Arcade.PLC: A Verification Platform for Programmable Logic Controllers

Sebastian Biallas
Embedded Software Laboratory
RWTH Aachen University
Aachen, Germany
biallas@embedded.rwth-aachen.de

Jörg Brauer
Verified Systems International GmbH
Bremen, Germany
brauer@verified.de

Stefan Kowalewski
Embedded Software Laboratory
RWTH Aachen University
Aachen, Germany
kowalewski@embedded.rwth-aachen.de

ABSTRACT

This paper introduces Arcade.PLC, a verification platform for programmable logic controllers (PLCs). The tool supports static analysis as well as $\forall$CTL and past-time LTL model checking using counterexample-guided abstraction refinement for different programming languages used in industry. In the underlying principles of the framework, knowledge about the hardware platform is exploited so as to provide efficient techniques. The effectiveness of the approach is evaluated on programs implemented using a combination of programming languages.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*Formal methods, Model checking*

General Terms

Verification

Keywords

PLC, static analysis, model checking

1. INTRODUCTION

PLCs [7] are control devices mostly used in the automation industry to operate and monitor systems such as power plants and oil rigs. Since failures in such systems may have hazardous effects on humans or the environment, the application of formal methods to ensure correctness is highly recommended [8]. Yet, their application remains difficult as five different programming languages have been standardized, which can be combined in programs. A PLC typically operates in the *cyclic scanning mode*, which consists of three phases, each of which is executed atomically: (1) sensing inputs, (2) executing the program, and (3) writing outputs. These particularities necessitate techniques tailored to PLCs.

1.1 Arcade.PLC

This paper presents the Arcade.PLC[1] model checker for PLCs, which supports specifications in $\forall$CTL and past-time LTL (ptLTL). In principle, Arcade.PLC implements the counterexample-guided abstraction refinement (CEGAR) scheme. Yet, we have developed different adaptations to account for the hardware platform. To break the dependencies between the different implementation languages, we first translate the programs, which consist of a collection of modules, into an intermediate representation (IR). The tool then accounts for the cyclic scanning mode by hiding intermediate states that appear within a cycle, and are thus invisible to the environment. The end-user can specify observable input-output relations, which dovetails with typical specifications for PLCs. To summarize, Arcade.PLC offers the following advantages compared to other model checkers:

- PLC programs are supported natively, i. e., neither manual program transformation nor preprocessing is necessary.
- It is possible to verify programs composed of modules written in different languages, which is typical for real-world PLC implementations.
- The cyclic operation mode and the atomicity of the phases are exploited by using a symbolic and a non-symbolic abstraction step.

1.2 Related Work

The first work for the formal verification of PLC programs goes back to Moon [9], who translated PLC programs written in *ladder diagram* into the input language of the model checker Svm. Similar in spirit are the works of Rausch et al. [12], Canet et al. [4] and Pavlovic et al. [10]. Later, Gourcuff et al. explored the verification of *structured text* [5] and abstractions [6]. These approaches have in common that they required a transformation are limited to a subset of languages or language constructs.

To the best of our knowledge, Arcade.PLC is the first tool to combine fully automatic verification, efficient abstraction techniques, support for different PLC programming languages and a graphical user interface. It emerged from [mc]square [13], a model checker for microcontroller software.

[1]Aachen Rigorous Code Analysis and Debugging Environment for PLCs
`http://arcade.embedded.rwth-aachen.de`

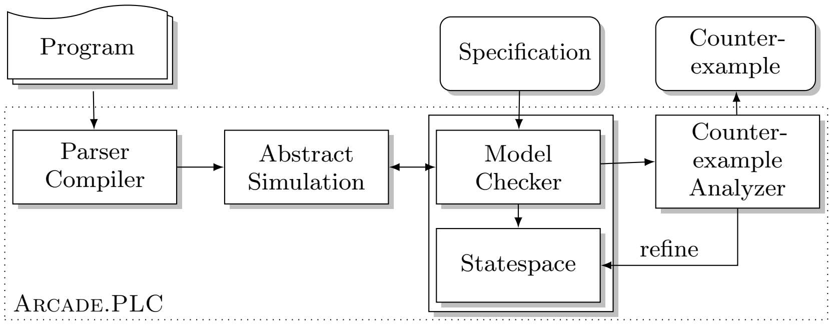


Figure 1: Model checking process with Arcade.PLC

2. OVERVIEW AND IMPLEMENTATION

PLC Programs are composed of modules called *program organization units* [7] (POU). A *function block* (FB) is a POU with a fixed interface, which comprises variables used for input and for output and an implementation that is executed upon a call. Variables can retain their value for the next execution to maintain an internal state between calls (henceforth called *global variables*). The actual implementation can be provided in different languages. These languages include graphical representations resembling electric circuits, automaton representations similar to Petri nets, assembler dialects and high-level imperative languages [7, Part 3]. Some vendors use proprietary languages or extensions. The other types of POUs are *functions*, which are FBs equipped with an explicit return value but no internal state, and *programs*, which are FBs to be used as the main program.

