# Formale Verifikation von Sicherheits-Funktionsbausteinen der PLCopen auf Modell- und Code-Ebene

Sebastian Biallas[*], Georg Frey[°], Stefan Kowalewski[*], Bastian Schlich[*], Doaa Soliman[°]

[*]Lehrstuhl Informatik 11
Software für eingebettete Systeme
RWTH Aachen, 52074 Aachen
{biallas|schlich|kowalewski}@embedded.rwth-aachen.de

[°]Lehrstuhl für Automatisierungstechnik
Universität des Saarlandes
66123 Saarbrücken
{soliman|frey}@aut.uni-saarland.de

**Abstract:** Im vorliegenden Beitrag wird der Einsatz formaler Methoden zum Nachweis der Korrektheit von Sicherheitssteuerungen demonstriert. Im Bereich der speicherprogrammierbaren Steuerungen hat die PLCopen Software-Bausteine für Sicherheitssteuerungen spezifiziert. Auf Basis dieser semi-formalen Spezifikationen entwickeln wir sowohl formale Modelle als auch konkrete Implementierungen. Auf beiden Ebenen erfolgt eine Verifikation der spezifizierten Eigenschaften mit jeweils geeigneten Model-Checking-Verfahren und Werkzeugen (Uppaal auf Modellebene sowie [mc]square auf Code-Ebene). Der Beitrag zeigt wie die beiden Ebenen ineinander greifen und demonstriert, dass durch den vorgestellten Ansatz auch Unklarheiten bzw. Interpretationsspielräume in der semi-formalen Spezifikation aufgedeckt werden können. Das Vorgehen wird exemplarisch am Beispiel eines Notaus-Bausteins demonstriert.

**Stichworte:** Funktionale Sicherheit, Speichprogrammierbare Steuerung, Formale Verifikation, Model-Checking

## 1 Einleitung

Der Entwurf von Sicherheitssteuerungen gewinnt zunehmend an Bedeutung in der industriellen Praxis. Zum einen steigen die gesetzlichen Anforderungen an die funktionale Sicherheit, und zum anderen dringt die Automatisierungstechnik in immer neue – oft auch kritische – Bereiche vor. Besonders schwierig ist in diesem Zusammenhang der Nachweis der Sicherheit von programmierbaren Systemen, also der Software. Die IEC 61508 zum Entwurf sicherer programmierbarer Systeme empfiehlt hier explizit den Einsatz formaler Methoden (ohne jedoch genauer darauf einzugehen wie dieser erfolgen soll). Zur Umsetzung der IEC 61508 im Bereich speicherprogrammierbarer Steuerungen wurde von der PLCopen (Nutzerorganisation der IEC 61131) ein Satz von Funktionsbausteinen spezifiziert, aus denen eine sichere Steuerungsapplikation durch Verknüpfung aufgebaut werden kann. Die Spezifikation [PLCopen2006] enthält neben allgemeinen Punkten zum Vorgehen insbesondere die detaillierte, semi-formale Beschreibung von zwanzig spezifischen Funktionsbausteinen. Für jeden dieser Bausteine beinhaltet die Beschreibung:

1) eine Definition der Schnittstellen,
2) eine textuelle Beschreibung der Eigenschaften,
3) einen Automatengraphen zur Beschreibung der Dynamik und
4) einige Timing-Diagramme zur Veranschaulichung des temporalen Verhaltens.

In Tabelle 1 ist diese Spezifikation exemplarisch für den Baustein SF_EmergencyStop zur Realisierung einer Notaus-Funktion dargestellt.

In einem ersten Schritt wird nun auf Basis dieser Spezifikation ein formales Modell des Bausteins entworfen. Unter Berücksichtigung des ersten Teils der Spezifikation wird der dritte Teil in einen zeitbewerteten Automaten in Uppaal überführt. Dessen Zeitverhalten wird dann mittels Simulation gegen die im vierten Teil spezifizierten Verläufe validiert. Schließlich werden die im zweiten Teil gegebenen Eigenschaften formalisiert und mittels Model-Checking am Modell verifiziert [SoliFrey2009].

