



ΧΑΡΟΚΟΠΕΙΟ ΠΑΝΕΠΙΣΤΗΜΙΟ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΜΑΤΙΚΗΣ

Ενσωμάτωση και εφαρμογή του Μοντέλου Ελέγχου και της
Στατικής Ανάλυσης στην Ανάπτυξη Προγραμμάτων για
Προγραμματιζόμενους Λογικούς Ελεγκτές

Πτυχιακή εργασία

Τσιπλάκη Σπηλιοπούλου Χριστίνα

Τριμελής Εξεταστική Επιτροπή

Dr. Βαρλάμης Ηρακλής (Επιβλέπων)

Dr. Καμαλάκης Θωμάς

Dr. Μιχαήλ Δημήτριος

Αθήνα, 2017

Περίληψη

Η βιομηχανία εξελίσσεται σε τέτοιο βαθμό όπου η ανθρώπινη παρέμβαση για τη λειτουργία των συστημάτων, τείνει να εξαφανιστεί με το πέρασμα του χρόνου. Σαν αποτέλεσμα, έστω και το πιο μικρό λάθος σε ένα σύστημα ελέγχου μπορεί να αποβεί μοιραίο. Συνεπώς, η ανάγκη ανάπτυξης ισχυρών, ασφαλών και αξιόπιστων συστημάτων ελέγχου είναι επιτακτική για τους μηχανικούς ελέγχου. Προκειμένου να μπορούν να εγγυηθούν τα παραπάνω, τόσο το λογισμικό (software) όσο και το υλισμικό (hardware) πρέπει να αναλυθούν και να είναι βέβαιο ότι πληρούν τις απαραίτητες προϋποθέσεις.

Η πιο ευρέως διαδεδομένη συσκευή ελέγχου στην βιομηχανία ονομάζεται προγραμματιζόμενος λογικός ελεγκτής (PLC). Τα PLC χρησιμοποιούνται σε όλο το κόσμο σε εκατομμύρια βιομηχανίες και συνεπώς η εγγύηση της ασφάλειας που παρέχει ένα τέτοιο σύστημα αποτελεί μια απαιτητική πρόκληση για τους μηχανικούς. Δοκιμές και τυπικές μέθοδοι χρησιμοποιούνται προκειμένου λοιπόν να ελεγχθεί η ορθότητα ενός PLC προγράμματος.

Στόχος αυτής της πτυχιακής εργασίας είναι η βελτίωση της διασφάλισης των PLC προγραμμάτων και η μείωση των σφαλμάτων που περιέχονται στο λογισμικό τους με την ενσωμάτωση και την εφαρμογή στατικής ανάλυσης κώδικα και μίας τυπικής μεθόδου. Ταυτόχρονα, στόχος της πτυχιακής είναι μέσα από αυτό να εξαλειφθεί κάθε πολυπλοκότητα από την πλευρά του χρήστη.

Τα πειράματα και οι μεθοδολογίες που χρησιμοποιήθηκαν σε αυτή την πτυχιακή έχουν εφαρμοστεί σε PLC προγράμματα που αναπτύχθηκαν στο CERN και χρησιμοποιούνται στα πειράματα που διεξάγονται καθημερινά.

Abstract

Industry processes are evolving to the point where human interventions tend to disappear with the passage of time and as a result even a small mistake in these control systems can have catastrophic consequences. For that reason the need of developing robust, safe and reliable control systems is fundamental for control engineers. To guarantee the above both the hardware and the software have to be analysed to ensure that they fulfil the requirements.

The most popular control device in the process industry is the Programmable Logical Controller (PLC). PLCs are used all over the world for millions of industrial processes. Guaranteeing the safety of such a system is a challenging task for engineers. Testing and formal methods are used to check the correctness of a PLC program and ensure its safety.

The goal of this thesis is to improve the safety assurance of PLC programs and reduce the number of flaws in the software by integrating and applying static analysis and one formal method technique in the development process and at the same time hide any complexity from the developer.

The experiments and the methodologies used in this thesis have been applied to real-life PLC programs developed at CERN.

Η Τσιπλάκη-Σπηλιοπούλου Χριστίνα, δηλώνω υπεύθυνα ότι:

1. Είμαι η κάτοχος των πνευματικών δικαιωμάτων της πρωτότυπης αυτής εργασίας και από όσο γνωρίζω η εργασία μου δε συκοφαντεί πρόσωπα, ούτε προσβάλλει τα πνευματικά δικαιώματα τρίτων.
2. Αποδέχομαι ότι η ΒΚΠ μπορεί, χωρίς να αλλάξει το περιεχόμενο της εργασίας μου, να τη διαθέσει σε ηλεκτρονική μορφή μέσα από τη ψηφιακή Βιβλιοθήκη της, να την αντιγράψει σε οποιοδήποτε μέσο ή/και σε οποιοδήποτε μορφότυπο καθώς και να κρατά περισσότερα από ένα αντίγραφα για λόγους συντήρησης και ασφάλειας.

Περιεχόμενα

1	Εισαγωγή	7
1.1	Γενικό πλαίσιο	8
1.2	Συνεισφορά και Κίνητρα για την πτυχιακή	9
2	Υπόβαθρο	10
2.1	Εισαγωγή	10
2.2	Programmable Logic Controllers	10
2.2.1	PLC Hardware	10
2.2.2	PLC Software	11
2.2.3	UNICOS	12
2.3	Μοντέλο ελέγχου και Στατική ανάλυση	12
2.3.1	Τυπικές μέθοδοι	12
2.3.2	Μοντέλο ελέγχου	12
2.3.3	Στατική ανάλυση κώδικα	13
2.4	Σχετική εργασία	15
2.4.1	Εφαρμογή του μοντέλου ελέγχου στα προγράμματα PLC	15
2.4.2	Εφαρμογή στατικής ανάλυσης στα προγράμματα PLC	15
2.5	Παρουσίαση τεχνολογιών που χρησιμοποιήθηκαν για την εκπόνηση της πτυχιακής εργασίας	16
2.5.1	Apache Subversion	16
2.5.2	Jenkins	16
2.5.3	Spoofax	16
2.5.4	Xtext	16
3	Ενσωμάτωση του μοντέλου ελέγχου στην ανάπτυξη PLC προ- γραμμάτων	17
3.1	Εισαγωγή	17
3.2	Συνεισφορά	19
3.3	Παρατηρήσεις	21
4	Υλοποίηση μιας αφηρημένης τεχνικής	21
4.1	Εισαγωγή	21
4.2	Συνεισφορά	24
5	Στατική ανάλυση κώδικα	27
5.1	Εισαγωγή	27
5.2	Υλοποίηση	27
6	Συμπεράσματα και Μελλοντικές επεκτάσεις	29
6.1	Συμπεράσματα	29
6.2	Μελλοντικές επεκτάσεις	30
	Appendices	31
A	Introduction	32
A.1	Context	33
A.2	Contributions of the Thesis and Motivation	34
B	Background and Related Work	36
B.1	Introduction	36
B.2	Programmable Logic Controllers	36
B.2.1	PLC Hardware	36
B.2.2	PLC Software	37
B.2.3	UNICOS	39

C	Model Checking and Static Analysis	39
C.1	Formal Methods	39
C.1.1	Model checking	39
C.1.2	Static analysis	41
D	Related Work	42
D.1	Model checking applied to PLC programs	43
D.2	Static analysis applied to PLC programs	43
E	Technologies used for this thesis	43
E.1	Apache Subversion	43
E.2	Jenkins	44
E.3	Spoofax	44
E.4	Xtext	44
F	Integration of Model Checking in the development of PLC programs	45
F.1	Introduction	45
F.2	Contribution	49
F.3	Conclusions	52
G	Implementation of an Abstraction Technique	53
G.1	Introduction	53
G.2	Contribution	54
G.3	Experiments	60
G.4	Analysis and Conclusions	64
H	Static Code Analysis	65
H.1	Introduction	65
H.2	Static analysis tools evaluation	65
H.3	Contribution	66
H.3.1	Introduction	66
H.3.2	Spoofax approach	69
H.3.3	PLCverif approach	76
H.4	Analysis and conclusions	80
I	Conclusions and future work	82
I.1	Conclusions	82
I.2	Future work	82

Κατάλογος Σχημάτων

1	Πυραμίδα επιπέδων των συστημάτων ελέγχου	7
2	CERN accelerator complex [14]	9
3	Basic PLC components	11
4	SCL editor	18
5	Περίπτωση επαλήθευσης	18
6	Αναφορά επαλήθευσης	19
7	Επισκόπηση της προσέγγισης	20
8	Παράδειγμα αναφοράς της ανάλυσης της περίπτωσης επαλήθευσης που αποστέλλεται στο χρήστη με email	20
9	UNICOS πάνελ επισκόπησης των πρότζεκτ	21
10	Βήματα του αλγορίθμου variable abstraction [20]	22
11	Το παραγόμενο Variable dependency graph από το παράδειγμα του Listing 5	24
12	Ροή εργασιών για τις προσεγγίσεις του εργαλείου στατικής ανάλυσης.	28

13	Control system layers	32
14	CERN accelerator complex [14]	34
15	Basic PLC components	37
16	Schema of the cyclic scanning mode	38
17	Model checker process	40
18	Methodology overview	46
19	SCL editor	46
20	Requirement pattern	47
21	Verification case	47
22	Verification report	48
23	Example of an email report provided by Jenkins to the user after modification on the PLC source code	50
24	UNICOS overview panel	50
25	High level overview of the approach	50
26	User workflow	52
27	Steps of the variable abstraction algorithm [20]	56
28	Variable dependency graph example of Listing 5	58
29	Position calculation schema of OnOff object. [40]	61
30	Static analysis approach workflow.	67
31	User workflow for the Java approach.	75
32	Static analysis report.	76
33	SLC editor in PLCverif approach of static analysis tool.	79
34	Static analysis report produced by PLCverif.	80

Κατάλογος Πινάκων

1	Πίνακας σύγκρισης I των εργαλείων στατικής ανάλυσης για προγράμματα PLC.	27
2	Πίνακας σύγκρισης II των εργαλείων στατικής ανάλυσης για προγράμ- ματα PLC.	27
3	Σύγκριση μεταξύ των προσεγγίσεων για το εργαλείο στατικής ανάλυσης κώδικα.	29
4	Counterexample for p on AM'_1	59
5	Counterexample for p on $AM'_1 + 1$ invariant	59
6	Counterexample for p on $AM'_1 + 2$ invariants	60
7	PCO UNICOS object tested with nuXmv'IC3 algorithm	62
8	PCO UNICOS object tested with nuXmv CTL algorithm	62
9	ANALOG UNICOS object tested with nuXmv'IC3 algorithm	63
10	ANALOG UNICOS object tested with nuXmv CTL algorithm	63
11	ONOFF UNICOS object tested with nuXmv'ic3 algorithm	63
12	ONOFF UNICOS object tested with nuXmv CTL algorithm	64
13	Comparison table I of existed static analysis tools for PLC programs. .	66
14	Comparison table II of existed static analysis tools for PLC programs. .	66
15	Comparison between static analysis approaches.	81
16	PCO requirements tested with nuXmv'IC3.	97
17	PCO requirements tested with nuXmv.	98
18	Analog requirements tested with nuXmv and nuXmv'IC3.	99
19	OnOff requirements tested with nuXmv'IC3.	100
20	OnOff requirements tested with nuXmv.	101

1 Εισαγωγή

Η τεχνολογία, εξελισσόμενη με ταχείς ρυθμούς τις τελευταίες δεκαετίες, έχει επηρεάσει σχεδόν κάθε πτυχή της καθημερινής μας ζωής. Μετά τον 18ο αιώνα η βιομηχανική επανάσταση σφράγισε σε μεγάλο βαθμό την ιστορία καθώς αποτέλεσε το έδαφος για οικονομικές, πολιτικές και κοινωνικές αλλαγές. Ακόμα, η αυτοματοποίηση έφερε και συνεχίζει να φέρνει οφέλη στην ανθρώπινη κοινωνία.

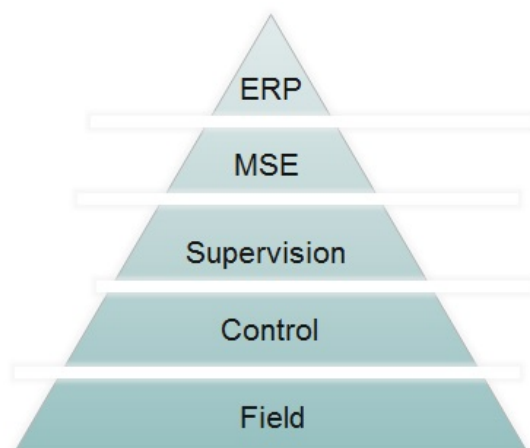
Η βιομηχανική αυτοματοποίηση εξελίσσεται και αποτελεί ένα σημαντικό παράγοντα στην ανάπτυξη του βιομηχανικού τομέα. Παίζει σημαντικό ρόλο στην βελτίωση της καθημερινής μας ζωής πράγμα που αποδεικνύεται και από το γεγονός ότι ο μέσος όρος ζωής και ο δείκτης πληθυσμού άρχισαν να αυξάνονται χάρη στην ανάπτυξη νέων τεχνολογιών και συσκευών. Πιο συγκεκριμένα, το μεσο βιοτικό επίπεδο και ο δείκτης πληθυσμού άρχισαν να αυξάνονται χάρη στην ανάπτυξη νέων τεχνολογιών και συσκευών. Εισάγοντας στην βιομηχανία την αυτοματοποίηση, η παρέμβαση του ανθρώπου μειώθηκε ή και αντικαταστάθηκε από αυτοματοποιημένες λειτουργίες όταν επρόκειτο για επικίνδυνες εργασίες.

Η βιομηχανική αυτοματοποίηση, ήρθε να 'ελευθερώσει' τους ανθρώπους από την κουραστική και πολύωρη επιτήρηση εργασιών που σχετίζονται με την αλληλεπίδραση μεταξύ αυτών και των συστημάτων ελέγχου. Η υπάρχουσα θεωρία αλλά και οι τεχνολογίες γύρω από τα συστήματα ελέγχου αποτελούν τη βάση για το σχεδιασμό συστημάτων με τις επιθυμητές συμπεριφορές τα οποία δεν χρειάζονται την επέμβαση του ανθρώπινου παράγοντα για να λειτουργήσουν.

Γενικότερα, τα συστήματα ελέγχου χωρίζονται σε τρεις βασικές κατηγορίες:

1. Συστήματα εποπτείας (supervision): σε αυτό το επίπεδο, το εργαλείο εποπτείας που συνήθως ονομάζεται SCADA παρέχει τη διεπαφή με τον χειριστή των διεργασιών.
2. Συστήματα ελέγχου (control): αυτό το επίπεδο αποτελείται από τις συσκευές ελέγχου (π.χ. PLC) οι οποίες περιέχουν την λογική για να αυτοματοποιηθεί μία διεργασία.
3. Πεδίο ελέγχου (field): αυτό το επίπεδο συντάσσεται από αισθητήρες και ενεργοποιητές οι οποίοι λαμβάνουν τις πληροφορίες από την διεργασία και εκτελούν την λογική που παρέχεται από τις συσκευές ελέγχου.

Παρ' όλα αυτά, τα σύγχρονα συστήματα ελέγχου χωρίζονται σε πέντε μέρη αντί για τρία συμπεριλαμβάνοντας επίσης το επίπεδο επιχείρησης (Enterprise Resource Planning) και το επίπεδο εργοστασίου (Manufacturing Execution System). Τα επίπεδα ενός σύγχρονου συστήματος ελέγχου παρουσιάζονται στο Σχήμα 1.



Σχήμα 1: Πυραμίδα επιπέδων των συστημάτων ελέγχου

Από το 1968, όταν ο πρώτος προγραμματιζόμενος λογικός ελεγκτής χρησιμοποιήθηκε για να ελέγχει συστήματα σε βιομηχανίες, περισσότερα κριτικά συστήματα άρχισαν να αυτοματοποιούνται. Καθώς ένα πιθανό σφάλμα σε οποιοδήποτε από αυτά τα συστήματα μπορεί να προκαλέσει ανεπανόρθωτες ζημιές τόσο στο ανθρώπινο είδος όσο

και στο περιβάλλον και την οικονομία, η ανάγκη για τη διασφάλιση της ποιότητας τους είναι επιτακτική.

Ένα από τα πολλά παραδείγματα είναι ένα λάθος στο λογισμικό που προκάλεσε το ατύχημα του Mars Climate Orbiter [58], ένα ρομποτικό διαστημόπλοιο που εκτοξεύθηκε από τη NASA το Δεκέμβριο του 1998 για να ερευνήσει το Αρειανό κλίμα. Μία μαθηματική ασυμφωνία κόστισε στη NASA ένα διαστημόπλοιο \$125-εκατομμυρίων. Ένα ακόμη παράδειγμα που κόστισε τη ζωή 6 ανθρώπων είναι το σφάλμα που εντοπίστηκε στο λογισμικό της ραδιενεργού μηχανής Therac-25 [61] η οποία χρησιμοποιούνταν για θεραπεία ασθενών με καρκίνο. Εξαιτίας ενός λάθους υπολογισμού η μηχανή χορηγούσε τεράστιες ποσότητες ραδιενέργειας στους ασθενείς.

Στόχος αυτής της πτυχιακής εργασίας είναι η ανάλυση PLC προγραμμάτων και η βελτίωση της ποιότητας τους, εστιάζοντας στη μείωση των σφαλμάτων στο λογισμικό. Τα PLC είναι οι πιο διαδεδομένες συσκευές ελέγχου που χρησιμοποιούνται στη βιομηχανία και τα τελευταία χρόνια γίνονται όλο και πιο διαδεδομένοι στα Safety Instrumented Systems, συστήματα που σχεδιάστηκαν προκειμένου να εγγυώνται την ασφάλεια και την αξιοπιστία των διεργασιών και καθορίζονται από τις προδιαγραφές των προτύπων IEC 61511 και IEC 61508.

1.1 Γενικό πλαίσιο

Καθώς τα συστήματα ελέγχου χρησιμοποιούνται σε μεγάλο βαθμό όχι μόνο στη βιομηχανία αλλά και σε τομείς όπως η αεροναυπηγική και οι πυρηνικές εγκαταστάσεις, είναι σημαντικό να αποτραπούν λάθη σαν τα προαναφερθέντα. Παρόλα αυτά μόνο λίγες εταιρίες και ερευνητικά κέντρα εμπλέκονται στο συγκεκριμένο πρόβλημα και το CERN (European Organization for Nuclear Research) είναι ένα από αυτά.

Το παρόν πόνημα, αποτελεί την πτυχιακή μου εργασία την οποία εκπόνησα στο τελευταίο έτος των σπουδών μου στο Χαροκόπειο Πανεπιστήμιο στο τμήμα Πληροφορικής και Τηλεματικής και η οποία εφαρμόστηκε στα PLC προγράμματα που χρησιμοποιούνται στο CERN. Το CERN το οποίο ιδρύθηκε το 1954, αποτελεί το μεγαλύτερο εργαστήριο σωματιδιακής φυσικής στην Ευρώπη και βρίσκεται στα Γαλλο-Ελβετικά σύνορα. Η κύρια λειτουργία του αφορά την παροχή επιταχυντών σωματιδίων και άλλων υλικοτεχνικών υποδομών που χρειάζονται για την πειραματική έρευνα στο πεδίο της φυσικής υψηλών ενεργειών.

Είναι πανάρχαια η ανάγκη των ανθρώπων να δώσουν απάντηση στο ερώτημα "Πώς δημιουργήθηκε το σύμπαν και η ζωή" και είναι διαρκής η προσπάθεια τους να ανακαλύψουν όσο γίνεται περισσότερα στοιχεία που θα τους επιτρέψουν να απαντήσουν. Ο LHC (Large Hadron Collider) (Σχήμα 2), αυτή τη στιγμή αποτελεί τον μεγαλύτερο επιταχυντή αδρονίων στον κόσμο, βρίσκεται 100 μέτρα υπό το έδαφος και έχει διάμετρο 27 χιλιομέτρων. Ο βασικός του στόχος είναι να αναπαράξει τις συνθήκες που δημιουργήθηκαν αμέσως μετά την μεγάλη έκρηξη (Big Bang). Ο επιταχυντής χρησιμοποιείται κυρίως για την έρευνα φαινομένων που θα προκύψουν από τη σύγκρουση δεσμών πρωτονίων-πρωτονίων, σε πολύ μεγάλες ενέργειες ελαφρώς μικρότερες από την ταχύτητα του φωτός. Εφτά πειράματα (CMS, ATLAS, LHCb, MoEDAL, TOTEM, LHC-forward and ALICE) είναι τοποθετημένα γύρω από τον επιταχυντή και το καθένα από αυτά μελετά τις συγκρούσεις από διαφορετική διάσταση και με διαφορετικές τεχνολογίες. Αναλύοντας τις συγκρούσεις οι επιστήμονες στοχεύουν στο να αποδείξουν ή να διαψεύσουν ποικίλες θεωρίες γύρω από το τομέα τις φυσικής αλλά και να κατανοήσουν τον κόσμο καλύτερα.

Προκειμένου να παρέχουν τις βέλτιστες συνθήκες για τους επιταχυντές, χρησιμοποιούνται βιομηχανικές διεργασίες (εξαερισμός, ψύξη, κρυογονική). Μεταξύ άλλων η πιο διαδεδομένη συσκευή ελέγχου που χρησιμοποιείται στο CERN για τις βιομηχανικές διεργασίες και τις ανάγκες των πειραμάτων είναι το PLC. Το PLC είναι μία ισχυρή συσκευή ελέγχου που διαβάζει καταχωρήσεις (inputs) από συσκευές τροφοδοσίας (π.χ. διαλογείς, βαλβίδες, αισθητήρες), τις επεξεργάζεται και μεταδίδει τα αποτελέσματα σε άλλα συστήματα.

Περισσότερα από 1000 PLC διατηρούνται και χρησιμοποιούνται στο CERN και περίπου το 1/3 αυτών συντηρείται από το γκρουπ Industrial Control and Safety (ICS) που βρίσκεται στο beams (BE) τμήμα. Πιο συγκεκριμένα, το τμήμα Process Control Systems section (PCS) αναπτύσσει, εφαρμόζει και συντηρεί τις εφαρμογές ελέγχου. Το θέμα της πτυχιακής εργασίας σχετίζεται με τις εργασίες του παραπάνω τμήματος και στοχεύει να βελτιώσει την ποιότητα του λογισμικού των PLC εισάγοντας τις τυπικές μεθόδους – και πιο συγκεκριμένα τον έλεγχο μοντέλων (model checking) – και την στατική ανάλυση κώδικα στην προγραμματιστική διαδικασία.

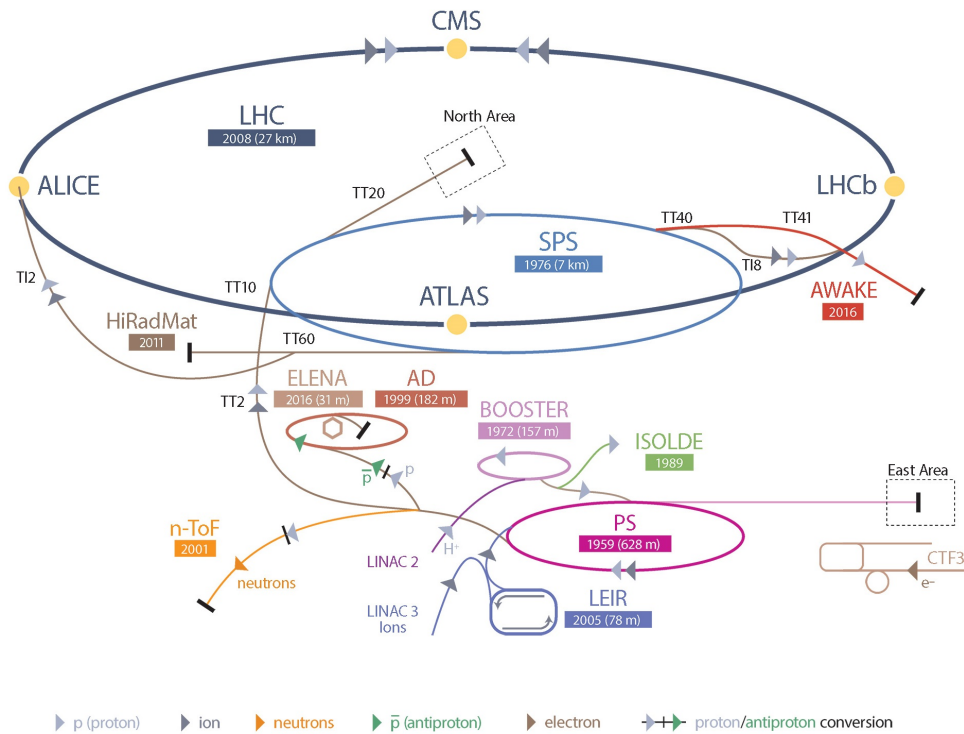


Figure 2: CERN accelerator complex [14]

1.2 Συνεισφορά και Κίνητρα για την πτυχιακή

Η απόκτηση ενός ισχυρού και αξιόπιστου PLC προγράμματος το οποίο να είναι ταυτόχρονα συμβατό με τις απαιτούμενες προδιαγραφές και να μην περιέχει σφάλματα, είναι το κύριο μέλημα των ανθρώπων που εμπλέκονται στην όλη εργασία. Ωστόσο η δημιουργία ενός τέτοιου συστήματος αποτελεί μια απαιτητική εργασία. Η πτυχιακή εργασία στοχεύει να ενσωματώσει τις παραπάνω τεχνικές στην προγραμματιστική διαδικασία PLC προγραμμάτων προκειμένου να διασφαλιστεί η συμβατότητα του προγράμματος με τις προδιαγραφές αλλά και η ποιότητα του κώδικα με τον έγκυρο εντοπισμό προβληματικού κώδικα στην πρώιμη φάση της ανάπτυξης του προγράμματος. Ταυτόχρονα επιχειρεί να εξαλείψει κάθε πολυπλοκότητα τόσο από τον προγραμματιστή όσο και από τον χρήστη.

Για να γίνει αυτό, η πτυχιακή εργασία χωρίζεται σε 3 κυρίως μέρη:

1. Ενσωμάτωση του μοντέλου ελέγχου στην διαδικασία ανάπτυξης PLC προγραμμάτων.
2. Υλοποίηση μιας αφηρημένης τεχνικής που στοχεύει στην βελτίωση της απόδοσης του μοντέλου ελέγχου στα PLC προγράμματα.
3. Την υλοποίηση ενός πρωτότυπου εργαλείου για στατική ανάλυση κώδικα PLC προγραμμάτων.

2 Υπόβαθρο

2.1 Εισαγωγή

Στο κεφάλαιο αυτό γίνεται μια επισκόπηση των τεχνικών, των μεθοδολογιών αλλά και των τεχνολογιών που χρησιμοποιήθηκαν για την ενσωμάτωση και την εφαρμογή του ελέγχου μοντέλων και της στατικής ανάλυσης κώδικα στα προγράμματα PLC. Τα PLC τις τελευταίες δεκαετίες αποτελούν καίριας σημασίας κομμάτι για την αυτοματοποίηση στα εργοστάσια, τον βιομηχανικό τομέα αλλά και το CERN.. Στο κεφάλαιο 2.2 θα περιγραφούν πιο αναλυτικά τα βασικά χαρακτηριστικά των PLC.

Από την άλλη, παρόλο που το PLC αποτελεί την πιο διαδεδομένη συσκευή ελέγχου, δεν υπάρχουν πολλές εφαρμογές μοντέλων ελέγχου και στατικής ανάλυσης στον συγκεκριμένο τομέα. Οι δύο αυτές τεχνικές θα μελετηθούν στις ενότητες 2.3 και 2.4 αντιστοίχως.

2.2 Programmable Logic Controllers

Ο πρώτος PLC παρουσιάστηκε πρώτη φορά στα τέλη του 1960 σαν έκφυμα του PC (προγραμματιζόμενος ελεγκτής) [53]. Η ανάγκη για πιο ισχυρά, εύκολα στην διαμόρφωση και αξιόπιστα συστήματα συντέλεσε στην δημιουργία των PLC. Η εταιρεία Bedford Associates ονόμασε την πρώτη συσκευή 084. Προκειμένου να συντηρήσουν, να αναπτύξουν, να υποστηρίξουν και να πουλήσουν το νέο αυτό προϊόν, δημιούργησαν μια νέα εταιρεία που ονομάστηκε *Modicon*. Ο Dick Morley, είναι ένας από τους πρώτους που δούλεψαν σε αυτό το έργο και πιθανών θεωρείται ο ‘πατέρας’ των PLC [56].

Παρά το γεγονός ότι άλλες ηλεκτρονικές συσκευές είναι πιο ισχυρές, πιο ευέλικτες και πιο εκλεπτυσμένες, τα PLC, είναι τα πιο δημοφιλή.

2.2.1 PLC Hardware

Τα PLC χρησιμοποιούν μνήμη η οποία μπορεί να προγραμματιστεί και χρησιμοποιείται για την εσωτερική αποθήκευση οδηγιών και την υλοποίηση ειδικών συναρτήσεων που αποσκοπούν στο χειρισμό ποικίλων τύπων μηχανών ή διεργασιών. Γενικά, τα PLC λαμβάνουν καταχωρήσεις τόσο από αναλογικές όσο και από ψηφιακές συσκευές (π.χ. αισθητήρες, βαλβίδες) και μεταδίδουν σήματα σε άλλα ηλεκτρικά συστήματα.

Οι μονάδες εισόδων-εξόδων (I/O) παρέχουν την σύνδεση μεταξύ του PLC και του εξοπλισμού. Το PLC, επεξεργάζεται το ίδιο πρόγραμμα συνεχώς. Ο χρόνος που χρειάζεται για να εκτελέσει το PLC ένα πλήρη κύκλο λειτουργίας ονομάζεται χρόνος κύκλου(scan cycle) και αποτελείται από τα παρακάτω βήματα:

1. Οι καταχωρήσεις σκανάρονται από τις συσκευές και στη συνέχεια εκτελούνται οι εντολές του προγράμματος το οποίο περιέχει μια σειρά απο λογικές πράξεις.
2. Μετά την εκτέλεση του προγράμματος τα αποτελέσματα καταχωρούνται στην έξοδο.

Το βασικό εξάρτημα ενός PLC είναι η κεντρική μονάδα επεξεργασίας (Central Processing Unit) η οποία είναι υπεύθυνη για την εκτέλεση των παραπάνω βημάτων. Ακόμα, τα PLC είναι συνδεδεμένα με τη μονάδα τροφοδοσίας η οποία τροφοδοτεί την CPU, τα I/O σήματα, τη μνήμη και τις συσκευές που είναι συνδεδεμένες με τα PLC. Τα παραπάνω παρουσιάζονται στο Σχήμα 3.

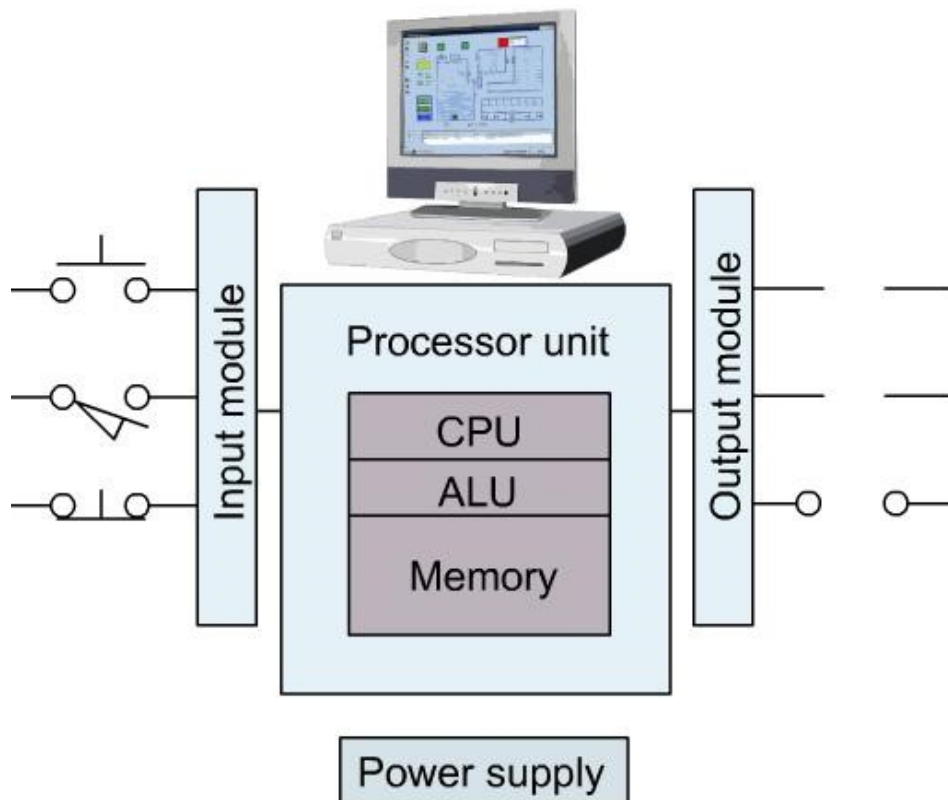


Figure 3: Basic PLC components

2.2.2 PLC Software

Η πτυχιακή εργασία επικεντρώνεται στα PLC που κατασκευάζονται από τη Siemens, καθώς είναι αυτά που χρησιμοποιούνται περισσότερο στα πειράματα στο CERN. Ωστόσο, οι διαφορές μεταξύ των PLC που παρέχονται από τη Siemens και των PLC που παρέχονται από άλλους κατασκευαστές είναι ελάχιστες καθώς όλα τα PLC βασίζονται στο πρότυπο IEC 61131.

Πιο συγκεκριμένα το πρότυπο IEC 61131-3 καθορίζει τις διάφορες συμβολικές γλώσσες ή τα διαγράμματα που χρησιμοποιούνται για τον προγραμματισμό των PLC. Όλες οι PLC γλώσσες προγραμματισμού ακόμα και αν προέρχονται από διαφορετικούς προμηθευτές πρέπει να προσαρμόζονται στο παραπάνω πρότυπο και να ακολουθούν τις συστάσεις και τις προδιαγραφές που αυτό παρέχει. Πέντε γλώσσες προγραμματισμού ορίζονται από το πρότυπο IEC 61131-3:

1. ST ή Structured Text: είναι μια γλώσσα προγραμματισμού συντακτικά παρόμοια με την Pascal.
2. SFC ή Sequential Function Chart: είναι μία διαδοχική λειτουργική γλώσσα.
3. IL ή Instruction List: είναι μια χαμηλού επιπέδου γλώσσα προγραμματισμού παρόμοια με την Assembly.
4. FBD ή Function Block Diagram: είναι μία γλώσσα λογικών γράφων, είναι γραφική σαν τη Ladder, αλλά χρησιμοποιεί «κουτιά», όπου κάθε κουτί αντιπροσωπεύει κάποια επιμέρους λειτουργία και τελικά το αντίστοιχο λογικό κύκλωμα.
5. The Ladder Diagram ή LD :με τη χρήση γραφικών εργαλείων δομείται ένα λογικό πρόγραμμα, ικανό να ακολουθήσει την λογική συνδεσμολογία ενός κλασικού αυτοματισμού.

Για τον προγραμματισμό ενός Siemens PLC, η Siemens ανέπτυξε τις δικές της γλώσσες προγραμματισμού βασιζόμενη στις παραπάνω πέντε. Για την πτυχιακή εργασία χρησιμοποιήθηκε μόνο η γλώσσα προγραμματισμού SCL (Structured Control Language) η οποία είναι ισοδύναμη με την ST.

Η SCL παρέχει πέντε διαφορετικά είδη blocks (OBs, FBs, FCs, DBs, UDTs) και το κάθε ένα χρησιμοποιείται για την κατάλληλη λειτουργία του συνολικού προγράμματος.

1. OB ή Organization Blocks: αποτελεί τη διασύνδεση του λειτουργικού συστήματος της CPU και του προγράμματος του χρήστη.
2. FB ή Function Blocks: σε κάθε FB αποδίδεται ένα block δεδομένων. Καθώς λοιπόν τα FB περιέχουν μνήμη οι τιμές των μεταβλητών διατηρούνται μετά την εκτέλεση τους.
3. FC ή Functions : σε αντίθεση με τα FB τα FC δεν περιέχουν μνήμη, περιέχουν όμως εκτελέσιμο κώδικα. Ωστόσο, στις παραμέτρους που ορίζονται σε αυτά, αποδίδονται πραγματικές τιμές μόνο όταν καλούνται από το πρόγραμμα.
4. DB ή Data Blocks: χρησιμοποιούνται για την αποθήκευση των δεδομένων του χρήστη.
5. UDT ή User-defined data types: αποτελούν δομημένα δεδομένα που ορίζονται από το χρήστη και χρησιμοποιούνται σαν να ήταν blocks.

2.2.3 UNICOS

Το UNICOS (UNified Industrial COntrol System) πρόκειται για ένα framework που προγραμματίστηκε στο CERN. Το framework αυτό παράγει PLC κώδικα για συστήματα ελέγχου τα οποία βασίζονται στα PLC, γραμμένο στην γλώσσα SCL της Siemens. Για τις ανάγκες των πειραμάτων που διεξάχθηκαν σε αυτή την πτυχιακή χρησιμοποιήθηκαν PLC προγράμματα από την βιβλιοθήκη του framework.

