Automotive – Safety & Security 2006

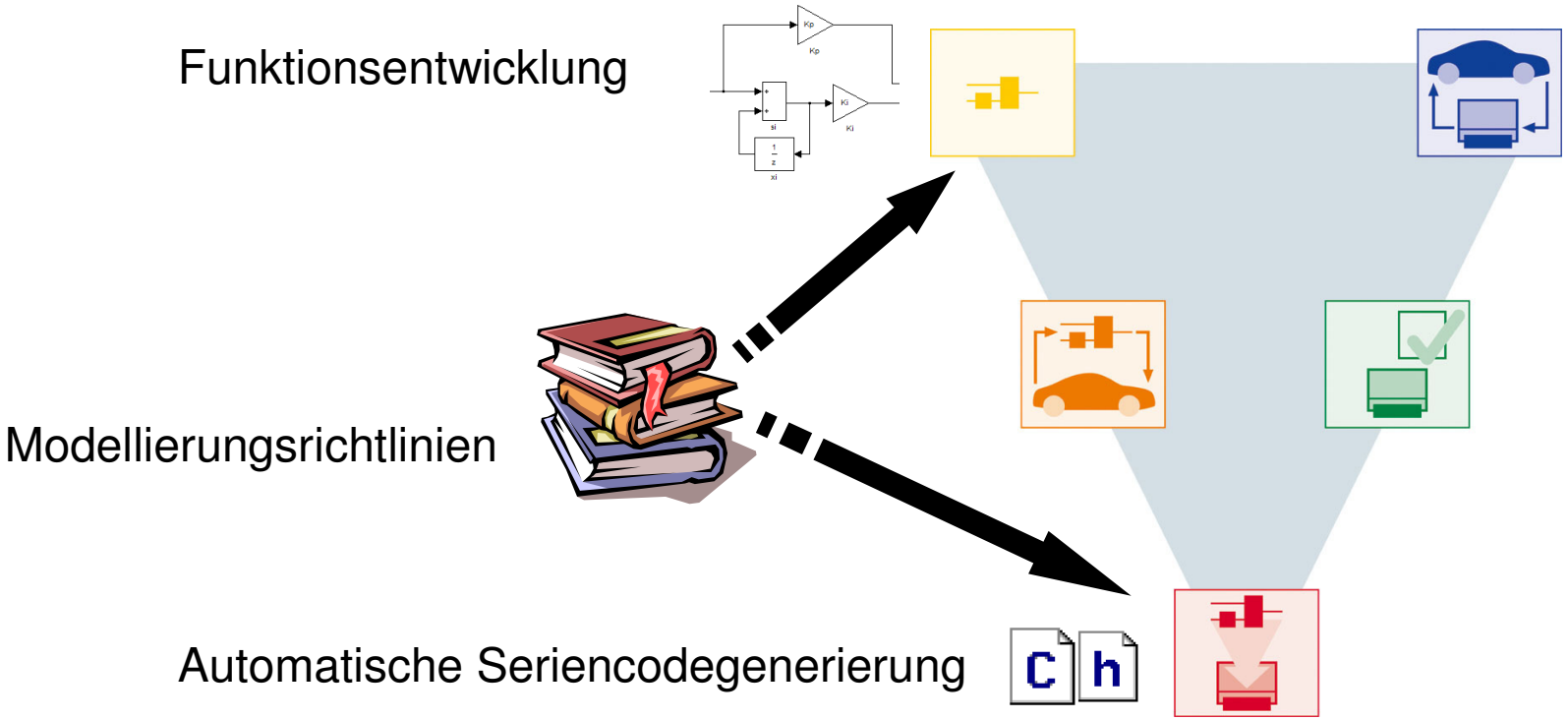
Maßnahmen zur Qualitätssicherung beim Einsatz von automatischer Code-Generierung

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

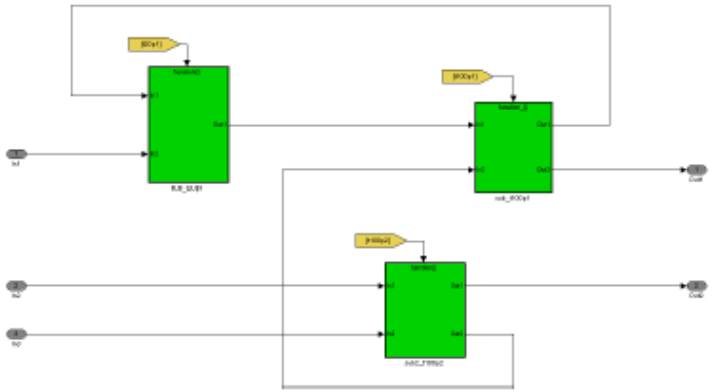
Modellierungsrichtlinien für Simulink/Stateflow/TargetLink Toolkette.

