Modular Regression Verification for Reactive Systems*

Alexander Weigl[✉], Mattias Ulbrich, and Daniel Lentzsch

Karlsruhe Institute of Technology, Karlsruhe, Germany
`weigl@kit.edu`

Abstract. Reactive software is often deployed in long-running systems with high dependability requirements. Despite their safety- and mission-critical use, their functionalities must occasionally be adapted, for example to support new features or regulations. But software evolution bears the risk of introducing new malfunctions. Regression verification helps preventing the introduction of unintended, faulty behaviour.
In this paper we present a novel approach for modular regression verification proofs for reactive systems based on the idea of relational regression verification contracts. The approach allows the decomposition of a larger regression verification proof into smaller proofs on its subcomponents. We embedded the decomposition rule in a new algorithm for regression verification, which orchestrates several light- and heavyweight techniques. We implemented our approach for software used by Programmable Logic Controllers (PLC) written in Structured Text (IEC 611131-3) and show the potential of the approach with selected scenarios of a Pick-and-Place-Unit case study.

1 Introduction

Reactive software driving technical systems is often in operation for long periods of time, sometimes for many years or even decades. Guarantees regarding its correctness must be ensured over the entire system lifetime, and the software must go along and maintain quality through all hardware and software evolution steps. To avert malfunctions which may cause harm to humans or substantial financial losses, such reactive software is typically very thoroughly tested before deployment. In long-living systems, the confidence that the system's control software behaves correctly increases also with its successful operation time, as experience of its behaviour are gathered in various configurations and situations.

When software changes during an evolution step, thorough testing helps to identify software flaws and to increase confidence in the correctness of the system. While this is the standard in the industry, it has drawbacks: A test suite can never cover all possible scenarios, and confidence gained by experience with an old software revision cannot be transferred to a new revision.

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

A solution to this problem is *regression verification*: Instead of using two separate specifications for two revisions, the two revisions are compared directly to each other, where the old revision serves as a functional specification for the new one. The goal is then to prove equivalence, or a different specified relation, between both revisions. To this end, both software revisions are transformed into a logical representation, and then the combination is passed to a model checker for verification [1]. In previous work, we have shown that the resulting proof obligations can be discharged in some cases, but that the size of the system may make the verification approach suffer from the potential problem of state space explosion. Even for a simple case study with less complexity than real-world scenarios, proving equivalence with the approach described above took up to a day of computing time.

As a response to this challenge, this paper presents a technique to modularise regression verification by decomposing the verification condition into smaller subgoals which can be regression-verified individually. The novelty in comparison to existing model checking modularisation approaches is not that individual programs are decomposed into manageable fragments, but that the programs are split into pairwise blocks combined to be verified relationally.

The modular verification approach is embedded into a new regression verification algorithm which combines different lightweight (syntactical) and more heavyweight regression verification analyses.

Modularising the verification has multiple gains: Firstly, it reduces the state space of proof obligations, allowing them to be more feasible for model checking. Moreover, it introduces a locality principle: Parts of a program not touched at all by a refactoring can be factored out and equivalence be proven by simpler, syntactical techniques. For modules that occur more often, the verification effort can also be reduced since they only need to be analysed once.

A characteristic of reactive systems is that their code is executed periodically to react to changes in their application environment. Software for reactive systems is usually limited in the used (or allowed) programmatic constructs because they have to ensure real-time guarantees with deterministic runtimes. In many cases this means that there are no unbounded loops (or unconstrained recursion) in reactive systems software which allows us to unroll the code fully (thus eliminating the need for loop invariants or bounded analyses).

Contribution In this paper, we present a sound modularisation technique for the regression verification of reactive system software; it requires that relational specifications of subroutines are given (by the user). Moreover, we present a new algorithm for regression verification which orchestrates a collection of diverse heavy- and lightweight verification techniques making the new modular analysis more powerful in practice. We implemented the algorithm for PLC software, and demonstrate the feasibility of our approach on the Pick-and-Place-Unit (PPU) [12] – a community demonstrator for showing the evolution of manufacturing systems.

Outline In Sect. 2 we present the foundations of regression verification for reactive systems which is the basis for the modularisation approach. The approach itself is presented in Sect. 3 along with a formal definition on a basic program structure and specification. The approach is embedded into a new verification algorithm which we present in Sect. 4. The evaluation on the PPU and its results are discussed in Sect. 5.

2 Foundations

2.1 Regression Verification

Throughout their lifetime, systems have to adapt to new situations (bug fixes, hardware replacements, new function requirements, etc.) and many system changes will also incorporate changes in the software. Each software modification potentially introduces incorrect behaviour as a side-effect. To avoid the effect, *regression tests* are widely used in the industry as they yield good results and can easily be extended to new functionality of the software. However, software testing cannot guarantee correct behaviour since there will always be scenarios which are not covered by the test suite. Functional verification can help overcoming this problem: A formal specification describes the expected behaviour of the software and a verification system analyses whether the specification holds in all possible scenarios. But, the specification must be user-provided and is, in most cases, not trivial to find, especially when developers are less experienced with formal specification. For the verification in an evolutionary environment, two specifications are required: one for the existing software revision, and one for the new revision.

In regression verification, instead of using two specifications for two revisions, both revisions are compared directly to each other, and the old software revision serves as a functional specification for the new one. However, the old revision can only *partially* specify the new one since only those scenarios (input sequences) where the behaviour should not change can be checked for equivalence. The input sequences, for which the behaviour has been intentionally changed, need to be verified separately using functional verification or testing. Regression verification does not necessarily imply the software behaves correctly for all inputs, it rather says that the software has the same behaviour as the previous revision – including all potentially undiscovered errors. The confidence and trust gained by experience in the earlier revision is thus transferred to the new revision, as has been elaborated in [4].

