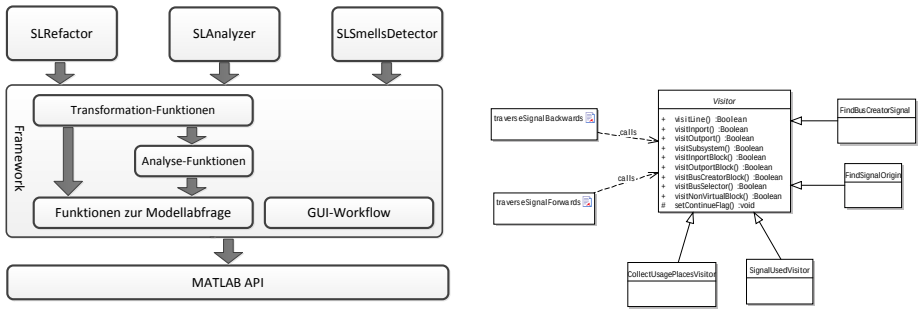
Signalführung: Für die Beschreibung von Algorithmen in Simulink müssen Signalinformationen über Subsysteme hinweg verteilt werden. Aus diesem Grund ist die Führung eines Signals über die Subsystemhierarchie hinweg eine sehr häufig durchgeführte Aufgabe bei der Modellerstellung. Die zu dieser Kategorie gehörenden Refactoring-Operationen dienen dazu, neue Signale über Subsystemgrenzen hinweg hinzuzufügen bzw. bestehende Signalpfade zu entfernen. *Add Cross Hierarchical Signal* erzeugt eine Signalverbindung zwischen zwei Ports über die Subsystemhierarchie hinweg, in dem Inport-/Outport-Blöcke und Linien auf dem Pfad zwischen den Ports hinzufügt werden. *Add Deep Bus Signal* führt ein Signal zu einem Ziel. Dabei wird das Signal durch Busse weitergeleitet. *Replace Goto/From with Explicit Signal* ersetzt korrespondierende *Goto/From*-Blöcke durch eine explizite Signalverbindung, die gegebenenfalls über die Subsystemgrenzen hinweg verlaufen kann. Hingegen ersetzt die inverse Operation *Replace Explicit Signal with Goto/From* eine Signalverbindung durch *Goto/From*-Blöcke. Bei *Remove Signal Backwards* wird ein Signalpfad bis zur Signalquelle oder bis zu einer vorgegebenen Hierarchieebene entfernt. Hingegen entfernt *Remove Signal Forwards* in Vorwärtsrichtung alle Signalpfade, die aus einem Signal hervorgehen.

Schnittstellen: Die Eingabe- und Ausgabeschnittstelle eines Subsystems wird durch die Eingangs- und Ausgangsports des Subsystems definiert. Bei der Erstellung und Bearbeitung eines Modells spielt die Definition der Subsystemschnittstellen zur Verdeutlichung des Informationsflusses eine wichtige Rolle. Die Refactoring-Operationen dieser Kategorie unterstützen den Entwickler daher bei Veränderung der Schnittstellen. *Insert Subsystem Inport/Outport* fügt einen neuen Eingangs- bzw. Ausgangsport an einer beliebigen Position zu einem Subsystem hinzu. *Remove Subsystem Inport/Outport* entfernt einen Eingangs- bzw. Ausgangsport an einem bestimmten Index. Mit Hilfe von *Merge Ports* können Signa-

le mehrerer Ports zusammengefasst und mit *Split Port* auf mehrere Ports verteilt werden. *Create Effective Input/Output Interface* erzeugt für ein Subsystem zusätzlich ein sogenanntes INMAP-bzw. OUTMAP-Subsystem. Für eine detailiertere Darstellung des Modellierungsmusters mit INMAP- und OUTMAP-Subsystemen sei der Leser auf [Ra02] verwiesen.

5 Werkzeugkonzept und Implementierung

Im Rahmen dieser Arbeit ist ein Refactoring-Framework für Simulink entwickelt worden, welches im Folgenden *SLRefactor-Framework* genannt wird. Das Framework wurde mithilfe der objektorientierten Sprache M-Script realisiert. Das SLRefactor-Framework definiert einen GUI-basierten Workflow in abstrakter Weise mit Hilfe des *Template Method*-Entwurfsmusters [Ga94]. Das Refactoring-Werkzeug *SLRefactor* benutzt das Framework und implementiert die Refactoring-Operationen des Katalogs.



