A Verification-Supported Evolution Approach to Assist Software Application Engineers in Industrial Factory Automation*

Sebastian Ulewicz, Mattias Ulbrich, Alexander Weigl, Michael Kirsten, Franziska Wiebe, Bernhard Beckert, and Birgit Vogel-Heuser, *Senior Member, IEEE*

Abstract— **Automated production systems (aPS) are complex systems with high reliability standards which can – besides through traditional testing – be ensured by verification using formal methods. In this paper we present a development process for aPS software supported by efficient formal techniques with easy-to-use specification formalisms to increase applicability in the aPS engineering domain. Our approach is tailored to the development of evolving aPS as existing behavior of earlier revisions is reused as specification for the verification. The approach covers three verification phases: regression verification, verification of critical interlock invariants and delta specification and verification. The approach is designed to be comprehensible by aPS software engineers: Two practically applicable specification means are presented.**

Formal methods have not yet been widely adapted in industrial aPS development since they lack (a) scalability, and (b) concise and comprehensible specification means. This paper shows concepts how to tackle both issues by referring to existing behavior during evolution verification to advance towards the goal of applicability in the aPS engineering domain.

A laboratory case study demonstrates the feasibility and performance of the approach and shows promising results.

I. Introduction

Automated production systems (aPS) [1], such as industrial manufacturing plants, are commonly automated with Programmable Logic Controllers (PLCs), which are computing devices specially tailored to control automated production systems in dependable or safety-critical real time environments. As malfunctions may cause severe damage to the system itself, the processed payload and interacting persons, high quality requirements are imposed on an aPS and on its software in particular. These requirements are commonly ensured by software testing. While testing is a common fault detection method, it often fails to detect rare problematic events. A single test case describes only one specific behavioral pattern, therefore testing can usually not cover the complete system behavior and cannot prove correctness. Complementary to testing, formal methods can be used to mathematically and exhaustively prove the compliance of a system to a specification, including all possible corner cases and rare events.

*Research supported by the DFG (German Research Foundation) in Priority Programme SPP1593: Design for Future – Managed Software Evolution (VO 937/28-2, BE 2334/7-2 and UL 433/1-2).

S. Ulewicz, F. Wiebe and B. Vogel-Heuser are with the Technical University of Munich, Institute of Automation and Information Systems, 85748 Garching near Munich, Germany (phone: +49-89-289-16400; e-mail: {sebastian.ulewicz; franziska.wiebe; vogel-heuser}@tum.de).

M. Ulbrich, A. Weigl, M. Kirsten and B. Beckert are with the Karlsruhe Institute of Technology, Institute of Theoretical Informatics (e-mail: {ulbrich; weigl; kirsten; beckert}@kit.edu).

In factory automation engineering, quality assurance by formal methods is not common yet. There are two main reasons for that: (a) today's formal verification tools fail at verifying industrial sized problems (state space explosion) and (b) the specifications required for formal verification are extensive and not intuitive for the engineer. Applying a formal method and interpreting its results require a deep understanding of the underlying concepts and are unduly labor-intensive.

Evolution verification addresses both issues by reusing old software revisions as specification for the desired behavior of the new revision. In its purest form, evolution verification would require the new system to behave identically to an old revision. Evidently, this requirement is too strict for most use cases. We therefore enrich this form of verification (called regression verification [2]) by means to formulate which parts of the behavior remain the same, which parts have changed and how they have changed. Thus, evolution verification aims at formally proving that software remains correct throughout its evolution, changes have the desired effect, and no new bugs are introduced. The benefit is twofold: It avoids the need to write full functional specifications for the system, which is the main bottleneck for routine practical use of formal verification. It is plausible to assume that the specification of the change in behavior is far more feasible than the full specification – both regarding complexity and size. Moreover, the subsequent verification effort mainly depends on the difference between the programs and not on their overall size and complexity.

The main contribution of this paper is an approach for a verification-supported development process for the evolution of aPS software, integrating three different verification phases that address different aspects of software correctness in the face of evolution. The goal is to enhance formal verification towards industrial applicability by reducing the overall verification complexity, minimizing required specification effort and supporting the application engineer in their task: Regression verification for supposedly unchanged behavior is complemented with verification of critical invariants (rules never to be broken) and delta verification of specific changed parts of the behavior. Furthermore, two appropriate notations are presented which allow an intuitive specification of intended new behavior. Formal verification techniques are outlined which allow to verify the arising verification obligations efficiently. In a case study, the approach is evaluated for feasibility and performance using a laboratory aPS example.

The paper is structured as follows: An overview over related work in the adjacent fields to the developed approach is given in Section II. The concept of our approach is

presented in Section III, which is applied and discussed in a case study in Section IV. In the last section, a conclusion and an outlook are given.

