Komplexe Schutzfunktionen mit SPHINX realisieren

Methodik und Programmierumgebung zur Normerfüllung

In diesem Beitrag wird ein innovatives Werkzeug zur Implementierung von komplexen modellbasierten Schutzfunktionen vorgestellt. Es besteht aus einer Programmierumgebung, auf der eine problemorientierte, leicht verständliche Syntax genutzt wird. Aus dieser Beschreibung generiert es automatisch C-Code. Außerdem enthält das Werkzeug Strategien zur Fehlervermeidung, erzeugt Laufzeit-Tests und ermöglicht einen halbautomatisierten Zweigüberdeckungstest. Dieses Werkzeug lässt sich bei der Implementierung von Schutzfunktionen mittels Software als Basis zur Erfüllung der IEC 61508 nutzen.

SCHLAGWÖRTER Modellbasierte Schutzfunktionen / Engineering-Umgebung / Sicherheitsgerichtete speicherprogrammierbare Steuerungen

SPHINX – Platform for the Realization of complex Safety Functions – A method to fulfill IEC 61508 during the Implementation of model based safety functions

An innovative tool is introduced for the implementation of complex model based safety functions. It consists of a programming environment with an understandable, problem oriented syntax and generates C-Code automatically. In addition, the tool realizes failsafe strategies, generates tests which are conducted during the runtime, and allows a semi-automatic branch coverage test. The tool can be used to comply with the standard IEC 61508 when implementing safety functions as software.

KEYWORDS Model based safety function / Engineering environment / Failsafe programmable logic controller

SUSANN HAASE, BASF

Mit zunehmender Komplexität der modellbasierten Schutzfunktionen steigt der Bedarf nach Realisierungsplattformen mit höherer Funktionalität, als ihn die üblichen Sprachen nach IEC 61131 [4] bieten. Viele Realisierungen solcher Sprachen nutzen die Leistung der Hardware nicht komplett aus. Sprachen wie AWL oder ST stellen keine expliziten Testmöglichkeiten zur Erfüllung der IEC 61508 [3] zur Verfügung. Für komplexe, zum Beispiel modellbasierte Schutzfunktionen besteht daher Bedarf nach einer Plattform, die Tests einbindet, die während der Laufzeit durchgeführt werden, und auch syntaktische und semantische Analysen bei der Kompilierung bietet. Ebenso sollte diese Plattform Werkzeuge, beispielsweise zur Zweigüberdeckungsanalyse, bieten. Weder AWL noch ST leisten dies.

Eine Plattform, die diese Anforderungen abdeckt, besteht sinnvollerweise aus zwei Teilen: einem Engineering-System und einer Laufzeitumgebung. Dieser Beitrag stellt eine Engineeringumgebung vor, die auf die Nutzung einer künftigen Version der Himax (Hima) zugeschnitten ist. In dieser neuen Version wird es möglich sein, C-Code in einer sicherheitsgerichteten Umgebung ablaufen zu lassen und somit auch modellbasierte Schutzkonzepte, die auf komplexen mathematischen Algorithmen aufbauen, zu implementieren. In Kombination mit der beschriebenen Plattform können damit viele Anforderungen der IEC 61508 abgedeckt werden (1).

1. KURZVORSTELLUNG DES WERKZEUGS SPHINX

Ein Prototyp einer solchen Engineeringumgebung wurde von BASF SE entwickelt. Das Werkzeug SPHINX (Software Platform for the Highly reliable Implementation of Numerics using XML) besteht aus einer XML-Programmieroberfläche, einem Precompiler und einer Testsuite. Die XML-Programmieroberfläche ermöglicht aufgrund des problemorientierten Befehlsvorrats eine Komplexitätsreduktion im Gegensatz zu C-Implementierungen. Der eingebaute Precompiler prüft den XML-Code auf Wohlgeformtheit, übersetzt ihn mittels lexikalischer Analyse zu C-Code und erzeugt automatisch Tests, die zur Laufzeit ausgeführt werden. Die Testsuite ermöglicht es, eine Abdeckung einiger geforderter Tests, darunter einen Zweigüberdeckungstest, und Tests, die auf der abstrakten Interpretation aufbauen, durchzuführen.