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));
```

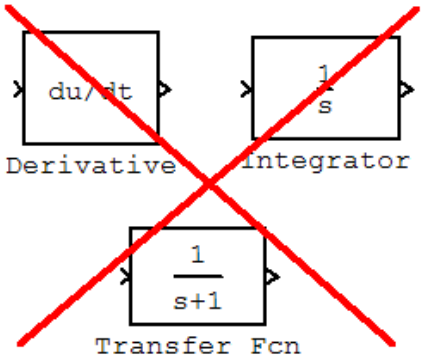
Modellierungsrichtlinien - Wahl eines Subsets

Auswahl eines Subsets der Sprache für die Seriencode-Generierung:

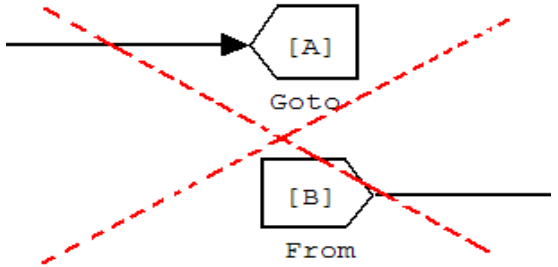
- Auswahl eines speziellen Blocksets
- Vermeidung bestimmter Sprachkonstrukte

Beispiele:

- Keine zeitkontinuierlichen Blöcke für Seriencode-Generierung



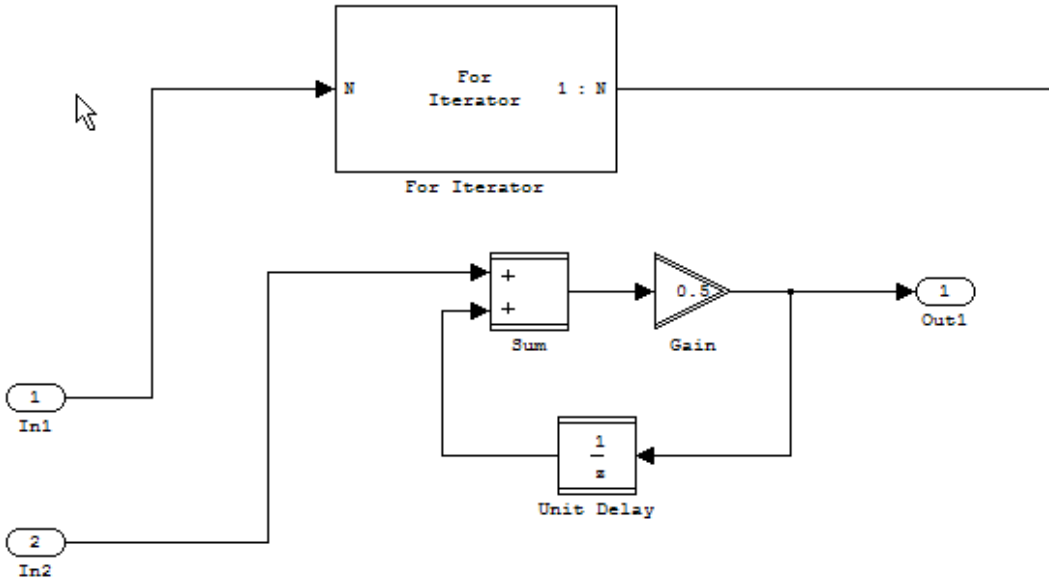
- Spärlicher Gebrauch von *From / Goto* Blöcken, um einen transparenten Signalfluss zu gewährleisten



Modellierungsrichtlinien – Unsichere Konstrukte

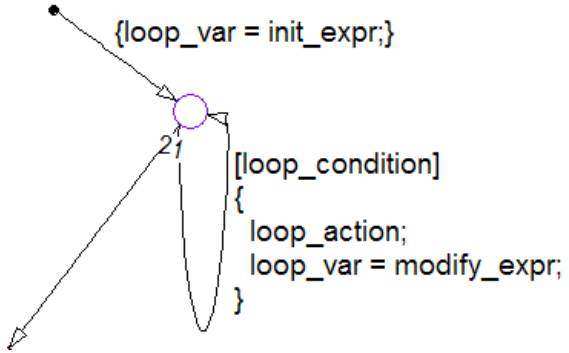
Beispiel:

- Schleifen mit intransparenten Iterationsgrenzen



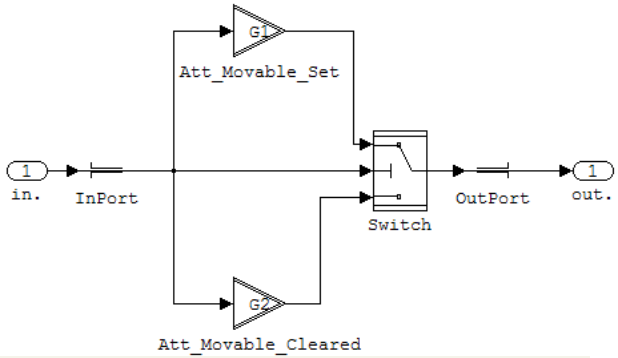
Erschwert beispielsweise die Analyse der Worst Case Execution Time (WCET)

