



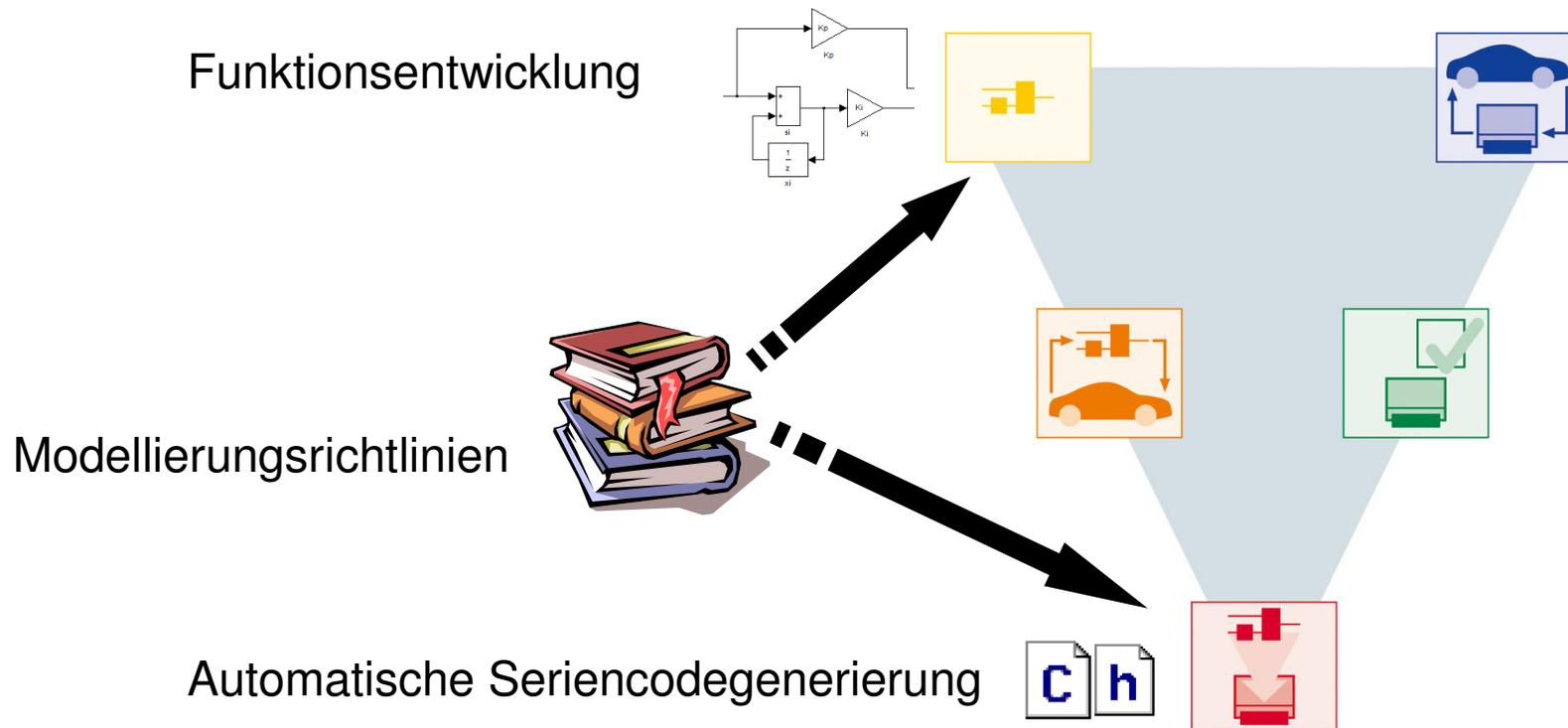
Maßnahmen zur Qualitätssicherung beim Einsatz von automatischer Code-Generierung

Ulrich Eisemann, Michael Beine, Christian Wewetzer, dSPACE GmbH

Übersicht

- Anwendung von Modellierungsrichtlinien für eine Simulink/TargetLink-Toolkette
- MISRA Konformität auf Code/Modellebene
- Autocode Validierungssuite
- Automatische Detektierung von Laufzeitfehlern im modell-basierten Entwicklungsprozess

Modellierungsrichtlinien – Motivation und Ziele



Ziel: Einheitlicher Standard für das Aussehen und die Struktur von Modellen sowie die Bedatung zur automatischen Code-Generierung



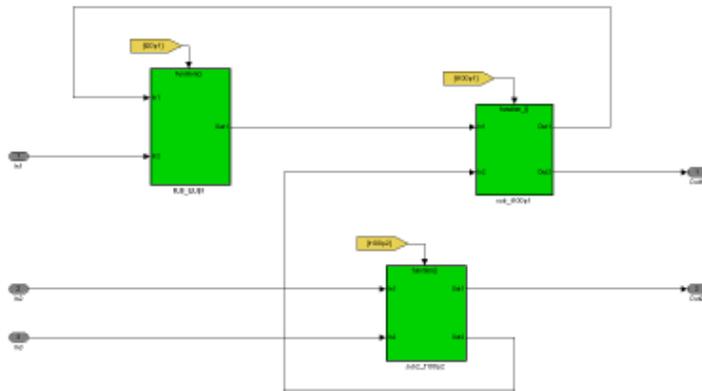
Modellierungsrichtlinien für Simulink/Stateflow/TargetLink Toolkette.

