Provably Forgetting of Information in Manufacturing Systems*

Verification of the KASTEL** Industry Demonstrator

Alexander Weigl

Institute of Theoretical Informatics
Karlsruhe Institute of Technology

Abstract. During the manufacturing process, information are generated and aggregated that constitute a business secrets and therefore need a high protection. On the other hand, if we can prove, that an information is absented, the effort for the protection for this system could be invested on different information, aspects or systems.

For this, we develop the notion of information forgetting of a reactive system. This notion describes that a reactive system needs to forget the information about a secret within a certain amount of cycles. This property limits the amount of historical information an attacker can learn by observing a manufacturing system. Moreover, we formalise and prove the notion of an *information forgetting* system with *Relational Test Tables.*

We evaluate the verification on the industry demonstrator for KASTEL SVI project, which was provided by the Fraunhofer IOSB and developed by industrial third-party contractor. In this demonstrator, we are able to show, that a selected business secret – the number of wheel turns – is not forgotten. We suggest and prove a fix of the leak.

We close with an elaborate discussion on the verification and results and also with remarks to the how *information forgetting* relates supports quantifiable security.

Keywords: Information flow control, information forgetting, formal security, Relational Test Tables

1 Introduction

In the era of the industrial revolution (IR4.0), information security becomes an increasingly important aspect of industrial manufacturing systems. As these

* This work was supported by the German Federal Ministry of Education and Research within the framework of the project KASTEL SVI in the Competence Center for Applied Security Technology (KASTEL).

** "The Competence Center for Applied Security Technology (KASTEL) is one of three competence centers for cyber security in Germany, which were initiated by the Federal Ministry of Education and Research (BMBF) in March 2011." — KASTEL Website, accessed 2019-11-06 system should be more configurable and adaptable, the amount of software within these system increases. Moreover the manufacturing system and the enterprise resource planing system (ERP) needs to share more information, e.g. the manufacturing system needs to announce finished work pieces, the ERP configures the manufacturing system according to the customer's wishes of the next job. The information becomes a valuable target, either for violating confidentiality or integrity of the manufacturing process.

In this report, we try to verify whether a business secret is stored inside the control unit of a manufacturing system. The control unit, often called Programmable Logic Controller (PLC), is a computer, on which reactive programs are executed. Execution of a program is triggered every n milliseconds. It begins with reading of the sensors values and ends with writing of the computed actuator values to the underlying bus system.

The configuration and processing information of manufacturing system can contain very sensitive and crucial information of the manufacturing process or turnovers. These business secrets[1] are protected by the German law. To gain this protection, a company needs to protect the data by using state-of-the-art methods (cf. [6] and § 2 Nr. 1 lit. b) GeschGehG). Therefore, a company is interested to know in which components their data is stored to apply protection measurements more purposeful.

Our demonstrator is a configurable colour wheel. The PLC software controls the direction and speed of the connected rotating colour wheel. The PLC software is either controlled manually by the operator via an human machine interface (HMI), or runs a runtime-defined control sequence. Our goal is to verify that the number of turns of the colour wheel is not stored within the state of the software.

The verification subject. The program to be verified is the control software of an automated production system (aPS), that does not produce any work pieces, but uses the real components and programming languages of the aPS domain. The aPS consists of a Programmable Logic Controller, an HMI interface and motor rotating a colour wheel. The software lets the colour wheel spin, either by inputs from the HMI, or automatically in configurable sequences.

The software was developed by a sub-contractor of the Fraunhofer IOSB. Originally, the demonstrator was designed for demonstrating a replay-attack [7] on the network level and how this attack is detected by an anomaly detection. The components are connected via Ethernet. This gives the opportunity to an infiltrator to manipulate the sensor and actuator commands. In the intended attack, an attacker sends malicious packets to control the colour wheel.

Business Secrets. For the KASTEL demonstrator, we follow a different story: Business secrets are confidential information of a company, and protected by law. This protection requires efforts by the owning company to protect their data following the state of the art. A typical business secret is the amount of work pieces, e.g. cars or enriched uranium, that are produced within a time interval.

[1] in german: Geschäftsgeheimnis

For demonstration, we assume that the amount of colour wheel turns represents a business secret. We want to show, that the program fulfills the following property.

Definition 1 (Informal Property). *The PLC software* does not *store the number of turns of the colour wheel.*

Therefore any attacker is not able to derive this information by observing a single internal state of the PLC.

Outline. We explain the software components, architecture and information flow in Section 2.1. In Section 2.2, we present the steps that were taken to obtain a verifiable program, e. g. the removal of floating point variables. The property and verification are presented in Section 3.

2 Program to be Verified

In this section we explain the software to be verified. First, we give an overview over the structure (Section 2.1). Second, we identify the verified fragment and needed preparation steps (program transformations, Section 2.2).

The software was developed by a industrial third-party contractor in charge of the Fraunhofer IOSB designed to demonstrate their Intrusion Detection System for replay attacks in industrial communication networks. These attacks are closer described in [7]. The hardware of the demonstrator is shown in Fig. 1.

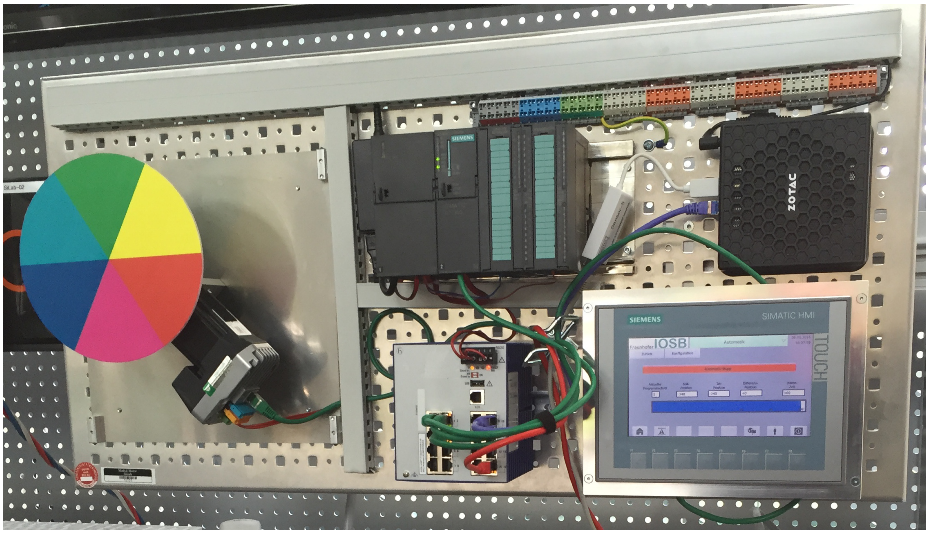

Fig. 1. Hardware components of the system to be verified. Image provided by Fraunhofer IOSB

The following paragraphs are paraphrased from the technical documentation provided from the Fraunhofer IOSB. The core functionality of the PLC is to control the motor via EtherCAT. The PLC supports two modes: automatic and manual operation. The mode is selected by an integrated HMI.

In the automatic mode, the PLC executes a user-defined sequence of steps. A step consists of a target position (angle), velocity, acceleration, deceleration and waiting time. The PLC drives the wheel to the target position with the defined velocity and ac- and deceleration. If the position is reached, it waits the defined waiting time and then proceeds with the next step. Depending on the configuration the system leaves the automatic mode after the sequence is completely executed or restarts with the first step. The automatic mode can be paused or aborted. In the manual mode, the users can interact with the system more directly via the HMI. The user can stop and spin the wheel in both directions with a user-defined, or predefined velocity. Also the manual mode allows to set the reference position of the wheel.

2.1 Software Architecture

The software is mainly implemented in Structured Text (ST) (IEC 61131-3) and consists out of 16 user-defined data types, two function blocks, two function (initialisation and communication with HMI) and the main program[2]. Function blocks are special to IEC 61131-3. Function blocks are functions with an internal state. A function block can be instantiated in other function blocks or programs and consecutive invocations are executed on the internal state and the input variables. A function block can have multiple output variables.

The Fig. 2 visualises the internal architecture and the execution. The main program is executed cycle-wise every n ms. It starts with the call to the Function `Initialisation()`. This function ensures a correct initialised the global state. Mainly it ensures that the error messages are defined in String variables and all the arrays are pre-filled. The initialisation function is only executed once, i. e. in the first cycle. In the second step the current values from the HMI are transferred to the global state. Third, the main program determines the operation mode, either STOP, MANUAL or AUTOMATIC[3]. The fourth step invokes the function block `SequenceAutomaton`, which only handles the automatic mode. This automaton decides whether the motor needs to move, the target position is reached, or the waiting time is elapsed and the next step should be executed. These decisions are based on the sequence of user-defined entries within the global state. A distinct internal variable describes the current state of the sequence execution (cf. Fig. 2). The call to `MainAxis` triggers the most important part of the software: the motor control. There are 15 modes in this function block. The mode variable is set internally or externally by the main program or the sequence automaton. A

[2] Additionally, there are seven auxiliary functions, mostly for converting to and from external sensor values.

