TRANSPORT PROBLEMS 2024 Volume 19 Issue 3
PROBLEMY TRANSPORTU DOI: 10.20858/tp.2023.19.3.09

Keywords: ladder programs; formal methods; static analysis

Michał GRZYBOWSKI[1]*, Jakub MŁYŃCZAK[2], Piotr FOLĘGA[3], Lucyna SOKOŁOWSKA[4], Daniel PIETRUSZCZAK[5]

DETECTION OF LADDER PROGRAM UNSTABLE STATES

Summary. This paper presents a modern method of detecting unstable states in ladder programs. Ladder programs are standard formalism used in a wide range of automation applications, especially in railway signaling systems. This formalism is characterized by a lack of explicit program control flow, which can result in the presence of unstable states. A state is unstable, if it leads to cyclic state transitions not anticipated by the designer (loop). The presence of unstable states is one of the possible program defects. This kind of defect is hard to detect and can harm program reliability. The presence of unstable states can be verified with formal methods by the construction of a ladder program model and analysis of its properties. The authors propose a method of static analysis of ladder programs by translating them into predicate logic formulas and construction of formulas expression stability of the program, which can be analyzed with SAT solvers. The presented method allows for automatic verification of the presence of unstable states in the program. The method is conservative (i.e., it concludes that the program has no unstable states only if it is the case). Preliminary experiments performed by authors with a Z3 solver indicate that the method is suitable for use for verification of interlocking programs of computer-based railway signaling systems.

1. INTRODUCTION

Computer systems are widely applied in all kinds of applications. An important industrial application is industrial automation systems. These kinds of systems are often programmed with ladder diagrams [17]. Ladder diagram formalism allowed for an easy migration path from relay circuit automation systems of the past to modern computer-based systems. This is especially true for railway signaling systems, which are often upgraded from relay-based systems to computer-based systems based on ladder diagram programs. The use of computer technology allows for greater functionality of automation systems, but with greater functionality comes greater complexity of modern systems, especially of the control program. This trend forces the development of new techniques of software design and verification, which ensure satisfactory system reliability even in the face of its greater complexity. This work fits the research theme and presents a novel method of detection of one class of defects that may

[1] Silesian University of Technology, Faculty of Transport and Aviation Engineering; Krasińskiego 8, 40-019 Katowice, Poland; e-mail: michal.grzybowski@polsl.pl; orcid.org/0000-0002-4841-147X
[2] Silesian University of Technology, Faculty of Transport and Aviation Engineering; Krasińskiego 8, 40-019 Katowice, Poland; e-mail: jakub.mlynczak@polsl.pl; orcid.org/0000-0003-2947-7980
[3] Silesian University of Technology, Faculty of Transport and Aviation Engineering; Krasińskiego 8, 40-019 Katowice, Poland; e-mail: piotr.folega@polsl.pl; orcid.org/0000-0001-6775-7559
[4] Railway Research Institute IK; Józefa Chłopickiego 50, 04-275 Warsaw, Poland; e-mail: lsokolowska@ikolej.pl; orcid.org/0000-0002-0699-4312
[5] Casimir Pulaski Radom University, Faculty of Transport, Electrical Engineering and Computer Science; Malczewskiego 29, 26-600 Radom, Poland; e-mail: d.pietruszczak@uthrad.pl; orcid.org/0000-0001-8943-0805
* Corresponding author. E-mail: michal.grzybowski@polsl.pl

be present in ladder diagrams. Although the method applies to ladder diagram programs in the general sense, the method is targeted specifically to the detection of errors in railway interlocking systems.

The work is motivated by the authors' experience with modern computer-based railway signaling systems. The railway interlocking is a signaling system responsible for ensuring the safety of the railway traffic. The interlocking, which is the central element of the railway signaling system, is in essence a specialized industrial automation system. It usually realizes several simple automation functions. The functions encompass point machine operation, railway signal aspect determination, and similar operations, all while satisfying interlocking dependencies specified by the railway infrastructure manager. In the first half of the 20th century [1], a relay-based interlocking was invented. A high degree of automation, ease of use, and reliability compared to older interlocking systems led to the wide use of relay-based interlockings and ensuing decades of further development and standardization, until the introduction of computer-based systems. This development made use of ladder diagrams in computer-based interlockings a very pragmatic choice. A large number of railway infrastructure managers have developed standard relay diagrams for relay-based interlocking, which could be used as a base for ladder diagram programs in the first computer-based interlocking systems (e.g., [2]).

Ladder diagrams used in computer-based railway signaling systems have certain differences compared to ladder diagrams used in standard industrial automation, which raises their design effort. A major difference is the more sophisticated evaluation model. Common of the shelf (COTS) PLC controller has usually simple mode of operation. Its program execution model consists of:

1. Input sensing,
2. Ladder diagram program execution,
3. Output setting.

In contrast, some railway signaling systems repeat step 2 (ladder diagram program execution) in a loop until program variables stop changing (until the program stabilizes). This behavior stems from the fact, that relay ladder diagram systems do not have any cycle notion. Relays switch their state (pick up and release) essentially as soon as current starts/stops flowing through their coils. Thus, dynamic, intermediate states are short-lived given the relay's short switching time of around 100 milliseconds.

Computer-based interlocking usually works in cycles and cycle time may be about 0.5 seconds [3]. Such relatively long cycle time makes intermediate states observable, so in order to increase evaluation performance, Boolean equations are evaluated during a single cycle as many times as needed in order to propagate all variable changes through the program. This behavior puts a strict requirement on ladder diagram program construction. If it enters an evaluation loop (every ladder scan causes a change in the program variable values), then interlocking essentially detects it as a critical system failure. Thus, it can be argued that while the presence of evaluation loops does not negatively impact safety, it negatively impacts the system's perceived reliability.