2. MOTIVATION

Der Wunsch nach der Implementierung von modellbasierten Schutzfunktionen ist in den vergangenen Jahren stärker geworden. Dies hat mehrere Gründe. Zum einen ist die Systemtechnik von einer voranschreitenden Entwicklung geprägt. Mittlerweile ist es in Sicherheitssteuerungen möglich, C-Code zu implementieren. Damit lassen sich auch komplexere Schutzfunktionen, die auf mathematischen Algorithmen aufbauen, umsetzen. Zum anderen wurden robuste Modelle entwickelt, die eine deutlich realitätsnähere Beschreibung der Prozesse erlauben als bisher. Die Verbindung beider Entwicklungen kann die Auslastung von Reaktoren erhöhen.

Komplexe modellbasierte Schutzkonzepte sind Schutzkonzepte, die auf physikalisch-chemischen Modellen des Prozesses aufbauen. Der Modellbegriff unterscheidet sich damit von der Verwendung im Software-Engineering. Häufig führt die Mathematik, die hinter diesen Modellen steckt, zu deutlich schwieriger strukturierbaren Implementierungen als es bei einfachen Schutzabschaltungen, wie zum Beispiel Druck- oder Temperaturabschaltungen der Fall ist. Aufgrund des Overheads grafischer Programmierung können auch bei gut strukturierbaren Modellen Performanzprobleme auftreten. Daraus folgt der Bedarf nach flexibleren Realisierungsplattformen.

Die Implementierung mathematischer Algorithmen führt meist zu schwer überschaubarem C-Code. Daher ist es nicht ratsam, komplexere Modelle ungeachtet der Komplexität als Schutzfunktionen in C zu implemen-

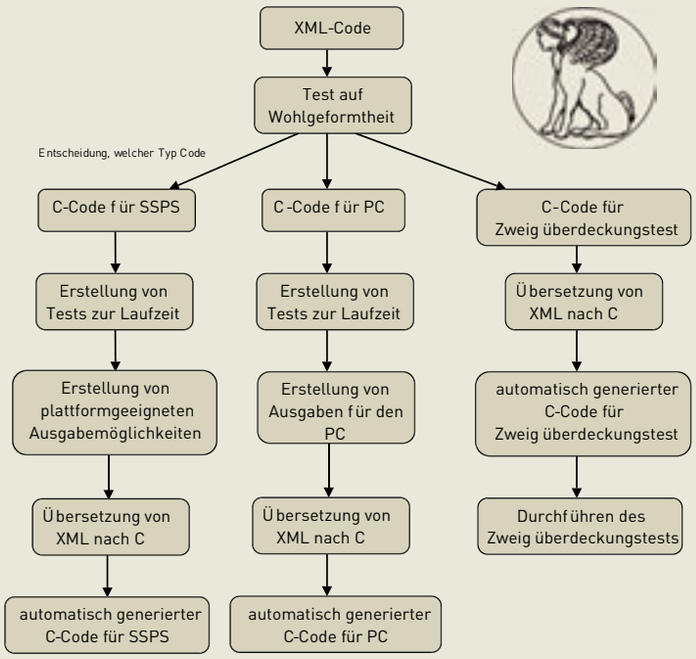

BILD 1: Ablaufdiagramm des SPHINX-Tools

tieren. Schutzfunktionen sollten möglichst einfach gestaltet sein und wenig Fehlerpotenzial aufweisen. Selbst ein C-Code, der nur den eher einfachen Gauß-Algorithmus zum Lösen linearer Gleichungssysteme abbildet, ist nicht auf den ersten Blick erfassbar. Daraus folgt, dass eine Engineering-Umgebung die Komplexität reduzieren sollte.

