Statische Analysetechniken für speicherprogrammierbare Steuerungen

Sébastien Bornot Ralf Huuck Ben Lukoschus

Christian-Albrechts-Universität zu Kiel
Institut für Informatik und Praktische Mathematik
Preußerstraße 1–9, 24105 Kiel

{seb|rhu|bls}@informatik.uni-kiel.de

Zusammenfassung

In diesem Beitrag wird die Methode der statischen Analyse für die standardisierte Sprache „Anweisungsliste“ (AWL) für speicherprogrammierbare Steuerungen vorgestellt. Statische Analyse umfaßt Techniken, die Laufzeitverhalten ermitteln bzw. nachweisen, ohne den Code selbst auszuführen. Insbesondere die Technik der abstrakten Interpretation spielt hier eine wichtige Rolle. Wir werden diese Techniken mit Bezug auf AWL vorstellen und aufzeigen, welche statischen Analyseziele für AWL erreichbar sind. Ferner gehen wir darauf ein, was statische Analyse im Vergleich zu anderen Verifikationstechniken leistet und wie diese in Kombination genutzt werden können.

1 Einleitung

Speicherprogrammierbare Steuerungen (SPS) finden in der Automatisierungstechnik eine stark zunehmende Verbreitung. Während sie in der Vergangenheit meist nur zur Steuerung einfacher Abläufe eingesetzt wurden, sind SPS heutzutage verantwortlich für die Steuerung komplexer hybrider Systeme, sei es in der petrochemischen Industrie oder in Kraftwerken. Daher ist eine korrekt funktionierende Steuerungssoftware von grundsätzlicher Bedeutung.

Beim Einsatz formaler Methoden zur Überprüfung der Korrektheit treten zwei wesentliche Probleme auf. Zum einen stoßen Techniken wie das *Model-Checking* [8, 2] bei komplexen Systemen schnell an ihre Grenzen, und zum anderen sind die Ersteller der Steuerungssoftware häufig Personen, die ihre Aufgabenschwerpunkt in der praktischen Umsetzung des zu steuernden Prozesses und nicht im Einsatz formaler Methoden haben. Daher ist es neben einer Bewältigung des Komplexitätsproblems auch von Bedeutung, gut automatisierbare Methoden einzusetzen, die vom Benutzer dieser Methoden keine tiefgehenden Kenntnisse in der Verifikation von Software erfordern.

Statische Analyse [3] ist ein Weg diesen Anforderungen zu begegnen. Sie erlaubt es, über das Laufzeitverhalten von Programmen zu räsonieren, ohne diese Programme direkt auszuführen. Obwohl das Laufzeitverhalten nur approximativ bestimmt wird, kann die statische Analyse effektiv dazu beitragen, Programmeigenschaften zu überprüfen.

Im folgenden stellen wir einen Algorithmus zur statischen Analyse von SPS-Programmen vor. Nach einer Einführung in die verwendete Programmiersprache und die grundlegenden Analysetechniken beschreiben wir den Algorithmus und diskutieren abschließend seine mögliche Einbettung in andere Verifikationstechniken.

1.1 Die Programmiersprache „Anweisungsliste“

Der Standard IEC 1131-3 [5] definiert fünf Sprachen für die Programmierung von SPS-Systemen. Diese basieren auf den Programmiersprachen, die auf den verschiedenen SPS-Entwicklungssystemen zur Verfügung stehen und in verschiedenen Ländern unterschiedlich stark verbreitet sind. Eine Sprache, die in Europa stark Verwendung findet, ist die Anweisungsliste (AWL, engl. *instruction list*).

Bei AWL handelt es sich um eine assemblerähnliche Sprache. Ein AWL-Programm ist eine Liste von Anweisungen, die aus einem Operator und einem oder mehreren Operanden bestehen. Die Operatoren manipulieren ein spezielles Register, welches *Current Result* (CR) genannt wird. Dieses Register kann Variablenwerte lesen, durch Operatoren verändert und wieder in Variablen zurückgeschrieben werden. Die Auswahl der Operatoren reicht von einfachen arithmetischen (Addition, Multiplikation) über Vergleichsoperatoren bis hin zur Benutzung von Uhren und extern definierter Funktionen. Zur Steuerung des Programmflusses stehen bedingte und unbedingte Sprunganweisungen zur Verfügung.

