Model Checking Industrial Robot Systems

Markus Weißmann[1], Stefan Bedenk[2], Christian Buckl[3], and Alois Knoll[1]

[1] Technische Universität München, Fakultät für Informatik
Boltzmannstrasse 3, 85748 Garching, Germany
weissmann@ini.tum.de
knoll@in.tum.de
[2] AUDI AG
85057 Ingolstadt, Germany
stefan.bedenk@audi.de
[3] fortiss GmbH
Guerickestr. 25, 80805 München, Germany
buckl@fortiss.org

Abstract. Modern production plants are highly automated complex systems consisting of several robots and other working machines. Errors leading to damage and stop of production are extremely expensive and must be avoided by all means. Hence, the state of practice is to test control programs in advance which implies high effort and comes with high costs. To increase the confidence into the control systems and to reduce the necessary effort, this paper proposes to use model checking to verify certain properties. It presents a compiler that can transform industrial robot programs into PROMELA models. Since the statements of the robot programming language can not be mapped directly into PROMELA statements, we apply compiler optimization techniques to close the semantic gap. In case of a specification violation the trace is mapped to the original context so that the robot programmer can reconstruct the problem. As a case study we applied the tool to verify the absence of collisions and deadlocks. We were able to detect one deadlock in a car-body welding station with 9 robots, correct the program and verify the correctness of the resulting system.

Keywords: model checking, abstract interpretation, industrial robots, distributed systems

1 Introduction

Modern production plants are highly automated, complex systems. They use industrial robots for lifting, welding, bonding and other tasks for which the robots may have to work in the same area. Design errors resulting e. g. in collisions or deadlocks can lead to downtime of the plant which are extremely expensive. This is especially true for modern production plants that use just in time production; in these systems each industrial robot system easily becomes a single point of failure the outage of which can quickly result in an outage of the whole production line.

To address this problem upfront, the control programs are tried to be kept as simple as possible. This approach is counteracted upon by the necessity for more and more dynamic production processes. For example car body production plants are used to produce different variants of cars, e. g. a fastback and station wagon, different numbers of doors, etc. This flexibility has to be provided by the control programs.

Virtual commissioning is an approach to find problems in the programming upfront, before the plant is built. Most commonly it is carried out as a computer simulation of the industrial robot system. Different levels of detail on the physical part of the plant are used, reaching from a plain emulation of the programs to 3D simulations with a hardware-in-the-loop setup of the controllers. Using this simulation the programs can be tested. However this approach implies great testing effort and does not guarantee the correctness of the system.

To target this issue, this paper proposes the use of model checking techniques to verify the correctness of the system with respect to certain properties. This approach has the advantage that it can prove the absence of errors – in contrast to tests. To apply model checking on the control programs, they must first be transformed into a formal model. Since engineers without a background in formal verification should be able to apply this method, too, a manual creation of this formal model is not feasible. An automated translation solves this problem and also the need to keep the formal model and the original robot programs synchronized.

This paper presents a compiler that can transform industrial robot programs into PROMELA models as input for the SPIN model checker [12]. Since the statements of the robot programming language can not be mapped directly into PROMELA statements, we apply compiler optimization techniques to close the semantic gap. To verify certain properties our tool extends the PROMELA code with assertions. The engineer can add further properties in form of linear temporal logic (LTL) formulas. To simplify the formulation of these LTL formulas, LTL patterns for each property to verify are provided.

If SPIN detects a violation of one property, it generates an error trace leading to the violation in the PROMELA model. To help a robot programmer understand the result of the analysis, the presented tool-chain provides a projection of the error-trace in the model back to the original robot programs.

As a case study we applied the tool to verify the absence of collisions and deadlocks. We were able to detect one deadlock in a car-body welding station with 9 robots, correct the robot control programs and verify the correctness of the resulting system.

The structure of the paper is as follows: Section 2 contains preliminaries on the problem and gives an overview of previous work; Section 3 discusses the problem details; our solution is presented in section 4. The results of our analysis of two industrial robot systems can be found in section 5. We conclude in section 6.

2 Background

An industrial robot system typically consists of a programmable logic controller (PLC) and several robots. The PLC and the robots are running the control programs. They control further devices like welding guns and riveting systems, which are not programmable devices; they are either controlled by the PLC or a robot. The PLC controls the system as a whole; it tells the robots which control program to use for the current process. The PLC is also the master hub of the industrial robot system: All up-stream communication from the system devices is routed through the PLC; hence the network topology of the system is a tree.