2.1 Realisierung in Großserien

Ein Blick in andere Industriezweige offenbart, dass es durchaus möglich ist, die Komplexität zu beherrschen und modellbasierte Schutzkonzepte erfolgreich einzusetzen. In den Produkten der Luftfahrt- und Automobilindustrie werden beispielsweise seit Jahren Systeme (ABS, ESP) eingesetzt, die mathematische Berechnungen enthalten und wegen deren Komplexität einen beachtlichen Validierungsaufwand benötigen [5], [6].

Um ein modellbasiertes Schutzsystem zu realisieren, muss zunächst das Modell entwickelt werden. Wenn anschließend die Implementierung auf einer kostengünstigen Plattform stattfindet, ist ein hoher Verifikationsaufwand nötig, da die Hardware, das Betriebssystem und die Anwendung verifiziert werden müssen. Dies ist im Automobilbereich durchaus wirtschaftlich, da die kostenintensive Entwicklung eines ABS in eine günstige Massenfertigung mündet.

2.2 Realisierung in der chemischen Industrie

In der chemischen Industrie besteht hingegen das Problem, dass die modellbasierten Online-Berechnungen, die in Sicherheitssteuerungen zu implementieren sind, meist nur für zirka 10 bis 100 Reaktoren in Frage kommen. Es besteht folglich der Bedarf nach einer preiswerten Realisierungsplattform, damit sich die Anwendung von modellbasierten komplexen Schutzkonzepten auch tatsächlich rechnet.

Wir verwenden aus diesem Grund eine sicherheitsgerichtete speicherprogrammierbare Steuerung (SSPS) als Basis, die von vornherein SIL 3-fähig ist, sodass wir uns ausschließlich um die Verifikation des Bausteins kümmern müssen, der C-Code enthält. Alle anderen Schritte werden durch den Lieferanten abgedeckt.

3. ANFORDERUNGEN AN DIE REALISIERUNGSPLATTFORM

Die speicherprogrammierbare Steuerung, auf der das modellbasierte Schutzkonzept realisiert wird, stellt systemimmanente Anforderungen an das C-Programm.

Es sind keine Befehle für die Bildschirmausgabe zulässig. Folglich gibt es keine inhärente Möglichkeit, im Fehlerfall die Fehlerstelle und den Grund zu ermitteln.

Ferner besteht die Anforderung, dass das Programm innerhalb eines Zyklus terminiert. Tritt also ein Fehler auf (zum Beispiel durch unerlaubte Eingangswerte), darf das Programm nicht abbrechen, sondern muss kontrolliert terminieren. Da das C-Programm im Grunde ein ganz normaler Baustein ist, sollten auch die Bausteinausgänge, mindestens jedoch der Ausgang, der anzeigt, ob das Ergebnis korrekt ist, belegt werden. Dieses kontrollierte Beenden des Programms ist aufwendig und fehleranfällig, wenn es händisch implementiert wird; insbesondere, wenn verschiedene Unterfunktionen genutzt werden. Es ist also sinnvoll, das Beenden des Programms durch die Realisierungsplattform automatisiert zu gewährleisten.

Die einzige Hochsprache, in der auf aktuellen sicherheitsgerichteten speicherprogrammierbaren Steuerungen implementiert werden kann, ist C. Objektorientierte Programmierung ist bei derzeitigem Stand der Technik nicht möglich. Hier liegen die drei Hauptforderungen vor:

3.1 Anforderungen aus Anwendersicht

Bei der Entwicklung einer Realisierungsplattform besteht die Frage nach den Merkmalen und Anforderungen sinnvoller Schutzfunktionen.

- Die Schutzfunktion sollte einfach zu „engineeren“ sein. Wenn das Modell das erste Mal implementiert und die Implementierung verifiziert ist, sollte es möglichst wenig Aufwand bereiten, die Schutzfunktion für einen konkreten Reaktor einzusetzen.
- Schutzfunktionen sollten durch das Bedienpersonal einfach zu überwachen sein.