Ein Beispiel für ein Programm in der Sprache AWL befindet sich in der linken Spalte von Abbildung 2. Der Befehl `LD` (*load*) lädt einen Wert ins CR. Die Operation `ST` (*store*) erlaubt es, den Inhalt des CR in eine Variable zu schreiben, während `ADD` eine Addition im CR ausführt und `LE` (*less or equal*) den Inhalt des CR mit einer Variable oder Konstante vergleicht und das Ergebnis, einen Booleschen Wert, zurück ins CR schreibt. Die Ausführung der bedingten Sprunganweisung `JMPC` (*jump conditional*) ist von dem Booleschen Wert des CR abhängig. Falls das CR den Booleschen Wert *wahr* enthält, wird zu der angegebenen Marke gesprungen, ansonsten wird die nächste Anweisung ausgeführt.

1.2 Statische Analyse

Statische Analyse [3] ist eine Methode zur Bestimmung von Eigenschaften eines Programmes, die in jeder Programmausführung gültig sind, sogenannte *Invarianten.* Im Gegensatz zum *Testen,* wo nur *einige* Ausführungen beobachtet werden, oder dem *Model-Checking* [8, 2], das mit Hilfe einer vollständigen Transitionsstruktur *alle* Ausführungen durchläuft, wird bei der statischen Analyse das Programm überhaupt nicht ausgeführt. Stattdessen wird eine Überapproximation des Programmverhaltens bestimmt. Diese approximative Programmanalyse ist pessimistisch in dem Sinne, daß sie in der Regel mehr Programmverhalten miteinbezieht, als das zu untersuchende Programm konkret besitzt, d. h., falls die Analyse ergibt, daß eine gewisse Eigenschaft nicht erfüllt ist, muß dieses nicht unbedingt zutreffen. Auf der anderen Seite sind Resultate, die eine gewisse Eigenschaft bestätigen, stets verläßlich.

Der entscheidende Vorteil der statischen Analyse liegt darin, daß auch für sehr große Programme, wie sie in der Industrie üblich sind, durchaus wertvolle Informationen für den Entwicklungsprozeß gewonnen werden können, ohne daß der Anwender der Analyse zusätzliche Angaben (z. B. eine Spezifikation der Programmfunktionalität) über das zu prüfende Programm machen muß. Typische Ziele von statischer Analyse sind unter anderem das Prüfen von Array-Grenzen, die Suche nach nichtterminierenden Schleifen, unbenutzten oder redundanten Programmteilen, Typinkonsistenzen, sowie die Erkennung von Laufzeitfehlern wie Überlauf oder Division durch Null.

1.3 Abstrakte Interpretation

Ein häufig benutztes Verfahren der statischen Analyse ist die abstrakte Interpretation [4]. Dieses Verfahren weist Programmvariablen *abstrakte Werte* aus einem abstrakten Wertebereich zu. Ein abstrakter Wert speichert im Gegensatz zum *konkreten* Wert einer Variablen (z. B. eine Zahl, die durch eine bestimmte Programmausführung berechnet wird) nur eine Überabschätzung einer Variablen (z. B. ein Intervall, das alle möglichen Variablenwerte enthält), die für alle Ausführungen gültig ist [3].

Der Vorteil einer solchen Überabschätzung liegt darin, den Abstraktionsalgorithmus so konstruieren zu können, daß er für jedes gegebene Programm terminiert, während bei Simulation oder Ausführung eines Programms die Termination nicht gewährleistet ist.