To reduce costs, increase reliability and to shorten product cycles, manufacturers want to have a high quality control software before starting the commissioning process. The current approach is to model the industrial robot system in a hardware-in-the-loop (HIL) simulation, the so-called virtual commissioning. This method allows testing up-front, but finding bugs is still a tedious task and the absence of bugs can not be shown. To address this problem, we want to introduce formal verification to analyze the control software.

Several approaches to formal verification for embedded systems [7] and more specifically industrial control systems using PLCs [19] are available. A common solution is to use model checking, e. g. with the SPIN model checker; other approaches include abstract interpretation [17] or manually created, computer based proofs [18].

A basic solution for making use of a model checker is to manually create a model of the PLC program, then use a model checker to verify it [14]. As our goal includes providing a system for non-experts in verification, manually created proofs are not a suitable option. More advanced approaches use automatic translation from PLC programs in Instruction List (IL) [4, 15] or in Sequential Function Charts (SFC) [1]. This method is suitable not only for PLC programming languages: The Java PathFinder can be used to verify Java programs [11] by compiling them to PROMELA, the modeling language of SPIN. Bandera follows a similar approach and can analyze Java programs; it uses an optimizing compiler to translate them to either a Spin or NuSMV model [5]. This has also been done with programs written in C with a mixed-mode translation; the programs are compiled and combined with manually written system calls [8]. Care has to be taken that the extraction provides a faithful model of the system [10].

A remaining problem of the automatic model extraction [13] is that there is still the need for an expert to interpret the resulting trace in case of an error. This problem can be solved by creating highly specialized model checkers that can use the original program as model, e. g. in C [2] and in PLC/IL language [16].

Abstract interpretation has been used on modeling languages [9] and also on PLC/IL programs to perform range analysis [3].

Our approach uses model extraction on the robot programs, while the PLC task is generated. The resulting model consisting of multiple processes can then be verified. In case an error is found in the model, the resulting error-trace is

projected back to the original context. Example properties that can be checked are deadlocks, collisions and kill-switch violations.

3 Problem Statement

In this section we provide an overview of the system we want to verify. We will discuss the robot programming language and the interlock algorithm provided by the PLC.

An industrial robot system typically consists of up to 10 robots and a PLC which communicate over field bus. This distributed system shares common resources in the form of physical space that can be occupied by a single robot only. If more than one robot tries to use the same area at the same time, this behavior leads to a collision of the robots. To avoid this behavior the robots use a software locking mechanism with the PLC as central arbiter.

Before an industrial robot system's start of production (SOP) its behavior is tested to assure it is working correctly, e. g. it is free of collisions. This testing is tedious, expensive and can not guarantee the absence of errors.

Our goal is to provide a system that:

- does not require much effort to set up for a concrete industrial robot system
- can quickly verify the absence of certain programming errors
- is easy to use for a robot programmer, i. e. does not require expertise in formal verification

3.1 Robot Programming Language

The robot control language (VKRC) is an imperative, sequential language. Our compiler handles the core language, which is sufficient to detect the faults we want to find. This core language includes assignments, control flow commands and movement commands for the robotic arm. For simplicity we will refer to this subset as VKRC.

Variables The language only has boolean expressions. Depending on their type, variables can either store a boolean value or a boolean expression. Variables need not be declared, the set of variables is fixed. There are local variables, outgoing, incoming and symbolic variables; variables have a one-character prefix that determines their type followed by a natural number (see table 1). The outgoing and incoming variables are for communication with the PLC: Their contents are sent to the PLC by the robots runtime system, received from it respectively. This communication is completely transparent to the robot programmer. The symbolic variables can store an expression which gets evaluated when the symbolic variable is read. The kill-switch condition variable is a special purpose symbolic variable: When moving, the robot continuously checks this variable and triggers an emergency stop if the stored expression evaluates to false.

Variable	Semantics
`F`n	Local variable
`A`n	Outgoing variable
`E`n	Incoming variable (read-only)
`M`n	Symbolic variable (local)
`FB SPS`	Kill-switch condition (only on left-hand side)

Table 1. Types of variables in VKRC ($n \in \mathbb{N}^+$)

Statements The assignment statement is the predominant command of VKRC programs. If the variable on the left-hand side is of type local or outgoing, the expression on the right-hand side gets evaluated and the result is stored in the target variable. If the target variable is a symbolic variable, the expression itself is stored without binding any values of the variables used in the expression.