2.2 Programmable Logic Controllers

The techniques in this paper are applicable to all kinds of reactive software systems. However, we put a special focus on Programmable Logic Controllers (PLC) as an example application area for the approach.

PLCs are computing units which are used to drive and control automated production systems. They are thus reactive real-time systems, and are usually in operation for a long time. In a PLC, the code is repeatedly executed once every few milliseconds. The constant time between two runs is called the *cycle time.*[1]

A family of programming languages for PLCs has been defined in the standard IEC 61131-3 [8]. While the languages are Turing-complete, PLC programs hardly ever contain general while-loops. If they contain loops, they have a known fixed upper bound on the number of iterations since PLC code has to meet strict real-time conditions. Dynamic memory allocation is not possible in the programming languages which makes them more predictable. This makes the state space for the software bounded, and the correctness problem theoretically decidable.

IEC 61131-3 has a concept of *modules* for structuring programs, similar to those used in common imperative programming languages. They allow one to encapsulate functionality into so-called *function blocks.* A function block consists of a variable signature (input, state and output variables) and an operation defined on this signature. It can be instantiated multiple times in other function blocks, and the invocation of an operation evaluates against the state of a particular instance.

In each execution cycle, the PLC obtains the current sensor values, executes the program, and emits newly computed actuator commands. Thus a PLC program is in continuous interaction with its environment. Besides the sensor and actuator values, PLC programs maintain an internal state. To work with deterministic and causal PLC programs formally, we hence model a PLC software as a function $P : I \times \Sigma \to O \times \Sigma$ which takes sensor readings (a value in I) and its current state from Σ and produces actuator values (in O) and the modified state. We lift P to $\overline{P} : I^* \to O^*$ with $\overline{P}(i_0, \ldots, i_n) = (o_0, \ldots, o_n)$, which takes a sequence of input values and returns the sequence of output values. $\overline{P}$ captures the iterated sequential execution of the effects of P. Formally, $\overline{P}$ is defined by the execution of $P(i_k, \sigma_k) = (o_k, \sigma_{k+1})$ for $0 \leq k \leq n$, and a sequence of memory states $(\sigma_0, \ldots, \sigma_n) \in \Sigma^*$ with a fixed initial memory state σ_0.

2.3 Formal Equivalence Relations

We briefly repeat the regression verification notions from [1], as these notion form the base of the modularisation our approach.

When we consider regression verification formally, we need to set two program behaviours into relation. The first notion that comes to mind is *perfect equivalence*, which requires that the behaviours of two PLCs programs P and Q are identical, i.e., that they produce the same output when presented with the same input trace. Formally, this means that they – interpreted as the two functions $\overline{P}$ and $\overline{Q}$ – are equal:

$$\text{for all } \bar{i}, \bar{i}' \in I^* : \; i = i' \implies \overline{P}(\bar{i}) = \overline{Q}(\bar{i}') \; . \tag{1}$$

However, they may very well differ on the chain of memory states reached in their traces, i.e., P and Q need not be identical. Perfect equivalence is a very strict notion for evolution scenarios, as it does not allow any behavioural difference

[1] There are also different execution modes for PLCs (event-driven, continuous, ...) that we do not consider here.

between the old and new revision. Still it is useful to prove that a *software refactoring* maintains the system behaviour.

In many evolution cases, behavioural differences must be taken into consideration to capture intended changes, like bug fixes or performance optimisations. The differences can be handled with the more flexible notions of *conditional and relational equivalence*. They extend perfect equivalence in two ways: Firstly, conditional equivalence allows us to filter scenarios that should not be included in the equivalence analysis using a predicate τ on the input values. Secondly, in relational equivalence one can replace the equalities in (1) by different relations that express the equivalence between input ($\approx_{in}$) and output ($\approx_{out}$) values:

$$\text{for all } \bar{i}, \bar{i}' \in I^* : \; \tau(\bar{i}, \bar{i}') \wedge \bar{i} \approx_{in} \bar{i}' \implies \overline{P}(\bar{i}) \approx_{out} \overline{Q}(\bar{i}') \; . \tag{2}$$

The triple $C = (\tau, \approx_{in}, \approx_{out})$ that parameterises (2) is called a *semantical regression verification contract* for P and Q. Perfect equivalence EQ is a special case of a regression verification contract with $EQ = (\mathit{true}, =, =)$. This generalises the ideas of design-by-contract [10] for single program properties to multi-program analyses. The condition (2), which we denote as $RV(C, P, Q)$, defines when the contract C is satisfied by the programs P and Q.

3 Modularisation

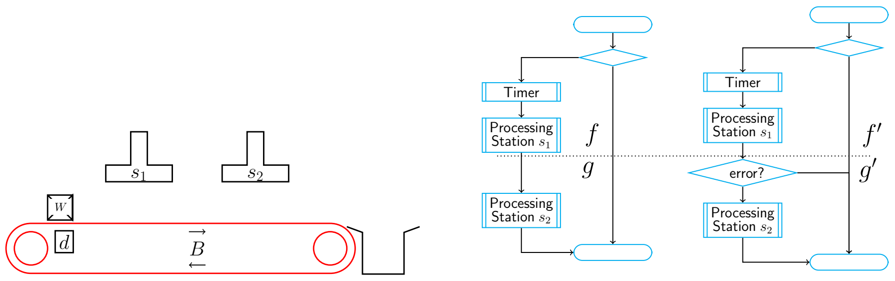
Modularisation is a technique to split up the program code into individual separate modules with defined interfaces. The effects of a module are limited to a specific scope, allowing a separate analysis. Wherever one module calls another module, the effects of the call can be abstracted rather than to include the full module implementation. Thus the complexity introduced by the control flow and internal state of the submodule are invisible in the caller module.