(a) Architektur der Umsetzung (b) Besucher-Entwurfsmuster

Abb. 4: Umsetzung

Die in Abb. 4(a) veranschaulichte Schichtenarchitektur des SLRefactor-Frameworks ist durch das Eclipse-Refactoring-Framework [Wi06] inspiriert und besteht aus den folgenden Bausteinen:

Funktionen zur Modellabfrage von Modellen Das Framework stellt eine Sammlung höherwertiger Utility-Funktionen zum Abfragen von Modellen bereit.

Infrastruktur zur Analyse von Simulink-Modellen: Die meisten Refactoring-Operationen erfordern eine Analyse des Modells. Beispielsweise muss bei der Refactoring-Operation *Add Cross Hierarchical Signal* zum Erstellen eines ebenen-übergreifenden Signalflusses zwischen zwei Ports unterschiedlicher Ebenen berechnet werden, wie das Signal über die Hierarchieebenen hinweg geführt werden muss. Hierfür muss das *letzte gemeinsame Subsystem* beider Ports ermittelt werden.

Zusätzlich stellt die Infrastruktur Funktionen zur Verfolgung von Signalen bereit. Zur Signalverfolgung gehören Vorwärts- und Rückwärtsverfolgung eines Signals. Vorwärtsverfolgung wird genutzt, um die Blöcke zu ermitteln, an denen ein Signal verwendet wird. Die Analyse der Signalnutzung stellt eine wesentliche Voraussetzung für die Operationen

Remove Signal Forwards und *Create Effective Input Interface* dar. Analog dazu dient die Rückwärtsverfolgung dazu, die Quelle eines Signals zu finden. Die Refactoring-Operation *Remove Signal Backwards* benutzt diese Analyse, um zu ermitteln, bis zu welchem Block ein Signal ohne Auswirkungen auf andere Modellteile entfernt werden kann. Des Weiteren lässt sich für ein in einem Bus geführtes Signal durch Rückwärtsverfolgung herausfinden, an welchem *BusCreator*-Block der Bus erzeugt wird. Diese Analyse ist unter anderem relevant, um die Refactoring-Operationen *Move Down/Up Signal in Bus* und *Swap Signals in Bus* umzusetzen.

In der Implementierung wird eine Traversierungsstrategie (Vorwärts- und Rückwärtsverfolgung) und eine konkrete Logik durch eine Variante des Besucher-Entwurfsmusters [Ga94] getrennt (siehe Abb. 4 (b)). Außerdem sind die Funktionen zur Vorwärts- und Rückwärtsverfolgung in der Lage, ein Signal durch Busse und über Subsystemgrenzen hinweg zu verfolgen.

Infrastruktur zur Transformation von Simulink-Modellen: Neben den Analyse-Funktionen stellt das SLRefactor-Framework mehrere Transformationsschritte bereit. Transformationsschritte sind Funktionen, die atomare Änderungen an Simulink-Modellen durchführen. Es wird zwischen elementaren und zusammengesetzten Transformationsschritten unterschieden. Ein elementarer Schritt modifiziert ein Simulink-Modell, ohne andere Schritte zu benutzen. Beispiele für elementare Schritte sind *addLine* zum Hinzufügen einer Linie zwischen Ports derselben Hierarchieebene und *addInport/addOutport* zum Hinzufügen eines *Inport-/Outport*-Blocks zu einem Subsystem.

Ein zusammengesetzter Schritt wird durch eine Abfolge elementarer oder zusammengesetzter Schritte definiert. Beispielsweise ist der Schritt *addCrossHierarchical* zum Hinzufügen eines ebenenübergreifenden Signals zwischen zwei Ports unterschiedlicher Hierarchieebenen ein aus aus *addLine*, *addInport* und *addOutport* bestehender zusammengesetzter Schritt.

SLRefactor-Werkzeug: Das *SLRefactor*-Werkzeug benutzt die Infrastruktur des Frameworks und implementiert die meisten Refactoring-Operationen aus dem Katalog in Abb. 3. Das Werkzeug wurde in den Simulink-Editor integriert. Die Refactoring-Operationen können wie in Abb. 5 gezeigt während der Modellierung über ein Kontext-Menü aufgerufen werden. Zur Verbesserung des Modell-Layouts nach einer Refactoring-Operation wird ein automatischer Layouting-Algorithmus für Simulink-Modelle [Kl12] verwendet.