Modellierungsrichtlinien - Stil und Layout Aspekte

Stilfragen für das Aussehen von Modellen sind mindestens genauso wesentlich wie Kodierrichtlinien für Software:

- Layout für Modelle
- Namenskonventionen
- Hinreichende Kommentierung
- Hierarchische Strukturierung

Layouts für Modelle



Codierungs-Richtlinien

```
/* update of inport for controller/Linearization */
Sa2_POSITION = Sa1_POS;

/* call of function: controller/Linearization */
Sa2_Linearization();

/* Sum: controller/e */
Sa1_e = (Int16) (((UInt16) Sa1_REF) - ((UInt16) Sa2_LIN_POS));

/* Sum: controller/si */
Sa1_si = (Int16) (((UInt16) (Int16) (Sa1_e >> 1)) + ((UInt16) X_Sa1_xi));
```

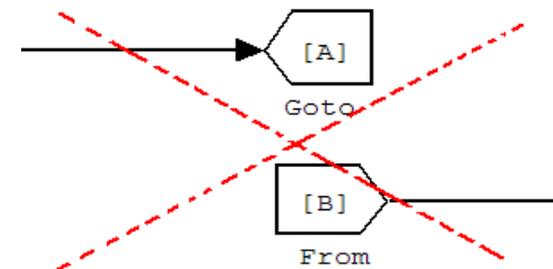
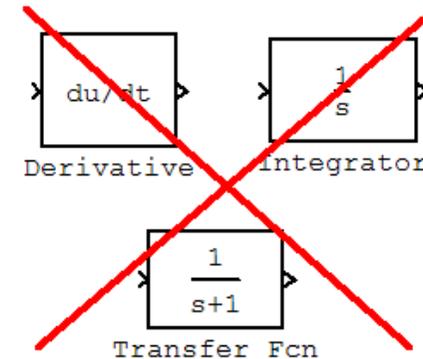
Modellierungsrichtlinien - Wahl eines Subsets

Auswahl eines Subsets der Sprache für die Seriercode-Generierung:

- Auswahl eines speziellen Blocksets
- Vermeidung bestimmter Sprachkonstrukte

Beispiele:

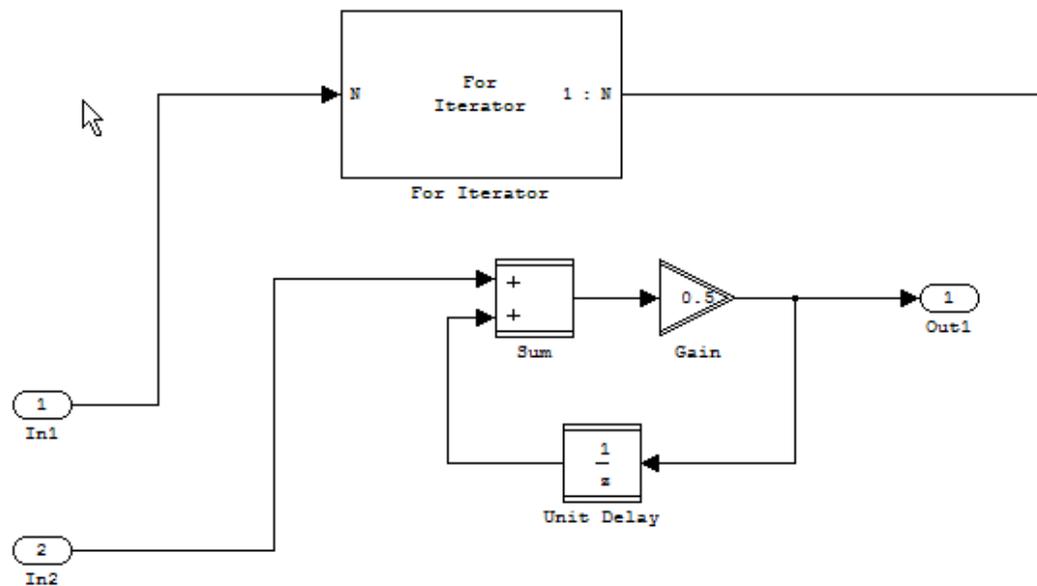
- Keine zeitkontinuierlichen Blöcke für Seriercode-Generierung
- Spärlicher Gebrauch von *From* / *Goto* Blöcken, um einen transparenten Signalfluss zu gewährleisten



Modellierungsrichtlinien – Unsichere Konstrukte

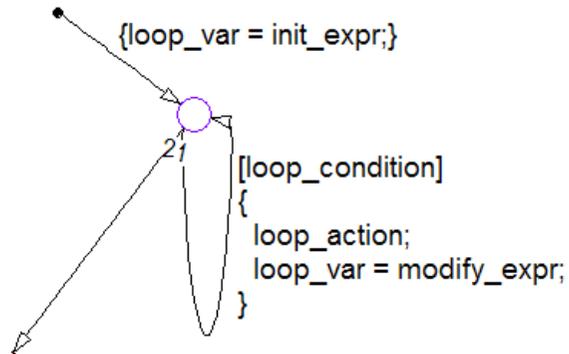
Beispiel:

- Schleifen mit intransparenten Iterationsgrenzen



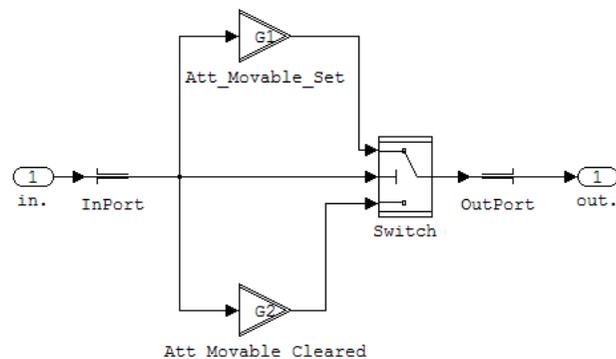
Erschwert beispielsweise die Analyse der Worst Case Execution Time (WCET)

Modellierungsrichtlinien – Effizienter Code



Design Pattern für Kontrollfluss in Stateflow

- Übersetzung in effiziente Code Konstrukte
- Leichtere Modell- wie auch Code-Reviews



Attribute zur Steuerung der Optimierung

- Löschen überflüssiger Variablen
- Verschiebung in bedingt ausgeführte Zweige
- Sicherstellung der Messbarkeit / Kalibrierbarkeit

Production code options	
Description:	estimated EGO
Variable:	<input type="text"/>
Class:	OPT_GLOBAL
Type:	Int16
Scaling:	<input type="text"/>
Name:	<input type="text"/>
Address:	<input type="text"/>
Width:	<input type="text"/>
Element:	<input type="text"/>

Style Checker für Modellierungsrichtlinien

Zur Sicherstellung der Einhaltung von Modellierungsrichtlinien bietet sich der Einsatz von Style Checkern zur automatischen Überprüfung der Regeln an, wo immer möglich.



Simulink und TargetLink bieten eine Programmierschnittstelle, um Zugriff auf alle Modell- und Code-Generierungseigenschaften zu erhalten.

➤ Automatisierung durch Style Checker prinzipiell möglich.

Übersicht

- Anwendung von Modellierungsrichtlinien für eine Simulink/TargetLink-Toolkette
- **MISRA Konformität auf Code/Modellebene**
- Autocode Validierungssuite
- Automatische Detektierung von Laufzeitfehlern im modell-basierten Entwicklungsprozess

MISRA Konformität

MISRA-C:1998 und MISRA-C:2004 definieren ein „sicheres“ Subset der Programmiersprache C.

Definition erfolgte im Hinblick auf:

- Typische Programmierfehler
 - Leichte Lesbarkeit des Codes und erhöhte Transparenz
 - Lücken in der Sprachdefinition von ISO 9899:1990 C
 - Typische Compiler/Linker Probleme
-
- Derzeit werden MISRA Checks auch auf automatisch erzeugten Code angewandt, obwohl die Regeln primär im Hinblick auf Handprogrammierung entwickelt wurden.
 - MISRA C erlaubt Abweichungen von den Regeln, sofern diese wohldurchdacht und dokumentiert sind.



MISRA Konformität – Relevanz für Autocode

MISRA Regeln mit eingeschränkter Relevanz für Codegeneratoren:



MISRA Regel 1: ANSI C Compliance

Portabilität wird durch Blockdiagramme sichergestellt. Target-spezifische Codegenerierung erhöht die Effizienz und Sicherheit.

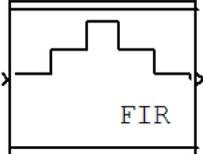
Beispiel: Gesättigte Operationen

```
e = (_sat Int16)((_sat Int16)REF-(_sat Int16)POS);
```

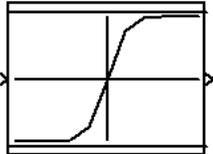


MISRA Regel 101: No pointer arithmetic

```
Int8 Tab1DS4I2T1563_a(const MAP_Tab1DS4I2T1563_a * map, Int8 x)
{
    ....
    z_table = map->z_table;
    ...
    Aux_U8_b = (UInt8) (((UInt8) (((UInt8) x) - ((UInt8) (Aux_U8_a
    z_table += Aux_U8_a;
    Aux_S8 = *((z_table)++);
    ....
}
```



FIR Filter



Look-Up Table

MISRA Konformität – Relevanz für Autocode II

MISRA Regeln mit Relevanz für Code-Generatoren:



MISRA Regel 37: Bitwise operations shall not be performed on signed integer types.

Code-Generator Option:

SeparateSearchFunction	Generate separate search functions for tables. [ON OFF]
SfStateEncoding	Set Stateflow state encoding method
ShareFunctionsBetweenTLSubs	Enable function sharing between TL subsystems. [ON OFF]
ShiftMode	Controls shift mode (1 => don't shift right; 2 => don't shift left; 3 => don't shift right or left)
StateActivityEncodingLimit	Use state activity encoding if the number of states exceeds this limit
StateflowUseBitfields	Use bitfields for state encoding and flags for Stateflow. [ON OFF]
StructCompClassUsesLinkerSection	Derived struct component classes use the linker sections of their template classes. [ON OFF]
TLSimDir	Control creation of simulation frame include file <tlsubsystemname>_fri.h.