We present a decomposition rule which allows us to exploit the modularisation of reactive software to break down the regression proof obligation $RV(C, P, Q)$ into simpler proof obligations.

3.1 Motivational Example

Consider the plant in Fig. 1a representing an assembly line with a conveyor belt B and two processing stations s_1 and s_2 (e.g., a drill and a stamp). A detector d at the beginning of the conveyor belt recognises the arrival of a work piece W. Once a work piece has arrived, the automatic process starts, and W is moved from left to right, passing both processing stations, and eventually falling into the basket at the end of the belt.

In the original software revision, every work piece is unconditionally processed by both processing stations. While a piece is being processed, the conveyor belt halts for a defined amount of time. Let us assume that experience has shown that the process at s_1 may occasionally fail. The software has hereupon been adapted, and, after the revision, the plant can recognise work pieces for which s_1 has failed. If a faulty work piece leaves s_1, the second processing station should be skipped and the piece should be sent to the output basket directly.



(a) Schematic of the plant consisting of a conveyor belt B with two processing stations s_1 and s_2.

(b) Sketch of the program flow: the original revision on the left and the adapted revision on the right.

Fig. 1. Motivating Example

Software structure Fig. 1b shows a sketch of the program flow of the main program for both revisions. The difference is that a branching statement has been introduced after s_1. The modules "Timer" and the code for the processing stations remain unchanged.

Regression Verification and Modularisation Obviously, both software revisions behave differently when a faulty work piece occurs. To apply regression verification, a regression verification contract is required that specifies when both revisions should behave equally. In this example, the two revisions behave equally if no faulty work piece occurs. The contract for this example would therefore encode in the filter predicate that no faulty work piece is ever detected.

The non-modular approach for regression verification in [1] does not exploit the fact that the subroutines for controlling the hardware components remain unchanged. The full code of both programs is encoded for the translation against the regression contract with a model checker. The evaluation [1] (revisited in Sect. 5) shows that some evolution scenarios cannot be solved in a reasonable amount of time.

With the approach that we introduce here, we are able to replace the implementation of the modules in the encoding by their contracts, and can hence lower the verification effort by this abstraction which can thus become an enabler for the regression verification for larger programs.

This abstraction does not come for free. For a successful abstraction, sufficiently strong contracts that imply the necessary properties must be found. Finding them automatically may be as difficult as the whole program analysis itself. In the presented approach the user has to come up with suitable contracts.

3.2 Formalisation

The goal of this section is to look at composed programs and to introduce an inference rule that allows one to modularise regression verification proofs for such programs. Let therefore the two programs P, Q be implemented as a composition of two subprograms, say $P = f; g$ and $Q = f'; g'$. We have introduced programs as functions and the semicolon operator is the forward composition of functions (i.e. $(f; g)(x) = g(f(x))$).

For the modular analysis, it must be possible to identify the similar subprograms in P and Q that then become the corresponding parts between the two revisions. In the example from Sect. 3.1, for instance, the two programs can be split into two subprograms along the dotted line.

If one pair of corresponding subprograms can be verified in isolation (in this example g and g') for a contract C_g, this result can be used for the verification of the relation of the remainder programs where g and g' can be abstracted by (uninterpreted) placeholder function symbols x and x' which stand in for the programs g and g'. As a precondition in this proof obligation, we may assume the regression verification contract C_g for x, x' without knowing the exact functionality of g and g'.

The inference rule for the verification of $RV(C,\ f; g,\ f'; g')$ for a regression verification contract C has two premises which encode (1) that C_g is a valid regression verification contract for g and g' and (2) that the two programs satisfy contract C under the modular assumption that g and g' satisfy C_g.

$$\frac{RV(C_g, g, g') \qquad \forall x, x'.\ RV(C_g, x, x') \to RV(C,\ f; x,\ f'; x')}{RV(C,\ f; g,\ f'; g')} \tag{3}$$

3.3 Modularisation for Conditional and Relational Equivalence

In this section we present how the modularisation rule (3), formulated over functions, can be concretely used for the regression verification of programs. We start with the definition of a very general concept of a reactive programming language with frame structures, then introduce the decomposition rule, and close this section with remarks on properties of the rule.

Programs We consider simple loop-free programs, containing assignment- and if-statements. Additionally, we introduce a `frame`-construct for marking program parts which should be modularised. Programs are constructed by the grammar

$$\begin{aligned} \langle \mathit{Prg} \rangle \to\ & \langle \mathit{name} \rangle := \langle \mathit{expr} \rangle \mid \langle \mathit{Prg} \rangle \ ; \langle \mathit{Prg} \rangle \\ & \mid \texttt{if}\ (\langle \mathit{expr} \rangle)\ \{\ \langle \mathit{Prg} \rangle\ \} \mid \texttt{frame}(\langle \mathit{name} \rangle)\ \{\ \langle \mathit{Prg} \rangle\ \} \end{aligned} \tag{4}$$

in which the $\langle \mathit{name} \rangle$ denotes identifiers and $\langle \mathit{expr} \rangle$ side-effect-free expressions.

The set for programs produced by *Prg* is rather abstract and limited. However, it is expressive enough to encode reactive programs without (unbounded) loops. Programs in the low-level language (4) can, e.g., be constructed from more complex program languages like Structured Text or C by unwinding (bounded) loops and arrays, unfolding record data types and inlining procedure calls.

Frames and the Scope of Variables Frames structure the otherwise unstructured programs into modules. During the translation from input programs into the low-level language (4), structuring elements from the source language, like function-blocks or method invocations, are translated into frames. Frames can also be manually added by a user – to be able to handle complex code refactorings which took place across the boundaries of the structural elements in the source code, e.g., when a computation from inside a method is pulled out to the method caller.