[3] The automatic mode is split into a mode for pre-selection of the auto settings (AUTO) and executing the automatic mode (ACTIVE AUTO).

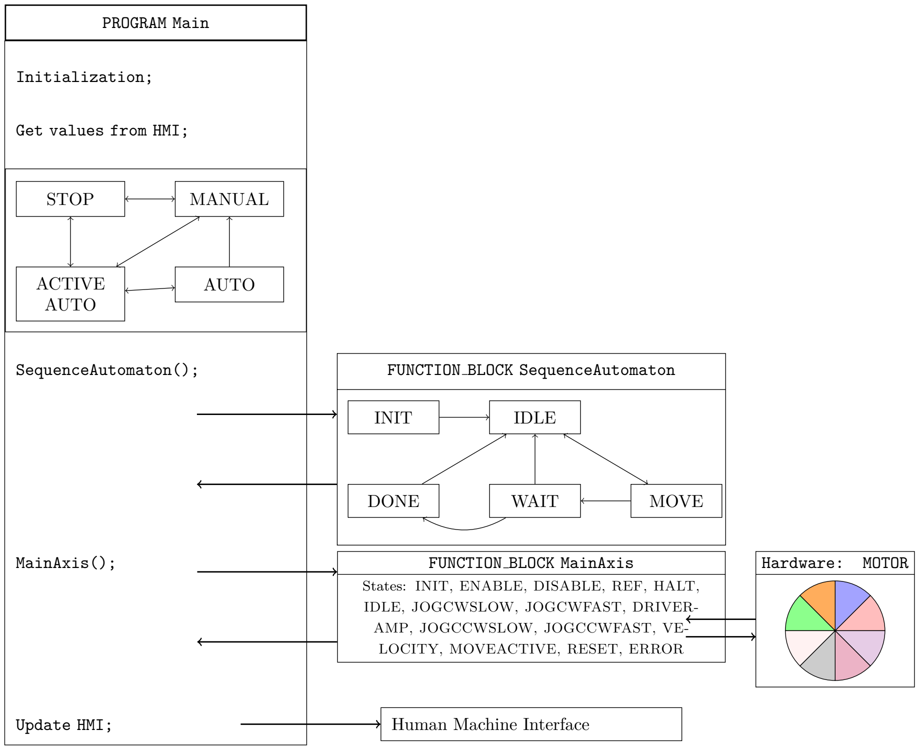

Fig. 2. Architecture of the software consisting out of four structural elements: main program, sequence automaton, main axis control and HMI.

mode corresponds to specified parameter set in calls of motor control driver[4]. For example: if the mode is HALT, the driver parameters are set to stop the motor, and at the end of this function block the driver is called. Erroneous and success calls are handled by `MainAxis` by jumping to the IDLE or ERROR mode. The code of `MainAxis` and its dependencies before the preparation for the verification is given in Appendix B.

The function blocks communicate by setting variables in the global state. For example the function block `Sequence Automaton` sets the mode of `MainAxis` directly and `MainAxis` sets value for the HMI.

The program sizes are: `Initialisation` has 54 LoC, Program `Main` 97 LoC (reading from HMI 40 LoC, operation mode 45 LoC), Function Block `MainAxis` has 362 LoC, Function Block `SequenceAutomaton` has 65 LoC, and writing to HMI has 81 LoC.

[4] The driver function blocks are an extension of PLCOpen Motor Control.

2.2 Preparations for Verification

For the verification we concentrate on the function block `MainAxis`. But before the verification, we need apply program transformations to bring this function block into a supported shape for the symbolic execution and the model checker. In the remaining paper we do not distinguish between state and output variables of the function block. We consider the output variables as part of the state.

The starting point is original implementation of the function block `MainAxis`, the global state and the auxiliary functions. We start by simplifying the function block into ST_0 – a simplified version of Structured Text where all loops are unwound and invocations to function blocks are inlined. The transformation is described in [2]. Secondly, we need to apply simplifications customised for the given software.

We remove assignments to `dScratch` and `VSObj_McFaultDescription`. The first location is a global variable that is never read, but is written. The second one holds a String value of the current error cause in the HMI, and unsupported by the model checker.

The model checker can not handle floating values. Therefor we transform variables of type REAL to INT. Additionally, we need to remove the conversion functions `REAL_TO_INT` and `INT_TO_REAL` with the identity function. We apply the same for the used – and not needed anymore – rounding of values (`(x/1000)*1000`).

In the last transformation, we slice the program to remove all variables, that are neither read or written, and remark the remaining variables as input and output according to their reading and write access.

The resulting program for the verification code is 421 LoC.

3 Verification

Proof obligation. Our goal is to verify, that the given function block `MainAxis` does not store any information about the business secret: the number of turns of the wheels. In the following we analyse and break down the informal specification the into a formal property, that can be checked. The final proof obligation is given in Section 3.1.

Non-Interference Property. Physically, the number of wheel turns can be derived by integrating the angular velocity $v_a(t)$ in a time interval $[n, m]$:

$$\#turns := \left\lfloor \frac{1}{360} \int_n^m v_a(t)\, dt \right\rfloor \quad .$$

$\#turns$ represents the precise amount of wheel rotations, whereas the PLC is only capable to capture an estimation of $\#turns$, due to limitations of the data type, sensor values and impreciseness in triggering the cycle. In the remaining section we do not distinguish between precise or estimated number of turns. An attacker should not learn anything about the number of turns $\#turns$, after they have observed the current state σ of the PLC. Mathematically expressible using probabilities:

$$P(\#turns \mid \sigma) = P(\#turns) \ , \tag{1}$$

where P($\#turns$) describes the apriori probability distribution of the number of turns, where as P($\#turns \mid \sigma$) is the aposteri distribution after observing the state, and $\#turns$ and σ correspond to the same point in time. Equation (1) corresponds to the non-interference of $\#turns$ and σ.

Equation (1) expresses a strict property that an attacker learns nothing by observing a state σ of the PLC software. This property is too restrict. The initial state is already a counter example: An attacker can determine that a given state is the initial state[5]. And from this information, they can derive $\#turn = 0$.

Additionally, the presented non-interference property (1) can not be modelled by a forbidden information flow between variables. An information flow exists if the state of a program variable h influences a different variable l. For confidentiality considerations, the variable h represents the secret information and l the public observable output. In our case, the complete state of the PLC is public observable. The secret is the number of turns that is not directly available by observing a state, but it is derivable from a given path. Alternatively, we could use the sensor value of the velocity as the secret. This is more restrictive, because then nothing is allowed to depend on this sensor value and therefore this variable could silently be removed. In general, forbidding an information flow from a sensor value to the internal state is too limiting for manufacturing system. The system needs to react to events and these events are recognised by sensor values. In the demonstrator it is easy to recognise that the velocity sensor is read and stored internally.

3.1 Property

Information Forgetting. We need to find a relaxed information flow property, that allows that a secret can be stored for a short time inside the state, and will eventually be forgotten later on.

Let us make a gedanken experiment with two instances of the demonstrator. First, we run both demonstrators for an arbitrary amount of time and different velocities, resulting in different number of turns. Second, we synchronise the sensor inputs of both systems for a short amount of time. Third, we stop both systems and inspect their internally state. If the states are indistinguishable, then the number of wheel rotations are not derivable anymore.

In contrast to information flow we introduce an *annealing phase*. During the annealing phase the secret information should be superseded. We prove that an attacker – observing a single state of the PLC – can only derive information about the k last cycles. For manufacturing systems, cycle times are rather small ($\leq 10ms$) and therefor the time window, in which the states are distinguishable, are short. The formalisation is given in form of a relational test table (rtt) in Fig. 3.

[5] The internal variable `FC_Init` is set to true only in the very first state.

Relational test tables. Relational test tables are a canonical extension of generalised test tables (gtt) [4, 1]. Gtts are a table-based specification language. The columns correspond to input, output or state variables. The rows are the steps in the test, which are executable from top to bottom. Each row has an interval as a time constraint, which describes how often a row can be applied consecutively. Rows may be skippable. The cells contain the constraints for the corresponding column and row. For easy accessibility, gtts allow the use of abbreviations, e.g. the cell content > 10 enforces that the designated column variable should be bigger than 10 or an interval $[5, 10]$ for defining an allowed value range. Abbreviations are translated into Boolean expression.

Applying a row means, that we pick up an input adhering the constraints of columns for the input variables and the current row, executing the system, and checking if the emitted output obtains the corresponding output constraints. During verification we check every possible sequence of rows and every possible input composition. We say a system is conform to a gtt, if we cannot find a sequence of described input values that leads to a violation of the output constraints [1, cf. WEAK conform].