SLAnalyzer und SLSmellsDetector: Neben *SLRefactor* sind im Rahmen dieser Arbeit zwei weitere Werkzeuge *SLAnalyzer* und *SLSmellsDetector* umgesetzt. *SLAnalyzer* bietet erweiterte Operationen zur Modellanalyse, während *SLSmellsDetector* strukturelle Schwachstellen (sogenannte „Model-Smells“) in einem Modell durchsucht. Allerdings werden diese Werkzeuge in dieser Veröffentlichung nicht mehr weiter vertieft.

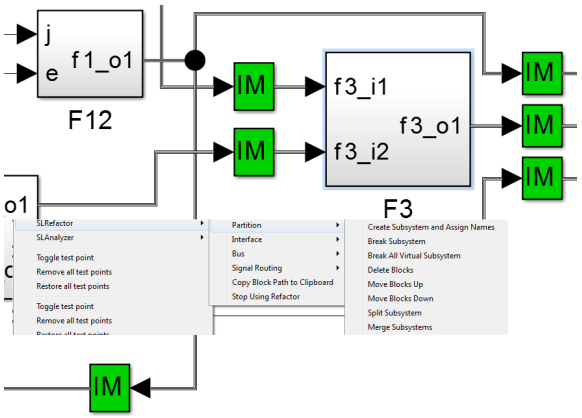


Abb. 5: Integration von SLRefactor im Simulink-Editor

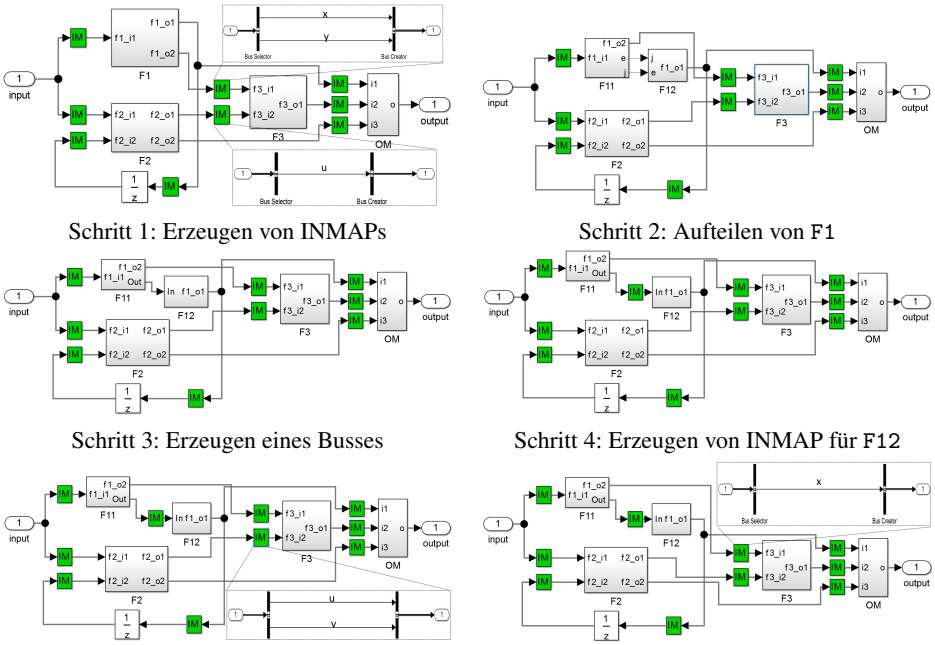
6 Beispiel: Umbau des Modells mit Hilfe von SLRefactor

In diesem Abschnitt wird durch ein Beispiel gezeigt, wie der Entwickler SLRefactor benutzen kann, um die Struktur des Modells sukzessiv effizient zu verbessern und zu erweitern. Das Hauptmerkmal des hier verwendeten Modellierungsmusters ist die Nutzung von INMAP-Subsystemen zur Darstellung von effektiven Schnittstellen der Module. Die im Folgenden beschriebenen sechs Modifikationsschritte werden zudem durch die Darstellungen in Abb. 6 unterstützt.

Schritt 1: Erzeugen INMAPs für die Module Zur Verdeutlichung der Signalflüsse werden im ersten Schritt für jedes Modul sogenannte *INMAP-Subsysteme* erzeugt, die alle im jeweiligen Modul verwendeten Signale explizit auswählen. Hierfür wird die Refactoring-Operation *Create Effective Input Interface* benutzt. Durch die Verwendung von INMAP-Subsystemen wird die sogenannte *effektive Schnittstelle* der Module sichtbar. Das bedeutet, dass die im Modul verwendeten Signale durch INMAP-Subsysteme explizit spezifiziert werden. Somit ist beispielsweise erkennbar, dass das Modul `F3` das Signal `u` von Modul `F2` verwendet. In einem realen Modell können hunderte Signale in einem INMAP-Subsystem selektiert und zu einem Bus zusammengefasst werden.