II. Related Work in *(Model-based) Testing*, *Virtual Commissioning* and *Formal Verification*

The aim of software quality assurance methods is to support the developer in identifying software faults and to fix them efficiently. *(Model-based) Testing*, *virtual commissioning,* and *formal verification* are the prevalent research directions in the field of factory automation.

Many advances are made in the area of model-based testing [3]. Here, tests can be generated from formal models based on UML specifications [4]–[6], which can be extended by using virtual commissioning techniques [7], [8] to include behavior regarding the controlled hardware and technical process in the factory automation system. Recent approaches based on regression testing [9] aim at increasing the efficiency of these approaches during evolution by supporting the selection of appropriate test cases. Testing allows to validate the system behavior in a restricted time frame, but possesses weaknesses regarding the detection of rare events, as test cases relate to very specific scenarios of a specification rather than proving correct behavior. However, these rare events are potentially very critical. To counter this, some approaches generate input sequences for test cases directly from the code [10], [11] relying on coverage criteria. Yet, these approaches lack conformance criteria (the *test oracle*) and are constrained to simple programs, such as single program organization units (POUs).

Complementary to testing and virtual commissioning approaches, formal verification techniques can be applied to close this gap. Instead of analyzing a specific application scenario, formal techniques aim at analyzing state space models exhaustively, i.e., covering all reachable states [12], [13]. Automatic verification can also be beneficial compared to simulation and testing, because it can – besides exploring all reachable states – be applied earlier in the design phase [14]. Several works focus on verifying PLC code using model checkers [15]–[18]. Efficient translation methods from Sequential Function Charts (or similar) to precisely formalized (timed) automata are given by [16], [19], [20]. Yet, the state explosion problem is a struggle or even a reason to fail for these approaches when applied to industrial PLC software due to its complexity. In addition, most approaches rely on precise functional specifications or environment models, which oftentimes do not exist in the domain of automation engineering.

Regression verification focuses on analyzing whether regressions, e.g., software bugs or undesired behavior, are introduced by changes to a system. It was first introduced by [21], where the equivalence of C programs using a software model checker is proved. Many approaches have been developed for the verification of program equivalence in the area of computer science since. In [22], automatic inference of coupling predicates is used to prove the equivalence of C programs. In recent work, these ideas are extended for use in PLC software [23]. In many cases, the sole relation between old and new behavior is not easily described to cover evolution steps sufficiently, as verification of changed behavior and its influence is not regarded. Extending prior work, we widen our scope and embed regression verification into a holistic approach, covering the full evolution process.

Formal specifications defining supposedly correct behavior can also contain faults, just as the corresponding software. Thus, a finding by [24] is of high importance. Here, an experimental study of PLC programming analyzed different kinds of faults when writing code using different types of UML/SysML languages or IEC 61131-3 Function Block Diagrams (FBD). In a lab-based study with students, technicians and trainees, a detailed observation of error causes was conducted. Results indicate that an insufficient understanding of the notation's syntax was the cause for most of the faults in the code. Therefore, using unfamiliar and complex notations within the industry does not seem advisable, neither for specifying behavior nor for programming. However, most related verification approaches rely on manually formalized mathematical artefacts or unfamiliar notations, e.g., petri nets [25]. For generating test cases, notations such as timing diagrams [26] were successfully used and evaluated in industrial scenarios with experts in this domain. In [27], control code for avoiding critical conditions (interlock code) is generated from CAEX documents. In addition, a cause & effect matrix is generated for documentation and further analyses of criticality. To gain acceptance in industry, notations must be easily learnable and close to notations already in use.

III. Concept – Three-phase Formal Verification of Software Evolution

As an aPS evolves, its software must often be adapted to changed requirements or processes [28]. It is not unlikely that software modifications and extensions have an unintended impact on the system behavior, i.e., they deviate from the given requirements specification. A modification may break existing good behavior or introduce undesired new behavior into the system. Figure 1 outlines the verification concept schematically. The original behavior of the initial software revision is denoted "old". The software modification induces a change in behavior (marked Δ) which overlays the original behavior, modifying parts of it and leaving other parts uninfluenced.

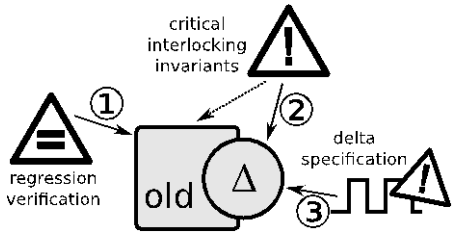


Figure 1. The three verification phases of our approach: (1) Verify old behavior using regression verification, (2) verify critical invariants by reusing existing specifications and (3) verify new behavior (delta).

Our approach proposes three analysis phases building on top of each other to verify different aspects of the correctness of aPS software evolution: First regression verification is employed to show that defined parts of the old behavior are inherited by the new software. Then, general critical interlock invariants are checked for the newly introduced behavior.