2.3 Μοντέλο ελέγχου και Στατική ανάλυση

2.3.1 Τυπικές μέθοδοι

Όπως αναφέρθηκε προηγουμένως, τα σφάλματα στα συστήματα ελέγχου μπορούν να βάλουν σε κίνδυνο την ανθρώπινη ζωή αλλά και το περιβάλλον και την οικονομία. Συνεπώς, συνήθως χρειάζεται πιο πολύς χρόνος για την επαλήθευση του συστήματος παρά για την κατασκευή του software ή του hardware του. Με τη χρήση των τυπικών μεθόδων η ασφάλεια ενός τέτοιου συστήματος μπορεί να διασφαλιστεί έτσι ώστε να παρθούν ευκολότερα οι αποφάσεις για τον σχεδιασμό του. Δίνονται διάφοροι ορισμοί για την περιγραφή των τυπικών μεθόδων και παρακάτω είναι ένας από αυτούς:

“Τυπικές μέθοδοι είναι μαθηματικές προσεγγίσεις για το λογισμικό και την ανάπτυξη του συστήματος που υποστηρίζει την αυστηρή προδιαγραφή, το σχεδιασμό και τον έλεγχο των συστημάτων πληροφορικής.”[18]

2.3.2 Μοντέλο ελέγχου

Το 1980 επινοήθηκε μια εναλλακτική τεχνική επαλήθευσης από τους Clarke, Emerson, Quielle και Sifakis, που ονομάστηκε χρονική λογική έλεγχου μοντέλων (temporal logic). Σε αυτή την προσέγγιση οι αλγόριθμοι και τα πρωτόκολλα μοντελοποιούνται ως συστήματα μετάβασης καταστάσεων.

“ Το μοντέλο ελέγχου είναι μια μέθοδος για την επίσημη επαλήθευση παράλληλων συστημάτων πεπερασμένων καταστάσεων. Οι προδιαγραφές για το σύστημα εκφράζονται ως χρονική λογική σε μαθηματικές φόρμουλες, και αποτελεσματικο συμβολικοί αλγόριθμοι χρησιμοποιούνται για να διασχίσουν το μοντέλο που ορίζεται από το σύστημα προκειμένου να ελέγξουν αν η προδιαγραφή ισχύει ή όχι.” [13]

Ο «έλεγχος μοντέλων» περιλαμβάνει τρία στάδια:

1. Formalization των προδιαγραφών που πρόκειται να ελεγχθούν.
2. Μοντελοποίηση του συστήματος.
3. Εκτέλεση του αλγορίθμου του μοντέλου ελέγχου.

Formalization των προδιαγραφών που πρόκειται να ελεγχθούν

Πριν την επαλήθευση, είναι απαραίτητο να καθοριστούν οι προδιαγραφές που το σχέδιο πρέπει να ικανοποιεί. Ο καθορισμός αυτός δίνεται συνήθως σε κάποια μορφή λογικού φορμαλισμού με τη χρήση χρονικής λογικής (temporal logic), χάρη στην οποία γνωρίζουμε τη συμπεριφορά του συστήματος σε συνάρτηση με το χρόνο [60].

Μοντελοποίηση του συστήματος

Αφού οι προδιαγραφές του σχεδίου έχουν εκφραστεί σε μορφή χρονικής λογικής, το επόμενο στάδιο είναι η μοντελοποίηση του συστήματος. Το μοντέλο ενός συστήματος, αντιπροσωπεύει και περιγράφει τις συμπεριφορές του αρχικού συστήματος συνήθως με ακριβές τρόπο.

Εκτέλεση του αλγορίθμου του μοντέλου ελέγχου

Το μοντέλο ελέγχου προκειμένου να εκτελέσει τον αλγόριθμο επαλήθευσης χρειάζεται σαν εισόδους από το χρήστη: το μοντέλο του συστήματος και τις προδιαγραφές εκφρασμένες σε λογική φόρμουλα. Στην συνέχεια, το μοντέλο ελέγχου θα εκτελέσει τον αλγόριθμο και θα αναλύσει τα αποτελέσματα. Μετά την ανάλυση θα επαληθεύσει αν η προδιαγραφή ικανοποιείται από το μοντέλο του συστήματος ή όχι. Στην τελευταία περίπτωση, ο ελεγκτής μοντέλων παρέχει στο χρήστη ένα παράδειγμα counterexample όπου μπορεί να δει που ακριβώς βρίσκεται το λάθος και να το διορθώσει.

Πλεονεκτήματα και μειονεκτήματα του αλγορίθμου του ελεγκτή μοντέλων

Συνοψίζοντας, ο έλεγχος μοντέλων (model checking) παρουσιάζει σημαντικά πλεονεκτήματα:

1. Πρόκειται για μια διαδικασία η οποία είναι αυτόματη και όπου δεν απαιτούνται πολλές ενέργειες από το χρήστη.
2. Αν η προδιαγραφή δεν επαληθευτεί, ο χρήστης έχει τη δυνατότητα να δει που βρίσκεται το λάθος.
3. Προκειμένου ο ελεγκτής μοντέλου να εγγυηθεί για τη συμβατότητα του μοντέλου και της προδιαγραφής, εξερευνεί όλους τους δυνατούς συνδυασμούς των μεταβλητών που εμπλέκονται στην προδιαγραφή.
4. Με τη χρήση της χρονικής λογικής, μπορούν να εκφραστούν πολλές από τις ιδιότητες που χρειάζονται.

Τα κύρια μειονεκτήματα του μοντέλου ελέγχου είναι:

1. Η έκρηξη καταστάσεων (στατε εξπλοσιον), που μπορεί να συμβεί αν το σύστημα που επαληθεύεται αποτελείται από πολλά στοιχεία που κάνουν παράλληλες μετατροπές. Σε αυτή την περίπτωση ο ελεγκτής μοντέλου μπορεί να μην μπορέσει να επαληθεύσει το αποτέλεσμα.
2. Δυσκολία στη χρήση χρονικής λογικής για τις προδιαγραφές.
3. Περιπλοκότητα στη δημιουργία μοντέλων του αρχικού συστήματος.

Μερικά από τα πιο γνωστά εργαλεία επαλήθευσης είναι τα: UPPAAL, BIP, SPIN, KRONOS και το NuSMV το οποίο θα χρησιμοποιηθεί στην πτυχιακή εργασία.

2.3.3 Στατική ανάλυση κώδικα

Στατική ανάλυση κώδικα ή στατική ανάλυση προγράμματος, είναι η ανάλυση ενός προγράμματος που γίνεται χωρίς να εκτελεστεί το πρόγραμμα. Ο όρος συνήθως αναφέρεται στην ανάλυση που γίνεται με κάποιο αυτόματο εργαλείο, και η ανάλυση προγράμματος είναι αντίστοιχη με την κατανόηση προγράμματος που γίνεται από τον άνθρωπο. Το βασικό πλεονέκτημα της στατικής ανάλυσης είναι η ανίχνευση σφαλμάτων σε πρώιμο προγραμματιστικό στάδιο με σκοπό την βελτίωση της ποιότητας του κώδικα. Ο στόχος αυτής της τεχνικής είναι ο εντοπισμός λαθών στον κώδικα ακόμα και αν αυτά δεν συμβάλουν στην αποτυχία του συστήματος. Καθώς συχνά είναι δύσκολο να ελεγχθεί ένα

ολόκληρο πρόβλημα εξαιτίας του μεγέθους του, τα εργαλεία στατικής ανάλυσης παρέχουν την δυνατότητα ελέγχου των προγραμμάτων κατά τη διάρκεια υλοποίησης τους αλλά και μετά από αυτήν. Παρόλα αυτά, τα παραπάνω δεν σημαίνουν ότι η στατική ανάλυση από μόνη της είναι επαρκής για την διασφάλιση της ποιότητας του κώδικα. Η δυναμική ανάλυση του προγράμματος δεν πρέπει να αγνοείται καθώς αποτελεί και αυτή σημαντικό κομμάτι στην προγραμματιστική διαδικασία.

Στις μέρες μας, τα εργαλεία στατικής ανάλυσης χρησιμοποιούνται ευρέως και πολλά εργαλεία είναι διαθέσιμα για τις καθιερωμένες γλώσσες προγραμματισμού (Java, Python, C). Προκειμένου να αναλυθεί το λογισμικό ενός προγράμματος υπάρχουν διάφορες τεχνικές και δομές υλοποίησης της στατικής ανάλυσης όπως για παράδειγμα: η ανάλυση ροής δεδομένων, η αφηρημένη διερμηνεία, τα διαγράμματα ελέγχου ροής. Η συγκεκριμένη εργασία επικεντρώνεται σε μια τεχνική που ονομάζεται Rule-based AST ανάλυση των προγραμμάτων PLC.

Rule-based AST

Σε αυτήν την τεχνική αναλύεται με βάση κάποιους προκαθορισμένους κανόνες, το δένδρο αφηρημένης σύνταξης ή αλλιώς AST ενός προγράμματος.

Οι κανόνες αυτοί υλοποιούνται πάνω στο AST και συνήθως μπορούν να εντοπίσουν στο κώδικα σφάλματα όπως τα παρακάτω: [47]:

1. Ονοματικές συμβάσεις: σύνολο κανόνων που ορίζουν το σύνολο χαρακτήρων που θα χρησιμοποιηθούν για τα αναγνωριστικά προκειμένου να βελτιωθεί η αναγνωσιμότητα και να διευκολυνθεί η συντήρηση του προγράμματος. Ένα παράδειγμα μπορεί να θεωρηθεί το ότι όλες οι μεταβλητές λογικού τύπου πρέπει να ξεκινάνε με το χαρακτήρα B.
2. Code smells: πρόκειται για μοτίβα κώδικα τα οποία συχνά προκαλούν προβλήματα. Μπορεί να μην δημιουργούν σοβαρά προβλήματα στον κώδικα αλλά καλό είναι να αποφεύγονται προκειμένου να διευκολύνεται η συντήρηση του προγράμματος. Μερικά παραδείγματα δίνονται παρακάτω:
 - Δυπλότυπος κώδικας.
 - Κώδικας ο οποίος δεν χρησιμοποιείται ή είναι απρόσιτος.
 - Μέθοδοι που περιέχουν πολλές γραμμές κώδικα.
 - Μεγάλες λίστες παραμέτρων.
 - Μια μεταβλητή βρόχου που διαβάζεται μετά το βρόχο.
3. Εξάρτηση δυναμικών προτάσεων: Υπάρχουν καθορισμένοι περιορισμοί που αφορούν την σειρά εκτέλεσης ορισμένων συσχετισμένων συναρτήσεων. Για παράδειγμα, η εντολή open ενός φακέλου πρέπει να προηγείται της εντολής close.

Πλεονεκτήματα και μειονεκτήματα

Η στατική ανάλυση, όπως και κάθε τεχνολογία, έχει πλεονεκτήματα και μειονεκτήματα [28]. Πλεονεκτήματα:

1. Ανάλυση του κώδικα χωρίς την εκτέλεση του.
2. Διαβεβαιώνει ότι μία σειρά από ήδη προκαθορισμένους κανόνες τηρείται από το λογισμικό του προγράμματος.
3. Συμβάλει στην πρώιμη ανίχνευση προβλημάτων στον κώδικα.
4. Συμβάλει στην διατήρηση και στην συντήρηση του προγράμματος.

Μειονεκτήματα:

1. Παράγουν πολλά ψευδώς θετικά ή αλλιώς ψευδείς προειδοποιήσεις που στη πραγματικότητα δεν αποτελούν πρόβλημα.
2. Το εργαλείο δεν θα ανιχνεύσει τα προβλήματα που σχετίζονται με τη ρύθμιση παραμέτρων καθώς δεν αναπαρίσταται στον κώδικα του προγράμματος.
3. Το εργαλείο δε μπορεί να ελέγξει την ορθότητα της συμπεριφοράς του προγράμματος.

2.4 Σχετική εργασία

Σε αυτό το κεφάλαιο παρουσιάζεται η σχετική έρευνα και οι εργασίες οι οποίες αφορούν τον ελεγκτή μοντέλου και την στατική ανάλυση στον τομέα των PLC. Περίπου 20 χρόνια πριν βιομηχανίες με κριτικά συστήματα ξεκίνησαν να χρησιμοποιούν τυπικές μεθόδους, ενώ στις μέρες μας όλο και περισσότερες τις εφαρμόζουν για την επαλήθευση των συστημάτων τους. Παρά το γεγονός ότι ανάμεσα στις βιομηχανίες υπάρχουν διαφορετικοί στόχοι και αντιπροσωπεύονται διαφορετικές ιδέες και εργασίες, η αλγοριθμική επαλήθευση με τεχνικές όπως το μοντέλο ελέγχου και η στατική ανάλυση αποτελούν κοινή πρακτική για όλες.

Ένα παράδειγμα που αποδεικνύει την εφαρμοσιμότητα των τυπικών μεθόδων στα συστήματα της καθημερινότητας είναι το σύστημα ελέγχου της εναέριας κυκλοφορίας στο Ηνωμένο Βασίλειο. Προκειμένου να διαχειριστούν την αυξανόμενη κίνηση αναβάθμισαν το σύστημα διαχείρισης της εναέριας κυκλοφορίας αναπτύσσοντας το CCF (Central Control Function) το οποίο μπορεί να διαχειριστεί από πολλά συστήματα. Ένα από αυτά είναι το CDIS (CCF Display Information System) για το οποίο χρησιμοποιήθηκαν τυπικές μέθοδοι για να σχεδιαστεί και να επαληθευτεί [46].

2.4.1 Εφαρμογή του μοντέλου ελέγχου στα προγράμματα PLC

Το μοντέλο ελέγχου αποτελεί την πιο γνωστή αλγοριθμική τεχνική τυπικής επαλήθευσης και παρόλο που υπάρχουν αρκετές προσεγγίσεις για την εφαρμοσιμότητα του στα PLC προγράμματα δεν χρησιμοποιείται ακόμα στην βιομηχανία. Πολλές έρευνες όπου το μοντέλο ελέγχου ήταν η επιλεγόμενη μέθοδος για την επαλήθευση PLC προγραμμάτων βρίσκονται στην βιβλιογραφία και μπορούν να χωρισθούν σε τρεις ομάδες:

1. Στόχευση στη γλώσσα προγραμματισμού των PLC προγραμμάτων: οι περισσότερες από τις τεχνικές του μοντέλου ελέγχου στοχεύουν σε PLC προγράμματα γραμμένα στην γλώσσα IL [9, 33, 51, 52, 43] και SFC [4, 27]. Μερικοί ερευνητές εφάρμοσαν το μοντέλο ελέγχου και σε προγράμματα γραμμένα σε FDB [3, 64] και σε Ladder Diagram (LD)[36]. Τέλος, υπάρχει μία προσέγγιση ενός εργαλείου το οποίο επαληθεύει προγράμματα γραμμένα σε όλες τις γλώσσες που ορίζονται από το πρότυπο IEC 61131-3[48, 25].
2. Στόχευση στις προδιαγραφές: Παραδόξως, δεν υπάρχουν πολλές προσεγγίσεις που να στοχεύουν στην ανάπτυξη ιδιοτήτων για την επαλήθευση του μοντέλου του δεδομένου μοντέλου. Μια προτεινόμενη προσέγγιση είναι η δημιουργία χρονικής λογικής έχοντας ως βάση τεχνικές UML [22, 49]. Μια άλλη προσέγγιση [8] έχει σαν βάση προκαθορισμένα μοτίβα που εκφράζουν τις προδιαγραφές και τις μεταφράζουν αυτόματα σε μία μορφή λογικού φορμαλισμού.
3. Στόχευση στους περιορισμούς της μοντελοποίησης και των προδιαγραφών: μερικοί ερευνητές εφάρμοσαν το μοντέλο ελέγχου σε μικρού μεγέθους προγράμματα και χωρίς πρώτα να απλουστεύσουν το μοντέλο [35, 9, 42, 44, 50, 54, 7, 24, 31], ενώ σε άλλες περιπτώσεις, υπήρχε περιορισμός ως προς την ιδιότητα όταν εφαρμοζόταν αφαίρεση (abstraction) στο μοντέλο [31][7]. Επιπλέον, σε πολλές από τις έρευνες, το πρόγραμμα δεν μοντελοποιούνταν αυτόματα σε τυπικό μοντέλο [3, 4, 7, 42, 63, 54].

2.4.2 Εφαρμογή στατικής ανάλυσης στα προγράμματα PLC

Αρκετές έρευνες έχουν διεξαχθεί από μικρές και μεγάλες εταιρίες προκειμένου να αποτιμηθεί η σημαντικότητα της στατικής ανάλυσης στις εφαρμογές λογισμικού. Τα αποτελέσματα αυτών των ερευνών δείχνουν ότι αποτελεί μία πολύτιμη συνεισφορά στην προγραμματιστική κοινότητα [37][2][65][23]. Στον τομέα των PLC, τα εργαλεία στατικής ανάλυσης σπανίζουν παρόλο που σε άλλους τομείς χρησιμοποιούνται ευρέως. Ωστόσο καλές προσπάθειες δημιουργίας εργαλείων έχουν γίνει τόσο από πανεπιστήμια [45] όσο και από ερευνητικά κέντρα [26] τα εργαλεία όμως δεν είναι προς το παρόν διαθέσιμα. Μερικές εξαιρέσεις αποτελούν τα εργαλεία: PLC Checker από την Itrix Automation

Company, Codesys Static Analysis από την CoDeSyS και το Arcade.PLC το οποίο είναι ένα ακαδημαϊκό εργαλείο που αναπτύχθηκε στο RWTH Aachen University[6]. Όλα αυτά τα εργαλεία θα αναληθούν με περισσότερες λεπτομέρειες στο κεφάλαιο 4.

2.5 Παρουσίαση τεχνολογιών που χρησιμοποιήθηκαν για την εκπόνηση της πτυχιακής εργασίας

2.5.1 Apache Subversion

Apache Subversion ή αλλιώς SVN πρόκειται για ένα σύστημα ελέγχου που διαχειρίζεται τα αρχεία, τους καταλόγους και τις αλλαγές που γίνονται σε αυτά με την πάροδο του χρόνου. Χρησιμοποιείται στο CERN για τις εργασίες γύρω από τον τομέα των PLC.

2.5.2 Jenkins

Το Jenkins είναι ένας server λογισμικού ανοιχτού κώδικα. Υποστηρίζει συστήματα ελέγχου όπως το SVN προκειμένου να εξασφαλιστεί ότι μια εργασία μπορεί να εκχωρηθεί ανά πάσα στιγμή. Με συνεχή ενσωμάτωση, το Jenkins συμβάλει στην αυτοματοποίηση της προγραμματιστικής διαδικασίας.

2.5.3 Spoofax

Το Spoofax είναι μία πλατφόρμα για την αποτελεσματική ανάπτυξη DSL γλωσσών βασισμένη στο Eclipse. Περιλαμβάνει εργαλεία και υψηλού επιπέδου μετα-γλώσσες για τον καθορισμό του συντακτικού, των τύπων, τους δεσμούς ονομάτων και τους μετασχηματισμούς των δένδρων.

2.5.4 Xtext

Το Xtext είναι ένα ανοιχτού λογισμικού πλαίσιο για την ανάπτυξη DSL και μη γλωσσών προγραμματισμού. Σε αντίθεση με άλλες γεννήτριες, το Xtext παρέχει πλήρη υποδομή καθώς παρέχει επίσης μοντέλα κλάσεων για τα δένδρα αφηρημένης σύνταξης.

3 Ενσωμάτωση του μοντέλου ελέγχου στην ανάπτυξη PLC προγραμμάτων

3.1 Εισαγωγή

Περίπου τέσσερα χρόνια πριν ένα πρότζεκτ ξεκίνησε στο CERN με σκοπό να βελτιώσει την ποιότητα των PLC προγραμμάτων και να μειώσει τον αριθμό των σφαλμάτων που περιέχονται στον κώδικά τους. Ο κύριος στόχος του πρότζεκτ αυτού, είναι η διασφάλιση του λογισμικού των προγραμμάτων που χρησιμοποιούνται στα συστήματα ελέγχου με την χρήση τυπικών μεθόδων. Οι τυπικές μέθοδοι, όπως προαναφέρθηκε, βασίζονται σε μαθηματικές τεχνικές για την περιγραφή των προδιαγραφών, την ανάπτυξη και την επαλήθευση του λογισμικού και του υλισμικού των συστημάτων.

Στο κεφάλαιο αυτό παρουσιάζεται το πρώτο μέρος της πτυχιακής εργασίας το οποίο αποτελεί συνεισφορά στο πρότζεκτ που περιγράφηκε παραπάνω. Σε αυτό το κεφάλαιο, περιγράφεται η ενσωμάτωση του μοντέλου ελέγχου στην προγραμματιστική διαδικασία των μηχανικών ελέγχου. Παράλληλα, το κεφάλαιο απαρτίζεται από την περιγραφή του προβλήματος, την μεθοδολογία που ακολουθήθηκε, την συνεισφορά και τα συμπεράσματα που εξήχθησαν.

Μεταξύ πολλών τυπικών μεθόδων, το μοντέλο ελέγχου είναι το πιο κατάλληλο για την περίπτωση μας. Αυτό γιατί όχι μόνο δίνει τη δυνατότητα αυτόματης επαλήθευσης του PLC προγράμματος αλλά και γιατί παρέχει τη δυνατότητα να απαλειφθεί κάθε πολυπλοκότητα από το χρήστη. Ωστόσο, δεν υπάρχουν πολλά εργαλεία που να στοχεύουν σε PLC προγράμματα γραμμένα σε γλώσσα SCL. Στο CERN, αναπτύχθηκε ένα εργαλείο που ονομάζεται PLCverif [16] προκειμένου να εφαρμόσει το μοντέλο ελέγχου και να επαληθεύσει τα PLC προγράμματα στη γλώσσα SCL από την Siemens η οποία αντιστοιχεί στην γλώσσα ST που παρέχεται από το πρότυπο IEC 61131. Περισσότερες πληροφορίες για το εργαλείο PLCverif παρουσιάζονται στο κεφάλαιο Φ.1 της αγγλικής εκδόχης της πτυχιακής η οποία βρίσκεται στο Appendix.

Χρησιμοποιώντας το PLCverif, ο χρήστης, έχει τη δυνατότητα να εισάγει στο εργαλείο ήδη υπάρχοντα PLC προγράμματα, να τα επεξεργαστεί ή και να δημιουργήσει καινούρια μέσα από το πρόγραμμα επεξεργασίας που παρέχεται από το εργαλείο (Σχήμα 4). Στη συνέχεια, μπορεί να δημιουργήσει μια ‘περίπτωση επαλήθευσης’ (verification case) όπως ονομάζεται, η οποία θα περιλαμβάνει την συνθήκη την οποία θέλει να επαληθεύσει (Σχήμα 5). Αφού εκτελέσει το εργαλείο, το μοντέλο ελέγχου θα παράξει μία αναφορά (Σχήμα 6) η οποία θα περιέχει το αποτέλεσμα της επαλήθευσης και στοιχεία σχετικά με το χρόνο εκτέλεσης, τον ελεγκτή μοντέλου που χρησιμοποιήθηκε, την συνθήκη που ελέγχθηκε και άλλα. Αν το αποτέλεσμα είναι αρνητικό, αν δηλαδή το σύστημα και η προδιαγραφή δεν συμπίπτουν, τότε η αναφορά θα περιέχει και ένα παράδειγμα με τις μεταβλητές που η συμπεριφορά τους δεν συμβαδίζει με την προδιαγραφή που έδωσε ο χρήστης και πιθανόν να είναι υπαίτιες για την ασυμβατότητα της.

Επιπλέον, ο χρήστης έχει τη δυνατότητα να διαμορφώσει το χρόνο που θα ξοδέψει το εργαλείο για την κάθε περίπτωση επαλήθευσης. Αυτό συμβαίνει προκειμένου να αποφεύγονται οι πολύωρες επαληθεύσεις και σε τέτοιες περιπτώσεις να ακολουθούνται διαφορετικές στρατηγικές επαλήθευσης.

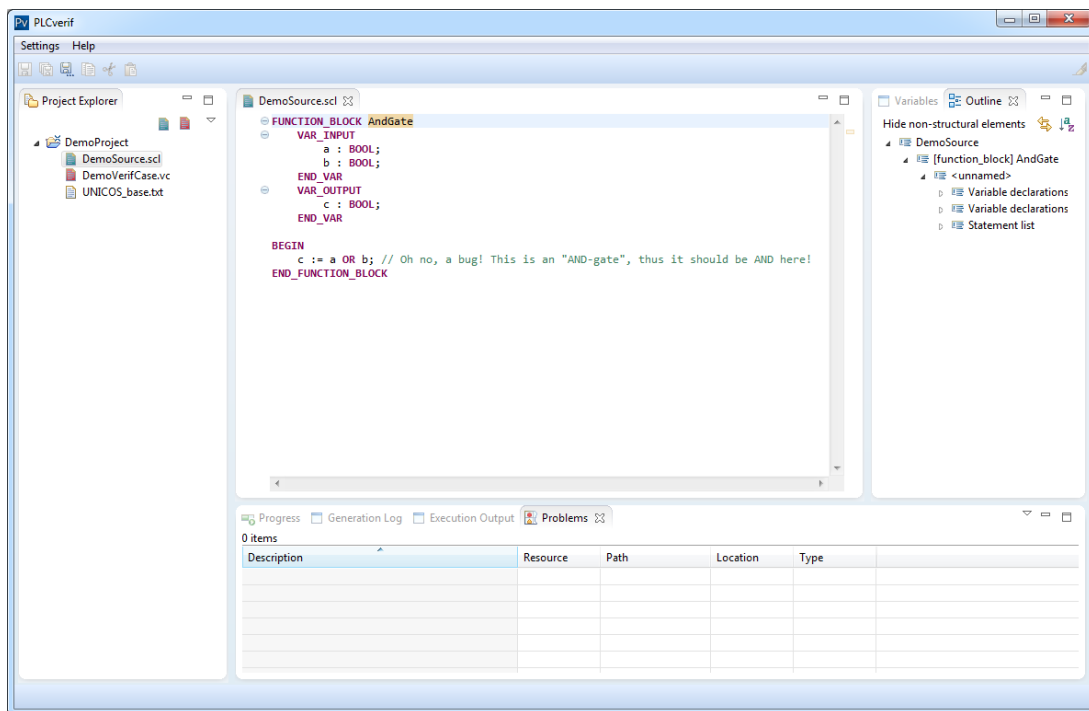
Μέχρι σήμερα κάθε φορά που ο προγραμματιστής ήθελε να τροποποιήσει ένα PLC κώδικα, όλες οι περιπτώσεις επαλήθευσης έπρεπε να ελεγχθούν μία προς μία. Επιπλέον, όταν πάνω από μία περιπτώσεις επαλήθευσης έπρεπε να δημιουργηθούν και να ελεγχθούν, η διαδικασία δεν ήταν αυτοματοποιημένη. Ένα άλλο πρόβλημα προέρχεται από το γεγονός ότι το μοντέλο ελέγχου δεν γνωρίζει πόσο χρόνο χρειάζεται μέχρι να παραχθεί το αποτέλεσμα της επαλήθευσης και πολλές φορές η διαδικασία μπορεί να διαρκέσει μέρες. Ο λόγος είναι ότι συνήθως τα μοντέλα που παράγονται από τα PLC προγράμματα αντιμετωπίζουν συχνά το πρόβλημα της έκρηξης των καταστάσεων.

Το πρώτο μέρος της πτυχιακής, το οποίο εισάγεται παρακάτω, αποτελεί συνεισφορά στο γενικό πρότζεκτ που περιγράφηκε παραπάνω. προσπαθεί να ξεπεράσει τα παραπάνω προβλήματα. Για το κομμάτι αυτό, εφαρμόστηκαν τρεις τεχνολογίες: PLCverif,

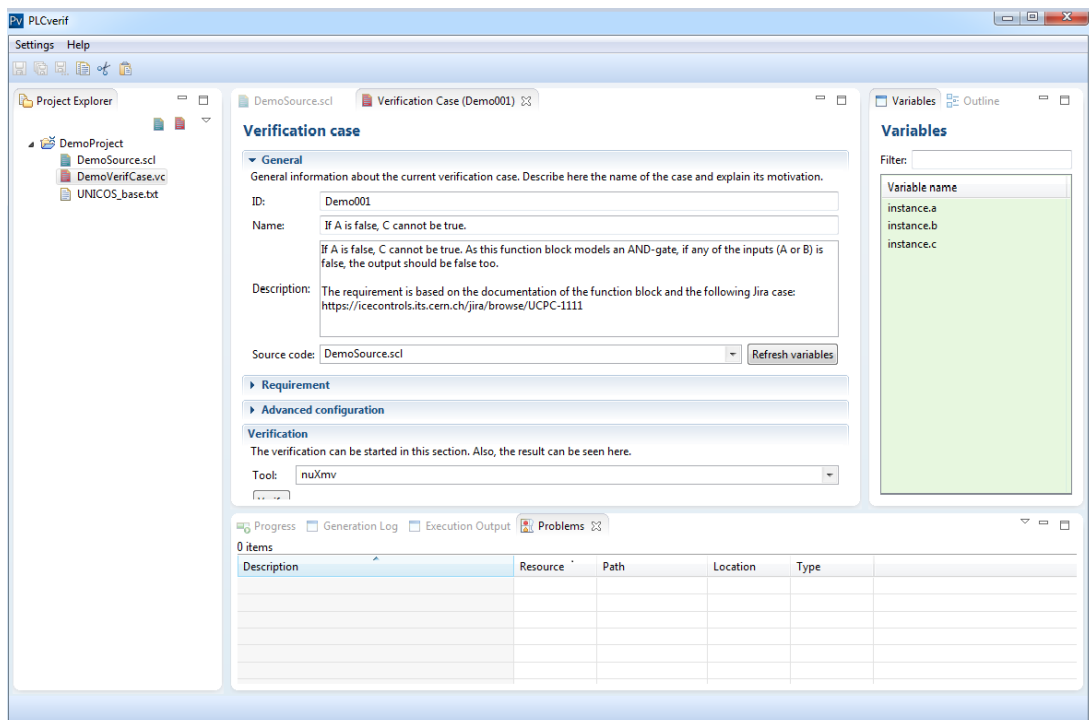
Subversion (SVN) και Jenkins.

Η δομή αυτού του κεφαλαίου είναι η ακόλουθη:

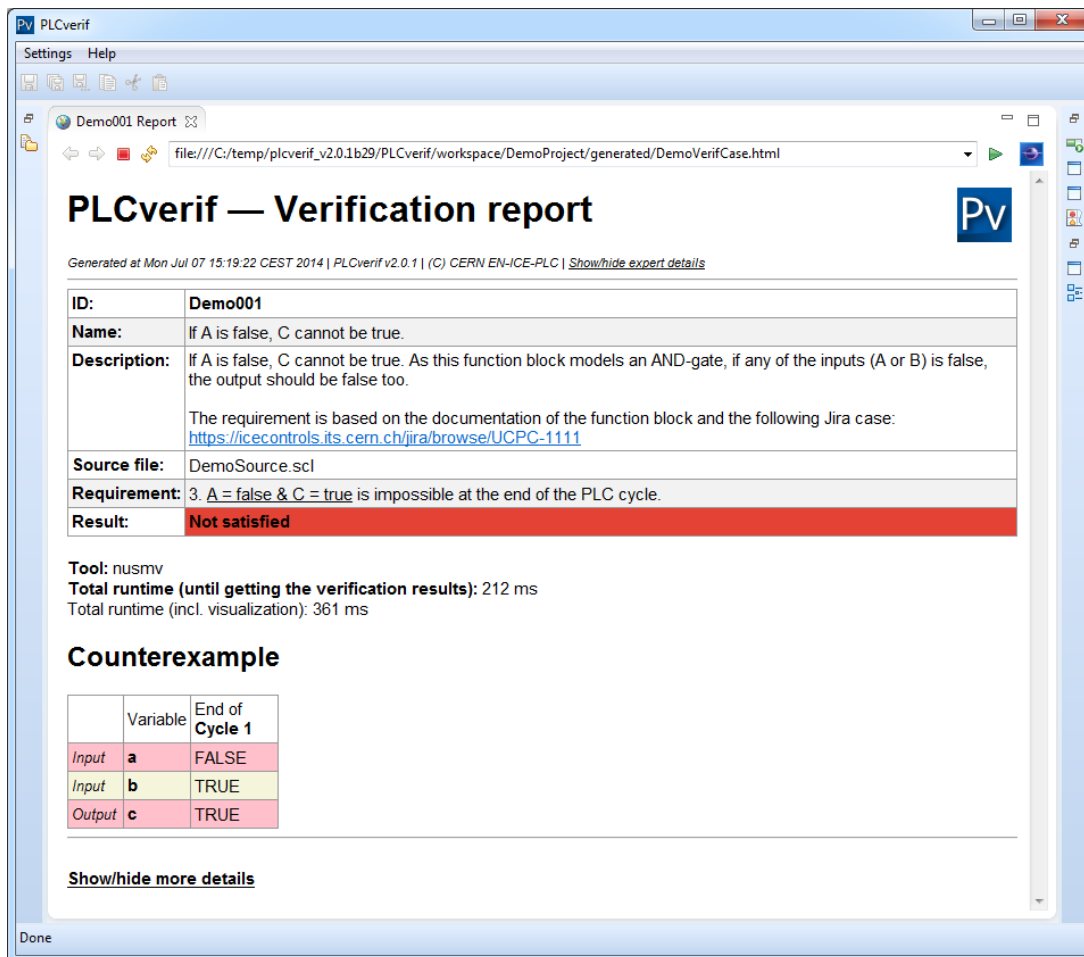
1. Στο κεφάλαιο 3.2 παρουσιάζεται η συνεισφορά για το πρώτο μέρος της εργασίας.
2. Στο κεφάλαιο 3.3 παρουσιάζονται τα οφέλη της ενσωμάτωσης του μοντέλου ελέγχου στην προγραμματιστική διαδικασία.



Σχήμα 4: SCL editor



Σχήμα 5: Περίπτωση επαλήθευσης



Σχήμα 6: Αναφορά επαλήθευσης

3.2 Συνεισφορά

Τα PLC προγράμματα που χρησιμοποιούνται στο CERN αναπτύσσονται κυρίως στο πλαίσιο που παρέχεται από το UNICOS. Το UNICOS παρέχει ένα σύνολο αντικειμένων (UNICOS Baseline Function Block) τα οποία αναπαριστούν συσκευές όπως βαλβίδες, αισθητήρες, αντλίες και άλλα στα PLC προγράμματα. Ένα PLC πρόγραμμα γραμμένο στη πλατφόρμα του UNICOS, συνδέει αυτά τα αντικείμενα προκειμένου να προσφέρει τον έλεγχο ενός ολοκληρωμένου σχεδίου παραγωγής.

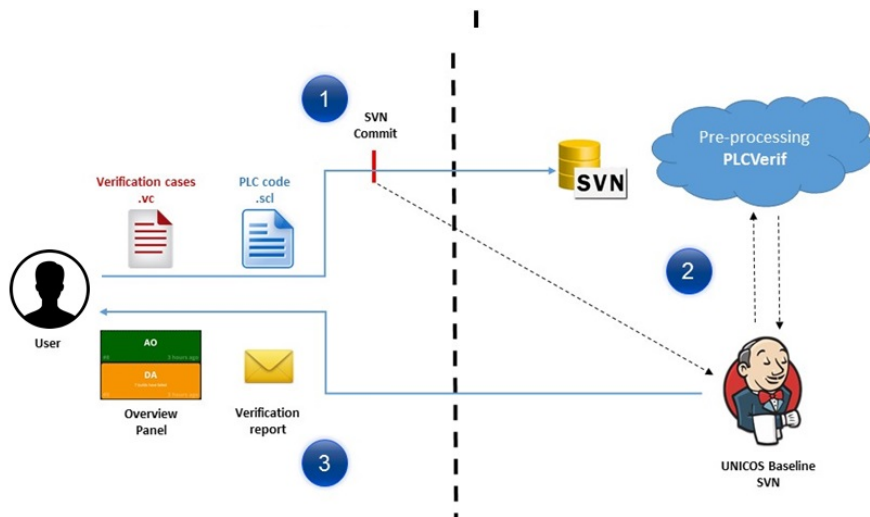
Μερικά PLC προγράμματα χρησιμοποιούν στον κώδικα τους blocks που παρέχονται από το υλισμικό, την ίδια την συσκευή, και περιέχουν ορισμένες συναρτήσεις χρησιμότητας και κατασκευαστικές συναρτήσεις όπως για παράδειγμα το TON [21]. Ο πηγαίος κώδικας για αυτές τις συναρτήσεις δεν γράφεται ξανά από τον προγραμματιστή αλλά παρέχεται σε ξεχωριστό φάκελο από το PLCverif. Ταυτόχρονα, καθώς η σύνταξη-γραμματική του εργαλείου δεν είναι ακόμα ολοκληρωμένη, υπάρχουν κάποια χαρακτηριστικά στα προγράμματα, συνήθως εντολές ή εκχωρήσεις, τα οποία δεν αναγνωρίζει. Αυτά τα χαρακτηριστικά, τα οποία δεν αλλοιώνουν την συμπεριφορά του προγράμματος, πρέπει είτε να τροποποιηθούν είτε να αφαιρεθούν από το πρόγραμμα προκειμένου το εργαλείο να μπορεί να επεξεργαστεί τον κώδικα.

Προς το παρόν, το εργαλείο μπορεί να επεξεργάζεται μόνο ένα αρχείο με πηγαίο κώδικα τη φορά, όμως οι συναρτήσεις που παρέχονται από το υλισμικό βρίσκονται σε ξεχωριστά αρχεία και επιπλέον υπάρχει ένα ακόμα αρχείο που αντιπροσωπεύει ένα block το οποίο αποτελεί κοινή βάση για τα υπόλοιπα. Επομένως, καθώς μόνο ένα αρχείο μπορεί να αναλυθεί κάθε φορά, πρέπει όλα τα αναφερόμενα να συγχωνευτούν σε ένα αρχείο.