According to the authors' experience, this kind of defect can be very difficult to detect. Ladder diagram programs may not have explicit flow control facilities as such (i.e., conditional or loop constructs), so it is difficult to verify program behavior by manual review. Evaluation loops may be triggered by very specific combinations of system inputs and system state, which allows for these kinds of defects to remain undetected even after thorough system testing.

Although the presented background justifies research on the presence of evaluation loops, the example given by the authors can be considered as a niche domain and, thus, is of no interest to the greater research community. It should be noted however, that this kind of program defect can be present in the standard industrial automation domain and can have various negative consequences. If the effects of the evaluation loop are observable on system inputs, then the system may essentially exhibit behaviors not foreseen by designers and may incorrectly drive the supervised automation processes. Depending on the system mission profile and defect type, these behaviors can lead to the excessive use of resources (e.g., increased energy consumption in heating/cooling systems), increased component wear due to oscillating system outputs, or downright equipment damage.

Ladder diagram features constitute an active research area. Verification of ladder program properties has been considered in [4, 5]. Work [4] proposed an approach based on theorem proving. It presents how to map ladder diagram program fragments to algebraic expressions and means of manipulating them to prove program properties. The theorem proving is further analyzed in [5], which presents

the mapping of ladder diagram program constructs into WhyML specification language, which can used in the Why3 environment for automatic proving of theorems on program execution.

Authors consider work presented in [6, 7] as directly related to the current research problem. The authors of these papers have considered closely related problems of relay races, which they have defined as changes in system output from scan to scan (between ladder diagram program executions) with fixed inputs, timers, and counters. The relay race detection method is based on a generalized dataflow analysis of the considered ladder diagram program. Based on the structure of the considered program, a set of constraints is generated, which can be used to reason about program values. Afterwards, the program input space is randomly sampled and checked generated constraints to detect the presence of a relay race for selected random input assignment. By sampling a sufficiently large number of input assignments, one can answer with high probability if the program is race-free.

The analysis is conservative, given that the algorithm detects only actual relay races. On the one hand, it increases confidence in algorithm results (all indicated relay races shall be investigated because they are guaranteed to be actual relay races). On the other hand, one cannot be sure that if a program is indicated as race-free by the algorithm, then it is race-free. It can be shown that the risk of undetected race is small, but it cannot be eliminated. The algorithm presented in the current work has opposite properties, as it confirms the lack of unstable states only when it is actually the case, but it can indicate potentially unstable states due to a limited search horizon as will be evident from algorithm construction.

Ladder diagram programs can be modeled by different formalisms (e.g., Petri nets). The authors of [8] present a colored Petri net model of a ladder diagram program, which can be used for program simulation and verification of its properties. A colored Petri net formalism is a sophisticated one, which can increase the effort in the actual analysis. In [9] a different model is presented in basic Petri net formalism. The developed model has been applied to relay race detection in [10]. This method allows for automatic verification of the presence of relay races and presents useful information for analyzing them, but it has one major flaw: the analysis essentially considers the entire state space of the program. Thus, it applies only to small programs, as it suffers from the state explosion problem. In the author's opinion, the algorithm could be applied to larger ladder diagram programs, but it would require the size of analyzed state space to be limited, which would result in similar properties to the algorithm presented in [7].

The authors of [18] formalized the notion of a relay race in computation tree logic (CTL) formalism. The formalization allowed for the automatic detection of relay races with the use of a model checker. It should be noted that the experimental results have been obtained for small programs (three inputs and two outputs). Thus, the method has not been validated for use with industrial, large programs. This doubt is justified because a well-known limitation of model checking of CTL formulas is the state-space explosion problem ([19]), which results in a non-linear increase of analysis time for larger programs due to rapidly growing state space, which needs to be checked by the model checker.

Research on program analysis is also carried out in the domain of computer-based railway signaling systems. A common design is interlocking based on Boolean equations, in which the actual interlocking functions are implemented as a list of Boolean expressions, which govern the program execution (e.g., [2]). Such an interlocking program is essentially a ladder diagram program written in domain-specific language.

Due to the safety-critical nature of such programs, they are the subject of active research aimed at formal modeling and formal verification of program properties. The best example of this theme is presented in [11], which discusses the relation between railway interlocking and ladder diagram programs, introduces a formal model of ladder diagram program in the form of a propositional logic formula, and presents how the model can be used to formally describe and verify the fulfillment of safety requirements. The model presented in [11] describes how the program executes in terms of transition between the current state and the next program state. Thus, the model provides a good insight into program execution. The current work can be viewed as an application of these ideas not to verify particular properties but to verify the lack of defects of a particular type in the considered program.

It should be noted that there is no single correct model. Depending on the modeling goals, vastly different models can be created, even for similar problem domains. A good example of this is developments done for the Sternol programming language, presented in [12, 13]. These papers focused

on more specific Boolean equation-based systems, which follow the evaluation method sketched by the authors in the discussion of research background (repeated program evaluation until program state stops changing). In such a system, it is not particularly important how the program state transforms from one Sternol program execution to another but rather in what state the Sternol program ends up at the end of the evaluation cycle. In other words, of particular interest are the program's fixed points.

A common technique for analyzing formal model properties is the analysis by reduction to the SAT problem. This technique is based on the derivation of propositional (SAT problem) or predicate logic (SMT problem) formulas for a given problem, and then on analysis of the resulting formulas (which can be often large). The problem of deciding whether there exists an assignment of formula variables, under which the formula is true, is a SAT / SMT problem. In case of propositional formulas it is the SAT problem and in case of predicate logic formulas it is the SMT problem. An introduction to this technique is presented in [14]. As will be seen later, the technique of analysis by reduction to the SAT problem is also used by the authors of this work. Given that the SAT problem is a well-defined, general problem, there is active research on more sophisticated SAT solvers [15], which allows SAT-based methods to improve the performance/scale to larger problems using newer generations of SAT solvers.