- Wartungen und Anpassungen von Schutzfunktionen sollten einfach durchführbar sein. Beispielsweise sollte eine überschaubare Erweiterung oder Änderung es nicht notwendig machen, dass das gesamte Modell und seine Implementierung „re-engineert“ werden müssen.

Diese Anforderungen schließen selbstverständlich eine fest vorgegebene Strukturierung und einen eingeschränkten problemorientierten Befehlsvorrat mit ein. Dies ermöglicht eine leichtere Einarbeitung in den implementierten Code.

3.2 Anforderungen der Norm 61508

Insbesondere Schutzfunktionen, die als Software realisiert sind, müssen den Anforderungen der Norm IEC 61508 entsprechen. Hier kann die Realisierungsplattform SPHINX bei drei wesentlichen Aspekten unterstützen.

- Die im Code benutzte Syntax muss eingeschränkt werden. Es sollte beispielsweise nicht der gesamte C-Sprachumfang erlaubt sein. Zum Beispiel sollte die Verwendung von dynamischer Allokierung und Pointern in sicherheitsrelevanten Programmen vermieden werden.
- Der implementierte Code sollte Tests enthalten, die während der Laufzeit durchgeführt werden und die korrekte Ausführung des Programms sicherstellen.
- Der implementierte Code sollte geeignet dokumentiert sein, um eine spätere Nachvollziehbarkeit zu gewährleisten.

4. SPHINX IM DETAIL

SPHINX stellt eine Programmieroberfläche zur Verfügung, auf der der Anwender mithilfe von kurzen XML-Befehlen XML-Tags programmieren kann. Diese werden im Folgenden XML-Tags genannt. XML wurde aufgrund der vielen vorhandenen Standardwerkzeuge gewählt. Durch Knopfdruck startet der Precompiler, der automatisch aus dem XML-Programm C-Code generiert. Der interne Ablauf von SPHINX ist in Bild 1 dargestellt.

4.1 Die XML-Programmieroberfläche

Die XML-Tags sind kurz gehalten und stellen Rechenoperationen und numerische Verfahren dar. Außerdem sind XML-Tags für die Programmstrukturierung und zum eingeschränkten Programmieren in C vorgesehen. Ein Beispiel für ein kurzes XML-Programm zeigt Bild 2. Hier wird das lineare Gleichungssystem $Ax = b$ gelöst.

XML-Tags für arithmetische Funktionen
Häufig benutzte Variablentypen wie Fließkommazahlen, Vektoren und Matrizen werden durch Strukturtypen (Struct) dargestellt. Es werden Elemente, wie in Tabelle 1 abgebildet, definiert, die die eigentlichen Werte des Typs beinhalten. Zusätzlich gibt es Elemente, die die Länge/Größe von Vektoren beziehungsweise Matrizen angeben. Die allokierte Länge/Größe wird ebenfalls durch ein Element angegeben. Mit diesen Hilfsmitteln lässt sich das Problem überschrittener Vektorgrenzen leicht lösen. Ferner wird der Wertebereich der Elemente definiert. Jede Struct hat außerdem ein error-integer, das gleich 0 ist, wenn die Struct fehlerfrei ist und sonst einen fehlerspezifischen Wert ungleich 0 enthält. Ist die Variable fehlerbehaftet, liegt zum Beispiel ein Wert außerhalb des Wertebereichs, werden weitere Berechnungen mit dieser Variable verboten und das Programm wird kontrolliert beendet.

Durch XML-Tags werden übliche arithmetische Rechenoperationen wie die Addition, Subtraktion, Multiplikation und Division von sinnvollen Kombinationen verschiedener Operanden (Fließkommazahlen, Vektoren und Matrizen) abgebildet. Zum Beispiel können Skalarmultiplikationen oder die Multiplikation eines Vektors mit einer Matrix dargestellt werden, ebenso typische Operationen wie die Berechnung des Skalarprodukts.