A gtt allows us to describe the behaviour of single runs of a system, but our property talks about two runs of two system. Rtts overcome this restriction and allow us to specify k-safety properties [9]. Our gedanken experiment is a 2-safety property [5]. Rtts brings two changes into gtts: First, the variable access needs to qualify to which program run they correspond. Second, the program runs can be paused independently. During the pause, they stutter in their local state, but the input variables may change. For each program run there is an extra PAUSE column, that determines if the program run should be paused (`TRUE`) or not (`FALSE`, default value). For more details on rtts, refer to [9].

Extensions to relational test tables. For readability of our specification (Fig. 3) we make two extensions to gtts.

First, we allow that a column can also be designated to a function. The abbreviations of the cell contents are checked against the evaluation of the corresponding column header. If the column header contains a variable, the abbreviation are expanded against value of the variable. Additionally, we now allow that column header is a function on the input, output and state values. Then, the abbreviation are expanded against the evaluation of the function in the current state.[6] This extension correspond to the widely used concept of *model variables*. The model variables, like column functions, exist only in the specification domain.

Second, we add a new cell abbreviation for separately defined predicates. If a cell entry consists only out of a predicate $\mathcal{P}$ without arguments, we interpret the cell as the application of the predicate to the evaluation of column header. Let f be a function, which returns a tuple of values $f : \sigma \mapsto (x_1, \ldots, x_n)$, of the corresponding column, then we interpret $\mathcal{P}$ as $P(x_1, \ldots, x_n)$. Scalar values are

[6] We can say, a variable v in the column header describes a function $f_v\colon \sigma \mapsto \sigma(v)$, where σ is the current state.

silently lifted. For rtts, a function in the column header has the same arity as the number of program runs.

Both extensions do not extend the expressibility of gtts, but allow us to write more comprehensible generalised test tables by reducing and externalising of variables and expression. Instead of writing complex expression in the table, we can concentrate on the most important: the consecutive execution flow of the test.

Proof obligation. Figure 3 shows the relational test table that captures our gedanken experiments. We define $V \otimes V'$ for two variable signatures V, V' as a projection function of two states, to two tuples representing the values of the variables in V and V'.

$$V \otimes V' := \qquad \lambda\sigma, \sigma'.\ (\pi_V(\sigma), \pi_{V'}(\sigma')) \tag{2}$$

where $\pi_V(\sigma) := (\sigma(v_1), \ldots, \sigma(v_n))$ (with $n = |V|$) denotes the projections of the state σ to a tuple of the values of variables in V.

We define S to be the variable signature that contains the local state variables, analogue I_L for the low input, and I_H for the high input variables. $S \otimes S$ maps two states to a tuple, where the first element matches the local state of the first run, analogue the second element for the second run. The same is valid for $I_L \otimes I_L$ for the low input and $I_H \otimes I_H$ for the high output variables.

#	**PAUSE**		**INPUT**			**OUTPUT**	⏲
	0	1	$S \otimes S$	$I_L \otimes I_L$	$I_H \otimes I_H$	$S \otimes S$	
0			=	=	—	—	1
1			—	=	—	—	—
2			—	=	=	—	k
3			—	=	=	=	ω

Fig. 3. Template of relational test table for information forgetting

In the rtt, we use following relations on these state projections:

- The relation "—" stands for don't-care, and does not enforce any constraint on the column value.
- The relation "=" is the symbolic equality and enforces that the first and second element of the tuple are equal.

In Fig. 3, Row 0 expresses that local states and low inputs of both program runs need to be equal. Where our secret inputs (the velocity) can differ between both runs. We do not care about the state (and output variable) at the end of the invocation. In Row 0, we allow that both state can differ, caused by the different values for secret inputs, but the input remains equivalent for I_L. For the demonstrator we have $|S| = 32$ $|I_L| = 51$, and $|I_H| = 1$. The exact variables for I_L, I_H and S are given in Appendix A.

You can non-deterministically decide whether you stay in Row 1 or start the annealing phase in Row 2. Row 2 enforces that also the secret I_H is equivalent in both runs. After k cycle, the states of both runs need to be equivalent (Row 4)—indicating that the secret previously injected is forgotten.

For efficiency, we start both systems in equal states. This is an over-approximation as this formalisation also includes the initial states described by the language semantics. This trick reduces the diameter of the of the search space.

3.2 Result

The complete transformation pipeline is implemented in our verification library for automated production systems and is publicly available[7]. After the translations, the state space in the model checker is 566 bits large (270 bits input, 296 bits state).

We instantiated our property (Fig. 3) with $k = 2$ for the annealing phase. For the verification we used nuXmv 1.1.1 [3] on an Intel® Core™ i5-6500 (3.20GHz) with 16 GB RAM.

The system does not adhere our property (Fig. 3). nuXmv finds a counter example in 1.85 sec. over(median, $n = 3$). So there exists a run that does not lead to vanish of the secret information about the past velocities.

Inspecting the counter example shows the reason why the different velocities – given via `ActStep.rVelocity` – are result into different value in the state variable `MoveAxis1.Velocity` after $k = 2$ cycles of equal input. The visited states of `MoveAxis` are states: `MOVESTATE_ABSOLUTE`, `MOVESTATE_MOVEACTIVE`, and `MOVESTATE_INIT` (cf. Appendix B).

Fixing the leak. Inspection of the counter examples gives us a hint which variable leaks the secret information: `MoveAxis1.Velocity`. Further, we can proof that all the others variables do not inferred with secret anymore. We are using the same formalisation but exclude `MoveAxis1.Velocity` from set of state variables S.

Going further, we have three possibilities to remove the information of the leaking variable: First, we manually inspect the variable and its information flow, and conclude that this variable is independent of the number of turns. Second, we can modify the program and overwrite the variable. This step changes the behaviour of the program and has to checked against the documentation. Third we we could enforce that the system has to visit a state that enforce an overwrite of `MoveAxis1.Velocity`. After inspecting the code, you may find out that this variable is only written if `state=MOVESTATE_ABSOLUTE`. This steps alters and limits the verification condition, in such a way that you assume that certain occurs occasionally.

We decided for the second alternative and added two assignments at the end of the code (cf. Appendix B):

```
MoveAxis1.Velocity := 0; MoveAxis1.Execute := FALSE;
```

[7] https://github.com/verifaps/verifaps-lib

The first assignments overrides the velocity, s.t. the variable does not leak this information. The second assignments disables the command execution of the instance `MoveAxis1` of the Function Block `MC_MoveRelative` [8] in the next cycle. The driver function blocks are called at the end of the Function Block `MoveAxis` for sending commands to the motor controller of the colour wheel. Setting `Execute` prevent that the controller that the a velocity of 0 is sent to the controller in the next cycle. If a new velocity needs to be set, the `MoveAxis1` re-enables the execution and also sets the velocity.

Model checking runtime. The Table 3.2 gives an overview about the runtime: for finding the counter example in the original leaky program (A), proving that only `MainAxis1.Velocity` leaks(B), and proving the fixed version (C). Sample size is $n = 2$. We omit the standard derivation; in all runs it was lower than 20 seconds. The runtime of the model checker depends heavily on the number of cycles k to forget the information. The parameter k highly influences the depth search space, as it determines the number of unwinding the system definitions.

Table 1. Runtime of the model-checker for proving or finding a counterexamples of the information forgetting for various annealing phases k and scenarios (A) original leaky version, (B) original leaky version proving all other variables do not leak, and (C) fixed (non-leaky) version.

$k =$	2	3	5	7	10
(A)	3.39 sec	2.95 sec	2 min 52 sec	9 min 24 sec	2 h 50 min
(B)	57.40 sec	45.82 sec	3 min 32 sec	10 min 29 sec	2 h 36 min
(C)	40.93 sec	29.74 sec	3 min 5 sec	10 min 46 sec	1 h 33 min

4 Discussion

After inspecting the code, we are sure, that the software holds the informal property (Definition 1) of not storing the number of wheel turns. The formal property (Fig. 3) is still stronger than the informal property. It is important to note, that abstracting the environment (other function block and hardware) of the `MoveAxis` and letting the systems start in arbitrary equal states can lead to spurious counter examples. Also note, we avoid the problem of the leak in the initial state by considering only traces with at least two cycles (time duration in Row 0 in Fig. 3).