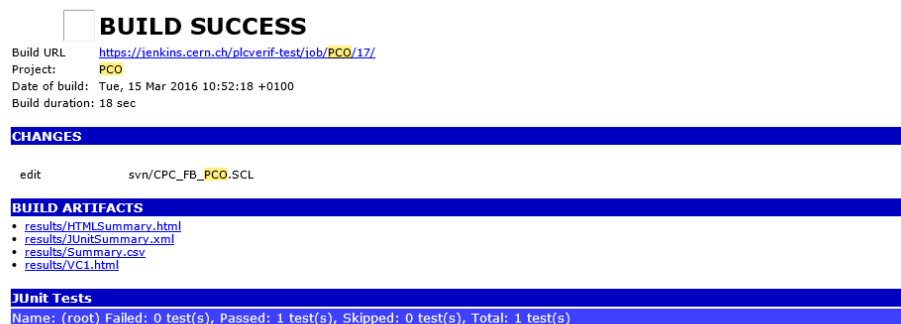
Με τη χρήση του SVN και του Jenkins στόχος της πτυχιακής είναι να απλοποιηθεί όλη αυτή τη διαδικασία για τον χρήστη. Αναπτύσσοντας δύο script, τα αρχεία συγχω-

νεύονται σε ένα και ταυτόχρονα γίνονται και όλες οι τροποποιήσεις στον κώδικα. Η προσέγγιση που ακολουθήθηκε μπορεί να χωριστεί σε τρία μέρη (Σχήμα 7):

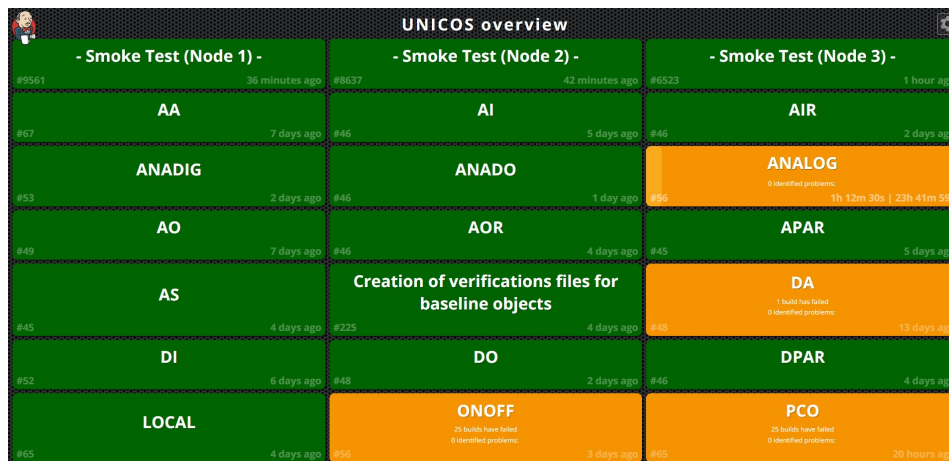
1. Ο χρήστης τροποποιεί το κώδικα από ένα UNICOS Baseline Function Block το οποίο αντιστοιχεί σε ένα αρχείο .SCL ή προσθέτει/τροποποιεί μια περίπτωση επαλήθευσης (verification case) και στη συνέχεια την εισάγει στο ΣΨΝ.
2. Αυτή η δράση πυροδοτεί το Jenkins να εκτελέσει τα ακόλουθα βήματα:
 - να επεξεργαστεί τα .SCL αρχεία με τη χρήση Python scripts.
 - να εκτελέσει το PLCverif να εφαρμόσει το μοντέλο ελέγχου.
3. Μόλις όλες οι περιπτώσεις επαλήθευσης για κάθε τροποποιημένο UNICOS Baseline Function Block έχουν αναλυθεί, το Jenkins θα στείλει στο χρήστη την αναφορά ανάλυσης με email. Στο Σχήμα 8 φαίνεται ότι δεν υπήρχε κάποια παράβαση στην περίπτωση επαλήθευσης μετά τις τροποποιήσεις που έγιναν στον κώδικα του προγράμματος PLC. Η αναφορά περιλαμβάνει λεπτομέρειες σχετικά με το όνομα του πρότζεκτ, την ημερομηνία εκτέλεσης, τη διάρκεια εκτέλεσης, το είδος των αλλαγών που υπέστη αλλά και τα αρχεία που ελέγχθηκαν μετά από αυτές. Επιπλέον, στο Jenkins παρέχεται ένα πάνελ όπου ο χρήστης μπορεί να δει τη τρέχουσα κατάσταση όλων των πρότζεκτ (Σχήμα 9)



Σχήμα 7: Επισκόπηση της προσέγγισης



Σχήμα 8: Παράδειγμα αναφοράς της ανάλυσης της περίπτωσης επαλήθευσης που αποστέλλεται στο χρήστη με email



Σχήμα 9: UNICOS πάνελ επισκόπησης των πρότζεκτ

3.3 Παρατηρήσεις

Η παραπάνω μεθοδολογία εφαρμόστηκε και στα 19 UNICOS Baseline Function Block τα οποία αποτελούνται από εκατοντάδες γραμμές κώδικα και διάφορα είδη περιπτώσεων επαλήθευσης επαληθεύτηκαν. Πλέον ο χρήστης πρέπει απλά να εισάγει τον καινούριο ή τροποποιημένο .SCL κώδικα του ή την καινούρια ή τροποποιημένη περίπτωση επαλήθευσης στο SVN. Στη συνέχεια το Jenkins αφού πυροδοτηθεί, αναλαμβάνει τα υπόλοιπα. Όταν θα υπάρχει αποτέλεσμα για τις νέες περιπτώσεις επαλήθευσης ή αν θα προκληθεί κάποια παράβαση λόγω τροποποίησης του πηγαίου κώδικα το Jenkins θα ενημερώσει τον χρήστη αποστέλλοντας του ένα email με τα αποτελέσματα της επαλήθευσης.

Με τη χρήση του script, η διαδικασία είναι πλέον αυτοματοποιημένη και κάθε πολυπλοκότητα είναι κρυμμένη από το χρήστη. Συνεπώς το όφελος είναι ότι ο χρήστης πλέον ξοδεύει πολύ λιγότερο χρόνο στην διαδικασία της επαλήθευσης. Ο χρήστης μπορεί να προσθέσει και να επαληθεύσει πολύ πιο αποδοτικά τις προδιαγραφές για το κάθε μοντέλο ενός PLC προγράμματος. Τέλος, δεν χρειάζεται πια να τροποποιεί και να συγχωνεύει ο ίδιος τα PLC αρχεία προκειμένου να χρησιμοποιήσει το εργαλείο.

4 Υλοποίηση μιας αφηρημένης τεχνικής

4.1 Εισαγωγή

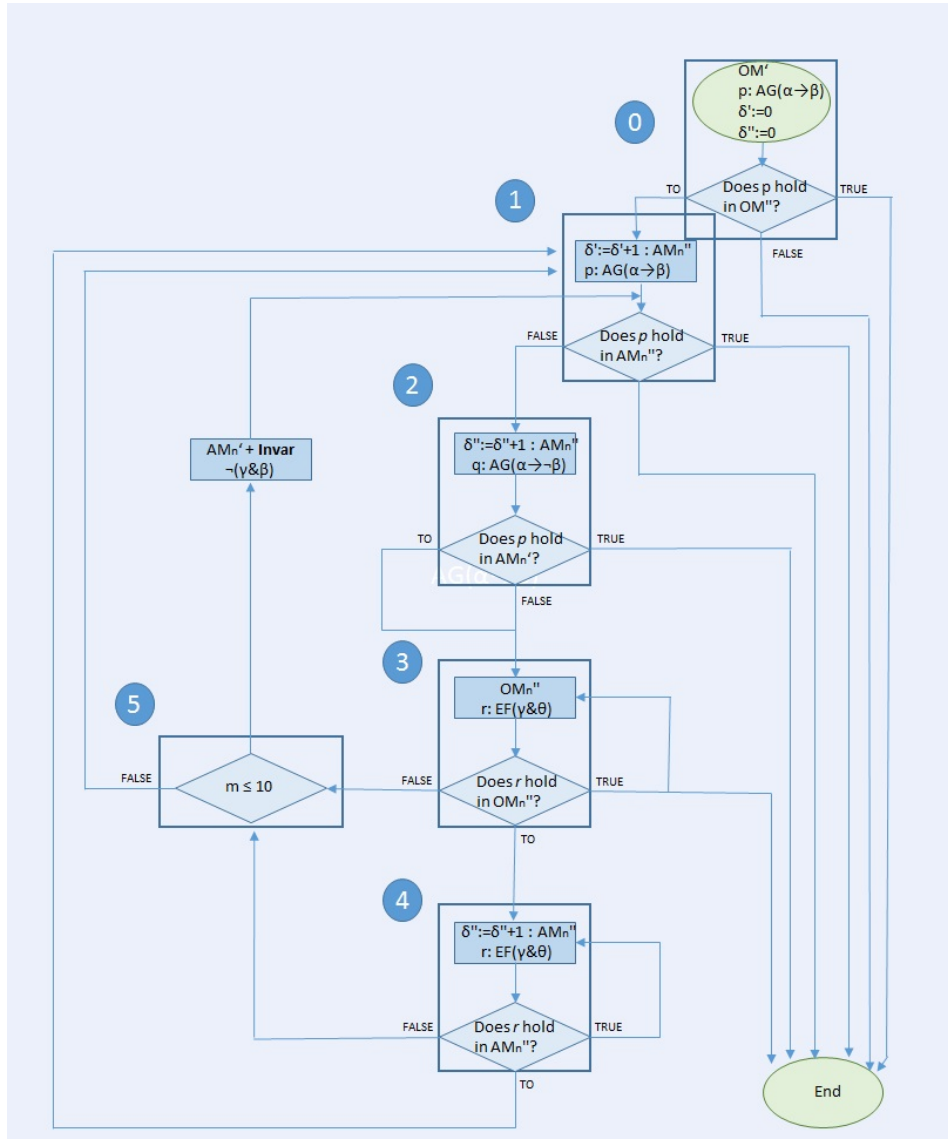
Στο κεφάλαιο 3 αναφέρθηκε το πρόβλημα της έκρηξης των καταστάσεων [12] [30]. Καθώς ο αριθμός των μεταβλητών ενός συστήματος αυξάνεται, αυξάνονται εκθετικά και οι καταστάσεις του συστήματος. Η έκρηξη του χώρου καταστάσεων προκαλείται όταν ο ελεγκτής μοντέλου προσπαθεί να επαληθεύσει ένα σύστημα με πολλά στοιχεία τα οποία κάνουν παράλληλες μετατροπές και επομένως ο αριθμός των συνδυασμών τους είναι πολύ μεγάλος. Τα PLC προγράμματα συνήθως περιέχουν πολλές μεταβλητές διαφόρων τύπων. Συνεπώς ο χώρος των καταστάσεων είναι τεράστιος.

Προκειμένου να μοντελοποιηθεί το σύστημα και να επαληθευτούν οι προδιαγραφές του, ο PLC κώδικας μεταφράζεται σε ένα ενδιάμεσο μοντέλο (intermediate model (IM) και τεχνικές απλούστευσης και σμίκρυνσης εφαρμόζονται σε αυτό. Οι τεχνικές που εφαρμόζονται στα μοντέλα μας μέχρι τώρα είναι [15]:

1. Cone of Influence (COI): αυτή η τεχνική αφαιρεί όλες τις μεταβλητές και τις εκχωρήσεις που δεν σχετίζονται με την προδιαγραφή που παρέχει ο χρήστης από το IM.
2. Rule-based reduction: αυτή η τεχνική περιορίζει τις καταστάσεις και τις μεταβλητές, αφαιρεί τα άδεια κλαδιά και συγχωνεύει μεταβάσεις και μεταβλητές.
3. Mode selection: με αυτή την τεχνική, οι παράμετροι με σταθερή τιμή μπορούν να αντικατασταθούν από μία σταθερά.

Οι παραπάνω τεχνικές δεν είναι πάντα αρκετές για να λύσουν το πρόβλημα της έκρηξης των καταστάσεων και να παρέχουν ένα αποτέλεσμα για την επαλήθευση της προδιαγραφής. Για να αντιμετωπιστεί αυτό το πρόβλημα και να βελτιωθεί η απόδοση της επαλήθευσης, μία νέα αφηρημένη τεχνική περιγράφεται σε μία διδακτορική διατριβή [20] η οποία όμως δεν είχε υλοποιηθεί μέχρι σήμερα. Χρησιμοποιώντας τον αλγόριθμο της τεχνικής αυτής, υπάρχουν περισσότερες πιθανότητες να επαληθευτούν ορισμένες ιδιότητες των μοντέλων, όπως για παράδειγμα απλές invariants της ακόλουθης μορφής: *αν μια είσοδος είναι ορισμένη, μια συγκεκριμένη έξοδος πρέπει να οριστεί επίσης*. Μία invariant είναι μία έκφραση η οποία πρέπει πάντα να ικανοποιείται από τον ελεγκτή μοντέλου.

Γενική ιδέα της αφηρημένης τεχνικής: Ο αφηρημένος αυτός αλγόριθμος ο οποίος θα αναφέρεται ως “*iterative variable abstraction*”, χωρίζεται σε 5 βήματα τα οποία απεικονίζονται στο Σχήμα 10. Όταν το PLCverif δεν μπορεί να δώσει αποτέλεσμα για την επαλήθευση μιας ιδιότητας στο αρχικό μοντέλο, ο αλγόριθμος δημιουργεί αφηρημένα μοντέλα (AM) βασισμένα στο αρχικό. Με βάση αυτή την τεχνική λοιπόν AM δημιουργούνται επαναληπτικά καθώς ταυτόχρονα προσπαθούν να αποδείξουν ότι το αποτέλεσμα από την επαλήθευση της ιδιότητας είναι θετικό για το τρέχον AM'_n . Το AM αποτελεί μια υπερ-προσέγγιση του αρχικού μοντέλου.



Σχήμα 10: Βήματα του αλγορίθμου variable abstraction [20]

Δημιουργία AM: Μερικές από τις μεταβλητές που περιέχονται στο αρχικό μοντέλο, πρέπει να αφαιρεθούν από το variable dependency graph του μοντέλου και να αντικατασταθούν από μη-ντετερμινιστικές τιμές προκειμένου να δημιουργηθεί το AM. Καθώς αυτές οι μεταβλητές δεν εξαρτώνται από άλλες μεταβλητές στο πρόγραμμα, ο αλγόριθμος COI θα μπορέσει να απλουστεύσει και να μικρύνει το μοντέλο εξαλείφοντας περισσότερες μεταβλητές.

Στο Σχήμα 11 φαίνεται ένα παράδειγμα του variable dependency graph για το μοντέλο του προγράμματος που περιγράφεται παρακάτω στο παράδειγμα 1 και για την προδιαγραφή ότι “Αν το b είναι αληθές στο τέλος του PLC κύκλου, τότε και το a πρέπει να είναι πάντα αληθές στο τέλος του ίδιου κύκλου”. Τα κόκκινα βέλη απεικονίζουν τις εξαρτήσεις μεταξύ των εκχωρήσεων, ενώ με γκρι είναι οι μεταβλητές που περιέχονται στη προδιαγραφή που πρέπει να επαληθευτεί. Επιπλέον, όταν δύο μεταβλητές είναι συνδεδεμένες με ένα κόκκινο βέλος, η απόστασή τους είναι ίση με 1. Οι αποστάσεις χρησιμοποιούνται από τον αλγόριθμο “iterative variable abstraction” προκειμένου να κατασκευαστούν τα AM όπως περιγράφεται παρακάτω. Στο συγκεκριμένο παράδειγμα λοιπόν, με γκρι χρώμα είναι οι μεταβλητές a , b γιατί είναι αυτές που περιέχονται στην προδιαγραφή. Επίσης στο a εκχωρούνται οι μεταβλητές d και e γι αυτό και στο γράφημα το a είναι συνδεδεμένο με κόκκινα βέλη με αυτές τις δύο μεταβλητές. Αντίστοιχα στο b εκχωρείται η μεταβλητή f και αυτό αναπαρίσταται και στο γράφημα. Με την ίδια λογική γίνονται και οι υπόλοιπες αντιστοιχίες.

Listing 1: Example of SCL code

```
FUNCTION Test
VAR

  a : BOOL;
  b : BOOL;

  c : BOOL;
  d : BOOL;
  e : BOOL;
  f : BOOL;

END_VAR

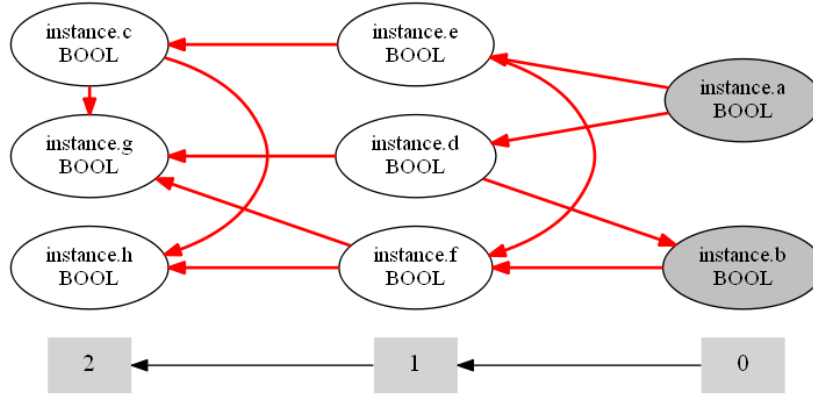
VAR_INPUT

  g : BOOL;
  h : BOOL;
END_VAR

BEGIN

  f := g AND h;
  b := f;
  c := h AND NOT g;
  e := f AND NOT c;
  d := b AND g;
  a := d AND e;

END_FUNCTION
```

Σχήμα 11: Το παραγόμενο Variable dependency graph από το παράδειγμα του Listing 5

Χάρη σε αυτή την αφηρημένη τεχνική:

- το μέγεθος του χώρου των καταστάσεων ή PSS (Potential State Space) του αφηρημένου μοντέλου είναι μικρότερο καθώς περιέχει λιγότερες μεταβλητές,
- συνήθως, στο AM αντιπροσωπεύεται ένα μεγαλύτερο εύρος πιθανών συμπεριφορών των μεταβλητών που περιέχονται στη προδιαγραφή σε σχέση με το αρχικό μοντέλο (OM).

Το PSS (Potential State Space) αντιπροσωπεύει τις πιθανές καταστάσεις που μια διεργασία μπορεί να έχει. Για παράδειγμα, ένα σύστημα αποτελούμενο από n διεργασίες όπου η κάθε μία μπορεί να έχει m καταστάσεις, θα έχει έναν χώρο καταστάσεων $n*m$.

Ο αλγόριθμος χρησιμοποιεί δύο είδη ιδιοτήτων: reachability properties και safety properties. Οι reachability properties διατυπώνουν ότι μια συγκεκριμένη κατάσταση δεν μπορεί να επιτευχθεί ενώ οι safety properties διατυπώνουν ότι ένα συγκεκριμένο συμβάν δεν πρέπει ποτέ να συμβεί κάτω από συγκεκριμένες συνθήκες (η safety property έχει τη μορφή: $(AG(\alpha \rightarrow \beta))$). Εφαρμόζοντας το μοντέλο ελέγχου στη safety property if p holds on AM'_n , συνεπάγεται ότι τόσο το αφηρημένο μοντέλο όσο και το αρχικό είναι συμβατά με την προδιαγραφή καθώς περισσότερες συμπεριφορές αντιπροσωπεύονται στο AM. Στην αντίθετη περίπτωση, όπου η ιδιότητα p δεν είναι συμβατή με το μοντέλο AM'_n , ένα counterexample c παράγεται από τον ελεγκτή μοντέλου.

Τα counterexample μπορούν να είναι αρκετά περίπλοκα καθώς περιέχουν μεταβλητές από διαφορετικούς PLC κύκλους. Για να επαληθευθεί ότι η ιδιότητα είναι συμβατή με το αρχικό μοντέλο απαιτείται περαιτέρω ανάλυση του counterexample c μέσα από τα βήματα 3 και 4 του αλγορίθμου ώστε να αποσαφηνιστεί αν αυτό είναι αληθές ή ψευδές. Όταν υπάρχουν ψευδή αληθή στο στάδιο επαλήθευσης ή όταν η ιδιότητα δεν είναι συμβατή με το μοντέλο, τότε το counterexample μπορεί να θεωρηθεί ψευδές.

Η δομή αυτού του κεφαλαίου είναι η ακόλουθη:

1. Στο κεφάλαιο 4.2 περιγράφεται η υλοποίηση του αλγορίθμου variable abstraction.
2. Στο κεφάλαιο 4.3 παρουσιάζονται τα αποτελέσματα των πειραμάτων που διεξήχθησαν για να ελεγχθεί η αποδοτικότητα του αλγορίθμου.
3. Στο κεφάλαιο 4.4 περιγράφονται τα συμπεράσματα που προέκυψαν μετά τη διεξαγωγή των πειραμάτων.

4.2 Συνεισφορά

Στις παρακάτω παραγράφους περιγράφονται πιο αναλυτικά τα βήματα του αλγορίθμου και η υλοποίησή τους σε ένα script. Το script είναι υλοποιημένο σε Python και ο χρήστης μπορεί να το εκτελέσει από τη γραμμή εντολών βάζοντας σαν παράμετρο το path στο οποίο βρίσκεται η περίπτωση επαλήθευσης που θέλει να ελέγξει. Το script καλεί επαναληπτικά την To command line έκδοση του PLCVerif line [32] και περιέχει όλα τα βήματα του αλγορίθμου. Τα αρχεία των περιπτώσεων επαλήθευσης έχουν την κατάληξη .vc και είναι σε μορφή XML και για την επεξεργασία τους χρησιμοποιήθηκε

Αντιστοίχιση συμβόλων:

- OM' : το αποτέλεσμα της εφαρμογής των property preserving reductions στο παραγόμενο μοντέλο.
- OM'' : το αποτέλεσμα της εφαρμογής των property preserving reductions για την reachability property r .
- δ' : η απόσταση μεταξύ των μεταβλητών στο variable dependency graph για το OM' .
- δ'' : η απόσταση μεταξύ των μεταβλητών στο variable dependency graph για το OM'' .

Στην προσπάθεια του PLCverif να επαληθεύσει μια ιδιότητα στο OM' , ο ελεγκτής μοντέλου μπορεί να μην μπορέσει να δώσει αποτέλεσμα λόγω λήξης του χρόνου –ή TO (time out). Σε αυτή την περίπτωση η εκτέλεση του αλγορίθμου πυροδοτείται προκειμένου να ελέγξει την ίδια ιδιότητα. Λόγω αυτού το αρχικό αυτό βήμα δε συμπεριλαμβάνεται στα υπόλοιπα 5 του αλγορίθμου [20]

Υλοποίηση και περιγραφή των βημάτων του αλγορίθμου

1. Έλεγχος της αρχικής safety property (p) στο αφηρημένο μοντέλο (AM').
2. Έλεγχος της safety property (q) στο ίδιο αφηρημένο μοντέλο (AM').
3. Έλεγχος της reachability property (r) στο αρχικό μοντέλο (OM'').
4. Έλεγχος της reachability property (r) στο αφηρημένο μοντέλο (AM'').
5. Εξαγωγή invariant από το counterexample αν ο αριθμός του συνόλου των invariants είναι μικρότερος από 10.

Βήμα 1: Έλεγχος της αρχικής safety property (p) στο αφηρημένο μοντέλο (AM') Σε αυτό το βήμα ένα AM' του αρχικού μοντέλου OM' , παράγεται αυτόματα εξάγοντας μεταβλητές σαν εισόδους από την απόσταση $\delta' = 1$. Μετά από αυτό, ο έλεγχος μοντέλου προσπαθεί να επαληθεύσει αν η ιδιότητα p ικανοποιεί το AM' ή όχι.

Υλοποίηση: από το script, δημιουργείται ένα αρχείο στο οποίο θα αποθηκευτεί το περιεχόμενο της περίπτωσης επαλήθευσης για το AM' για το πρώτο βήμα. Το περιεχόμενο του είναι το ίδιο με την περίπτωση επαλήθευσης του αρχικού μοντέλου. Στη συνέχεια οι μεταβλητές εισόδου εξάγονται από την απόσταση δ' που η τιμή της αντιστοιχεί στην τρέχουσα επανάληψη και προστίθενται στις μεταβλητές που αντιπροσωπεύουν τις αρχικές εισόδους. Ο αλγόριθμος εξάγει τις μεταβλητές εισόδου με βάση το variable dependency graph που παράγεται στο αρχικό βήμα.

Βήμα 2: Έλεγχος της safety property (q) στο ίδιο αφηρημένο μοντέλο (AM') Στο δεύτερο βήμα, ο αλγόριθμος προσπαθεί να εξάγει περισσότερες πληροφορίες για την επαλήθευση του p στο AM'_n . Για να γίνει αυτό, ελέγχει την safety property q στο ίδιο αφηρημένο μοντέλο AM'_n . Η ιδιότητα q είναι η ίδια με την p απλά αυτή τη φορά σχηματίζεται άρνηση στην παράμετρο β ($AG(\alpha \rightarrow \neg\beta)$).

Υλοποίηση: δημιουργείται ένα αρχείο στο οποίο θα αποθηκευτεί το περιεχόμενο της περίπτωσης επαλήθευσης για το δεύτερο βήμα. Το αρχείο έχει το ίδιο περιεχόμενο με το περιεχόμενο του αρχείου που δημιουργήθηκε στο πρώτο βήμα. Η safety property ελέγχεται στο ίδιο AM'_n απλά με άρνηση στη παράμετρο β .

Βήμα 3: Έλεγχος της reachability property (r) στο αρχικό μοντέλο (OM'') Στο βήμα 3 ο αλγόριθμος επιχειρεί να αναλύσει το counterexample c εξάγοντας από αυτό την reachability property r . Προσθέτοντας την reachability property, ένα νέο OM'' δημιουργείται και πολλές φορές είναι διαφορετικό από το OM' . Αυτό μπορεί να συμβεί γιατί η ιδιότητα r μπορεί να περιέχει διαφορετικές μεταβλητές από την ιδιότητα p και συνεπώς οι τεχνικές μείωσης θα απαλείψουν διάφορες μεταβλητές από το μοντέλο.

Υλοποίηση: δημιουργείται ένα αρχείο στο οποίο θα αποθηκευτεί το περιεχόμενο της περίπτωσης επαλήθευσης για το τρίτο βήμα. Το περιεχόμενο του αρχείου θα είναι ίδιο με το περιεχόμενο του αρχείου της αρχικής περίπτωσης επαλήθευσης (βήμα 0). Η reachability property θα εξαχθεί είτε από το βήμα 1 είτε από το βήμα 5. Οι μεταβλητές που περιλαμβάνει η ιδιότητα θα αντικαταστήσουν τις παραμέτρους της περίπτωσης επαλήθευσης. Αν το counterexample έχει περισσότερους από έναν κύκλους και αν το αποτέλεσμα που παράγει ο ελεγκτής μοντέλου είναι 'Αληθές', τότε όλοι οι κύκλοι πρέπει να εξεταστούν (εκτός αν το περιεχόμενό τους είναι ίδιο) μέχρι το αποτέλεσμα της επαλήθευσης να είναι διαφορετικό.

Βήμα 4: Έλεγχος της reachability property (r) στο αφηρημένο μοντέλο (AM''). Σε αυτό το βήμα, ένα AM''_n δημιουργείται μετά το TO που προκύπτει στο βήμα 3. Αυτή τη φορά οι μεταβλητές εισόδου που χρειάζονται για τη δημιουργία του AM εξάγονται από την απόσταση δ'' της τρέχουσας επανάληψης με τον ίδιο ακριβώς τρόπο όπως και στο βήμα 1.

Υλοποίηση: δημιουργείται ένα αρχείο στο οποίο θα αποθηκευτεί το περιεχόμενο της περίπτωσης επαλήθευσης για το τέταρτο βήμα το οποίο έχει το ίδιο περιεχόμενο με το αρχείο του βήματος 3. Ο αλγόριθμος εξάγει τις μεταβλητές εισόδου από την απόσταση δ'' με βάση το variable dependency graph που παράγεται στο βήμα 3.

Βήμα 5: Εξαγωγή invariant από το counterexample αν ο αριθμός του συνόλου των invariants είναι μικρότερος από 10. Το τελευταίο αυτό βήμα είναι υπεύθυνο για την προσθήκη invariant στο AM' και για την λήψη της απόφασης σχετικά με το αν ο αλγόριθμος πρέπει να προβεί σε νέα αφαίρεση του μοντέλου $AM'_n + 1$ αυξάνοντας το δ' κατά 1 αν το σύνολο των δυνατών invariant έχει φτάσει το όριο.

Για παράδειγμα, αν ο μέγιστος αριθμός των πιθανών invariant m είναι ≤ 10 , μία invariant θα προστεθεί στο τρέχον AM' . Αν το m είναι ≥ 10 ο αλγόριθμος θα προβεί σε καινούρια επανάληψη.

Υλοποίηση: δημιουργείται ένα αρχείο στο οποίο θα αποθηκευτεί το περιεχόμενο της περίπτωσης επαλήθευσης για το πέμπτο βήμα και το περιεχόμενο της είναι ίδιο με το αρχείο του βήματος 1 αν το $m = 1$ ή ίδιο με το περιεχόμενο του βήματος 5 αν το $m \geq 1$ και $m \leq 10$.

- Στο appendix στο κεφάλαιο Γ.2 περιγράφεται ένα αναλυτικό παράδειγμα για την εφαρμογή του αλγορίθμου.
- Στο κεφάλαιο Γ.3 στο appendix περιγράφονται αναλυτικά τα πειράματα που διεξήχθησαν με τον συγκεκριμένο αλγόριθμο καθώς και οι λεπτομέρειες των PLC προγραμμάτων των οποίων οι προδιαγραφές πέρασαν το στάδιο επαλήθευσης.
- Τέλος, στο κεφάλαιο Γ.4 περιγράφονται τα συμπεράσματα που προέκυψαν μετά την διεξαγωγή των πειραμάτων σχετικά με τα πλεονεκτήματα του αλγορίθμου και την αποδοτικότητά του.

5 Στατική ανάλυση κώδικα

5.1 Εισαγωγή

Στα προηγούμενα κεφάλαια, η πτυχιακή εργασία επικεντρωνόταν στην εφαρμογή του ελεγκτή μοντέλου στα PLC προγράμματα. Στο κεφάλαιο αυτό, ο κώδικας PLC θα εξεταστεί με διαφορετικό τρόπο. Η στατική ανάλυση είναι ένας διαφορετικός τρόπος να ελεγχθεί το PLC πρόγραμμα χωρίς να χρειάζονται οι προδιαγραφές του. Χρησιμοποιώντας στατική ανάλυση κώδικα, ένας πιθανώς λανθασμένος κώδικας ή μια απροσδιόριστη συμπεριφορά μπορούν να βρεθούν στο πρόγραμμα. Όπως αναφέρθηκε στο κεφάλαιο 2, υπάρχουν ήδη μερικά εργαλεία που στοχεύουν στην ανάλυση PLC προγραμμάτων. Τα εργαλεία PLC Checker από την Itrix Automation Company, Codesys Static Analysis από την CoDeSyS και το Arcade.PLC που αναπτύχθηκε στο RWTH Aachen University αναλύονται στο appendix στο κεφάλαιο H.2 και παρουσιάζονται και στους πίνακες 1 και 2. Επιπλέον, στο CERN, δύο τεχνολογίες που χρησιμοποιούνται για δύο διαφορετικά πρότζεκτ αποτέλεσαν την βάση για τη δημιουργία δύο πρωτότυπων εργαλείων στατικής ανάλυσης προκειμένου να αναλυθούν τα οφέλη της ως προς τα PLC προγράμματα.

Πίνακας 1: Πίνακας σύγκρισης I των εργαλείων στατικής ανάλυσης για προγράμματα PLC.

Tool	License	Report	Extensible rules	Language	Restrictions
CoDeSyS	Yes	Console	Yes	IEC 61131-3	Limited to CoDeSys platform
PLC Checker	Yes	PDF/e-mail	Yes	Siemens, OMRON, CoDeSys, Scheinder, Rockwell - Automation	So far .AWL and .ASC files are required to analyze an SCL file.
Arcade.PLC	No	Console	No	IEC 61131 ST, IEC 61131 IL, Siemens STL	-

Πίνακας 2: Πίνακας σύγκρισης II των εργαλείων στατικής ανάλυσης για προγράμματα PLC.

Tool	Rule categories	Exclude errors	Error Message
CoDeSyS	coding rules, naming conventions, metrics	No info	Error, warning
PLC Checker	naming rules, comments, writing rules, structure rules	Yes	Info, warning, error, fatal
Arcade.PLC	Non-rule based tool	No	Fatal errors, errors, warnings

5.2 Υλοποίηση

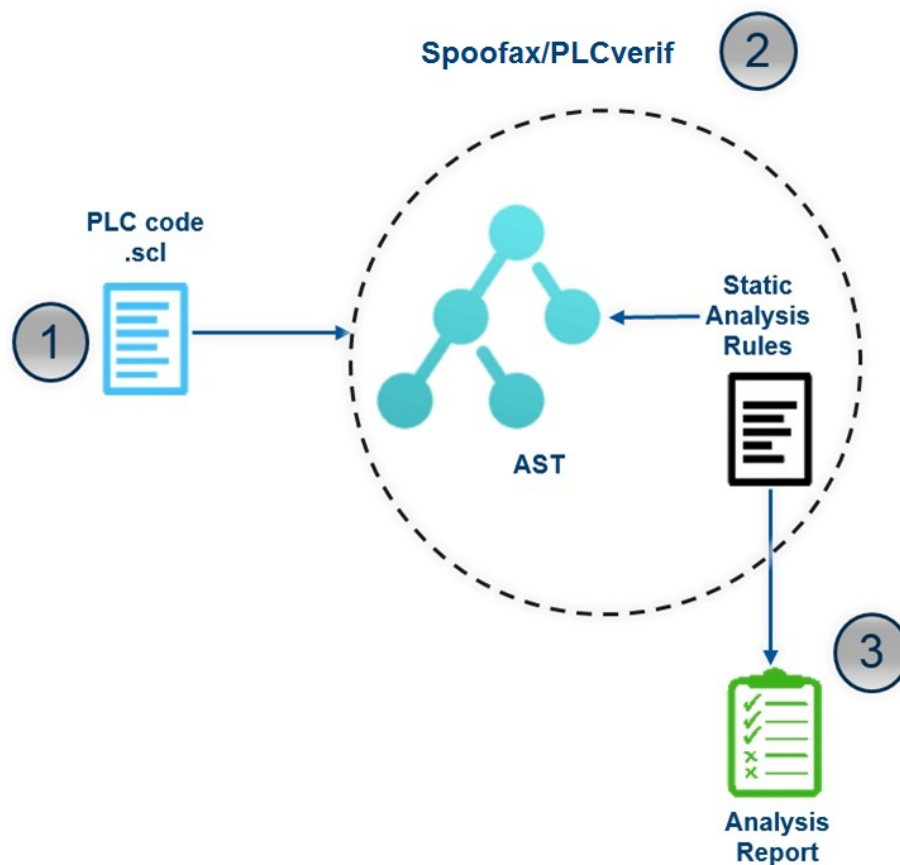
Όπως προαναφέρθηκε, δύο πρωτότυπα εργαλείων στατικής ανάλυσης αναπτύχθηκαν για προγράμματα PLC γραμμένα σε SCL. Ο στόχος των εργαλείων στη παρούσα φάση, είναι να ανιχνευθούν απλές παραβάσεις τόσο στη δομή του κώδικα όσο και στην ονοματολογία των POU (Program organization unit) και των μεταβλητών. Καθώς το πρότζεκτ είναι σε πειραματικό ακόμα στάδιο ακολουθήθηκαν οι δύο ακόλουθες προσεγγίσεις για τη δημιουργία του εργαλείου:

Η πρώτη προσέγγιση βασίζεται σε ένα πρότζεκτ για το οποίο αναπτύχθηκε μία γλώσσα ειδικού πεδίου προκειμένου να μετατρέψει το PLC πρόγραμμα σε μορφή δένδρου αφηρημένης σύνταξης. Τα δένδρα αφηρημένης σύνταξης είναι δομές δεδομένων

που χρησιμοποιούνται από τους μεταγλωττιστές. Χάρη στην ιδιότητα τους να αναπαριστούν τον κώδικα του προγράμματος, διευκολύνουν την προσπέλαση του από τον μεταγλωττιστή.

Αυτή η πρώτη προσέγγιση δημιουργήθηκε με τη χρήση της πλατφόρμας Spoofox. Χάρη στο δένδρο αφηρημένης σύνταξης που παράγεται, οι κανόνες της στατικής ανάλυσης μπορούν να αναπτυχθούν βασιζόμενοι σε αυτό. Για την ανάπτυξη του εργαλείου υπήρξαν δύο υποπροσεγγίσεις οι οποίες είναι βασισμένες σε μία διεπαφή του Eclipse στην πλατφόρμα του Spoofox.

Η δεύτερη προσέγγιση βασίζεται στο εργαλείο PLCverif. Το PLCverif το οποίο παρέχει και αυτό μια διεπαφή του Eclipse, χρησιμοποιεί την τεχνολογία Xtext προκειμένου να προσπελάσει τον κώδικα του PLC προγράμματος. Αφού ο κώδικας προσπελαστεί, το PLCverif τον μεταφράζει σε αναπαράσταση δένδρου αφηρημένης σύνταξης το οποίο μπορεί να χρησιμοποιηθεί για την ανάπτυξη των κανόνων στατικής ανάλυσης. Στο σχήμα 12 μπορούμε να δούμε την εικονική αναπαράσταση της διαδικασίας που ακολουθείται από τον προγραμματιστή η οποία είναι η ίδια και για τις δύο προσεγγίσεις.



Σχήμα 12: Ροή εργασιών για τις προσεγγίσεις του εργαλείου στατικής ανάλυσης.

Προσέγγιση βασισμένη στην πλατφόρμα Spoofox Η προσέγγιση αυτή χωρίζεται σε δυο υποπροσεγγίσεις. Η πρώτη χρησιμοποιεί την συναρτησιακή γλώσσα προγραμματισμού Stratego/XT ενώ η δεύτερη χρησιμοποιεί σαν γλώσσα προγραμματισμού την Java.