2.1 Features

Arcade.PLC can verify all kinds of POUs, i. e., either entire programs or just parts of it can be selected for analysis. Its overall structure is shown in Fig. 1. The user can supply a program as (one or many) text files containing different POUs. For now, we support programs written in the standardized languages *structured text* (ST) and *instruction list* (IL) and the proprietary *statement list* (STL) used by Siemens SIMATIC S7. Programs can use an implementation of standard library FBs (counters, edge detection, timers, etc.) written in ST. To handle FBs in different languages uniformly, and also to simplify abstract simulation, the programs are first compiled into an IR. This also ensures that our abstract simulator operates on a well-defined semantics without having to struggle with semantic nuances such as undefined or implementation-dependent behavior and vendor-specific extensions, which are thus hidden in the front-end.

For verification, the user can specify formulae in $\forall\mathsf{CTL}$ and ptLTL. In these formulae, it is possible to express propositions about the values of all non-temporary program variables, which are then evaluated at the end of each execution of a cycle.

2.2 State Space

Model checking is performed using an on-the-fly algorithm, which starts with a coarse abstraction that is successively refined [2]. Let V_I, V_O and V_G denote the sets of input, output and global variables, respectively. Our formal model comprises explicit states of the form $\langle I, G, O\rangle$, where I, G, O are assignments of all variables in V_I, V_G, V_O to a value in their domain. A transition between states $\langle I_1, G_1, O_1\rangle$ and $\langle I_2, G_2, O_2\rangle$ is assumed if the program outputs O_2 with internal state G_2 after executing one cycle on the internal state G_1 with inputs I_2. Boolean and integer variables are allowed to contain abstract values, taken from the reduced product of the interval and bit-set domains (each bit can either be $\{0\}$, $\{1\}$ or $\{0, 1\}$). By starting with the initial state and iteratively creating successors using abstract simulation until a fixed point is reached, we obtain the reachable state space for model checking. Observe that we do not need to store inputs I, which are not referred to in the formula, in the state space, thereby giving a more space-efficient representation.

2.3 Control Flow Determinization

To find a suitable abstraction, we first determinize the control flow. This is done incrementally by refining the abstraction of each variable, until they dictate a single trace through the program. To do so, we first guard all conditional jumps with predicates. Then, for each unvisited state $s = \langle I, G, O\rangle$, we simulate it with unknown inputs, i. e., $s' = \langle I^{\top}, G, O\rangle$ where $I^{\top} = \{v \to \top \mid v \in V_I\}$. We simulate the program until either the cycle terminates, which amounts to finding a successor, or until an abstract value is ambiguous w. r. t. one of the jump predicates. In the latter case, we use predicate transformer semantics to find the weakest precondition that makes the jump predicate unambiguous. This backward transformation is based on symbolic expressions which we generate for each operation in the trace. The result gives rise to a predicate on an input or global variable which is refined so that abstract simulation on the refined values satisfies the jump predicate. Two specifics aid in making this efficiently possible: (a) We are always dealing with a single path through the program and (b) PLC programs have to obey the cycle frequency, i. e., they have to produce a result before the start of the next cycle and thus consist of a small number of operations per cycle. Likewise, refinement is applied if the results at the end of a cycle is ambiguous w. r. t. atomic propositions of the specification.

Example 1. Consider a conditional jump that branches iff $a > 100$, where a is a register that contains the interval $[30, 140]$ and the symbolic constraint $a = (i + 20)$ for some input i. Then, $wp(a = i + 20, a > 100) = (i > 80)$, which entails that assigning $[10, 79]$ and $[80, 120]$ to i determinizes the control flow. We thus discard the result of the previous simulation and restart the cycle with inputs $[10, 79]$ and $[80, 120]$ for i.

2.4 CEGAR with Global Variables

We do not store symbolic expressions in the state space, e. g., for global variables x, y we do not know whether $x < y$ if their intervals overlap. Thus, refining a global variable may introduce additional behavior, i. e., transitions infeasible in the concrete model. To cope with this, we use a CEGAR approach: If a universally quantified formula is valid in the abstract state space, it is also valid in the concrete model; otherwise, we generate a counterexample that can potentially be spurious. This is checked by replaying the counterexample, i. e., repeating the successor-building and following the states comprising the counterexample. If, during replay, there is an ambiguous jump or atomic proposition in the trace which depends on global variables, we store the respective predicates as so-called *lemmas*. Then, we guard the program with all lemmas, guiding the refinement of the state space similar to control flow determinization. The idea of this approach is that the lemmas are checked at the end of a cycle, where the model checker is still aware of symbolic/relational information (which is not stored in the state space). Restarting model checking until either the formula is valid or we do not find further lemmas thus eventually resolves over-approximate behavior that leads to spurious counterexamples.

If a counterexample is legitimate, i. e., all transitions are possible without relying on further lemmas, it is presented to the user. All transitions are labeled with input values necessary for reaching the destination. Due to abstraction refinement from above, they typically are maximal in the sense that increasing the set of possible values of a variable would make control flow ambiguous. Thus, the counterexample shows the complete set of failure inducing input. For clarity, inputs containing $\top$ (i. e., irrelevant inputs) are omitted from display. This further aids in debugging erroneous behavior by drawing attention to the relevant inputs.

2.5 Hierarchical Predicate Abstraction

In addition, we have developed a predicate abstraction to further reduce the state space. To illustrate the idea, consider a state s where $x = [0, 50]$ and the formula holds. Then, a new state s' identical to s but with, e.g., $x = [7, 23]$, is entailed by s. Thus, s' does not have to be inspected. The key idea to achieve this is to organize the state space in a form that supports efficient entailment checks while still having explicit (though abstract) states. We do so by using predicates which are represented in a tree-structure. Each level in the tree partitions the state space (and so do sub-levels, but the predicates in the different levels differ). Entailment checking then amounts to traversing this tree. New predicates are automatically derived from atomic proposition taken from the formula and by analyzing lemmas necessary for counterexamples. In principle the overall idea follows the spirit of OBDD-based approaches, with the exception that we use the leaves of our structure to store the transition relation. As of now, the predicate abstraction is only implemented for checking invariants.