The current work differs from the prior art [6, 7, 10, 18] in the problem considered. The prior art considers only the problem of relay race, while the current one of the more general notion of state stability. The distinction may be small, but it is important. Ladder diagrams may employ variables used for the detection of criteria triggering some action. Such a scheme may result in benign race – activation of the criteria variable, execution of action, and deactivation of the criteria variable, with fixed inputs. An example of such a scheme is route release in railway signaling system E, which is widely used in the Polish railway network ([20]). The presence of route release criteria activates the Zw relay, which causes the activation of the U relay and the deactivation of the Zw relay. If the ladder diagram implementing this design provided output informing on the state of Zw, then prior art methods would detect relay race, while current work would permit accepting such behavior during the analysis.

This paper aims to precisely define the hinted previous evaluation loop by introducing a definition of the program's unstable state and presentation of the decision procedure to verify the presence of unstable states in the given ladder diagram program. For this purpose, this paper suggests using the ladder diagram formal model, which allows for precise problem formulation and automatic analysis with the aid of readily available SAT solvers.

2. MATERIALS AND METHODS

2.1. Ladder diagram program formal model

Formally, the ladder diagram program P can be defined as a tuple

$$P = (I, O, M, C, E, Init, S) \tag{1}$$

where:

I – set of program input variables (variables read by the program, set by external means),

O – set of program output variables (variables written by the program, read by external means),

M – set of program internal variables (variables read and written by the program),

C – condition set – smallest set satisfying following relations for $v \in I \cup M$, $c_1, c_2 \in C$:

$$v \in C \tag{2}$$

$$\sim c_1 \in C \tag{3}$$

$$(c_1 \wedge c_2) \in C \tag{4}$$

$$(c_1 \vee c_2) \in C \tag{5}$$

E – Boolean expressions – set of pairs $(v, cond)$, which specify condition $cond \in C$, under which variable v is assigned to value "true",

$Init$ – initial assignment – Boolean-valued (from set $Bool$) function Init: $I \cup O \cup M \rightarrow Bool$, which assigns to each program variable its initial value,

S – schedule – sequence of all elements of E, which specifies evaluation order of program variables in single program execution.

As is evident from the construction of C, the conditions are program variables, condition negations, their conjunctions, and disjunctions.

As mentioned in the introduction, the formal model construction depends on properties, which will be analyzed with models help. This work considers how the program operates; thus, a model that can describe it is of interest. Execution of the program can be viewed as transitions in program state space, which are formed by all possible combinations of program variable values (all program variable assignments), which is graphically depicted in Fig. 1.

The program may be modeled as predicate logic formulas describing its initial state and possible transitions (as a result of program execution). In the ensuing discussion, v will denote the program variable in the transition source state, and v' will denote variable v in the transition target state. The value of variable v will be denoted as $[v]$. One can treat $[\cdot]$ as valuation function $[\cdot]:$ set of all model variables $\rightarrow Bool$, which assigns each model variable its value. In the considered model, transitions are represented with valuations. Description of program behavior can thus be achieved by describing its valuations.

$InitState(P)$ formula, where P is ladder diagram program, is defined as:

$$InitState(P) = \bigwedge_{v \in I \cup O \cup M} [v] = Init(v) \tag{6}$$

In other words, a transition starts in the initial state if all variables in the transition source state have initial values. $Transition(P)$ formula describes program execution and is defined as:

$$Transition(P) = \bigwedge_{v \in O \cup M} \bigwedge_{(v,cond) \in E} ([v'] \Leftrightarrow ScheduledCond(cond, v, S)) \tag{7}$$

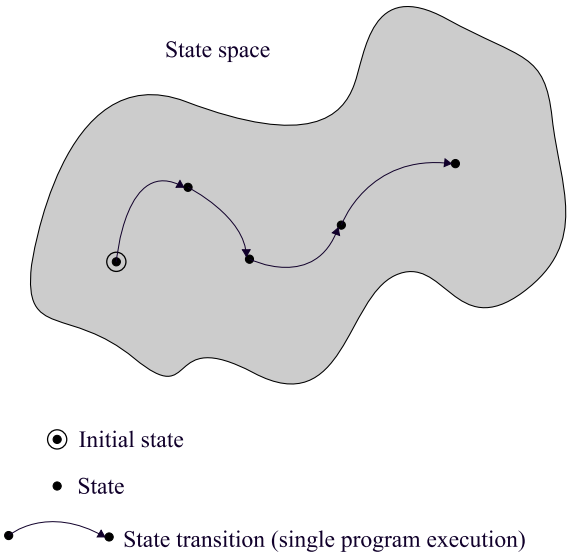

Fig. 1. Program execution as state transitions (source: original work)

The formula formalizes the notion that as a result of state transition, variables assume values according to defining conditions. In order to finish the description of the formula, it is necessary to define $ScheduledCond(cond, v, S)$ formula, which captures the impact of variable evaluation order on program execution:

$$ScheduledCond(v_1, v_2, S) = \begin{cases} [v_1] \text{ if } v_1 \text{ is evaluated after } v_2 \text{ according to } S \\ [v_1'] \text{ if } v_1 \text{ is evaluated before } v_2 \text{ according to } S \end{cases} \tag{8}$$

$$ScheduledCond(\sim c, v, S) = \sim ScheduledCond(c, v, S) \tag{9}$$

$$ScheduledCond(c_1 \wedge c_2, v, S) = ScheduledCond(c_1, v, S) \wedge ScheduledCond(c_2, v, S) \tag{10}$$

$$ScheduledCond(c_1 \vee c_2, v, S) = ScheduledCond(c_1, v, S) \vee ScheduledCond(c_2, v, S) \quad (11)$$

As is readily seen, schedule S affects only which model variable is present in the formula representing condition – a variable from target state (if according to the schedule, the condition variable is evaluated before considered variable) or variable from source state (if according to the schedule, the variable is evaluated after considered variable).