Restriction to the Function Block. We decided to verify the Function Block `MoveAxis` as it is the most complex and critical software part inside this software project, and deals finally with the sensors and actuators. Hence, every control request passes this piece of code. It was out of our scope whether the other function blocks adheres the property. This includes the human-machine-interface (HMI) and also Function Blocks of the motor driver. Both sub systems are not completely accessible from the PLC software, but may be observable by an attacker. The PLC software can only access the shared variables for communication or the given input and output parameter, especially this includes the current velocity. The internal state, i. e. the user-interface elements, is not accessible or modelled for verification. On a real attack, the attacker sees also the complete user-defined program sequence, containing the information of the current segment, its position, velocity, etc. From these program sequence, an attacker might guess an estimation of the previous amount of turns, but also an estimation of the future amount of turns. Moreover, we only looked at the PLC software. Information may be stored inside the physical plant itself and are fed back to the PLC via sensors. Without a suitable environment model, information flow in the physical plants are not traceable. The internal actions of the PLC, e. g. reading sensors values, setting actuators values, debug interface, are also uncovered from our considerations.

Single observable state. We limit the leakage in our attacker model to one PLC software state. In practise attacks expand over several days to months. An attacker may see every observable state during this infiltration period. Our approach keeps still useful: the attacker can not guess information, which are lying past its infiltration without additional consumption. One of these consumption could be that the attacked industrial system is running the same program with the same parameter before its successful infiltration.

Program transformation. For the verification we apply some program transformation, i. e. demoting floating point variable to integer variables, removing string, unread or unwritten variables. These transformation can be critical and need a justification case by case. For example, code lines could become unreachable using integer instead of floating-point arithmetic. In contrast, symbolic execution and other simplification, like structure unfolding, are uncritical as they are not change the semantic of the program, special the set of reachable states remain the same.

Why verification on the PLC level? In our demonstrator, we prove the privacy on the second lowest level of the automation pyramid. The field or electronic level – containing the sensors and actuators – is beneath the PLC, and upper PLC is the HMI-SCADA system, the manufacturing enterprise system (MES) and the enterprise resource planing system. The upper level are gathering the process from the lower levels, and may store the business information for which we tried to prove that they are forgotten. Nonetheless, verification of the PLC are needed. Due to their real-time requirements, protection of PLC against attacks are hard to achieve without threaten the functionality. The upper level are built with *standard* PC components and may be protected with standard equipment. On the other side, attacks on the lower sensor and actuator level were observed, which made the protection of the PLC more difficult.

Other formalisation. There may exist other formalisation of Definition 1. For example: We can assure that for every trace t with number of wheel turns

$\#turns_t$, there exists a second possible trace t' with $\#turns_{t'} \neq \#turns_t$, and the last states of t and t' are equal

$$\forall t \in traces.\ \exists t' \in traces. \quad t_{|t|} = t'_{|t'|} \ \wedge\ \#turns_{t'} \neq \#turns_t\ .$$

After observing a state of PLC, an attacker could not distinguish whether $\#turns_t$ or $\#turns_{t'}$ is the real value of turns (assuming that no additional information are known). But in the worst case, there are only two possible turns $P(\#turns_t) \geq 0$ and $P(\#turns_{t'}) \geq 0$ and both numbers of turns are almost identical: $|\#turns_t - \#turns_{t'}| = 1$.

5 Quantification

Our presented approach is a quantification of security, because we can quantify *how fast* information is forgotten and *how much* information is forgotten. Both numbers are on an ordinal scale – they can help in the comparison of the security of systems. Because, a system that forgets *more* information *faster* is more secure. This ordinal can be considered from the view of *risk assessment*. A risk is formed by two factors: entry probability and costs in the event of damage or loss. Our approach does not prevent that an attacker can successfully capture a PLC system. But if a successful attack occurs, the attacker sees a limited and known amount of information. Therefore, if a system forgets more information faster, it has a lower risk, because of the reduced costs – whereby entry probability keeps the same. On the other side, we do not have an interval scale, as it is invalid to state, that a system is two times more secure than an other system if it forgets the same information two times faster. For the cost assessment, it is crucial which information are kept in the system.

6 Conclusion

In this paper we develop a notion and formalisation of an information-forgetting system. This notion is a relaxed variant of an information flow property, where we give a system a time span (annealing phase) in which the system needs to forget secrets. A system that dependently forgets the Business Secrets, is not protected against successful intrusion, but in case of an intrusion the amount of leaked secrets are reduced.

We apply this notion to a manufacturing system provided by the Fraunhofer IOSB with the goal to prove that a certain business secret – the number of wheel turns – are not derivable if the attacker has access to one local state. We prove, that the information of the velocity flow only into one single state variable, and by code revision we see that the velocity is only assigned and not accumulated.

References

[1] Bernhard Beckert et al. "Generalised Test Tables: A Practical Specification Language for Reactive Systems". In: *Integrated Formal Methods*. Ed. by Nadia Polikarpova and Steve Schneider. Cham: Springer International Publishing, 2017, pp. 129–144. ISBN: 978-3-319-66845-1.

[2] Bernhard Beckert et al. "Regression Verification for Programmable Logic Controller Software". In: *17th International Conference on Formal Engineering Methods (ICFEM 2015)*. Vol. 9407. LNCS. Springer, Dec. 2015, pp. 234–251.

[3] Roberto Cavada et al. "The nuXmv Symbolic Model Checker". In: *Computer Aided Verification (CAV)*. LNCS 8559. Springer, 2014, pp. 334–342.

[4] Suhyun Cha et al. "Applicability of generalized test tables: a case study using the manufacturing system demonstrator xPPU". In: *at - Automatisierungstechnik* 66.10 (Oct. 2018), pp. 834–848. DOI: `10.1515/auto-2018-0028`. URL: `https://doi.org/10.1515/auto-2018-0028`.

[5] Michael R. Clarkson and Fred B. Schneider. "Hyperproperties". In: *Journal of Computer Security* 18.6 (2010), pp. 1157–1210. ISSN: 0926227X. DOI: `10.3233/JCS-2009-0393`.

[6] Dirk Müllmann. "Auswirkungen der Industrie 4.0 auf den Schutz von Betriebs- und Geschäftsgeheimnissen". In: *Wettbewerb in Recht und Praxis (WRP) 2018* 64.10 (2018), pp. 1177–1184.

[7] Steffen Pfrang et al. "Design and Architecture of an Industrial IT Security Lab". In: *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*. Springer International Publishing, Nov. 2016, pp. 114–123. DOI: `10.1007/978-3-319-49580-4_11`. URL: `https://doi.org/10.1007/978-3-319-49580-4_11`.

[8] TC2 Task Force Motion Control. *Technical Specification: Part 1 -Function blocks for motion control*. eng. Tech. rep. Version 2.0. Mar. 17, 2011. 141 pp.

[9] Alexander Weigl et al. *Relational Test Tables: A Practical Specification Language for Evolution and Security*. Tech. rep. Karlsruher Institut für Technologie (KIT), 2019. 10 pp. DOI: `10.5445/IR/1000099122`.

A Variables

State variables S: `JogAxis1$Backward`, `JogAxis1$Fast`, `JogAxis1$Forward`, `Mode`, `MoveAxis1$Distance`, `MoveAxis1$Execute`, `MoveAxis1$Velocity`, `MoveVelAxis1$Execute`, `MoveVelAxis1$Velocity`, `PowerAxis1$Enable`, `ReadActPosAxis1$Enable`, `ReadActVelAxis1$Enable`, `ReadStatusAxis1$Enable`, `RefAxis1$Execute`, `RefAxis1$Mode`, `RefAxis1$Position`, `Reset$Execute`, `SetDriveRampAxis1$Acceleration`, `SetDriveRampAxis1$Deceleration`, `SetDriveRampAxis1$Execute`, `StatusAxis1$ActPosition`, `StatusAxis1$ActVelocity`, `StatusAxis1$DeltaPosition`, `StopAxis1$Execute`, `bAxisInPosition`, `bNotMoving`, `iScratch`, `rLastPositionCmd`, `stHmiInt$stMCStatus$tMC_Cmd`, `state`, `tTimeout$IN`, `tTimeout$PT`

Low input variables (I_L): ActStep$rAccel, ActStep$rDeccel, ActStep$rPosition, JogAxis1$Busy, JogAxis1$CommandAborted, JogAxis1$Error, MoveAxis1$Busy, MoveAxis1$CommandAborted, MoveAxis1$Error, MoveVelAxis1$CommandAborted, MoveVelAxis1$Error, MoveVelAxis1$InVelocity, PowerAxis1$Error, PowerAxis1$Status, ReadActPosAxis1$Error, ReadActPosAxis1$Position, ReadActPosAxis1$Valid, ReadActVelAxis1$Error, ReadActVelAxis1$Valid, ReadActVelAxis1$Velocity, ReadStatusAxis1$Error, RefAxis1$Done, RefAxis1$Error, Reset$Done, Reset$Error, Sequence$bAutoRelease, Sequence$eState, SetDriveRampAxis1$Busy, SetDriveRampAxis1$Done, SetDriveRampAxis1$Error, StatusAxis1$Disabled, StatusAxis1$Error, StatusAxis1$StandStill, StopAxis1$Done, StopAxis1$Error, stHmiInt$rIncrement, stHmiInt$rStartVel, stHmiInt$stMCStatus$bMC_Error, stHmiInt$stReq$bReset, stHmiInt$stReq$stMan$bDecrVel, stHmiInt$stReq$stMan$bDisable, stHmiInt$stReq$stMan$bHome, stHmiInt$stReq$stMan$bIncrVel, stHmiInt$stReq$stMan$bJogFFwd, stHmiInt$stReq$stMan$bJogFRev, stHmiInt$stReq$stMan$bJogFwd, stHmiInt$stReq$stMan$bJogRev, stHmiInt$stReq$stMan$bStartVel, stHmiInt$stReq$stMan$bStop, stHmiInt$stStepData$rDeltaPos, tTimeout$Q