For a sound abstraction and modularisation, the scopes of variables must be restricted, and the `frame` constructs mark these scopes. With every frame identifier N we associate three disjoint sets of variables: input (in_N), state ($state_N$) and output (out_N) variables. Every variable v occurring inside a frame named N must belong to one of them. The variables in these categories are constrained as follows: Input variables are only read within the frame, but may be written from outside the frame. For state variables read and written access inside the frame is allowed, but any access outside the frame is forbidden. Output variables are write-only within the frame, and read-only outside the frame. Global variables do not fit into this scheme, but can be encoded into it by an automatic program transformation.[2] Therefore, such a variable categorisation can always be established.

In a modularisation step, frames will be replaced by an abstraction using their contracts. The variables play an important role then: They manifest the interface at which the frame is abstracted for modular treatment. The input variables must adhere to a precondition on entry of the frame, the state variables can be removed from the program when the frame is abstracted, and the output variables assume values which adhere to a postcondition for the frame.

It is important to note that frame identifiers can occur on several frames within the same program. This models the case that multiple operations are invoked on the same module within a program. This happens, e.g., if the same function-block is invoked twice in an IEC-61131 context, or if a (stateful) procedure is called multiple times from the original program.

Frames that modify the same variables must have the same identifier, and all frames with the same identifier must have the same code and the same variable signature. This is not a restriction: If different functionalities access the same variables (e.g., different methods of an object in an object-oriented setting), programs can be refactored such that all frames contain the same integrated code that implements all functionalities. An additional parameter together with a case distinction is used to decide the concrete functionality in each frame.

Specification and Verification For both modular functional and modular regression verification, one needs contracts for the abstraction. In Sect. 2.3 we have already

[2] The program transformation introduces a new input and output variable for each global variable, which occurs in the frame. The global variable is assigned to the input variable at the beginning of the frame. The effect of the frame on a global variable is captured in the output variable, which is assigned to global variable after the frame.

encountered the concept of regression verification contracts on the semantic level. We will refine this notion now to program entities. Let two loop-free programs P and Q be given. A *regression verification contract* is a triple (ϕ, α, ω) of three formulas: the functional precondition ϕ, the relational precondition α and the relational postcondition ω. The semantics of these regression verification contracts are semantical contracts (Sect. 2.3). The formula ϕ evaluates to the filter predicate τ, and the interpretation of α and ω are the input and output equivalence relations.

The programs P and Q operate on disjoint sets of variables such that their statements programs cannot interfere with each other's state spaces, and are only connected in formulas within contracts. We can therefore use the sequential composition $P\,;Q$ to obtain the effects of their independent executions. The proof obligation which needs to be verified reads – written as a Hoare triple [7] –

$$\{\phi \wedge \alpha\}\ P\,;Q\ \{\omega\}\ . \tag{5}$$

In Section 4 we will describe efficient techniques to encode such proof obligations for decision procedures.

Modularisation Rule Let in the scenario introduced above, f and g be frame identifiers such that a frame for f occurs in P and a frame for g occurs in Q. For modular treatment, we need to look at the programs that abstract from the code of inner frames within their enclosing programs (as a parallel to the replacement of x for g in (2)):

Definition 1 (Factor program). *Let P be a program according to* (4) *and f be an identifier. The frames for f in P all have a unique occurrence number i. The factor program ${}^{P}\!/_{f}$ is then derived from P by replacing each frame i for identifier f with the following sequence of statements:*

1. $\mathtt{in}_i := \mathtt{in}$ *for every input variable* `in`
2. $\mathtt{count}_f := \mathtt{count}_f + \mathtt{1}$
3. $\mathtt{out} := \mathtt{out}_i$ *for every output variable* `out`

The freshly introduced variable $\mathtt{count}_f$ for the factored frame f is used to keep book about the number of invocations of f during a run of the program, and is needed to make the upcoming modularisation rule sound.

In non-regression program verification, modularised subprograms are often replaced by an obligation to show the precondition of the block and an assumption of the postcondition afterwards. Since in regression verification, we deal with two programs at a time, all we can do in the local context is to *remember* the values of all invocations for a global, program-spanning argument to take them into account. The following inference rule does precisely that. Instead of proving (5), one can show the two formulas that together imply it: (a) f and g together satisfy a regression verification contract $(\phi^{fg}, \alpha^{fg}, \omega^{fg})$, and (b) the factored programs satisfy the original regression contract. The intermediate variables in_i and out_i introduced by the factor program allow us to specialise a formula and set it into the context of one concrete call-site of the frame identifier. For a formula γ over the variables of P and Q, the instantiated formula $\left[\gamma\right]_{i,j}$ denotes the formula in which all occurring variables from P have been replaced by the counterpart of the i-th invocation and all variables in Q with the variables of the j-th invocation. For perfect equivalence $\varepsilon = (in^f = in^g \to out^f = out^g)$, the instantiated formula $\left[\varepsilon\right]_{1,2}$ would read $in_1^f = in_2^g \to out_1^f = out_2^g$.

Definition 2 (Modular regression verification). *For two programs P and Q (with disjoint variables) and frame identifiers f and g, let π_f and π_g denote the programs which are inside the corresponding frames f and g and let n (m) be the number of occurrences of f (g) in P (Q). For a regression verification contract $(\phi^{fg}, \alpha^{fg}, \omega^{fg})$ for π_f and π_g the inference rule*

$$\frac{\{\phi^{fg} \land \alpha^{fg}\}\ \pi_f\,;\pi_g\ \{\omega^{fg}\} \qquad \{\phi \land \alpha \land \kappa \land \Gamma\}\ {}^{P}\!/_{f}\,;{}^{Q}\!/_{g}\ \{\omega \land \kappa\}}{\{\phi \land \alpha\}\ P\,;Q\ \{\omega\}}$$

with $\Gamma = \bigwedge_{i=1}^{n} \bigwedge_{j=1}^{m} \left[\phi^{fg} \land \kappa \land \alpha^{fg} \to \omega^{fg}\right]_{i,j}$ and $\kappa = (\mathtt{count}_f = \mathtt{count}_g)$ is called the modularity rule.