Schritt 2: Aufteilen von `F1` in zwei kleinere Module Das Modul `F1` implementiert zwei Funktionalitäten, die nach Einschätzung eines Entwicklers in jeweils eigene Module gehören. Mit Hilfe der Refactoring-Operation *Split Subsystem* kann `F1` in zwei kleinere Module `F11` und `F12` aufgeteilt werden. Dabei werden die Signalverbindungen der betroffenen Blöcke beibehalten. Mit Hilfe der Refactoring-Operation *Split Subsystem* kann `F1` in zwei kleinere Module `F11` und `F12` aufgeteilt werden. Dabei werden die Signalverbindungen der betroffenen Blöcke beibehalten.

Schritt 3: Erzeugen eines Busses Nach der Aufteilung von `F1` kommunizieren `F11` und `F12` über elementare Signale. Gemäß Modellierungsmuster sollen Module auf oberste Ebene allerdings über Busse kommunizieren. Aus diesem Grund werden die elementaren



Schritt 1: Erzeugen von INMAPs

Schritt 2: Aufteilen von `F1`

Schritt 3: Erzeugen eines Busses

Schritt 4: Erzeugen von INMAP für `F12`

Schritt 5: Hinzufügen eines Signalpfades

Schritt 6: Entfernen eines Signalpfades

Abb. 6: Schrittweiser Umbau des Modells

Signale zwischen `F11` und `F12` durch die Refactoring-Operation *Create Bus* zu einem Bus zusammengefasst. Die dabei entstandenen *BusCreator*- und *BusSelector*-Blöcke werden mittels der Refactoring-Operation *Move Blocks Down* in `F11` bzw. `F12` verschoben.

Schritt 4: Erzeugen von INMAP für F12 Das Modul `F12` hat einen Eingangsbus, aber kein INMAP-Subsystem, in dem die effektive Schnittstelle beschrieben wird. Mit Hilfe von *Create Effective Input Interface* wird ein INMAP-Subsystem für `F12` erstellt.

Schritt 5: Hinzufügen eines Signals Nun wird festgestellt, dass ein Submodul innerhalb von `F3` das Signal `v` von `F2` benötigt. Die Refactoring-Operation *Add Deep Bus Signal* führt das Signal `v` von `F2` zu dem Zielmodul. Dabei wird das Signal durch die INMAP-Subsysteme geleitet. In der Abbildung lässt sich erkennen, dass das Signal `v` in einem INMAP-Subsystem von `F3` selektiert und zu dem Bus hinzugefügt wird.

Schritt 6: Entfernen eines Signals Der Entwickler stellt fest, dass das Signal `y` von `F1` in `F3` nicht mehr benutzt wird. Mit Hilfe der Refactoring-Operation *Remove Deep Signal Backwards* kann `y` entfernt werden. Dabei hat der Entwickler die Möglichkeit zu entscheiden, bis zu welcher Stelle `y` entfernt wird. In der Abbildung wird gezeigt, dass das Signal `y` in einem INMAP-Subsystem von `F3` entfernt wird.