Das XML-Tag für die Addition von zwei Vektoren hat beispielsweise folgende Struktur:

```
<add_vect vector1="a" vector2="b" res="c"/>
```

Der Vorteil dieser kurzen Darstellung ist, dass die for-Schleife, die in C nötig wäre, hier wegfällt. Durch integrierte Tests sind Tippfehler nahezu ausgeschlossen. Auch die in C häufiger auftretenden Copy-Paste-Fehler werden durch die verkürzte Darstellung minimiert. Die Übersetzung des XML-Tags führt zu dem Aufruf einer geprüften C-Funktion. Somit können an dieser Stelle auch out-of-bounds-Fehler ausgeschlossen werden.

XML-Tags für numerische Algorithmen
Übliche numerische Verfahren wie das explizite Eulerverfahren oder das Gauß-Verfahren zum Lösen linearer Gleichungen können ebenfalls durch XML-Tags dargestellt werden. In dem übersetzten C-Code wird eine fertig geprüfte C-Funktion eingebunden, die Tests enthält, die während der Laufzeit ausgeführt werden. So wird zum Beispiel geprüft, ob die Größen der übergebenen Variablen zusammenpassen, ob bei einem linearen Gleichungssystem die Matrix auch tatsächlich quadratisch ist und die Zeilenanzahl der Matrix auch der Länge des übergegebenen Vektors entspricht. Nach dem Lösen des Gleichungssystems wird durch Rückwärtseinsetzen geprüft, ob der berechnete Vektor mit hinreichender Genauigkeit eine Lösung des Gleichungssystem darstellt.

Strukturierende XML-Tags
Durch die XML-Programmieroberfläche kann ein besser dokumentierbarer Code implementiert werden. Hierzu tragen nicht nur die geringere Länge der XML-Tags bei,

sondern auch jene XML-Tags, die beim Strukturieren des Codes helfen. Es sind zum Beispiel XML-Tags vorgesehen, die eine Umgebung schaffen, in der Variablen deklariert und initialisiert werden, in der Funktionen eingebunden und Schnittstellen zu der äußeren Programmumgebung definiert werden. Somit können Anwender ihre Programme strukturiert und übersichtlich anlegen, während für Implementierer, die Änderungen am XML-Code vornehmen wollen, die Einarbeitung deutlich vereinfacht wird.

XML-Tags für spezifische C-Programmierung
Es besteht die Möglichkeit, dass trotz des vorhandenen XML-Sprachumfangs nicht alle Funktionen, die die Sicherheitsfunktion erfüllen muss, implementiert werden können. Es ist auch möglich, dass bei langen mathematischen Berechnungen die Performance leidet, wenn die Berechnungen auf XML-Tags aufgesplittet werden und jedes XML-Tag einzeln bei der Übersetzung automatisch Tests einfügt, die während der Laufzeit ausgeführt werden. Hierfür gibt es die Zusatzoption, direkt in C zu programmieren. Diese Umgebung wird durch XML-Tags „geöffnet“ und „geschlossen“. Um auch hier einen eingeschränkten Sprachumfang zu gewährleisten, werden durch ein Whitelisting nur vier Sprachkonstrukte erlaubt: for- und while-Schleifen, if-Bedingungen und Zuweisungen. Entspricht eine Zeile nicht einem dieser Konstrukte, wird der Code nicht übersetzt und dem Implementierer ein Fehler angezeigt. Natürlich werden in dieser Umgebung auch Pointer verboten.

In dieser C-Umgebung können somit anwendungsspezifische Funktionen sicher realisiert werden.

4.2 Der Precompiler

Der Precompiler umfasst mehrere Funktionen. Er überprüft zunächst den XML-Code auf Wohlgeformtheit. Anschließend übersetzt er ihn mittels lexikalischer Analyse nach C und baut simultan Tests ein, die während der Laufzeit durchgeführt werden. Das Ergebnis ist ein C-Code mit inhärenten Tests und einer eingeschränkten Syntax.