Finally, the new behavior is verified against a formal delta specification.

The three phases target at different aspects of the observable software behavior, evoked by a sequence of sensor values and characterized by a sequence of actuator readings as output by the software over time (i.e., PLC scan cycles). First, regression verification [2] is performed, which analyses whether the new revision behaves equivalently to the old version for all cases where no change in behavior is intended. Secondly, the cases in which the behavior is supposed to change are checked against existing critical interlocking properties (avoiding situations possibly leading to collisions or damages). Thirdly, the behavior which has not been present in the old revision – the same that is checked for critical properties in the second phase – is verified against lightweight delta specifications (written for that occasion), which outline the behavior of the newly introduced code.

The regression verification in the first step covers unchanged behavior, which in this case is regarded as intended behavior; interlock and approved behavior guarantees are thus directly inherited from the old revision. Formal verification in the latter two phases can thus be limited to the new behavior not covered by regression verification, i.e., those parts of the software which are influenced by the change. This reduces the verification complexity drastically and allows efficient verification.

In all three verification phases violations of the implementation against the desired behavior are presented to the user by counter examples in the form of input traces in textual form.

A. Phase 1: Detecting Undesired Regression in Previously Existing Intended Behavior using Regression Verification

Regression verification in its purest form verifies perfect behavioral equivalence between two PLC software revisions. That means, for all sequences of sensor readings the produced sequence of actuator outputs is identical. Yet, in practice, regression verification rarely means asserting perfect equivalence between software revisions. Usually, the intention is that *specific parts* of the behavior are related in some way, e.g., behave identically. For formal verification, the cases in which equivalence is expected need to be described precisely. This description only specifies the input sequences to be considered, while the intended behavior is defined by the old revision, resulting in a very low specification effort. For that reason, it seems reasonable to conduct regression verification for a part of the behavior (the old behavior) as the first step in the verification of an evolution event to ensure that the newly introduced behavior does not affect this intended behavior.

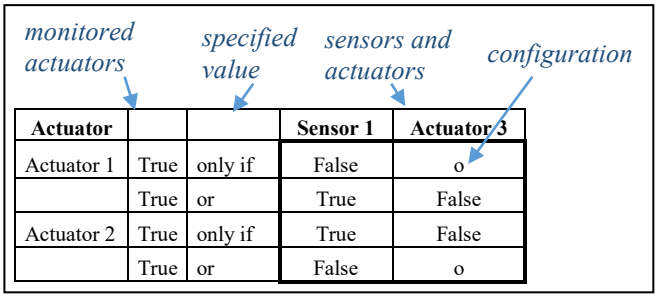
The only specification that is required for so called conditional regression verification is a characterization of considered behavior as a condition on the sequence of sensor values. One possibility for such specification is structured natural text constraining the possible input sequences. Simple phrases like “Sensor 1 is always true”, “Sensor 2 is true while Sensor 3 is false” or more complex descriptions like “Whenever Sensor 4 is true, Sensor 4 remains true until Sensor 5 is true” can be interpreted formally. The meaning of such phrases is a formula in linear temporal logic (LTL). For instance, “Sensor1 is always true” becomes **G** Sensor1 when interpreted as temporal logic formula.

The verification is conducted using a symbolic model checker which is fed with translations of the two software revisions. The proof obligation to check is that the condition on the input values implies equivalence on the actuator output values in each PLC scan cycle. Both software revisions are modelled as parallel input for the model checker. To cover all possible behavior, every input signal is modelled as a non-deterministic choice.

B. Phase 2: Ensuring Global Interlock Invariants

Most aPSs possess fundamental interlock rules which must be maintained at all costs during evolution. Examples for such properties are critical sensor / actuator combinations that must not emerge since they may cause damage to the hardware, e.g., a crane carrying a part must not be lowered during horizontal movement to avoid collisions. These conditions need to hold for all cases of the behavior of a system, whether it is inherited from the old revision or newly introduced. For the parts of the aPS behavior covered by regression verification, the interlock conditions need not be re-verified – based on the assumption that they have been ensured at an earlier evolution stage. Thus, regression verification and invariant verification complement each other when performed in this sequence, which is why this order was chosen. In addition, the interlock conditions considered in this step are to be obeyed at all points in time and for any software revision, and can therefore be specified as global invariants that are not bound to a specific revision and can be reused. As these global invariant conditions are readily available after changing a system (from the last evolution step), this verification step requires low to no effort regarding the creation of a specification.

Even though the interlock conditions themselves can be quite simple, the amount of cross connections between them can become confusing over time for the involved user(s). From our experience, many companies use simple tables to specify interlock conditions, as they give a clearly arranged overview and are easily comprehensible. As similar specifications are already commonly used in industry, we propose using a table to specify interlock invariants for this verification phase.