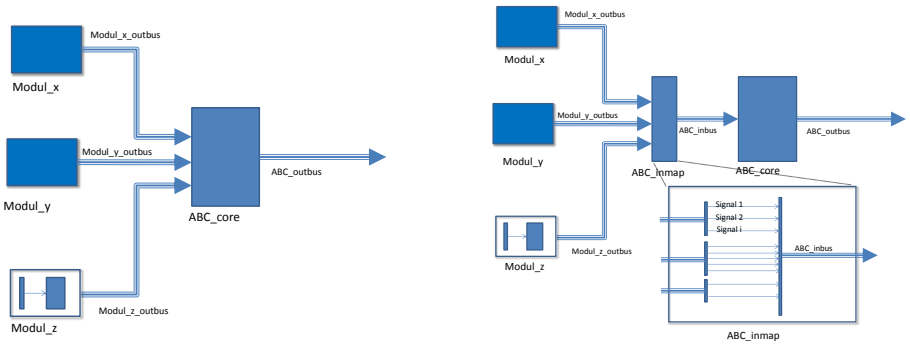
7 Verwandte Arbeiten

Eine Arbeit, die sich ausgiebig mit Transformation von Simulink-Modellen beschäftigt, ist das MATE-Projekt [Le10]. In diesem Projekt wurde untersucht, ob Graphtransformationstechniken für visuelle Spezifikation und Transformation von Simulink- und Stateflow-Modellen geeignet sind. In Vergleich zu der vorliegenden Arbeit liegt der Fokus des MATE-Projekts jedoch nicht auf der systematischen Erfassung von Refactoring-Operationen. Außerdem hat sich gezeigt, dass eine rein visuelle und auf Graphtransformationen basierende Spezifikationssprache für komplexe Spezifikationsszenarien nicht ausreichend ist. Beispielsweise ist die Spezifikation regulärer Ausdrücke, komplexer mathematischer Berechnungen sowie komplexer Navigation durch eine Netzliste verbundener Objekte nicht oder nur schwer möglich.

T. Gerlitz et al. [GSK13] entwickelen ein Duplikatserkennungsverfahren auf Basis von Layoutinformationen. Darüber hinaus stellen sie ein Rahmenwerk vor, welches dieses Verfahren und verschiedene andere Duplikatserkennungsverfahren kombiniert, um höhere Präzision bei der Duplikatserkennung zu erhöhen. Die gefundenen Duplikaten können mithilfe einer Refactoring-Operation in einen gemeinsamen generischen Bibliotheksblock extrahiert werden. Die Duplikaten werden durch einen entsprechend parametrisierten Bibliotheksblock ersetzt werden.

8 Feldstudie

Der hier vorgestellte Ansatz wurde über ca. zwei Jahre in verschiedenen Bereichen der Vor- und Serienentwicklung bei der Daimler AG erprobt. In regelmäßigen Gesprächen berichten die Entwickler, dass sie durch den Einsatz von SLRefactor Simulink-Modelle effizienter bearbeiten können. Im Folgenden wird anhand eines realen Anwendungsfalls die Effizienzsteigerung durch die Anwendung des Ansatzes demonstriert.



(a) Ausgangs-Modularchitektur (b) Modularchitektur nach Refactoring

Abb. 7: Modularchitektur des Modells

In Abb. 7a wird ein Ausschnitt der Modularchitektur des Ausgangsmodells des Fallbeispiels veranschaulicht. Der Entwickler erhält das Modell von einer Nachbarabteilung und möchte das Modell überarbeiten und erweitern. Hier werden Blöcke durch Module in Form

von Subsystemen funktional voneinander getrennt. Diese Module kommunizieren über Busse miteinander. In diesem Modell führen drei Busse in das Modul `ABC_core` hinein, die jeweils Signale der Module `Modul_x`, `Modul_y` und `Modul_z` übertragen.

Bei der Durchsicht des Modells stellt der Entwickler aber fest, dass das Modell strukturell nicht optimal ist. Insbesondere ist nicht sofort erkennbar, welche Signale der Busse tatsächlich im Modul `ABC_core` verwendet werden. Zur Verbesserung der Modellstruktur entscheidet der Entwickler, zunächst die potentiellen Schnittstellen auf *effektive Schnittstellen* mit Hilfe der Refactoring-Operation *Create Effective Input Interface* zu reduzieren, die nur die tatsächlich verwendeten Signale darstellen. Abb. 7b zeigt die Modularchitektur nach der Anwendung von *Create Effective Input Interface*. Außerdem möchte der Entwickler für jedes Modul ein einziges INMAP-Subsystem erzeugen. Damit werden die Eingangssignale des Moduls `ABC_core` so zusammengefasst, dass das Modul genau einen Eingangsbus mit allen in dem Modul verwendeten Signalen erhält.

In dem Fallspiel gibt es insgesamt 9 Module, die wie `ABC_core` Signale anderer Module über eingehende Busse verwenden. Tabelle 1a zeigt sowohl die Anzahl der Blöcke, als auch die Hierarchietiefe dieser 9 Module. Um das Modellierungsmuster mit INMAP

Modul	Σ Blöcke	Hierarchietiefe
1	1349	5
2	1046	4
3	697	4
4	233	3
5	1304	4
6	29	1
7	243	2
8	297	2
9	255	3

(a) Umfang der Module

Modul	Σ verwendete Signale
1	156
2	179
3	157
4	130
5	50
6	76
7	11
8	130
9	60
	949