2.2. Ladder diagram program stability

The definition of the unstable state is introduced in order to precisely capture the intuitive notion of the evaluation loop. A ladder diagram program state is stable if the program can perform a finite number of state transitions with fixed inputs (timer state shall be treated as program input) from the state. A ladder diagram program state is unstable if the program is not stable (i.e., the program may perform an infinite number of state transitions with fixed inputs from the state). It should be noted that this definition captures the essence of the notion of the evaluation loop. If the program enters an unstable state, then each transition it makes will trigger a new transition, thus program variables will not settle down (or the program will not reach a fixed point).

It should be noted that due to the rigid structure of the ladder diagram program (a fixed number of variables with finite ranges) if the program contains an unstable state, then program there must exist an execution path that starts and ends in the same state and for which all states in the execution path have fixed values of input variables.

If the program contains no such execution path, then it cannot contain an unstable state, for then all execution paths with fixed values of input variables must consist of finite sequences of unique program states. The graphically unstable state is illustrated in Fig. 2. A program is called stable if all its states are stable. Conversely, a program is called unstable if it contains an unstable state.

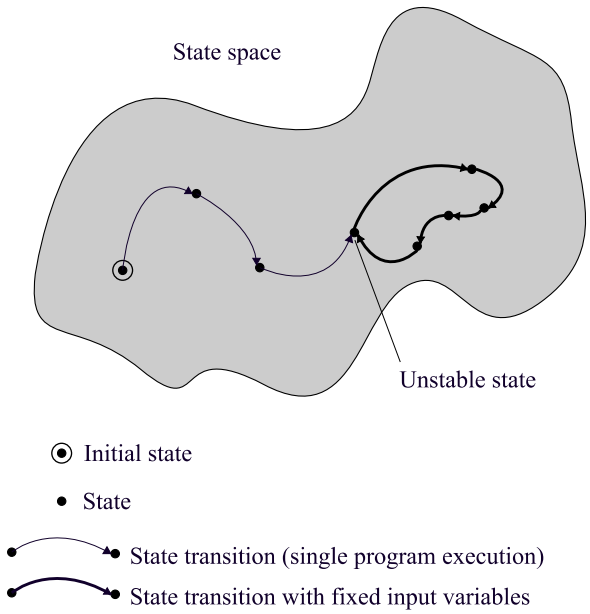

Fig. 2. Graphical interpretation of unstable state (source: original work)

The definition of a stable state can be formalized in the model introduced in paragraph 2.1. For this, additional predicates need to be introduced. First is $FixInput(P)$, which describes a state transition, in which input values are fixed:

$$FixInput(P) = \wedge_{v \in I}[v'] = [v] \quad (12)$$

The introduction of this predicate allows for the definition of $Fixpoint(P)$, which formalizes that a state is a fixed point for the program:

$$Fixpoint(P) = FixInput(P) \Rightarrow \bigwedge_{v \in I \cup O \cup M} [v'] = [v] \tag{13}$$

Now, the notion of stabilization can be formalized. In ensuing the formula's notation $Predicate'(x)$, it shall be understood as predicate $Predicate(x)$, in which variables have added an apostrophe (i.e., variable x is replaced with x', variable x' is replaced with x'', and so on). In addition, the notation $Predicate^{(n)}(x)$ shall be understood as $\left(Predicate^{(n-1)}\right)'(x)$. Any $n \in \mathbb{N}$ formula $StableAfter(P, n)$ describes a stable state of the program, which may perform at most n state transitions with fixed inputs:

$$StableAfter(P, n) = \left(\bigwedge_{i=0}^{n} Transition^{(i)}(P) \wedge FixInput^{(i)}(P)\right) \Rightarrow FixPoint^{(n)}(P) \tag{14}$$

Now, a stable state may be described with the formula $Stable(P)$:

$$Stable(P) = \bigvee_{n \in \mathbb{N}} StableAfter(P, n) \tag{15}$$

In practical terms, all program states are stable if formula $Stable(P)$ is satisfied for all possible valuations $[\cdot]$. A program state is unstable, if formula $Stable(P)$ is not satisfied with some valuations starting from the state.

2.3. Ladder diagram program stability analysis

The structure of formula $Stable(P)$ may be used to devise an algorithm for analyzing if the program is stable. Note that for any natural number $n \in \mathbb{N}$, $StableAfter(P, n)$ is essentially a formula specifying relations between program values in $n + 1$ states. To be more precise, the formula specifies properties of valuations of variables from $n + 1$ states. Such formulas can be effectively analyzed with freely available SAT solvers. Valuation values can be represented as free variables in a formula manipulated by an SAT solver. Essentially one can ignore the presence of valuations and instead for each transition assign to each program variable two free variables – one in the source state and one in the target state. Then, the SAT solver may be used to analyze if the formula is satisfiable, or in terms of the model if there is a valuation that satisfies the considered formula.

The last observation that is needed is that an analysis of some property $Pred(x)$ holds for all valuations can be reduced to analysis if its negation $\sim Pred(x)$ holds for any valuation, which can be readily answered by the SAT solver. Thus, one can conservatively analyze if the ladder diagram program P is stable with the following algorithm:

1. Set considered number of steps $i \coloneqq 1$.
2. Check with an SAT solver if formula $StableAfter(P, i)$ is true (satisfied by all valuations).
3. If formula $StableAfter(P, i)$ is true, then the program is stable and stabilizes after at most i steps – end of the algorithm.
4. If formula $StableAfter(P, i)$ is not true and the considered number of steps i is smaller than the maximum considered number of steps n_{max}, then increase the considered number of steps $i \coloneqq i + 1$ and go to step 2.
5. If $StableAfter(P, i)$ is not true for the maximum considered number of steps $i = n_{max}$, then the program is potentially unstable.