Das so gewonnene verifizierte Modell eines Sicherheitsfunktionsbausteins kann nun zur Verifikation von Sicherheitsapplikationen benutzt werden. Dazu wird eine Bibliothek formaler Modelle aufgebaut, aus der sich dann analog zur implementierten Applikation ein formales Modell instanziieren lässt.

In einem zweiten Schritt wird basierend auf der PLCopen-Spezifikation ausgehend vom bereits verifizierten formalen Automatenmodell eine Implementierung in Anweisungsliste (AWL) nach IEC 61131-3 entworfen. Diese Implementierung kann entweder per Hand oder auch automatisch durchgeführt werden. Der so entstehende Block wird nun seinerseits gegen die Eigenschaftsbeschreibung verifiziert. Dabei werden zum einen die Formeln verwendet, welche bereits im vorhergehenden Schritt verwendet wurden, um die Eigenschaften aus dem zweiten Teil der Spezifikation zu zeigen. Zum anderen werden Formeln verwendet, welche implementationsabhängige Details überprüfen, die auf Ebene der zeitbewerteten Automaten nicht sichtbar sind.

Durch die Verwendung von formaler Verifikation auf Ebene der Modelle und auf Ebene des Codes wird es möglich, unterschiedliche Fehler aufzudecken und zu korrigieren. Auf der Ebene der Modelle ist eine erste Überprüfung der funktionalen Eigenschaften einfacher, da Implementationsdetails auf dieser Ebene keine Rolle spielen. Sobald das Modell fertig gestellt ist, kann eine Implementierung erfolgen. Nach der Implementierung muss überprüft werden, ob die geforderten Eigenschaften weiterhin erfüllt sind und ob die Implementierung korrekt ist. Dieses ist sowohl im Falle einer manuellen Implementierung als auch im Falle einer automatischen Code-Erstellung notwendig, da in den seltensten Fällen verifizierte Compiler verwendet werden.

Im Folgenden wird exemplarisch einer der PLCopen-Bausteine beschrieben (Abschnitt 2). Danach wird zunächst näher auf die Verifikation auf Modellebene eingegangen (Abschnitt 3) und anschließend wird die Verifikation auf Code-Ebene betrachtet (Abschnitt 4). Die Vorteile,

die sich aus der getrennten Untersuchung auf beiden Ebenen ergeben, werden abschließend diskutiert (Abschnitt 5).

## 2 Beschreibung des Bausteins

SF_EmergencyStop ist ein sicherheitsrelevanter Baustein zur Ansteuerung eines Notausschalters. Er kann zum Betrieb von Notausschaltungen (Stoppkategorie 0), mit zusätzlicher Hardware aber auch zum Betrieb einer Nothaltfunktion (Stoppkategorie 1 oder 2) verwendet werden. Die Beschreibung aus der PLCopen Spezifikation findet sich in Tabelle 1. Der Baustein besitzt fünf Eingänge und vier Ausgänge. Der Ausgang Ready wird aktiviert (TRUE) wenn der Eingang Activate gesetzt wird. Am Eingang S_EStopIn wird das Notaussignal angeschlossen. Der Eingang ist in negativer Logik ausgeführt. D.h. das Signal ist TRUE solange kein Notaus angefordert wird. Sobald das Signal auf FALSE geht, wird der Ausgang S_EStopOut, der per Default TRUE ist, auch auf FALSE zurückgesetzt. Dieser Ausgang zeigt also den Notaus-Fall an. Um den Notaus-Fall wieder verlassen zu können, reicht eine Rücknahme (auf TRUE zurücksetzen) des Eingangs S_EStopIn im Allgemeinen nicht aus. Vielmehr muss zuerst ein Reset durchgeführt werden. Hierfür sieht die Spezifikation vier Möglichkeiten vor (immer bei S_EStopIn = FALSE und Activate = TRUE):

1. Wenn S_AutoReset = TRUE gilt, erfolgt ein automatisches Rücksetzen.
2. Für den Fall S_AutoReset = FALSE wird auf eine ansteigende Flanke am Eingang Reset gewartet
3. Für den Fall S_StartReset = TRUE erfolgt ein automatisches Rücksetzen beim Neustart (Aktivierung) des Bausteins.
4. Für den Fall S_StartReset = FALSE muss wiederum zunächst eine Bestätigung durch eine ansteigende Flanke am Eingang Reset erfolgen.