Ihr Nachteil liegt darin, daß wir wie oben angesprochen im Normalfall mehr Programmverhalten bestimmen als in allen konkreten Ausführungen tatsächlich vorhanden ist. Je nach Wahl der abstrakten Wertebereiche und der Algorithmen zur Bestimmung der abstrakten Werte führt dies zu einer mehr oder weniger großen Ungenauigkeit der Ergebnisse. Jedoch ist es mit einer sicheren Interpretation der abstrakten Werte in jedem Fall ausgeschlossen, daß Programmeigenschaften als gültig bestimmt werden, wenn sie es tatsächlich nicht sind. Andererseits kann die Überabschätzung dazu führen, daß Eigenschaften als ungültig angesehen werden, wenn die Eigenschaften nur durch das zusätzliche Programmverhalten verletzt werden, auf der konkreten Ebene aber erfüllt sind.

1.4 Ein kombinierter Ansatz

Abstrakte Interpretation und statische Analyse werden in unserem Ansatz, wie in Abbildung 1 zu sehen, miteinander verbunden.

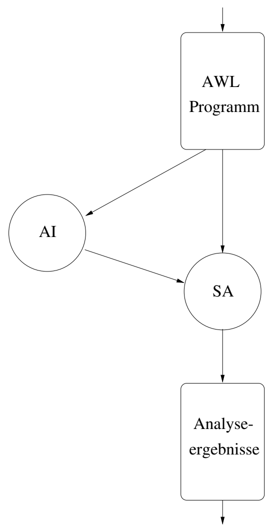

Abbildung 1: Konzeptuelle Darstellung der Analyse von AWL-Programmen.

Wir setzen ein AWL-Programm als gegeben voraus. Dieses kann zunächst statisch analysiert werden auf z. B. Typinkonsistenzen, überflüssigen oder nicht genutzten Code. Um jedoch weitere und präzisere Aussagen zu verifizieren, kann als Vorstufe eine abstrakte Interpretation für das AWL-Programm durchgeführt werden, d. h. wir ermitteln auf endlichem algorithmischen Weg für jede Programmvariable und jede Anweisung eine abstrakte Belegung dieser Variablen. Die statische Analyse erlaubt es dann, Aussagen über z. B. die Einhaltung von Arraygrenzen, das Vorhandensein von totem Code oder das Auftreten von Divisionen durch Null zu treffen.

2 Statische Analyse von AWL-Programmen

Dieser Abschnitt beschreibt die statische Analyse auf Basis abstrakter Interpretation von AWL-Programmen. Hierbei wird für ein gegebenes AWL-Programm zu jeder Anweisung eine abstrakte Variablenbelegung ermittelt. Dieses geschieht iterativ auf Grundlage eines Verfahrens aus [3]. Als abstrakten Wertebereiche für numerische Variablen wählen wir Intervalle, und für Boolesche Variablen die Menge $\{wahr, falsch, \perp\}$, wobei $\perp$ ausdrückt, daß der konkrete Wert nicht bekannt ist.

Eine besondere Rolle in unserem Verfahren kommt dem Current Result zu. Wir verbinden es mit anderen Variablen, indem wir es als einen Ausdruck darstellen, der Konstanten und Variablen enthält. Sein abstrakter Wert kann leicht berechnet werden, indem die Variablenvorkommen durch ihre abstrakte Werte ersetzt werden und der Ausdruck dann berechnet wird. Zum leichteren Verständnis und zur Veranschaulichung werden wir den Algorithmus nicht im Detail präsentieren, sondern an einem konkreten Programmbeispiel illustrieren.

In Abbildung 2 wissen wir zum Beispiel nach der Operation `MUL Y`, daß folgendes gilt: $\mathtt{CR} = \mathtt{X} \cdot \mathtt{Y}$, $\mathtt{X} \in [2,20]$ und $\mathtt{Y} \in [1,+\infty]$. Daher läßt sich das Current Result leicht zu $\mathtt{CR} \in [2,+\infty]$ berechnen. Desweiteren ist das Current Result dynamisch getypt, d. h., es kann von unterschiedlichen Typs während einer Programmausführung sein. Dieses ist der Grund dafür, daß in dem Beispielprogramm an einigen Stellen, der Wert ‘unbekannt’ für `CR` benutzt wird.