The assumption Γ of the second premise couples the variables modelling the invocations of f and g. Whenever the input values for invocation occurrences i and j satisfy the precondition $\left[\phi^{fg} \land \alpha^{fg}\right]_{i,j}$ of the regression verification contract, the relational postcondition $\left[\omega^{fg}\right]_{i,j}$ is known to hold on the output values.

This rule is quite similar to the differential assertion checking approach using mutual function summaries by Lahiri et al. [9], but is applied here to frames with potentially more than one invocation and in the context of reactive systems in which the programs are called repeatedly. To allow for that, additional checks (encoded using the counting variables $\mathtt{count}_f$ and $\mathtt{count}_g$ in κ) have to be included that ensure that the number of invocations of the two abstracted frames is the same in both programs.

Properties This modularisation rule is sound. Although we do not elaborate on this here, an induction proof on the number of coupled invocations (captured in the program variables $\mathtt{count}_f$ and $\mathtt{count}_g$ introduced for this reason) can be conducted. The approach is not complete since we require that both systems invoke their frames equally often. There are systems which fulfil a regression contract, but do not have this property. Then this approach can currently not be applied. The rule is compositional, in the sense that it can applied recursively on the resulting proof obligations.

4 An Algorithm for Modular Regression Verification

In this section we construct a new regression verification algorithm for reactive software that combines a number of different modular and non-modular verification techniques. The algorithm takes two programs and a regression verification

```
function $Reve(f, f')$:
    Input: Two frames $f, f'$
    Data: A regression verification contract $(\phi, \alpha, \omega)$ for $f$ and $f'$.
    Output: true iff $f$ and $f'$ together satisfy the contract
    if check cache for $(f, f', \phi, \alpha, \omega)$ then
        // earlier results are cached
        return cached result;
    end
    if $(\phi, \alpha, \omega) = (true, =, =)$ then
        // only applicable for perfect equivalence
        return if true $EqualSource(f, f')$;
        return if true $EqualSE(f, f')$;
    end
    return if true $EqualSmt(f, f', \phi, \alpha, \omega)$;
    return if true $EqualAbstraction(f, f', \phi, \alpha, \omega)$;
    return $EqualityMC(f, f', \phi, \alpha, \omega)$;
```

Algorithm 1: Algorithm to check the equivalence of two frames

contract as input and checks if the programs satisfy the relational specification. We assume both programs have a top-level frame with the identifier `main` that contains all program statements. The algorithm works recursively – comparing first the outermost frames, trying to establish equality from going top to bottom in the program structures, recursively verifying the equality of enclosed subframes.

The algorithm orchestrates several different checkers and runs them in sequence returning on the first positive result. In the orchestration, we call the more syntactical, faster, but imprecise checkers first before falling back to more powerful, and more precise, but slower checkers. All checkers are sound: If they report that frames conform to their contract, then this is the case. They are not necessarily complete, and some checkers are only applicable on a restricted set of cases, for example perfect equivalence. The full algorithm – shown in Algorithm 1 – *is* complete as the last checker *EqualityMC* uses heavyweight model checking without abstractions and is complete.

The following sections briefly introduce the involved checkers.

4.1 Conformance by Syntactical Congruence

In case a contract specifies perfect equivalence, the checker *EqualSource* checks equivalence via a comparison of the syntax trees of the two source code artefacts. Prior to the comparison, we normalise the code (remove comments, unify capitalisation, . . .). Identical normalised source code implies equal software behaviour. Despite its severe restrictions, this method is a fast and useful checker, especially for frames resulting from often reused standard library procedures.

4.2 Conformance by Symbolic Execution

Checking equality by comparing the source code is very restricted, and fails, e.g., if two independent lines are swapped, or an irrelevant new variable is introduced. The next checker in the orchestration is *EqualSE*, which can handle such cases. It is still a syntactical checker; hence, it is also only able to handle perfect equivalence. This checker is based on symbolic execution to compute the symbolic results of a frame.

The result of the symbolic execution of a frame f is a function $F\colon \mathit{Var} \to \mathit{Expr}$ which maps every state and output variable to an expression which is the aggregation of all assignments to the variable in f. The term $F(v)$ computes to the value of v at the end of the frame and may depend on the input and state variables of f.

One possibility to show perfect equivalence between two frames f and f' is to establish syntactical equality between the symbolic execution results for all output variables. The equality must also be checked for those state variables which occur in the aggregated expressions of output variables to guarantee that the following cycles will produce equal output.

Thus far, we described the case where all input, output and state variables have the same name in both frames. To make this analysis more flexible, we allow arbitrary one-to-one mappings of variables between frames where the correspondence of input and output variables is given by a conjunction of equalities between variables in α and ω in the regression verification contract. For state variables, the mapping can be inferred. Furthermore, the mapping can be lifted from equalities over variables to equalities over expressions. The equality between output expressions given in the relational post-condition ω can be checked modulo the equalities in the relational precondition α.

The checker *EqualSE* is able to show the equality of $o = 2 * i + s$ and $o' = 2 * i' + t'$, where s, t' are state and i, i' are input variable. A matching needs to include the equality $s = t'$, and $i = i'$. Moreover, the equality of $i = i'$ (input variables) must be justified by the given regression contract ($\alpha \models i = i'$).

Due to its syntactical nature, this checker is incomplete, e.g., the equality between $o = 1 + 1$ vs. $o' = 2$, cannot be handled.