Όπως περιγράφηκε στο Σχήμα 12, η διαδικασία για την στατική ανάλυση του κώδικα αποτελείται από 3 βήματα:

1. Το PLC πρόγραμμα προσπελάσσεται αφού δοθεί σαν είσοδος στην πλατφόρμα και στη συνέχεια μεταφράζεται σε αναπαράσταση δένδρου αφηρημένης σύνταξης.
2. Έπειτα το δένδρο αφηρημένης σύνταξης αναλύεται και οι κανόνες στατικής ανάλυσης μπορούν να γραφτούν με βάση αυτό.
3. Τέλος, μετά την εκτέλεση του εργαλείου, τα αποτελέσματα της ανάλυσης του κώδικα παρουσιάζονται στην κονσόλα του Eclipse ή και σε μορφή αναφοράς στην πε-

ρίπτωση της δεύτερης υποπροσέγγισης (Java).

Η κύρια διαφορά μεταξύ των δύο αυτών υποπροσεγγίσεων, είναι ο τρόπος με τον οποίο έχουμε πρόσβαση στο δένδρο αφηρημένης σύνταξης για την υλοποίηση των κανόνων. Το δένδρο αφηρημένης σύνταξης που παρέχεται από την πρώτη υποπροσέγγιση Stratego/XT είναι χαμηλού επιπέδου. Για παράδειγμα τα ονόματα διαφορετικών τύπων μεταβλητών θα αναπαρίστανται σε διαφορετικά μοτίβα του δένδρου. Σαν αποτέλεσμα, προκειμένου να συλλεχθούν τα ονόματα των μεταβλητών, όλα τα μοτίβα πρέπει να ελεγχθούν. Ωστόσο, το δένδρο αφηρημένης σύνταξης αναπαριστά όλο τον PLC κώδικα του προγράμματος.

Προκειμένου να υπάρχει μεγαλύτερη ευελιξία στον τρόπο γραφής των κανόνων ως προς τη γλώσσα προγραμματισμού, δημιουργήθηκε και η δεύτερη προσέγγιση η οποία βασίζεται στην Java. Το ίδιο δένδρο αφηρημένης αναπαράστασης μπορούσε να χρησιμοποιηθεί και για αυτήν την προσέγγιση επειδή όμως ήταν χαμηλού επιπέδου επιλέχθηκε να αντικατασταθεί. Μέχρι στιγμής μόνο κανόνες που στοχεύουν στο μέρος του κώδικα που αφορά την δήλωση των μεταβλητών μπορούν να γραφτούν καθώς το κομμάτι που αφορά την προσπέλαση των εκχωρήσεων τιμών στις μεταβλητές δεν έχει υλοποιηθεί ακόμα. Επιπλέον, η δεύτερη αυτή υποπροσέγγιση παρέχει στο χρήστη τη δυνατότητα να επιλέξει τους κανόνες που θέλει να συμπεριλάβει στην στατική ανάλυση του προγράμματος που θα εφαρμόσει.

Περισσότερες λεπτομέρειες αναγράφονται στο κεφάλαιο H.3 στο appendix σχετικά με τον τρόπο υλοποίησης της κάθε υποπροσέγγισης, με τους κανόνες που δημιουργήθηκαν αλλά και με παραδείγματα κώδικα.

Προσέγγιση βασισμένη στο εργαλείο PLCverif Στην τελευταία προσέγγιση, το εργαλείο στατικής ανάλυσης είναι ενσωματωμένο με το εργαλείο επαλήθευσης που παρέχει το PLCverif. Το δένδρο αφηρημένης σύνταξης που παράγεται από το PLC μπορεί να υποστηρίξει όλα τα Xtext προγράμματα που χρησιμοποιούνται στο CERN και είναι γραμμένα σε SCL. Ωστόσο, η γραμματική του εργαλείου δεν είναι πλήρης καθώς καλύπτει προς το παρόν μόνο τις ανάγκες των PLC προγραμμάτων που χρησιμοποιούνται στο CERN.

Η λογική ανάπτυξης των κανόνων είναι η ίδια με την πρώτη προσέγγιση και περισσότερες λεπτομέρειες περιγράφονται στο κεφάλαιο H.3 στο appendix.

Πίνακας 3: Σύγκριση μεταξύ των προσεγγίσεων για το εργαλείο στατικής ανάλυσης κώδικα.

Approach	Grammar	Implementation	Pros	Cons
Stratego/XT	Spoofax	Stratego	Complete grammar	Low-level AST access
Java	Spoofax	Java	Simple rules	Partial AST support Manual AST manipulation
PLCverif	Xtext	Java	Simple rules	Partial grammar

Τέλος στο κεφάλαιο H.4 στο appendix περιγράφονται τα συμπεράσματα που προέκυψαν για το εργαλείο στατικής ανάλυσης που δημιουργήθηκε.

6 Συμπεράσματα και Μελλοντικές επεκτάσεις

6.1 Συμπεράσματα

Αυτή η πτυχιακή εργασία έχει ως σκοπό τη βελτίωση της ποιότητας των PLC προγραμμάτων με τη βοήθεια της στατικής ανάλυσης κώδικα και του ελεγκτή μοντέλων. Επιπλέον εφαρμόζοντας τον αλγόριθμο της αφηρημένης τεχνικής επιτεύχθηκε η μερική αντιμετώπιση του προβλήματος εκρήξεων καταστάσεων, πρόβλημα που οι προγραμματιστές Αύτη η πτυχιακή εργασία δημιουργήθηκε με σκοπό να βελτιωθεί η ποιότητα των PLC προγραμμάτων με τη βοήθεια της στατικής ανάλυσης κώδικα και του ελεγκτή μοντέλων. Επιπλέον εφαρμόζοντας τον αλγόριθμο της αφηρημένης τεχνικής επιτεύχθηκε

η μερική αντιμετώπιση του προβλήματος εκρήξεων καταστάσεων, πρόβλημα που οι προγραμματιστές αντιμετωπίζουν προσπαθώντας να επαληθεύσουν τις προδιαγραφές του συστήματος στο ίδιο το σύστημα. Για να διασφαλιστεί η εφαρμογή των παραπάνω, τέθηκαν τρεις διαφορετικοί στόχοι:

- ενσωμάτωση του ελεγκτή μοντέλων στον προγραμματισμό των PLC. Κρύβοντας κάθε πολυπλοκότητα από τον χρήστη πλέον μπορούν να επαληθευθούν περισσότερες από μία περιπτώσεις επαλήθευσης μέσω του PLCverif χωρίς την παρέμβαση του χρήστη.
- ανάπτυξη του αλγόριθμου της αφηρημένης τεχνικής «variable abstraction». Αντιμετωπίστηκε μερικώς το πρόβλημα της έκρηξης των καταστάσεων ιδιαίτερα για τις προδιαγραφές που ήταν συμβατές με το μοντέλο του συστήματος.
- ανάλυση υπάρχοντων εργαλείων στατικής ανάλυσης PLC προγραμμάτων και ανάπτυξη δύο πρωτότυπων εργαλείων στατικής ανάλυσης προκειμένου να εντοπιστούν παραβάσεις στον κώδικα των PLC προγραμμάτων που χρησιμοποιούνται στο CERN.

Συνδυάζοντας τον ελεγκτή μοντέλου και τη στατική ανάλυση, μπόρεσε να επιτευχθεί μια πιο ισχυρή επαλήθευση του κώδικα. Συνεπώς, για την επαλήθευση της ορθότητας ενός συστήματος, είναι απαραίτητες και οι δύο τεχνικές καθώς η κάθε μια παρέχει διαφορετικό είδος ανάλυσης. Σε αντίθεση με τον ελεγκτή μοντέλου, η στατική ανάλυση μπορεί να εφαρμοστεί στο πρόγραμμα χωρίς να χρειάζεται ο χρήστης να δώσει σαν είσοδο στο εργαλείο τις προδιαγραφές του συστήματος προκειμένου να επαληθεύσει την ορθότητά του. Ταυτόχρονα, καθώς κάθε πολυπλοκότητα έχει αφαιρεθεί, το μοντέλο ελέγχου μπορεί εύκολα να χρησιμοποιηθεί και συνήθως να εγγυηθεί την επαλήθευση του μοντέλου. Παρόλο που όλα τα διεξαγόμενα πειράματα για τα προγράμματα που χρησιμοποιούνται στο CERN δείχνουν ότι ο αλγόριθμος “variable abstraction” σε συνδυασμό με το μοντέλο ελέγχου μπορούν να δώσουν αποτέλεσμα για την επαλήθευση του μοντέλου, το πρόβλημα της έκρηξης των καταστάσεων παραμένει για αρκετές περιπτώσεις.

6.2 Μελλοντικές επεκτάσεις

Η μελλοντική εργασία αποσκοπεί στην αντιμετώπιση των προβλημάτων που παραμένουν άλυτα. Για να αντιμετωπιστεί πιο αποτελεσματικά το πρόβλημα της έκρηξης των καταστάσεων νέες βελτιωμένες τεχνικές μείωσης και αφαίρεσης πρέπει να εφαρμοστούν στα μοντέλα των συστημάτων. Μια πολλά υποσχόμενη είναι λεγόμενη predicate abstraction τεχνική που χρησιμοποιείται για να επαληθεύσει ιδιότητες πεπερασμένων καταστάσεων και μη συστημάτων. Δίνοντας ένα συγκεκριμένο πεπερασμένης ή μη κατάστασης σύστημα και ένα σύνολο κατηγορημάτων, παράγεται μια αφαίρεση πεπερασμένων καταστάσεων και με αυτό το τρόπο το διάστημα των καταστάσεων μειώνεται.

Επιπλέον, ο αλγόριθμος variable abstraction θα είναι πιο αποδοτικός μετά από ορισμένες μικρές τροποποιήσεις στην υλοποίηση του. Εκτελώντας παράλληλα τα βήματα του τα οποία δεν σχετίζονται μεταξύ τους το αποτέλεσμα της επαλήθευσης θα παρεχόταν πιο γρήγορα.

Τέλος, το πρωτότυπο εργαλείο της στατικής ανάλυσης, πρέπει να επεκταθεί με πιο περισσότερους κανόνες οι οποίοι θα ανιχνεύουν σοβαρές παραβάσεις στον PLC κώδικα. Μετά από περαιτέρω έρευνες και έπειτα από συζήτηση με τους προγραμματιστές θα είναι πιο εύκολο να εξαχθούν πιο ουσιαστικοί κανόνες προς υλοποίηση.

Οι κοινότητες που ασχολούνται με την αυτοματοποίηση και τις τυπικές μεθόδους βρίσκονται ένα βήμα πιο κοντά στο να διασφαλίσουν την ορθότητα των PLC προγραμμάτων. Παρόλα αυτά χρειαζόμαστε ακόμα χρόνο ώστε να είμαστε απόλυτα σίγουροι ότι ένα σύστημα ελέγχου είναι ισχυρό και αξιόπιστο

Appendices

Abstract

Industry processes are evolving to the point where human interventions tend to disappear with the passage of time and as a result even a small mistake in these control systems can have catastrophic consequences. For that reason the need of developing robust, safe and reliable control systems is fundamental for control engineers. To guarantee the above both the hardware and the software have to be analysed to ensure that they fulfil the requirements.

The most popular control device in the process industry is the Programmable Logical Controller (PLC). PLCs are used all over the world for millions of industrial processes. Guaranteeing the safety of such a system is a challenging task for engineers. Testing and formal methods are used to check the correctness of a PLC program and ensure its safety.

The goal of this thesis is to improve the safety assurance of PLC programs and reduce the number of flaws in the software by integrating and applying static analysis and one formal method technique in the development process and at the same time hide any complexity from the developer.

The experiments and the methodologies used in this thesis have been applied to real-life PLC programs developed at CERN.

A Introduction

Technology with its rapid progress during the last decades has affected most aspects of our life. After the eighteenth century the industrial revolution marks a major point in history as it was the ground for significant economical, political and social changes in our society. Overall automation and automatic controls have brought and continue to bring benefits to human society.

Industrial automation has been evolving since its early start and represents an important element in the development of the industrial domain nowadays. Moreover, it plays an important role in solutions of improving our daily life. In particular, the average standard of living but also the rate of the population started to increase due to the development of new machines and technologies. By introducing the industrial automation human intervention has decreased and dangerous assembly operations were replaced with automated ones.

Industrial automation came to free humans from tiresome and long monitoring tasks while interacting with control systems. Control theory and control technologies were the elements to design systems with desired behaviours that can perform without the need of the human intervention.

In general control systems are divided in three main categories:

1. Supervision: in this layer the supervision tool that usually called SCADA (Supervisory Control and Data Acquisition) provides the interface with the process operator.
2. Control: this layer contains the control devices (PLC) where one or several input variables affect other variables.
3. Field: this layer is composed of sensors and actuators that take the information from the process and execute the logic given by the control devices.

However the modern control systems are divided in five parts instead of three by including also ERP (Enterprise Resource Planning) and MSE (Manufacturing Execution System). The thesis is focusing only in the control layer. The control system layers can be seen in Figure 13.

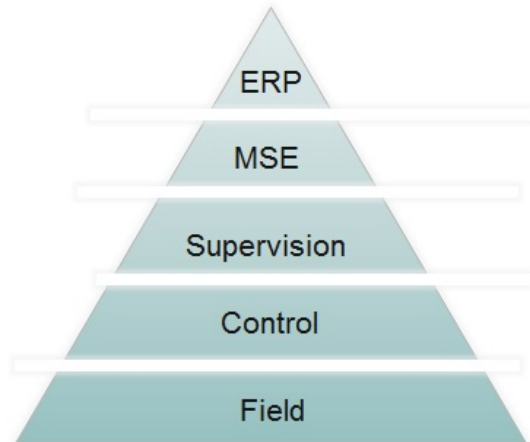


Figure 13: Control system layers

Since 1968, when the first programmable logic units were used to control systems at industrial processes, more critical systems have become more

automated. A potential failure or malfunction in any control system especially in a safety one, could cause severe damages. Therefore, one of the main needs in industrial automation is the ability to design reliable control systems that offer a robust level of safety and are compliant with their specifications. To guarantee this both hardware and software of the control system have to be analysed. A software failure in a control system can provoke irreparable damages not only to the human being but also to the environment and to the economy.

One of the many examples is the software error which provoked the accident of Mars Climate Orbiter [58], a robotic space probe launched by NASA on December of 1998 to study the Martian climate. A mathematical mismatch costed a \$125-million spacecraft to NASA. Bugs in such software can clearly have disastrous consequences. Another example of a software flaw that costed the life of 6 people is the bug in the control of the radiation therapy machine Therac-25. Due to a wrong computation the machine was giving massive overdoses of radiation to patients suffering from cancer.[61]

The goal of this thesis is to analyse programmable logic controllers (PLC) programs and improve their quality by focusing in the reduction of the number of bugs in the PLC software. PLC is the most widespread control device being used in the process industry and in the recent years is becoming popular also in the Safety Instrumented Systems, systems which are designed to ensure the safety and reliability of processes and are defined by IEC 61511 and IEC 61508 standards. In particular, the thesis focuses on the applicability of a formal method technique named model checking and of static analysis in the PLC software.

A.1 Context

As control systems are used massively not only in the industrial process but also in other domains such as airspace systems and nuclear installations, it is vital to prevent mistakes like the forenamed. Only few companies and research centres are involved in the particular problem and CERN (European Organization for Nuclear Research) is one of them.

The thesis has been performed in terms of my degree program on Informatics and Telematics in Harokopio University of Athens (Greece) and has been applied in a real world case at CERN. CERN which was established in 1954, is the biggest particle physics laboratory in Europe located on the French-Swiss border. It operates a complex of particles accelerators in order to perform the experiments.

People were always trying to discover as much as possible about the very first start of the universe and the way it was created. The LHC (Large Hadron Collider) (see Figure 14), currently is the biggest particle accelerator in the world; it is located 100 metres underground and has circumference of 27 km. The primary goal of this machine is to create the conditions that existed immediately after the Big Bang. The accelerator accelerates the particles to a very high energy and at a speed slightly less than the speed of light. Seven experiments (CMS, ATLAS, LHCb, MoEDAL, TOTEM, LHC-forward and ALICE) are located around the collider and each of them studies particle collisions from a different aspect and with different technologies. By analysing the collisions scientists aim to prove or disprove various theories of physics and understand the universe better.

In order to provide the optimised conditions for accelerators industrial processes are being used (e.g. cryogenics, cooling and ventilation). Among others the most widespread control device that is being used at CERN for the industrial processes and the needs of the experiments is PLC [59]. A PLC is a robust device that reads inputs from an input device (e.g. sensors,

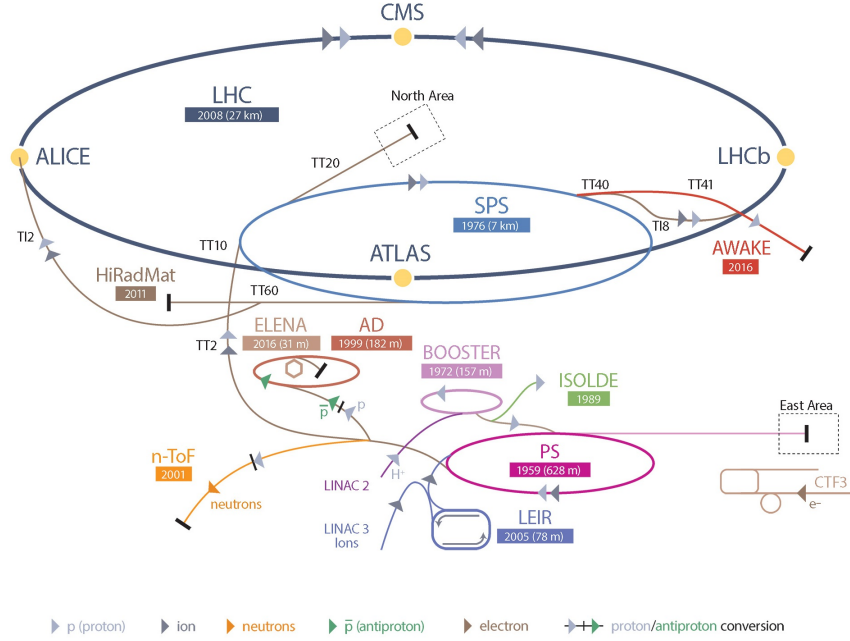


Figure 14: CERN accelerator complex [14]

selector, valves) process them and writes outputs to machines that can be controlled.

More than 1000 PLCs are maintained and developed for the industrial processes at CERN and almost 1/3 of them are maintained by the Industrial Control and Safety (ICS) group inside the beams (BE) department. More specific the Process Control Systems section (PCS) develops, implements and maintains protection and process control applications. Therefore, the topic of the thesis is related to the work of the mentioned section and it aims to improve the quality assurance of the PLC software by introducing formal methods which are recommended by the IEC 61508 standard and static analysis. For that reason, for the integration and the application of one formal method technique – model checking– and static analysis are used

A.2 Contributions of the Thesis and Motivation

Having a robust and reliable PLC program fully compliant with the specifications, but also a program that does not contain any bugs, is the main goal of the people who are involved in this project. However creating such a system is a very tricky and challenging task. The thesis, by contributing to the project, aims to bring the above techniques to the development process of the PLC program in order to guarantee compliance of the PLC program to the specifications but also to improve the quality of the code by detecting problematic code constructs in the early stage of development process and at the same time hide any complexity from the developer. To do so the thesis is divided in three main parts:

1. The integration of model checking in the PLC program development process by hiding the complexity fro the user and at the same time reduce the manual preparation.
2. The implementation of an abstraction technique to improve the performance of model checking in our PLC programs.

3. The development of a prototype for a static analysis tool for PLC programs.

B Background and Related Work

B.1 Introduction

This chapter overviews the techniques, the methods and the technologies that had been used for the application and the integration of model checking and static analysis to the PLC programs.

Some of the control devices are PCs, FPGAs (Field-Programmable Gate Array) and PLCs. PLCs have been an integral part for decades for most factory automation and industrial control processes and also at CERN. Section 2.2 describes the main characteristics of PLC control systems.

On the other hand, even though PLC is the most popular control device, there are not many applications of formal methods and static analysis yet in the field. These two techniques, model checking from formal methods and static analysis, which are used in this thesis are described in Section 2.3 and 2.4 respectively.

B.2 Programmable Logic Controllers

The first PLC was introduced in the late 1960s and was an outgrowth of the programmable controller [53]. The need for better, more robust, easy to configure and reliable control devices gave birth to the PLC. The Bedford Associates company came with the winning proposal of this electronic replacement named *084*. To maintain, develop, support but also sell this new product they started a new company named *Modicon*. Dick Morley, is one of the people who were working in this project and he is probably the “father” of the PLC [56].

Despite the fact that other electronic apparatus are more powerful, flexible and sophisticated, PLCs are the most popular and widespread device for controlling industrial processes. PLC is an operating electronic device that continuously monitors the status of devices connected as inputs and provides alternatives in interfacing with peripheral systems or devices.

B.2.1 PLC Hardware

PLCs use a programmable memory for the internal storage of instructions and the implementation of specific functions to control processes or various types of machines. Mainly the PLC receives inputs from both digital and analog instrumentation (e.g. sensors, valves, pumps) and transmits electronic and electronical signals to other electrical systems.

Execution schema: The I/O system provides the physical connection between the PLC and the equipment. PLC, by being a dedicated controller, is processing the same program over and over again. The execution schema or scan cycle as it is called, consists of two main steps:

1. The inputs are being scanned from the devices and the logic based on them is executed.
2. By executing the user program the outputs are triggered accordingly in a repeated loop called operating cycle which is executed very quickly (in the range of 1/1000th of a second).

The basic component of the PLC is the CPU (Central Processing Unit) which is controlled by the operating system and controls all the above steps of the operating cycle. However a scan cycle can be interrupted if an event (e.g. timer, hardware error, handler) triggers the execution of an interrupt handler. PLC interrupts can have different kinds and priorities. Moreover

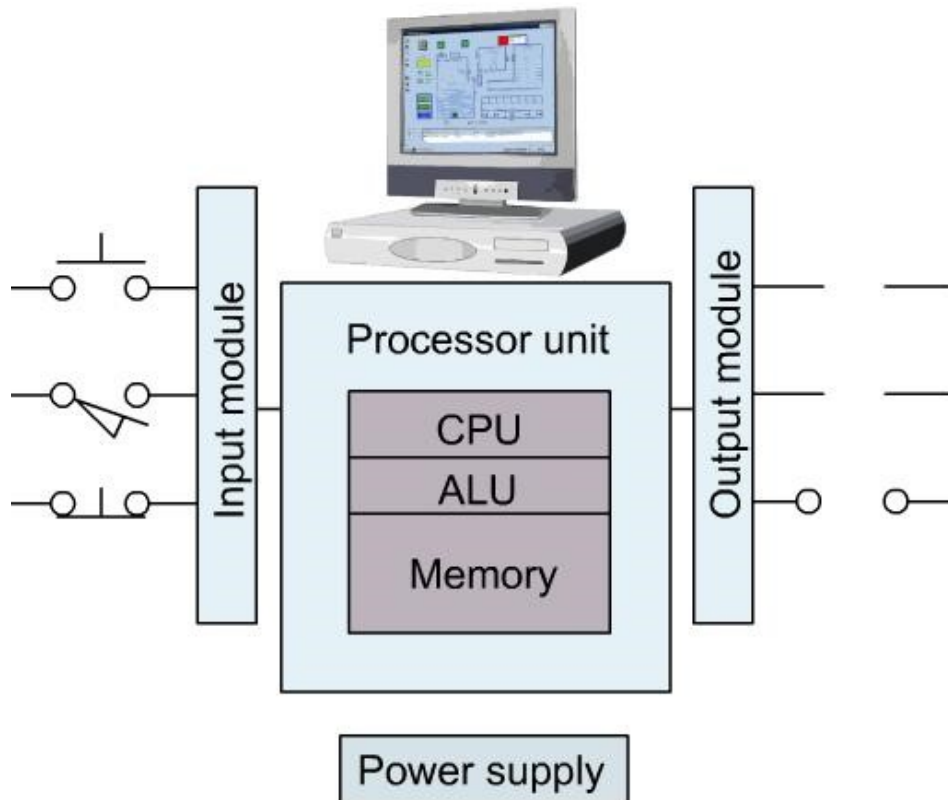


Figure 15: Basic PLC components

PLCs are equipped with power supply which drives the CPU, the I/O signals, the memory unit and some peripheral devices. An overview of the components that consist the PLC hardware can be seen above in Figure 15.

B.2.2 PLC Software

This thesis is based on Siemens PLCs, as they are the ones used for the industrial processes at CERN. However the differences between the Siemens PLCs and the PLCs provided by other manufacturers are minor. All the PLCs are based on the IEC 61131 standards in order to resolve topics related to control programming and to support the use of international standards in this field.

The third part of the IEC 61131 standard defines the different software resources that are required to build a PLC program. All PLC programming languages even from different vendors are following this standard by adapting on it or develop a slightly different language following the requirements and the recommendations that it provides. Five languages are defined by the IEC 61131 standard :

1. ST or Structured Text is a high level language syntactically similar to Pascal.
2. SFC or Sequential Function Chart is a graphical programming language based on steps, transitions and directed links between them.
3. IL or Instruction List is a low level language similar to assembly.
4. FBD or Function Block Diagram is a graphical language that can describe the function between input variables and output variables and it is based on logic gates.
5. The Ladder Diagram or LD is a graphical language based on the circuit diagram of the relay logic hardware.

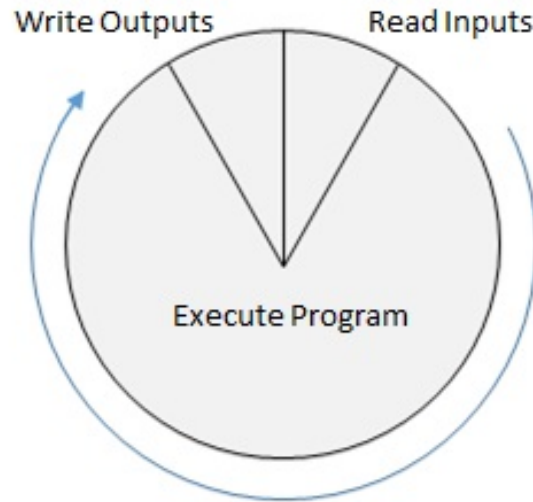


Figure 16: Schema of the cyclic scanning mode

However to program a Siemens PLC, Siemens developed its own versions of languages based on the above five ones. For this thesis only SCL (Structured Control Language) language is used and is the equivalent of ST language from the standard.

SCL Language: Provides the following five block types available [1].

1. OB or Organization Blocks form the interface between the CPU operating system and the program. OBs are the entry points of the system.
2. FB or Function Blocks are logic block with static data that their parameters can be accessed at any point in the program.
3. FC or Functions are logic blocks without memory so the calculated values must be processed after the function is called. These blocks contain executable code.
4. DB or Data Blocks are areas responsible for the storage of the data. There are two types of data blocks: *shared* ones that can be accessed by any block type and *instance data blocks* that are assigned to a specific FB call. Data blocks do not contain executable code.
5. UDT or User-defined data types are structured data defined by the user and they are handled as if they were blocks.

Listing 2: Example of ST code

```
FUNCTION_BLOCK FB100
VAR_INPUT
    a : BOOL;
END_VAR
VAR_TEMP
    b : BOOL;
END_VAR
VAR
    c : BOOL;
END_VAR
BEGIN
    b := NOT a;
    c := b;
```

```

END_FUNCTION_BLOCK

DATA_BLOCK DB1 FB100
BEGIN
END_DATA_BLOCK

ORGANIZATION_BLOCK OB1
VAR_TEMP
    info : ARRAY[0..19] OF BYTE; // reserved
END_VAR
BEGIN
    FB100.DB1(a := FALSE);
    Q1.0 := DB1.c;
END_ORGANIZATION_BLOCK

```

Listing 2 above shows an example of ST code. This example code defines a function block (FB100) with three variables (a , b , c). There is an instance data block (DB1) defined for FB100. In the organization block OB1, the FB100 is called using the instance data block DB1 with input parameter $a=false$. Then the c variable of this instance is assigned to the output Q1.0.

B.2.3 UNICOS

UNICOS (UNified Industrial Control System) is an industrial control framework developed at CERN. The UNICOS framework can generate PLC code for PLC-based control systems written in Siemens SCL. The experiments presented in the thesis use PLC programs from the UNICOS library.[11]

C Model Checking and Static Analysis

C.1 Formal Methods

As it was mentioned before, mistakes in the control systems can put in danger not only the economy and the environment but also the human. For this reason in software but also in hardware design of complex systems usually more time is spent on the verification of the system than on the construction. In some cases the traditional prevention techniques such as testing are not sufficient to reach and verify the desired safety level. However formal methods can help to assure the safety of the system. Formal methods are defined by several authors and below is one of many definitions of them:

“Formal methods are mathematical approaches to software and system development which support the rigorous specification, design and verification of computer systems. The use of notations and languages with a defined mathematical meaning enable specifications, that is statements of what the proposed system should do, to be expressed with precision and no ambiguity.” [18].

C.1.1 Model checking

Model checking is a formal method technique which was developed by Clarke and Emerson. Queille and Sifakis independently discovered a similar verification technique shortly thereafter in early 1980s.

“Model checking is a method for formally verifying finite-state concurrent systems. Specifications about the system are expressed as temporal logic formulas, and efficient symbolic algorithms are used to traverse the model defined by the system and check if the specification holds or not.” [13]

The model checking process consists of three steps:

1. Formalization of the requirement to be checked.
2. System modelling.
3. Execution of the model checker algorithm.

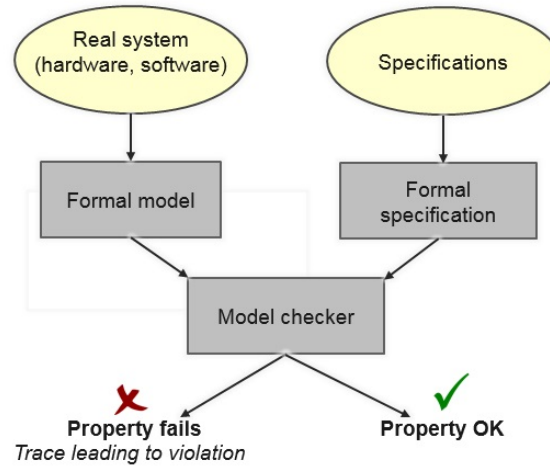


Figure 17: Model checker process

Formalization of the requirement to be checked: Engineers when design a system constantly face the problem of faulty design requirements. The cost of errors in requirements is often high as it will probably lead to an incorrect system behaviour something that they do not desire especially for a safety system.

In order to produce correct and precise requirements the use of formal methods is mandatory. In model checking techniques the requirement or property is expressed by using a temporal logic as a property specification language. Temporal logic is any system of rules and symbolism for representing, and reasoning about, propositions qualified in terms of time [60].

System modelling: Once the requirements to be verified are described in a temporal logic the construction of the formal model of the system is another necessary step. Models of systems describe and represent the behaviours of the original system usually in an accurate way.

Execution of the model checker algorithm: Model checker in order to execute the algorithm needs as inputs the formal characterization of the requirement to be checked and the model of the system that was built. Once those two inputs are provided the model checker will run the algorithm and will proceed to the analysis phase. At that point is going to give a result about whether the property is satisfied or there was a violation in the property. In the later case scenario model checker may provide a counterexample to the user that contains uncover errors in design by indicating how the model would reach the undesired state.

Advantages and disadvantages of model checking

Summarizing the main advantages of model checking techniques are:

1. It is an automatic verification method.
2. If the requirement is not satisfied, a counterexample that shows why the requirement does not hold on the model is produced by the model checker.

3. In order to guarantee that the property holds on the model, model checking explores all the possible combinations of the state space.
4. Temporal logic can express many of the properties that are needed.

The main disadvantages of model checking techniques are:

1. State explosion - sometimes the number of the possible combinations can be enormous and model checker is not able to give a result.
2. Difficulty in using temporal logics specifications
3. Complexity of building formal models.

To automate the verification process all the algorithms of model checking are implemented in verification tools. Some of the most popular verification tools are: UPPAAL, BIP, SPIN, KRONOS and NuSMV which is the one that is going to be used in this thesis.[10]

C.1.2 Static analysis

Static code analysis is an analysis technique that examines a software program without actually execute the program. Static analysis is performed by an automated tool and is similar to code review or program comprehension which is done by the developer. The main benefit of static code analysis is the early detection of potential bugs in the development cycle and the improvement of the program quality. The goal of static analysis is to find potential defects even though they might not cause failures. While it can often be difficult to test the whole programs due to the size of software projects, static analysis tools can be used to analyse the on-going project for violations as they are being created [29]. However, this is not to say that static analysis can be sufficient for testing the code, therefore dynamic analysis can not be ignored as it is also an important part of the software testing process [17].

Nowadays static analysis tools are widely used and various of tools are available for the established programming languages (e.g. Java, C++, C, Python). In order to analyse source code there are several techniques and structures such as data flow analysis, control flow graph, taint analysis, often derived from compiler technologies. The thesis is focusing in the *Rule-based AST* analysis of PLC programs.

Rule-based AST Analysis: Rule-based AST analysis works by analyzing the abstract syntax tree (AST) of a program based on some configured defined rules.

The rules developed on top of the AST usually can detect in the source code violations like the followings:

1. Naming conventions: A naming convention is a set of rules for choosing the character sequence to be used for identifiers in order to improve the readability and maintainability of the code. Some examples are given below:
 - Names of Boolean variables should start with B
 - The length of a variable name has to be between three and fifteen letters.
2. Code smells: Code smells are code patterns that frequently cause problems. The might not cause fatal problems to the code but still should better be avoided in order to maintain the quality of the code and make it understandable. Some examples are given below:
 - Duplicated code
 - Dead or unreachable code

- Long method
 - Long parameter list
 - Loop variable read after loop
3. Problematic task interleaving and race conditions: Multiple and severe problems can be occurred due to multi-tasking issues such as for example race conditions. An example of a race condition is the wrong usage of the stop flag that can be overwritten by the cyclic process and as a result the process will not stop.
 4. Dynamic statement dependency: There are defined constraints on the execution order of some related function calls. An example of a violation like this is when a file open is not followed by a file close operation.

Advantages and Disadvantages of Static Code Analysis: Static code analysis tools look for patterns, defined to them as rules, which can cause code quality problems and malfunction of the program. But like every other technology, static analysis has its set of advantages and disadvantages [28].

Advantages:

1. Analysis of the code without its execution.
2. Ensure that certain rules already specified are respected without any manual intervention.
3. Help in the early detection of bugs in the program.
4. Help to maintain the code quality.

Disadvantages:

1. High number of false positives; the tool will often produce warnings for a possible violation that in reality does not represent an issue.
2. Will not detect always the configuration issues as they are not represented in the code.
3. Can not check the correctness of the behaviour of the program.

D Related Work

This section presents the related work on formal verification methods applied to PLC programs. As it was mentioned before the thesis aims to assure the safety and the quality of the PLC programs by applying and integrating model checking and static analysis in the development process. Around 20 years ago industries with critical systems started to use formal methods and nowadays more and more of them apply them to verify the correctness of the developed system. Despite the fact that different industries have different goals and represent different projects and ideas, static analysis and algorithmic verification techniques such as model checking seems to be a common practise for them.

An example that proves the applicability of formal methods in real life system is the air traffic control system of UK. To handle the increasing traffic, the UK upgrade its air-traffic management system by developing the CCF (Central Control Function) which is handled by several systems. One of them, the CDIS (CCF Display Information System) used formal methods in order to be designed and verified.[46]

The usage of formal methods for PLCs has been studied for many years and the rest of the section is divided into two parts: the related work in model checking and the related work in static code analysis, both for PLC programs.

D.1 Model checking applied to PLC programs

Model checking still is not used in the industry even though it is the most popular algorithmic formal verification technique and there are several approaches of its application in PLC programs. The lack of applicability is mainly because all of the approaches translate the PLC program into the input language of an existing model checker and thus suffer from certain problems. Many studies where model checking was the selected method to verify PLC programs can be found in literature and they can be divided into 3 groups:

1. Targeting the PLC language: most of the model checking techniques target to PLC programs written in IL [9, 33, 51, 52, 43] and SFC [4, 27]. A few researches have been applied also to the FDB [3, 64], the ST language [5] and the Ladder Diagram (LD) language [36]. Finally there is an approach of a tool to verify PLC programs written in all languages from the standard IEC 61131-3 [48, 25].
2. Targeting the specification: paradoxically, there are not many approaches devoted in the development of accurate properties to verify the given model. A proposed approach is the creation of temporal logic based on UML techniques [22] [49]. Another proposed approach [8] is based on defined patterns that express the properties of interest and are translated automatically into temporal logic formulas.
3. Targeting the modelling and the specification constraints : some authors follow the approach of applying model checking to small examples without apply any reductions on the model [35, 9, 42, 44, 50, 54, 7, 24, 31] while in other cases there is a limitation in the used property when the abstraction is applied [31][7]. Moreover in many researches the PLC program is not transformed automatically to a formal model [3, 4, 7, 42, 63, 54].

D.2 Static analysis applied to PLC programs

Static analyses solutions have been also proposed by some authors and the several researches that have been made from small or medium enterprises in order to evaluate the importance of static analysis tools for software applications show that it is a valuable contribution in the software community [37][2][65][23].

In the PLC domain static code analysis tools are rarely available even though such tools are widely used other domains. A great deal of research and attempts to implement static code analysis tools have been made by universities [45] and small companies [26] but the tools are not available. The few exceptions are the tools PLCChecker from Itris, Codesys Static Analysis from CoDeSys and Arcade.PLC. which is an academic tool developed in RWTH Aachen University [6]. All of these tools, will be analysed below in Chapter 4.

E Technologies used for this thesis

E.1 Apache Subversion

Apache Subversion (often abbreviated SVN) is a software versioning and revision control system distributed as free software under the Apache License. At CERN is the mainly used repository to maintain current and historical versions of files such as source code and documentation for PLC programs.

E.2 Jenkins

Jenkins is an open source automation server written in Java. It support revision control tools including SVN and Subversion in order to ensure that the software of a project can be released at any time. With a continuous integration, Jenkins help to automate the whole software development process.