Die ersten beiden Fälle beschreiben somit das Rücksetzen im laufenden Betrieb der Sicherheitsfunktion, die beiden letzteren das Rücksetzen nach einer neuen Aktivierung.

Die beiden verbleibenden Ausgänge (Error und DiagCode) dienen der Anzeige von Fehlerzuständen sowie der Ausgabe von Diagnose-Informationen. Sie sind für das funktionale Verhalten zunächst nicht relevant und werden im Folgenden auch nicht weiter betrachtet.

Der in der Tabelle 1 angegebene Automatengraph macht den Ablauf der Aktivierung des Bausteins und insbesondere die Rücksetzfunktion nochmals deutlich. Dabei ist zu beachten, dass die am rechten Rand notierte Ausgangsbelegung jeweils für alle Zustände ober bzw. unterhalb der gestrichelten Linien gilt. Ready ist also nur im Zustand *Idle* FALSE, sonst TRUE. Wohingegen S_EStopOut im Zustand *Safety Output Enabled* TRUE ist und sonst FALSE. Die Zahlen in den kleinen Kreisen an den Transitionen geben Prioritäten an (kleinere Zahl entspricht

höherer Priorität). Zusätzlich ist zu beachten, dass aus jeden Zustand mit höchster Priorität ein Übergang in den Zustand *Idle* erfolgt sobald der Eingang Activate zurückgesetzt wird (FALSE). Das Zeitdiagramm zeigt für ein gegebenes Szenario am Eingang den entsprechenden Verlauf der Ausgangsgrößen.

## 3 Verifikation der Zeitautomaten der PLCopen-Bausteine

Die Umsetzung der PLCopen-Bausteine in Zeitautomaten (TA) in Uppaal erfolgt direkt auf Basis des gegebenen Automatengraphen. Dabei ist zu beachten, dass in diesem Transitionen teilweise nicht explizit angegeben sind. Zusätzlich ist die Priorisierung der Transitionen in Uppaal durch entsprechende Erweiterungen an den Transitionsbedingungen zu realisieren (vgl. Bild 1).