Actuator			Sensor 1	Actuator 3
Actuator 1	True	only if	False	o
	True	or	True	False
Actuator 2	True	only if	True	False
	True	or	False	o

Figure 2. The invariant table can be used to specify invariant rules relating to actuator values

As shown in Figure 2, an invariant table defines invariants for actuator values, e.g., “Actuator 1 may only be true if Sensor 1 is false or if Sensor 1 is true and Actuator 3 is false at the same time”. For this, an actuator variable is

brought into relation with necessary conditions. The invariant table can be extended and adapted in every evolution step.

For verification, invariant tables are interpreted as the conjunction of the global invariants described by the entries in the table. The first two lines in the table in Figure 2, e.g., are translated into the LTL property

$$\mathbf{G}\ (\text{Actuator1} \rightarrow \neg\text{Sensor1} \vee (\text{Sensor1} \wedge \neg\text{Actuator3}))\ .$$

During model checking, the translation of the new version of the PLC software is checked against these properties. To reduce the search space, the negation of the condition may be used in regression verification as additional premise since those cases have already been considered. Therefore, we employ the IC3 approach [29], which computes an over-approximation of the set of reachable states. It iteratively adds formulas supporting the interlock invariant. This computation is repeated as an iterative refinement of the approximation until there is no violation of the invariant.

C. Phase 3: Delta Verification Targeted at Changed Behavior

In the third and final verification phase – after verifying unchanged behavior (Section A) and global invariants (Section B) –, the newly introduced behavior itself is specified and verified. Here, the changed behavior (the delta) is analyzed in detail. Hence, a general critical interlock specification as in the last section is not sufficient to describe the newly introduced behavior; the specification is likely to become more complex than in Section B. However, formal delta specification and verification can be restricted to the parts of the behavior which are new.

For the visualization and specification of complex signal sequences, timing diagrams have been established in factory automation industry [26]. Existing know-how in industry can be leveraged by using this specification method in this phase. The structure of a Boolean timing diagram consists of the considered actuators on the left side, and their correlated signal sequences on the right side, which are both embedded in a coordinate system with a timing axis and a Boolean value axis (see Figure 3). For verification, the timing sequence is interpreted as the sequence progression of changing Boolean values (i.e., the sequence of falling and rising edges). Using a model checker, the sequence of the actuator values is verified for following the diagram pattern: Edges which are vertically aligned in the diagram (connected by a dashed line) are to happen synchronously. Whether the duration of the signals in the diagram and the actual duration in the behavior are the same, is not checked in our approach. Only the chronological order of the signal values at the end of scan cycles is compared.

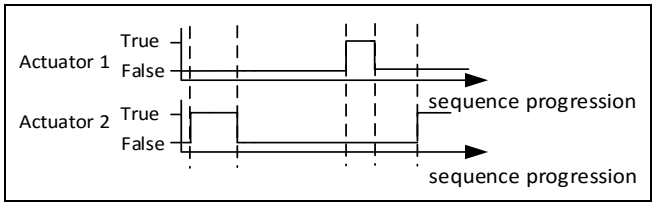


Figure 3. Schematic of a Timing Diagram which is used as a specification for signal sequences in the presented approach

Usually, behavioral changes affect only specific segments and specific variables of the input/output traces, e.g., steps in a process, which have been newly introduced or optimized. Outside the scope of this specification, the behavior is intended to be the same as in the previous revision – and thus its correctness is covered by regression verification.

The two trigger points at which the specified behavior starts and ends respectively need to be specified in order to define the limits of the segments. Therefore, delta verification goes hand in glove with the regression verification phase. Together, the verification covers all complete input/output traces and the two synchronization points (start and end of the delta behavior) define which of the two verification techniques covers the part of the behavior.

Ultimately, regression and delta verification can be combined into one model checking task which requires no more specification than the trigger points and the timing diagram. Hereby, the old revision is wrapped into a new reactive system: This system uses output values from the old revision during regression verification. Within the segments defined by the trigger points, values are defined by the specification of new behavior, e.g., as a timing diagram.

D. Environment Models

Formal verification considers all possible sequences of input signals (*traces*), and thus never misses observable machine behavior. Yet, it may also look at traces that cannot possibly occur in reality, which may cause false alarms. In order to reduce the number of such alarms and to make the verification more precise, models of the environment of the software (i.e., the physics of the plant, the sensors, interaction with work pieces, etc.) can be added to the verification conditions. Environment models are temporal specifications constraining the observable signal values, generally assumed to be non-deterministic. An engineer can use, for instance, schematic timing diagrams (such as in Section C), invariant tables (e.g., the actuator specifications in Section B), or more general mechanisms to define finite automata that describe the possible input signals.