2.6 Static Analysis

Finally, ARCADE.PLC features a static analysis framework, which also operates on our IR and uses the abstract interpretation to: (a) infer range and value-set information about the program variables, (b) perform liveness analysis and (c) perform slicing depending on the formula to be checked. For (a), we use basically the same algorithm for the successor generation as the model checker, but rely on the value-set domain to capture more precise information. Instead of building the reachable state space, however, we join the abstract values of all variables for each possible successors state after simulating a cycle. This ensures quick convergence, while still generating valuable range information.

3. EXPERIMENTS

This section presents an evaluation of ARCADE.PLC on programs of varying complexity written in different languages. All experiments where performed on a desktop computer equipped with an Intel Core i5 processor and 16 GiB RAM. ARCADE.PLC itself has been implemented in JAVA.

3.1 Benchmarks

The first collection of five programs consists of safety-critical FBs defined by the PLCopen consortium [11] and constitute up to 14 inputs of type Boolean and Integer. They are specified in terms of automata, timing diagrams and semi-formal descriptions, while the implementation of the FBs itself is left to the developer. All FBs, irrespective of their implementation language, use the standard library in ST. The specifications that we checked formalize correctness requirements from the specifications. The second case study examines programs for controlling conveyor belts using a Siemens SIMATIC S7 PLC. The conveyor belts all operate independently and are controlled with a merely Boolean program using light curtains. In the third case study, we verified a controller for a 3D robot, also using a SIMATIC S7. This robot has 3 motors that allow to move its arm in all dimensions, and one motor for its mechanical grab. All motor axes are connected to step counters, which allow the program to count the number of rotations of the respective motors. Since unlimited forth/back and up/down movement is mechanically impossible (and would ultimately destroy the motor), we verified that the counters remain within range. This case study was performed with increasing complexity by raising the number on controlled axes. In one case, we altered the formula slightly to provoke a counterexample.

3.2 Evaluation

The experimental results using both techniques discussed in this paper are given in Tab. 1. We chose a time-out of 10 minutes for all programs (indicated by ∞). The table presents the number of states and runtime using both the CEGAR and the predicate abstraction (PA) approach (checking ptLTL formulae using predicate abstraction is not possible yet, hence the *n/a* in row 4).

Small function blocks as they are used for safety-critical functions are checked with ARCADE.PLC in minutes or even seconds. When combining them to larger programs, the time for model checking increases but it is possible to check all programs in reasonable time. It is interesting to observe that CEGAR and hierarchical predicate abstraction can be seen as orthogonal techniques. In any except one case, hierarchical predicate abstraction leads to more compact state spaces using the tree-like state space representation. Depending on the structure of the program, the runtime may increase (cp. Robot), but in other cases it improves performance by a factor of at least 30.

Table 1: Evaluation with Arcade.PLC

	Program	Lang.	#loc	Spec.	res.	#states CEGAR	time CEGAR	#states PA	time PA
PLCOpen	`SF_Antivalent`	ST	108	∀CTL	*true*	45	< 1s	5	< 1s
	`SF_EmergencyStop`	IL&ST	226	∀CTL	*CE*	12	< 1s	46	< 1s
	`SF_ModeSelector`	ST	188	∀CTL	*true*	743	15s	17	< 1s
	`SF_ModeSelector`	ST	188	ptLTL	*true*	710	14s	n/a	n/a
	`SF_ModeSelector`	IL&ST	424	∀CTL	*true*	-	∞	45	50s
	`SF_GuardLocking`	IL&ST	400	∀CTL	*true*	39,265	45s	3	< 1s
	`SF_MutingSeq`	IL&ST	827	∀CTL	*true*	-	∞	3	151s
Belt	1 Belt	S7	92	∀CTL	*true*	128	1s	3	< 1s
	4 Belts	S7	323	∀CTL	*true*	137	7s	3	< 1s
Robot	1 Axis	S7	66	∀CTL	*true*	178	< 1s	85	23s
	4 Axes	S7	102	∀CTL	*true*	314	4s	85	157s
	4 Axes	S7	102	∀CTL	*CE*	178	< 1s	86	132s

4. CONCLUSION

In Arcade.PLC, we have implemented different seemingly orthogonal approaches. On the one hand, we account for the specific hardware platform and its cyclic scanning mode. On the other hand, we made our framework independent from implementation languages by translating them into an IR. Our work thus dovetails with recent work that uses IRs for binaries [1, 3], which aims at providing a generic interface for low-level analyses. Indeed, Arcade.PLC is the first verification tool for PLCs that can handle modules written in different languages. Using domain-specific variants of CEGAR and predicate abstraction, it can verify programs that involve complex control flow and heavy interaction with the environment.

5. ACKNOWLEDGMENTS

Sebastian Biallas is supported by the DFG. Further, this work is supported by the DFG Research Training Group 1298 *Algorithmic Synthesis of Reactive and Discrete-Continuous Systems* (AlgoSyn) and by the DFG Cluster of Excellence on Ultra-high Speed Information and Communication (UMIC), German Research Foundation grant DFG EXC 89.