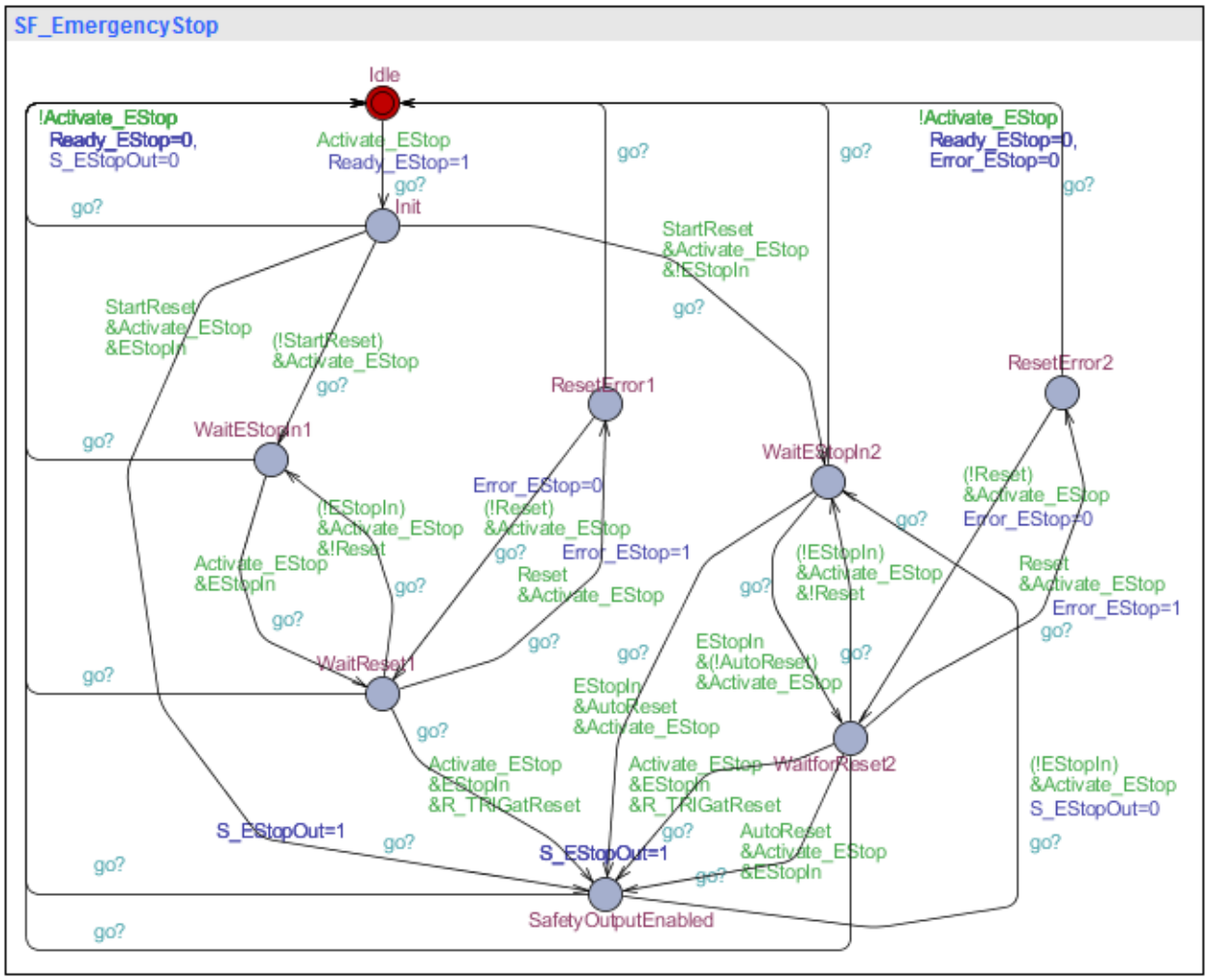



**Bild 1: Formales Modell des SF_EmergencyStop in Uppaal**

Nach Umsetzung eines Bausteins wird zunächst in der Simulation das Zeitverhalten mit den gegebenen Timingdiagrammen verglichen. Im Anschluss daran erfolgt die formale Verifikation. Hierbei werden zunächst Zustandseigenschaften überprüft, die z. B. die korrekte Ausgabe in verschiedenen Situationen beschreiben. Auch die Deadlockfreiheit wird generell untersucht. Schließlich werden die spezifischen Eigenschaften eines Bausteins aus der Spezifikation zunächst in die Uppaal-Eingangssprache (angelehnt an Computation Tree Logic (CTL)) übersetzt und dann verifiziert. Im Beispiel des SF_EmergencyStop lautet eine der Eigenschaften:

*(1)* *The S_EStopOut enable signal is reset to FALSE as soon as the S_EStopIn input is set to FALSE.*

Diese Eigenschaft bezieht sich auf das Rücksetzen des Ausgangs S_EStopOut von FALSE nach TRUE. Dieses Rücksetzen soll genau dann erfolgen wenn der Eingang S_EStopIn FALSE ist. Die Eigenschaft kann direkt in die Eingangssprache des Modelcheckers von Uppaal übersetzt und am Modell verifiziert werden:

*(1)* *A[] S_EStopOut &!S_EStopIn imply !S_EStopOut*

Eine weitere, umfangreichere Eigenschaft lautet:

*(2)* *The S_EStopOut enable signal is reset to True only if the S_EStopIn input is set to True and a reset occurs. The enable reset depends on the defined S_StartReset, S_AutoReset, and Reset inputs.*

Diese Eigenschaft muss aufgrund der „only if“ Bedingung für die Formalisierung in zwei Teile aufgespalten werden. Zunächst die Überprüfung, dass kein Rücksetzen erfolgt, wenn die entsprechende Bedingung nicht gilt (2a). Zusätzlich die Überprüfung, dass ein Rücksetzen erfolgt, wenn die Bedingung gilt (2b). Dies kann wie folgt in Uppaal ausgedrückt und verifiziert werden:

*(2a)* *A[] Ready & (!S_EStopOut) & !(S_EStopIn & (R_TRIGatReset | S_AutoReset)) imply !S_EStopOut*