For moving the robotic arm, VKRC has several movement commands that will make the runtime system move the arm to the designated target coordinate. The different commands are used for specifying the movement trajectory with which the robotic arm is moved. The movement commands feature a plethora of arguments with which the movement can be further specified. As we do not take into account all available options that fine-tune the movement, the movement commands presented here are simplified.

The VKRC language has subroutine calls. VKRC subroutines do not take parameters nor do they return a value. The subroutine call is only available as a conditional call for which the programmer must provide a condition expression. Subroutines must not be called recursively.

Control flow statements provide a conditional jump similar to a C goto/label. Furthermore the repeat statement lets the programmer call a subroutine several times or unless a given condition is met. At last there is a blocking wait statement that will block until the given expression evaluates to true.

Command	Semantics
variable `=` *expression*	Assignment of expression to variable
`GOTO LABEL` *id* `=` *expression*	Conditional jump to label *id*
`LABEL` *id*	Target of a jump
subroutine `=` *expression*	Conditional subroutine call of *subroutine*
`REPEAT` *subroutine* `=` n `STOP =` *expression*	Call subroutine n-times or unless condition is met
`WARTE BIS` *expression*	Wait until *expression* is met
n `PTP`	Move the robot arm to position n (point-to-point)
n `LIN`	Move the robot arm to position n (linear)
n`,` m `CIRC`	Move the robot arm to position m via n (circular)

Table 2. Statements of the VKRC language

Boolean Expression	Semantics
expression `&` *expression*	Logical and of two expressions
expression `+` *expression*	Logical or of two expressions
`!` *expression*	Logical inversion of expression
`(` *expression* `)`	Brackets for changing order of precedence
variable	See table 1
`AUS`	false (literally *off*)
`EIN`	true (literally *on*)

Table 3. Boolean functions in VKRC

Communication The robot uses the input and output variables to communicate with the PLC; this communication is handled by a field bus. This communication is completely transparent to the robot programmer; to him the robot appears to have a shared memory with the PLC.

The PLC can either handle the incoming and outgoing variables of the robot with a PLC program, or just map the outgoing variables to incoming ones of another robot.

3.2 Interlock algorithm

The robot programming language does not have lock/unlock primitives. The robot programs rely on the PLC as an arbiter to guarantee that a lock can only be taken by a single robot. The actual lock is between two robots only; the PLC only manages the token which allows a robot to acquire a lock.

If a robot wants to lock a certain area, it first requests the corresponding token from the PLC. The PLC will hand out this token only to one robot at a time and will withdraw it only if the robot gives it back. A robot that has a token is allowed to take the actual lock – if it is not already taken. After obtaining the actual lock, the robot will give back the token. When it has left the corresponding area, the robot must unlock the area (see figure 1). The corresponding program code of the robot can be seen in figure 2.

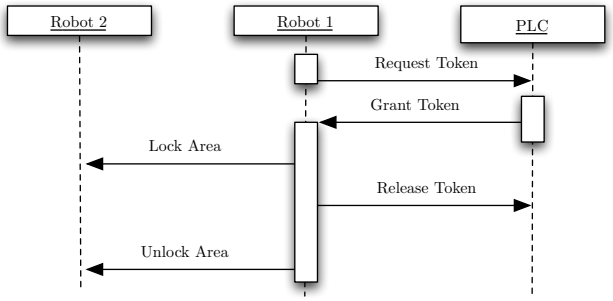

Fig. 1. Robot 1 successfully locks a shared working area

```
-- request token from the PLC
A81 = EIN
-- block until area is free and token is granted
WARTE BIS E41 & E81
-- lock area
A41 = AUS
-- return token
A81 = AUS

-- may now operate in shared working area
...

-- unlock area
A41 = EIN
```

Fig. 2. Robot program code to lock area 41/81

4 Solution

Our tool chain (see Figure 3) allows a robot programmer to verify programs without requiring any knowledge in model checking. The VKRC compiler translates the robot programs into PROMELA code. Together with the PLC logic and the error description, this forms our model. This model is then verified with SPIN. If an error is found, we transform the trace back into a VKRC trace.