6. REFERENCES

[1] S. Bardin, P. Herrmann, J. Leroux, O. Ly, R. Tabary, and A. Vincent. The BINCOA Framework for Binary Code Analysis. In *CAV*, pages 165–170, 2011.

[2] S. Biallas, J. Brauer, and S. Kowalewski. Counterexample-Guided Abstraction Refinement for PLCs. In *SSV*, pages 1–9. USENIX, 2010.

[3] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz. BAP: A Binary Analysis Platform. In *CAV*, volume 6806 of *LNCS*, pages 463–469. Springer, 2011.

[4] G. Canet, S. Couffin, J.-J. Lesage, A. Petit, and P. Schnoebelen. Towards the automatic verification of PLC programs written in instruction list. In *2000 IEEE International Conference on Systems, Man, and Cybernetics, Nashville*, volume 4, pages 2449–2454. IEEE Computer Society Press, 2000.

[5] V. Gourcuff, O. De Smet, and J. M. Faure. Efficient representation for formal verification of PLC programs. In *8th International Workshop on Discrete Event Systems*, pages 182–187, 2006.

[6] V. Gourcuff, O. De Smet, and J.-M. Faure. Improving large-sized PLC programs verification using abstractions. In *Proceedings of the 17th IFAC World Congress*, pages 5101–5106, 2008.

[7] International Electrotechnical Commission. *IEC 61131-3: Programmable Controllers — Part 3 Programming languages.* International Electrotechnical Commission, Geneva, Switzerland, 1993.

[8] International Electrotechnical Commission. *IEC 61508: Functional Safety of Electrical, Electronic and Programmable Electronic Safety-Related Systems.* International Electrotechnical Commission, Geneva, Switzerland, 1998.

[9] I. Moon. Modeling programmable logic controllers for logic verification. *IEEE Control Systems Magazine*, 14(2):53–59, 1994.

[10] O. Pavlovic, R. Pinger, and M. Kollmann. Automated formal verification of PLC programms written in IL. In *VERIFY*, number 259 in Workshop Proce., pages 152–163. CEUR-WS.org, 2007.

[11] PLCopen TC5. *Safety Software Technical Specification, Version 1.0, Part 1: Concepts and Function Blocks.* PLCopen, Germany, 2006.

[12] M. Rausch and B. Krogh. Formal verification of PLC programs. In *In Proc. American Control Conference*, pages 234–238, 1998.

[13] B. Schlich and S. Kowalewski. Model checking C source code for embedded systems. *International Journal on Software Tools for Technology Transfer (STTT)*, 11(3):187–202, 2009.

APPENDIX

A. USER INTERFACE AND DEMONSTRATION

To assess ARCADE.PLC's user interface, we present some screenshots of its GUI. If accepted, this GUI will be presented in the tool demonstration session

In the first screenshot, the model checker interface is shown. Different formulae (in different logics) can be checked in parallel. The right-hand side of the window shows the input, output and global variables used in the program, together with their initial value. ARCADE.PLC also supports batch-processing, which is a useful feature when analyzing complex programs or a library of function blocks: A list of verification tasks is defined, which are executed one after another or in parallel.

If there is a counterexample or witness for a formula, PLC input assignments which yield a path violating the specification are presented graphically, as shown in Fig. 3. In the given example, the states themselves are represent with an evaluation for the atomic propositions used in the formula, whereas the transitions between the states are labeled with necessary inputs. The upper window on the right-hand side contains all variables (input, output and global) and shows their values in the selected state of the counterexample. The lower window on the right-hand side shows the truth values of all subformulae in the analyzed specification after it has been put into normal form.

Finally, the screenshot in Fig. 4 shows the built-in simulator, which can be used to debug and analyze the PLC code written in either of the supported implementation languages. In this view, ARCADE.PLC automatically determines potential successor states and provides the user with a list of transitions that can be taken next to ease simulation. This screenshot also highlights a static analysis which automatically determines possible ranges and value-sets from the program variables directly in the GUI.

B. PROGRAMS

All PLC programs we are allowed to publish are available at:
`http://arcade.embedded.rwth-aachen.de/ase12_casestudy_plc.zip`

C. SPECIFICATIONS

In our case studies, we used the following specifications:

C.1 PLCopen Safety Function Blocks

Our formulae were derived from the PLCopen specifications:

- SF_Antivalent: `S_AntivalentOut` implies that `S_ChannelNC` (*normally closed*) is set and `S_ChannelNO` (*normally open*) is not set:

$$AG\,(\texttt{S\_AntivalentOut} \implies \texttt{S\_ChannelNC} \wedge \neg\texttt{S\_ChannelNO}) \quad (1)$$

- SF_EmergencyStop:

$$AG\,(\neg\texttt{S\_EStopIn} \implies \neg\texttt{S\_EStopOut}) \quad (2)$$

- SF_ModeSelector ($\forall$CTL): In the locked state there is at most one mode selected:

$$AG\,(\texttt{DiagCode} = \texttt{ModeLocked} \implies (\texttt{S\_Mode0Sel} + \texttt{S\_Mode1Sel} + \texttt{S\_Mode2Sel} + \texttt{S\_Mode3Sel} + \texttt{S\_Mode4Sel} + \texttt{S\_Mode5Sel} + \texttt{S\_Mode6Sel} + \texttt{S\_Mode7Sel} \leq 1)) \quad (3)$$