Das AWL-Programm wird iterativ an jeder Anweisung mit den abstrakten Werten (Überabschätzungen) für die Variablen und das Current Result versehen. Falls die Anweisung ein Laden (`LD`) oder Speichern (`ST`) ist, wissen wir, daß der Wert des Current Result dem Wert der geladenen bzw. gespeicherten Variable entspricht. Bei anderen Operationen erweitern wir das Current Result um eben diese.

Für bedingte Sprunganweisungen können wir den den nachfolgenden Wert des Current Result dadurch ermitteln, daß wir unterscheiden, ob der Sprung ausgeführt wurde oder nicht und entsprechend die vorkommenden Variablen einschränken. Falls der bedingte Sprung in dem Beispielprogramm ausgeführt wird, gilt $\mathtt{CR} = wahr$, und somit ist $\mathtt{X} \leq 10$. Da für den abstrakten Wert von `X` gilt, daß $\mathtt{X} \in [2,20]$ ist, folgt somit $\mathtt{X} \in [2,10]$. Falls der Ausdruck für das Current Result zu viele Operationen oder Variablen enthält, kann es sein, daß diese zusätzlichen Informationen nicht mehr sinnvoll genutzt werden können, um abstrakte Variablenwerte einzuschränken. Allerdings ist dieses selten der Fall, da AWL-Programme in der Regel nur wenige Operationen zwischen `LD`- und `ST`-Befehlen haben.

Gibt es Verzweigungen (insbesondere Rücksprünge) im Programm, so müssen die abstrakten Werte z. T. mehrmals bestimmt werden. Um dabei die Terminierung des Algorithmus zu erzwingen, kommt ein Verfahren namens *widening* zum Einsatz. Offensichtlich ist es a priori nicht möglich, zu sagen, wie oft die Schleife (zwischen `label` und `JMPC label`) im Beispielprogramm durchlaufen wird. Allerdings kann man feststellen, daß Variablenwerte erhöht werden. Durch Betrachtung des abstrakten Wertes für `Y` wissen wir, daß an dem Kontrollpunkt `label` der Wert in einem Bereich $[1,a]$ liegt, wobei a sich bei jedem Schleifendurchlauf erhöhen wird. Um nun eine Terminierung des Algorithmus zu garantieren, verwenden wir die *widening*-Technik. In dem obigen Fall durchläuft der

```
VAR
      X,Y : INT;
END_VAR
```

AWL	CR	X	Y
	CR unbekannt	$X \in [-\infty, +\infty]$	$Y \in [-\infty, +\infty]$
`LD 1`			
	$CR = 1$	$X \in [1,1]$	$Y \in [-\infty, +\infty]$
`ST X`			
	$CR = X$	$X \in [1,1]$	$Y \in [-\infty, +\infty]$
`ST Y`			
	$CR = Y$	$X \in [1,1]$	$Y \in [1,1]$
`label`			
	CR unbekannt	$X \in [1,10]$	$Y \in [1, +\infty]$
`LD X`			
	$CR = X$	$X \in [1,10]$	$Y \in [1, +\infty]$
`ADD X`			
	$CR = X + X$	$X \in [1,10]$	$Y \in [1, +\infty]$
`ST X`			
	$CR = X$	$X \in [2,20]$	$Y \in [1, +\infty]$
`MUL Y`			
	$CR = X \cdot Y$	$X \in [2,20]$	$Y \in [1, +\infty]$
`ST Y`			
	$CR = Y$	$X \in [2,20]$	$Y \in [2, +\infty]$
`LD X`			
	$CR = X$	$X \in [2,20]$	$Y \in [2, +\infty]$
`LE 10`			
	$CR = (X \leq 10)$	$X \in [2,20]$	$Y \in [2, +\infty]$
`JMPC label`			
falls Sprung ausgeführt:	$CR =$ *wahr*	$X \in [2,10]$	$Y \in [2, +\infty]$
falls Sprung nicht ausgeführt:	$CR =$ *falsch*	$X \in [11,20]$	$Y \in [2, +\infty]$
`LD Y`			
	$CR = Y$	$X \in [11,20]$	$Y \in [2, +\infty]$
`DIV X`			
	$CR = Y/X$	$X \in [11,20]$	$Y \in [2, +\infty]$