High input variables (I_H):

ActStep$rVelocity

B Program to be Verified

```
(****************************************************************************)
(*** ENUMS ***)

TYPE

    McCmds_t    : (MCCMD_NONE, MCCMD_POWER, MCCMD_HALT, MCCMD_MOVEJOG,
                   MCCMD_MOVEVEL, MCCMD_MODMOVE, MCCMD_RESET, MCCMD_REFPOS);

    ModeState_t : (MODE_NOMODE, MODE_MANUAL, MODE_AUTO, MODE_AUTOACTIVE);

    MoveState_t : (MOVESTATE_INIT, MOVESTATE_ENABLE, MOVESTATE_DISABLE,
                   MOVESTATE_REF, MOVESTATE_IDLE, MOVESTATE_HALT, MOVESTATE_SETDRIVERAMP,
                   MOVESTATE_ABSOLUTE, MOVESTATE_VELOCITY, MOVESTATE_MOVEACTIVE,
                   MOVESTATE_JOGCWSLOW, MOVESTATE_JOGCWFAST, MOVESTATE_JOGCCWSLOW,
                   MOVESTATE_JOGCCWFAST, MOVESTATE_ERROR, MOVESTATE_RESET );

   SeqState_t   : (SEQSTATE_INIT, SEQSTATE_RESET, SEQSTATE_IDLE,
                   SEQSTATE_MOVE, SEQSTATE_WAIT, SEQSTATE_DONE);
END_TYPE

(****************************************************************************)
(***  RECORDS ***)
TYPE

ST_AxisStatus : STRUCT
 Valid              : BOOL; (*status is available*)
 Busy               : BOOL; (*busy*)
 Error              : BOOL; (*error occured*)
 Errorstop          : BOOL; (*drive stopped due to an error*)
 Disabled           : BOOL; (*drive is disabled*)
 Stopping           : BOOL; (*drive is stopping*)
 Referenced         : BOOL; (*drive is referenced*)
 StandStill         : BOOL; (*drive is not moving*)
 DiscreteMotion     : BOOL; (*drive moves in discrete motion*)
 ContinuousMotion   : BOOL; (*drive moves in continuous motion*)
 SynchronizedMotion : BOOL; (*drive moves in synchronized motion*)
 Homing             : BOOL; (*drive is referencing*)
 ConstantVelocity   : BOOL; (*drive moves with constant velocity*)
 Accelerating       : BOOL; (*drive is accelerating*)
 Decelerating       : BOOL; (*drive is decelerating*)
 ActPositionUsr     : DINT; (*Actual Position usr*)
 ActPosition        : REAL; (*Actual Position ° *)
 DeltaPosition      : REAL;
 ActVelocityUsr     : INT;  (*ActualVelocity = rpm*)
 ActVelocity        : REAL; (* Actual Velocity °/s        *)
 ActAccelerationUsr : INT;  (*Actual Acceleration = rpm/s2*)
 ActAcceleration    : REAL; (*Actual Acceleration = °/s2*)
 ActDecelerationUsr : INT;  (*Actual Acceleration = rpm/s2*)
 ActDeceleration    : REAL; (*Actual Acceleration = °/s2*)
 RefVelocityUsr     : INT;
 RefVelocity        : REAL;
END_STRUCT;

stSm : STRUCT
  bStatechange : BOOL;
  bAutoRelease : BOOL;
  bCycleStop   : BOOL;
  bReset       : BOOL;
  bStepFwd     : BOOL;
  bStepRev     : BOOL;
  reStepper    : BOOL;
  iActStep     : INT;
  iMaxStep     : INT;
  eState       : SeqState_t;
  tStepDelay   : TON;
END_STRUCT;

stSeqParams : STRUCT
  rPosition : REAL; (*  Command position in 360.000° * 1000 *)
  rVelocity : REAL; (*  Command velocity in °/sec  *)
  rAccel    : REAL; (*  Command acceleration in °/sec² *)
  rDeccel   : REAL; (*  Command decceleration in °/sec² *)
  dPause    : DINT; (*  Dwelltime between steps *)
END_STRUCT;



    ST_Hmi_Segment : STRUCT
        StartAngle : INT;
        EndAngle : INT;
        Angle : INT;
        Color : DWORD;
        Invisible : BOOL;
    END_STRUCT;

ST_McOutputs : STRUCT
        Done : BOOL;
        CommandAborted : BOOL;
        Error : BOOL;
        Busy : BOOL;
        Status : BOOL;
        Valid : BOOL;
        ErrorID : UDINT;
    END_STRUCT;

stHmi : STRUCT
        bWatchdog : BOOL;
        stReq : stHmi_Req;
        stStepData : stHMI_ActStepData;
        stMCStatus : stHMI_MCStatus;
        rStartVel : REAL;
        rIncrement : REAL;
        strOpMode : string;
```

```
        rActVelo : REAL;
        bDirectionCW : BOOL;
        bDirectionCCW : BOOL;
        stSegments : array[0..9] of ST_Hmi_Segment;

    END_STRUCT;

    stHMI_ActStepData : STRUCT
        iStep : INT;
        rCmdPos : REAL;
        rActPos : REAL;
        rDeltaPos : REAL;
        stTimes : stHMI_ActStepData_Times;
    END_STRUCT;

    stHMI_ActStepData_Times : STRUCT
        dPT : DINT;
        dET : DINT;
        dRT : DINT;
    END_STRUCT;

stHMI_MCStatus : STRUCT
        bMC_Error : BOOL;
        tMC_Cmd : McCmds_t;
        udMC_ErrorID : UDINT;
        strMC_ErrorString : string;
    END_STRUCT;

stHmi_Req : STRUCT
        stMan : stHmi_Req_Man;
        stSeq : stHmi_Req_Seq;
        bReset : BOOL;
        bAuto : BOOL;
        bManual : BOOL;
        bStart : BOOL;
    END_STRUCT;

stHmi_Req_Man : STRUCT
        bJogFwd : BOOL; (* Request Axis Jog CW (Fwd) *)
        bJogFFwd : BOOL; (* Request Axis Jog CW (Fwd) *)
        bJogRev : BOOL; (* Request Axis Jog CCW (Rev) *)
        bJogFRev : BOOL; (* Request Axis Jog CCW (Rev) *)
        bIncrVel : BOOL;
        bDecrVel : BOOL;
        bStartVel : BOOL;
        bStop : BOOL; (* Request Axis Jog Stop ? Useles when JOGGING? *)
        bHome : BOOL; (* Request Axis Home *)
        bDisable : BOOL; (* Disable Axis for Home Request *)
    END_STRUCT;

stHmi_Req_Seq : STRUCT
        bReset : BOOL; (* Reset Sequence to first step *)
        bFwd : BOOL; (* Goto next step *)
        bRev : BOOL; (* Goto previous step *)
        bCycleStop : BOOL; (* Inhibit sequence from starting over *)
    END_STRUCT;

stSeqParams : STRUCT
        rPosition : REAL; (* Command position in 360.000° * 1000 *)
        rVelocity : REAL; (* Command velocity in          °/sec *)
        rAccel : REAL; (* Command acceleration in °/sec² *)
        rDeccel : REAL; (* Command decceleration in °/sec² *)
        dPause : DINT; (* Dwelltime between steps *)
    END_STRUCT;

    stSM : STRUCT
        bStatechange : BOOL;
        bAutoRelease : BOOL;
        bCycleStop : BOOL;
        bReset : BOOL;
        bStepFwd : BOOL;
        bStepRev : BOOL;
        reStepper : BOOL;
        iActStep : INT;
        iMaxStep : INT;
        eState : SeqState_t;
        tStepDelay : TON;
    END_STRUCT;


   Axis_Ref_ETC_ILX : STRUCT END_STRUCT;
END_TYPE

FUNCTION_BLOCK MC_Power_ETC_ILX
    VAR_INPUT
        Enable : BOOL;
        Axis : Axis_Ref_ETC_ILX;
    END_VAR
    VAR_OUTPUT
        Status: BOOL;
        Error : BOOL;
    END_VAR
END_FUNCTION_BLOCK

FUNCTION_BLOCK MC_Jog_ETC_ILX
    VAR_INPUT
        Forward, Backward, Fast : BOOL;
        TipPos, WaitTime, VeloSlow, VeloFast: DINT;
        Axis : Axis_Ref_ETC_ILX;
    END_VAR
    VAR_OUTPUT
        Done, Busy, CommandAborted, Error: BOOL;
    END_VAR
END_FUNCTION_BLOCK


FUNCTION_BLOCK MC_MoveRelative_ETC_ILX
    VAR_INPUT
        Execute : BOOL;
        Distance, Velocity: DINT;
        Axis : Axis_Ref_ETC_ILX;
    END_VAR
    VAR_OUTPUT
        Done, Busy, CommandAborted, Error: BOOL;
```

```
    END_VAR
END_FUNCTION_BLOCK


FUNCTION_BLOCK MC_MoveVelocity_ETC_ILX
    VAR_INPUT