4.3 Conformance by Reduction to SMT

If these last syntactical checkers fail or are not applicable, the first semantical checker is triggered. This checker is backed up by a reduction to a Satisfiability Modulo Theories (SMT) problem using the previously computed symbolic execution results $F(v)$ and $F'(v')$ of the given frames. This checker is not limited to perfect equivalence, but can be used for arbitrary regression verification contracts.

The checker *EqualSMT* verifies an inductive relational invariant χ over the state variables of the two frames. In the simplest form we show that any state variables s and s' in f and f' evolve identically (i.e. $s = s'$). The formula to be checked for satisfiability is then

$$\left(\bigwedge_{v\in V} v^{+} = F(v)\right) \wedge \left(\bigwedge_{v\in V'} v^{+} = F'(v)\right) \wedge \phi \wedge \alpha \wedge \chi \wedge \neg(\omega^{+} \wedge \chi^{+}) \qquad (6)$$

where the sets of variables V and V' contain all output and state variables of f and f'. Variable v^{+} holds the result of the symbolic execution for v (via the function F or F'). It differs from v to distinguish variables before the execution from after it. A predicate χ^{+} results from χ by replacing v with v^{+}. If this formula is not satisfiable, χ is an invariant for the frames and, additionally, they conform to the regression verification contract (ϕ, α, ω).

As an example, consider the following contract $(\mathit{true}, i = i', o = o')$ for $o = 2 * i + s$ and $o' = t' + 2 * i'$. The instantiated SMT formula (6) for this example is

$$\underbrace{(o^{+} = 2 * i + s \wedge s = s^{+})}_{v^{+}=F(v)} \wedge \underbrace{(o'^{+} = t' + 2 * i' \wedge t' = t'^{+})}_{v'^{+}=F'(v)}$$
$$\wedge \underbrace{i = i'}_{\alpha} \wedge \underbrace{s = t'}_{\phi} \wedge \underbrace{o = o'}_{\omega} \wedge \neg(\underbrace{o^{+} = o'^{+}}_{\omega^{+}} \wedge \underbrace{s = t'}_{\chi^{+}}) \ ,$$

where o and o' are the output variables, s and t' state variables, and i and i' input variables, respectively. The relational invariant χ has been chosen as $s = t'$ in the example. It is a parameter of the checker, and in general non-trivial to infer. In our implementation we use the equality of equally named state variables for χ. In a further SMT verification condition (not shown here), it has to be shown that the initial memory states (cf. Section 2.2) of f and f' initially satisfy the coupling invariant χ.

4.4 Conformance by Modular Abstraction

The checker *EqualAbstraction* is the checker that exploits the modularisation rule introduced in Definition 2. Therefore, given two frames f, f', this checker starts with abstracting the top-level frames inside f and f', and uses Algorithm 1 for checking contract conformance of inner subframe pairs.

We assume that the subframes in f and f' are collected in pairs and that each frame pair is specified with a regression verification contract. Let g be a subframe in f, and g' in f', respectively. After the body of all subframes have been abstracted, we obtain the two factor programs f/g and f'/g' of both original frames together with a regression verification contract that has additional assumptions and postconditions. The regression verification algorithm is called recursively for $\mathit{Reve}(g, g')$ of each subframe pair and for $\mathit{Reve}(f/g, f'/g')$.

The modularisation rule may be applicable to several different subframes. In our implementation we eagerly apply it to all possible subframe combinations. The recursive procedure is applied recursively and exhaustively, but will eventually terminate since the frames are always finitely nested in a program.

If the modular abstraction step fails, it produces a counterexample (a finite trace, see Section 4.5) which may describe a genuine flaw in the system or it may be spurious if a regression verification contract does not hold or is not strong enough to serve as a suitable abstraction in the proof.

4.5 Conformance by Model Checking

The final checker is the most precise and most powerful one and encodes the verification condition into a model checking problem. This checker makes use of the non-modular regression verification approach by Beckert et al. [1] and verifies a regression verification contract specification between two complete frames f, f' without using abstraction. More precisely, the target is a problem in which an invariant (derived from the regression verification contract) for the system consisting of the two compared frames must be verified. Experience has shown that invariant-inferring techniques like the IC3 [2] approach (in particular the implementation within the model checker nuXmv [3]) work quite well for this type of regression verification problems.

Since the state space is finite, this checker is theoretically complete, i.e., returns within finite time for any input. However, experience shows that it can take hours or even days until the model checker comes back with a result. The modularisation technique and the combination with simpler techniques in *Reve* have been devised to reduce the time needed for regression verification challenges.

The model checker returns either that the inductive invariant has been proved (implying correctness of the contract), or it produces a counterexample, which is a concrete trace, i.e., finite sequence of assignments of input, state and output variables for both frames exemplifying the violation of the contract. We currently do not provide tool support, but these values can be used as inputs for a simulation of the reactive system like it is present in many modern IDEs for reactive software.

5 Evaluation

In this section, we show the applicability of our new regression verification algorithm on selected scenarios of the Pick-and-Place Unit (PPU) community

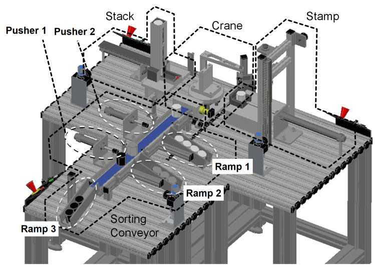


Fig. 2. Community demonstrator: the Pick-and-Place Unit

demonstrator [1, 12]. The PPU is a down-scaled model of a manufacturing plant employing industry-level hardware components that has been designed for researching the management of the evolution (hardware and software) of automated manufacturing systems. Therefore, there are multiple evolution scenarios – with software and/or hardware changes – of this plant. We selected representative evolution scenarios to cover different situations.