E.3 Spoofax

Spoofax is an Eclipse-based platform for efficient development of textual domain-specific languages (DSL). Its language workbench includes tools and high-level meta languages for defining syntax, name bindings, types and tree transformations, among others [34].

E.4 Xtext

Xtext [19] is an open-source framework for developing programming languages and domain-specific languages (DSLs). Unlike standard parser generators, Xtext provides a full infrastructure as it generates not only a parser, but also a class model for the abstract syntax tree, as well as providing a fully featured, customizable Eclipse-based IDE.

F Integration of Model Checking in the development of PLC programs

F.1 Introduction

Four years ago a project was started at CERN to improve the quality of PLC programs and reduce the number of bugs in the code. An error in a control software can result in loss of reputation, expenses and even injuries. In order to avoid it, the project's main goal is to ensure correctness of the software used in the control systems by using formal methods. Formal methods are mathematically based techniques for the specification, development and verification of software and hardware systems. This chapter presents the first contribution of this thesis which is part of this project and aims to fully integrate the methodology in the development process of PLC software which is described below. The first part of the project is the integration of model checking into the development process of control engineers. The structure followed in this chapter, consists of the problem description, the methodology, the contribution and the conclusions from the conducted experiments.

Among several formal verification techniques, model checking is so far the most appropriate for our case, not only because it is possible to provide automated verification for PLC programs but also because it has the potential to hide complexities from the developer. There are not many tools available for verification of PLC programs written in SCL language. At CERN a tool named PLCverif [16] was developed in order to apply model checking and do verification for PLC programs written in SCL (Siemens language equivalent to ST from IEC61131). The tool is developed using Xtext [19] and Eclipse Modelling Framework [55] technologies. In order to verify PLC programs the tool generates automatically formal models out of PLC programs, translates the user requirements to its formal characterisation and provides a report to the user with the results of model checking.

The methodology [21] implemented by the tool consists of the following steps (see Figure 18):

1. The PLC code is translated into an intermediate model (IM) which is based on an automata network model consisting of synchronised automata.
2. The specified requirements provided by the user are translated in temporal logic formulas (e.g. LTL, CTL) [38] through a set of specification patterns.
3. After the PLC program is translated to the IM, reduction and abstraction techniques are applied to the IM to reduce verification time.
4. The formal model of the PLC code is translated into the input language for the model checkers. Several model checkers can be introduced in the methodology (e.g. nuXmv).
5. The model checker is executed providing the generate specific formal model and the temporal logic formula to verify the requirement against IM.
6. After the verification is performed, the results of the analysis are presented in a report.

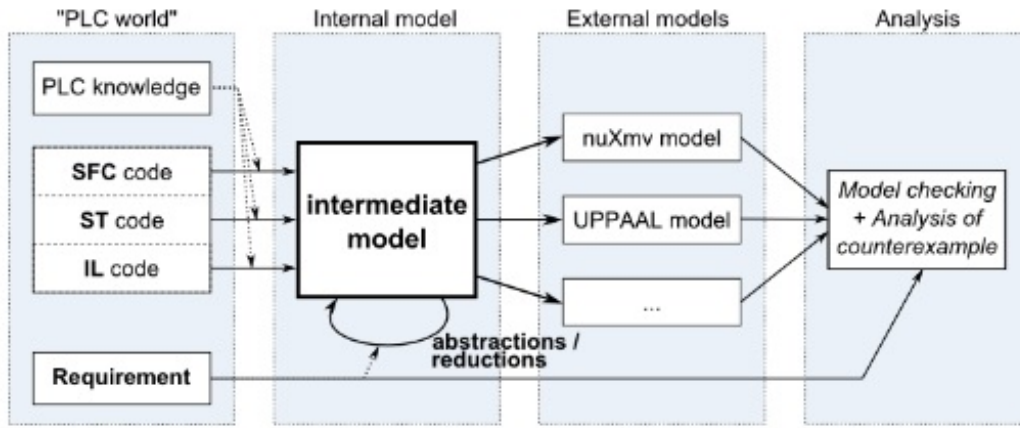


Figure 18: Methodology overview

The user has the ability to configure the maximum time given that the tool will spend to verify each verification case (TO) (see Figure ??). This feature was created to avoid too long verification times. When this happens maybe a different requirement or verification strategy should be applied. The user work flow consists of the following four steps [16]:

1. Write the PLC code in the provided SCL editor or import if it already exists (see Figure 19).
2. Create the requirement to be verified (see Figure 20).
3. Execute the verification (see Figure 21).
4. Analyse the verification report (see Figure 22).

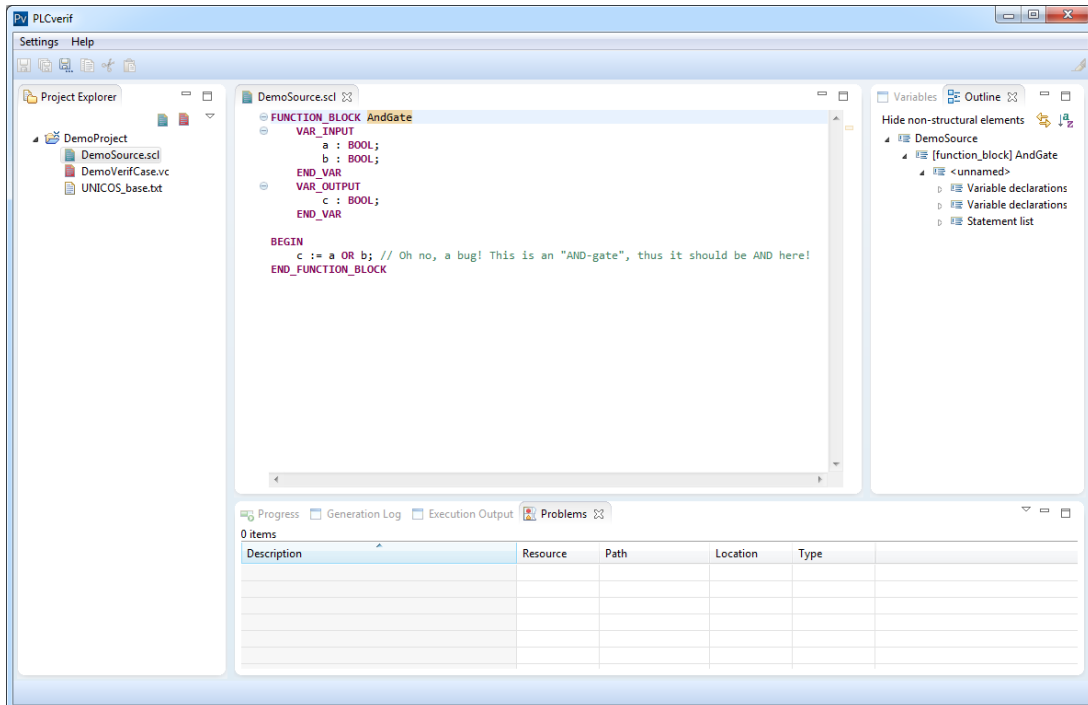


Figure 19: SCL editor

In Figure 19, the SCL editor is presented. The user can import an existing project and even modify the code, or create a new one by using the editor. ON the left, in the project explorer, the user can see the existed projects in the workspace.

▼ Requirement

The requirement to be checked should be defined in this section.

Requirement pattern:

5. State change during a cycle: If {1} is true at the beginning of the PLC cycle, then {2} is alw

Pattern params:

[1] FoMoSt_aux = true AND AuAuMoR = true AND ManReg01[8] = false

[2] AuMoSt = true

5. State change during a cycle: If **FoMoSt_aux = true AND AuAuMoR = true AND ManReg01[8] = false** is true at the beginning of the PLC cycle, then **AuMoSt = true** is always true at the end of the same cycle.

Figure 20: Requirement pattern

In Figure 21, the interface to create the verification case (user-requirement) is presented. The user can fill some general informations for the requirement, such as for example the ID, the name or the description. Then in the requirement field (see Figure 20), the user specifies the requirement pattern from a list with predefined patterns and fills the parameters which represent the PLC program variables whose behaviour needs to be checked.

The screenshot shows the PLCverif application window. On the left is a 'Project Explorer' showing a 'DemoProject' with files 'DemoSource.scl', 'DemoVerifCase.vc', and 'UNICOS_base.txt'. The main area is titled 'Verification Case (Demo001)' and contains several sections: 'General' with fields for ID (Demo001), Name (If A is false, C cannot be true.), and Description (If A is false, C cannot be true. As this function block models an AND-gate, if any of the inputs (A or B) is false, the output should be false too. The requirement is based on the documentation of the function block and the following Jira case: https://icecontrols.its.cem.ch/jira/browse/UCPC-1111); 'Requirement' section with a dropdown for 'Source code' (DemoSource.scl) and a 'Refresh variables' button; 'Advanced configuration' section; and 'Verification' section with a 'Tool' dropdown (nuXmv). On the right is a 'Variables' panel with a filter and a list of variable names: instance.a, instance.b, and instance.c. At the bottom is a table with columns: Description, Resource, Path, Location, Type, and a row for '0 items'.

Description	Resource	Path	Location	Type
0 items				

Figure 21: Verification case

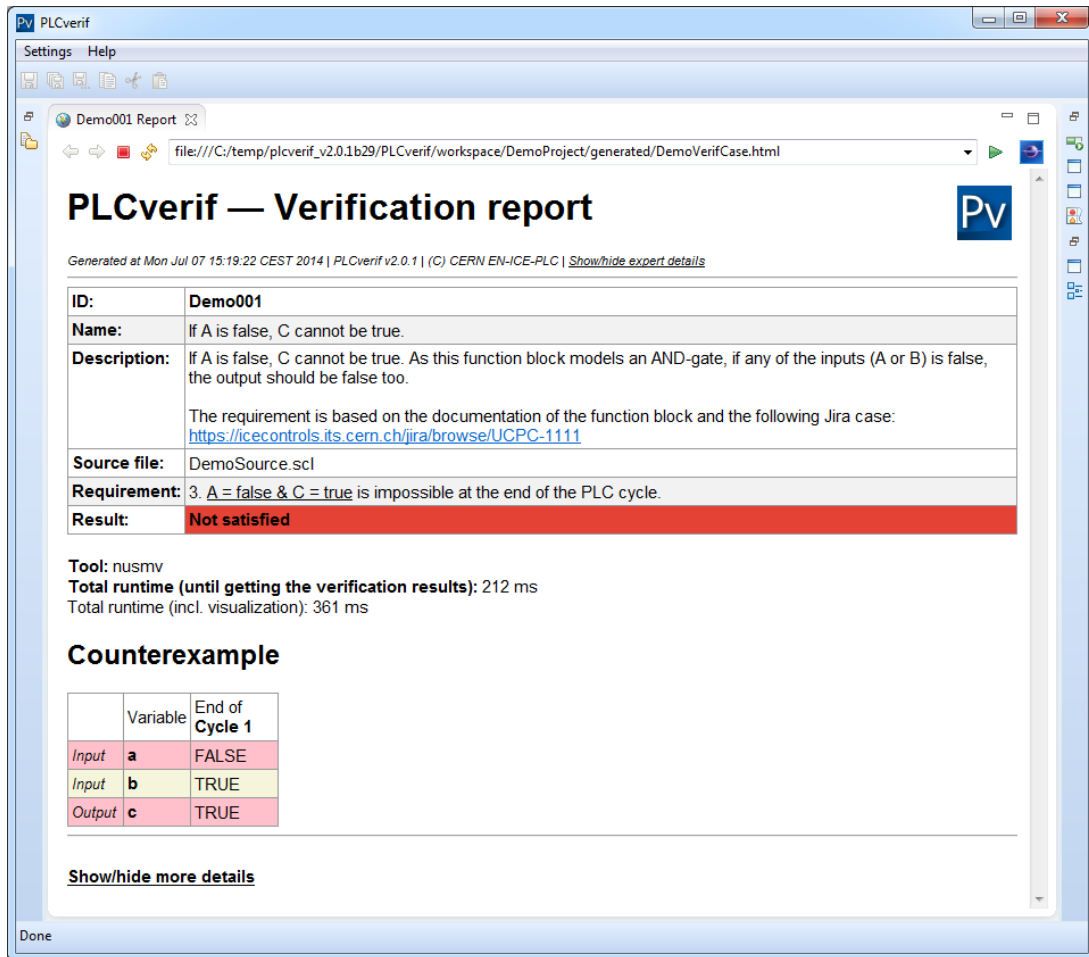


Figure 22: Verification report

In Figure 22 the report produced by the model checker is presented. The report contains some general information for the requirement such as for example the description or the source file that its based on and of course the result of the model checker. If the result is not satisfied like in this specific example, then the counterexample is presented in the report. In the counter example, we can see the traces that could provoke a violation of the requirement in the PLC program. In red color are the variables that are involved in the requirement of the verification case, while in yellow are the input variables.

So far every time the developer wanted to make a modification on the PLC code all the requirements needed to be checked again one by one. Moreover when more than one verification case needs to be created and checked, they have to be executed manually. The problem is that the time that the model checker needs to obtain a result is not known and sometimes it can take days until a result is produced. The reason of this, is that models of real-life PLC programs usually face the problem of a state space explosion. The first task of this thesis, that it is introduced below, tries to overcome these two problems. Three technologies are applied for this contribution: PLCverif, Subversion (SVN) and Jenkins.

The structure of this chapter is the following:

1. In Section 3.2 the contribution for the first part of the thesis is presented.
2. In Section 3.3 the benefits of the integration of model checking in the development process are presented.

F.2 Contribution

PLC programs used at CERN are mainly developed in the UNICOS framework. UNICOS provides a set of objects, the so called Baseline objects, that represent physical devices (e.g. valves, sensors, pumps, etc.) in the PLC programs. A PLC program developed with the UNICOS framework, instantiates and connects these objects to provide the control of a full plant. In order to verify the correctness of these systems they use, apart from testing, the PLCverif tool to apply model checking.

Some PLC programs use some blocks which are provided by the PLC device itself (hardware) and contain utility and building functions (e.g. TON [21]). The source code for these functions is not provided by the PLC programmer but is implemented by PLCverif in a separate file. As PLCverif currently does not have a complete grammar, it cannot handle some features that exist in each object's code, so some assignments and commands which in reality do not affect the behaviour of the object have to be removed or modified. Currently, the tool supports only one source file to be analysed at a time, however the utility functions described above and another block which is a common base for all the objects, are placed in different files. Therefore, as several files cannot be analysed at the same time by PLCverif, they need to be concatenated into one file. By using SVN and Jenkins, the thesis aims to simplify the procedure for the PLC developer. By developing two scripts the target files are concatenated and all the necessary modifications are made in the code. The approach followed can be divided in three steps (see Figure 25):

1. The user modifies a UNICOS Baseline Function Block which corresponds to a .SCL file or adds/modifies a verification case (see figure 21) and commits it to SVN.
2. This action triggers Jenkins to perform the following sub-steps:
 - pre-process of the .SCL files by using Python scripts.
 - execute PLCverif to perform model checking.
3. Once all verification cases for each modified UNICOS Baseline Function Block have been analysed, Jenkins sends a verification report via email to the user. In Figure 23 we can see that there was no violation in the verification cases after the modifications to the source code of the PLC program. Details are provided regarding the name of the project, the date of the build, the build duration, the kind of changes that were made to the code and the files which were tested after them. Moreover in Jenkins there is a panel where the user can see the current status for all the projects, in Figure 24 an example of the overview panel is presented. In green color are the projects where their build was successful, which means that PLCverif provided a "satisfied" result, while in orange are the projects that contain verification cases where PLCverif was not able to provide a result for the given TO, or the verification result was "not satisfied".

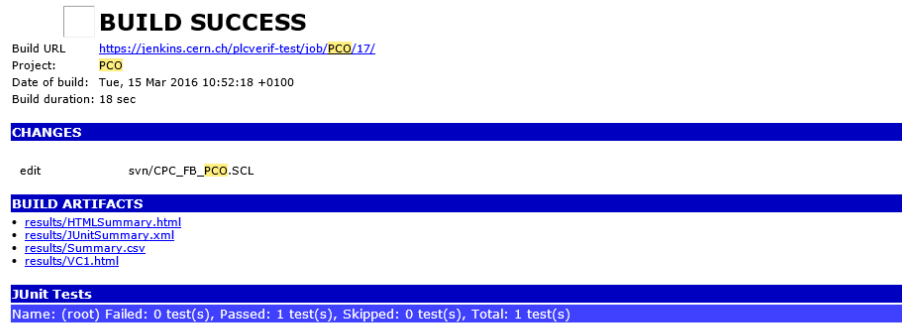


Figure 23: Example of an email report provided by Jenkins to the user after modification on the PLC source code

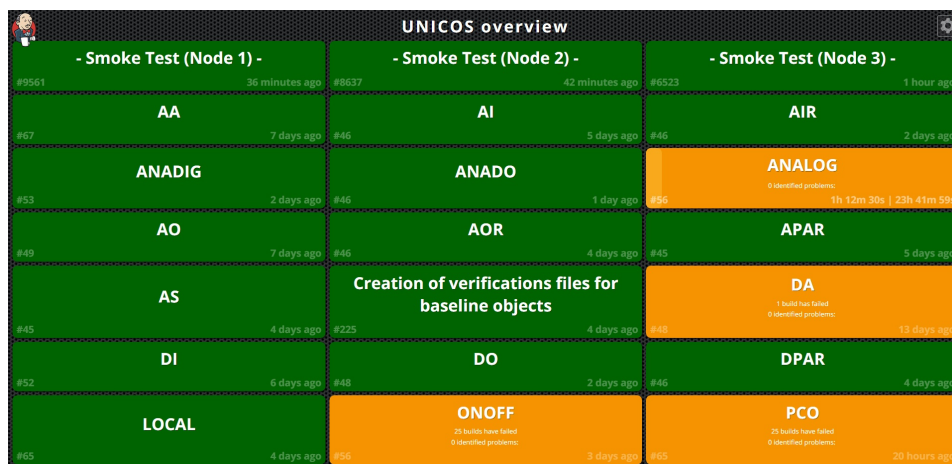


Figure 24: UNICOS overview panel

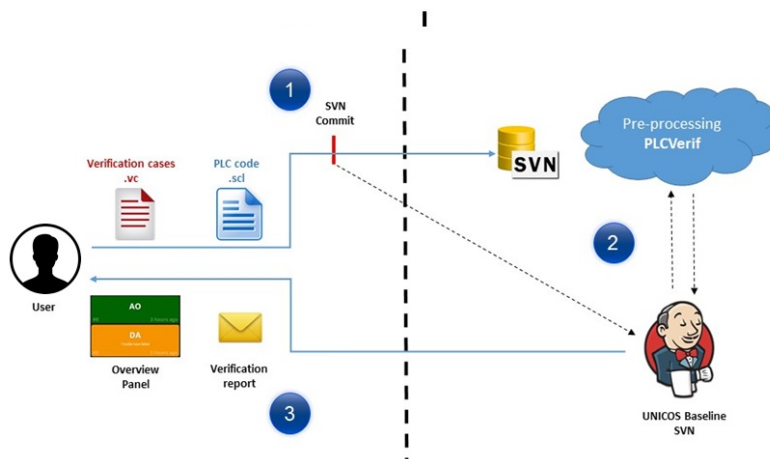


Figure 25: High level overview of the approach

In order to implement the described approach, configuration files and two Python scripts had to be developed.

Configuration files: The configuration files are being used to define the replacements and to take care of the files that need to be concatenated during the execution of the script. More specifically these types of configuration files were created in JSON format. JSON (JavaScript Object Notation) uses JavaScript syntax and its format is text which makes it easily be used as a data format by any other programming language. The configuration files created are the following:

- Configuration files for each Baseline object.
- A configuration file called “replacements” which is common for all the Baseline objects. The replacements file contains the keywords that should be replaced in the concatenated source code.

Each JSON file (see Listing 3) that corresponds to a Baseline object is consisted of:

- The paths of the .SCL files in SVN to be concatenated,
- The path of the replacements.json file,
- The path of the target file where the concatenated source code with the replacements will be placed.

Listing 3: JSON file for the ONOFF baseline object

```
{
  "FilesToConcat": [
    "../.. /svn/CPC_FB_ONOFF.SCL",
    "../.. /svn/CPC_BASE_Unicos.SCL",
    "builtin_functions_and_unicos_workarounds.txt"
  ],
  "ReplacementsFile": "ConfigFile/Rep/replacements.json",
  "targetFile": "../ONOFF/CPC_FB_ONOFF.SCL"
}
```

Python scripts: The *first* script, has to return all the configuration files, place them as an argument in a function and then call the second script. The *second* script, once it is called, processes .json files by concatenating the source code of the target objects. Then it makes the necessary replacements by using the replacements.json file and finally creates a file which can be fed to PLCverif.

More specifically the user workflow consists of the following steps (see Figure 26):

1. The user makes a commit to SVN.
2. That commit triggers Jenkins to pre-process the PLC code to be verified by using the first script.
3. The script opens the .SCL files and a .txt file that contains building functions that do not exist in the original code and had to be implemented.
4. The files are concatenated through the execution of the second script and the needed replacements, which are defined by regular expressions in a .json file, are executed.
5. The result is stored in a new file created by the second script.
6. The file is committed to a SVN repository.
7. Finally that triggers Jenkins to execute a second step; PLCverif to perform model checking.

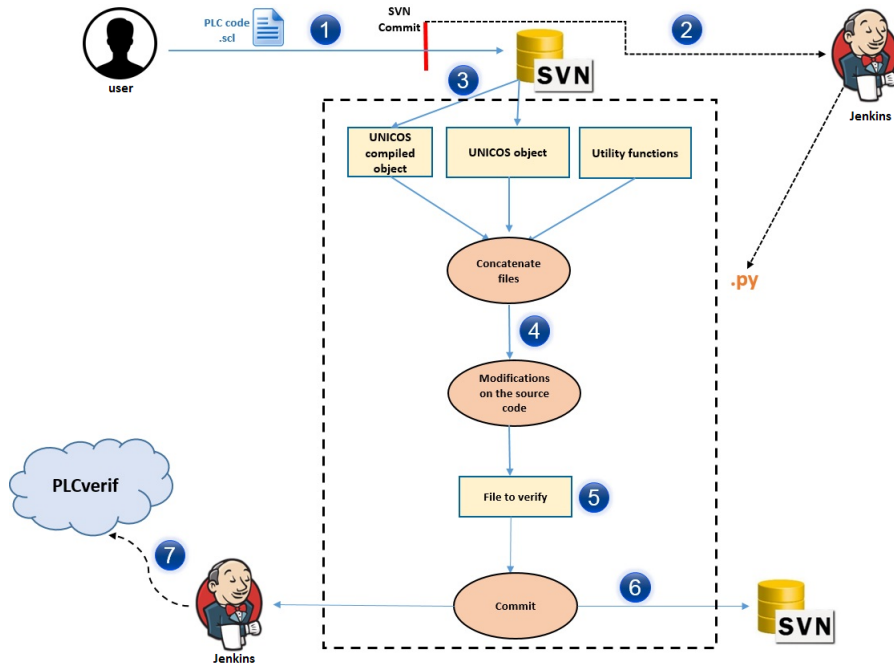


Figure 26: User workflow

F.3 Conclusions

The methodology has been applied to 19 UNICOS baseline objects consisted of few hundred lines of source code and different kinds of requirements have been verified. From now on the user only has to commit to the SVN repository the new or modified .SCL file or the new created verification case. Jenkins, by being triggered by the commit, takes care of the rest. Once a result for the newly created verification case is obtained or once a violation is triggered due to a modification on the source code, Jenkins is going to inform the PLC developer by sending him an email with the verification results. By using the script, the process is automated now and the complexity is hidden from the developer. Therefore the benefit is that the developer has to spent much less time in the verification process. Now the developer can add and verify more efficiently requirements for a PLC code. Moreover there is no need for manual preparation of the PLC source files.

G Implementation of an Abstraction Technique

G.1 Introduction

In chapter F the problem of the state space explosion [12] [30] was mentioned. As the number of state variables in the system grows, the size of the system state space increases exponentially. The state explosion, occurs when the model checker tries to verify a system with many components making transitions in parallel where the number of possible combinations is too big. PLC programs usually contain many variables of different types including boolean, integers and float variables. Therefore, the resulting formal models have a huge state space. Even though in the previous section the methodology to automate model checking and hide complexity from the developer was described, the problem of the state space remained unsolved.

In figure 18 it can be seen that after the PLC code is translated to an intermediate model (IM) reduction techniques are applied to it. The reduction techniques applied to our models so far are [15]:

1. Cone of Influence (COI) : This reduction technique removes all the irrelevant variables, assignments and guards from the IM.
2. Rule-based reduction : This reduction technique eliminates states and variables, removes empty branches and merges transitions and variables.
3. Mode selection: With this reduction technique parameters with a fixed value can be replaced by a constant value.

However, these reduction techniques, are not always sufficient in order to overcome the state space problem and have a verification result for the requirement. In order to deal with this problem and improve the performance of the verification, a new abstraction technique was described in a PhD Thesis [20] but it has not been implemented so far. By using this algorithm, we have more possibilities to verify certain properties, for example simple invariants such as *if an input is set, a certain output has to be set as well*. An invariant is an expression that has to always be kept satisfied by the model checker.

High-level idea of the abstraction technique: The abstraction algorithm, called *“iterative variable abstraction”*, is divided into 5 steps that can be seen in figure 27. When a result can not be obtained by PLCverif while trying to verify a property in the original model, the main idea of the algorithm, is to create abstract models (AM) out of it. By using this technique abstract models are created iteratively while trying to prove that the safety property is satisfied in the AM'_n . AM is an over-approximation of the original model.

How to create an AM? Some of the variables included in the original model have to be extracted from the variable dependency graph of the model and be replaced with non-deterministic values in order to create the AM. As these new variables have no dependencies with other variables, the COI algorithm will be able to reduce the model by eliminating more variables. The variable dependency graph and the concept of creating AM based on it will be explained below in the example of applicability.

Thanks to this abstraction technique:

- the size of the PSS (Potential State Space) of the abstract model is smaller as it contains less variables,
- usually a bigger range of possible behaviours is represented in comparison to the OM (Original Model).

The Potential State Space, represents the number of the values that a process can take. Based on the variables of the model that are represented in its declaration part, the number of the possible combination of these variables forms the PSS. For example in a PLC code that consists of 3 boolean variables the size of the PSS is 2^3 .

The algorithm uses two kind of properties: reachability properties and safety properties. Reachability properties ($EF(\gamma \& \theta)$) state that a particular situation can be reached, while safety properties express that a specific event should never occur under specific conditions (consider a safety property ($AG(\alpha \rightarrow \beta)$)). By applying model checking with the safety property if p holds on AM'_n , it implies that both the abstract model and the original model are compliant with the specification, as a bigger range of possible behaviours is represented with the abstract model. In the opposite case, where the property p does not hold on AM'_n , a counterexample c is generated by the model checker.

Counterexamples can be complex as they contain variable values from different PLC cycles. To verify that the property holds on the OM requires analysing the counterexample c through the algorithm (step 3 & step 4) and define if it is real or spurious. When there are false positives in the verification phase or when only one of the properties is not satisfied, then the counterexample can be considered as spurious.

The structure of this chapter is the following:

1. In Section 4.2 the implementation of the variable abstraction algorithm is described.
2. In Section 4.3 the results of the conducted experiments are presented.
3. In Section 5.4 the analysis of the conducted experiments and the extracted conclusions are presented.

G.2 Contribution

The following paragraphs describe in more detail the steps of variable abstraction and their implementation in a script. The script is implemented in Python and the user can execute it from the command line by entering as an argument the path of the verification case. The script calls iteratively the command line version of PLCVerif [32] and contains all the steps of the algorithm. The .vc files are in an XML format (see Listing 4) and in order to parse them XPath [62] and regular expressions are used.

Listing 4: A verification case example

```

1 <?xml version="1.0" encoding="ASCII"?>
2   <requirement patternString="1. If {1} is true at
   the end of the PLC cycle,
3   then {2} sholud always be true at the end of the
   same cycle.">
4     <settings tool="nuxmv_ic3" >
5       <inputVariableFqns>instance/g</
         inputVariableFqns>
6       <inputVariableFqns>instance/h</
         inputVariableFqns>
7       <inputVariableFqns>DB_DI_all/DI/*/HFPos</
         inputVariableFqns>

```

```

8      <inputVariableFqns>DB_AI_all/AI/*/HFPos</
      inputVariableFqns>
9      </settings>
10     <parameters>b</parameters>
11     <parameters>a</parameters>
12   </requirement>
13 </ch.cern.en.ice.plcverif.
      verifdescriptor:VerificationCase>

```

Symbol correspondence:

- OM' : the result of applying the property preserving reductions to the generated model (these algorithms aim to preserve only those behaviours of the system that are relevant from the specification point of view.)
- OM'' : the result of applying the property preserving reductions for the reachability property r .
- δ' : distance between variables in the variable dependency graph for OM' .
- δ'' : distance between variables in the variable dependency graph for OM'' .

When verifying a property on the OM' in PLCverif(step 0), the model checker might not be able to give a result because a TO (time out) is obtained. In this case the variable abstraction algorithm is triggered in order to verify the same property. Due to this, the initial step is not included in the five steps of the algorithm.[20]

Implementation and description of algorithm's steps

1. Checking the original safety property (p) on an abstract model (AM').
2. Checking the safety property (q) on the same abstract model (AM').
3. Checking the reachability property (r) on the original model (OM'').
4. Checking the reachability property (r) on an abstract model (AM'').
5. Extracting an invariant from the counterexample to include information about the process obtained in step 2 if the number of potential invariants (m) is ≤ 10 .

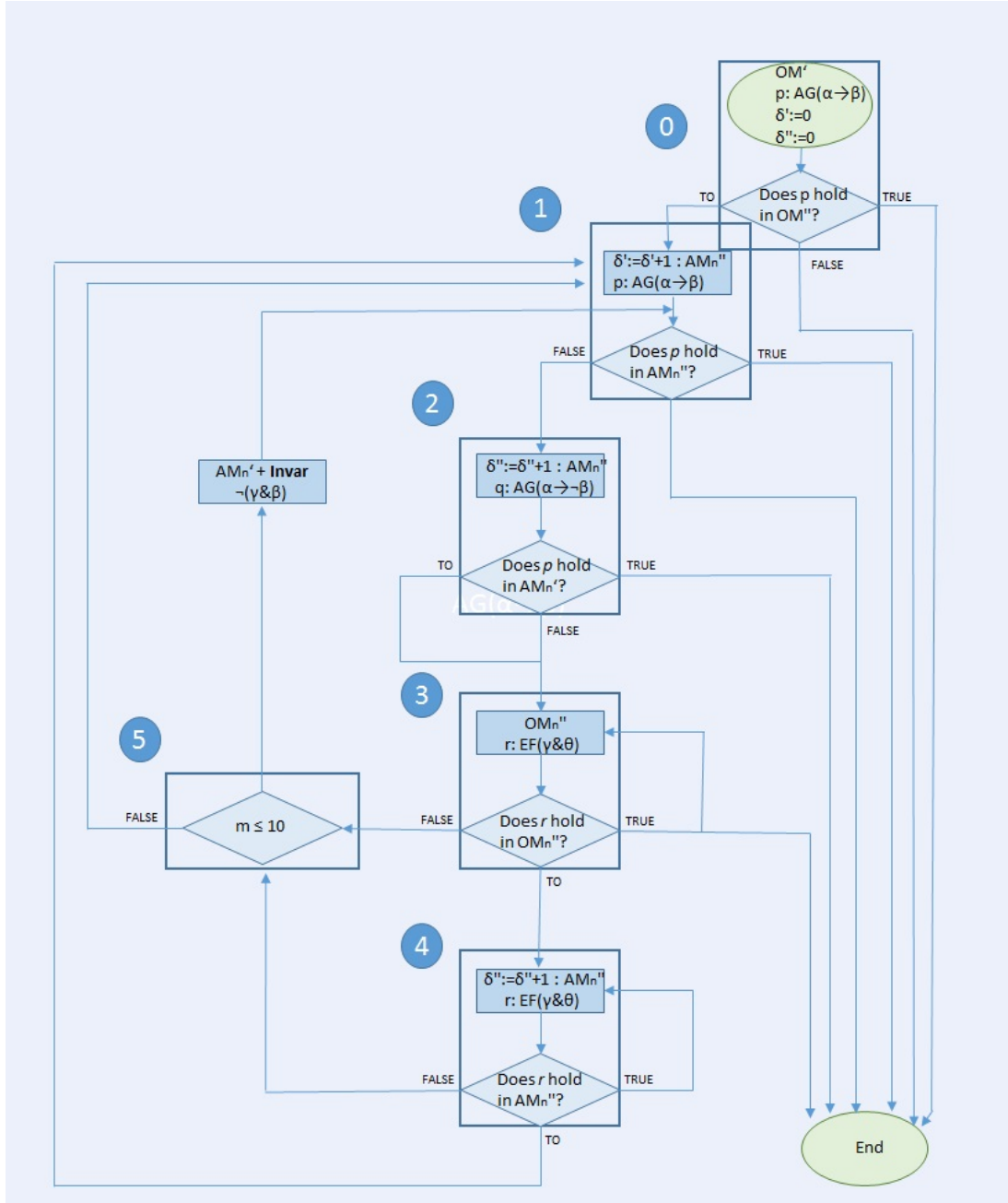


Figure 27: Steps of the variable abstraction algorithm [20]

Step 1: checking the original safety property (p) on an abstract model (AM'). In this step an AM' of the OM' is generated automatically by extracting variables as inputs from $\delta' = 1$. After that, the model checker tries to verify if the property p holds on the AM' . **Implementation:** A verification case file is created from the Python script with the same content as the *original verification case*. Then the input variables are extracted from the δ' that corresponds to the current iteration and they are appended right after the initial inputs variables in the verification case of Step 1.

Step 2: checking the safety property (q) on the same abstract model (AM'). In the second step the algorithm tries to extract more information about the verification of p on the AM'_n . To do so it checks the safety property q in the same abstract model AM'_n . The q property is the same with the p but with a negation on β parameter ($AG(\alpha \rightarrow \neg\beta)$). **Implementation:** A verification case file for step 2 is created with the

same content as the verification of step 1. The safety property is being checked on the same AM'_n but with a negation in the β parameter in the verification case of step 2.

Step 3: checking the reachability property (r) on the original model (OM'') Step 3 of the algorithm aims to analyse the provided counterexample c by extracting the reachability property r from it. By adding the reachability property, a new OM'' is created that can be most of the time different than OM' . This may happens because property r can contain different variables with property p and therefore the reduction techniques will eliminate different variables from the model. **Implementation:** A verification case file for step 3 is created with the same content as the original verification case of step 0. The reachability property can be extracted either from step 1 or step 5. The variables that the property contains will replace the parameters of the verification case. If there are more than one cycle in the counterexample and the result of the model checker is “True”, all of them are examined (unless they have the same content) until the result is different or the cycles are over.

Step 4: checking the reachability property (r) on an abstract model (AM''). In this step an AM''_n is created after a TO is obtained from step 3. This time the inputs for the creation of the abstract model are taken from δ'' of the current iteration by following the same procedure. **Implementation:** A verification case file is created for step 4 with the same content as the verification case of step 3. The algorithm extracts the “input variables” from δ'' from the variable dependency graph generated in step 3.

Step 5: extracting an invariant from the counterexample obtained in step 2 if m is ≤ 10 . This step is responsible for adding an invariant to the AM' or taking the decision of moving to a new abstraction $AM'_n + 1$ by incrementing the δ' if the limit of the maximum invariants number is reached.

For example if the maximum number of possible invariants (m) is ≤ 10 an invariant is added to the current AM' . If m is ≥ 10 the algorithm moves to a new iteration. **Implementation:** A verification case file is created for step 5 with the same content as the verification case of step 1 if $m=1$ or of step 5 if $m \geq 1$ and $m \leq 10$. The algorithm extracts the invariant either from step 1 or step 5 and adds it to the verification case of step 5.

Example of applicability Listing 5, represents an example of an SCL PLC program. The program consists of a simple function with 6 Boolean variables and two input variables. The requirement to be verified is the following:

“If b is true at the end of the PLC cycle, then a should always be true at the end of the same cycle.”

Figure 28 shows the variable dependency graph of the model from the PLC program presented in Listing 5. The red edges are the assignment dependencies, while the gray variables are part of the requirement. Moreover when two different variables are connected by an edge then they are at a distance which equals to 1. The concept of distance between variables is used to build the abstract models as described below.

Listing 5: Example of SCL code

```

1  FUNCTION Test
2  VAR
3
4    a : BOOL;
5    b : BOOL;
6
7    c : BOOL;
8    d : BOOL;
9    e : BOOL;
10   f : BOOL;
11
12  END_VAR
13
14  VAR_INPUT
15
16    g : BOOL;
17    h : BOOL;
18  END_VAR
19
20  BEGIN
21
22    f := g AND h;
23    b := f;
24    c := h AND NOT g;
25    e := f AND NOT c;
26    d := b AND g;
27    a := d AND e;
28
29  END_FUNCTION

```

Step 1: in the first iteration, AM'_1 is generated using $\delta := 1$. Variables e and d are at distance 1 from α according to the variable dependency graph so they are converted to non-deterministic or input variables.

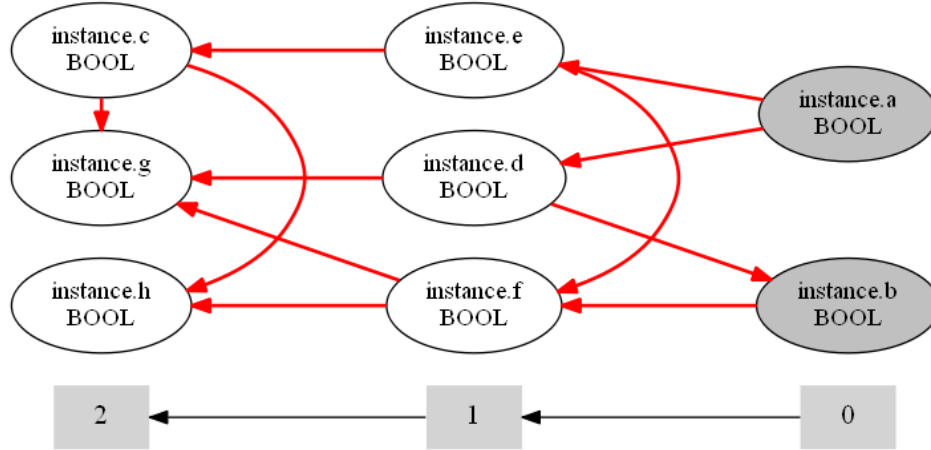


Figure 28: Variable dependency graph example of Listing 5

Due to this abstraction and based on the variable dependency graph, the variables c , d , g and h are eliminated and the variables converted to input variables are d and e and a . The verification result for p is false, and the generated counterexample c is shown in Table 4.