        Execute : BOOL;
        Velocity: DINT;
        Axis : Axis_Ref_ETC_ILX;
    END_VAR
    VAR_OUTPUT
        InVelocity, Done, Busy, CommandAborted, Error: BOOL;
    END_VAR
END_FUNCTION_BLOCK

FUNCTION_BLOCK MC_Stop_ETC_ILX
    VAR_INPUT
        Execute : BOOL;
        Axis : Axis_Ref_ETC_ILX;
    END_VAR
    VAR_OUTPUT
        Done, Busy, Error: BOOL;
    END_VAR
END_FUNCTION_BLOCK



FUNCTION_BLOCK MC_ReadActualPosition_ETC_ILX
    VAR_INPUT
        Enable : BOOL;
        Axis : Axis_Ref_ETC_ILX;
    END_VAR
    VAR_OUTPUT
        Valid, Busy, Error: BOOL;
        Position : DINT;
    END_VAR
END_FUNCTION_BLOCK



FUNCTION_BLOCK MC_ReadActualVelocity_ETC_ILX
    VAR_INPUT
        Enable : BOOL;
        Axis : Axis_Ref_ETC_ILX;
    END_VAR
    VAR_OUTPUT
        Valid, Busy, Error: BOOL;
        Velocity : INT;
    END_VAR
END_FUNCTION_BLOCK

FUNCTION_BLOCK MC_ReadStatus_ETC_ILX
    VAR_INPUT
        Enable : BOOL;
        Axis : Axis_Ref_ETC_ILX;
    END_VAR
    VAR_OUTPUT
        Valid, Busy, Error, Errorstop, Disabled,
        Stopping, Referenced, Standstill, DiscreteMotion,
        ContinuousMotion, SynchronizedMotion, Homing, ConstantVelocity,
        Accelerating, Decelerating: BOOL;
    END_VAR
END_FUNCTION_BLOCK


FUNCTION_BLOCK SetDriveRamp_ETC_ILX
    VAR_INPUT
        Execute : BOOL;
        Acceleration, Deceleration: UINT;
        Axis : Axis_Ref_ETC_ILX;
    END_VAR
    VAR_OUTPUT
        Valid, Busy, Error, Done: BOOL;
    END_VAR
END_FUNCTION_BLOCK


FUNCTION_BLOCK MC_Reset_ETC_ILX
    VAR_INPUT
        Execute : BOOL;
        Axis : Axis_Ref_ETC_ILX;
    END_VAR
    VAR_OUTPUT
        Valid, Busy, Error, Done: BOOL;
    END_VAR
END_FUNCTION_BLOCK

FUNCTION_BLOCK MC_SetPosition_ETC_ILX
    VAR_INPUT
        Execute : BOOL;
        Position : DINT;
        Mode : BOOL;
        Axis : Axis_Ref_ETC_ILX;
    END_VAR
    VAR_OUTPUT
        Valid, Busy, Error, Done: BOOL;
    END_VAR
END_FUNCTION_BLOCK

FUNCTION ABS : INT
VAR_INPUT a,b:INT; END_VAR
IF a <= b THEN ABS := a; ELSE ABS:=b; END_IF
END_FUNCTION

(*****************************************************************************)
(* GLOBAL VARIABLES **********************************************************)

VAR_GLOBAL
 Axis1                      : Axis_Ref_ETC_ILX;
 ReadStatusAxis1            : MC_ReadStatus_ETC_ILX;
 StatusAxis1                : ST_AxisStatus;
 state                      : MoveState_t; (* state machine state *)

 MCDiagAxis1                : ARRAY[0..16] OF ST_McOutputs;
```

```

 ReadActPosAxis1           : MC_ReadActualPosition_ETC_ILX;
 //ReadActPosAxis1Out        : ST_McOutputs; (* debug function block output data *)

 ReadActVelAxis1           : MC_ReadActualVelocity_ETC_ILX;
 //ReadActVelAxis1Out        : ST_McOutputs; (* debug function block output data *)

 SetDriveRampAxis1         : SetDriveRamp_ETC_ILX;
 //SetDriveRampAxis1Out      : ST_McOutputs; (* debug function block output data *)

 PowerAxis1                : MC_Power_ETC_ILX;
 //PowerAxis1Out             : ST_McOutputs; (* debug function block output data *)

 RefAxis1                  : MC_SetPosition_ETC_ILX;
 //RefAxis1Out               : ST_McOutputs; (* debug function block output data *)

 StopAxis1                 : MC_Stop_ETC_ILX;
 //StopAxis1Out              : ST_McOutputs; (* debug function block output data *)

 JogAxis1                  : MC_Jog_ETC_ILX;
 //JogAxis1Out               : ST_McOutputs; (* debug function block output data *)

 MoveAxis1                 : MC_MoveRelative_ETC_ILX;
 //MoveAxis1Out              : ST_McOutputs; (* debug function block output data *)

 MoveVelAxis1              : MC_MoveVelocity_ETC_ILX;
 //MoveVelAxis1Out           : ST_McOutputs; (* debug function block output data *)
 MoveVelAxis1OutAtVelocity : BOOL;

 Reset                     : MC_Reset_ETC_ILX;
 //ResetOut                  : ST_McOutputs; (* debug function block output data *)

 bFS                       : BOOL; (* First scan flag *)

 Mode                      : ModeState_t; (* device mode states *)
 bModechange               : BOOL; (* change in device mode, true for one cycle *)

 fbSequence                : FB_Sequence;
 bAxisInPosition           : BOOL;

 stHmiInt                  : stHmi; (* Interface structure to HMI *)
END_VAR

VAR_GLOBAL PERSISTENT
 Sequence   :   stSM;
 aSeqParams :   ARRAY[0..16] OF stSeqParams;
END_VAR

FUNCTION_BLOCK TON

VAR_INPUT
    IN : BOOL;
    PT : USINT;
END_VAR

VAR_OUTPUT
    Q  : BOOL;
    ET : USINT;
END_VAR

END_FUNCTION_BLOCK

FUNCTION ActSetDriveRamp        : VOID END_FUNCTION
FUNCTION ActPower               : VOID END_FUNCTION
FUNCTION ActSetPosition         : VOID END_FUNCTION
FUNCTION ActStop                : VOID END_FUNCTION
FUNCTION ActMoveJog             : VOID END_FUNCTION
FUNCTION ActMove                : VOID END_FUNCTION
FUNCTION ActMoveVel             : VOID END_FUNCTION
FUNCTION ActReset               : VOID END_FUNCTION
FUNCTION ActReadActualVelocity  : VOID END_FUNCTION
FUNCTION ActReadStatus          : VOID END_FUNCTION
FUNCTION ActReadActualPosition  : VOID END_FUNCTION
FUNCTION ActReadStatus          : VOID END_FUNCTION




FUNCTION FC_Init : BOOL

VAR iIndex : INT; END_VAR

(*** Initialise SeqParams on first scan ***)

Sequence.iMaxStep := 7;
FOR iIndex := 0 TO 7 DO
        aSeqParams[iIndex].rPosition := (iIndex) * 60;
        IF (iIndex MOD 2 = 0) THEN
                aSeqParams[iIndex].rPosition := aSeqParams[iIndex].rPosition * -1;
        END_IF
        IF iIndex = 0 THEN
                aSeqParams[iIndex].rVelocity := 2000;
        ELSE
                aSeqParams[iIndex].rVelocity := 100;
        END_IF
        aSeqParams[iIndex].rAccel := 1000;
        aSeqParams[iIndex].rDeccel := 1000;
        aSeqParams[iIndex].dPause := 2000;
END_FOR

Sequence.bCycleStop := FALSE;
stHmiInt.rStartVel := 1000.0;
stHmiInt.rIncrement := 1000.0;
FC_Init := TRUE;
END_FUNCTION


PROGRAM MainAxis
VAR
  dScratch         : DINT;
  iScratch         : INT;
  rLastPositionCmd : INT;
  bReverse         : BOOL;
  bForward         : BOOL;
```

```
  tTimeout         : TON;
  bNotMoving       : BOOL;
END_VAR

(*** update the axis status at the beginning of each cycle ***)
ReadStatusAxis1.Enable := TRUE;
ActReadStatus();
IF ReadStatusAxis1.Error THEN
    state := MOVESTATE_ERROR;
END_IF;

(*** actual position value ***)
ReadActPosAxis1.Enable := TRUE;
ActReadActualPosition();
IF ReadActPosAxis1.Valid THEN
    StatusAxis1.ActPosition := USRPOS_TO_POS(ReadActPosAxis1.Position);
ELSIF ReadActPosAxis1.Error THEN
    state := MOVESTATE_ERROR;
END_IF;

(*** actual velocity value  ***)
ReadActVelAxis1.Enable := TRUE;
ActReadActualVelocity();
IF ReadActVelAxis1.Valid THEN
    StatusAxis1.ActVelocity := USRVEL_TO_VEL(ReadActVelAxis1.Velocity);