It should be stressed that the algorithm is conservative, as indicated by step 5. According to predicate $Stable(P)$ definition, in order to decide whether the program is stable or not, one would need to consider an arbitrary number of steps n and corresponding predicates $StableAfter(P, n)$, so, obviously, the presented algorithm is only an approximation. At the same time, the program is stable if the predicate $StableAfter(P, n)$ is true for some n, and, thus, the algorithm is sound. The algorithm decides that the program is stable only if predicate $Stable(P)$ is true. This stems from the fact that the algorithm is essentially the search for the smallest n, for which $StableAfter(P, n)$ is true, which implies $Stable(P)$. The algorithm may incorrectly classify the input program as unstable if the number of considered steps n_{max} is too small, or in other words if $StableAfter(P, n)$ is true for some $n > n_{max}$.

2.4. Example

A trivial example is presented in Fig. 3 to illustrate algorithm operation,. The program consists of one input variable (C) and two internal variables A and B. The initial state of all variables is assumed false.

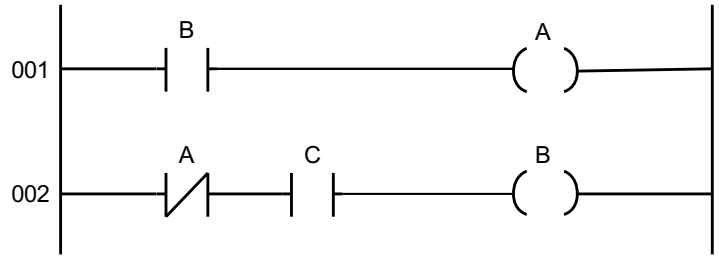

Fig. 3. Example ladder program (source: original work)

As can be seen, the program's stable states are precisely those, in which variable C is false, and unstable states are those, in which variable C is true. For this program, the model is as follows:

$$InitState(P) = {\sim}[A] \wedge {\sim}[B] \wedge {\sim}[C] \tag{16}$$

$$Transition(P) = ([A'] \Leftrightarrow [B]) \wedge \big([B'] \Leftrightarrow ({\sim}[A'] \wedge [C'])\big) \tag{17}$$

And the formulas of interest $FixInput(P)$ and $FixPoint(P)$ are as follows:

$$FixInput(P) = ([C'] = [C]) \tag{18}$$

$$FixPoint(P) = ([A'] = [A]) \wedge ([B'] = [B]) \wedge ([C'] = [C]) \tag{19}$$

The formulas expressing stability after 1 and 2 steps are as follows:

$$\begin{aligned} &StableAfter(P,1) \\ &= ([A'] \Leftrightarrow [B]) \wedge \big([B'] \Leftrightarrow ({\sim}[A'] \wedge [C'])\big) \wedge ([C'] = [C]) \\ &\wedge ([A''] \Leftrightarrow [B']) \wedge \big([B''] \Leftrightarrow ({\sim}[A''] \wedge [C''])\big) \wedge ([C''] = [C']) \Rightarrow \\ &([A''] = [A']) \wedge ([B''] = [B']) \wedge ([C''] = [C']) \end{aligned} \tag{20}$$

and

$$\begin{aligned} &StableAfter(P,2) \\ &= ([A'] \Leftrightarrow [B]) \wedge \big([B'] \Leftrightarrow ({\sim}[A'] \wedge [C'])\big) \wedge ([C'] = [C]) \\ &\wedge ([A''] \Leftrightarrow [B']) \wedge \big([B''] \Leftrightarrow ({\sim}[A''] \wedge [C''])\big) \wedge ([C''] = [C']) \\ &\wedge ([A'''] \Leftrightarrow [B'']) \wedge \big([B'''] \Leftrightarrow ({\sim}[A'''] \wedge [C'''])\big) \wedge ([C'''] = [C'']) \\ &\Rightarrow \\ &([A'''] = [A'']) \wedge ([B'''] = [B'']) \wedge ([C'''] = [C'']) \end{aligned} \tag{21}$$

Both formulas $StableAfter(P,1)$ and $StableAfter(P,2)$ are not tautology – the first one is falsified by valuation $[A] = [A'] = [B] = [B''] = [C] = [C'] = [C''] = \text{true}$ and $[A''] = [B'] = \text{false}$, while the second is falsified by valuation $[A] = [A'] = [A'''] = [B] = [B''] = [C] = [C'] = [C''] = [C'''] = \text{true}$ and $[A''] = [B'] = [B'''] = \text{false}$.

If the program is modified by removing the normally closed contact of A in the second rung, then the formula $StableAfter(P,1)$ will have the following form:

$$\begin{aligned} &StableAfter(P,1) \\ &= ([A'] \Leftrightarrow [B]) \wedge ([B'] \Leftrightarrow [C']) \wedge ([C'] = [C]) \wedge ([A''] \Leftrightarrow [B']) \\ &\wedge ([B''] \Leftrightarrow [C'']) \wedge ([C''] = [C']) \Rightarrow \\ &([A''] = [A']) \wedge ([B''] = [B']) \wedge ([C''] = [C']) \end{aligned} \tag{22}$$

This formula is not tautology (is falsified by valuation $[A''] = [B'] = [B''] = [C] = [C'] = [C''] = \text{true}$ and $[A] = [A'] = [B] = \text{false}$). The formula $StableAfter(P,2)$ will have the following form:

$$StableAfter(P,2) = ([A'] \Leftrightarrow [B]) \wedge ([B'] \Leftrightarrow [C']) \wedge ([C'] = [C]) \wedge ([A''] \Leftrightarrow [B']) \wedge ([B''] \Leftrightarrow [C'']) \wedge ([C''] = [C']) \wedge ([A'''] \Leftrightarrow [B'']) \wedge ([B'''] \Leftrightarrow [C''']) \wedge ([C'''] = [C'']) \Rightarrow ([A'''] = [A'']) \wedge ([B'''] = [B'']) \wedge ([C'''] = [C'']) \quad (23)$$

As can be seen, formula (23) is a tautology, thus the modified program has no unstable states.