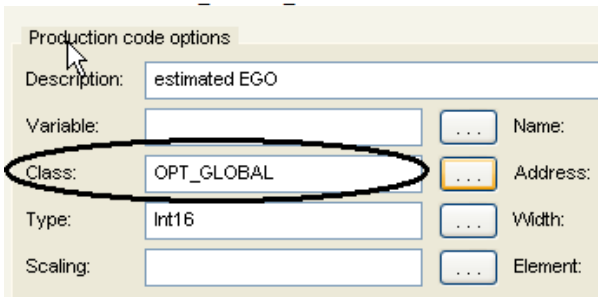
Modellierungsrichtlinien – Effizienter Code



Design Pattern für Kontrollfluss in Stateflow

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



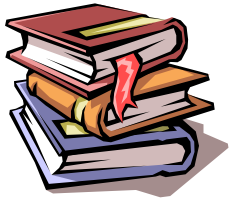


Attribute zur Steuerung der Optimierung

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

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

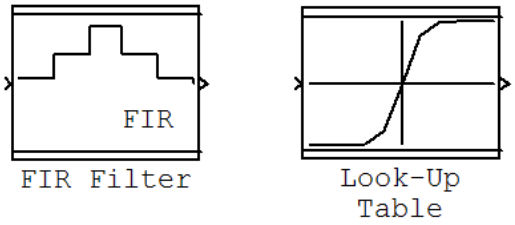
Portabilität wird durch Blockdiagramme sichergestellt. Target-spezifische Codegenerierung erhöht die Effizienz und Sicherheit.

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)++);
   ....
```



MISRA Konformität – Relevanz für Autocode II

MISRA Regeln mit Relevanz für Code-Generatoren:

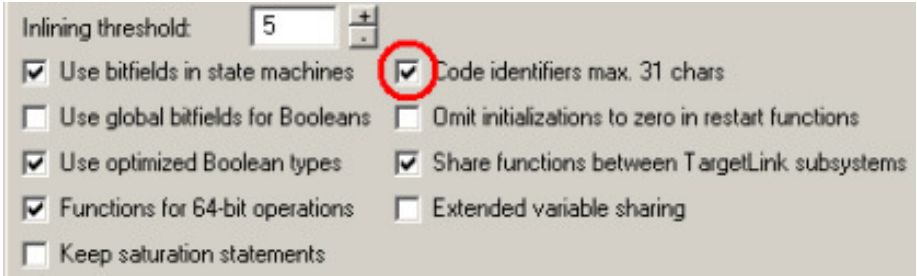
MISRA Regel 37: Bitwise operations shall not be performed on signed integer types.