E. Feedback to the application engineer

Whenever the verification process succeeds, the application engineer gets a confirmation that the system under investigation is correct with respect to the specified property (there are no false negatives). The verification may fail either due to a wrong or insufficient specification or because of an actual mistake in the implementation. In either case, the model checker yields a counterexample trace contradicting the specification.

Currently, this counterexample is presented to the engineer in its raw textual form. In future research, more descriptive notations, e.g., time sequence diagrams [26], will be investigated to further enhance the applicability of this approach in industry. Through displaying traces directly in the specification instead of a textual form, the application engineer is expected to gain a better understanding of the reason why the model checking process failed and will be provided a starting point for further investigation of the cause, e.g. a fault in the software or a specification which is too strict. Subsequently, the engineer can adjust either the

specification or the software, and reiterate the verification process until a satisfactory outcome is reached.

IV. Case Study – Pick and Place Unit

To demonstrate the feasibility and assess the performance properties of the presented approach, the approach has been applied to an evolution step of a laboratory plant. The plant, its software implementation and the examined evolution scenario are adapted from [30]. Additional software bugs were introduced a posteriori into the scenario to demonstrate the bug-finding capacities of the approach. The scenarios, specifications and verification conditions are described and discussed in the following.

A. Base Scenario

The laboratory sized automated production system "PPU" (Pick and Place Unit) consists of four components (cf. Figure 4): A stack for storing work pieces (WP), a stamp for labeling white WPs, a conveyor for sorting WPs and a crane for pick and place operations between the modules. The function of the PPU in the base scenario is to pick and place black and white cylindrical WPs from a stack to a conveyor for further processing. All white WPs are labeled by a stamp unit before being placed on the conveyor; all black WPs are to be directly placed on the conveyor. The WPs placed on the conveyor are sorted onto different slides using pneumatic pushers. Using an optical sensor, the color of the WPs pushed out by the stack is determined (white or black). The crane can move up and down to pick and place WPs by means of a pneumatic suction unit and turn clockwise and counterclockwise to move between the modules. Two binary sensors detect the vertical position, one binary sensor detects a suctioned WP and three binary sensor detect whether the crane is at the respective station (stamp, stack and conveyor). The conveyor module in the software detects a placed WP using one binary sensor and signals the conveyor belt to move.

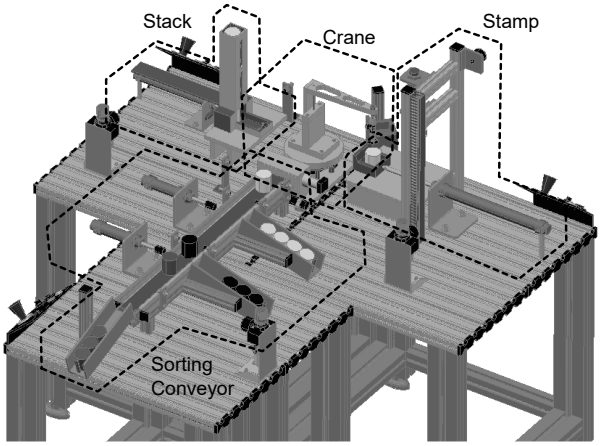


Figure 4. Laboratory aPS "PPU" used in the Case Study (see [30])

In the original revision of the plant, only one WP is processed at a time. To increase the throughput of the plant, the plant is subjected to an evolution step in which the PLC software is optimized while leaving the hardware unchanged. The new, optimized revision changed its behavior at one point: As soon as the crane places a white WP into the stamp module, the crane is to transport black WPs (if available) from the stack to the conveyor instead of waiting for the stamping process to finish. After placing the black WP on the conveyor, the stamped white WP is transported to the conveyor. This new feature only directly affects the software of the crane. A verification of the crane module alone is not possible because of dependencies between modules [30]. The new revision consists of four POUs, containing 117 Boolean and 11 integer variables, resulting in a state space of approximately 300 bits in the model checker.

To verify that the new feature is implemented correctly while leaving the previously working behavior unaffected, the three-phase approach presented in Section III is used. To demonstrate the possibilities of the three verification steps, we have implanted exemplary errors into the code which were uncovered in the respective phase. Additional material for the case study is assembled on a companion website [31].

B. Phase 1: Verifying supposedly unchanged behavior

Conditional regression verification is used in this first phase to prove that specified parts of the behavior are preserved despite the optimization of the software. The specification of the condition under which equivalence is expected, is provided in structured natural language. In this case, if a sequence of exclusively white WPs or exclusively black WPs is placed into the machine, the newly introduced feature is not used and the behavior should therefore be unaffected. Changes to this behavior are not intended and are to be uncovered. In structured language, the specification confining the regression verification to this behavior reads as follows:

"If OpticalSensor is always true or OpticalSensor is always false, then both revisions behave identically."