3. RESULTS

The authors have successfully applied the presented algorithm for the analysis of railway interlocking object programs written in the Sternol programming language. The language has been selected for evaluation of ideas presented in this work due to the authors' area of expertise (the authors work professionally on railway signaling systems developed with Sternol) and from the perceived difficulty of assuring lack of unstable states in the Sternol program with standard verification methods. Sternol can be considered a derivative of ladder diagram programs. Its overview is given in [13]. The Sternol is a graphical programming language, in which one may define a series of object types, each consisting of a set of variables. The program specifies the finite domain of each variable, the logical expressions specifying the value to assume, and the evaluation order of variables in a single object. The example Sternol variable is presented in Fig. 4.

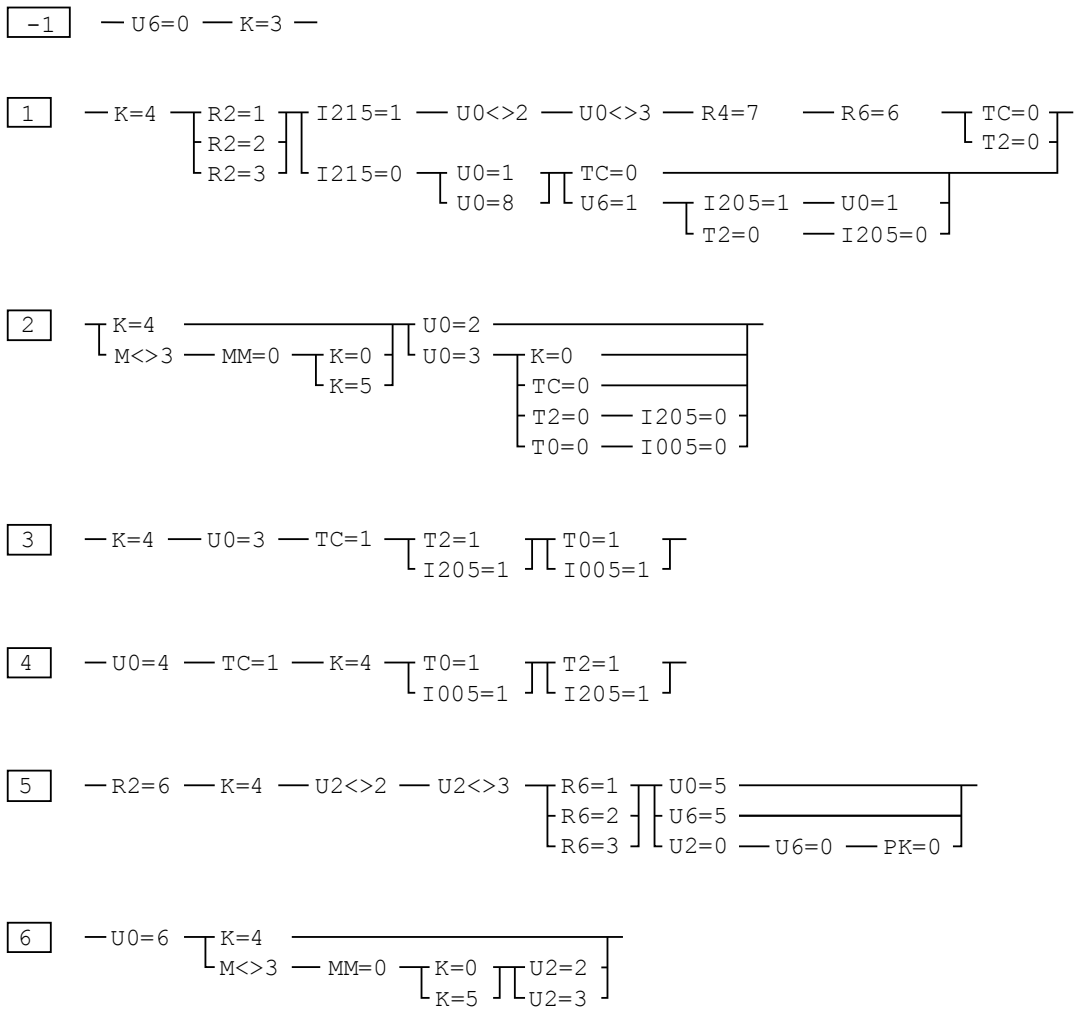

Fig. 4. Example Sternol variable (source: [2])

The expression language used in Sternol follows standard ladder diagram notation. Horizontal connections represent logical conjunction and vertical connections represent logical alternative. Contrary to standard ladder diagrams, terms present in Sternol logical expressions are arithmetic relations between Sternol variables.

Due to additional features of Sternol, such as variables spanning finite domains, the algorithm has been successfully extended and implemented with the aid of the Z3 SMT solver ([16]). The SMT solver is an extension of the SAT solver. While the SAT solver analyses propositional formulas, the SMT solver may be used for the analysis of predicate logic formulas from some well-specified logic system. The prototype implementation has been applied to the analysis of some example programs available to the authors.

Experimental runs conducted by authors indicated that for considered stable programs analysis time was very short (about 5–20 seconds). For unstable programs, time varied on program complexity and number of considered steps, spanning from 5 (for programs of up to 50 variables) to 30 seconds per step (for programs of around 300 variables). Practical experiments indicate that algorithm conservativeness in practice is not problematic, as the programs are either considered stabilized after at most nine evaluations or contained unstable states. In hindsight such a limit on the number of steps needed for analysis should be obvious – after all authors of considered programs try to avoid unstable states and the best method is the correct design of the program to stabilize as quickly as possible. Nevertheless, authors have found new, previously unknown unstable states in most of the considered programs (thought to be stable by their designers).

