Proving Equivalence between Control Software Variants for Programmable Logic Controllers

Using Regression Verification to Reduce Unneeded Variant Diversity

Sebastian Ulewicz, Birgit Vogel-Heuser
Technische Universität München
Institute for Automation and Information Systems
Munich, Germany
{ulewicz; vogel-heuser}@ais.mw.tum.de

Mattias Ulbrich, Alexander Weigl, Bernhard Beckert
Karlsruhe Institute of Technology
Institute of Theoretical Informatics
Karlsruhe, Germany
{ulbrich; weigl; beckert}@kit.edu

***Abstract*—Automated production systems are usually driven by Programmable Logic Controllers (PLCs). These systems are long-living and have high requirements for software quality to avoid downtimes, damaged product and harm to personnel. While commissioning multiple systems of similar type, pragmatic adjustments of the software are often necessary, which results in two or more similar variants of initially identical software. For further evolution of the software, an equivalence analysis of the software's behavior is beneficial to merge divergent development branches into a single program version. This paper presents a novel method for regression verification of PLC code, which allows one to prove that two variants of a plant's software behave identically in specified situations, despite being implemented differently. For this, a regression verification method for PLC code was designed, implemented and evaluated. The notion of program equivalence for reactive PLC code is clarified and defined. Core elements of the method are the translation of PLC code into the SMV input language for model checkers, the adaptation of the coupling invariants concept to reactive systems, and the implementation of a toolchain using a model checker. The approach was successfully evaluated using the Pick-and-Place Unit benchmark case study.**

Keywords*—*Manufacturing automation; Formal verification; Software quality; Software maintenance

I. Introduction

Automated production systems (aPS) [1], such as industrial manufacturing plants, are usually automated with *Programmable Logic Controllers* (PLCs). These computing devices are specially tailored to controlling automated production systems in dependable or safety-critical real time environments. A malfunction may cause severe damage to the system itself or to the payload, or even harm persons within the reach of the system. This results in high quality requirements on the software, which is commonly ensured by software testing. In many cases, the software cannot be tested as a whole before commissioning, i.e. the installation and initial startup of the machine, as the software functionality relies on feedback from the hardware and the controlled technical process. As commissioning is driven by meeting deadlines, this process is very straining on involved personnel and often results in quick and pragmatic fixes of software faults. For multiple systems of the same type, similar automation tasks (functions) are programmed by different engineers at different customer sites during start-up, but are oftentimes not transferred back to a base version or the other systems. This results in multiple variants of code snippets with similar behavior, yet different implementation. For further evolution of the systems, high numbers of variants increase the effort for implementation and testing. It is thus economically useful to keep the number of similar variants low. In these cases, an analysis of the behavioral equivalence of the two or more snippets programmed on-site is beneficial to decide whether the snippets are really equivalent in behavior and – if this is the case – which of the snippets should be used in the future to reduce similar variants to the best variant. Another use case with the same resulting challenge is parallel development of code parts for basic functions due to lack of communication between teams, e.g. an implementation of a new drive in different teams for different plant types. This cannot be achieved by simple differencing of code: As shown in Fig. 1, code can exhibit equivalent behavior with different implementations. In this paper, a formal regression verification method and toolchain are presented which approach this problem.

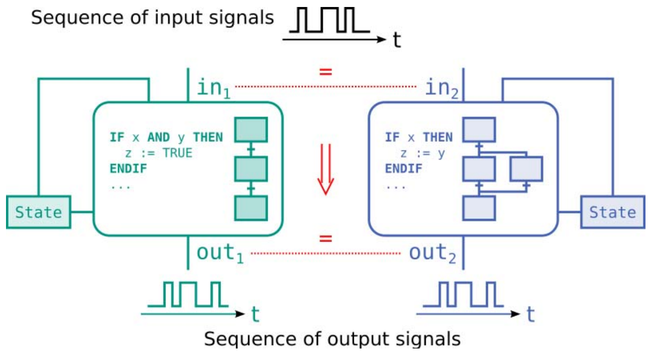

Fig. 1. Behavioral equivalence with different implementations

The main advantage of regression verification in comparison to usual functional verification is that no behavioral specification is needed. As opposed to (regression) testing, regression verification proofs cover all possible input values and not just selected test cases. Another advantage is that no hardware test

978-1-4673-7929-8/15/$31.00 ©2015 IEEE

bed or executable hardware model is needed for regression verification.