Table 4: Counterexample for p on AM'_1

Variable	End of Cycle1
a	FALSE
b	TRUE
d	FALSE
e	FALSE

Step 2: the safety property q is the following: $(AG(\beta \rightarrow \neg\alpha))$. The verification result is false and we cannot define yet if c is real or spurious. In this case, the algorithm moves to the step 3.

Step 3: The extracted counterexample c , is transformed in a reachability property. When verifying the reachability property r in the new OM'' , the result is false, therefore c is spurious and the algorithm moves to the step 5.

Step 5: an invariant is added to the same abstract model (AM'_1) as the number of potential invariants for AM'_1 is 8. The invariant is represented by the variables and their values which are contained in the counterexample obtained in Step 3.

Step 1: p is verified again on AM'_1 but containing the new invariant. The verification result is false and the new counterexample c is shown in Table 5.

Table 5: Counterexample for p on $AM'_1 + 1$ invariant

Variable	End of Cycle1
a	FALSE
b	TRUE
d	TRUE
e	FALSE

Step 2: the safety property q is verified again on $AM'_1 +$ invariant. The verification result is false again and we cannot define yet if c is real or spurious. The algorithm moves to the step 3.

Step 3: The extracted counterexample c , is transformed in a reachability property. When verifying the reachability r in the new OM'' , the result is false, therefore c is spurious and the algorithm moves to the step 5.

Step 5: as the number of potential invariants for AM'_1 is 8, a second invariant is added to the same abstract model (AM'_1).

Step 1: p is verified again on AM'_1 but also containing the second invariant. The verification result is false and the new counterexample c is shown in Table 6.

Table 6: Counterexample for p on $AM'_1 + 2$ invariants

Variable	End of Cycle1
a	FALSE
b	TRUE
d	FALSE
e	TRUE

Step 2: the safety property q is verified again on $AM'_1 + 2$ invariants. In this case, the verification result is false again and we cannot define yet if c is real or spurious. Therefore the algorithm moves to the step 3.

Step 3: The extracted counterexample c , is transformed in a reachability property. When verifying the reachability r in the new OM'' , the result is false, therefore, c is spurious and the algorithm moves to the step 5.

Step 5: as the number of potential invariants for AM'_1 is 8, a third invariant is added to the same abstract model (AM'_1).

Step 1: p is verified again on AM'_1 but also containing the third invariant. The verification result is true, the algorithm is over and we can conclude that p holds on OM' .

G.3 Experiments

In order to test the efficiency of the algorithm several experiments were conducted to verify the UNICOS objects. The selected objects to present the experimental results in this section are the OnOff, Analog and PCO UNICOS objects. All of the requirements created for each object were verified both with the nuXmv CTL (Computation Tree Logic) algorithm and nuXmv IC3 algorithm. nuXmv [10] is the successor of NuSMV which is a symbolic model checker developed by FBK-IRST, Carnegie Mellon University, the University of Trento and the University of Genova. By using CTL algorithm, the properties(e.g. safety properties,reachability properties) can be expressed as a combination of path quantifiers and linear-time operators. While IC3 is an algorithm using abstraction refinement. In order to verify each requirement and get a faster result, different time out (TO) had to be checked by modifying them manually in the property file which is included in the headless PLCverif version. Some of the created requirements check random behaviours in the PLC code just to test the efficiency of the algorithm and the rest of them represent real behaviours of the objects.

Function blocks description:

1. *OnOff*: This object represents a process equipment driven by digital signals, e.g. an on-off valve, heaters or motors. The object contains 29 input variables (20 BOOL, 3 WORD, 2 ARRAY of 16 BOOL and 4 TIME), 31 output variables (27 BOOL, 2 WORD, and 2 ARRAY of 16 BOOL) and 82 Internal variables (73 BOOL, 1 TIME, 2 TP timer instances, 1 TON timer instance, 1 REAL and 4 INT). More details about the OnOff object can be found in the UNICOS documentation [40].

2. *Analog*: This object represents a process equipment driven by analog signals, control valves or control heaters. The object contains 56 input variables (40 BOOL, 2 WORD and 14 FLOAT), 46 output variables (39 BOOL, 2 WORD and 5 FLOAT). More details about the Analog object can be found in the UNICOS documentation [39].
3. *PCO*: This object represents the Process Control Object and it operates in UNICOS operation modes Auto, Manual and Forced. The object contains 48 input variables (35 BOOL, 2 WORD, 2 FLOAT and 8 STRING), 51 output variables (47 BOOL, 2 WORD and 2 FLOAT). More details about the PCO object can be found in the UNICOS documentation [41].

In order to extract requirements for the experiment, the schemas presented in the UNICOS documentation were analysed. In Figure 29 the schema of a part of the *OnOff* object is presented. The logic gates (AND, OR and NOT) that affect the behaviour of each parameter are presented, as well as the switches that define their priority. Based on this schema, the specification and at the same time based also on the PLC code, developers extract requirements, which represent the behaviour of the variables of the code, and then test them with PLCverif in order to verify the accuracy of the current model. A potential requirement that we could extract for example by looking at Figure 29 is: “ *If (HFO_n and PHFO_n) OR (NOT PHFO_n AND instance.PHFOff AND PAnim AND NOT HFOff) is true at the end of the PLC cycle, then OnSt should always be true at the end of the same cycle* ”.

We tried to verify each requirement in PLCverif first without abstraction and then by applying the abstraction technique on them. The results of the experiments are presented below in tables. For each object and for each technique there is a table that contains the name of the requirement, the TO used for the algorithm, the execution time (in seconds) of the algorithm until a result is provided, the time that PLCVerif needed to provide a result, the Potential State Space and the result of the requirement. When the PLCverif+VA column is marked with “-” then it means that PLCverif was too fast so the algorithm could not be executed. Moreover, for PLCverif tool the TO setted for the verification time was 1h. A detailed description of the requirement for each table is presented in the Appendix II.

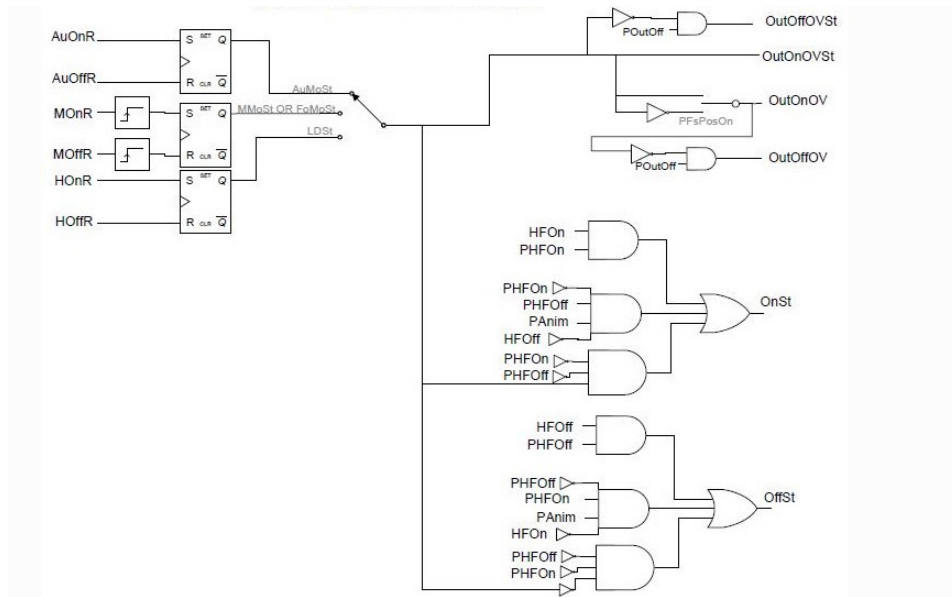


Figure 29: Position calculation schema of OnOff object. [40]

In Table 7, 18 requirements were tested for the PCO object and only 1 of them was satisfied. For the conducted experiments, variable abstraction had the same performance with PLCVerif. Similarly in Table 8, 12 requirements have been tested in total and 2 of them were satisfied. In this case variable abstraction was faster than PLCverif only for 3 of the requirements.

Table 7: PCO UNICOS object tested with nuXmv IC3 algorithm

REQ	TO(s)	PLCverif+VA	PLCVerif	PSS	RESULT
REQ 1	1	00:00:25	00:02:24	2.51×10^{33}	FALSE
REQ 2	20	00:06:26	00:18:36	2.51×10^{33}	FALSE
REQ 3	2	00:00:05	00:00:03	1.00×10^{34}	FALSE
REQ 4	20	00:00:26	00:07:05	2.51×10^{33}	FALSE
REQ 5	8	00:00:21	00:04:10	2.51×10^{33}	FALSE
REQ 6	300	00:48:26	00:04:50	2.51×10^{33}	FALSE
REQ 7	50	00:43:13	00:10:25	2.51×10^{33}	FALSE
REQ 8	20	00:04:09	00:01:21	6.18×10^{32}	FALSE
REQ 9	20	00:03:28	00:01:07	6.18×10^{32}	FALSE
REQ 10	20	00:01:41	00:01:17	2.51×10^{33}	FALSE
REQ 11	50	00:04:45	00:05:40	6.18×10^{32}	FALSE
REQ 12	1	00:00:05	00:00:03	2.51×10^{33}	TRUE
REQ 13	20	00:00:39	00:24:35	6.18×10^{32}	FALSE
REQ 14	300	00:37:12	00:13:00	6.18×10^{32}	FALSE
REQ 15	200	00:18:49	00:00:57	6.18×10^{32}	FALSE
REQ 16	20	00:00:46	00:01:13	5.03×10^{33}	FALSE
REQ 17	20	00:00:21	00:01:21	5.03×10^{33}	FALSE
REQ 18	20	00:01:43	00:04:20	6.18×10^{32}	FALSE

Table 8: PCO UNICOS object tested with nuXmv CTL algorithm

REQ	TO(s)	PLCverif+VA	PLCVerif	PSS	RESULT
REQ 1	20	00:00:35	00:00:08	2.51×10^{33}	FALSE
REQ 2	5	00:01:58	00:00:11	6.18×10^{32}	TRUE
REQ 3	5	00:00:21	00:00:07	2.51×10^{33}	FALSE
REQ 4	5	00:08:11	00:10:47	2.10×10^{34}	FALSE
REQ 5	5	00:00:16	00:00:08	2.51×10^{33}	FALSE
REQ 7	5	00:00:27	00:04:50	2.51×10^{33}	FALSE
REQ 8	3	00:00:22	00:00:03	2.51×10^{33}	FALSE
REQ 9	5	00:02:19	00:01:18	2.51×10^{33}	FALSE
REQ 10	1	00:00:18	00:00:04	4.08×10^{10}	TRUE
REQ 11	5	00:01:16	00:00:01	5.03×10^{33}	FALSE
REQ 12	3	00:09:10	00:02:16	5.03×10^{33}	FALSE

In Table 9, 13 requirements were tested for the ANALOG object and 10 of them were satisfied. The abstraction algorithm was faster for 11 of them. Similarly in Table 10, 12 requirements have been tested in total and 10 of them were also satisfied. The abstraction algorithm was faster for 10 of them.

Table 9: ANALOG UNICOS object tested with nuXmv IC3 algorithm

REQ	TO(s)	PLCverif+VA	PLCVerif	PSS	RESULT
REQ 1	5	00:00:10	00:00:54	1.00×10^{209}	TRUE
REQ 2	0	-	00:00:01	1.44×10^{12}	FALSE
REQ 3	3	00:02:32	00:00:04	7.21×10^{11}	TRUE
REQ 4	10	00:00:13	TO	5.75×10^{198}	TRUE
REQ 5	10	00:00:10	TO	5.75×10^{198}	TRUE
REQ 6	20	00:00:20	TO	5.75×10^{198}	TRUE
REQ 7	200	00:00:07	TO	5.00×10^{208}	TRUE
REQ 8	200	00:00:06	TO	5.00×10^{208}	TRUE
REQ 9	200	00:00:09	TO	2.15×10^{218}	TRUE
REQ 10	200	00:00:06	TO	2.15×10^{218}	TRUE
REQ 11	200	00:00:13	TO	2.15×10^{218}	FALSE
REQ 12	200	00:00:15	TO	2.15×10^{218}	FALSE
REQ 13	200	00:00:15	TO	5.75×10^{198}	TRUE

Table 10: ANALOG UNICOS object tested with nuXmv CTL algorithm

REQ	TO(s)	PLCverif+VA	PLCVerif	PSS	RESULT
REQ 1	10	00:00:19	00:00:54	1.00×10^{209}	TRUE
REQ 2	0	-	00:00:01	1.44×10^{12}	FALSE
REQ 3	0	-	27ms	7.21×10^{11}	TRUE
REQ 4	20	00:00:26	TO	5.75×10^{198}	TRUE
REQ 5	20	00:00:23	TO	5.75×10^{198}	TRUE
REQ 6	20	00:00:26	TO	5.75×10^{198}	TRUE
REQ 7	40	00:00:21	TO	5.00×10^{208}	TRUE
REQ 8	40	00:00:24	TO	5.00×10^{208}	TRUE
REQ 9	40	00:00:23	TO	2.15×10^{218}	TRUE
REQ 10	40	00:00:25	TO	2.15×10^{218}	TRUE
REQ 11	40	00:01:18	TO	2.15×10^{218}	FALSE
REQ 12	40	00:01:10	TO	2.15×10^{218}	FALSE

In Table 11, 16 requirements were tested for the ONOFF object and all of them were satisfied. The abstraction algorithm was faster for 12 of them. Similarly in Table 12, 12 requirements have been tested in total and all of them were also satisfied. The abstraction algorithm was faster for 10 of them.

Table 11: ONOFF UNICOS object tested with nuXmv ic3 algorithm

REQ	TO(s)	PLCverif+VA	PLCVerif	PSS	RESULT
REQ 1	10	00:00:05	00:00:22	4.44×10^{107}	TRUE
REQ 2	10	00:00:05	00:00:25	4.44×10^{107}	TRUE
REQ 3	50	00:01:24	TO	1.11×10^{107}	TRUE
REQ 4	1	00:00:07	TO	5.50×10^{106}	TRUE
REQ 5	1	00:00:06	00:00:21	1.10×10^{107}	TRUE
REQ 6	0	-	28ms	5.50×10^{106}	TRUE
REQ 7	0	-	2ms	5.50×10^{106}	TRUE
REQ 8	10	00:00:56	TO	1.11×10^{107}	TRUE
REQ 9	30	00:01:32	TO	1.11×10^{106}	TRUE
REQ 10	10	00:01:43	TO	5.50×10^{106}	TRUE
REQ 11	10	00:00:10	00:01:00	4.44×10^{107}	TRUE
REQ 12	10	00:00:08	00:01:18	4.44×10^{107}	TRUE
REQ 13	10	00:00:06	00:00:07	4.44×10^{107}	TRUE
REQ 14	10	00:00:05	00:00:52	4.44×10^{107}	TRUE
REQ 15	3	00:00:05	00:00:05	4.44×10^{107}	TRUE
REQ 16	10	00:00:08	00:00:07	3.74×10^{74}	TRUE

Table 12: ONOFF UNICOS object tested with nuXmv CTL algorithm

REQ	TO(s)	PLCverif+VA	PLCVerif	PSS	RESULT
REQ 1	10	00:00:05	00:11:36	4.44×10^{107}	TRUE
REQ 2	10	00:00:05	00:25:21	4.44×10^{107}	TRUE
REQ 3	50	00:01:17	TO	1.11×10^{107}	TRUE
REQ 4	1	00:00:08	00:00:03	5.50×10^{106}	TRUE
REQ 5	1	00:00:05	00:00:08	1.10×10^{107}	TRUE
REQ 6	0	-	916ms	5.50×10^{106}	TRUE
REQ 7	10	00:00:53	TO	5.50×10^{106}	TRUE
REQ 8	10	00:00:05	00:13:20	4.44×10^{107}	TRUE
REQ 9	10	00:00:06	00:12:33	4.44×10^{107}	TRUE
REQ 10	10	00:00:05	00:00:10	3.74×10^{74}	TRUE
REQ 11	10	00:00:06	00:21:55	3.74×10^{74}	TRUE
REQ 12	3	00:01:10	00:16:03	3.74×10^{74}	TRUE
REQ 13	10	-	00:17:13	3.74×10^{74}	TRUE

G.4 Analysis and Conclusions

In this chapter, we described the implementation of variable abstraction and introduced the conducted experiments of some PLC programs. By using variable abstraction we were able to verify various requirements for real-life PLC programs used at CERN.

A limitation of variable abstraction is that currently we can verify only safety properties. The reason for this is that the verification requirements for complete UNICOS PLC programs are presently safety properties. Safety properties are considered the properties with the following pattern: if α is true at the end of the PLC cycle, then β should always be true at the end of the same cycle which is translated to the following pattern: $\text{AG}(\alpha \rightarrow \beta)$. The Greek letters represent Boolean logical expressions which contain multiple variables.

After analysing the conducted experiments some interesting results found with this abstraction algorithm:

1. When there are a lot of “input variables” of integer type in the verification case, then the algorithm is not always efficient due to the exponential growth in the number of input bits. In this case the state space remains huge even after the applied abstractions and model checker still can not deal with it.
2. When it comes to *FALSE* properties, the algorithm was able to extract a result almost always from *step 3*. Out of the 33 *FALSE* requirements, only 4 of them obtained their result from step 2.
3. When it comes to *FALSE* properties where the model is not compliant with the specification the algorithm does not always provide a result. Moreover PLCverif is usually faster than variable abstraction in obtaining a result .
4. On the contrary, when it comes to *TRUE* properties the algorithm in the majority of the experiments is way faster than PLCVerif. Therefore the performance of the verification of true properties is improved.
5. Any complexity linked with formal methos and formal verification is hidden from the developer.

H Static Code Analysis

H.1 Introduction

In the previous chapters, the thesis focused in the applicability of model checking to PLC programs. In this chapter, the PLC code will be examined in a different way. Apart from formal methods and testing, static analysis is another way to check PLC programs without the need for specifications supplied from the user. By using static code analysis, a potentially spurious PLC code or an undefined behaviour can be found. As mentioned in Chapter 2, there are already a few static analysis tools for PLC programs. In this chapter, several of these tools are analysed over their applicability to our PLC programs. In addition, 3 prototypes based in some technologies already made at CERN have been developed. These prototypes but also the already existing tools were applied to some of our CERN PLC programs, allowing to our PLC program developers to find errors and improve their programs.

The structure of this chapter is the following:

1. In Section 5.2 the existing static analysis tools are briefly evaluated.
2. In Section 5.3 the implementation and design of 2 static analysis tool prototype is described.
3. In Section 5.4 the results of the conducted experiments are presented.
4. In Section 5.5 the analysis of the tool and the extracted conclusions are presented.

H.2 Static analysis tools evaluation

In this section the already existed static analysis tools will be analysed. The tools were compared and evaluated in order to extract information for the prototypes that were developed.

PLC Checker : The tool PLC Checker which is launched by Itris Automation Company analyse PLC Programs written in Schneider (Unity Pro), SIEMENS (Simatic Step7), Rockwell Automation (RSLogix5000) and OMRON (Sysmac Studio) platforms and it contains a predefined set of rules aiming to detect naming conventions, structure errors and bad programming techniques such as the lack of comments in the code. PLC Checker was designed to analyse complete PLC programs written in languages provided by SIEMENS. However the PLC programs written in the SIEMENS platform that are being used at CERN are not “complete”. A complete PLC program for the tool means to provide for analysis apart from the .SCL file also some other type of files that so far are not developed because there are unnecessary for the control systems which are used at CERN.

Arcade.PLC: Arcade.PLC is an academic tool which was not developed for commercial purposes. Arcade.PLC, developed in RWTH Aachen University, supports PLC programs written in ST and IL languages from the IEC 61131 standard and also in STL from Siemens S7. Moreover, so far, it does not support the analysis of PLC programs written in SCL. Also, Arcade.PLC is a non-rule based analysis tool but instead it does a semantic analysis of the code rather than matching different rules or pattern. It uses abstract interpretation to compute sound value ranges for variables and the warnings that are generated in the code analysis are derived from these value-range results which is out of our scope.

CODESYS: CODESYS Static Analysis tool is a software tool developed by 3S-Smart Software in Kempten of Germany that contains also a predefined set of rules and it supports PLC programs written in IL, ST, LD, FBD and SFC languages from the IEC 61131 standard. CODESYS static analysis tool is so far limited to the CODESYS platform. Despite the fact that it contains a variety of static analysis rules and that by using the Automation Platform (a platform for the extension of the CODESYS development system) rules can be extended, currently it analyses PLC programs written only on the standard IEC 61131-3 languages.

Table 13: Comparison table I of existed static analysis tools for PLC programs.

Tool	License	Report	Extensible rules	Language	Restrictions
CoDeSyS	Yes	Console	Yes	IEC 61131-3	Limited to CoDeSys platform
PLC Checker	Yes	PDF/e-mail	Yes	Siemens, OMRON, CoDeSys, Scheinder, Rockwell - Automation	So far .AWL and .ASC files are required to analyze an SCL file.
Arcade.PLC	No	Console	No	IEC 61131 ST, IEC 61131 IL, Siemens STL	-

Table 14: Comparison table II of existed static analysis tools for PLC programs.

Tool	Rule categories	Exclude errors	Error Message
CoDeSyS	coding rules, naming conventions, metrics	No info	Error, warning
PLC Checker	naming rules, comments, writing rules, structure rules	Yes	Info, warning, error, fatal
Arcade.PLC	Non-rule based tool	No	Fatal errors, errors, warnings

H.3 Contribution

H.3.1 Introduction

At CERN two technologies are used for two different projects, that could help us to build a static analysis tool and analyse the benefits of static analysis in PLC programs. Therefore, two prototypes of a static analysis tool for PLC programs written in SCL were developed. The goal of the tool in its current status, is to detect simple violations in the structure of the code as well as naming convention errors. To do so a set of rules had to be developed. Since the project is still in a research phase, two different approaches were followed to develop the tool and test its effectiveness.

The first approach is based on an implemented project where a DSL (Domain Specific Language) was developed to transform the code of the PLC programs into an AST (Abstract Syntax Tree) representation. Abstract syntax trees are data structures widely used in compilers. Due to their property of representing the structure of program code, the code can be parsed easily[57]. This Siemens SCL code DSL is developed using the Spoofox platform. Thanks to the AST provided by the parser, static analysis rules were developed navigating on the AST. For the development of

the tool both of the approaches are based on an Eclipse-based user interface built on the Spoofax platform. The third approach, is based on the PLCverif tool. PLCverif which is implemented as an Eclipse RCP application, is using Xtext technology to parse the PLC code. After the code is parsed, PLCverif translates it from concrete syntax to an AST representation that can be used as a base to develop the static analysis rules. In figure 30 we can see the process followed for the development of the rules, which is the same for the two approaches

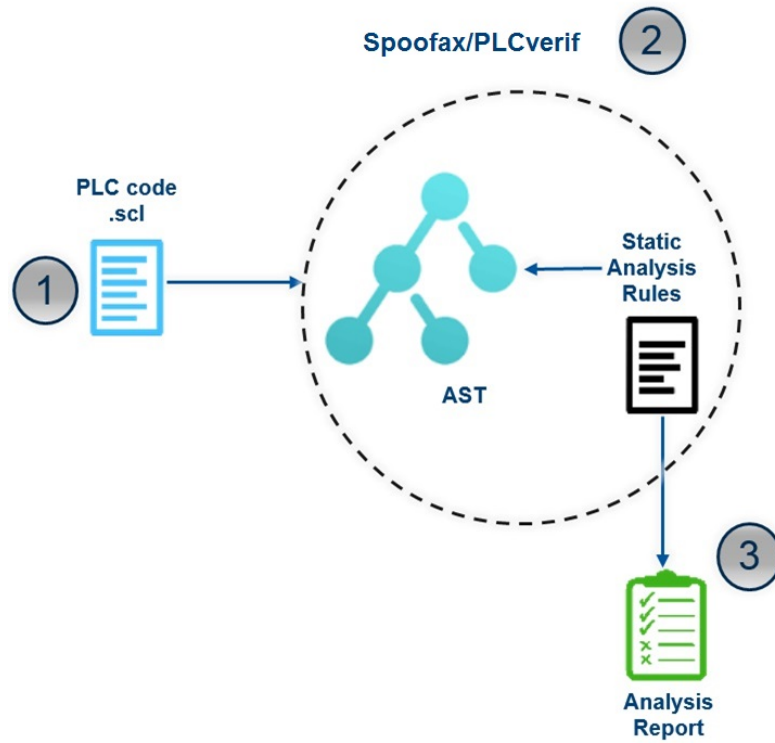


Figure 30: Static analysis approach workflow.

In Listing 16, the AST representation of the PLC code presented in Listing 9, is shown as is produced by Spoofax. Different program units, are represented by different blocks of terms in the AST. For example at lines 2-28 is the representation of the function block (FBBegin) which contains several terms that include:

- The name of the function block (line 3).
- The title, the version, the author and other information that concerns the function block of the code and are shown inside a list in the Block-Attributes term (lines 4-9).
- The declaration part (lines 10-18) where the declared variables and their types are shown.
- Finally the statement declaration part (lines 19-28) which contains all the statements and their assigned values.

Listing 6: Generated AST for the PLC code presented in Listing 9.

```

1 SCLProgram(
2   [ FBBegin(
3     Some(RegularIdentifier("test"))
4     , BlockAttributes(
5       [ NoSemiColon(Title(Quoted("'DEMO'")))
6         , NoSemiColon(Version(Quoted("6", "6")))
7       ]
8     )
9   )

```

```

7      , NoSemiColon(Name(" 'OBJECT '"))
8    ]
9  )
10  , Declarations(
11    [ StaticVarSubsection(
12      [ VarDeclNoInit([RegularIdentifier(
13        "a"]), BoolDT())
14      , VarDeclNoInit([RegularIdentifier(
15        "b"]), BoolDT())
16    ]
17  )
18  , InputVarSubsection([VarDeclNoInit([
19    RegularIdentifier("c")], BoolDT())])
20 ]
21 )
22 , [ StatementDecl(
23   []
24   , ValueAssign(VariableValue(
25     RegularIdentifier("b")), Not(
26     VariableValue(RegularIdentifier("a"))
27   )))
28   , StatementDecl(
29     []
30     , ValueAssign(VariableValue(
31       RegularIdentifier("c")),
32       ConstantValue(ImplicitFalse("FALSE"))
33     ))
34   )
35 ]
36 )
37 , FunctionBegin(
38   Some(RegularIdentifier("test"))
39   , Some(VoidDT())
40   , BlockAttributes([])
41   , Declarations(
42     [ StaticVarSubsection(
43       [ VarDeclNoInit([RegularIdentifier(
44         "c")], BoolDT())
45       , VarDeclNoInit([RegularIdentifier(
46         "d")], BoolDT())
47     ]
48   )
49   , InputVarSubsection([VarDeclNoInit([
50     RegularIdentifier("e")], BoolDT())])
51 ]
52 )
53 , [ StatementDecl(
54   []
55   , ValueAssign(VariableValue(
56     RegularIdentifier("d")),
57     VariableValue(RegularIdentifier("c"))
58   ))
59   )
60   , StatementDecl(
61     []
62     , ValueAssign(VariableValue(
63       RegularIdentifier("e")),
64       ConstantValue(ImplicitTrue("TRUE")))
65   )
66 ]
67 )
68 ]

```

H.3.2 Spoofax approach

The Spoofax approach allows us to write static analysis rules based on Stratego/XT and Java.

Stratego/XT-based rules The first approach was developed by using the Stratego/XT language, which is a functional programming language provided by the Spoofax platform. The Stratego/XT language provides rewrite rules and strategies for expressing basic transformations and controlling the application of rules. Moreover, it supports domain-specific languages, compilers, program generators, and a wide range of meta-programming tasks. This approach operates on the AST which is produced by the Spoofax parser. The grammar is complete which makes the produced AST complete. As a result, all of the elements of the PLC program can be represented by nodes. By using Stratego/XT, several simple rules were implemented to detect bugs in the source code of the PLC program. The currently implemented rules in the Stratego/XT approach are divided in two groups: naming rules and structure rules; some of them are described below.

- Detect nested if statements in the code: in Listing 7 the example code contains four nested ifs. When there are more than three nested ifs in the code we consider it not to be a good programming technique as this can lead to confusions and decrease code readability.
- Define type prefixes for variables: using a variable's name to also communicate other attributes can improve readability and help reducing programming mistakes. For example boolean variables should start with the letter *b*.
- Detect variables that do not have the recommended length: all variables should have a specified acceptable length. Otherwise, the maintenance of the code is more difficult and is not understandable.
- Detect if different variables have the same name in the code: in Listing 9 an example where different variables have the same name is shown. Re-using the same global name for any variable type would make the code difficult to read.
- Detect if a function was called inside itself: functions should not call themselves directly or indirectly. This kind of violation should be detected in the code, otherwise it can affect the behaviour of the model and create severe errors.
- Detect if a function was not called in the whole program: a function which is not called in the program can be considered as dead code.
- Detect if physical addresses are used: system depended physical addresses shall be avoided as this can lead to less portable code (see Listing 8).

Listing 7: Example of nested if statement in an SCL code

```

1 FUNCTION_BLOCK A
2
3   VAR
4       a : BOOL;
5       b : BOOL;
6
7   END_VAR
8
9   VAR_INPUT
```

```

10      c : BOOL;
11      d : INT;
12      e : INT;
13      f : BOOL;
14  END_VAR
15
16  BEGIN
17      b := NOT a;
18      c := FALSE;
19      IF c THEN
20          d := 1;
21          IF b THEN
22              IF a THEN
23                  b := FALSE;
24                  IF f THEN
25                      e := 5;
26                  END_IF;
27              END_IF;
28          END_IF;
29      END_IF;
30
31  END_FUNCTION_BLOCK

```

Listing 8: Example of an SCL code where physical addresses are displayed as variables.

```

1  FUNCTION_BLOCK Test
2
3  VAR
4      IW12: REAL;
5      A:INT;
6  END_VAR
7
8  BEGIN
9      IW12:= 1.1;
10     A:= PQB9;
11
12 END_FUNCTION_BLOCK

```

Listing 9: Example where different element types have the same name

```

1  FUNCTION_BLOCK TEST_A
2  VAR
3      a : BOOL;
4      b : BOOL;
5
6  END_VAR
7
8  VAR_INPUT
9      c : BOOL;
10 END_VAR
11
12 BEGIN
13     b := NOT a;
14     c := FALSE;
15
16 END_FUNCTION_BLOCK
17
18 FUNCTION TEST_B: VOID
19 VAR
20     c : BOOL;

```

```

21      d : BOOL;
22
23      END_VAR
24
25      VAR_INPUT
26          e : BOOL;
27      END_VAR
28
29      BEGIN
30          d := c;
31          e := TRUE;
32
33      END_FUNCTION

```

As described in Figure 30, our analysis process comprises three steps:

1. The PLC program is parsed as an input to Spoofax and then is translated to an AST representation.
2. After, the AST is analysed and the static analysis rules are written on top of it in Spoofax.
3. Finally, after the execution of the tool, all the detected violations appear in the console.

Implement a rule using Stratego/XT: Each rule is represented by a different .str file (Stratego file). To create a rule, a top level rule has to be created, which consists of simpler rules. A rule for Stratego is like a function: it implements a task. Moreover, there are predefined rules but the developer can implement more of them by defining their logic. The first step to create a rule is to collect all the target variables. Then, additional rules need to be created for all the tasks that have to be defined for the collected variables.

In Listing 10 an example of a Stratego rule to detect whether an input variable is assigned inside the function, is presented. In line 1, the top rule is introduced. Then, in lines 2-13, the strategy of the top rule which is built from more simple ones, is defined. In lines 16-19, the first rule, which collects all the input variables, is implemented. Finally, at lines 22-27, the second rule which maps the statements over the inputs and prints the error messages in the console, is defined.

Listing 10: Example of a stratego rule

```

1 variable-name-analysis = validate-input
2 validate-input:
3
4 input -> <fail>
5 where
6     <
7         ?FunctionBegin(_,_,_,Declarations(decls),
8             stmts) +
9         ?FunctionNoBegin(_,_,_,Declarations(decls),
10             stmts) + ?ProgramBegin(_,_,
11             Declarations(decls),stmts) +
12         ?ProgramNoBegin(_,_,Declarations(decls),
13             stmts)
14     > input
15
16 ; inputs := <collect-all-inputs> decls
17 ; <map(try(validate-input-usage(|stmts)))>
18     inputs
19
20
21
22
23
24
25
26
27

```



```

16 collect-all-inputs:
17 list -> inputvars
18 where
19     inputvars := <collect-all-unfiltered(?
                    InputVarSubsection(_));collect-all-
                    unfiltered(?RegularIdentifier(_))> list
20
21
22     validate-input-usage(|stmts):
23     pattern@RegularIdentifier(name) -> name
24
25     where
26         <collect-all-unfiltered(?ValueAssign(
                    VariableValue(RegularIdentifier(name))
                    , _));not(?[])> stmts
27 ; <log-error(|pattern)> $[Input variable:[name
    ] should not been assigned inside the
    function]

```

Java-based rules The main difference between the two approaches is the way we access the AST and the language which is used when implementing the rules. On the one hand in Stratego/XT approach, the AST is a low-level one. For example different types of variables will be represented in different AST patterns. As a result, in order to collect all the names of variables, all possible patterns must be checked. However, on the other hand, the AST in this approach covers the complete grammar of the PLC code, which means that all the units and terms of the program can be represented.

In order to have more flexibility in writing rules, a Java interface was developed using the Spoofox platform. Therefore, for the second approach, still by using Spoofox, rules can be developed now in Java. For the Java approach, the same AST that was used for the first approach can be used. Nonetheless, because its a low level one, the creation of a new AST representation was attempted. The reason for this, is that when it comes to complex rules, access a low level AST can be complicated and as much more patterns of it need to be checked, the performance of the tool can be affected in a negative way. An SCL program consists by a list of units and a unit can be a Function, a Function Block, a Data Block etc. Moreover, each of them has a name, a return type, variables and statements and to create the AST, each unit should be represented by a Java class. So far only rules that concern the declaration part can be developed because the part that matches the statements is still not implemented. On the contrary, the tool now is more flexible as the rules can be activated or dis-activated by the user according to the needs of the analysis. The rules implemented in Java, concern so far only naming rules and are almost the same with the ones implemented in Stratego/XT.

The procedure for the analysis is the same as in the Stratego/XT approach. The PLC code, is given to Spoofox platform as an input, then it is translated into an AST representation and on top of this the rules are developed. The difference between the two approaches is in the analysis results where now, are also given as an analysis report to the user.

Implement a rule using Java: To create a new rule, a new Java class has to be created as each rule is represented by a different class. In this class, a method that is used to *exclude* or *include* rules from the analysis procedure of the code according to the current needs, has to be created.

Then the rule is divided into two parts : the *traversal* of the AST and the *logic* that is followed to create the rule. In the *traversal* part, all the needed elements have to be collected from the AST. In the *logic* part, these elements are used in order to implement the behaviour of the rule.

Listing 11: Example of a Java rule

```

1 package scl_static_analysis.strategies;
2
3 public class VariableAnalyses {
4
5     public static void isUpperCase(SCLProgram program)
6     {
7         if(Exclusions.isExcluded(VariableAnalyses.class))
8         {
9             return;
10        }
11
12        List<AbstractVariable> variables = program.
13            getSclUnits().stream()
14                .filter(unit -> unit instanceof
15                    Function)
16                .map(function -> function.
17                    getVariables())
18                .flatMap(list -> list.stream())
19                .collect(Collectors.toList());
20
21        for(AbstractVariable variable : variables){
22            if(!Character.isUpperCase(variable.getName().
23                .charAt(0))){
24                run_static_analysis_0_0.print("Warning @
25                    Line" + (Integer.parseInt(variable.
26                        getOriginLine().toString())+1)
27                    + " on " + variable.getOriginText().
28                        toString() + ":
29                        Variable should start with a
30                        capital letter.");
31            }
32        }
33    }
34 }

```

In Listing 12, an example of a rule class is presented. Between lines 5-8 the method which is responsible for the execution of the rule is defined. Between lines 10-14 the elements of the program will be collected from the AST and placed in a list. Finally, in lines 16-21 is the logic of the rule where every variable where its first letter is not upper-case will appear in the static analysis report.