- SF_ModeSelector (LTL): The `ModeSelected` state is only acknowledged if there was no timeout since the `ModeChanged` state:

$$\texttt{DiagCode} = \texttt{ModeSelected} \implies [\texttt{DiagCode} = \texttt{ModeChanged}, \texttt{Timer.Q}) \quad (4)$$

- SF_GuardLocking: `S_GuardLocked` can only be set in the `Ready` state.

$$AG\,(\texttt{S\_GuardLocked} \implies \texttt{Ready}) \quad (5)$$

- SF_MutingSeq: `S_MutingActive` can only be set in the `Ready` state.

$$AG\,(\texttt{S\_MutingActive} \implies \texttt{Ready}) \quad (6)$$

C.2 Conveyor Belt

To test whether the program acknowledges the (negated) motor stop signal we used the following formulae:

$$AG\,(\texttt{Beltmot1\_on} \implies \texttt{nStop}) \quad (7)$$

$$AG\,(\texttt{Beltmot1\_on001} \implies \texttt{nStop001}) \quad (8)$$

C.3 Robot

To test whether the number of rotations on axis 0 are within its upper bound, we used the formula:

$$AG\,(\texttt{C0} \leq 40) \quad (9)$$

A counterexample was found using the formula:

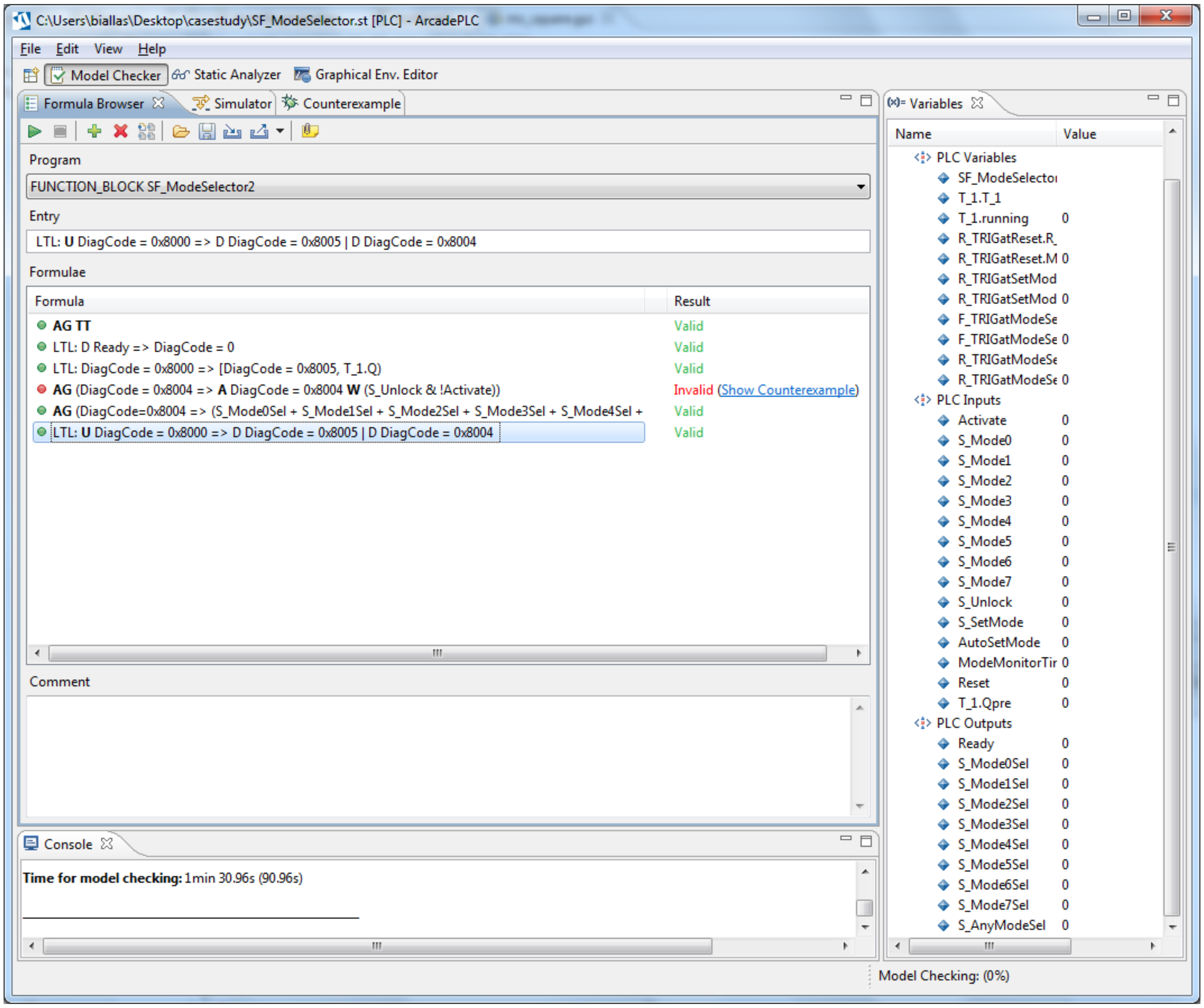
$$AG\,(\texttt{C0} < 40) \quad (10)$$

D. FURTHER STATISTICS

Table 2 presents further statistics from our case study. The column *#created CEGAR* shows the number of states which were created (but not necessarily stored since they were already existing). Likewise, *#created PA* indicates the number of created states for the predicate abstraction after the last counterexample was analyzed. In the *#analyzed CE* column the number of counterexamples, which were analyzed during the model checking, is shown.