The sensor "OpticalSensor" detects the brightness of the WPs at the stack position. If it is true, the WP is white, otherwise it is black. The value of "OpticalSensor" is fixed to be constant (either true or false) to encode the condition of regression verification, all remaining input signals are modelled as non-deterministic choices. The implanted error was found in the implementation by a divergence in the signals for the crane movement after 22 PLC scan cycles, beginning with a cold start of the hardware. This difference occurs due to an incorrect transition guard in the new SFC revision. For this and the following proofs, we use NuXmv [32] as the model checker together with the invariant generation engine IC3. The model checker returns a trace of the counter example in 26 seconds and needs 22 minutes to prove equivalence with a correct version of the new revision.[1]

C. Phase 2: Ensuring Invariants

To avoid dangerously critical situations, invariant tables can be used to specify restrictions on actuator values w.r.t. sensor value combinations. In this scenario, it is assumed that the developers defined constraints for the crane to avoid collisions at the conveyor: The crane should not move if it carries a WP and there is a WP at the conveyor, as this would result in a collision. Table I represents these invariants. The sensor "WP suctioned" indicates a vacuum buildup in the

[1] We measured all runtimes in this paper on an Intel Core i7 860 2.8 GHz. The given runtimes are the rounded values of the median of 5 samples.

suction cup of the handling device of the crane, which occurs if a work piece is picked up. Thus, "WP suctioned = True" indicates that the crane is transporting a work piece in which case it is only allowed to move (Crane_CW or Crane_CCW is true) if there is no work piece on the conveyor (Conveyor occupied is false). If the crane does not carry any payload (WP suctioned is false), it is allowed to turn, even if the conveyor is occupied as there is no danger of collision.

TABLE 1: INVARIANT TABLE FOR VERIFYING "CRANE NOT MOVING WHILE WP TRANSPORTATION AND ANOTHER WP ON CONVEYOR"

Actuator			WP suctioned	Conveyor occupied
Crane_CW	True	only if	False	o
	True	or	True	False
Crane_CCW	True	only if	True	False
	True	or	False	o

"CW": Clockwise, "CCW": Counterclockwise, "o": omitted

To reduce the possible sensor values in the verification process, a simple environment model was created and used. This environment model describes the possible "Conveyor occupied" sensor values dependent on previous actuator and signal occurrences. In this scenario, the environment model is a simple state machine with two states which ensures that the Conveyor can only become occupied if the crane has released a WP at the conveyor.

The proof performed by the model checker checks that the interlock properties are global invariants throughout all PLC scan cycles. It took 5 seconds to prove that the old revision satisfies the invariant, and 18 seconds to find a counter example for the implanted fault in the new revision. The counterexample trace can be used as a support to start investigating the cause of the regression within the source code.

D. Phase 3: Verifying changed behavior

To specifically verify the changed behavior in this case study, the signal sequence of the new feature was modeled by means of a timing diagram. The new feature is the changed movement of the crane while a white WP is being labeled at the stamp and a black WP is available at the stack and subsequently placed on the conveyor. Figure 5 depicts a part of the diagram, which was used to derive the sequence of signals relevant for describing the newly introduced behavior. The behavioral description of the new feature is partial in the sense that it does not make statements about the course of signals before and after the considered segment, nor does it state anything about sensors/actuators not explicitly mentioned in the model. These excluded parts of the behavior are required to behave as in the previous revision of the program, since a diversion would be unintended.

The actuator variable "CraneLower" is used for vertical movement of the crane ("true" lowers the crane, "false" invokes an upward motion), whereas the actuators "Crane_CW" and "Crane_CCW" invoke the clockwise and counterclockwise rotation of the crane. The depicted sequence in Figure 5 shows the crane being moved upwards ("CraneLower" is set to false). The crane is subsequently turned clockwise (Figure 5, Position 1) until it reaches the desired position (Figure 5, Position 2), after which it is lowered (Figure 5, Position 3) to pick up a WP and moves back up (Figure 5, Position 4). The specified segment of the sequence continues, describing the new feature in detail.

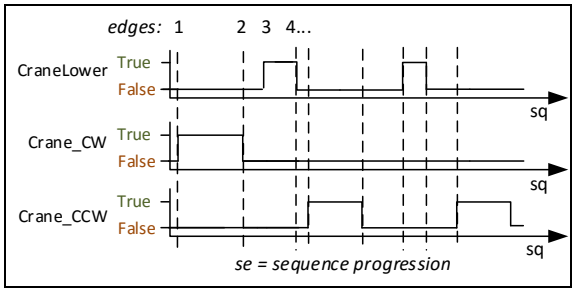


Figure 5. Timing-diagram of new behavior in the evolution process

Besides the timing diagram, the specification includes local conditions (trigger points) which determine the start and the end of the behavioral segment described in the timing diagram. Through this, every behavior was split into unchanged segments (behavior before and after the specified sequence which are subject to regression verification) and segments corresponding to the new features (which are subject to delta verification w.r.t. the timing diagram). Thus, the entire behavior is covered by the combination of the two verification techniques.