Listing 12: Example of a Java rule

```

1 package scl_static_analysis.strategies;
2
3 public class VariableAnalyses {
4
5     public static void isUpperCase(SCLProgram program)
6     {
7         if(Exclusions.isExcluded(VariableAnalyses.class))
8         {
9             return;
10        }
11
12        List<AbstractVariable> variables = program.
13            getSclUnits().stream()
14                .filter(unit -> unit instanceof
15                    Function)
16                .map(function -> function.
17                    getVariables())
18                .flatMap(list -> list.stream())
19                .collect(Collectors.toList());
20
21        for(AbstractVariable variable : variables){
22            if(!Character.isUpperCase(variable.getName().
23                .charAt(0))){
24                run_static_analysis_0_0.print("Warning @
25                    Line" + (Integer.parseInt(variable.
26                        getOriginLine().toString())+1)
27                    + " on " + variable.getOriginText().
28                        toString() + ":
29                        Variable should start with a
30                        capital letter.");
31            }
32        }
33    }
34 }

```

```

10     List<AbstractVariable> variables = program.
        getSclUnits().stream()
11         .filter(unit -> unit instanceof
            Function)
12         .map(function -> function.
            getVariables())
13         .flatMap(list -> list.stream())
14         .collect(Collectors.toList());
15
16 for(AbstractVariable variable : variables){
17     if(!Character.isUpperCase(variable.getName()
        .charAt(0))){
18         run_static_analysis_0_0.print("Warning @
            Line" + (Integer.parseInt(variable.
            getOriginLine().toString())+1)
19         + " on " + variable.getOriginText()
            .toString() +
20         ":Variable should start with a
            capital letter.");
21     }
22 }
23 }
24 }

```

When programming PLC codes in UNICOS, there are some general abbreviations concern the naming conventions of the variables. To have more maintainable and readable code, the names of the variables should be constructed only from the abbreviations (e.g. a variable with the name AuMoSt corresponds to AutoModeStatus). In order to detect the variable names that do not respect this restriction, a rule was developed. As it was mentioned before, the current AST does not fully support all PLC programs yet. For this reason, in this approach, except of the rule for the abbreviations, the rules about naming conventions that were implemented in Stratego/XT approach were implemented also in Java.

To produce the analysis report which contains the overview of the analysis of the PLC code, a script had to be developed. More specifically, the static analysis tool, produces a .CSV file that contains analysis information. The script extracts these information from the generated .CSV and generates an .html file where each rule is linked to a PDF file that contains its documentation (see figure 31). In the current version of the tool the user workflow is not yet fully automated. The developer, for the purpose of checking the code, he has to provide as an input to the tool the PLC code and execute the tool. In addition he has to also execute the script in order to have the HTML analysis report (see Figure 32).

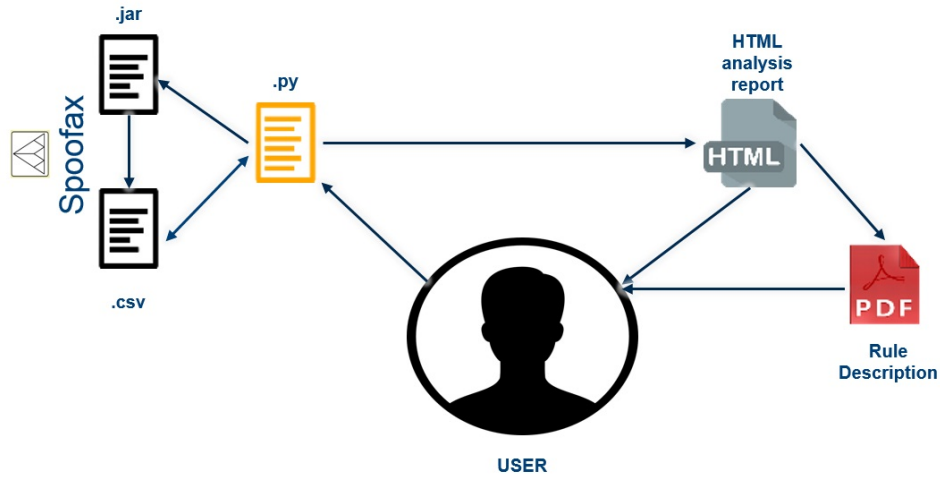


Figure 31: User workflow for the Java approach.

Listing 13: Example of an SCL code

```

1 FUNCTION Test:void
2 VAR
3 a : BOOL;
4 b : BOOL;
5 c : BOOL;
6 END_VAR
7 END_FUNCTION
8
9 FUNCTION_BLOCK Test
10 VAR_INPUT
11 a : BOOL;
12 d : BOOL;
13 END_VAR
14 END_FUNCTION_BLOCK

```

In figure 32 the generated analysis report is which corresponds to the analysis of the code presented in Listing 13. Each rule corresponds to a unique id and N stands for the Naming convention rules category. Rules according to their severity level are distinguished in three categories: info (when the detected code does not affect the of the program), severe (when the error might decrease the readability and the effectiveness of the code) and fatal (when the detected error affects the logic of the program).

Static Analysis Report

Filter by RuleID:

Rule ID	Rule Severity	Rule Description	Error Line	Error Message
N6	Info	Variables should have a defined acceptable name length	Line: 4	Variable d: the name of the variable length shouldn't be less than 3.
N6	Info	Variables should have a defined acceptable name length	Line: 9	Variable a: the name of the variable length shouldn't be less than 3.
N6	Info	Variables should have a defined acceptable name length	Line: 10	Variable b: the name of the variable length shouldn't be less than 3.
N6	Info	Variables should have a defined acceptable name length	Line: 11	Variable c: the name of the variable length shouldn't be less than 3.
N9	Fatal	Different element types should not bear the same name.	Line: 1	Function Test: the same name is used by FUNCTION_BLOCK at Line 7
N4	Info	UpperCamelCase should be used	Line: 3	Variable a: should be written in UpperCamelCase
N4	Info	UpperCamelCase should be used	Line: 4	Variable d: should be written in UpperCamelCase
N4	Info	UpperCamelCase should be used	Line: 9	Variable a: should be written in UpperCamelCase
N4	Info	UpperCamelCase should be used	Line: 10	Variable b: should be written in UpperCamelCase
N4	Info	UpperCamelCase should be used	Line: 11	Variable c: should be written in UpperCamelCase

Generated at 28 Nov 2016 10:50:55 | (C) CERN EN-ICS-PCS

Figure 32: Static analysis report.

H.3.3 PLCverif approach

In this last approach, the static analysis tool, is integrated with the verification tool of PLCverif. The AST, which is produced by Xtext can support all of our PLC programs which are written in SCL. However, the grammar that the tool provides currently is not complete as some features of SCL are not used in the PLC codes of CERN.

The logic to write rules is the same with the two previous approaches. The PLC code is given as an input to PLCverif, it is transformed to an AST representation and based on the AST the rules are written in Java. After the static code analysis an HTML report with all the detected violations, is provided to the user. In the HTML the user can see the error type, the ruleID, the error message and the line number. In addition, by clicking in the line number he has access to the code overview where the line with the error is underlined.

In this approach, there was more flexibility in writing rules as there are no restrictions. Three error categories were created: information, warning and error (when the violation can actually twist the behaviour of the program) and each rule can correspond to one of them. In order to develop a rule, a new class has to be created; by the usage of several predefined methods, the AST can be accessed and parsed and the logic of the rules can be defined. Moreover, the rules created for the needs of this approach, detect violations relevant to naming conventions, to structure issues and to unreachable code. All the rules implemented in the Stratego/XT and Java approach were also implemented in PLCverif with some additional rules described below:

- Prefixes for FB and OB: Function blocks and Organization blocks should

have a name prefix "FB" and "OB" respectively.

- Detect dead code: Rule to check unreachable code within POU's. Unreachable code may occur due to conditions that are always set to TRUE or FALSE, due to never-called part of the code, due to the use of jumps (statements like CONTINUE, RETURN, GOTO).
- Detect if an IF statement has an ELSE clause.
- Detect IF conditions where the IF body cannot be reached (See Listing 14) .
- Data types conversion should be explicit: according to the compiler the result may differ while multiply or divide different data types.

Listing 14: Example of an SCL code where IF conditions can not be satisfied.

```
1 FUNCTION Test:void
2   VAR
3
4       a : BOOL;
5       b : INT;
6       c : INT;
7
8   END_VAR
9
10 BEGIN
11   IF b<1 AND b>20
12
13       a:=TRUE;
14
15   END_IF;
16
17   IF c<1 AND c=20
18
19       a:=FALSE;
20
21   END_IF;
22
23 END_FUNCTION
```

In Listing 15, a Java rule example implemented in PLCverif is presented. In the method ruleId() the ID of the rule is returned. Then in the caseAssignmentStatement method, the types and the names of the variables are collected and finally if the types of the multiplied variables are different an error returns to the user in an HTML report format.

Listing 15: Example of a rule implemented in PLCverif that detects if data types conversion are explicit.

```
1
2 package ch.cern.en.ice.plcverif.staticanalysis.visitorrules
3 ;
4
5 import java.util.ArrayList;
6 import java.util.List;
7
8
9 public class MultiplyExpression extends
10     AbstractVisitorAnalysisRule {
11
12 @Override
13 public String ruleId() {
```

```

13
14     return "R18";
15 }
16
17 @Override
18 public AnalysisResultItem caseAssignmentStatement(
19     AssignmentStatement assignment) {
20     NamedElement rightSideVar;
21     NamedElement leftSideVar;
22     Expression constantVarLeft;
23     String leftSideVarName = "";
24     String leftVarType = "";
25     List<String> types = new ArrayList<>();
26
27     Expression constantVarRight;
28     String rightVarType = "";
29     String rightSideVarName = "";
30     for (EObject e : Helper.iterable(assignment.eAllContents())
31         ) {
32         if (e instanceof MultExpression) {
33             Expression leftSide = (((MultExpression) e).
34                 getMultLeft());
35             if (leftSide instanceof PrimaryExpression) {
36                 if (((PrimaryExpression) leftSide).getValue() !=
37                     null) {
38                     leftSideVar = ((PrimaryExpression) leftSide).
39                         getValue().getPostfix().getNamedElement();
40                 if (leftSideVar instanceof SingleVariable) {
41                     leftSideVarName = leftSideVar.getName();
42
43                     leftVarType = ((SingleVariable) leftSideVar).getType
44                         ();
45                 }
46             }
47         }
48         } else if (((PrimaryExpression) leftSide).
49             getUnnamedConstant() != null) {
50             constantVarLeft = ((PrimaryExpression) leftSide);
51             if (constantVarLeft instanceof SingleVariable) {
52                 leftSideVarName = ((SingleVariable)
53                     constantVarLeft).getName();
54                 leftVarType = ((SingleVariable) constantVarLeft).
55                     getType();
56             }
57         }
58     }
59     for (Expression rightSide : (((MultExpression) e).
60         getMultRight()) {
61         if (rightSide instanceof PrimaryExpression) {
62             if (((PrimaryExpression) rightSide).getValue() !=
63                 null) {
64                 rightSideVar = ((PrimaryExpression) rightSide).
65                     getValue().getPostfix().getNamedElement();
66                 if (rightSideVar instanceof SingleVariable) {
67                     rightVarType = ((SingleVariable) rightSideVar).
68                         getType();
69                     rightSideVarName = rightSideVar.getName();
70                     types.add(rightVarType);
71                 }
72             }
73         } else if (((PrimaryExpression) rightSide).

```

```

        getUnnamedConstant() != null)
63     {
64         constantVarRight = ((PrimaryExpression) rightSide)
            ;
65         if (constantVarRight instanceof SingleVariable) {
66             rightVarType = ((SingleVariable) constantVarRight
                ).getType();
67             rightSideVarName = ((SingleVariable)
                constantVarRight).getName();
68             types.add(rightVarType);
69         }
70     }
71 }
72 }
73 }
74 }
75 }
76 }
77 }}
78 for (String rightType : types) {
79     if (!(leftVarType.equals(rightType))) {
80         reportParameterizedAnalysisResultItem(assignment,
81             PlcverifSeverity.Info,
82             "Data types conversion should be explicit: '%s,%s'
                ",
83             rightType, leftVarType);
84     }
85 }
86
87     return super.caseAssignmentStatement(assignment);
88 }
89
90 }

```

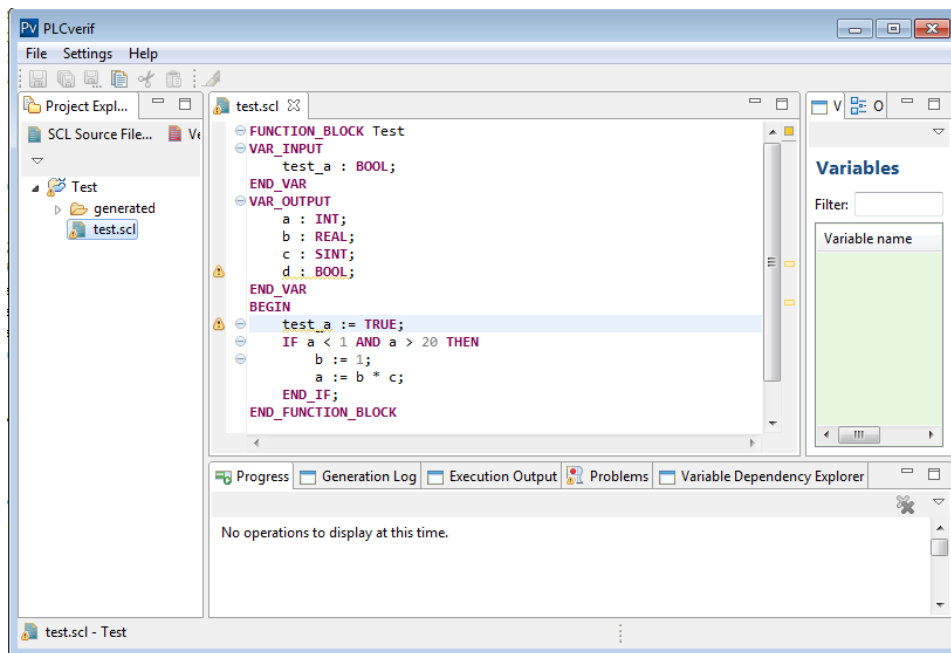
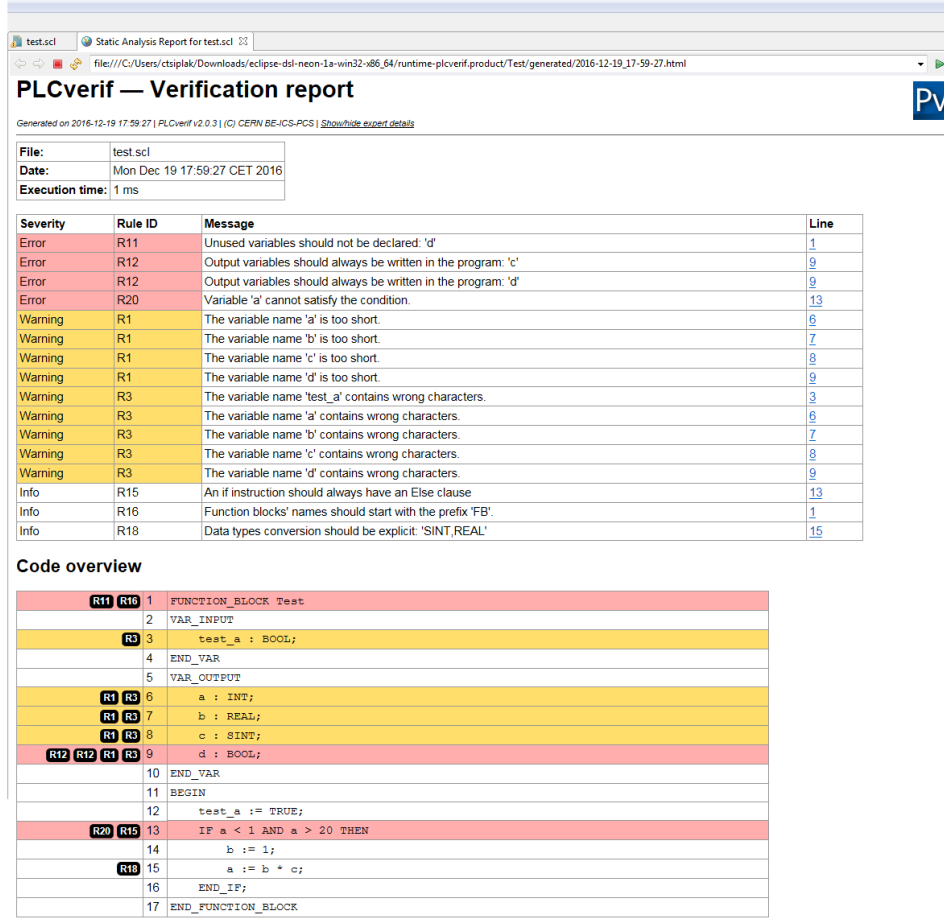


Figure 33: SLC editor in PLCverif approach of static analysis tool.

In figure 33 the editor of PLCverif is presented. The user can still use the features of PLCverif and import a PLC code written in SCL, even modify it or write a new code. Therefore, it can execute the tool to perform

static code analysis to the provided code. As a result, PLCverif is going to generate an HTML report as it can be seen in Figure 34. All the rules which corresponds to the "Error" category are represented in the report with red color, while "Warnings" and "Information" notifications are represented by yellow and white respectively. In the code overview, the actual code can be seen. Next to each line where a violation was detected, there is the rule id of the rules that were not respected. Moreover, some general information like the name of the analysed SCL file, the date and time of the analysis as long as the execution time of the tool are presented in the report.



PLCverif — Verification report

Generated on 2016-12-19 17:59:27 | PLCverif v2.0.31 | (C) CERN BE-ICS-PCS | [Show/hide expert details](#)

File:	test.scl
Date:	Mon Dec 19 17:59:27 CET 2016
Execution time:	1 ms

Severity	Rule ID	Message	Line
Error	R11	Unused variables should not be declared: 'd'	1
Error	R12	Output variables should always be written in the program: 'c'	9
Error	R12	Output variables should always be written in the program: 'd'	9
Error	R20	Variable 'a' cannot satisfy the condition.	13
Warning	R1	The variable name 'a' is too short.	6
Warning	R1	The variable name 'b' is too short.	7
Warning	R1	The variable name 'c' is too short.	8
Warning	R1	The variable name 'd' is too short.	9
Warning	R3	The variable name 'test_a' contains wrong characters.	3
Warning	R3	The variable name 'a' contains wrong characters.	6
Warning	R3	The variable name 'b' contains wrong characters.	7
Warning	R3	The variable name 'c' contains wrong characters.	8
Warning	R3	The variable name 'd' contains wrong characters.	9
Info	R15	An if instruction should always have an Else clause	13
Info	R16	Function blocks' names should start with the prefix 'FB'	1
Info	R18	Data types conversion should be explicit: 'SINT,REAL'	15

Code overview

```

1 FUNCTION_BLOCK Test
2   VAR_INPUT
3     test_a : BOOL;
4   END_VAR
5   VAR_OUTPUT
6     a : INT;
7     b : REAL;
8     c : SINT;
9     d : BOOL;
10  END_VAR
11  BEGIN
12    test_a := TRUE;
13    IF a < 1 AND a > 20 THEN
14      b := 1;
15      a := b * c;
16    END_IF;
17  END_FUNCTION_BLOCK

```

Figure 34: Static analysis report produced by PLCverif.

H.4 Analysis and conclusions

So far the rules implemented to test the UNICOS objects are: 18 in Stratego/XT approach, 8 in Java-Spoofax and 20 in PLCverif. Yet the tool is in a primary stage and most of the rules are not that meaningful, which means, that the potential violations that they can detect do not always affect the model and its behaviour. The goal was to create the base to build more meaningful ones that can be applied in the objects and detect violations that so far were not detected by testing or model checking.

Out of the three approaches taken PLCverif seems to be the most appropriate for our use case; this is because this approach is more flexible. The grammar might not be complete but it covers all the PLC programs that need to be checked and static analysis rules can be implemented for every part of the PLC code. The implementation of the rules can be done in Java as in the Java-Spoofax approach, but in that case the AST is not complete and it needs to be implemented manually. On the contrary, in

Stratego/Xt-Spoofax approach, the AST might be complete but it is a low level one and the developer has to deal with Stratego language which is not as widespread as Java. In addition, in PLCverif, by using the same interface the developer can apply static analysis to the code or model checking to the requirements that concern the code. Moreover, even for PLC code that consists of hundreds of lines the tool has really fast execution time (usually less than 1 second).

The advantage of the tool is that the analysis procedure is transparent for the user. All that the user has to do is to import the PLC code to be analysed and click in the static analysis button of the framework. Therefore, a static analysis report will be produced and the user will be able to see all the detected violations and fix them.

Table 15: Comparison between static analysis approaches.

Approach	Grammar	Implementation	Pros	Cons
Stratego/XT	Spoofax	Stratego	Complete grammar	Low-level AST access
Java	Spoofax	Java	Simple rules	Partial AST support Manual AST manipulation
PLCverif	Xtext	Java	Simple rules	Partial grammar

I Conclusions and future work

I.1 Conclusions

This thesis set out to improve the quality of PLC code by introducing static analysis and by integrating model checking to the development process. Moreover, by implementing the abstraction algorithm technique it was able to deal partially with the state space explosion challenge that developers face while trying to guarantee that their PLC code is compliant with the given specifications. To ensure the applicability of the above three main goals were set:

- Integrate model checking in PLC development; by hiding any complexity the user now can verify more than one requirement through PLCverif automatically.
- By implement and develop the variable abstraction algorithm, partially we dealt with the state space explosion problem mostly for *Satisfied* requirements.
- Analysis of the existing static analysis tools, development of a prototype of a static analysis tool and conduct experiments for the CERN PLC programs in order to detect potential violations in the PLC code.

By connecting model checking and static analysis results, a more powerful analysis was created. Therefore, to verify the correctness of a system, both of the techniques are necessary as each of them provide different kind of analysis. By using the implemented static analysis rules to check the PLC code, we were able to detect in our real life PLC programs some inconsistent use of variables and to implement specific checks. In comparison to model checking, static analysis can be applied to the code without any special effort from the user as there is no need to express requirements to verify the behaviour of the code. At the same time, as any complexity is hidden from the user, model checking can be easily used in the development process and usually ensure the verification of the provided model. Although the conducted experiments in real life PLC programs show good results and for most of them we were able to extract a verification result, the state space explosion problem still remains.

I.2 Future work

The future work is related to overcome the problems that still remain unsolved. To deal more effectively with the state space problem new improved reduction and abstraction techniques should be applied in the current methodology. A promising one is the predicate abstraction technique that is used to prove properties of finite- and infinite-state systems. By giving a concrete finite- or infinite-state system and a set of predicates, a conservative finite state abstraction is generated and like this the state space is reduced.

Moreover, the variable abstraction algorithm would be more effective under some minor modifications in the implementation. By executing the steps of the algorithm that are not related to each other in parallel a result would be obtained faster.

Finally, the static analysis tool prototype, has to be extended with more useful rules that would be able to detect severe violations in the PLC code. After more research and discussion with the PLC developers it would be easy to extract more meaningful rules to implement.

The communities developing automation and formal methods are getting closer to achieving their goal of ensuring the correctness of PLC programs.

However there is still a way to go until we can be completely sure that a control system is robust and reliable.

Appendix II

The following piece of ST code corresponds to the Siemens implementation of the OnOff UNICOS object.

Listing 16: Generated AST for the PLC code presented in Listing 9.

```
1 //UNICOS
2 // Copyright CERN 2013 all rights reserved
3
4 (* ON/OFF OBJECT FUNCTION BLOCK
   ***** )
5
6 FUNCTION_BLOCK CPC_FB_ONOFF
7 TITLE = 'CPC_FB_ONOFF'
8 //
9 // ONOFF Object
10 //
11 VERSION: '6.6'
12 AUTHOR: 'EN/ICE'
13 NAME: 'OBJECT'
14 FAMILY: 'FO'
15
16 VAR_INPUT
17
18     HFOOn:          BOOL;
19     HFOff:          BOOL;
20     HLD:            BOOL;
21     IOError:        BOOL;
22     IOSimu:         BOOL;
23     AlB:            BOOL;
24     Manreg01:        WORD;
25     Manreg01b AT Manreg01: ARRAY [0..15] OF
        BOOL;
26     HOnR:           BOOL;
27     HOffR:          BOOL;
28     StartI:         BOOL;
29     TStopI:         BOOL;
30     FuStopI:        BOOL;
31     Al:             BOOL;
32     AuOnR:          BOOL;
33     AuOffR:         BOOL;
34     AuAuMoR:        BOOL;
35     AuIhMMo:        BOOL;
36     AuIhFoMo:       BOOL;
37     AuAlAck:        BOOL;
38     IhAuMRW:        BOOL;
39     AuRstart:       BOOL;
40     POnOff:         CPC_ONOFF_PARAM;
41     POnOffb AT POnOff: STRUCT
42     ParRegb:        ARRAY [0..15] OF
        BOOL;
43     PPulseLeb:      TIME;
44     PWDtb:          TIME;
45     END_STRUCT;
46
47
48 END_VAR
49
50 VAR_OUTPUT
51
52     Stsreg01:        WORD;
53     Stsreg01b AT Stsreg01: ARRAY [0..15] OF
        BOOL;
```

```

54     Stsreg02:                WORD;
55     Stsreg02b AT Stsreg02:   ARRAY [0..15] OF
        BOOL;
56     OutOnOV:                 BOOL;
57     OutOffOV:                BOOL;
58     OnSt:                    BOOL;
59     OffSt:                    BOOL;
60     AuMoSt:                   BOOL;
61     MMoSt:                    BOOL;
62     LDSt:                     BOOL;
63     SoftLDSt:                 BOOL;
64     FoMoSt:                   BOOL;
65     AuOnRSt:                  BOOL;
66     AuOffRSt:                 BOOL;
67     MOnRSt:                   BOOL;
68     MOffRSt:                  BOOL;
69     HOnRSt:                   BOOL;
70     HOffRSt:                  BOOL;
71     IOErrorW:                 BOOL;
72     IOSimuW:                  BOOL;
73     AuMRW:                    BOOL;
74     AlUnAck:                  BOOL;
75     PosW:                     BOOL;
76     StartISt:                 BOOL;
77     TStopISt:                 BOOL;
78     FuStopISt:                BOOL;
79     AlSt:                     BOOL;
80     AlBW:                     BOOL;
81     EnRstartSt:               BOOL := TRUE;
82     RdyStartSt:               BOOL;
83
84
85 END_VAR
86
87 VAR    //Internal Variables
88
89     //Variables for Edge detection
90     E_MAuMoR:                 BOOL;
91     E_MMMoR:                  BOOL;
92     E_MFoMoR:                 BOOL;
93     E_MOnR:                   BOOL;
94     E_MOffR:                  BOOL;
95     E_MAlAckR:                BOOL;
96     E_StartI:                 BOOL;
97     E_TStopI:                 BOOL;
98     E_FuStopI:                BOOL;
99     E_Al:                     BOOL;
100     E_AuAuMoR:                BOOL;
101     E_AuAlAck:                BOOL;
102     E_MSoftLDR:               BOOL;
103     E_MEnRstartR:             BOOL;
104     RE_AlUnAck:               BOOL;
105     FE_AlUnAck:               BOOL;
106     RE_PulseOn:               BOOL;
107     FE_PulseOn:               BOOL;
108     RE_PulseOff:              BOOL;
109     RE_OutOVSt_aux:           BOOL;
110     FE_OutOVSt_aux:           BOOL;
111     FE_InterlockR:            BOOL;
112
113     //Variables for old values
114     MAuMoR_old:                BOOL;
115     MMMoR_old:                 BOOL;

```

```

116     MFoMoR_old:          BOOL;
117     MOnR_old:            BOOL;
118     MOffR_old:           BOOL;
119     MAlAckR_old:         BOOL;
120     AuAuMoR_old:         BOOL;
121     AuAlAck_old:         BOOL;
122     StartI_old:          BOOL;
123     TStopI_old:          BOOL;
124     FuStopI_old:         BOOL;
125     Al_old:              BOOL;
126     AlUnAck_old:         BOOL;
127     MSoftLDR_old:        BOOL;
128     MEnRstartR_old:      BOOL;
129     RE_PulseOn_old:      BOOL;
130     FE_PulseOn_old:      BOOL;
131     RE_PulseOff_old:     BOOL;
132     RE_OutOVSt_aux_old:  BOOL;
133     FE_OutOVSt_aux_old:  BOOL;
134     FE_InterlockR_old:   BOOL;
135
136     //General internal variables
137     PFsPosOn:            BOOL;
138     PFsPosOn2:          BOOL;
139     PHFOn:              BOOL;
140     PHFOff:             BOOL;
141     PPulse:             BOOL;
142     PPulseCste:         BOOL;
143     PHLd:               BOOL;
144     PHLDCmd:            BOOL;
145     PAnim:              BOOL;
146     POutOff:            BOOL;
147     PEnRstart:          BOOL;
148     PRstartFS:          BOOL;
149     OutOnOVSt:          BOOL;
150     OutOffOVSt:         BOOL;
151     AuMoSt_aux:         BOOL;
152     MMoSt_aux:          BOOL;
153     FoMoSt_aux:         BOOL;
154     SoftLDSt_aux:       BOOL;
155     PulseOn:            BOOL;
156     PulseOff:           BOOL;
157     PosW_aux:           BOOL;
158     OutOVSt_aux:        BOOL;
159     fullNotAcknowledged: BOOL;
160     PulseOnR:           BOOL;
161     PulseOffR:          BOOL;
162     InterlockR:         BOOL;
163
164     //Variables for IEC Timers
165     Time_Warning:        TIME;
166     Timer_PulseOn:       TP;
167     Timer_PulseOff:      TP;
168     Timer_Warning:       TON;
169
170     //Variables for interlock Ststus delay
        handling
171     PulseWidth:         REAL;
172     FSIinc:             INT;
173     TSIinc:             INT;
174     SIinc:              INT;
175     Alinc:              INT;
176     WTStopISt:         BOOL;
177     WStartISt:         BOOL;

```

```

178     WAlSt:                                BOOL;
179     WFuStopISt:                            BOOL;
180
181 END_VAR
182
183 BEGIN
184
185 (* INPUT MANAGER *)
186
187     E_MAuMoR      := R_EDGE(new:=ManReg01b[8],
188                             old:=MAuMoR_old);          (*
189                             Manual Auto Mode Request *)
190     E_MMMoR      := R_EDGE(new:=ManReg01b[9],
191                             old:=MMMold);              (*
192                             Manual Manual Mode Request *)
193     E_MFoMoR      := R_EDGE(new:=ManReg01b[10],
194                             old:=MFoMoR_old);          (*
195                             Manual Forced Mode Request *)
196     E_MSoftLDR    := R_EDGE(new:=ManReg01b[11],
197                             old:=MSoftLDR_old);        (*
198                             Manual Software Local Drive Request *)
199     E_MOnR        := R_EDGE(new:=ManReg01b[12],
200                             old:=MOnR_old);            (*
201                             Manual On/Open Request *)
202     E_MOffR       := R_EDGE(new:=ManReg01b[13],
203                             old:=MOffR_old);          (*
204                             Manual Off/close Request *)
205     E_MEnRstartR  := R_EDGE(new:=ManReg01b[1],
206                             old:=MEnRstartR_old);      (*
207                             Manual Restart after full stop Request *)
208     E_MAlAckR     := R_EDGE(new:=ManReg01b[7],
209                             old:=MAlAckR_old);          (*
210                             Manual Alarm Ack. Request *)
211
212     PFsPosOn      := POnOffb.ParRegb[8];
213
214     (*
215     1st Parameter bit to define Fail safe
216     position behaviour *)
217     PHFOn         := POnOffb.ParRegb[9];
218
219     (*
220     Hardware feedback On present*)
221     PHFOff        := POnOffb.ParRegb[10];
222
223     (*
224     Hardware feedback Off present*)
225     PPulse        := POnOffb.ParRegb[11];
226
227     (*
228     Object is pulsed pulse duration :
229     POnOff.PulseLe*)
230     PHLD          := POnOffb.ParRegb[12];
231
232     (*
233     Local Drive mode Allowed *)
234     PHLDCmd       := POnOffb.ParRegb[13];
235
236     (*
237     Local Drive Command allowed *)
238     PAnim         := POnOffb.ParRegb[14];
239
240     (*
241     Inverted Output*)
242     POutOff       := POnOffb.ParRegb[15];
243     PEnRstart     := POnOffb.ParRegb[0];
244
245     (*
246     Enable Restart after Full Stop *)
247     PRstartFS     := POnOffb.ParRegb[1];

```



```

(*
    Enable Restart when Full Stop still
    active *)
206 PFsPosOn2 := POnOffb.ParRegb[2];

(*
    2nd Parameter bit to define Fail safe
    position behaviour *)
207 PPulseCste := POnOffb.ParRegb[3];
    (* Pulse
    Constant duration irrespective of the
    feedback status *)

208
209 E_AuAuMoR := R_EDGE(new:=AuAuMoR,old:=
    AuAuMoR_old); (* Auto
    Auto Mode Request *)
210 E_AuAlAck := R_EDGE(new:=AuAlAck,old:=
    AuAlAck_old); (* Auto
    Alarm Ack. Request *)

211
212 E_StartI := R_EDGE(new:=StartI,old:=
    StartI_old);
213 E_TStopI := R_EDGE(new:=TStopI,old:=
    TStopI_old);
214 E_FuStopI := R_EDGE(new:=FuStopI,old:=
    FuStopI_old);
215 E_Al := R_EDGE(new:=Al,old:=Al_old);
216
217 StartISt := StartI;

    (* Start Interlock present *)
218 TStopISt := TStopI;

    (* Temporary Stop Interlock present *)
219 FuStopISt := FuStopI;

    (* Full Stop Interlock present *)

220
221 (* INTERLOCK & ACKNOWLEDGE *)
222
223 IF (E_MAlAckR OR E_AuAlAck) THEN
224     fullNotAcknowledged := FALSE;
225     AlUnAck := FALSE;
226 ELSIF (E_TStopI OR E_StartI OR E_FuStopI
    OR E_Al) THEN
227     AlUnAck := TRUE;
228 END_IF;
229
230 IF ((PEnrstart AND (E_MEnRstartR OR
    AuRstart) AND NOT FuStopISt) OR (
    PEnRstart AND PRstartFS AND (
    E_MEnRstartR OR AuRstart))) AND NOT
    fullNotAcknowledged THEN
231     EnRstartSt := TRUE;
232 END_IF;
233
234 IF E_FuStopI THEN
235     fullNotAcknowledged := TRUE;
236     IF PEnRstart THEN
237         EnRstartSt := FALSE;
238     END_IF;
239 END_IF;
240
241 InterlockR := TStopISt OR FuStopISt OR

```

```

242      FullNotAcknowledged OR NOT EnRstartSt OR
      (StartIst AND NOT POutOff
      AND NOT OutOnOV) OR
243      (StartIst AND POutOff AND ((
      PFsPosOn AND OutOVSt_aux)
      OR (NOT PFsPosOn AND NOT
      OutOVSt_aux)));
244
245      FE_InterlockR := F_EDGE (new:=InterlockR,
      old:=FE_InterlockR_old);
246
247      (* MODE MANAGER *)
248
249      IF NOT (HLD AND PHL) THEN
250
251          (* Forced Mode *)
252          IF (AuMoSt_aux OR MMoSt_aux OR
      SoftLDSt_aux) AND
253          E_MFoMoR AND NOT(AuIhFoMo)
      THEN
254              AuMoSt_aux := FALSE;
255              MMoSt_aux := FALSE;
256              FoMoSt_aux := TRUE;
257              SoftLDSt_aux := FALSE;
258          END_IF;
259
260          (* Manual Mode *)
261          IF (AuMoSt_aux OR FoMoSt_aux OR
      SoftLDSt_aux) AND
262          E_MMMoR AND NOT(AuIhMMo) THEN
263              AuMoSt_aux := FALSE;
264              MMoSt_aux := TRUE;
265              FoMoSt_aux := FALSE;
266              SoftLDSt_aux := FALSE;
267          END_IF;
268
269          (* Auto Mode *)
270          IF (MMoSt_aux AND (E_MAuMoR OR
      E_AuAuMoR )) OR
271          (FoMoSt_aux AND E_MAuMoR) OR
272          (SoftLDSt_aux AND E_MAuMoR) OR
273          (MMoSt_aux AND AuIhMMo) OR
274          (FoMoSt_aux AND AuIhFoMo) OR
275          (SoftLDSt_aux AND AuIhFoMo) OR
276          NOT(AuMoSt_aux OR MMoSt_aux OR
      FoMoSt_aux OR SoftLDSt_aux
      ) THEN
277              AuMoSt_aux := TRUE;
278              MMoSt_aux := FALSE;
279              FoMoSt_aux := FALSE;
280              SoftLDSt_aux := FALSE;
281          END_IF;
282
283          (* Software Local Mode *)
284          IF (AuMoSt_aux OR MMoSt_aux) AND
      E_MSoftLDR AND NOT AuIhFoMo
      THEN
285              AuMoSt_aux := FALSE;
286              MMoSt_aux := FALSE;
287              FoMoSt_aux := FALSE;
288              SoftLDSt_aux:= TRUE;
289          END_IF;
290

```

```

291      (* Status setting *)
292      LDSt      := FALSE;
293      AuMoSt    := AuMoSt_aux;
294      MMoSt     := MMoSt_aux;
295      FoMoSt    := FoMoSt_aux;
296      SoftLDSt  := SoftLDSt_aux;
297  ELSE
298      (* Local Drive Mode *)
299      AuMoSt    := FALSE;
300      MMoSt     := FALSE;
301      FoMoSt    := FALSE;
302      LDSt      := TRUE;
303      SoftLDSt  := FALSE;
304  END_IF;
305
306  (* LIMIT MANAGER *)
307
308      (* On/Open Evaluation *)
309      OnSt:= (HFOn AND PHFOn) OR
310
311      (*
312      Feedback ON present*)
313      (NOT PHFOn AND PHFOff AND PAnim
314      AND NOT HFOff) OR  (*
315      Feedback ON not present and
316      PAnim = TRUE*)
317      (NOT PHFOn AND NOT PHFOff AND
318      OutOVSt_aux);
319
320      (* Off/Closed Evaluation *)
321      OffSt:=(HFOff AND PHFOff) OR
322
323      (*
324      Feedback OFF present*)
325      (NOT PHFOff AND PHFOn AND PAnim
326      AND NOT HFOn) OR  (*
327      Feedback OFF not present and
328      PAnim = TRUE*)
329      (NOT PHFOn AND NOT PHFOff AND
330      NOT OutOVSt_aux);
331
332  (* REQUEST MANAGER *)
333
334      (* Auto On/Off Request*)
335
336      IF AuOffR THEN
337          AuOnRSt := FALSE;
338      ELSIF AuOnR THEN
339          AuOnRSt := TRUE;
340      ELSIF fullNotAcknowledged OR FuStopISt
341          OR NOT EnRstartSt THEN
342          AuOnRSt := PFsPosOn;
343      END_IF;
344      AuOffRSt:= NOT AuOnRSt;
345
346      (* Manual On/Off Request*)
347
348      IF (((E_MOffR AND (MMoSt OR FoMoSt OR
349      SoftLDSt))
350      OR (AuOffRSt AND AuMoSt)
351      OR (LDSt AND PHLDCmd AND HOffRSt)
352      OR (FE_PulseOn AND PPulse AND NOT
353      POutOff) AND EnRstartSt)