ELSIF ReadActVelAxis1.Error THEN
    state := MOVESTATE_ERROR;
END_IF;


(*** move axis using a state machine ***)
CASE state OF
    MOVESTATE_INIT:    (* initialisation *)
        (* initialize all function blocks *)
        PowerAxis1.Enable := FALSE;
        StopAxis1.Execute := FALSE;
        Reset.Execute := FALSE;
        SetDriveRampAxis1.Execute := FALSE;
        JogAxis1.Forward := FALSE;
        JogAxis1.Backward := FALSE;
        MoveAxis1.Execute := FALSE;
        MoveVelAxis1.Execute := FALSE;
        state := MOVESTATE_ENABLE;

    MOVESTATE_ENABLE:
        PowerAxis1.Enable := TRUE;
        stHmiInt.stMCStatus.tMC_Cmd := MCCMD_POWER;
        IF PowerAxis1.Status THEN
            state := MOVESTATE_IDLE;
        ELSIF PowerAxis1.Error THEN
            stHmiInt.stMCStatus.bMC_Error := TRUE;
            state := MOVESTATE_ERROR;
        END_IF

    MOVESTATE_DISABLE:
        PowerAxis1.Enable := FALSE;
        stHmiInt.stMCStatus.tMC_Cmd := MCCMD_POWER;
        IF NOT(PowerAxis1.Status) THEN
            state := MOVESTATE_IDLE;
        ELSIF PowerAxis1.Error THEN
            stHmiInt.stMCStatus.bMC_Error := TRUE;
            state := MOVESTATE_ERROR;
        END_IF

    MOVESTATE_REF:
        RefAxis1.Position := POS_TO_USRPOS(180.0);
        RefAxis1.Mode := FALSE;
        RefAxis1.Execute := TRUE;
        stHmiInt.stMCStatus.tMC_Cmd := MCCMD_REFPOS;
        IF RefAxis1.Done THEN
            RefAxis1.Execute := FALSE;
            state := MOVESTATE_IDLE;
        ELSIF RefAxis1.Error THEN
            stHmiInt.stMCStatus.bMC_Error := TRUE;
            state := MOVESTATE_ERROR;
        END_IF


    MOVESTATE_HALT:
        JogAxis1.Forward := FALSE;
        JogAxis1.Backward := FALSE;
        MoveAxis1.Execute := FALSE;
        MoveVelAxis1.Execute := FALSE;
        StopAxis1.Execute := TRUE;
        stHmiInt.stMCStatus.tMC_Cmd := MCCMD_HALT;
        IF StopAxis1.Done THEN
            StopAxis1.Execute := FALSE;
            state := MOVESTATE_IDLE;
        ELSIF StopAxis1.Error THEN
            stHmiInt.stMCStatus.bMC_Error := TRUE;
            state := MOVESTATE_ERROR;
        END_IF


    MOVESTATE_IDLE:
        JogAxis1.Forward := FALSE;
        JogAxis1.Backward := FALSE;
        MoveAxis1.Execute := FALSE;
        IF stHmiInt.stMCStatus.bMC_Error OR StatusAxis1.Error THEN
            state := MOVESTATE_ERROR;
        END_IF
        IF NOT(StatusAxis1.Error OR StatusAxis1.Disabled) THEN
           (*Axis enabled &amp; no fault condition -> normal operation    *)
            IF Sequence.bAutoRelease THEN                    (*Automatic operation    *)
                IF Sequence.eState = SEQSTATE_MOVE THEN
                    bAxisInPosition := TRUE;
                    state := MOVESTATE_SETDRIVERAMP;
                END_IF
            ELSE                                             (*Manual operation    *)
                IF stHmiInt.stReq.stMan.bJogFwd THEN
                    state := MOVESTATE_JOGCWSLOW;
                END_IF

                IF stHmiInt.stReq.stMan.bJogFFwd THEN
```

```
                state := MOVESTATE_JOGCWFAST;
            END_IF

            IF stHmiInt.stReq.stMan.bJogRev THEN
                state := MOVESTATE_JOGCCWSLOW;
            END_IF

            IF stHmiInt.stReq.stMan.bJogFRev THEN
                state := MOVESTATE_JOGCCWFAST;
            END_IF

            IF stHmiInt.stReq.stMan.bStartVel THEN
                stHmiInt.stReq.stMan.bStartVel := FALSE;
                state := MOVESTATE_VELOCITY;
            END_IF

            IF stHmiInt.stReq.stMan.bIncrVel THEN
                stHmiInt.stReq.stMan.bIncrVel := FALSE;
                stHmiInt.rStartVel := stHmiInt.rStartVel + stHmiInt.rIncrement;
                IF NOT(StatusAxis1.StandStill) THEN
                    state := MOVESTATE_VELOCITY;
                END_IF
            END_IF

            IF stHmiInt.stReq.stMan.bDecrVel THEN
                stHmiInt.stReq.stMan.bDecrVel := FALSE;
                stHmiInt.rStartVel := stHmiInt.rStartVel - stHmiInt.rIncrement;
                IF NOT(StatusAxis1.StandStill) THEN
                    state := MOVESTATE_VELOCITY;
                END_IF
            END_IF

            IF stHmiInt.stReq.stMan.bDisable THEN
                (* Disable axis to be able to rotate it manualy to its reference point    *)
                stHmiInt.stReq.stMan.bDisable := FALSE;
                IF StatusAxis1.StandStill THEN
                    state := MOVESTATE_DISABLE;
                END_IF
            END_IF
        END_IF    (* Every operation mode    *)
        IF stHmiInt.stReq.stMan.bStop THEN
            Mode :=MODE_MANUAL;
            state := MOVESTATE_HALT;
        END_IF
    ELSE    (*** Axis disabled or axis fault condition present ***)
        IF StatusAxis1.Disabled THEN  (*** Axis disabled ***)
            IF stHmiInt.stReq.stMan.bDisable THEN  (*** Enable axis for normal operation ***)
                stHmiInt.stReq.stMan.bDisable := FALSE;
                state := MOVESTATE_ENABLE;
            END_IF
            IF stHmiInt.stReq.stMan.bHome THEN (*** Set axis actual position to 0° ***)
                state := MOVESTATE_REF;
            END_IF
        END_IF
    END_IF

MOVESTATE_JOGCWSLOW:
    (*** Jog Axis in CW Direction ***)
    JogAxis1.Forward := TRUE;
    JogAxis1.Backward := FALSE;
    JogAxis1.Fast := FALSE;
    stHmiInt.stMCStatus.tMC_Cmd := MCCMD_MOVEJOG;
    IF JogAxis1.Busy THEN
        IF NOT(stHmiInt.stReq.stMan.bJogFwd) THEN
            JogAxis1.Forward := FALSE;
            state := MOVESTATE_IDLE;
        END_IF
    END_IF
    IF JogAxis1.CommandAborted OR JogAxis1.Error THEN
        stHmiInt.stMCStatus.bMC_Error := TRUE;
        state := MOVESTATE_ERROR;
    END_IF

MOVESTATE_JOGCWFAST:
    (*** Jog Axis in CW Direction  ***)
    JogAxis1.Forward := TRUE;
    JogAxis1.Backward := FALSE;
    JogAxis1.Fast := TRUE;
    stHmiInt.stMCStatus.tMC_Cmd := MCCMD_MOVEJOG;
    IF JogAxis1.Busy THEN
        IF NOT(stHmiInt.stReq.stMan.bJogFFwd) THEN
            JogAxis1.Forward := FALSE;
            state := MOVESTATE_IDLE;
        END_IF
    END_IF
    IF JogAxis1.CommandAborted OR JogAxis1.Error THEN
        stHmiInt.stMCStatus.bMC_Error := TRUE;
        state := MOVESTATE_ERROR;
    END_IF

MOVESTATE_JOGCCWSLOW:
    (*** Jog Axis in CCW Direction ***)
    JogAxis1.Forward := FALSE;
    JogAxis1.Backward := TRUE;
    JogAxis1.Fast := FALSE;
    stHmiInt.stMCStatus.tMC_Cmd := MCCMD_MOVEJOG;
    IF JogAxis1.Busy THEN
        IF NOT(stHmiInt.stReq.stMan.bJogRev) THEN
            JogAxis1.Backward := FALSE;
            state := MOVESTATE_IDLE;
        END_IF
    END_IF
    IF JogAxis1.CommandAborted OR JogAxis1.Error THEN
        stHmiInt.stMCStatus.bMC_Error := TRUE;
        state := MOVESTATE_ERROR;