In theory, there are no limitations for the approach, as with fixed finite data structures as used in PLC programs, equivalence can always be proved by model checking. However, scalability is an issue in practice. If equivalence between the two releases is only conditional or partial, the conditions that capture that partiality must be specified by the user.

The structure of the paper is as follows: An overview of related work is given in Section II. Section III presents the approach concepts, including the definition of equivalence of PLC programs, environment models to increase precision and the method and toolchain. In Section IV, a case study is presented to evaluate the approach. A conclusion and an outlook are given in the last section.

II. Related Work

The aim of software quality assurance methods – e.g., testing or formal verification – is to support the developer to identify software faults and to fix them efficiently. Software engineering techniques available in automation engineering today are not sufficient to thoroughly assure the required level of availability, functional safety and reliability in variant-rich aPS. Recent approaches based on regression testing [2], testing of variant-rich aPS using 150% UML state charts [3] or virtual commissioning techniques allow validating the system behavior in a restricted time frame, but possess weaknesses for detecting rare events.

For rare events, formal verification techniques are suitable, as the state space of the system is analyzed for compliance exhaustively. Several works focus on verifying PLC code through model checking [4] [5] [6]. All of the approaches fail or struggle with the state explosion problem when applied to industrial PLC software. In addition, most approaches rely on precise function specifications or environment models, which oftentimes do not exist in the domain of automation engineering.

Regression verification was first introduced by [7] who used the bounded software model checker CBMC to prove the equivalence of C programs. Many approaches have been developed for the verification of program equivalence since. In [8], sufficiently strong coupling predicates are automatically inferred to prove the equivalence of C programs. The new approach presented in this paper transfers and extends these ideas into the world of PLC software. An active research area related to regression verification is equivalence checking of hardware circuits, [9] presents a symbolic model checking approach for hardware similar to ours for software.

III. Concepts

To be able to reason about equivalence for control programs, a formal definition of equivalence of PLC programs is given, followed by the concept of environment models. These are used to avoid or reduce the number of false alarms by excluding impossible input signals. The developed method and toolchain for the regression verification approach based on these notions is subsequently presented.

A. Formalizing Equivalence of PLC Programs

There are various possibilities for defining system boundaries when modeling an aPS. Entire systems or individual components can be modeled. Even when focusing on the PLC, models of peripheral hardware components could still be included. However, the presented method concentrates on the software that runs on the controller and disregards all effects outside the software. Sect. III.B discusses measures to include the environment in the models.

PLCs are reactive systems with a cyclic data processing behavior, repeating the same control procedure indefinitely. In our approach we consider PLC programs with a constant cycle time which perform the following steps repeatedly: (1) read input values, (2) execute task(s), (3) write output values, (4) wait. PLCs can be modeled as functions that compute output values from input signals read at the beginning of a cycle. However, the same input may result in different output signals at different points in time since PLCs possess an internal state (memory) that may change over time. To examine the equivalence of PLC software, it does hence not suffice to look at the cycle's input/output function, but a notion of *equivalence of signal traces* must be used: If the two software variants are presented with the same sequence of input sensor readings, they must produce the same sequence of actuator outputs as illustrated in Fig. 1.

The *trace semantics* of a PLC program is a mathematical function that maps every trace (i.e., infinite sequence) of input signals to the corresponding trace of output signals computed by the PLC when it is presented with that input sequence. Two PLC programs are called *perfectly equivalent* if their trace semantics are equal (they compute the same function). They are called *conditionally equivalent* if their trace semantics produce the same results for a subset of possible input traces (the formal description of the subset is then called the condition).

B. Environment Models to Increase Precision

To reduce false alarms during the comparison of two revisions of a PLC program, it is sensible to include available knowledge about possible input traces into the verification. It is irrelevant if the programs behave differently on input traces which cannot occur in practice.

In simple cases properties of the physical system can be stated in form of conditions on the PLC inputs. In more complicated cases, these are difficult or error-prone to express. Then it is better to use a model of the context of the aPS which uses output of PLC program as input and delivers the possible sensor signals to the PLC program. This model represents the environment of the software, i.e. the uncontrolled (parts of the) plant and possible other factors of investigation, such as bus behavior or user interaction. This restricts the search space, increases precision of regression verification and avoids false alarms. Such models can be expressed in form of state machines.