Abbildung 2: Ein AWL-Programm und die zugehörigen Ergebnisse der abstrakten Interpretation.

Algorithmus nicht beliebig oft die Schleife, sondern setzt die obere Intervallgrenze sofort fest, sobald ein Wachsen der abstrakten Werte beobachtet wird. Da für `Y` keine obere Grenze ermittelt werden kann, wird der abstrakte Wert von `Y` auf $[1, +\infty]$ gesetzt. Der Wert für `X` wird auf $[1, 10]$ gesetzt, da die obere Grenze 10 aus der vorangegangenen Anweisung und dem bedingten Sprung extrahiert werden kann.

Die so gewonnenen Annotierungen am AWL-Programm können anschließend in vielfältiger Weise genutzt werden. So kann z. B. eine Division durch Null in der Anweisung `DIV x` mit Sicherheit ausgeschlossen werden, da die abstrakte Interpretation gezeigt hat, daß `x` im Intervall $[11, 20]$ liegt. Durch Analyse des abstrakten Wertes von CR vor bedingten Sprüngen lassen sich strukturelle Eigenschaften wie Endlosschleifen oder unbenutzter Code nachweisen. Auch Prüfungen auf Typkonsistenz sind möglich.

Wir haben einen Prototypen eines Werkzeugs entwickelt, der automatisch diese abstrakte Interpretation auf AWL-Programmen ausführt, bis auf die Tatsache, daß zur Zeit noch nicht die Informationen bei bedingten Sprunganweisungen ausgenutzt werden. Obwohl wir noch nicht eine breite Palette von Algorithmen zur statischen Analyse implementiert haben, die die Informationen aus der abstrakten Interpretation ausnutzten, können wir dennoch gewisse Eigenschaften identifizieren. So lassen sich zum Beispiel bedingte Sprunganweisungen finden, die deterministisch sind, d. h. immer oder nie springen, was zu nicht erreichbarem Code oder nichtterminierenden Schleifen führen kann.

Desweiteren ermittelt das Werkzeug automatisch eventuelle Typinkonsistenzen von Variablen oder dem Current Result.

3 Diskussion

In diesem Beitrag haben wir einen Ansatz zur abstrakten Interpretation von AWL-Programmen zum Ziel der statischen Analyse vorgestellt. Der Prototyp unseres Werkzeugs kann diese abstrakte Interpretation automatisch durchführen und einige Eigenschaften (z. B. Division durch Null) ermitteln. Der Vorteil dieses Ansatzes gegenüber anderen formalen Techniken, wie zum Beispiel dem Model-Checking, liegt in der Handhabbarkeit von sehr großen Programmen. Allerdings geschieht dieses auf Kosten der Genauigkeit, d. h., es wird teilweise Verhalten überapproximiert, was einem Informationsverlust gleichkommt.

Daher wird sich zukünftige Arbeit darauf richten, diesen Informationsverlust möglichst gering zu halten, um die Mächtigkeit des Werkzeugs zu erhöhen. Das bedeutet zum einem, daß das Werkzeug um weitere Facetten der abstrakten Interpretation bzw. der statischen Analyse erweitert wird, und zum anderen, daß unterschiedliche abstrakte Wertebereiche bzw. Darstellungsformen untersucht werden müssen, um festzustellen, welche sich am besten eignen.