Fig. 2 gives an impression of the PPU in a medium expansion stage, as hardware configuration depends on the scenarios. Briefly described, the PPU consists of a magazine for providing new work pieces, a stamp for imprinting, a conveyor belt for sorting, and a crane for transportation of work pieces. All of these components and their actuators are controlled by the software in a PLC written in Structured Text (ST) and Sequential Function Chart (SFC), which we translated into ST code automatically (cf. [8]).

5.1 Selected Evolution Scenarios

We briefly explain the three selected evolution scenarios. The software revisions correspond to the different scenarios of the PPU in [12, Fig. 48].

Revision 1 vs. Revision 2 A new sensor is introduced for detecting metallic work pieces as a preparation for the next evolution. The software mainly changes the Crane module, but changes on the top-level module are needed to route the sensors to this submodule. An influence to the system behaviour is not expected: Both revisions are perfectly equivalent.

Revision 3 vs. Revision 5 Revision 5 introduces an optimisation which allows using the waiting time during stamping to transport work pieces which do not need to be stamped to the conveyor belt. The optimisation is only triggered if work pieces of different types are present (metallic and non-metallic). If only metallic work pieces are present, the two revisions behave perfectly equivalently. The work piece type can be determined by the program using the input variable $\mathit{CapacitiveSensor}$. We obtain a regression contract $(\mathit{CapacitiveSensor} = \mathit{true}, =, =)$ which intuitively formalises that the old and new revisions behave equivalently (equal inputs give equal outputs) under the condition that the sensor variable $\mathit{CapacitiveSensor}$ is true in every cycle.

Revision 12 vs. Revision 13 In the old revision, the position of the crane is measured with three switches (with Boolean sensor values $\mathit{OnConveyor}$, $\mathit{OnMagazin}$ and $\mathit{OnStamp}$). These are replaced by a single angular sensor. We need to define a relation R between the three boolean sensor values and the angle position

$$\begin{gathered}(16160 < \mathit{AnalogPosition} \wedge \mathit{AnalogPosition} < 16260) = \mathit{OnConveyor} \wedge \\ (24290 < \mathit{AnalogPosition} \wedge \mathit{AnalogPosition} < 24390) = \mathit{OnMagazin} \wedge \\ (8160 < \mathit{AnalogPosition} \wedge \mathit{AnalogPosition} < 8260) = \mathit{OnStamp}\end{gathered}$$

which serves the relational precondition in the regression verification contract $(\mathit{true}, R, =)$.

Table 1. Results

Rev./Module	Runtime							Code Size	
	Non-Mod. Total [s]	Modular Total [s]	Checkers [ms]						
			Src	SE	SMT	Modul.	Classic	LoC	#Vars
1 vs. 2	8.96	1.51						744	136
Main			0	48	65	545	–	744	136
– Main/*			0	10	0	–	–	174	203
Crane			0	21	35	441	–	415	51
– Crane/*			0	19	28	–	386	403	207
Magazine			0	13	–	–	–	234	38
3 vs. 5	750.0	7.05						1,605	256
Main			–	–	90	5,213	–	1,605	256
– Main/*			–	–	43	–	2,846	294	364
Crane			–	–	101	2,130	–	810	74
– Crane/*			–	–	101	–	1,987	768	376
Stamp			0	–	–	–	–	402	56
Magazine			0	–	–	–	–	240	44
12 vs. 13 −SE	t/o	34.76						4,808	520
Main			0	–	512	24,544	–	4,808	520
– Main/*			0	–	79	–	6,727	453	1,250
Conveyor			0	–	–	–	–	468	50
Crane			0	–	227	14,408	–	1,326	77
– Crane/*			0	–	238	–	14,168	1,284	631
Pusher			3	–	–	–	–	2,144	154
Stamp			0	–	78	4,801	–	403	57
Stamp/*			0	–	–	–	4,680	375	639
Magazine			0	–	61	–	2,795	241	45
12 vs. 13 +SE	t/o	34.76						4,808	520
Main			0	440	–	-	–	4,808	520

5.2 Results

Table 1 summarises the performance of the verification. The runtimes are shown for each checker on a frame. The first column describes the compared revisions and modules, where *Main* or *Crane* denotes the regression verification between the corresponding frames of both revisions. *Main/** denotes the frame with all subframes factored out. For convenience, Table 1 only shows the first and second level of nested frames. In particular, the frequently used timer module is hidden.

"Non-Modular Total" is the comparison reference value of applying the non-modular approach as in [1] with our pipeline. In comparison, "Modular Total" gives the overall runtime of the modular pipeline. Both total columns state the runtime measured from the command line. Hence they include the work needed to prepare the programs (parsing, symbolic execution, etc.). In contrast, the checker runtimes are given in milliseconds and are measured internally. A checker is skipped (marked with a dash (–) in the table) if either it was not capable of proving the regression contract, or a checker invoked earlier was able to solve this case. Note, for the comparison of Rev 12. vs. Rev 13 ("12 vs.13 −SE"), we have disabled *EqualSE* to evaluate the modularisation rule, because we want to demonstrate the capabilities of the decomposition rule. *EqualSE* can solve this comparison directly in half a second (cf. "12 vs.13 +SE" in Table 1). The lines of code do not include empty lines or comments and cover both code modules.

Also the number of variables (#Vars) give the sum of input, state and output variables of both frames.

The runtimes (wall clock) are the median of three samples, computed on an Intel Core i7-8565U, 16 GB RAM, using the model checker nuXmv 1.1.1 [3] with IC3 for invariant checking, and z3 4.8.8 for solving the SMT instances. The time-out was set to 1 hour. Our algorithm implementation is single-threaded. All of the verification artefacts and a link to the source code are available online[3].