```

```

339         OR (E_FuStopI AND NOT PFsPosOn))
340         THEN
341             MOnRSt := FALSE;
342
343     ELSIF (((E_MOnR AND (MMoSt OR FoMoSt
344             OR SoftLDSt))
345             OR (AuOnRSt AND AuMoSt)
346             OR (LDSt AND PHLDCmd AND HOnRSt)
347             AND EnRstartSt)
348             OR (E_FuStopI AND PFsPosOn)) THEN
349         MOnRSt := TRUE;
350     END_IF;
351
352     MOffRSt:= NOT MOnRSt;
353
354     (* Local Drive Request *)
355
356     IF HOffR THEN
357         HOnRSt := FALSE;
358     ELSE IF HOnR THEN
359         HOnRSt := TRUE;
360     END_IF;
361     END_IF;
362     HOffRSt := NOT(HOnRSt);
363
364     (* PULSE REQUEST MANAGER*)
365     IF PPulse THEN
366         IF InterlockR THEN
367             PulseOnR:= (PFsPosOn AND NOT
368                 PFsPosOn2) OR (PFsPosOn AND
369                 PFsPosOn2);
370             PulseOffR:= (NOT PFsPosOn AND NOT
371                 PFsPosOn2) OR (PFsPosOn AND
372                 PFsPosOn2);
373             ELSIF FE_InterlockR THEN (*Clear
374                 PulseOnR/PulseOffR to be sure you
375                 get a new pulse after InterlockR*)
376                 PulseOnR:= FALSE;
377                 PulseOffR:= FALSE;
378                 Timer_PulseOn (IN:=FALSE,PT:=T#0s);
379                 Timer_PulseOff (IN:=FALSE,PT:=T#0s);
380             ELSIF (MOffRSt AND (MMoSt OR FoMoSt OR
381                 SoftLDSt)) OR (AuOffRSt AND AuMoSt)
382                 OR (HOffR AND LDSt AND PHLDCmd)
383                 THEN //Off Request
384                 PulseOnR:= FALSE;
385                 PulseOffR:= TRUE;
386             ELSIF (MOnRSt AND (MMoSt OR FoMoSt OR
387                 SoftLDSt)) OR (AuOnRSt AND AuMoSt)
388                 OR (HOnR AND LDSt AND PHLDCmd) THEN
389                 //On Request
390                 PulseOnR:= TRUE;
391                 PulseOffR:= FALSE;
392             ELSE
393                 PulseOnR:= FALSE;
394                 PulseOffR:= FALSE;
395             END_IF;
396
397         //Pulse functions

```

```

387     Timer_PulseOn (IN:= PulseOnR,PT:=
          POnOffb.PPulseLeb);
388     Timer_PulseOff (IN:=PulseOffR,PT:=
          POnOffb.PPulseLeb);
389
390     RE_PulseOn    := R_EDGE(new:=PulseOn,
          old:=RE_PulseOn_old);
391     FE_PulseOn    := F_EDGE(new:=PulseOn,
          old:=FE_PulseOn_old);
392     RE_PulseOff   := R_EDGE(new:=PulseOff,
          old:=RE_PulseOff_old);
393
394     //The pulse functions have to be reset
          when changing from On to Off
395     IF RE_PulseOn THEN
396         Timer_PulseOff (IN:=FALSE,PT:=T#0s)
          ;
397     END_IF;
398
399     IF RE_PulseOff THEN
400         Timer_PulseOn (IN:=FALSE,PT:=T#0s);
401     END_IF;
402
403     IF PPulseCste THEN      (* Pulse constant
          duration irrespective of feedback
          status *)
404         PulseOn  := Timer_PulseOn.Q AND NOT
          PulseOffR;
405         PulseOff := Timer_PulseOff.Q AND NOT
          PulseOnR;
406     ELSE
407         PulseOn  := Timer_PulseOn.Q AND NOT
          PulseOffR AND (NOT PHFOn OR (PHFOn
          AND NOT HFOn));
408         PulseOff := Timer_PulseOff.Q AND NOT
          PulseOnR AND (NOT PHFOff OR (PHFOff
          AND NOT HFOff));
409     END_IF;
410 END_IF;
411
412     (* Output On Request *)
413     OutOnOVSt := (PPulse AND PulseOn) OR
414                 (NOT PPulse AND ((MOnRSt
          AND (MMoSt OR FoMoSt OR
          SoftLDSt)) OR
          (AuOnRSt AND AuMoSt) OR
          (HOnRST AND LDSt AND
          PHLDCmd)));
415
416
417
418     (* Output Off Request *)
419     IF POutOff THEN
420         OutOffOVSt := (PulseOff AND PPulse)
          OR
421
          (NOT(PPulse) AND ((
          MOffRSt AND (MMoSt
          OR FoMoSt OR
          SoftLDSt)) OR (
          AuOffRSt AND
          AuMoSt) OR (
          HOffRST AND LDSt
          AND PHLDCmd)));
422
423     END_IF;

```

```

424      (* Interlocks / FailSafe *)
425
426      IF POutOff THEN
427          IF InterlockR THEN
428              IF PPulse AND NOT PFsPosOn2
429                  THEN
430                  IF PFsPosOn THEN
431                      OutOnOVSt := PulseOn;
432                      OutOffOVSt := FALSE;
433                  ELSE
434                      OutOnOVSt := FALSE;
435                      OutOffOVSt := PulseOff
436                          ;
437                  END_IF;
438              ELSE
439                  OutOnOVSt := (PFsPosOn AND
440                      NOT PFsPosOn2) OR (
441                      PFsPosOn AND PFsPosOn2);
442                  OutOffOVSt := (NOT PFsPosOn
443                      AND NOT PFsPosOn2) OR (
444                      PFsPosOn AND PFsPosOn2);
445              END_IF;
446          END_IF;
447      ELSE
448          IF InterlockR THEN
449              OutOnOVSt := PFsPosOn;
450          END_IF;
451      END_IF;
452
453      (* Ready to Start Status *)
454
455      RdyStartSt := NOT InterlockR;
456
457      (*Alarms*)
458
459      AlSt := Al;
460
461      (* SURVEILLANCE *)
462
463      (* I/O Warning *)
464      IOErrorW := IOError;
465      IOSimuW := IOSimu;
466
467      (* Auto<> Manual Warning *)
468      AuMRW := (MMoSt OR FoMoSt OR SoftLDSt)
469          AND
470          ((AuOnRSt XOR MOnRSt) OR (
471              AuOffRSt XOR MOffRSt)) AND
472          NOT IhAuMRW;
473
474      (* OUTPUT_MANAGER AND OUTPUT REGISTER *)
475      IF NOT POutOff THEN
476          IF PFsPosOn THEN
477              OutOnOV := NOT OutOnOVSt;
478          ELSE
479              OutOnOV := OutOnOVSt;
480          END_IF;
481      ELSE
482          OutOnOV := OutOnOVSt;
483          OutOffOV := OutOffOVSt;
484      END_IF;

```

```

478
479 (* Position warning *)
480
481     (* Set reset of the OutOnOVSt *)
482     IF OutOnOVSt OR (PPulse AND PulseOnR)
483         THEN
484         OutOVSt_aux := TRUE;
485     END_IF;
486     IF (OutOffOVSt AND POutOff) OR (NOT
487         OutOnOVSt AND NOT POutOff) OR (
488         PPulse AND PulseOffR) THEN
489         OutOVSt_aux := FALSE;
490     END_IF;
491
492     RE_OutOVSt_aux := R_EDGE(new:=
493         OutOVSt_aux,old:=RE_OutOVSt_aux_old)
494     ;
495     FE_OutOVSt_aux := F_EDGE(new:=
496         OutOVSt_aux,old:=FE_OutOVSt_aux_old)
497     ;
498
499     IF ((OutOVSt_aux AND ((PHFOn AND NOT OnSt)
500         OR (PHFOff AND OffSt)))
501         OR (NOT OutOVSt_aux AND ((PHFOff AND
502         NOT OffSt) OR (PHFON AND OnSt)))
503         OR (OffSt AND OnSt))
504         AND (NOT PPulse OR (POutOff AND PPulse
505         AND NOT OutOnOV AND NOT OutOffOV))
506     THEN
507         PosW_aux:= TRUE;
508     END_IF;
509
510     IF NOT ((OutOVSt_aux AND ((PHFOn AND NOT
511         OnSt) OR (PHFOff AND OffSt)))
512         OR (NOT OutOVSt_aux AND ((PHFOff AND
513         NOT OffSt) OR (PHFON AND OnSt)))
514         OR (OffSt AND OnSt))
515         OR RE_OutOVSt_aux
516         OR FE_OutOVSt_aux
517         OR (PPulse AND POutOff AND OutOnOV)
518         OR (PPulse AND POutOff AND OutOffOV)
519     THEN
520         PosW_aux := FALSE;
521     END_IF;
522
523     Timer_Warning(IN := PosW_aux,
524         PT := POnOffb.PWDtb);
525
526     PosW := Timer_Warning.Q;
527     Time_Warning := Timer_Warning.ET;
528
529 (* Alarm Blocked Warning*)
530
531     AlBW := AlB;
532
533 (* Maintain Interlock status 1.5s in Stsreg
534 for PVSS *)
535
536 PulseWidth := 1500 (* msec*) / DINT_TO_REAL(
537     TIME_TO_DINT(T_CYCLE));
538
539
540 IF FuStopIst OR FSIinc > 0 THEN

```

```

527     FSIinc := FSIinc + 1;
528     WFuStopISt := TRUE;
529 END_IF;
530
531 IF FSIinc > PulseWidth OR (NOT FuStopISt AND
    FSIinc = 0) THEN
532     FSIinc := 0;
533     WFuStopISt := FuStopISt;
534 END_IF;
535
536 IF TStopISt OR TSIinc > 0 THEN
537     TSIinc := TSIinc + 1;
538     WTStopISt := TRUE;
539 END_IF;
540
541 IF TSIinc > PulseWidth OR (NOT TStopISt AND
    TSIinc = 0) THEN
542     TSIinc := 0;
543     WTStopISt := TStopISt;
544 END_IF;
545
546 if StartISt OR SIinc > 0 THEN
547     SIinc := SIinc + 1;
548     WStartISt:= TRUE;
549 END_IF;
550
551 IF SIinc > PulseWidth OR (NOT StartISt AND
    SIinc = 0) THEN
552     SIinc := 0;
553     WStartISt := StartISt;
554 END_IF;
555
556 IF AlSt OR Alinc > 0 THEN
557     Alinc := Alinc + 1;
558     WALSt := TRUE;
559 END_IF;
560
561 IF Alinc > PulseWidth OR (NOT AlSt AND Alinc =
    0) THEN
562     Alinc := 0;
563     WALSt := AlSt;
564 END_IF;
565
566
567 (* STATUS REGISTER *)
568
569     Stsreg01b[8] := OnSt;           //
        StsReg01 Bit 00
570     Stsreg01b[9] := OffSt;         //
        StsReg01 Bit 01
571     Stsreg01b[10] := AuMoSt;        //
        StsReg01 Bit 02
572     Stsreg01b[11] := MMoSt;         //
        StsReg01 Bit 03
573     Stsreg01b[12] := FoMoSt;        //
        StsReg01 Bit 04
574     Stsreg01b[13] := LDSt;          //
        StsReg01 Bit 05
575     Stsreg01b[14] := IOErrorW;      //
        StsReg01 Bit 06
576     Stsreg01b[15] := IOSimuW;       //
        StsReg01 Bit 07
577     stsreg01b[0] := AuMRW;          //

```



```

        StsReg01 Bit 08
578     Stsreg01b[1] := PosW;           //
        StsReg01 Bit 09
579     Stsreg01b[2] := WStartISt;     //
        StsReg01 Bit 10
580     Stsreg01b[3] := WTStopISt;     //
        StsReg01 Bit 11
581     Stsreg01b[4] := AlUnAck;        //
        StsReg01 Bit 12
582     Stsreg01b[5] := AuIhFoMo;      //
        StsReg01 Bit 13
583     Stsreg01b[6] := WAlSt;         //
        StsReg01 Bit 14
584     Stsreg01b[7] := AuIhMMo;       //
        StsReg01 Bit 15
585
586     Stsreg02b[8] := OutOnOVSt;      //
        StsReg02 Bit 00
587     Stsreg02b[9] := AuOnRSt;       //
        StsReg02 Bit 01
588     Stsreg02b[10] := MOnRSt;       //
        StsReg02 Bit 02
589     Stsreg02b[11] := AuOffRSt;     //
        StsReg02 Bit 03
590     Stsreg02b[12] := MOffRSt;      //
        StsReg02 Bit 04
591     Stsreg02b[13] := HOnRSt;       //
        StsReg02 Bit 05
592     Stsreg02b[14] := HOffRSt;     //
        StsReg02 Bit 06
593     Stsreg02b[15] := 0;           //
        StsReg02 Bit 07
594     stsreg02b[0] := 0;            //
        StsReg02 Bit 08
595     Stsreg02b[1] := 0;            //
        StsReg02 Bit 09
596     Stsreg02b[2] := WFuStopISt ;   //
        StsReg02 Bit 10
597     Stsreg02b[3] := EnRstartSt;    //
        StsReg02 Bit 11
598     Stsreg02b[4] := SoftLDSt;      //
        StsReg02 Bit 12
599     Stsreg02b[5] := AlBW;          //
        StsReg02 Bit 13
600     Stsreg02b[6] := OutOffOVSt;    //
        StsReg02 Bit 14
601     Stsreg02b[7] := 0;            //
        StsReg02 Bit 15
602
603 (* Edges *)
604
605     DETECT_EDGE(new:=AlUnAck,old:=AlUnAck_old,re:
        =RE_AlUnAck,fe:=FE_AlUnAck);
606
607
608 END_FUNCTION_BLOCK

```

The following tables correspond to the requirements which were created for the need of the experiments presented in Chapter 4.

Table 16: PCO requirements tested with nuXmv'IC3.

No	Requirements
1	If RunOSt AND FOff is true at the end of the PLC cycle, then OffSt should always be true at the end of the same cycle.
2	If RunOSt AND Fon is true at the end of the PLC cycle, then OnSt should always be true at the end of the same cycle.
3	If OnSt is true at the end of the PLC cycle, then NOT OffSt should always be true at the end of the same cycle.
4	If CStopOSt is true at the end of the PLC cycle, then OffSt should always be true at the end of the same cycle.
5	If CStopOSt is true at the end of the PLC cycle, then NOT OnSt should always be true at the end of the same cycle.
6	If OnSt is true at the end of the PLC cycle, then RunOSt should always be true at the end of the same cycle.
7	If RunOSt is true at the end of the PLC cycle, then NOT OffSt should always be true at the end of the same cycle.
8	If AuRunOrder AND AuMoSt and NOT TStopI is true at the end of the PLC cycle, then RunOSt should always be true at the end of the same cycle.
9	If AuRunOrder AND AuMoSt is true at the end of the PLC cycle, then RunOSt should always be true at the end of the same cycle.
10	If AuRunOrder AND AuMoSt is true at the end of the PLC cycle, then OnSt should always be true,at the end of the same cycle.
11	If CStopOSt is true at the end of the PLC cycle, then NOT RunOSt should always be true at the end of the same cycle.
12	If NOT Fon is true at the end of the PLC cycle, then NOT OnSt should always be true at the end of the same cycle.
13	If NOT RunOSt is true at the end of the PLC cycle, then NOT CStopOSt should always be true at the end of the same cycle.
14	If TStopI is true at the end of the PLC cycle, then RunOSt should always be true at the end of the same cycle.
15	If AuMoSt AND NOT TStopI is true at the end of the PLC cycle, then RunOSt should always be true at the end of the same cycle.
16	If NOT Fon is true at the end of the PLC cycle, then OffSt should always be true at the end of the same cycle.
17	If NOT FOff is true at the end of the PLC cycle, then OnSt should always be true at the end of the same cycle.
18	If TStopI is true at the end of the PLC cycle, then NOT RunOnSt should always be true at the end of the same cycle.

Table 17: PCO requirements tested with nuXmv.

No	Requirements
1	If RunOSt AND FOff is true at the end of the PLC cycle, then OffSt should always be true at the end of the same cycle.
2	If NOT CStopFin AND AuCStopR AND AuMoSt AND RunOSt is true at the end of the PLC cycle, then CStopOSt should always be true at the end of the same cycle.
3	If OnSt is true at the end of the PLC cycle, then NOT OffSt should always be true at the end of the same cycle.
4	If NOT CStopFin AND AuCStopR AND AuMoSt AND RE'RunOSt is true at the end of the PLC cycle, then CStopOSt should always be true at the end of the same cycle.
5	If CStopOSt is true at the end of the PLC cycle, then NOT OnSt should always be true at the end of the same cycle.
6	If OnSt is true at the end of the PLC cycle, then RunOSt should always be true at the end of the same cycle.
7	If RunOSt is true at the end of the PLC cycle, then NOT OffSt should always be true at the end of the same cycle.
8	If AuRunOrder AND AuMoSt and NOT TStopI is true at the end of the PLC cycle, then RunOSt should always be true at the end of the same cycle.
9	If AuRunOrder AND AuMoSt is true at the end of the PLC cycle, then OnSt should always be true at the end of the same cycle.
10	If AuMoSt is true at the end of the PLC cycle, then NOT MMoSt should always be true at the end of the same cycle.
11	If NOT Fon is true at the end of the PLC cycle, then OffSt should always be true at the end of the same cycle.
12	If NOT Fon is true at the end of the PLC cycle, then NOT OnSt should always be true at the end of the same cycle.

Table 18: Analog requirements tested with nuXmv and nuXmv'IC3.

No	Requirements
1	If NOT(PFsPosOn AND NOT PFsNOut) is true at the end of the PLC cycle, then OutOV=PosRSt should always be true at the end of the same cycle.
2	If AuMoSt is true at the end of the PLC cycle, then NOT MMoSt should always be true at the end of the same cycle.
3	If AuIhFoMo OR E'MAuMoR is true at the end of the PLC cycle, then AuMoSt should always be true at the end of the same cycle.
4	If HFPos=10 AND PHFPos is true at the end of the PLC cycle, then PosSt=10 should always be true at the end of the same cycle.
5	If PHFPos is true at the end of the PLC cycle, then PosSt=HFPos should always be true at the end of the same cycle.
6	If PHFOff AND HFOff AND NOT(PHFOn AND HFOn) AND NOT PHFPos is true at the end of the PLC cycle, then PosSt=PAalog.PMinRan should always be true at the end of the same cycle.
7	If PosSt=20 AND PliOn=10 is true at the end of the PLC cycle, then OnSt should always be true at the end of the same cycle.
8	If PosSt=10 AND PliOff=20 is true at the end of the PLC cycle, then OffSt should always be true at the end of the same cycle.
9	If PliOff=30 AND PosSt=20 AND PliOn=10 is true at the end of the PLC cycle, then OnSt should always be true at the end of the same cycle.
10	If PliOff=30 AND PosSt=20 AND PliOn=10 is true at the end of the PLC cycle, then OffSt should always be true at the end of the same cycle.
11	If PliOff=20 AND PosSt=10 AND PliOn=12 is true at the end of the PLC cycle, then OnSt should always be true at the end of the same cycle.
12	If PliOff=20 AND PosSt=10 AND PliOn=12 is true at the end of the PLC cycle, then NOT OffSt should always be true at the end of the same cycle.
13	If AuMoSt is true at the end of the PLC cycle, then PosR=AuPosRSt should always be true at the end of the same cycle.

Table 19: OnOff requirements tested with nuXmv'IC3.

No	Requirements
1	If (HFOn AND PHFOn) OR (NOT PHFOn AND PHFOff AND PAnim,AND NOT HFOff) is true at the end of the,PLC cycle, then OnSt should always be true at the end of the same cycle.
2	If (HFOff AND PHFOff) OR (NOT PHFOff AND PHFOn AND PAnim AND NOT HFOn) is true at the end of the PLC cycle, then OffSt should always be true at the end of the same cycle.
3	If NOT OutOnOV AND POutOff is true at the end of the PLC cycle, then OutOffOV should always be true at the end of the same cycle.
4	If NOT POutOff AND NOT PFsPosOn is true at the end of the PLC cycle, then OutOnOV= OutOnOVSt should always be true at the end of the same cycle.
5	If RdyStartSt is true at the end of the PLC cycle, then NOT InterlockR should always be true at the end of the same cycle.
6	If AuMoSt is true at the end of the PLC cycle, then OutOnOVSt=AuOnR should always be true at the end of the same cycle.
7	If OutOnOVSt is true at the end of the PLC cycle, then OutOnOV should always be true at the end of the same cycle.
8	If POutOff is true at the end of the PLC cycle, then OutOffOV should always be true at the end of the same cycle.
9	If NOT OutOnOV is true at the end of the PLC cycle, then OutOffOV should always be true at the end of the same cycle.
10	If PFsPosOn is true at the end of the PLC cycle, then OutOnOV should always be true at the end of the same cycle.
11	If HFOn AND PHFOn is true at the end of the PLC cycle, then OnSt should always be true at the end of the same cycle.
12	If HFOff AND PHFOff is true at the end of the PLC cycle, then OffSt should always be true at the end of the same cycle.
13	If NOT PHFOn AND PHFOff AND PAnim AND NOT HFOff is true at the end of the PLC cycle, then OnSt should always be true at the end of the same cycle.
14	If NOT PHFOff AND PHFOn AND PAnim AND NOT HFOn is true at the end of the PLC cycle, then OffSt should always be true at the end of the same cycle.
15	If NOT PHFOn AND PHFOff AND PAnim AND NOT HFOff is true at the end of the PLC cycle, then NOT OffSt should always be true at the end of the same cycle.
16	If NOT PHFOff AND PHFOn AND PAnim AND NOT HFOn is true at the end of the PLC cycle, then NOT OnSt should always be true at the end of the same cycle.

Table 20: OnOff requirements tested with nuXmv.

No	Requirements
1	If (HFOn AND PHFOn) OR (NOT PHFOn AND PHFOff AND PAnim AND NOT HFOff) is true at the end of the PLC cycle, then OnSt should always be true at the end of the same cycle.
2	If (HFOff AND PHFOff) OR (NOT PHFOff AND PHFOn AND PAnim AND NOT HFOn) is true at the end of the PLC cycle, then OffSt should always be true at the end of the same cycle.
3	If NOT OutOnOV AND POutOff is true at the end of the PLC cycle, then OutOffOV should always be true at the end of the same cycle.
4	If NOT POutOff AND NOT PFsPosOn is true at the end of the PLC cycle, then OutOnOV = OutOnOVSt should always be true at the end of the same cycle.
5	If RdyStartSt is true at the end of the PLC cycle, then NOT InterlockR should always be true at the end of the same cycle.
6	If AuMoSt is true at the end of the PLC cycle, then OutOnOVSt = AuOnR should always be true at the end of the same cycle.
7	If PFsPosOn is true at the end of the PLC cycle, then OutOnOV should always be true at the end of the same cycle.
8	If HFOn AND PHFOn is true at the end of the PLC cycle, then OnSt should always be true at the end of the same cycle.
9	If HFOff AND PHFOff is true at the end of the PLC cycle, then OffSt should always be true at the end of the same cycle.
10	If NOT PHFOn AND PHFOff AND PAnim AND NOT HFOff is true at the end of the PLC cycle, then OnSt should always be true at the end of the same cycle.
11	If NOT PHFOff AND PHFOn AND PAnim AND NOT HFOn is true at the end of the PLC cycle, then OffSt should always be true at the end of the same cycle.
12	If NOT PHFOn AND PHFOff AND PAnim AND NOT HFOff is true at the end of the PLC cycle, then NOT OffSt should always be true at the end of the same cycle.
13	If NOT PHFOff AND PHFOn AND PAnim AND NOT HFOn is true at the end of the PLC cycle, then NOT OnSt should always be true at the end of the same cycle.

References

- [1] SIEMENS STEP 7. *Programming Guideline for S7-1200/1500*, 2014.
- [2] Nathaniel Ayewah, William Pugh, J. David Morgenthaler, John Penix, and YuQian Zhou. Evaluating static analysis defect warnings on production software. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '07, pages 1–8. ACM, 2007.
- [3] T. Barthá, A. Voros, A. Jambor, and D. Darvas. Verification of an industrial safety function using coloured Petri nets and model checking. In *14th International Conference on Modern Information Technology in the Innovation Processes of the Industrial Enterprises*, 2012.
- [4] N. Bauer, R. Huuck, S. Engell, S. Lohmann, B. Lukoschus, M. Remelhe, and O. Stursberg. Verification of PLC programs given as sequential function charts.

- In *Integration of Software Specification Techniques for Applications in Engineering*. Springer, 2004.
- [5] D.F. Bender, B. Combemale, X. Crégut, J.-M. Farines, B. Berthomieu, and F. Vernadat. Ladder metamodeling and PLC program validation through time Petri nets. In *Model Driven Architecture – Foundations and Applications*, pages 121–136. Springer, 2008.
 - [6] S. Biallas, J. Brauer, and S. Kowalewski. Arcade.PLC: A verification platform for Programmable Logic Controllers. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2012.
 - [7] S. Biallas, M. Giacobbe, and S. Kowalewski. Predicate abstraction for Programmable Logic Controllers. In *FMICS 2013 Proceedings of the 18th International Workshop on Formal Methods for Industrial Critical Systems*, pages 123–138. Springer, 2013.
 - [8] J. Creissac Campos, J. Machado, and E. Seabra. Property patterns for the formal verification of automated production systems. In *Proceedings of the 17th World Congress The International Federation of Automatic Control*. IFAC, 2008.
 - [9] G. Canet, S. Couffin, J.-J. Lesage, A. Petit, and Ph. Schnoebelen. Towards the automatic verification of PLC programs written in instruction list. In *IEEE International Conference on Systems, Man, and Cybernetics*, pages 2449 – 2454. IEEE, 2000.
 - [10] R. Cavada, A. Cimatti, G. Keighren, E. Olivetti, M. Pistore, and M. Roveri. NuSMV tutorial. <http://nusmv.fbk.eu/NuSMV/tutorial/v26/tutorial.pdf>.
 - [11] CERN. UNICOS, 1998. <http://unicos.web.cern.ch/>.
 - [12] E.M. Clarke, W. Klieber, M. Novacek, and P. Zulian. Model checking and the state explosion problem. In *Tools for Practical Software Verification: LASER, International Summer School 2011, Elba Island, Italy, Revised Tutorial Lectures*, pages 1–30. Springer, 2011.
 - [13] CMU. Model checking @ CMU. <https://www.cs.cmu.edu/~modelcheck/>.
 - [14] Science & Technology Facilities Council. CERN Accelerator Complex. <http://www.stfc.ac.uk/research/particle-physics-and-particle-astronomy/large-hadron-collider/cern-accelerator-complex/>.
 - [15] D. Darvas, B. Fernández Adiego, A. Vóros, T. Bartha, E. Blanco Viñuela, and V. M. González Suárez. Formal verification of complex properties on PLC programs. In Erika Ábrahám and Catuscia Palamidessi, editors, *Formal Techniques for Distributed Objects, Components, and Systems*, LNCS, pages 284–299. Springer, 2014.
 - [16] D. Darvas. PLCverif: A tool to verify PLC programs based on model checking techniques. In *Proceedings, 15th International Conference on Accelerator and Large Experimental Physics Control Systems (ICALEPCS 2015): Melbourne, Australia, October 17-23, 2015*, page WEPGF092, 2015.
 - [17] M. D. Ernst. Static and dynamic analysis: synergy and duality, 2003. <http://homes.cs.washington.edu/~mernst/pubs/staticdynamic-woda2003.pdf>.
 - [18] Formal Methods Europe. Formal methods. http://www.fmeurope.org/?page_id=2.
 - [19] M. Eysholdt and H. Behrens. Xtext: Implement your language faster than the quick and dirty way. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, pages 307–309. ACM New York, 2010.
 - [20] Borja Fernández Adiego. *Bringing automated formal verification to PLC program development*. PhD thesis, University of Oviedo, CERN, 2014.

- [21] Borja Fernández Adiego, Daniel Darvas, Enrique Blanco Vizueta, Jean-Charles Tournier, Víctor M. González Suárez, and Jan Olaf Blech. Modelling and formal verification of timing aspects in large PLC programs. In Edward Boje and Xiaohua Xia, editors, *Proceedings of the 19th IFAC World Congress 2014*, pages 3333–3339, 2014.
- [22] Stephan Flake, Wolfgang Müller, Ulrich Pape, and Jürgen Ruf. Specification and formal verification of temporal properties of production automation systems. In Hartmut Ehrig, Werner Damm, Jörg Desel, Martin GroSSe-Rhode, Wolfgang Reif, Eckehard Schnieder, and Engelbert Westkämper, editors, *Integration of Software Specification Techniques for Applications in Engineering: Priority Program SoftSpez of the German Research Foundation (DFG), Final Report*, pages 206–226. Springer, 2004.
- [23] Mario Gleirscher, Dmitriy Golubitskiy, Maximilian Irlbeck, and Stefan Wagner. On the benefit of automated static analysis for small and medium-sized software enterprises. In Stefan Biffl, Dietmar Winkler, and Johannes Bergsmann, editors, *Software Quality. Process Automation in Software Development: 4th International Conference, SWQD 2012, Vienna, Austria, January 17-19, 2012. Proceedings*, pages 14–38. Springer, 2012.
- [24] V. Gourcuff, O. de Smet, and J.-M. Faure. Efficient representation for formal verification of PLC programs. In *2006 8th International Workshop on Discrete Event Systems*, pages 182–187. IEEE, 2006.
- [25] Vincent Gourcuff, Olivier De Smet, and Jean-Marc Faure. Improving large-sized PLC programs verification using abstractions. In *17th IFAC World Congress, Seoul (Korea)*, pages 10.3182/20080706-5-KR-1001.1584, South Korea, July 2008.
- [26] A. Grimmer, F. Angerer, H. Prahofner, and P. Grunbacher. Supporting program analysis for non-mainstream languages: Experiences and lessons learned. In *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEE, 2016.
- [27] R. Huuck. Software verification for Programmable Logic Controllers, 2002.
- [28] Javarevisited. Why static code analysis is important. <http://javarevisited.blogspot.ch/2014/02/why-static-code-analysis-is-important.html>.
- [29] N. Kikuchu and T. Kikuno. Improving the testing process by program static analysis, 2002.
- [30] M. Kot. The state explosion problem. <http://www.cs.vsb.cz/kot/download/Texts/StateSpace.pdf>.
- [31] Tim Lange, Martin R. Neuhäusser, and Thomas Noll. Speeding up the safety verification of Programmable Logic Controller code. In Valeria Bertacco and Axel Legay, editors, *Hardware and Software: Verification and Testing: 9th International Haifa Verification Conference, HVC 2013, Haifa, Israel, November 5-7, 2013, Proceedings*, pages 44–60, Cham, 2013. Springer International Publishing.
- [32] M. Lettrich. Prototype of automated PLC Model Checking using continuous integration tools, 2015. <https://cds.cern.ch/record/2056706/files/SummerStudentReport.pdf>.
- [33] A. Mader and H. Wupper. Verification of PLC programs given as sequential function charts. In *Integration of Software Specification Techniques for Applications in Engineering*. Springer, 1999.
- [34] MetaBorg. The Spoofox language workbench. <http://metaborg.org/en/latest/>.
- [35] H.B. Mokadem, B. Berard, V. Gourcuff, O. De Smet, and J.-M. Rousel. Verification of a timed multitask system with UPPAAL. In *IEEE Transactions on Automation Science and Engineering*, pages 921–932. IEEE, 2010.

- [36] I. Moon. Modeling Programmable Logic Controllers for logic verification. In *IEEE control systems*. IEEE, 1994. <http://sdg.csail.mit.edu/6.894/dnjPapers/hall-cdis.pdf>.
- [37] Nachiappan Nagappan, Thomas Ball, Tom Ball, and Nachi Nagappan. Static analysis tools as early indicators of pre-release defect density. In *Proceedings of the 27th international conference on Software engineering*, pages 580–586. Association for Computing Machinery, Inc., May 2005.
- [38] Stanford Encyclopedia of Philosophy. Temporal logic. <https://plato.stanford.edu/entries/logic-temporal/>.
- [39] UNICOS CPC Resource Package. Analog. <http://ucpc-resources.web.cern.ch/ucpc-resources/1.8.0/objects/analog.html>.
- [40] UNICOS CPC Resource Package. OnOff. <http://ucpc-resources.web.cern.ch/ucpc-resources/1.8.0/objects/on-off.html>.
- [41] UNICOS CPC Resource Package. PCO (Process Control Object). <http://ucpc-resources.web.cern.ch/ucpc-resources/1.8.0/objects/pco.html>.
- [42] O. Pavlovic and H.-D. Ehrich. Model checking PLC software written in Function Block Diagram. In *2010 Third International Conference on Software Testing, Verification and Validation*, pages 439 – 489. IEEE, 2010.
- [43] O. Pavlovic, R. Pinger, and M. Kollmann. Automated formal verification of PLC programs written in IL. In *Proceedings of 4th International Verification Workshop in connection with CADE-21*, 2007. <https://pdfs.semanticscholar.org/ad01/c0ed7598223deae48a34ca6b946d5471ec64.pdf>.
- [44] M. Perin and J.M Faure. Building meaningful timed models of closed-loop –DES” for verification purposes. *Control Engineering Practice*, pages 1620 – 1639, 2013. Advanced Software Engineering in Industrial Automation (IN-COM’09).
- [45] H. Prahofer, F. Angerer, R. Ramler, H. Lacheiner, and F. Grillenberger. Research & implementation of a tool for iec 61131-3 languages, 2012.
- [46] Praxis. Using formal methods to develop an ATC information system. In *IEEE Software*, pages 66–76. IEEE, 1996.
- [47] H. Prahofer, F. Angerer, R. Ramler, H. Lacheiner, and F. Grillenberger. Opportunities and challenges of static code analysis of IEC 61131-3 programs. In *Emerging Technologies & Factory Automation (ETFA), 2012 IEEE 17th Conference, Krakow*. IEEE, 2012.
- [48] M. Rausch and B.H. Krogh. Formal verification of PLC programs. In *Proceedings of the American Control Conference*. IEEE, 1998.
- [49] Daniela Remenska, Tim A. C. Willemse, Jeff Templon, Kees Verstoep, and Henri Bal. Property specification made easy: Harnessing the power of model checking in UML designs. In Erika Abraham and Catuscia Palamidessi, editors, *Formal Techniques for Distributed Objects, Components, and Systems: 34th IFIP WG 6.1 International Conference, FORTE 2014, Held as Part of the 9th International Federated Conference on Distributed Computing Techniques, DisCoTec 2014, Berlin, Germany, June 3-5, 2014. Proceedings*, pages 17–32. Springer, 2014.
- [50] C.A Sarmento, J.R. Silva, P.E. Miyagi, and D.J.S Filho. Modeling of programs and its verification for Programmable Logic Controllers. In *Proceedings of the 17th World Congress*. IFAC, 2008.
- [51] Bastian Schlich, J. Brauer, J. Wernerus, and Stefan Kowalewski. Direct model checking of –PLC” programs in –IL”. *–IFAC” Proceedings Volumes*, pages 28 – 33, 2009. 2nd –IFAC” Workshop on Dependable Control of Discrete Systems.
- [52] Bastian Schlich, Jörg Brauer, Jörg Wernerus, and Stefan Kowalewski. *Direct Model Checking of –PLC” Programs in –IL”*, 2009. 2nd –IFAC” Workshop on Dependable Control of Discrete Systems.

- [53] V. Romero Segovia and A. Theorin. History of control history of PLC and DCS, 2012. http://www.control.lth.se/media/Education/DoctorateProgram/2012/HistoryOfControl/Vanessa_Alfred_report.pdf.
- [54] D. Soliman and G. Frey. Verification and validation of safety applications based on PLCopen safety function blocks using timed automata in Uppaal. In *IFAC Proceedings Volumes*, 2011.
- [55] D. Steinberg and F. Budinsky. *EMF: Eclipse Modeling Framework 2.0*. Addison-Wesley Professional, 2009.
- [56] G. Whalen. The father of the PLC – Dick Morley. <http://pattiengineering.com/blog/father-plc-dick-morley/>.
- [57] Wikipedia. Abstract syntax tree. https://en.wikipedia.org/wiki/Abstract_syntax_tree.
- [58] Wikipedia. Mars climate orbiter. https://en.wikipedia.org/wiki/Mars_Climate_Orbiter.
- [59] Wikipedia. Programmable logic controller. https://en.wikipedia.org/wiki/Programmable_logic_controller.
- [60] Wikipedia. Temporal logic. https://en.wikipedia.org/wiki/Temporal_logic.
- [61] Wikipedia. Therac-25. <https://en.wikipedia.org/wiki/Therac-25>.
- [62] Wikipedia. XPath. <https://en.wikipedia.org/wiki/XPath>, Accessed on 23/12/2016.
- [63] J. Yoo, S. Cha, and E. Jee. A verification framework for FBD based software in nuclear power plants. In *Software Engineering Conference*, pages 385–392. IEEE, 2008.
- [64] J. Yoo, S. Cha, and E. JEE. Verification of PLC programs written in FBD with VIS. In *Software Engineering and Formal Methods: 9th International Conference*. Springer, 2008.
- [65] J. Zheng, L. Williams, N. Nagappan, W. Snipes, J.P. Hudepohi, and M.A Vouk. On the value of static analysis for fault detection in software. In *IEEE Transactions on Software Engineering*, pages 240–253. IEEE, 2006.