Our verification toolchain required 20 seconds to prove that all segments limited by the trigger points are equal to the behavior specified in the timing diagram. The regression verification for the unchanged behavior segments took 29 minutes.

E. Discussion of Results

The presented approach propagates a tight integration of formal verification into the development process during software evolution. The threefold verification supports the engineer in focusing on the changed behavior. The first step is to differentiate what behavior is new and what is (supposed to be) unchanged. The regression verification supports the engineer and uncovers indirect or unintended changes in the software and misconceptions in the engineer's understanding of the technical process within the aPS early on. The second verification step serves as a sanity check which does not require new specifications but is performed quickly as available specifications are used and only the changed part of the system needs to be covered. The third part is closer to traditional specification and verification – but profits from the evolution scenario. Through the reference to the old revision, the specification becomes more comprehensible and concise, and allows the verification to run considerably more efficiently.

As preliminary results from this study, it was found that the required runtimes depend on multiple factors: the complexity of the trigger points within the delta specification, the similarity of the software revisions and the quality of the refinement process within the IC3 technique. In case of simple trigger points, very similar software revisions and a good formula choice for refinement in IC3, efficient runtimes can be expected.

To gain further knowledge about the scalability of the approach, additional studies are needed and are focus of future work.

V. Conclusion and Outlook

In this paper, a verification-supported evolution approach to assist software application engineers in industrial Factory Automation was presented. By combining regression verification techniques with invariant verification and focused verification of changed behavior, the complexity of the verification problem can be reduced and specification efforts are minimized for evolution scenarios. In addition, two intuitive and well-established specification notations are proposed for use in the verification process. Thus, the main problems currently hindering adoption of formal verification in factory automation are approached. In a case study applying the approach to a laboratory plant in an exemplary evolution scenario, the feasibility is shown and the performance was measured to allow a performance assessment.

In future research, further convergence with industrial requirements is aimed at by introducing a comprehensive visualization of counter examples which are produced by the model checker in case the verification process fails. Moreover, the approach is to be extended by further commonly used notations and a study within the industry to fully evaluate the applicability within the domain of factory automation. For this, an industrial case study will be conducted including several companies in the domain of food and pharmaceutical packaging. Investigations towards better scalability of our approach through applications within further case studies are another focus of future work.

References

[1] B. Vogel-Heuser, A. Fay, I. Schaefer, and M. Tichy, "Evolution of software in automated production systems: Challenges and research directions," *J. Syst. Softw.*, vol. 110, pp. 54–84, 2015.

[2] B. Beckert, M. Ulbrich, B. Vogel-Heuser, and A. Weigl, "Regression Verification for Programmable Logic Controller Software," in *Formal Methods and Software Engineering*, Springer International Publishing, 2015, pp. 234–251.

[3] S. Rösch, S. Ulewicz, J. Provost, and B. Vogel-Heuser, "Review of Model-Based Testing Approaches in Production Automation and Adjacent Domains—Current Challenges and Research Gaps," *J. Softw. Eng. Appl.*, vol. 08, no. 09, pp. 499–519, 2015.

[4] B. Kormann, D. Tikhonov, and B. Vogel-Heuser, "Automated PLC Software Testing using adapted UML Sequence Diagrams," in *IFAC Symposium of Information Control Problems in Manufacturing*, 2012, pp. 1615–1621.

[5] D. Winkler, S. Biffl, and T. Östreicher, "Test-Driven Automation: Adopting Test-First Development to Improve Automation Systems Engineering Processes," in *EuroSPI*, 2009, no. c, pp. 1–13.

[6] B. Kumar, B. Czybik, and J. Jasperneite, "Model based TTCN-3 testing of industrial automation systems - First results," in *IEEE Conference on Emerging Technologies in Factory Automation*, 2011.

[7] H. Carlsson, B. Svensson, F. Danielsson, and B. Lennartson, "Methods for Reliable Simulation-Based PLC Code Verification," *IEEE Trans. Ind. Informatics*, vol. 8, no. 2, pp. 267–278, May 2012.

[8] S. Süß, A. Strahilov, and C. Diedrich, "Behaviour simulation for Virtual Commissioning using co-simulation," in *IEEE Conference on Emerging Technologies in Factory Automation*, 2015.

[9] S. Ulewicz, D. Schütz, and B. Vogel-Heuser, "Software changes in factory automation: Towards automatic change based regression testing," *Annu. Conf. IEEE Ind. Electron. Soc.*, pp. 2617–2623, 2014.

[10] H. Simon, N. Friedrich, S. Biallas, S. Hauck-Stattelmann, B. Schlich, and S. Kowalewski, "Automatic test case generation for PLC programs using coverage metrics," in *IEEE Conference on Emerging Technologies in Factory Automation*, 2015.