*(2b)* *A[] Ready & (!S_EStopOut) & S_EStopIn & (R_TRIGatReset | S_AutoReset) imply S_EStopOut*

Der Ausdruck R_TRIGatReset beschreibt dabei die steigende Flanke am Eingang Reset. Mit diesen Bedingungen ist das Rücksetzen im laufenden Betrieb geprüft (Fälle 1 und 2 aus der Beschreibung in Abschnitt 2). Zwei weitere entsprechende Bedingungen sind nötig um das Rücksetzen bei erneuter Aktivierung (aus dem Zustand *Idle*) des Bausteins zu prüfen (Fälle 3 und 4 in Abschnitt 2). Nach einigen Iterationen in der Formulierung konnten auch alle weiteren Eigenschaften des Bausteins am Beispiel verifiziert werden.

# 4 Verifikation des Anweisungslisten-Codes der PLCopen-Bausteine

Die zuvor beschriebenen PLCopen-Bausteine wurden nach der Modellierung als Zeitautomaten in Uppaal in Code in der Programmiersprache Anweisungsliste (AWL) überführt. Dieser Vorgang wurde händisch anhand eines Algorithmus durchgeführt. In Zukunft soll diese Überführung automatisch mithilfe eines Werkzeugs durchgeführt werden.

In dem zweiten Verifikationsschritt des Vorhabens wird überprüft, ob alle Eigenschaften, die für die Zeitautomatenmodelle nachgewiesen werden konnten, auch im AWL-Code gelten, d.h., dass die Übersetzung keine der geforderten Eigenschaften verletzt. Zusätzlich soll nachgewiesen werden, dass bestimmte implementationsabhängige Eigenschaften erfüllt sind.

Der AWL-Code wird ebenfalls mit Model-Checking überprüft. Allerdings wird zum Model-Checking des AWL-Codes der Model-Checker [mc]square verwendet, welcher direkt auf AWL-Code arbeiten kann. [mc]square wurde am Lehrstuhl Software für eingebettete Systeme der RWTH Aachen entwickelt [SchlBrWe2009, SchlWeKo2009]. Zusätzlich zum AWL-Code wird im Model-Checking eine formale Spezifikation benötigt, welche in [mc]square mithilfe der Logik CTL beschrieben wird. Wenn die Spezifikation nicht erfüllt ist, erstellt [mc]square ein Gegenbeispiel, welches im AWL-Code und als Zustandsraumgraph dargestellt wird.

Um AWL-Code zu überprüfen wird dieser in [mc]square geladen. [mc]square enthält einen SPS-Simulator und kann so den Zustandsraumgraph automatisch erstellen. Dies geschieht durch sukzessive Belegung aller möglichen Eingangsvariablen und Ausführung der Zyklus, um die Nachfolgezustände zu erhalten.

Es werden alle Spezifikationen, die mit Uppaal überprüft wurden, nach CTL übersetzt, um sie auch mit [mc]square zu verifizieren. In Uppaal wird eine Untermenge der CTL-Operatoren unterstützt und daher ist diese Übersetzung immer möglich. Bei der Übersetzung müssen allerdings Aussagen über Zeit entfernt werden, da [mc]square keine Aussagen über Zeit zulässt. Die Aussagen über Zeit können aber umgeschrieben werden, da diese immer die Form haben: „Wenn ein Timer abgelaufen ist, dann passiert das folgende Ereignis.“ Diese Formeln können einfach in zwei Formeln umgeschrieben werden: in einer wird angenommen, dass der Timer abgelaufen ist und in der anderen Formel wird angenommen, dass der Timer noch nicht abgelaufen ist.

Zusätzlich zu diesen Formeln, werden auch Formeln hinzugefügt, die Aussagen über implementationsabhängige Details machen. Die PLCopen-Bausteine sind alle als Zustandsautomaten implementiert. Bei der Übersetzung in den AWL-Code kann es passieren, dass Zustände nicht stabil sind, Zustände übersprungen werden oder Zustände nicht erreicht werden. Dieses wird mithilfe von Formeln überprüft, die diese implementationsabhängigen Details abbilden. Da diese implementationsabhängigen Details sich nur im AWL-Code

widerspiegeln, können wir hier noch Fehler finden, die sich bei der Überprüfung der Zeitautomaten in Uppaal nicht feststellen lassen.