Die Überprüfung auf Wohlgeformtheit
Der geschriebene XML-Code wird zunächst auf Wohlgeformtheit geprüft. Hierzu wird die Dokumenttypdefinition (DTD) entsprechend festgelegt, in der die XML-Struktur definiert wird. Das bedeutet, es wird für jedes XML-Tag bestimmt, welche untergeordneten XML-Tags obligatorisch und welche optional sind. Zusätzlich sind zu jedem XML-Tag die dazugehörigen Attribute festgelegt, für die wiederum definiert wird, ob sie optional oder obligatorisch sind. Sämtliche anderen, nicht festgelegten Elemente und Attribute sind automatisch verboten. Die Struktur des zu schreibenden XML-Codes ist folglich hinreichend fest vorgegeben.

Auf diese Weise erhält der XML-Code eine standardisierte Übersichtlichkeit, die das Einarbeiten und Dokumentieren erheblich vereinfacht. Weiterhin können mit dieser Vorgehensweise viele Flüchtigkeitsfehler ausgeschlossen werden.

Das Übersetzen der XML-Befehle
Die XML-Tags und ihre Attribute werden lexikalisch analysiert und nach den Regeln der SPHINX-Spezifikation so übersetzt, dass kompilierbarer C-Code erzeugt wird. Dies geschieht, indem der XML-Code mit dem DOM XML Parser in Java gelesen wird. Die so erzeugte Baumstruktur bietet die Voraussetzung für die Umsetzung der Spezifikation.

Das Einbauen von Tests
Bei der Übersetzung werden neben dem C-Code zusätzlich Tests erstellt, die während der Laufzeit sicherstellen, dass der Code korrekt ausgeführt wird. Standardfunktionen, wie zum Beispiel die Addition von Vektoren, werden als Funktionsaufruf übersetzt. Die aufgerufene Funktion wiederum prüft, ob beispielsweise die

TABELLE 1: Aufbau eines Vektor-Structs

Struct-Element	Typ	Bedeutung
value[5]	double	Werte des Vektors (Beispiel: Länge 5)
number	int	Anzahl der belegten Elemente eines Vektors
num_max	int	maximale Anzahl der allokierten Elemente eines Vektors (kann auch eine globale Variable sein)
val_min	double	Mindestwert der Elemente
val_max	double	Maximalwert der Elemente
err	int	Fehlerbit des Structs, 0 = fehlerfrei

Länge beider Vektoren zueinander passt. Folgende Tests werden eingebaut:

- Vergleich der Länge beziehungsweise Größe von Vektoren oder Matrizen bei Rechenoperationen
- Abgleich, ob berechneter Wert innerhalb des vorgegebenen Wertebereichs liegt
- Test auf Division durch Null beziehungsweise einen Wert nahe Null
- Prüfung, ob die Eingangsmatrizen des Gauß-Algorithmus oder des vereinfachten Newton-Verfahrens regulär sind
- Prüfung durch Rückwärtseinsetzen, ob die Ergebnisse des Gauß-Algorithmus oder des vereinfachten Newton-Verfahrens korrekt sind

Da diese Tests automatisch generiert werden, sind Copy-Paste-Fehler, wie sie bei händischer Programmierung in C auftreten könnten, ausgeschlossen.

Hilfen zum Lokalisieren von Laufzeitfehlern
Wenn während der Laufzeit Fehler auftreten, ist die Lokalisierung im laufenden Betrieb meist schwierig. Als Hilfestellung sind in SPHINX im Fehlerfall folgende Ausgaben vorgesehen: der Fehlertyp, der Ort des Fehlers und die für den Fehler verantwortlichen Variablen.