    END_IF

MOVESTATE_JOGCCWFAST:
    (* Jog Axis in CW Direction                  *)
    JogAxis1.Forward := FALSE;
    JogAxis1.Backward := TRUE;
    JogAxis1.Fast := TRUE;
    stHmiInt.stMCStatus.tMC_Cmd := MCCMD_MOVEJOG;
    IF JogAxis1.Busy THEN
```

```
            IF NOT(stHmiInt.stReq.stMan.bJogFRev) THEN
                JogAxis1.Backward := FALSE;
                state := MOVESTATE_IDLE;
            END_IF
        END_IF
        IF JogAxis1.CommandAborted OR JogAxis1.Error THEN
            stHmiInt.stMCStatus.bMC_Error := TRUE;
            stHmiInt.stMCStatus.tMC_Cmd := MCCMD_MOVEJOG;
            state := MOVESTATE_ERROR;
        END_IF

    MOVESTATE_SETDRIVERAMP:
        IF NOT(SetDriveRampAxis1.Execute) THEN
            IF (SetDriveRampAxis1.Acceleration <>
                     ACC_TO_USRACC(aSeqParams[Sequence.iActStep].rAccel)) OR
                (SetDriveRampAxis1.Deceleration <>
                     ACC_TO_USRACC(aSeqParams[Sequence.iActStep].rDeccel )) THEN
                SetDriveRampAxis1.Acceleration :=
                     REAL_TO_UDINT(aSeqParams[Sequence.iActStep].rAccel);
                SetDriveRampAxis1.Deceleration :=
                     REAL_TO_UDINT(aSeqParams[Sequence.iActStep].rDeccel );
                SetDriveRampAxis1.Execute := TRUE;
            ELSE
                state := MOVESTATE_ABSOLUTE;
            END_IF
        ELSE
            IF SetDriveRampAxis1.Busy THEN
                SetDriveRampAxis1.Execute := FALSE;
            ELSIF SetDriveRampAxis1.Done THEN
                SetDriveRampAxis1.Execute := FALSE;
                state := MOVESTATE_ABSOLUTE;
            ELSIF SetDriveRampAxis1.Error THEN
                stHmiInt.stMCStatus.bMC_Error := TRUE;
                state := MOVESTATE_ERROR;
            END_IF
        END_IF

    MOVESTATE_ABSOLUTE :              (*** start to first position ***)
        bAxisInPosition := FALSE;
        (* Calculate Setpoints *)
        StatusAxis1.DeltaPosition := aSeqParams[Sequence.iActStep].rPosition
                                          - StatusAxis1.ActPosition;
        IF (aSeqParams[Sequence.iActStep].rPosition < 0) AND
            (* rotate CCW *)
            (ABS(aSeqParams[Sequence.iActStep].rPosition) > StatusAxis1.ActPosition)
            (* Cross 0° / 360° *)
        THEN
            StatusAxis1.DeltaPosition := -1 * (360 +
                  aSeqParams[Sequence.iActStep].rPosition + StatusAxis1.ActPosition);
        ELSIF (aSeqParams[Sequence.iActStep].rPosition > 0) AND
                (* rotate CW *)
            (aSeqParams[Sequence.iActStep].rPosition < StatusAxis1.ActPosition)
                (* Cross 0° / 360° *)
        THEN
            StatusAxis1.DeltaPosition := ((360 - StatusAxis1.ActPosition)
                  + aSeqParams[Sequence.iActStep].rPosition);
        ELSE
            StatusAxis1.DeltaPosition := ABS(aSeqParams[Sequence.iActStep].rPosition)
                                                  - StatusAxis1.ActPosition;
        END_IF

        (*** Setup axis command parameters ***)
        MoveAxis1.Distance := POS_TO_USRPOS(StatusAxis1.DeltaPosition);
        //.... leak is here:
        MoveAxis1.Velocity := VEL_TO_USRVEL(aSeqParams[Sequence.iActStep].rVelocity);
        MoveAxis1.Execute := TRUE;
        stHmiInt.stMCStatus.tMC_Cmd := MCCMD_MODMOVE;

        IF MoveAxis1.Busy OR
            (MoveAxis1.Distance = rLastPositionCmd) THEN
            MoveAxis1.Execute := FALSE;
            // Fix: MoveAxis1.Velocity := 0;
            state := MOVESTATE_MOVEACTIVE;
        END_IF
        IF MoveAxis1.CommandAborted OR MoveAxis1.Error THEN
            stHmiInt.stMCStatus.bMC_Error := TRUE;
            state := MOVESTATE_ERROR;
        END_IF
        IF NOT(Sequence.bAutoRelease OR Sequence.eState = SEQSTATE_MOVE) THEN
            (*** stop all active commands ***)
            state := MOVESTATE_HALT;
        END_IF

    MOVESTATE_VELOCITY:
        IF stHmiInt.rStartVel = 0 THEN
            state := MOVESTATE_HALT;
        ELSE
            MoveVelAxis1.Velocity := VEL_TO_USRVEL(stHmiInt.rStartVel);
            MoveVelAxis1.Execute := TRUE;
        END_IF
        stHmiInt.stMCStatus.tMC_Cmd := MCCMD_MODMOVE;
        IF MoveVelAxis1.InVelocity THEN
            MoveVelAxis1.Execute := FALSE;
            state := MOVESTATE_IDLE;
        ELSIF MoveVelAxis1.CommandAborted OR MoveVelAxis1.Error THEN
            stHmiInt.stMCStatus.bMC_Error := TRUE;
            state := MOVESTATE_ERROR;
        END_IF

    MOVESTATE_MOVEACTIVE:
        rLastPositionCmd := MoveAxis1.Distance;
        bNotMoving := (REAL_TO_INT(StatusAxis1.ActPosition * 1000) / 1000) = iScratch;
        tTimeout(IN := bNotMoving , PT := t#1000ms);
        iScratch := REAL_TO_INT(StatusAxis1.ActPosition * 1000) / 1000;
        IF NOT(Sequence.bAutoRelease OR Sequence.eState = SEQSTATE_MOVE
           OR stHmiInt.stReq.stMan.bStop) THEN
            (*** stop all active commands ***)
            state := MOVESTATE_HALT;
        END_IF
        IF bAxisInPosition THEN
            state := MOVESTATE_IDLE;
        ELSIF tTimeout.Q THEN
            state := MOVESTATE_ABSOLUTE;
        END_IF
```

```
    MOVESTATE_ERROR :
        IF StatusAxis1.Error OR stHmiInt.stMCStatus.bMC_Error THEN
            state := MOVESTATE_RESET;    (*** axis error requires reset ***)
        ELSE
            state := MOVESTATE_INIT;     (*** function block errors don't need a reset ***)
        END_IF

    MOVESTATE_RESET :
        Reset.Execute := stHmiInt.stReq.bReset;
        stHmiInt.stMCStatus.tMC_Cmd := MCCMD_RESET;
        IF Reset.Done THEN
            VSObj_McFaultDescription.stTextDisplay := 'Keine Fehler';
            stHmiInt.stMCStatus.bMC_Error := FALSE;
            state := MOVESTATE_INIT;
        ELSIF Reset.Error THEN
            stHmiInt.stMCStatus.bMC_Error := TRUE;
            state := MOVESTATE_INIT; (*** can't do anything here ***)
        END_IF

END_CASE


IF state = MOVESTATE_MOVEACTIVE AND StatusAxis1.StandStill THEN
    IF (stHmiInt.stStepData.rDeltaPos > -0.5) AND
    (stHmiInt.stStepData.rDeltaPos < 0.5) THEN
        bAxisInPosition := TRUE;
    END_IF
ELSE
    bAxisInPosition := FALSE;
END_IF

dScratch := ACC_TO_USRACC(rScratch);

ActSetDriveRamp() ;   (* call the set drive ramp function block *)
ActPower()        ;   (* call the power function block *)
ActSetPosition()  ;   (* call the set position function block *)
ActStop()         ;   (* call the halt function block *)
ActMoveJog()      ;   (* call the jog function block *)

// leaked value is used here:
ActMove()         ;   (* call the move function block *)

ActMoveVel()      ;   (* call the move function block *)
ActReset()        ;   (* call the reset function block *)


//fix: MoveAxis1.Velocity := 0;

END_PROGRAM
```