	Program	Spec.	#created CEGAR	#created PA	#analyzed CE PA
PLCOpen	SF_Antivalent	(1)	401	144	4
	SF_EmergencyStop	(2)	121	46	49
	SF_ModeSelector	(3)	245,211	5,928	0
	SF_ModeSelector	(4)	239,987	n/a	n/a
	SF_ModeSelector	(3)	-	736,740	0
	SF_GuardLocking	(5)	954,360	1146	0
	SF_MutingSeq	(6)	-	1,137,420	0
Belt	1 Belt	(7)	1,721	33	0
	4 Belts	(8)	1,852	33	0
Robot	1 Axis	(9)	184	411	128
	4 Axes	(9)	2,951	1641	128
	4 Axes	(10)	578	454	130

Table 2: Further statistics from our case study



Program

FUNCTION_BLOCK SF_ModeSelector2

Entry

LTL: **U** DiagCode = 0x8000 => D DiagCode = 0x8005 | D DiagCode = 0x8004

Formulae

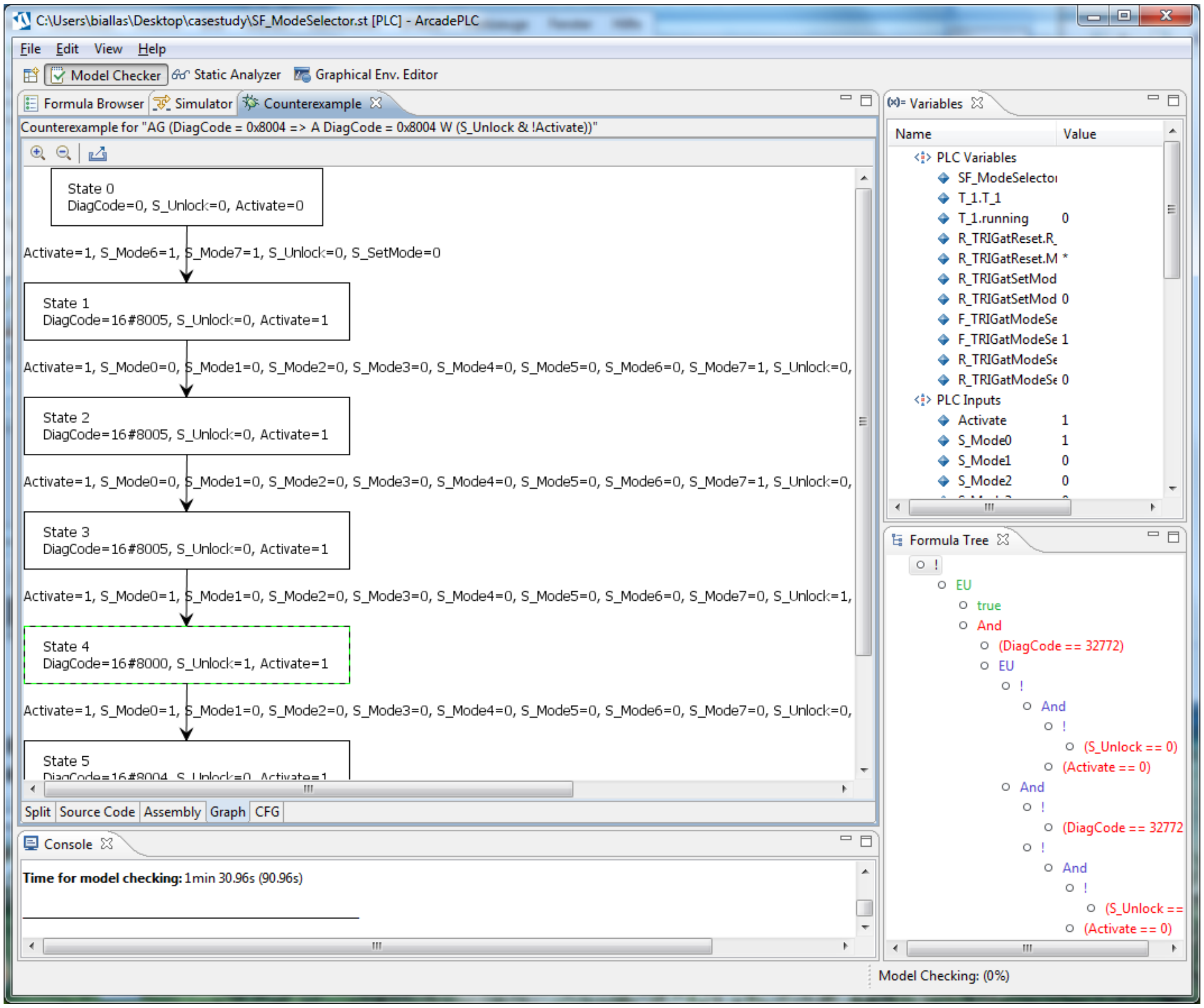
Formula	Result
AG TT	Valid
LTL: D Ready => DiagCode = 0	Valid
LTL: DiagCode = 0x8000 => [DiagCode = 0x8005, T_1.Q)	Valid
AG (DiagCode = 0x8004 => **A** DiagCode = 0x8004 **W** (S_Unlock & !Activate))	Invalid (Show Counterexample)
AG (DiagCode=0x8004 => (S_Mode0Sel + S_Mode1Sel + S_Mode2Sel + S_Mode3Sel + S_Mode4Sel +	Valid
LTL: **U** DiagCode = 0x8000 => D DiagCode = 0x8005 | D DiagCode = 0x8004	Valid