Weiterhin planen wir, AWL-Programme in die strukturierende SPS-Sprache *Sequential Function Charts* (SFC) einzubetten. SFCs sind hierarchische Automaten, deren Transitionen mit Bedingungen für Transitionsübergänge beschriftet sind und deren diskrete Zustände mit einer Liste von Aktionen versehen sind. Diese Aktionen können entweder selber wieder SFCs oder Programme andere Sprachen aus [5] sein, zum Beispiel auch AWL-Programme. Wir planen SFCs zu analysieren, bei denen diese Aktionen in AWL programmiert sind. Aus der Struktur der SFCs und aus der Analyse der AWL-Programme erhoffen wir uns eine feinere Approximation der abstrakten Variablen und somit genauere Analyseergebnisse.

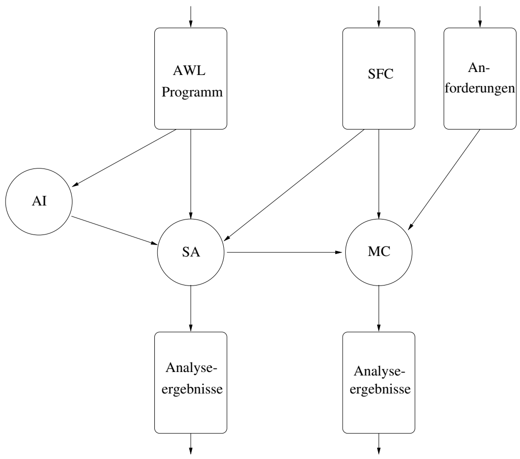

Abbildung 3: Erweiterung des Konzepts zur Analyse von SFCs mit AWL-Einbettung.

Abbildung 3 zeigt diese Erweiterung des Konzeptes aus Abbildung 1 zur Analyse von SFCs, die als Aktionen AWL-Programme enthalten. Die statische Analyse von AWL-Programmen erhält z. B. aus den Bedingungen der Transitionsübergänge in den SFCs Informationen über die Belegung der Variablen am Programmanfang. Umgekehrt kann die Analyse der SFCs von Informationen über die Variablenbelegung am Ende der AWL-Programme profitieren, da dort dann die Gültigkeit der Transitionsbedingungen wieder genauer eingegrenzt werden kann.

Die Ergebnisse der abstrakten Analyse von SFCs und AWL-Programmen können dann gemeinsam in das Model-Checking (MC) des Gesamtsystems einfließen. In [1] wurden bereits die Grundlangen für ein automatisiertes Verfahren vorgestellt, SFCs mit dem Model-Checking-Werkzeug SMV [6, 7] zu analysieren.

Durch diese Kombination unterschiedlicher formaler Methoden planen wir, SPS-Programme auch in einem industriellen Umfeld effektiv analysieren zu können.

Literatur

[1] Sébastien Bornot, Ralf Huuck, Yassine Lakhnech, and Ben Lukoschus. Verification of sequential function charts using SMV. Submitted to the special session "Formal Verification and Formal Methodologies in the Industrial Validation Flow" of PDPTA 2000, Las Vegas, USA, June 2000.

[2] E.M. Clarke, E.A. Emerson, and E. Sistla. Automatic verification of finite state concurrent systems using temporal logic specifications: A practical approach. In *10th ACM symposium of Programming Languages*. ACM Press, 1983.

[3] P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *Proceedings of the Second International Symposium on Programming, Paris, France, April 13–15, 1976*, pages 106–130, Paris, France, 1976. Dunod.

[4] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Los Angeles, California*, pages 238–252, New York, NY, 1977. ACM Press.

[5] International Electrotechnical Commission, Technical Committee No. 65. *Programmable Controllers – Programming Languages, IEC 61131-3*, second edition, November 1998. Committee draft.

[6] K.L. McMillan. *The SMV system.* Carnegie-Mellon University, February 1992. Draft manual describing SMV revision 2.2.

[7] K.L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[8] J.P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *5th International Symposium on Programming*, volume 137 of *Lecture Notes in Computer Science*, pages 337–351. Springer-Verlag, 1982.