MISRA Regel 11: Identifiers (internal and external) shall not rely on significance of more than 31 characters.

Code-Generator Option:

Inlining threshold: 5

<input checked="" type="checkbox"/> Use bitfields in state machines	<input checked="" type="checkbox"/> Code identifiers max. 31 chars
<input type="checkbox"/> Use global bitfields for Booleans	<input type="checkbox"/> Omit initializations to zero in restart functions
<input checked="" type="checkbox"/> Use optimized Boolean types	<input checked="" type="checkbox"/> Share functions between TargetLink subsystems
<input checked="" type="checkbox"/> Functions for 64-bit operations	<input type="checkbox"/> Extended variable sharing
<input type="checkbox"/> Keep saturation statements	

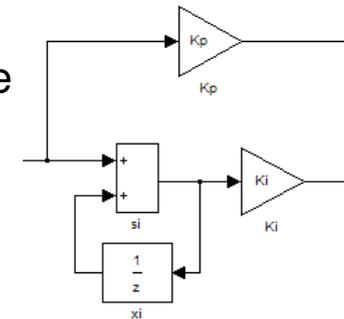
MISRA Konformität – MISRA-Aktivitäten

MISRA Aktivitäten in Bezug auf Code Generatoren:

- *MISRA Autocode Regeln*
Entwicklung eines „sicheren“ C subsets speziell für automatische Codegeneratoren
- Verlagerung der MISRA Regeln auf die Modellebene
- *MISRA Regeln für Simulink/TargetLink*

```
/* update of inport for controller/Linearization */  
Sa2_POSITION = Sa1_POS;  
  
/* call of function: controller/Linearization */  
Sa2_Linearization();  
  
/* Sum: controller/e */  
Sa1_e = (Int16) (((UInt16) Sa1_REF) - ((UInt16) Sa2_LIN_POS));  
  
/* Sum: controller/si */  
Sa1_si = (Int16) (((UInt16) (Int16) (Sa1_e >> 1)) + ((UInt16) X_Sa1_xi));
```

Übergang zur Modellebene



Übersicht

- Anwendung von Modellierungsrichtlinien für eine Simulink/TargetLink-Toolkette
- MISRA Konformität auf Code/Modellebene
- **Autocode Validierungssuite**
- Automatische Detektierung von Laufzeitfehlern im modell-basierten Entwicklungsprozess

Werkzeug Qualifizierung: AVS Arbeitsgruppe

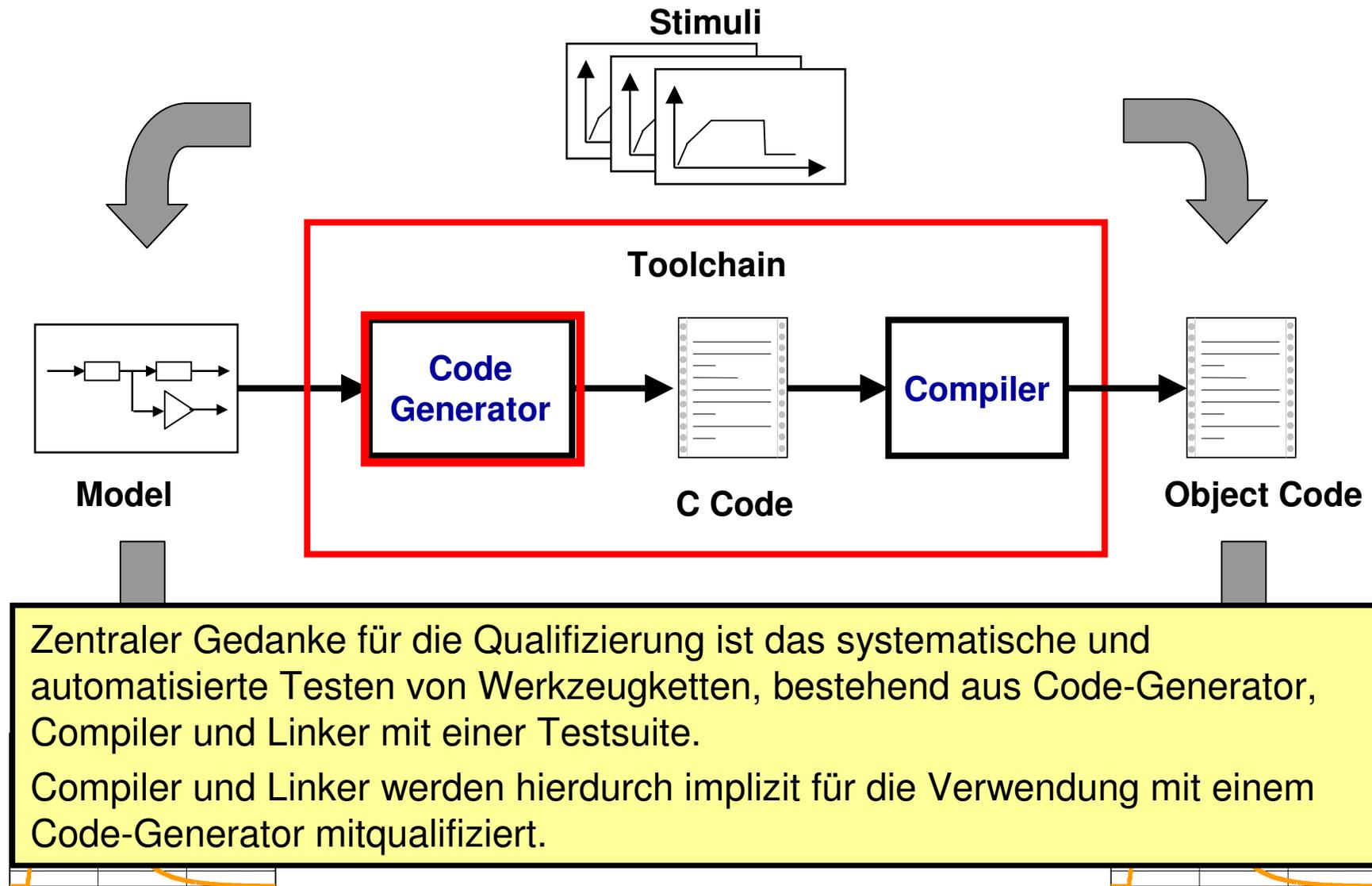
Zielsetzung:

Das Ziel des AVS Arbeitskreises ist die Entwicklung eines einheitlichen Vorgehens zur Qualifizierung von Codegeneratoren für den Einsatz in Automotive-Applikationen. Zentrales Merkmal des Vorgehens zur Qualifizierung ist die systematische, automatisierte Prüfung einer Werkzeugkette bestehend aus Codegenerator, Compiler und Linker mit Hilfe einer Testsuite. Compiler und Linker werden durch diese Vorgehensweise für den Einsatz mit dem betrachteten Codegenerator implizit mit abgesichert.

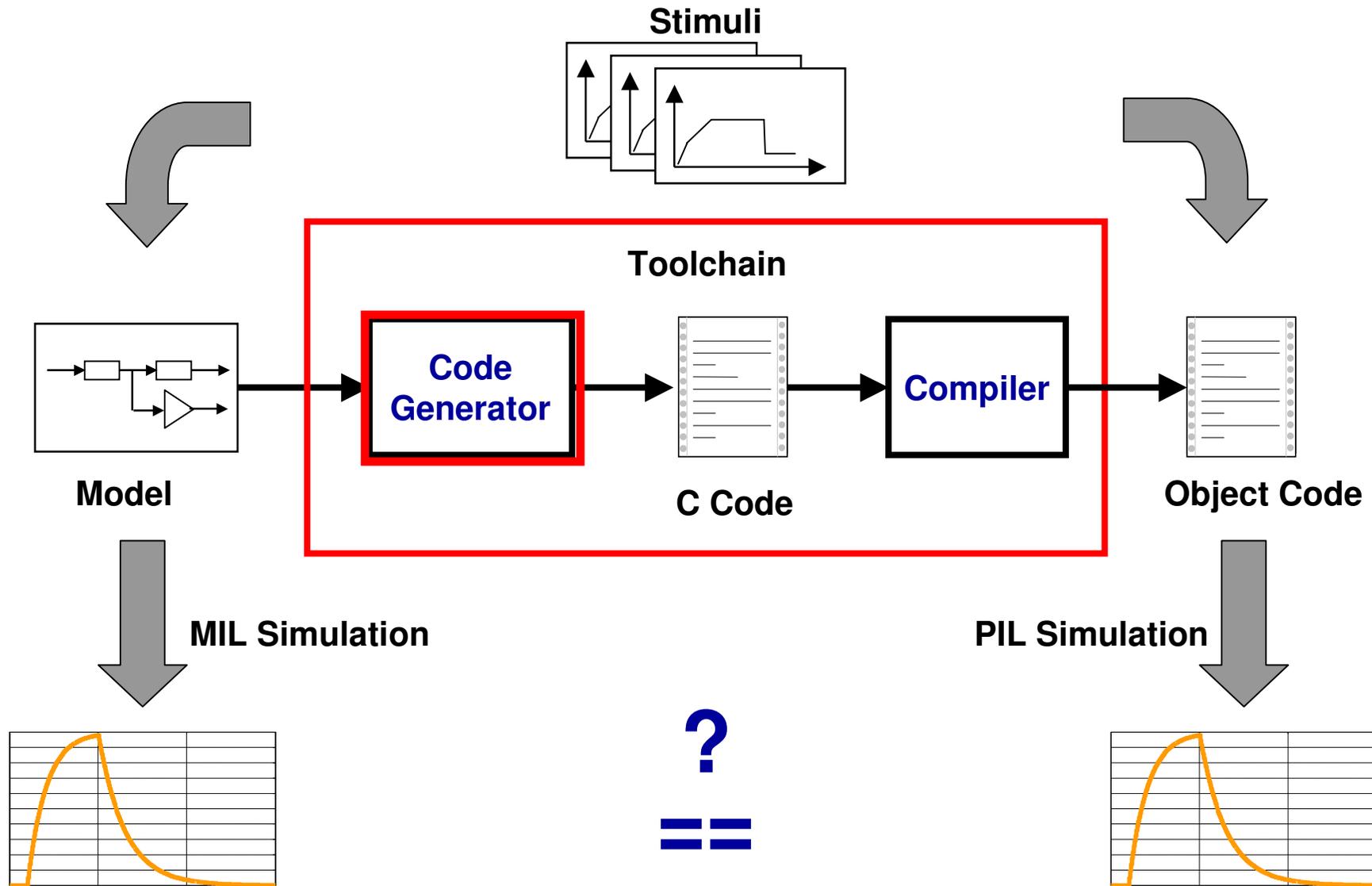
Der Arbeitskreis wurde im Herbst 2004 aus Bedarfsträgern und Werkzeugherstellern gebildet.