Die Ausgabe des Fehlertyps ist einfach gehalten. Der Anwender bekommt eine nummerierte Liste mit möglichen Fehlertypen, und das Programm gibt im Fehlerfall eine den Fehler kennzeichnende Kennung zurück.

Während der Übersetzung werden die meisten XML-Tags zu Funktionsaufrufen übersetzt. Diesen Funktionsaufrufen werden Kennungen übergeben (siehe Bild 3). Bei der Übersetzung wird eine Liste erzeugt, von der abzulesen ist, welche dieser Kennungen zu welchem Funktionsaufruf gehört. Auf diese Weise kann auch bei wiederholtem Aufruf einer Funktion bestimmt werden, in welchem Aufruf genau der Fehler aufgetreten ist.

In der SSPS-Version des C-Codes erfolgt die Ausgabe der Variablen, die für den Fehler verantwortlich sind, über freie Bausteinausgänge. In der PC-Version erfolgt dies über Bildschirmausgaben.

Das Lokalisieren von Laufzeitfehlern und die Bestimmung des Fehlertyps werden somit deutlich erleichtert.

5. SPHINX ALS BASIS ZUR NORMERFÜLLUNG

SPHINX ist für den Anwender eine Basis zur Erfüllung der Norm IEC 61508. Auf drei Anforderungen wurde besonderer Wert gelegt: eingebaute Tests, die während der Laufzeit durchgeführt werden und Strategien zur Fehlervermeidung; ein eingeschränkter Sprachumfang sowie Code, der für Dokumentationszwecke gut geeignet ist.

5.1 Tests zur Laufzeit und Strategien zur Fehlervermeidung

Zur Laufzeit können die in Abschnitt 4 erwähnten Fehler abgefangen werden. Copy-Paste-Fehler werden durch die standardisierten XML-Tags weitestgehend vermieden. Außerdem können geprüfte C-Module, die nach dem Übersetzen automatisch aufgerufen werden, durch den Anwender nicht geändert werden.

Diese Methoden erfüllen die Anforderung der IEC 61508, Tests zur Laufzeit und Strategien zum Fehlervermeiden zu nutzen.

5.2 Eingeschränkter Sprachumfang

Im zu prüfenden C-Code wird ein eingeschränkter Sprachumfang genutzt. Da der C-Code automatisch nach den Vorgaben der SPHINX-Spezifikation generiert wird, folgt daraus, dass im C-Code nur Ausdrücke vorkommen können, die explizit spezifiziert worden sind. Schon in der Spezifikation wird daher mit einem sehr eingeschränkten Sprachumfang gearbeitet.

```
<head>
  <def_local>
    <init type="matrix" name="A" safe="y" num_cols="3"
       num_rows="3" value="{{5 7 8} {1 -1 5} {-2 3 1}}"/>
    <init type="vector" name="b" safe="y" number="3„
       value="{43 14 7}"/>
    <dec  type="vector" name="x" safe="y" number="3"/>
  </def_local>
</head>
<body>
  <solve_ls matrix="A" vector="b" res="x"/>
</body>
```

BILD 2: XML-Code für die Lösung eines linearen Gleichungssystems

```
x = solve_ls(A, b, 1);
add_vect(x, b, 2);
...
mul_vect_matr(A, x, 10);
```

BILD 3: Generierter C-Code mit übergebener Kennung zur Lokalisierung

In den XML-Tags, in denen der Anwender direkt in C programmieren kann, wird durch das zuvor beschriebene Whitelisting der Sprachumfang ebenfalls eingeschränkt.

Somit enthält der generierte C-Code nur eine beschränkte Anzahl erlaubter C-Befehle. Damit ist die Anforderung der IEC 61508, eine eingeschränkte Syntax zu nutzen, erfüllt.

5.3 Dokumentation des Codes

Der XML-Code ist für die Dokumentation gut geeignet. Durch leistungsfähige XML-Tags verkürzt sich die Länge des zu dokumentierenden Codes auf ungefähr ein Viertel. Zusätzlich stehen XML-Tags zur Strukturierung zur Verfügung, die die Einarbeitung in den Code und dessen Verständnis erleichtern.