Comment

Time for model checking: 1min 30.96s (90.96s)

Name	Value
PLC Variables	
SF_ModeSelecto	
T_1.T_1	
T_1.running	0
R_TRIGatReset.R_	
R_TRIGatReset.M	0
R_TRIGatSetMod	
R_TRIGatSetMod	0
F_TRIGatModeSe	
F_TRIGatModeSe	0
R_TRIGatModeSe	
R_TRIGatModeSe	0
PLC Inputs	
Activate	0
S_Mode0	0
S_Mode1	0
S_Mode2	0
S_Mode3	0
S_Mode4	0
S_Mode5	0
S_Mode6	0
S_Mode7	0
S_Unlock	0
S_SetMode	0
AutoSetMode	0
ModeMonitorTir	0
Reset	0
T_1.Qpre	0
PLC Outputs	
Ready	0
S_Mode0Sel	0
S_Mode1Sel	0
S_Mode2Sel	0
S_Mode3Sel	0
S_Mode4Sel	0
S_Mode5Sel	0
S_Mode6Sel	0
S_Mode7Sel	0
S_AnyModeSel	0

Figure 2: The model checker interface of Arcade.PLC



C:\Users\biallas\Desktop\casestudy\SF_ModeSelector.st [PLC] - ArcadePLC

Counterexample for "AG (DiagCode = 0x8004 => A DiagCode = 0x8004 W (S_Unlock & !Activate))"

```
State 0
DiagCode=0, S_Unlock=0, Activate=0

Activate=1, S_Mode6=1, S_Mode7=1, S_Unlock=0, S_SetMode=0

State 1
DiagCode=16#8005, S_Unlock=0, Activate=1

Activate=1, S_Mode0=0, S_Mode1=0, S_Mode2=0, S_Mode3=0, S_Mode4=0, S_Mode5=0, S_Mode6=0, S_Mode7=1, S_Unlock=0,

State 2
DiagCode=16#8005, S_Unlock=0, Activate=1

Activate=1, S_Mode0=0, S_Mode1=0, S_Mode2=0, S_Mode3=0, S_Mode4=0, S_Mode5=0, S_Mode6=0, S_Mode7=1, S_Unlock=0,

State 3
DiagCode=16#8005, S_Unlock=0, Activate=1

Activate=1, S_Mode0=1, S_Mode1=0, S_Mode2=0, S_Mode3=0, S_Mode4=0, S_Mode5=0, S_Mode6=0, S_Mode7=0, S_Unlock=1,

State 4
DiagCode=16#8000, S_Unlock=1, Activate=1

Activate=1, S_Mode0=1, S_Mode1=0, S_Mode2=0, S_Mode3=0, S_Mode4=0, S_Mode5=0, S_Mode6=0, S_Mode7=0, S_Unlock=0,

State 5
DiagCode=16#8004, S_Unlock=0, Activate=1
```