As indicated in the previous paragraph, authors have used algorithm implementation not only to check if considered programs are stable but also to find actual unstable states. This could be accomplished owing to the fortunate side effect of algorithm implementation with the use of an SAT/SMT solver. As hinted in paragraph 2.3, the implementation checked if formula $StableAfter(P, i)$ is true by checking if its negation $\sim StableAfter(P, i)$ is satisfiable. The SMT solver used (Z3) reported valuations, which satisfied $\sim StableAfter(P, i)$. As discussed in paragraph 2.1, a valuation in this context is essentially a sample program execution, in which the program would not stabilize in i steps. This information has been sufficient to confirm if the algorithm found an actual unstable state and if so, how it looked.

4. DISCUSSION

The results confirm the method's correctness and effectiveness. The fact that the method could be applied to Sternol programs also shows that the method can be easily adapted by appropriate extension of the model of the ladder diagram program. It should be pointed out here that the authors extended the model not only to cover obvious Sternol features (variables with finite domains instead of Boolean domains) but also to cover more sophisticated features, such as timer or Sternol inter-object communication mechanisms (channels).

Compared to methods presented in [6,7], the method is based on different principles (predicate logic model of the program instead of dataflow analysis) and has a different accuracy profile. The methods presented in [6,7] correctly report the presence of relay races, while the method presented in the current work correctly reports the absence of unstable states. The algorithm presented in [10] is precise, but due to its construction (analysis of the entire state space), it is susceptible to state explosion and thus is either limited to short ladder diagram programs or needs to limit considered fragments of state space. Thus, in the authors' opinion, for large programs, it has similar properties to methods presented in [6, 7].

It correctly reports the presence of relay races but may not report all of them if it does not analyze the entire state space. It should be noted that the method presented in this work can be considered as a generalization of relay race detection. A program containing a relay race can be defined in the current context as one that has unstable states or stabilizes in more than zero steps. In addition, the algorithm given in this work can be considered more user-friendly, as the most obvious implementation leads to the generation of sample executions, which show a lack of stabilization in the required number of steps.

The algorithm presented in [18] models ladder program as timed automaton and defines relay race as property encoded in CTL formula. The algorithm presented in [18] considered a more limited problem (detection of a relay race) then current work. The method presented in the current work may be applicable in different domains and provides greater insight into program execution (a side effect of the analysis is the determination of the number of ladder scans before a fixpoint is reached). From a more practical perspective, the algorithms differ in the logic used. The algorithm presented in [18] employs CTL logic to describe globally the required properties, while the current work proposes the construction of predicate logic formulas describing sequences of executions (ladder scans).

From this difference stem different trade-offs of the algorithms. The use of CTL logic in [18] limits the number of false positives, provided that the entire state space can be analyzed. The method presented in this work needn't consult the entire state space of the program but may result in more false positives. The formulas describe the properties of transitions allowed by the program, without considering from which states the transitions can be taken. This leads to smaller model checker models (only a sequence of transitions of a given length is considered instead of all transitions from the initial state). The problem of false positives can be reduced by the introduction of additional invariants as described in [11] but requires the method user to state the required invariants.

The analysis described in this work shows that the ladder diagram model discussed in [11] can give additional insights into program execution compared to the fixed-point program model presented [13]. This result suggests that the ladder diagram model presented in [11] could be analyzed more deeply for applicability to the analysis of Sternol programs.

The algorithm presented in this work also shows that a good choice of program model can allow for simple characterization of interesting properties. The formulas used to specify program stability employed not much more than equality between variable value between source and target state. It could be argued that such a description is impossible in different program models, such as the Sternol fixed-point program model presented in [13].

It should be noted that there are several deficiencies or open points related to the current description of the method:

1. The analysis does not consider the initial state of the program. It considers any variable combination as a potential source state, which is certainly too simplistic. On the one hand, this increases confidence in the method (as it overapproximates considered state space, it certainly considers all reachable states); on the other hand, it may (and has in the authors' experiments) lead to spurious results stemming from considering unreachable program states. To combat this, the authors used in prototype implementation technique of maintaining additional program invariants described in [11] (p. 25), but in the authors' opinion, it warrants additional research if the initial state can be incorporated into the method or if there is any automatic method of deriving high-quality program invariants.
2. Experiments performed by the others suggest that the method is suitable for use in verifying the stability of Sternol object programs. Sternol object programs are used as building blocks of larger systems, so their complexity is limited to around 500 variables. It should be investigated whether the methods are suitable for the analysis of larger ladder diagram programs. Authors suspect that method may be applicable, but possibly with some program pruning algorithm, which can reduce the considered program to an interesting fragment (set of interdependent variables), which, in practice, should be limited in size.
3. The presented algorithm is simplistic, as it deems a program as unstable only if there can be found an execution that stabilizes in the considered number of steps. Experiments carried out suggest that many instabilities are simple oscillations of the program between two states. It is an open question if this insight can be used to improve the algorithm, for example, by considering that an SAT/SMT solver is used (and checking if generated example valuations do not contain such oscillations) or by formalizing the notion of oscillation.

5. CONCLUSIONS

The work discusses the problem of ladder diagram program stability. Ladder diagram programs are often used for the implementation of railway interlocking, which assures the safety of railway traffic. Ladder diagram program stability is an important property, which shall be possessed by every ladder diagram program, but especially by programs used in safety-critical applications, such as railway signaling. Under specific conditions, an unstable ladder diagram program can repeatedly change program values with fixed inputs, which can lead to incorrect operation (or actual failure) of the entire system.

A discussion is carried out in the framework of the ladder diagram program model in predicate logic (although it reduces to an essentially propositional formula). The selected formal model expresses relations between values before and after execution and, thus, allows for the precise formulation of an unstable state. Given that the model is constructed in predicate logic, it is amenable to analysis with an SAT solver.

The use of an SAT solver allows for the specification of an algorithm, which can automatically verify if a given ladder diagram program has unstable states. It has a different accuracy profile from existing state-of-the-art methods based on dataflow analysis and on Petri net models. The existing methods can be used to show the presence of relay races, while the method presented in this work can be used to verify their absence. This is common with methods based on CTL logic, but the current method is more practical due to the consideration of simpler state space. The method is conservative in the sense that it can falsely indicate the presence of unstable states.