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:



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*

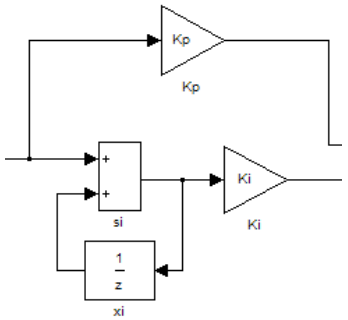
```
/* update of inport for controller/Linearization */
Sa2_POSITION = Sa1_POS;

/* 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

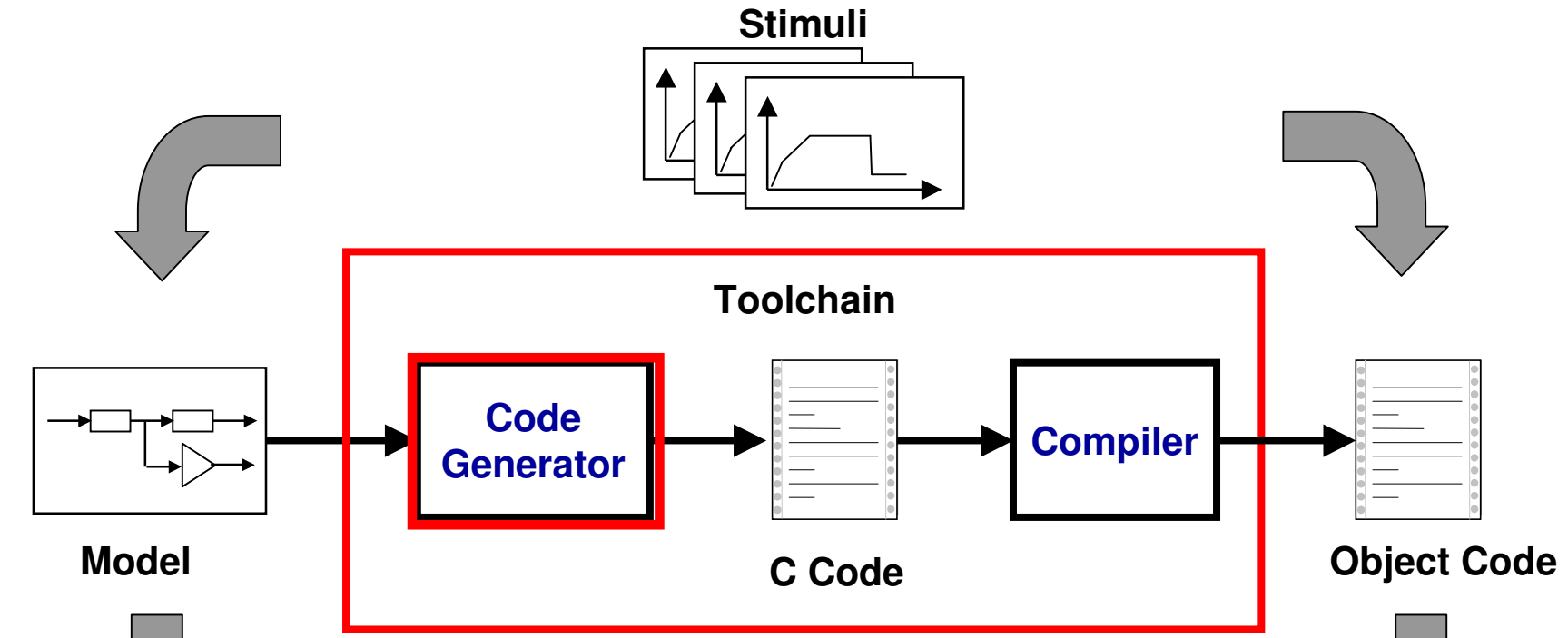
Werkzeug Qualifizierung: AVS Arbeitsgruppe

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.