Der XML-Code erfüllt daher die Anforderung der Norm nach geeigneter Code-Dokumentation.

6. TESTSUITE ZUR DYNAMISCHEN ANALYSE

Der nach klaren Regeln erstellte XML-Code ermöglicht durch die Übersetzung nach C die automatisierte Anwendung diverser weiterer Tests. Im Gegensatz zu kommerziellen Werkzeugen, die den Code dynamisch testen und hierzu erst aufwendig analysieren müssen, liefert SPHINX diese Analyse inhärent mit.

Beispielsweise ist die Realisierung des von der IEC 61508 geforderten Zweigüberdeckungstests verhältnismäßig einfach umsetzbar. Durch die gegebene Struktur lassen sich Zähler implementieren, die anzeigen, ob ein Zweig durchlaufen wurde. Die Funktion des Zweigüberdeckungstests lässt sich außerdem sehr einfach an- und abschalten. Ein Parameter beim Precompileraufruf gibt an, welcher Modus von Code erzeugt werden soll. Somit ist sichergestellt, dass beim Zweigüberdeckungstest der Code, der später auf der SSPS läuft, getestet wird.

7. FAZIT UND AUSBLICK

Die Engineering-Umgebung SPHINX reduziert die Komplexität des zu implementierenden Codes von modellbasierten Schutzkonzepten auf ein Minimum. Infolgedessen ist es möglich, auch Schutzfunktionen, die auf mathematischen Algorithmen aufbauen, kostengünstig in einer SSPS zu implementieren. Durch die übersichtliche Struktur des XML-Codes sind Änderungen leichter zu implementieren, und der automatisch generierte Code erleichtert die Prüfung der Schutzfunktion. Selbstverständlich können mit SPHINX auch logisch/diskrete Probleme bearbeitet werden.

Ferner stellt SPHINX die Basis zur Erfüllung der IEC 61508 dar. Durch ihre Eigenschaften werden die dargestellte Anforderungen erfüllt.

Die hier vorgestellte Plattform SPHINX ist Teil einer Gesamtmethodik mit der die IEC 61508 für modellbasierte Schutzkonzepte abgedeckt wird. Diese wird in einem folgenden Beitrag vorgestellt.

MANUSKRIPTEINGANG
06.12.2011
Im Peer-Review-Verfahren begutachtet

REFERENZEN

[1] Börcsök, J.: Funktionale Sicherheit. Hüthig Verlag, Heidelberg 2006
[2] Liggesmeyer, P.: Software-Qualität. Spektrum Akademischer Verlag 2002
[3] IEC 61508-3 EN: Functional safety of electrical/electronic/programmable electronic safety related systems – Part 3: Software Requirements. April 2010
[4] IEC 61131-3 EN: Programmable controllers – Part 3: Programming languages. Januar 2003
[5] Dniestrowski, A, Guillaume, J. M., Mortier, R.: Software engineering in avionics applications. ICSE '78 Proceedings of the 3rd international conference on Software engineering (1978) S. 124-131
[6] A. Dold and M. Trapp. Herausforderungen und Erfahrungen eines OEM bei der Gestaltung sicherheitsgerechter Prozesse. GI Jahrestagung (2007) H. 2, S. 536–540

AUTORIN

Dipl.-Math. **SUSANN HAASE** (geb. 1987) ist seit 2011 bei der BASF SE, Ludwigshafen, im Fachzentrum Automatisierungstechnik auf dem Arbeitsgebiet der funktionalen Sicherheit tätig. Themenschwerpunkt ist die Entwicklung und Anwendung eines Implementierungs- und Prüfkonzepts für modellbasierte Schutzsysteme.

BASF SE,
D-67056 Ludwigshafen,
Tel. +49 (0) 621 607 41 76,
E-Mail: susann.haase@basf.com