5.3 Discussion

The evaluation shows a huge speed-up against the previous non-modular approach from [1]. It shows the potential of modularisation to enable the handling of large reactive systems. For fair comparison, we repeated the experiments of [1], but we use the default bit-width for integers on PLC languages, and also we did not reduce the blocking time of the used timers. Rev. 12 against Rev. 13 ran into a time-out, [1] gives a clue that the verification can take more than 22 hours. Most of the performance should result from abstracting these timer, which are used to wait a particular amount of time. During this time span, the system stutters partially, resulting in long phases of forward searches in IC3.

6 Related Work

Beckert et al. [1] applies regression verification to PLC software and is the first base for this work. Subroutines in PLC software is handled by inlining the subroutines in its caller context. We reuse their notion of regression verification (Sect. 2.3). Also we use their pipeline to simplify PLC programs and prepare them for model checker.

Modularisation for regression verification is covered in [5] which serves as a second basis to our work. Godlin and Strichman [5], who also coined the term "regression verification" exploit both regression verification and decomposition to prove equivalence between similar programs. They are able to handle programs with recursive function calls and unbounded loops, both are paradigms are not common in software for reactive systems. Nevertheless, their work does not cover our topic completely: They only consider functions that do not have an internal state and require them to be perfectly equivalent. Moreover, the decomposition in [5] works bottom-up if possible. Our approach work from top to bottom.

The work on differential assertion checking [9] modularises relational proofs in a similar fashion to the one presented in this paper. They employ *mutual function summaries* to abstract two related functions blocks, which is essentially the same concept as our regression verification contracts. They do not target reactive systems but individual single function invocations, and use the intermediate verification language Boogie to encode their conditions rather than a model checking verification backend.

[3] `http://formal.iti.kit.edu/isola20`

The goal of Guthmann et al. [6] is similar to ours: Modularising the equivalence proof. For matched procedures two partial sets are computed. One contains input states where the procedures behave equivalent and one where they differ. Both sets are approximated. The approximation are made stronger the longer the algorithm runs. They extended their approach work with demand-based refinement of the approximated sets in [11].

7 Conclusion

In this paper, we have motivated and presented a new verification rule for the modular decomposition of regression verification proof obligations for reactive system software. Moreover, we have integrated the rule into a novel regression verification algorithm which orchestrates five different regression verification approaches into one proof technique. Thanks to the modularisation, simpler equality checkers allow one to show properties more easily on subproblems.

The evaluation indicates a tremendous performance improvement: Modularisation can allow regression verification proofs to run orders of magnitudes faster.

Future Work A drawback of the decomposition technique is the need for (user-specified) regression contracts. In most cases, these specification seem to be automatically inferable, e.g., by using heuristics, symbolic execution or Horn solvers. In our implementation, we have not used any sophisticated strategy to decide whether a frame should rather be kept inlined or be abstracted. The implementation tries to abstract all allowed frames at once, which seems to be a good strategy. A more restrictive selection could bring further advantage.

References

[1] Beckert, B., Ulbrich, M., Vogel-Heuser, B., Weigl, A.: Regression verification for programmable logic controller software. In: ICFEM 2015. LNCS, vol. 9407, pp. 234–251. Springer (2015)

[2] Bradley, A.R.: Sat-based model checking without unrolling. In: Jhala, R., Schmidt, D. (eds.) Verification, Model Checking, and Abstract Interpretation, LNCS, vol. 6538, pp. 70–87. Springer (2011)

[3] Cavada, R., Cimatti, A., Dorigatti, M., Griggio, A., Mariotti, A., Micheli, A., Mover, S., Roveri, M., Tonetta, S.: The nuXmv symbolic model checker. In: CAV 2014. LNCS, vol. 8559, pp. 334–342. Springer (2014)

[4] Cha, S., Ulbrich, M., Weigl, A., Beckert, B., Land, K., Vogel-Heuser, B.: On the preservation of the trust by regression verification of PLC software for cyber-physical systems of systems. In: INDIN 2019. pp. 413–418. IEEE (2019), `https://doi.org/10.1109/INDIN41052.2019.8972210`

[5] Godlin, B., Strichman, O.: Regression Verification: Proving the Equivalence of similar Programs. Software Testing, Verification and Reliability 23(3), 241–258 (2013)

[6] Guthmann, O., Strichman, O., Trostanetski, A.: Minimal unsatisfiable core extraction for SMT. In: Piskac, R., Talupur, M. (eds.) FMCAD 2016. pp. 57–64. IEEE (2016), https://doi.org/10.1109/FMCAD.2016.7886661

[7] Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM 12(10), 576–580 (1969), https://doi.org/10.1145/363235.363259

[8] International Electrotechnical Commission: IEC 61131: Programmable controllers – Part 3: Programming languages (Feb 2002)

[9] Lahiri, S.K., McMillan, K.L., Sharma, R., Hawblitzel, C.: Differential assertion checking. In: ESEC/FSE'13. pp. 345–355. ACM (2013), https://doi.org/10.1145/2491411.2491452

[10] Meyer, B.: Applying "design by contract". IEEE Computer 25(10), 40–51 (1992)

[11] Trostanetski, A., Grumberg, O., Kroening, D.: Modular demand-driven analysis of semantic difference for program versions. In: SAS 2017. LNCS, vol. 10422, pp. 405–427. Springer (2017), https://doi.org/10.1007/978-3-319-66706-5_20

[12] Vogel-Heuser, B., Legat, C., Folmer, J., Feldmann, S.: Researching evolution in industrial plant automation: Scenarios and documentation of the pick and place unit. Tech. rep., Institute of Automation and Information Systems, Technische Universität München (2014), https://mediatum.ub.tum.de/node?id=1208973