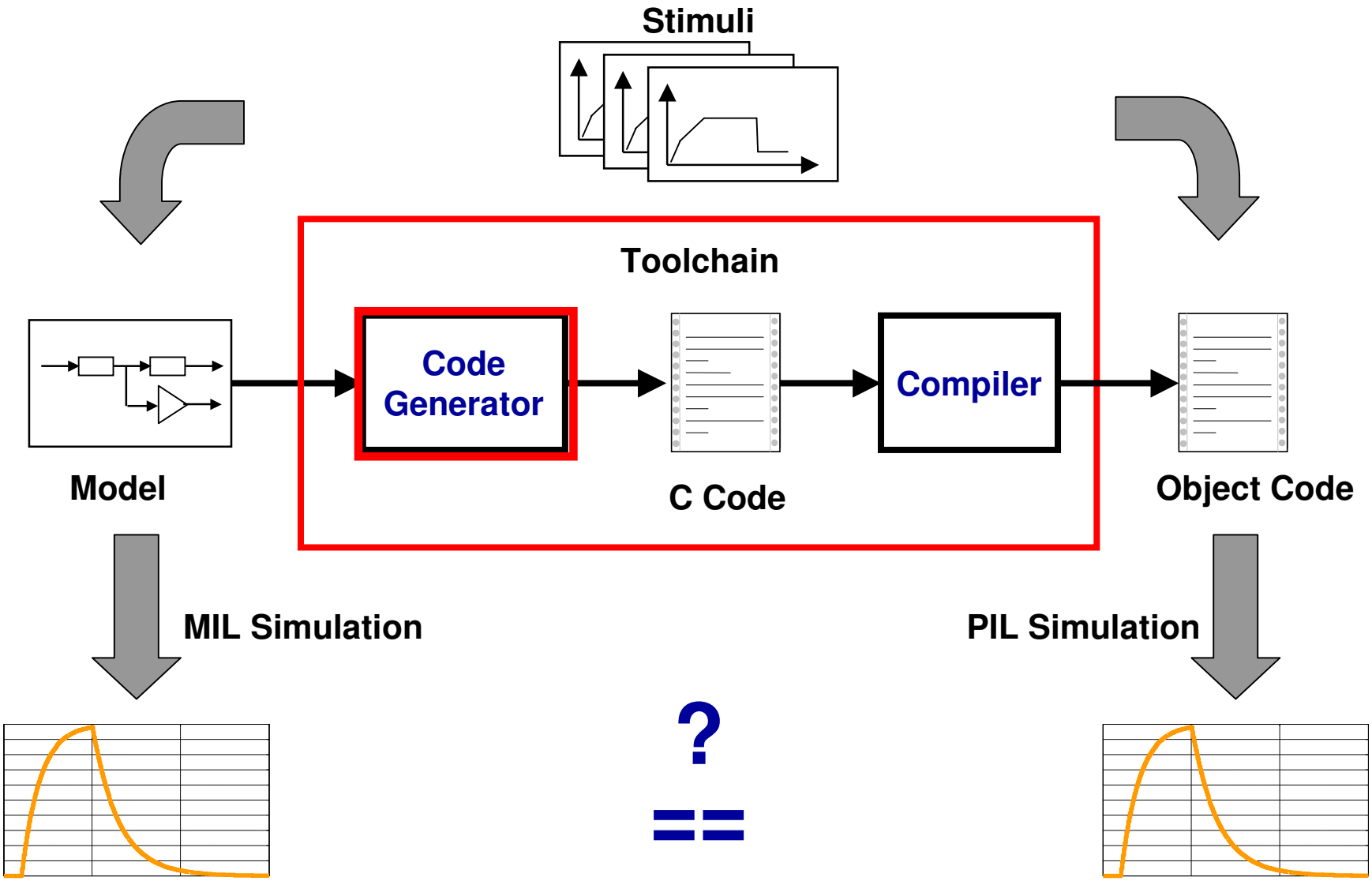
Grundlegende AVS Idee



Zentraler Gedanke für die Qualifizierung ist das systematische und automatisierte Testen von Werkzeugketten, bestehend aus Code-Generator, Compiler und Linker mit einer Testsuite.

Compiler und Linker werden hierdurch implizit für die Verwendung mit einem Code-Generator mitqualifiziert.