4.1 VKRC to PROMELA compiler

The VKRC compiler translates one robot program into a PROMELA process and a corresponding variable declaration list. The compiler is written in OCaml. It uses a classic lexer/parser front-end that first generates an abstract syntax tree (AST) from which a control flow graph (CFG) is built. The statements in the CFG are then translated from VKRC to PROMELA. Eventually the PROMELA CFG is used to generate PROMELA code.

The compiler is used to translate each robot program into a PROMELA process. The generated processes and variable declaration lists can then be combined with the process of the PLC to form a model of the industrial robot system.

Locations The lexer passes the location information about the origin of lexer tokens to the parser. The parser embeds this information into each statement of the AST, so that we know the corresponding file and line of each statement. This information is kept for each statement during all compiler phases. When the compiler backend generates PROMELA code, it not only creates PROMELA statements for the corresponding logic, but also print-statements. They print a

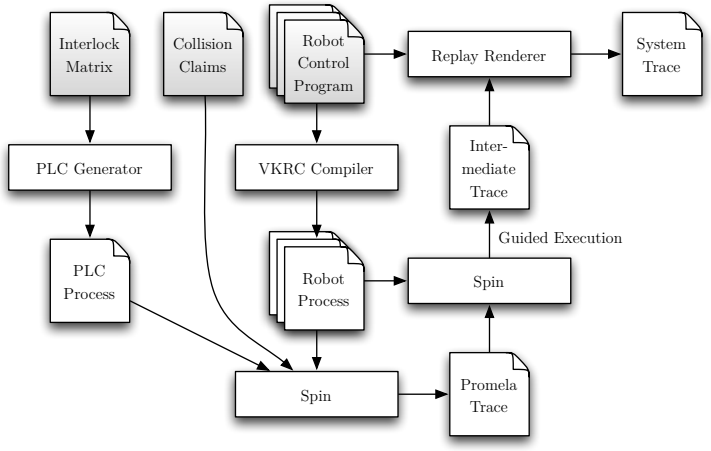

Fig. 3. Architectural Overview

line containing the location information and additional information about the values of the variables used in the statement (see table 4).

This allows the transformation of the SPIN trace into a VKRC trace by using the guided execution capability of SPIN with the error trace. During guided execution, SPIN executes the program in such a way that the fault is reached in the formal model. The execution of the print statements along this path generates a trace that can be directly mapped to the VKRC programs.

Loop unrolling, function inlining The VKRC language features a repeat-statement that calls a subroutine a fixed number of times – or until an additional condition is not met. To avoid introducing an additional integer for the loop counter and to keep the control flow graph as simple as possible, these loops are always unrolled. As the number of times the subroutine is called is a fixed integer value, the loop unrolling always succeeds in completely removing the loop. The unrolling is performed during the translation from the AST to the CFG.

PROMELA does not have functions or subroutines whereas VKRC has subroutines. To solve this mapping problem, the compiler inlines all subroutine calls. This approach is feasible as recursion is not allowed in VKRC. The VKRC subroutines do not take parameters which makes this inlining process straight forward: The compiler replaces the calls to subroutines by their body until the program is free of subroutine calls.

With all subroutine calls removed, the compiler also does not need to perform inter-function analysis during the next step. Inlining is performed with all subroutines being available in the VKRC CFG form.

Abstract Interpretation VKRC has symbolic variables that can store a boolean expression while PROMELA does not have a similar feature. The com-

piler uses abstract interpretation [6] to remove the need of the symbolic variables from the source language. If this is successful, the mapping of the resulting VKRC program to PROMELA is easy.

The abstract interpretation is performed on the VKRC CFG; The compiler can perform *constant evaluation*, *copy propagation* and *dead code elimination*: The goal is to remove the assignments to the symbolic variables. First the compiler performs constant evaluation to simplify the CFG. With this simplification in place, the copy propagation optimization can determine the value of the symbolic variables in the statements they are used in. After the copy propagation, the symbolic variables occuring in right-hand side values are replaced by their value. This step removes the usage of the symbolic variables in right-hand side values.

This in turn makes the assignments to the symbolic variables unnecessary. The dead code analysis can now determine which assignments are unnecessary and remove them. After this step, no symbolic variables should appear on left-hand side values anymore either.

The abstract interpretation allows us to get rid of the symbolic variables, so that no PROMELA code needs to get generated for them. Unfortunately, this method might not work for all VKRC programs. Examples are goto loops including symbolic variables. However, the method worked for all robot control programs we analyzed as such a construction contradicts typical programming standards. If this method does not succeed, this could well be a programming error.