(b) Umfang der INMAPs

Tab. 1: Statistische Daten

umzusetzen, müssen 9 INMAP-Subsysteme manuell erzeugt werden. Dabei müssen insgesamt fast 1000 Signale per Hand identifiziert werden, die in den jeweiligen Modulen verwendet werden (Tabelle 1b). Nach einer Expertenabschätzung benötigt ein Entwickler zur Durchführung der Änderung mehrere Stunden. Zudem ist die manuelle Änderung fehleranfällig. SLRefactor ermittelt die von den Modulen benötigten Signale automatisch und führt die Änderungen innerhalb weniger Minuten fehlerfrei durch. Danach kann der Entwickler durch die von SLRefactor bereitgestellten Refactoring-Operationen wie *Move Blocks Up/Down Split Subsystem*, *Merge Subsystems*, *Add Deep Signal*, *Remove Signal Backwards/Forwards* usw. die Struktur des Modells effizient verändern und erweitern.

9 Zusammenfassung und Ausblick

Im Rahmen der vorliegenden Arbeit ist erstmalig ein Refactoring-Konzept für Simulink-Modelle entwickelt worden. Das Konzept ist an bekannten Refactoring-Techniken für textuelle Programmiersprachen angelehnt. Hierzu wurde ein Katalog von Refactoring-Operationen erarbeitet. Das Konzept wurde in Form eines Prototyps umgesetzt, welcher in den Simulink-Editor integriert ist und die meisten Refactoring-Operationen aus dem Katalog bereitstellt. Durch eine Erprobung bei der Daimler AG konnte die Relevanz und Praxistauglichkeit des Ansatzes gezeigt werden. Der Ansatz beschränkt sich derzeit auf Simulink-Modelle. Grundsätzlich ist aber davon auszugehen, dass die Ideen auch auf andere vergleichbare datenflussorientierte grafische Modellierungssprachen übertragbar sind.

Literaturverzeichnis

[DRW12] Dziobek, Christian; Ringler, Thomas; Wohlgemuth, Florian: Herausforderungen bei der modellbasierten Entwicklung verteilter Fahrzeugfunktionen in einer verteilten Entwicklungsorganisation. In: Proceedings Dagstuhl-Workshop MBEES Modellbasierte Entwicklung eingebetteter Systeme. S. 1–10, 2012.

[Fo99] Fowler, Martin: Refactoring: Improving the Design of Existing Code. Addison-Wesley, 1999.

[Ga94] Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John: Design patterns: elements of reusable object-oriented software. Pearson Education, 1994.

[GSK13] Gerlitz, Thomas; Schake, Stefan; Kowalewski, Stefan: Ansatz zur Erstellung und Wartung von Simulink-Modellen durch den Einsatz von Transformationen/Refactorings und Generierungsoperationen. In: Proceedings Dagstuhl-Workshop MBEES Modellbasierte Entwicklung eingebetteter Systeme. S. 1–10, 2013.

[Kl12] Klauske, Lars K.: Effizientes Bearbeiten von Simulink Modellen mit Hilfe eines spezifisch angepassten Layoutalgorithmus. Dissertation, Technical Universität Berlin, 2012.

[Le10] Legros, Elodie; Schäfer, Wilhelm; Schürr, Andy; Stürmer, Ingo: MATE: A Model Analysis and Transformation Environment for MATLAB Simulink. In: Proceedings of the 2007 International Dagstuhl Conference on Model-based Engineering of Embedded Real-time Systems. MBEERTS'07. Springer-Verlag, S. 323–328, 2010.

[Ma14] Mathworks, The: , Matlab/Simulink. URL `http://www.mathworks.com/toolbox/simulink/`, 2014.

[QD13] Quang, Tran M.; Dziobek, Christian: Ansatz zur Erstellung und Wartung von Simulink-Modellen durch den Einsatz von Transformationen/Refactorings und Generierungsoperationen. In: Proceedings Dagstuhl-Workshop MBEES Modellbasierte Entwicklung eingebetteter Systeme. S. 1–10, 2013.

[Ra02] Rau, Andreas: On model-based development: A pattern for strong interfaces in SIMULINK. Gesellschaft für Informatik, FG, 2(1):12, 2002.

[TWD13] Tran, Quang M.; Wilmes, Benjamin; Dziobek, Christian: Refactoring of Simulink Diagrams via Composition of Transformation Steps. In: Proceedings of the 8th International Conference on Software Engineering Advances (ICSEA). 2013.

[Wi06] Widmer, Tobias: Unleashing the Power of Refactoring. In: Eclipse Magazine. Software & Support Verlag, 2006.