Der AWL-Code des SF_EmergencyStop-Blocks hat ungefähr 200 Zeilen Code. Um diesen zu überprüfen, werden zuerst die Formeln übernommen, die für die zeitbewerteten Automaten in Uppaal verwendet wurden. Direkt aus Uppaal wurden beispielsweise die oben erwähnten Formeln (1) und (2a)/(2b) übernommen und ebenfalls verifiziert.

Neben diesen Formeln werden auch Formeln verwendet, die die implementationsabhängigen Eigenschaften beschreiben. Für die Implementierung als Zustandsautomat wurden interne Variablen *S_1* bis *S_9* angelegt, die festlegen, ob sich der Baustein in entsprechendem Zustand befindet (siehe Tabelle 1). Es muss überprüft werden, ob es nur gültige Transitionen zwischen den Zustandsvariablen gibt, also ob beispielsweise nur die Zustände *S_1* und *S_2* von *S_2* zu erreichen sind. Dies wird durch diese Formel ausgedrückt:

(3) *AG (S_1 => AX (S_1 | S_2))*

Bei der Verifikation dieser Formel durch [mc]square wurde jedoch ein Gegenbeispiel gefunden. Das Gegenbeispiel zeigte einen direkten Übergang von *S_2* nach *S_7* bei einer bestimmten Reihenfolge der Belegung der Eingänge. Dieser Übergang kommt dadurch zustande, dass die Implementation des Bausteins mehrere Transitionen in einem Zyklus durchlaufen kann. Dies hat eine Unklarheit in der Spezifikation der PLCopen-Bausteine offengelegt: Die Semantik der PLCopen-Bausteine ist wie bereits erwähnt in Form von Zustandsautomaten spezifiziert. Eine bestimmte Belegung der Eingänge erfordert eine Transition von einem Zustand in einen anderen Zustand. Allerdings geht aus den Diagrammen nicht eindeutig hervor, ob nur eine einzige Transition pro Zyklus durchlaufen werden darf, also ob jeder Zustand wirklich einmal angenommen werden muss. Dies lässt sich am Beispiel des Zustandsautomaten des Notaus-Bausteins wie folgt nachvollziehen: Der Baustein beginnt im Zustand *Idle* (*S_1*). Ein Signal am Activate-Eingang versetzt den Baustein in den Zustand *Init* (*S_2*). Wird nun (S_StartRest AND NOT S_EStopIn) gesetzt, so wird Zustand *Wait for S_EStopIn2* (*S_7*) angenommen. Mit [mc]square fanden wir jedoch einen direkten Übergang des Programms von *S_1* nach *S_7*, wenn (Activate AND S_StartReset AND NOT S_EStopIn) gesetzt ist. Der Init-Zustand *S_2* wurde dabei also übergangen, weil in einem Zyklus direkt zwei Transitionen durchgeführt wurden.

Dieses Verhalten ist dann problematisch, wenn Zustände übersprungen werden können, die ein wichtiges Signal auf einen Ausgang legen. Das Programm wurde daraufhin umgeschrieben, so dass pro Zyklus nur ein Zustandsübergang möglich ist.

Eine andere Eigenschaft, die für eine derartige Implementierung außerdem gelten muss, ist, dass immer nur genau eine dieser Variablen gesetzt ist. Dies haben wir mit folgender Formel überprüft:

(4) $AG\ (S\_1 + S\_2 + S\_3 + S\_4 + S\_5 + S\_6 + S\_7 + S\_8 + S\_9 = 1)$

Wir bilden hier die Summe über die Zustandsvariablen *S_i*. Diese Formel sagt also aus, dass in jedem Zustand des Bausteins gilt, dass genau eine der internen Variablen *S_i* gesetzt (also gleich 1) ist. [mc]square hat jedoch festgestellt, dass diese Spezifikation vom AWL-Code verletzt wird. Es gab einen Durchlauf, der über *S_1*, *S_2* und *S_3* in einen Zustand führte, in dem *S_3* und *S_5* zugleich gesetzt sind. Die genaue Belegung der Eingänge, die zu diesem Verhalten führen, wurde in Form einen Gegenbeispiels von [mc]square ausgegeben.