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:

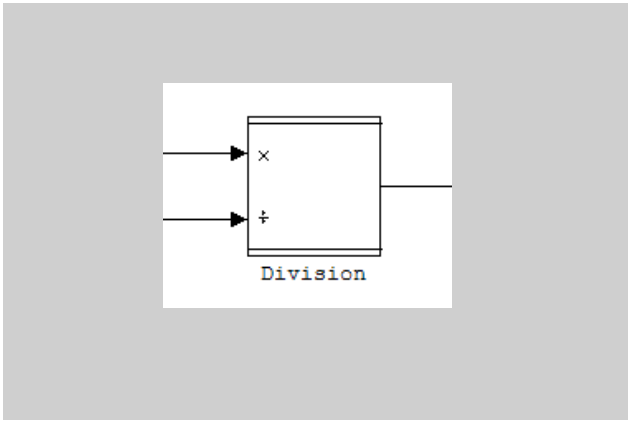
- Lesezugriff auf nicht-initialisierte Daten
- Zugriff auf Arrays außerhalb der Grenzen
- Dereferenzierung von Null-Zeigern
- ...

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

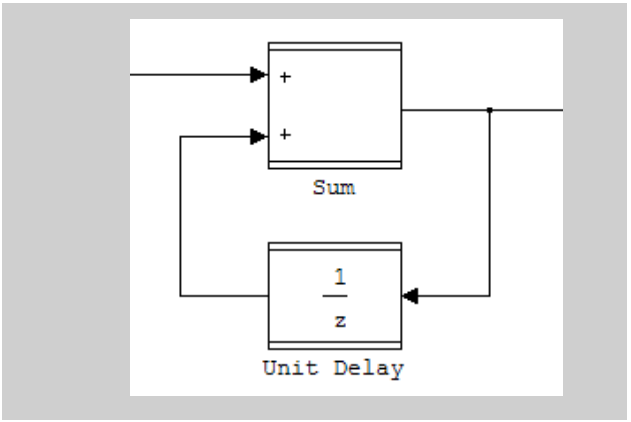
Laufzeitfehler sind auch für modellbasiertes Design relevant.

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:

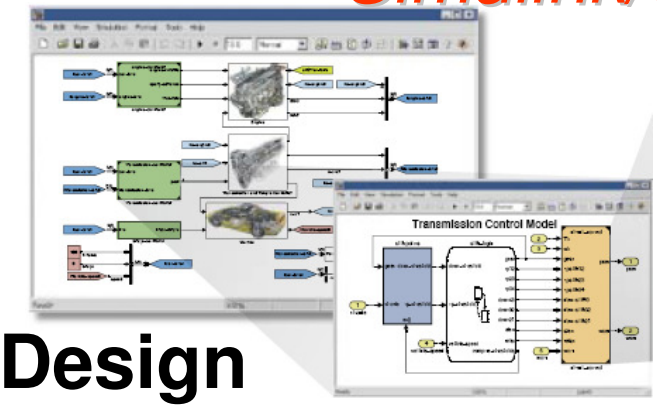
- PolySpace Verifier
- Integration für Code Generator TargetLink

TargetLink und PolySpace im Entwicklungsprozess

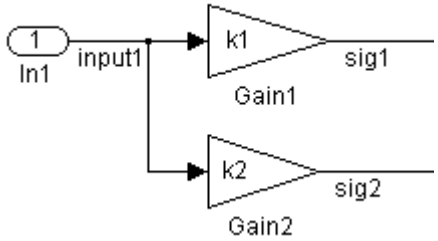
Spec

Simulink/Stateflow

Validation



Design



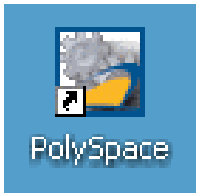
Simulation

dSPACE

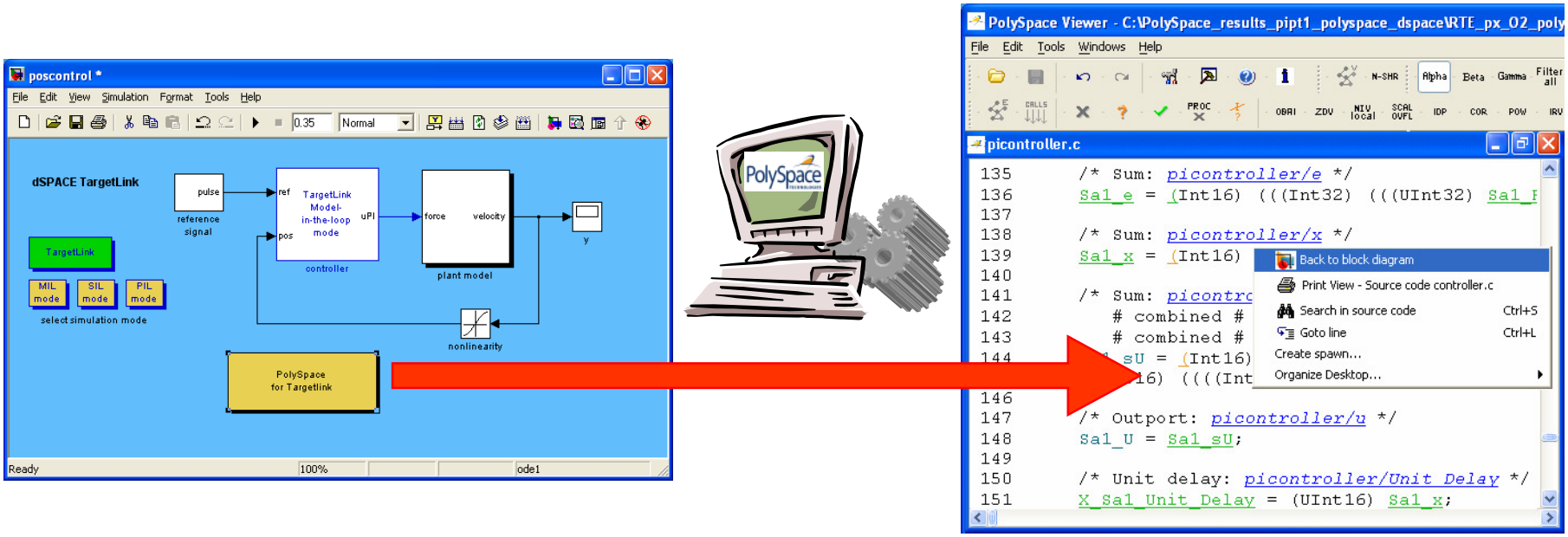
TargetLink

PolySpace TECHNOLOGIES

Code Verification (RTE free code)