While increasing model accuracy reduces the search space and allows more correct revisions to be verified, it at the same time enlarges the system description for the model checker. It is, hence, a good idea to abstract away from behavior for the regression verification wherever this is possible.

C. Method and Tool Chain

Regression verification for PLC software is achieved in the presented approach by construction of a verification condition from two PLC program variants and environment models. The workflow of our method, as shown in Fig. 2, covers several transformation steps.

In the final step, the resulting verification condition consisting of a transition system and a property is presented to a model checker that can come back with three possible results: It may report that the verification property holds for the transition system in which case the two PLC programs are trace equivalent. It may report a counterexample with a concrete (finite) input trace that leads to the equivalence violation. There are no "false positives": Every reported violation uncovers a case of unequal behavior. In case the condition is not valid, it may be that the environment is not modeled precisely enough, and that the failure is a false alarm in the sense that it cannot occur in practice with the real hardware. The variables range over finite datatypes and the model checking problem is, in theory, decidable. Depending on the size and complexity of the verification condition, it is still possible that the model checker runs out of resources (time or memory) and does not come back with an answer, which is the third possible result.

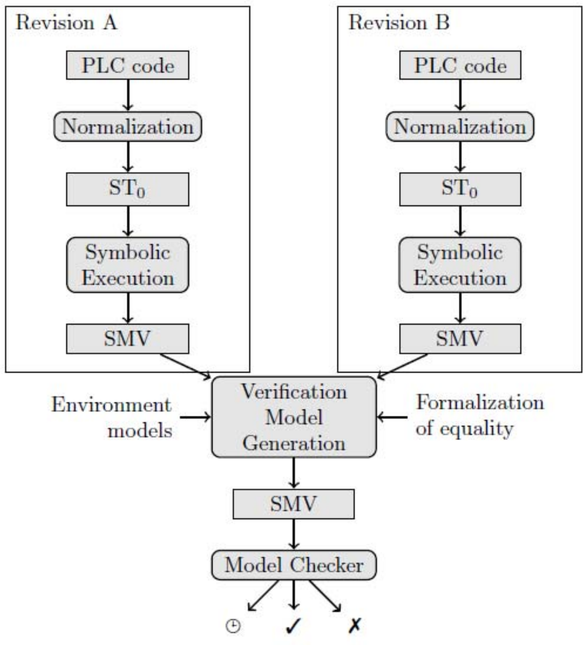

Fig. 2. Overview over the regression verification method

The IEC 61131-3 standard defines two textual and three graphical PLC programming languages. According to the ARC industry advisory group [10], the use of PLC systems compliant with IEC 61131-3 currently is and will remain the state of industrial practice for the next five to ten years. The approach considers input PLC programs written in the textual language Structured Text (ST) or the graphical language Sequential Function Chart (SFC).

For a uniform treatment of programs regardless of the particular language, an intermediate language into which all incoming programs are translated, is defined. This language ST_0 is essentially the loop- and call-free fragment of ST reduced to fewer, more basic datatypes. The only statements in ST_0 are assignments and if-then-else conditionals. Despite their notational differences, programs in all 61131-3 programming languages can be represented in ST_0 (provided they do not have unbounded loops, which are to be avoided in real time systems) [11]. During normalization to ST_0, loops are fully unwound and function block invocations are inlined. The normalized code in ST_0 is symbolically executed to derive a state transition system as model checker input.

Since ST_0 programs that result from translating SFC code involve many consecutive and nested if-statements to encode the original state machine, the number of paths through the program is huge and explicitly enumerating them is infeasible. For example, the last scenario (Ev14) of the case study (Sect. IV) yields some 13 billion paths, such that the resulting proof obligation would not fit into the available memory. To avoid this problem, a smaller program representation by not explicitly enumerating all paths but following the concept of Phi-nodes [12] to merge the effects of the branches of an if-statement is produced. While this procedure cannot guarantee that the result is not exponentially larger than the input, experiences showed that the results are acceptable in practice.

The proof obligation handed to the symbolic model checker consists of a state transition system and a property that is to be proved an invariant for it. The state transition system is a composition of the two systems that result from translating the two PLC program revisions A and B and the models for the environment as introduced in Sect. II.B. All variables of the input spaces of A and B make up the input variables of the combined model. If the sensor readings are constrained by an environment model, the input signals of that model are input signals of the entire state transition system while input signals of the PLC programs corresponding to sensor readings are taken from the outputs of the environment model. This has two effects: (1) The input space size is reduced and (2) the modeling is more precise.