The compiler can be told to only perform the optimizations on the symbolic variables instead of all variables. This can lead to minor differences in the time required to verify the model as can be seen in section 5.

Movement commands The robot control programs have movement commands. These commands will make the robotic arm move to the given position. The positions are given as integer numbers in the robot control program; a lookup-table exists where the actual movement coordinates are stored. We know from the CAD program the industrial robot system is planned in, which combinations of robots in which positions can lead to collisions.

The VKRC compiler translates the movement commands to assignments to a global position variable. The target points of the movements are enumerated already, so we use these numbers as right-hand side value of the assignment.

During movement, the robot continuously checks the kill-switch variable. This symbolic variable is evaluated and if returns false, will make the robot perform an emergency stop. To map this behavior, the compiler inserts an assert statement on the value of the kill-switch variable at the position of the movement command. This way the model checker will find a violation of this property.

4.2 PLC generator

To verify the locking mechanism, a model of the PLC is required. We have written a code generator that creates this model based on the interlock configuration.

VKRC	PROMELA
`A23 = F17 + E12`	`atomic{ A23=F17||E12; printf("UP1.vkrc::24") }`
`GOTO LABEL 4 = EIN`	*via control flow graph*
`LABEL 4`	*via control flow graph*
`MAKRO20 = EIN`	*inline contents of subroutine*
`REPEAT MAKRO20 = 5 STOP = E9`	*unroll loop (5x), then inline contents*
`WARTE BIS E78`	`atomic{ (E78); printf("MAKRO20.vkrc::25") }`
`7 PTP`	`atomic{ location=7; printf("UP1.vkrc::23") }`
`19 LIN`	`atomic{ location=19; printf("UP1.vkrc::34") }`
`20, 21 CIRC`	`atomic{ location=21; printf("UP2.vkrc::72") }`

Table 4. Translation of VKRC statements, examples

The PLC is modeled as a process that grants and removes locking-tokens based on the truth table in table 5. For every available lock between two robots, the PLC process features one set of entries. One lock is always exclusively between two robots. The PLC therefore has to take into account all possible states of token-requested and token-granted for exactly two robots. We generate a PLC process for the final model in PROMELA from this table.

The PLC process consists of an event loop and decides non-deterministically which token to grant or withdraw; the two robots n and m can request their tokens T_n and T_m by sending requests R_n and R_m. For every lock in the system, we generate 8 lines for the PLC process: The truth table 5 has 17 rows; the row with both robots requesting the locking token has a non-deterministic result and for this reason appears twice. For the error-lines, where both robots have been granted the locking token, no code needs to be generated as they are unreachable. The lines for which the PLC process does not need to take any action, can also be omitted.

The actual locking bits are exchanged directly between the robots. A robot is only allowed to claim an area if it owns the locking token and the area is not already locked. The locking token is not required to release an area.

The robots can not directly communicate with each other, but all communication is handled via the PLC. In the original system, the PLC performs a mapping from the corresponding output variables of one robot to the input variables of the partner robot. We model this mapping by renaming the input variables to the output variables they are connected to.

4.3 Fault description

We are searching for three classes of faults:

- Deadlocks
- Collisions
- Kill-switch violations

Deadlocks are searched for by SPIN automatically. We only need to specify that the PLC process does not have to terminate.

R_n	R_m	T_n	T_m	T'_n	T'_m	
0	0	0	0	-	-	
0	0	0	1	-	0	remove token from m
0	0	1	0	0	-	remove token from n
0	0	1	1	error		the token was granted to both robots
0	1	0	0	0	1	grant token to m
0	1	0	1	-	-	
0	1	1	0	0	1	
0	1	1	1	error		the token was granted to both robots
1	0	0	0	1	-	grant token to n
1	0	0	1	1	0	grant token to n, remove from m
1	0	1	0	-	-	
1	0	1	1	error		the token was granted to both robots
1	1	0	0	1	-	grant token to n
1	1	0	0	-	1	grant token to m
1	1	0	1	-	-	
1	1	1	0	-	-	
1	1	1	1	error		the token was granted to both robots

Table 5. Truth table of how the PLC grants and removes tokens T based on requests R

Collisions are found via the movement commands. We translate a movement command to an assignment of the unique movement point number to the `position` variable of the corresponding robot. The content of this variable specifies the coordinate, where the robot arm is heading. From the 3D model of the industrial robot system, we know which movement combinations of which robots can cause collisions. We use this information to formulate a never claim: Combinations of locations of two robots that would lead to a potential collision must not occur