Wir konnten nun den Code anhand dieses Gegenbeispiels analysieren und die Stelle im AWL-Code identifizieren, welche zur gleichzeitigen Aktivierung von *S_3* und *S_5* führt. Es wurde dabei festgestellt, dass der Fehler durch eine fehlerhafte Klammerung ausgelöst wurde. Dieser Fehler konnte nur durch die Analyse des AWL-Programmtextes gefunden werden, da er nicht im ursprünglichen Automatengraphen auftaucht, sondern erst durch die fehlerhafte Übertragung des Automaten in AWL entstand.

In der korrigierten Version waren alle obigen Formeln schließlich erfüllt.

## 5 Fazit

Dieser Beitrag beschreibt einen neuen Ansatz zur Verifikation von Software für Sicherheitssteuerungen, welcher auf zwei Ebenen arbeitet. Zum einen wird eine Verifikation auf Modell-Ebene durchgeführt, um Eigenschaften unabhängig von der später folgenden Implementierung zu zeigen. Auf dieser Ebene wird verifiziert, ob die Konzepte und Algorithmen richtig sind. Danach erfolgt entweder eine händische Implementierung oder eine automatische Code-Erstellung. Nun muss für den Code gezeigt werden, dass zum einen die Eigenschaften noch erfüllt sind, die im Modell erfüllt waren. Zum anderen muss gezeigt werden, dass die implementationsabhängigen Details auch korrekt arbeiten.

In diesem Beitrag wurde dieses Verfahren auf Sicherheits-Funktionsbausteine der PLCopen angewandt. Zuerst wurden zeitbewertete Automaten erstellt und diese wurden dann mithilfe von Uppaal validiert und verifiziert. Nach einer händischen Implementierung wurde der Model-Checker [mc]square verwendet, um auf der Code-Ebene zu zeigen, dass die Bausteine sich konform zur Spezifikation verhalten. Dabei wurden zwei Fehler in den implementationsabhängigen Details gefunden, die so auf Ebene der Modelle nicht gefunden werden können. Dieses zeigt, dass eine Verifikation von Sicherheits-Funktionsbausteinen möglich ist. Weiterhin zeigt sich, dass eine Verifikation sowohl auf Ebene der Modelle als auch auf Ebene des Codes für sicherheitskritische Systeme nötig ist. Auf beiden Ebenen können Informationen gewonnen werden, die auf der jeweils anderen Ebene nur schwer oder gar nicht zu gewinnen sind.

# 6 Literatur


[IEC61131-3] IEC 61131-3: Programmable Controllers Part 3: Programming Languages. 2nd Ed: 2003.

[IEC61508] IEC 61508: Funktionale Sicherheit sicherheitsbezogener elektrischer/ elektronischer/programmierbarer elektronischer Systeme. 1998.

[PLCopen2006] PLCopen TC5: Safety Software Technical Specification, Version1.0, Part 1: Concepts and Function Blocks. PLCopen, Germany, 2006.

[PLCopen2008] PLCopen TC5: Safety Software Technical Specification, Version1.0, Part 2: User Examples. PLCopen, Germany, 2008.

[SchlBrWe2009] Schlich, B., Brauer, J., Wernerus, J., Kowalewski, S.: Direct Model Checking of PLC Programs in IL. Proc. 2nd IFAC Workshop on Dependable Control of Discrete Systems (DCDS), Bari, Italy, 2009.

[SchlKoWe2009] Schlich, B., Kowalewski, S., Wernerus, J.: Verifikation von SPS-Programmen in AWL mit Hilfe von direktem Model-Checking. Proc. AUTOMATION 2009, Baden-Baden, 2009.

[SoliFrey2009] Soliman, D., Frey, G.: Verification and Validation of Safety Applications based on PLCopen Safety Function Blocks using Timed Automata in Uppaal. Proc. 2nd IFAC Workshop on Dependable Control of Discrete Systems (DCDS), pp. 39-44, Bari, Italy, 2009.