Grundlegende AVS Idee



Grundlegende AVS Idee II



Übersicht

- Anwendung von Modellierungsrichtlinien für eine Simulink/TargetLink-Toolkette
- MISRA Konformität auf Code/Modellebene
- Autocode Validierungssuite
- **Automatische Detektierung von Laufzeitfehlern im modell-basierten Entwicklungsprozess**

Potenzielle Laufzeitfehler

Mögliche Arten von Laufzeitfehlern im Code:

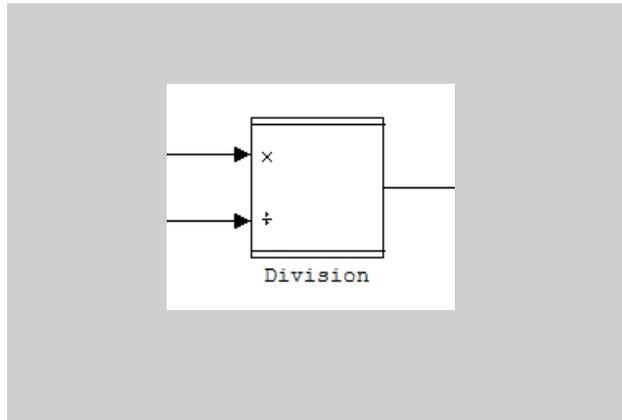
- Lesezugriff auf nicht-initialisierte Daten
- Zugriff auf Arrays außerhalb der Grenzen
- Dereferenzierung von Null-Zeigern
- ...

- undefinierte/unzulässige Berechnung
 - Über- oder Unterlauf
 - Division durch Null
 - Wurzel aus negativen Zahlen
- Nicht-terminierende Schleifen
- Dead Code
- ...

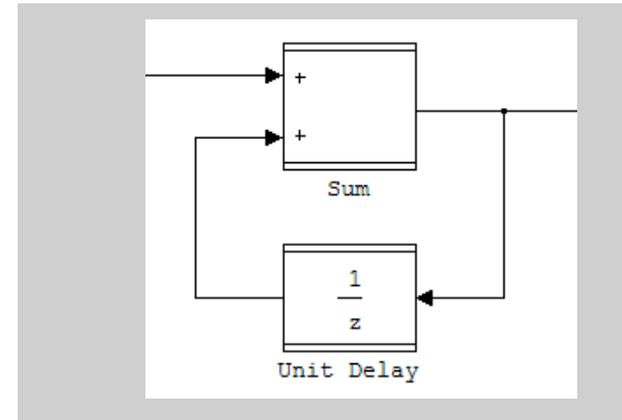
Laufzeitfehler sind auch für modellbasiertes Design relevant.

Laufzeitfehler durch inkorrektes Design

Eventuelle Fehler im Modell-Design:



Potenzielle Division durch Null



Potenzieller Über/Unterlauf

Laufzeitfehler im generierten Code entstehen nicht durch den Codegenerator sondern durch Fehler im Modell Design.

— Laufzeitfehler Analyse durch Abstrakte Interpretation —

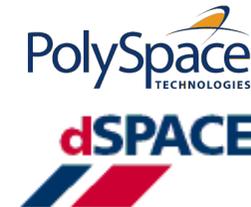
Detektierung von Laufzeitfehlern durch statische Analyse des generierten Codes mittels **Abstrakter Interpretation**.

Vorteile:

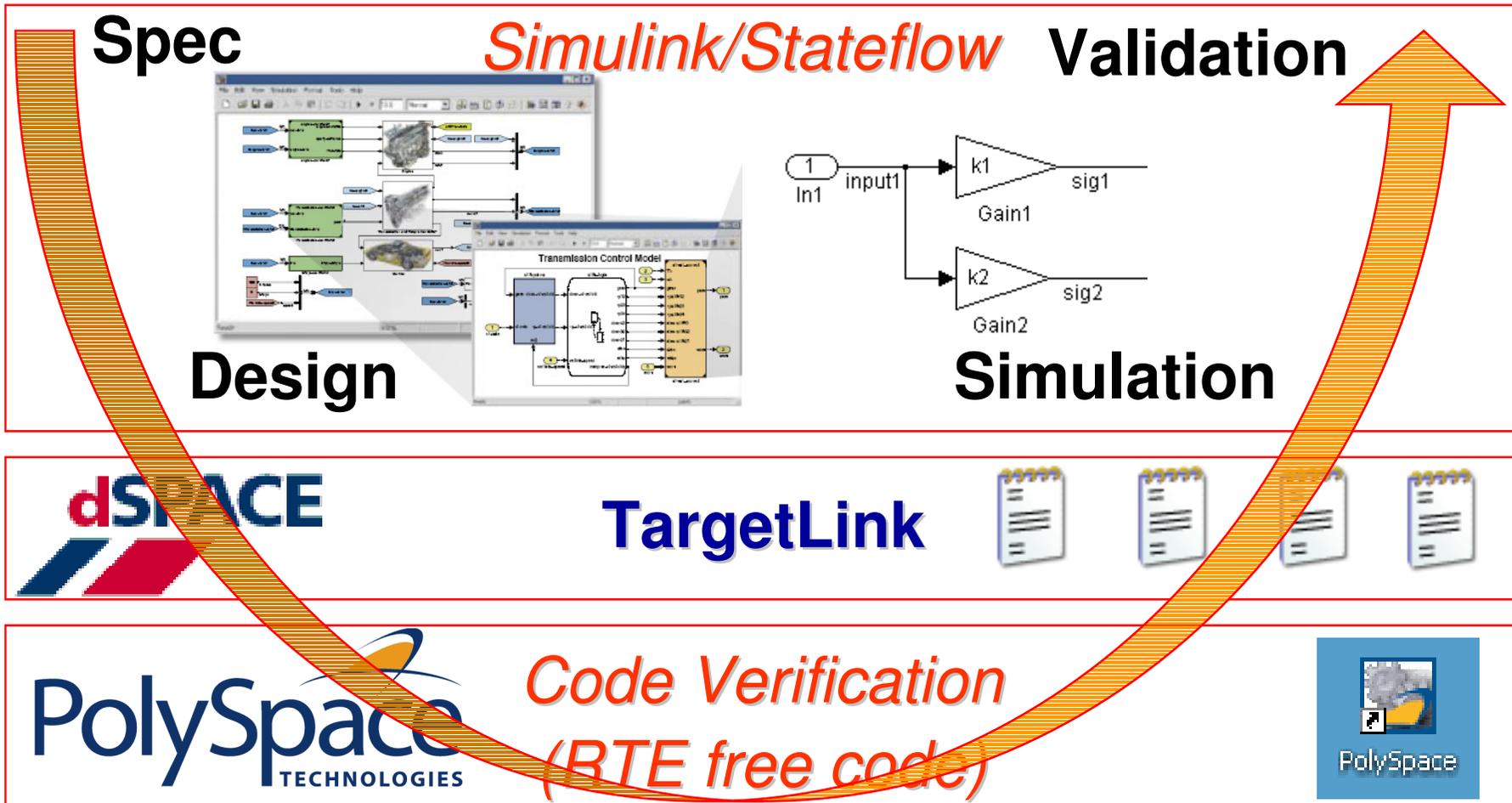
- Keine dynamischen Tests erforderlich
- Mathematischer Beweis für die Freiheit von Laufzeitfehlern
- Berechnung von Wertebereichsgrenzen durch den gesamten Code
- Überwiegend automatische Analyse des generierten Codes

Tools:

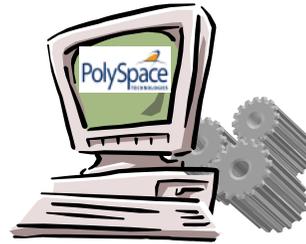
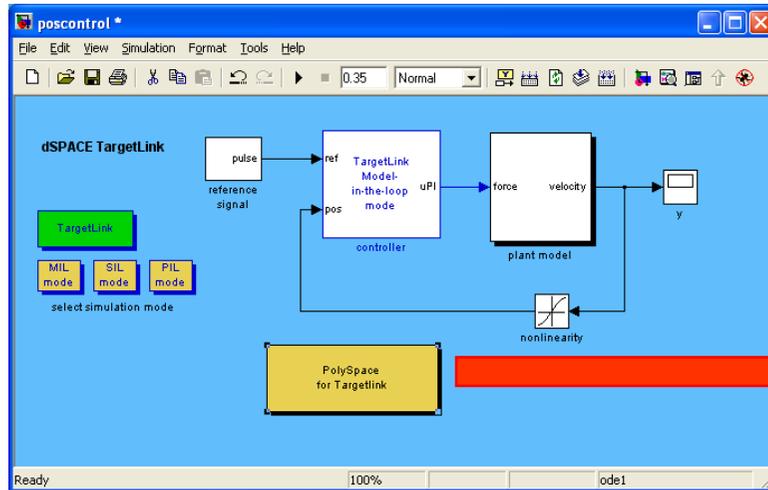
- PolySpace Verifier
- Integration für Code Generator TargetLink