TargetLink / PolySpace Integration – Workflow



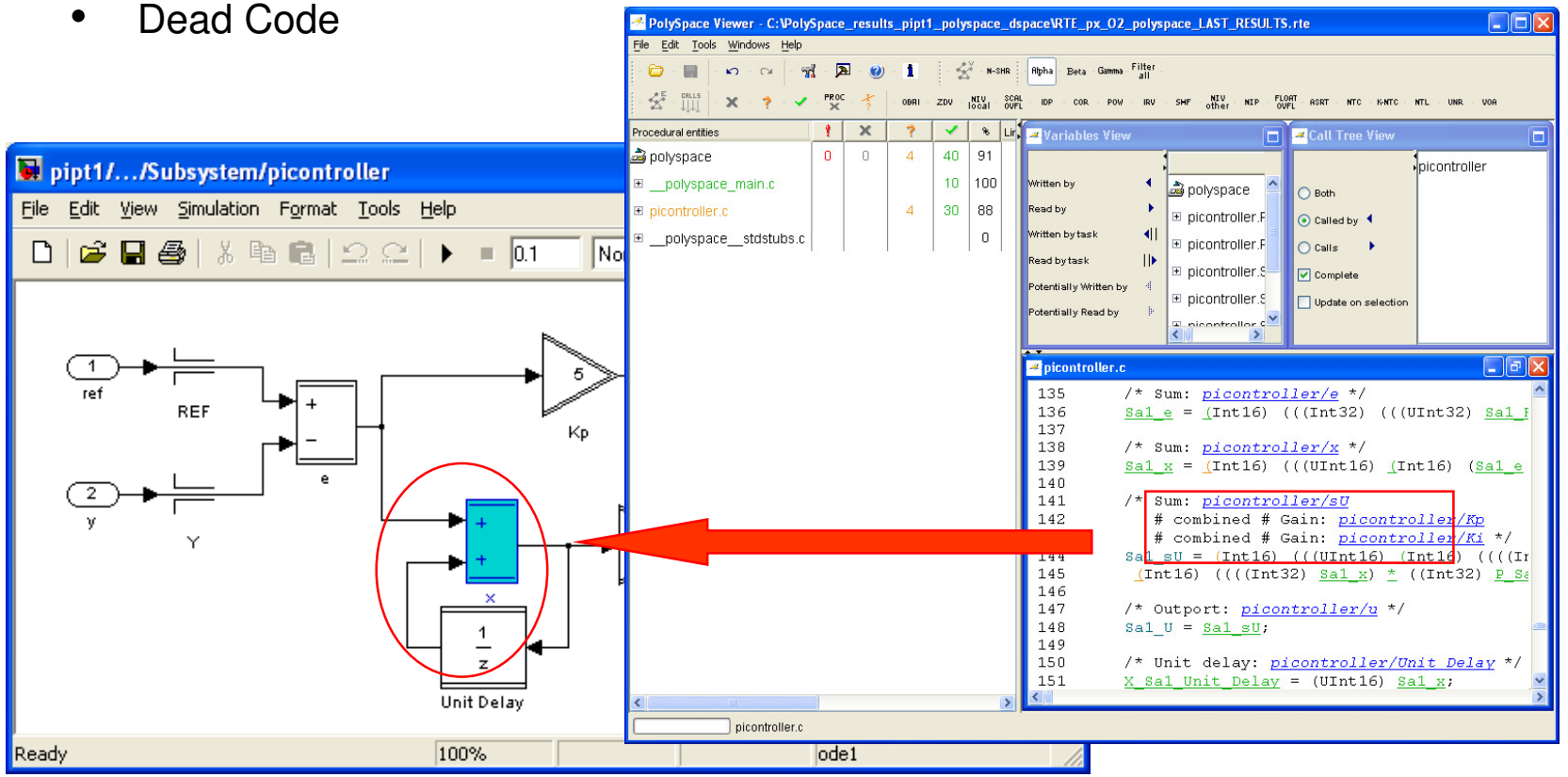
Workflow:

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

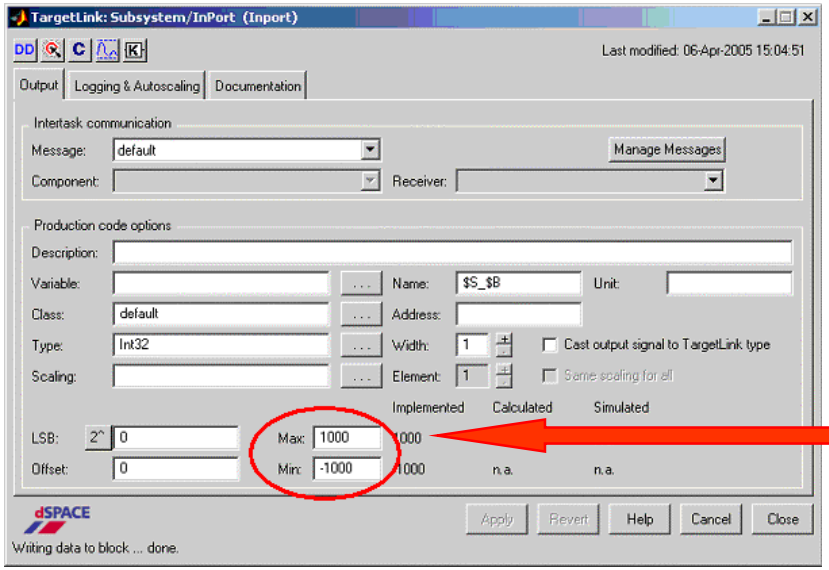


Laufzeitfehler Analyse – Vorteile auf der Modellebene

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



- 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 Bedatung 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