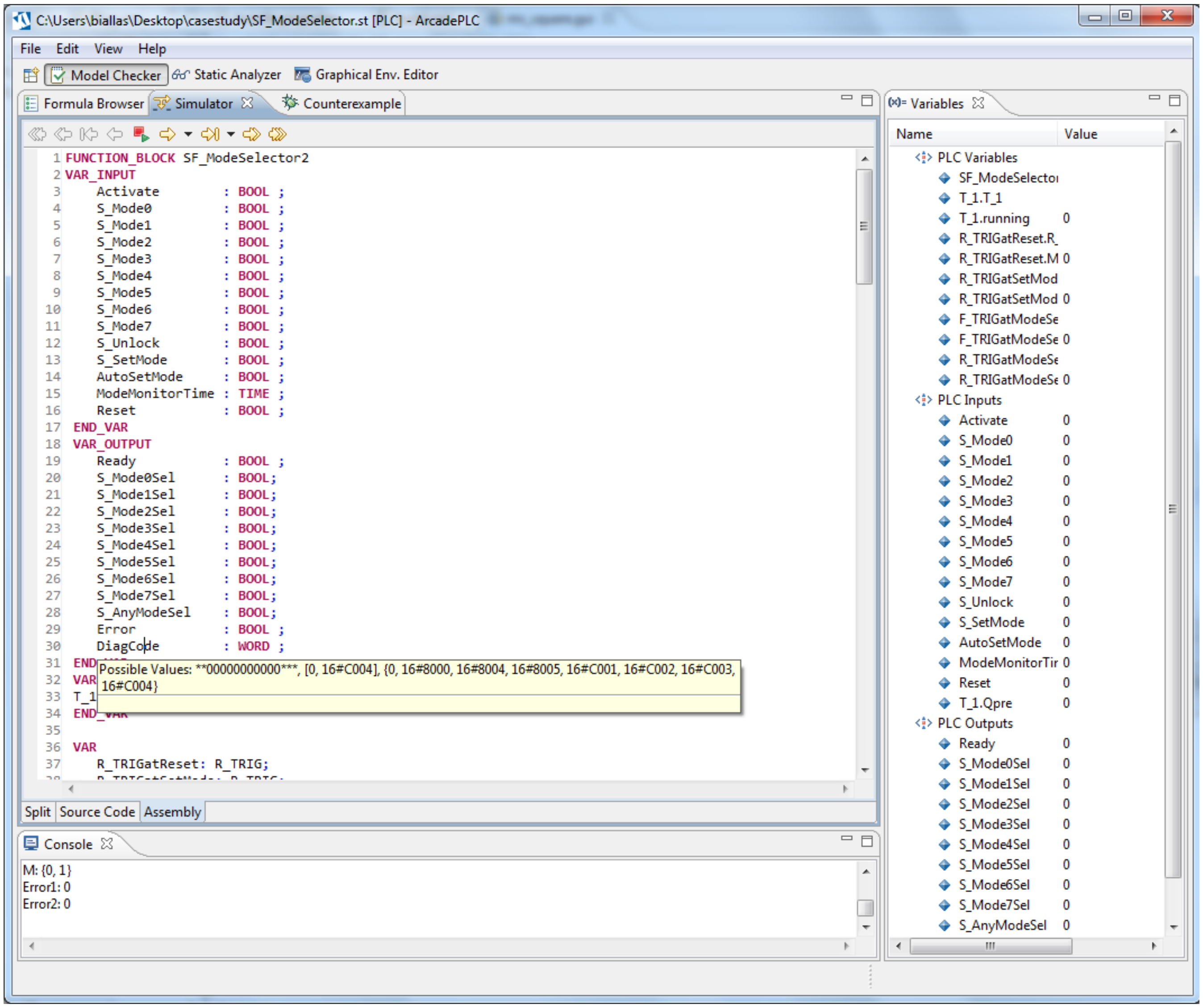
Name	Value
PLC Variables	
SF_ModeSelector	
T_1.T_1	
T_1.running	0
R_TRIGatReset.R_	
R_TRIGatReset.M	*
R_TRIGatSetMod	
R_TRIGatSetMod	0
F_TRIGatModeSe	
F_TRIGatModeSe	1
R_TRIGatModeSe	
R_TRIGatModeSe	0
PLC Inputs	
Activate	1
S_Mode0	1
S_Mode1	0
S_Mode2	0

```
!
  EU
    true
    And
      (DiagCode == 32772)
      EU
        !
          And
            !
              (S_Unlock == 0)
            (Activate == 0)
        And
          !
            (DiagCode == 32772
          !
            And
              !
                (S_Unlock ==
              (Activate == 0)
```

Time for model checking: 1min 30.96s (90.96s)

Model Checking: (0%)

Figure 3: Representation of a counterexample trace



```
FUNCTION_BLOCK SF_ModeSelector2
VAR_INPUT
    Activate        : BOOL ;
    S_Mode0         : BOOL ;
    S_Mode1         : BOOL ;
    S_Mode2         : BOOL ;
    S_Mode3         : BOOL ;
    S_Mode4         : BOOL ;
    S_Mode5         : BOOL ;
    S_Mode6         : BOOL ;
    S_Mode7         : BOOL ;
    S_Unlock        : BOOL ;
    S_SetMode       : BOOL ;
    AutoSetMode     : BOOL ;
    ModeMonitorTime : TIME ;
    Reset           : BOOL ;
 END_VAR
 VAR_OUTPUT
    Ready           : BOOL ;
    S_Mode0Sel      : BOOL;
    S_Mode1Sel      : BOOL;
    S_Mode2Sel      : BOOL;
    S_Mode3Sel      : BOOL;
    S_Mode4Sel      : BOOL;
    S_Mode5Sel      : BOOL;
    S_Mode6Sel      : BOOL;
    S_Mode7Sel      : BOOL;
    S_AnyModeSel    : BOOL;
    Error           : BOOL ;
    DiagCode        : WORD ;
 END
 VAR
 T_1
 END

 VAR
    R_TRIGatReset: R_TRIG;
```

Possible Values: **00000000000***, [0, 16#C004], {0, 16#8000, 16#8004, 16#8005, 16#C001, 16#C002, 16#C003, 16#C004}

```
M: {0, 1}
Error1: 0
Error2: 0
```

Name	Value
PLC Variables	
SF_ModeSelector	
T_1.T_1	
T_1.running	0
R_TRIGatReset.R_	
R_TRIGatReset.M	0
R_TRIGatSetMod	
R_TRIGatSetMod	0
F_TRIGatModeSe	
F_TRIGatModeSe	0
R_TRIGatModeSe	
R_TRIGatModeSe	0
PLC Inputs	
Activate	0
S_Mode0	0
S_Mode1	0
S_Mode2	0
S_Mode3	0
S_Mode4	0
S_Mode5	0
S_Mode6	0
S_Mode7	0
S_Unlock	0
S_SetMode	0
AutoSetMode	0
ModeMonitorTir	0
Reset	0
T_1.Qpre	0
PLC Outputs	
Ready	0
S_Mode0Sel	0
S_Mode1Sel	0
S_Mode2Sel	0
S_Mode3Sel	0
S_Mode4Sel	0
S_Mode5Sel	0
S_Mode6Sel	0
S_Mode7Sel	0
S_AnyModeSel	0

Figure 4: The simulator interface of Arcade.PLC showing the results of the range and value-set analysis