TargetLink und PolySpace im Entwicklungsprozess



TargetLink / PolySpace Integration – Workflow



```
PolySpace Viewer - C:\PolySpace_results_pipt1_polyspace_dspaceRTE_px_02_poly
File Edit Tools Windows Help
N-SHR Alpha Beta Gamma Filter all
DRILLS
PROC
OBRAI ZDV NIU local SCAL DUFL IDP COR POW IRV

picontroller.c
135 /* Sum: picontroller/e */
136 Sa1_e = (Int16) (((Int32) (((UInt32) Sa1_e
137
138 /* Sum: picontroller/x */
139 Sa1_x = (Int16)
140
141 /* Sum: picontroller/y */
142 # combined #
143 # combined #
144 Sa1_U = (Int16)
145 Sa1_U = (Int16) (((Int16) Sa1_x)
146
147 /* Outport: picontroller/u */
148 Sa1_U = Sa1_sU;
149
150 /* Unit delay: picontroller/Unit Delay */
151 X_Sa1_Unit_Delay = (UInt16) Sa1_x;
```

Workflow:

1. Code-Generierung für das Modell
2. Start der Analyse aus dem Modell heraus
3. Inspektion der Klassifizierung der Codezeilen im Viewer
4. Backtracking vom Code zu einzelnen Blöcken

Laufzeitfehler Analyse - Ergebnisdarstellung

Klassifikation der einzelnen Code-Fragmente:

- 100% Laufzeitfehlerfrei
- In 100% der Fälle tritt Laufzeitfehler auf
- Potenzielle Laufzeitfehler
- Dead Code

The screenshot displays the PolySpace Viewer interface. On the left, a block diagram of a control loop is shown. It includes a reference input 'ref' (1) and a feedback input 'y' (2). The error signal 'e' is calculated as $e = \text{ref} - y$. This error signal is multiplied by a gain 'Kp' (5). The resulting signal is summed with a feedback signal 'x' at a summing junction (highlighted with a red circle). The output of this summing junction passes through a unit delay block ($\frac{1}{z}$) to produce the feedback signal 'y'. The output of the summing junction is also the output of the system, 'u'.

In the center, the 'Procedural entities' table shows the following data:

Entity	?	X	?	✓	%	Lin
polyspace	0	0	4	40	91	
__polyspace_main.c				10	100	
picontroller.c			4	30	88	
__polyspace_stdsubs.c					0	

On the right, the 'Variables View' shows the relationship between variables in the code and the diagram. The 'Call Tree View' shows the call hierarchy for the 'picontroller' entity.

At the bottom right, the source code for 'picontroller.c' is displayed. A red box highlights the following code fragment, which is pointed to by a red arrow from the summing junction in the diagram:

```
141 /* Sum: picontroller/sU */
142 # combined # Gain: picontroller/Kp
143 # combined # Gain: picontroller/Ki */
144 Sal_sU = (Int16) (((UInt16) (Int16) (Sal_e
145 (Int16) (((Int32) Sal_x) * ((Int32) P_S:
146
147 /* Output: picontroller/u */
148 Sal_U = Sal_sU;
149
150 /* Unit delay: picontroller/Unit Delay */
151 X_Sal_Unit_Delay = (UInt16) Sal_x;
```

— Laufzeitfehler Analyse – Vorteile auf der Modellebene —

Ziel: Möglichst wenig falsche Alarme („false negative“), also fälschlich als unsicher klassifizierte Codestellen.

Vorteile auf Modellebene:

- Min/Max Werte für Applikationsparameter
- Natürliche Wertebereichsgrenzen für Eingangsvariablen
- Effizientes Arbeiten in gewohnter Umgebung

TargetLink: Subsystem/InPort (Inport) Last modified: 06-Apr-2005 15:04:51

Output | Logging & Autoscaling | Documentation

Intertask communication
Message: default Manage Messages
Component: Receiver:

Production code options
Description:
Variable: Name: \$S_\$B Unit:
Class: default Address:
Type: Int32 Width: 1 Cast output signal to TargetLink type
Scaling: Element: 1 Some scaling for all

	Implemented	Calculated	Simulated
LSB: 2 ⁿ 0	Max: 1000 000		
Offset: 0	Min: -1000 1000	n.a.	n.a.

Writing data to block ... done.

Apply Revert Help Cancel Close

- Vorgabe von Wertebereichen
- Ablage im Data Dictionary nach der Code Generierung
- Analyse nutzt Zusatzinformationen

Zusammenfassung

- Modellierungsrichtlinien für eine Toolkette basierend auf Simulink/TargetLink sichern einheitlichen Standard für das Aussehen und die Struktur von Modellen sowie die Bedienung zur automatischen Code-Generierung
- Bedeutung von Autocode führte zu MISRA Aktivitäten speziell für Code Generatoren und modellbasierte Entwicklung
- Arbeit an einer Autocode-Validierungssuite zur Qualifizierung von automatischen Code-Generatoren
- Automatische Laufzeitfehleranalyse dient zur Absicherung des generierten Codes und lässt sich vorteilhaft mit modellbasierter Entwicklung verbinden

Vielen Dank für Ihre Aufmerksamkeit!



Für weitere Fragen:

Ulrich Eisemann

Produkt Ingenieur TargetLink

Tel.: +49 5251 1638 592

Email: ueisemann@dSPACE.de