Modern model checkers allow the application of state abstraction methods to find proofs for safety properties more efficiently. Regression verification using symbolic model checkers with such abstractions is particularly promising, since the two software variants are closely related if they are based on the same variant, adapted to the same application scenario. In such cases, it is likely that the variants of the program have a similar – yet not necessary equal – encoding of their state spaces. Using an invariant over the state spaces of two program variants allows to reason about safety properties. Such a predicate, building a bridge between the state spaces, is called a *coupling predicate*. The more similar the state space encodings of two program variants are, the closer the coupling predicate is to equality on the state spaces.

If an existing software base is adapted simultaneously and independently to similar changed requirements, it is to be expected that the changes affect only a relatively small number of the state variables whereas much of the state remains unchanged (and leaves both revisions' behavior equivalent in many cases). An inductive invariant implying equivalence then comprises equality between the unmodified state variables, and a more general coupling invariant must be generated only for the affected variables.

The regression verification method using invariants is complete but the user of the verification tool would have to find and formalize all coupling invariants which can be large and unintuitive. Instead, the capabilities of state-of-the-art symbolic model checkers to automatically infer inductive invariants are used. The presented case study shows that even with large state spaces, this state abstraction mechanism allows to prove equivalence of non-trivial programs. The model checker *nuXMV* is capable of coming up with the required coupling predicates. If this invariant generation mechanism is switched off, the tool relies on more traditional symbolic model checking techniques, in which case even the simpler problems in the case study could not be solved.

IV. Case Study

We have evaluated our approach by applying it to the benchmark evolution scenarios of the Pick-and-Place Unit (PPU), which is illustrated in Fig. 3. The PPU is an open case study for the machine manufacturing domain [13]. The PPU has 22 digital input, 13 digital output, and 3 analogue output signals and defines a number of simple discrete event automation tasks. Despite being a bench-scale, academic demonstration case, the PPU is complex enough to demonstrate selected challenges that arise during engineering of aPSs. The toolchain was successfully applied to all scenarios from [13] that had a suitable change of software implementation. As the complexity of the case study is noticeably below industrial use cases, e.g. regarding the number of inputs and outputs and the use of analogue input and output values, scalability of the presented approach is focus of future work.

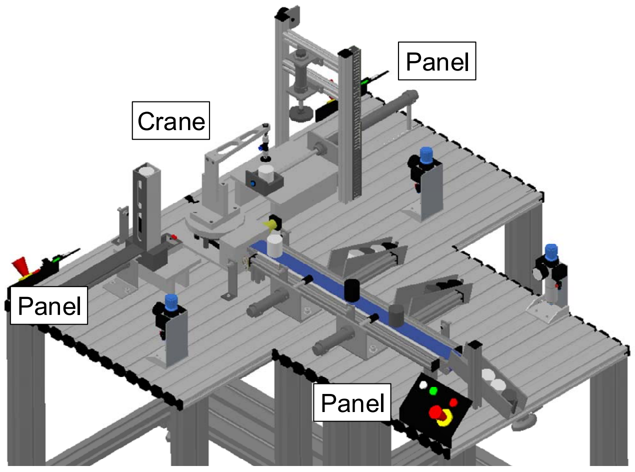

Fig. 3. Schematic of the hardware setup of the PPU case study [13]

Proving *conditional equivalence* was evaluated using scenario 3 (Table I. Sc3), in which the functionality of starting the machine was implemented in a different way. Trace equivalence could be proven using an LTL formula omitting other changes to the system (changes of the input panel hardware, see Fig. 3, “Panel”).

An *environmental model* was used to investigate scenario 6, in which handling of certain work pieces by the crane (Fig. 3, “Crane”) was implemented differently. Equivalence was assumed and analyzed for the individual work piece types (Table I. Sc6a, Sc6b) and both types (Sc6ab). Using an environmental model of the crane transporting the work pieces, the ad-hoc assumption could be more intuitively formulated (Sc6+EM).

Using our method and toolchain, automatic regression verification was successful for all selected scenarios from the PPU case study. Table 1 shows statistics for relevant experiments with the PPU. The evolution scenarios were verified using *nuXMV* version 1.0.1 on an Intel Dual-Core with 2.7 GHz and 4 GB RAM running OpenSUSE 12.2.