[11] E. Jee, J. Yoo, S. Cha, and D. Bae, "A data flow-based structural testing technique for FBD programs," *Inf. Softw. Technol.*, vol. 51, no. 7, pp. 1131–1139, 2009.

[12] B. Bérard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, P. Schnoebelen, and P. McKenzie, *Systems and Software Verification*, vol. 144. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001.

[13] E. Clarke, O. Grumberg, and D. Peled, *Model Checking*. MIT Press, 1999.

[14] J. Lahtinen, J. Valkonen, K. Björkman, J. Frits, I. Niemelä, and K. Heljanko, "Model checking of safety-critical software in the nuclear engineering domain," *Reliab. Eng. Syst. Saf.*, vol. 105, pp. 104–113, 2012.

[15] R. Huuck, "Semantics and Analysis of Instruction List Programs," *Electron. Notes Theor. Comput. Sci.*, vol. 115, no. SPEC. ISS., pp. 3–18, 2005.

[16] N. Bauer, S. Engell, R. Huuck, 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*, 2004, pp. 517–540.

[17] A. N. I. Wardana, J. Folmer, and B. Vogel-Heuser, "Automatic program verification of continuous function chart based on model checking," *Annu. Conf. IEEE Ind. Electron. Soc.*, pp. 2422–2427, Nov. 2009.

[18] S. Biallas, J. Brauer, and S. Kowalewski, "Arcade.PLC: A Verification Platform for Programmable Logic Controllers," in *Conference on Automated Software Engineering (CASE)*, 2012, pp. 338–341.

[19] M. P. Remelhe, S. Lohmann, O. Stursberg, S. Engell, and N. Bauer, "Algorithmic verification of logic controllers given as sequential function charts," in *IEEE International Conference on Robotics and Automation*, 2004, pp. 53–58.

[20] J. Provost, J.-M. Roussel, and J.-M. Faure, "Translating Grafcet specifications into Mealy machines for conformance test purposes," *Control Eng. Pract.*, vol. 19, no. 9, pp. 947–957, Sep. 2011.

[21] B. Godlin and O. Strichman, "Regression verification: proving the equivalence of similar programs," *Softw. Testing, Verif. Reliab.*, vol. 23, no. 3, pp. 241–258, May 2013.

[22] D. Felsing, S. Grebing, V. Klebanov, P. Rümmer, and M. Ulbrich, "Automating regression verification," *Int. Conf. Autom. Softw. Eng.*, pp. 349–360, 2014.

[23] S. Ulewicz, B. Vogel-Heuser, M. Ulbrich, A. Weigl, and B. Beckert, "Proving equivalence between control software variants for Programmable Logic Controllers," in *IEEE Conference on Emerging Technologies in Factory Automation*, 2015.

[24] K. C. Duschl, D. Gramß, M. Obermeier, and B. Vogel-Heuser, "Towards a taxonomy of errors in PLC programming," *Cogn. Technol. Work*, vol. 17, no. 3, pp. 417–430, 2015.

[25] T. Mertke and G. Frey, "Formal verification of PLC programs generated from signal interpreted Petri nets," *2001 IEEE Int. Conf. Syst. Man Cybern. e-Systems e-Man Cybern. Cybersp.*, vol. 4, pp. 2700–2705, 2001.

[26] S. Roesch, D. Tikhonov, D. Schütz, and B. Vogel-Heuser, "Model-Based Testing of PLC Software: Test of Plants' Reliability by Using Fault Injection on Component Level," in *IFAC World Congress*, 2014, vol. 19, pp. 3509–3515.

[27] R. Drath, A. Fay, and T. Schmidberger, "Computer-aided design and implementation of interlock control code," *Conf. Comput. Aided Control Syst. Des.*, pp. 2653–2658, 2007.

[28] B. Vogel-Heuser, S. Feldmann, J. Folmer, J. Ladiges, A. Fay, S. Lity, M. Tichy, M. Kowal, I. Schaefer, C. Haubeck, W. Lamersdorf, T. Kehrer, S. Getir, M. Ulbrich, V. Klebanov, and B. Beckert, "Selected challenges of software evolution for automated production systems," in *IEEE International Conference on Industrial Informatics*, 2015, pp. 314–321.

[29] A. R. Bradley, "SAT-Based Model Checking without Unrolling," in *Verification, Model Checking, and Abstract Interpretation*, Springer Berlin Heidelberg, 2011, pp. 70–87.

[30] 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," 2014.

[31] "Companion Web Site for this Paper." [Online]. Available: http://formal.iti.kit.edu/projects/improve-aps/case16/.

[32] "The nuXmv model checker." [Online]. Available: https://nuxmv.fbk.eu/.