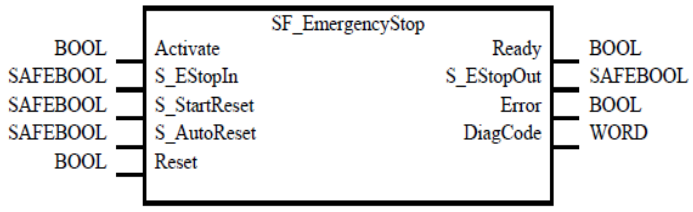
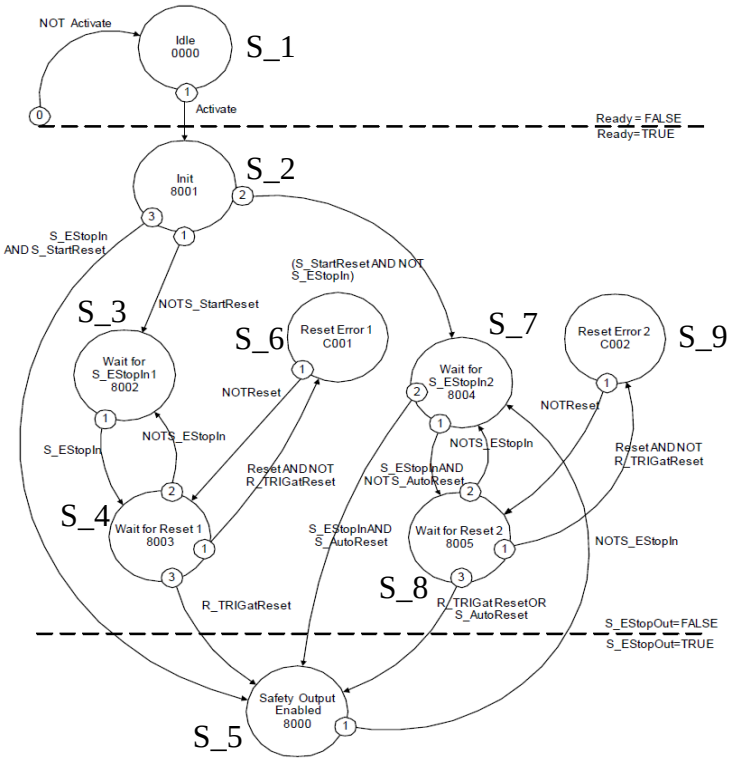
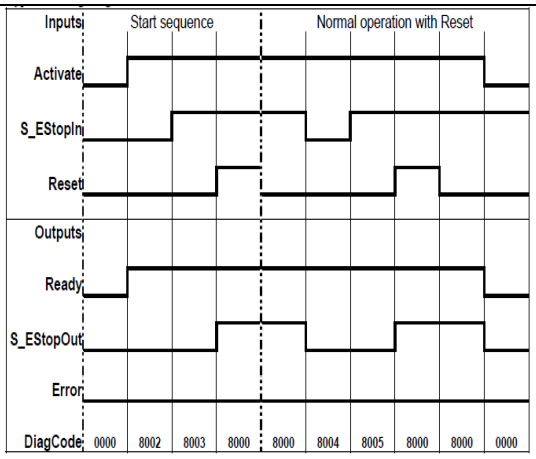
| | |
|---|---|
| 1 |  |
| 2 | The S_EStopOut enable signal is reset to FALSE as soon as the S_EStopIn input is set to FALSE. The S_EStopOut enable signal is reset to TRUE only if the S_EStopIn input is set to TRUE and a reset occurs. The enable reset depends on the defined S_StartReset, S_AutoReset, and Reset inputs.<br>If S_AutoReset = TRUE, acknowledgment is automatic.<br>If S_AutoReset = FALSE, a rising trigger at the Reset input must be used to acknowledge the enable.<br>If S_StartReset = TRUE, acknowledgment is automatic the first time the PES is started.<br>If S_StartReset = FALSE, a rising trigger at the Reset input must be used to acknowledge the enable. |
| 3 |  |
| 4 |  |

**Tabelle1: Spezifikation des Sicherheitsfunktionsbausteins SF_EmergencyStop nach PLCopen**