Method accuracy is controlled by the number of considered program execution steps (the higher the number considered, the more accurate the results). the design of the method based on the program model in predicate logic allows for easy extensibility. The adaptation of the method for different languages/different language features requires additional model variables and formulas describing their change during program execution. The design of the algorithm based on SAT/SMT solvers allows for easy implementation of the method by reusing existing solvers.

A prototype implementation of the method has been implemented for the Sternol language based on the Z3 solver. The prototype demonstrates the method's feasibility and ease of extension to ladder diagram language derivatives. Experiments carried out with the prototype indicate, that SAT solver models generated as a side effect of the analysis provide sufficient information to analyze the unstable states and to find out why they arise in the program. The generated models describe actual program executions, which do not lead to a stable state in the considered number of steps, which provides the program designer with sufficient information to debug the program. The experiments support the argument that the method may be used to increase the reliability of railway signaling systems by automatic uncovering of latent defects.

The work presents also open points for further research: consideration of the program's initial state in order to increase the method's accuracy, experiments related with establishing method performance characteristics, and method refinements for quicker detection of unstable states.

References

1. Huang, L. The Past, Present and Future of Railway Interlocking System. In: *2020 IEEE 5th International Conference on Intelligent Transportation Engineering*. Beijing: Curran Associates. Inc. 2020. P. 170-174.
2. Hagelin, G. ERICSSON Safety System for Railway Control. In: *Software Diversity in Computerized Control Systems*. Springer. 1998. P. 11-21.
3. Lindqvist, L. & Jadhav, R. Application of communication based Moving Block system on existing metro lines. *WIT Transactions on State of the Art in Science and Engineering*. 2010. Vol. 46. P. 55-64.
4. Roussel, J.M. & Denis, B. Safety properties verification of ladder diagram programs. *Journal Européen des Systèmes Automatisés (JESA)*. 2002. Vol. 36. No. 7. P. 905-917.

5. Belo Lourenço, C. & Cousineau, D. & Faissole, F. & Marché, C. & Mentré D. & Inoue, H. Automated formal analysis of temporal properties of Ladder programs. *International Journal on Software Tools for Technology Transfer*. 2022. Vol. 24. No. 6. P. 977-997.
6. Su, Z. *Automatic Analysis of Relay Ladder Logic Programs*. Report No. UCB/CSD-97-969. Berkeley: University of California. 1997. P. 1-26.
7. Aiken, A. & Manuel, F. & Su, Z. Detecting races in Relay Ladder Logic programs. *International Journal on Software Tools for Technology Transfer*. 2000. Vol. 3. No. 1. P. 93-105.
8. da Silva Oliveira, E.A. & da Silva, L.D. & Gorgônio, K. & Perkusich, A. & Martins, A.F. Obtaining formal models from Ladder diagrams. In: *2011 9th IEEE International Conference on Industrial Informatics*. Lisbon: IEEE. 2011. P. 796-801.
9. Chen, X. & Luo, J. & Qi, P. Method for translating ladder diagrams to ordinary 0 nets. In: *2012 IEEE 51st IEEE Conference on Decision and Control (CDC)*. Maui: IEEE. 2012. P. 6716-6721.
10. Luo, J. & Zhang, Q. & Chen, X. & Zhou, M. Modeling and Race Detection of Ladder Diagrams via Ordinary Petri Nets. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*. 2018. Vol. 48. No. 7. P. 1166-1176.
11. Kanso, K. & Moller, F. & Setzer, A. Automated Verification of Signalling Principles in Railway Interlocking Systems. *Electronic Notes in Theoretical Computer Science*. 2009. Vol. 250 No. 2. P. 19-31.
12. Petersen, J.L. *Methods for Validating Railway Interlocking Systems*. PhD thesis. Kongens Lyngby: Technical University of Denmark. 1998.
13. Borälv, A. Case Study: Formal Verification of a Computerized Railway Interlocking. *Formal Aspects of Computing*. 1998. Vol. 10. No. 4. P. 338-360.
14. Claessen, K. & Een, N. & Sheeran, M. & Sörensson, N. & Voronov, A. & Åkesson, K. SAT-Solving in Practice, with a Tutorial Example from Supervisory Control. *Discrete Event Dynamic Systems*. 2009. Vol. 19. No. 4. P. 495-524.
15. Alouneh, S. & Abed, S. & Al Shayeji, M.H. & Mesleh, R. A comprehensive study and analysis on SAT-solvers: advances, usages and achievements. *Artificial Intelligence Review*. 2019. Vol. 52. No. 4. P. 2575-2601.
16. de Moura, L. & Bjørner, N. Z3: An Efficient SMT Solver. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Budapest: Springer. 2008. P. 337-340.
17. Walker, M. & Bissell, C. & Monk, J. The PLC: A Logical Development. *Measurement and Control*. 2010. Vol. 43. No. 9. P. 280-284.
18. Mesli-Kesraoui, S. & Goubali, O. & Kesraoui, D. & Eloumami, I. & Oquendo, F. Formal Verification of the Race Condition Vulnerability in Ladder Programs. In: *2020 IEEE Conference on Control Technology and Applications (CCTA)*. Montreal. 2020. P. 892-897.
19. Boukala, M.C. & Petrucci, L. Distributed model-checking and counterexample search for CTL logic. *International Journal of Critical Computer-Based Systems 3*. 2012. Vol. 3. No. 1-2. P. 44-59.
20. Dąbrowa-Bajon, M. *Podstawy sterowania ruchem kolejowym.* [In Polish: *Basics of rail traffic control*]. Third Edition. Warsaw: Publishing House of the Warsaw University of Technology. 2014. 391 p.

Received 02.03.2023; accepted in revised form 03.09.2024