TABLE I. Results of Experiments

Scenario	Sensor inputs [bits]	State space [bits]	Min. time required	Max. time required
Sc3	19	246	9s	17s
Sc6a	19	284	15.1m	155.4h
Sc6b	19	284	8.9m	9.1h
Sc6ab	19	284	18.1m	13h
Sc6+EM	11	299	2m	21m

The verification times for the same problem on the same machine may vary considerably in multiple runs due to random choices in the symbolic model checker which have a great impact on the verification time.

The regression verification method can not only be used for verifying equivalence of PLC programs, but unintentional differences between programs can also be found using the presented approach. The evaluation revealed that in scenario 3, new intermediate code blocks are added into SFCs that cause a difference by delaying the system answer one cycle for each work piece. Since the cycle time is very short in the PPU (4 ms), the discrepancy between the programs was not found by testing, but points to an inferior implementation.

V. Conclusion and Future Work

We have presented a method and toolchain for the automatic regression verification of PLC software by means of a symbolic model checker. In this process, a software variant serves as specification for the other one. Conditions can be specified under which systems must behave equivalently and models of environment can be added to make the process more precise. Evaluation proved the presented method to be applicable to non-trivial PLC software. Automatic regression verification was successful for all selected scenarios from the PPU case study.

Currently, the developed toolchain supports notions that compare PLC behavior cycle by cycle. Future work will allow for conditions and relations to relate variables of different cycles. Another interesting path of investigation is the use of abstractions to factor out parts of PLCs that have not been touched by evolution and need not be proved equivalent. In addition, the scalability of the approach in real industrial use is to be investigated.

REFERENCES

[1] B. Vogel-Heuser, C. Diedrich, A. Fay, S. Jeschke, S. Kowalewski, M. Wollschläger and P. Göhner. "Challenges for Software Engineering in Automation," Journal of Software Engineering and Applications, vol. 7, no. 5, pp. 1-12, 2014.

[2] S. Ulewicz, D. Schütz and B. Vogel-Heuser. "Software Changes in Factory Automation - Towards Automatic Change Based Regression Testing," Conference of the IEEE Industrial Electronics Society, pp. 1-7, 2014

[3] M. Lochau, J. Bürdek, S. Lity, M. Hagner, C. Legat, U. Goltz and A. Schürr. "Applying model-based software product line testing approaches to the automation engineering domain," at – Automatisierungstechnik, vol. 62, no.11, pp. 771–780, 2014.

[4] R. Huuck. "Semantics and analysis of instruction list programs," Electr. Notes Theor. Comput. Sci. 115, pp. 3–18, 2005.

[5] N. Bauer, S. Engell, R. Huuck, S. Lohmann, B. Lukoschus, M. Remelhe and O. Stursberg. "Verification of PLC programs given as sequential function charts," Integration of Software Specification Techniques for Applications in Engineering, LNCS 3147, pp. 517–540, 2004.

[6] A. Wardana, J. Folmer and B. Vogel-Heuser. "Automatic Program Verification of Continuous Function Chart based on Model Checking," Conference of the IEEE Industrial Electronics Society (IECON 2009), Jan. 2009, pp. 2422-2427.

[7] B. Godlin and O. Strichman. "Regression verification: proving the equivalence of similar programs," Software Testing, Verification and Reliability, vol. 23, no. 3, pp. 241–258, 2013.

[8] D. Felsing, S. Grebing, V. Klebanov, P. Rümmer and M. Ulbrich. "Automating regression verification," ACM/IEEE International Conference on Automated Software Engineering (ASE), pp. 349–360, 2014.

[9] C. A. J. van Eijk. "Sequential equivalence checking based on structural similarities," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 19, no. 7, pp. 814–819, 2000.

[10] ARC Advisory Group. "PLC & PLC-based PAC worldwide outlook: Five year market analysis and technology forecast through 2016," 2011.

[11] A. Weigl. "Regression verification of programmable logic controller software," Master's Thesis, Karlsruhe Institute of Technology, 2015.

[12] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman and F. K. Zadeck. "An effcient method of computing static single assignment form," ACM Symposium on Principles of Programming Languages (POPL 89), ACM, 1989.

[13] B. Vogel-Heuser, C. Legat, J. Folmer and S. Feldmann. "Researching Evolution in Industrial Plant Automation: Scenarios and Documentation of the Pick and Place Unit," mediaTUM, Munich, Germany, 2014.