The user has to supply a matrix of possible collisions based on the movement points of the robot programs. From this matrix we generate claims in LTL; for every possible collision between two robots – robot n at position i and robot m at position k – we add the claim:

$$\Box\neg((\text{position}_\text{n} = i) \wedge (\text{position}_\text{m} = k))$$

Kill-switch violations are directly generated by the compiler as assertions in the PROMELA process. While in movement the robot continuously checks the `FB SPS` variable for its contents. If this is false, the robot will trigger an emergency stop. This kind of emergency situation must not occur during normal operation; hence, it is considered as an error in the model.

During translation the compiler determines the value of the `FB SPS` variable at each movement command. A movement command is then translated to the assignment to the position and a preceding PROMELA-assert statement. This assertion guarantees the expression in the kill-switch variable to be true during movement.

4.4 Putting the model together

To create a model for the entire industrial robot system, we combine the generated files: We create a main PROMELA file that first includes the variable declaration files of all robots. It then includes the files of the robot processes and of the PLC process. The verification engineer then has to add a never claim for the potential collision combinations. Finally we write a main process that initializes certain variables and then starts all robot processes and the PLC process.

As of now, we have several devices that are missing from our model, e. g. welding devices. To allow the robot program to work as intended, we manually set the variables that represent the feedback from those devices to their desired value. This means these events always succeed and the behavior of the system with these events failing is not taken into account.

4.5 Replay trace

If the model checker finds a counter-example, it generates a trace leading to this error. This trace can be replayed in *guided execution*, executing the PROMELA program in such a way, that it leads to the found violation.

When executed in this way, the model checker will also execute the `printf` statements that were inserted by the compiler. The back-end of the compiler generates an accompanying `printf` statement for every statement generated. For a VKRC assignment, a PROMELA assignment is generated which is placed together with a `printf` statement inside an atomic block. This block then outputs the original location of the VKRC statement and the values of the variables used in the original statement.

This output is now a trace we can directly relate to the original robot control programs. The replay renderer takes this VKRC trace and generates a replay based on the robot programs. This replay shows the execution of the robot programs according to our trace; it displays the programs in the same way as in a teach panel, in a way the robot programmers are accustomed to. Variables in the VKRC program get colored depending on their current value.

The current replay can display two robot programs side by side, the current statement and the value of the variables. It consists of several HTML files so that a robot programmer does not need to install additional software to view the replay.

This replay allows a robot programmer to understand the way an error can occur in the system and help him find and remove the source of the problem. A screenshot of the current prototype can be seen in figure 4.

5 Evaluation

We conducted verification runs on a training station and on several recent car-body stations. The training station has 2 robots, while the car-body stations have 9.

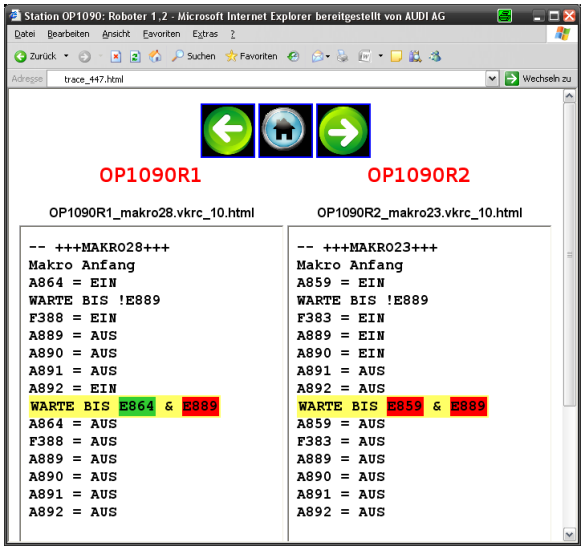

Fig. 4. Trace replay at the final deadlock situation between two robots

We found a deadlock in the robot control programs of the training station. A robot programmer was able to quickly track the error with our replay tool and to remove the source of the error: A missing variable initialization. The results of the benchmarks on the training station (see table 6) were as expected: With the deadlock in place, the error was found with minimal amount of CPU-time. Without the deadlock, the verification run still finished in under 1 second. The activation of the optional compiler optimizations reduced the CPU-time required by 5%.

After the training station, we tried to verify several recent car-body stations with 9 robots each. We were able to find a deadlock in one of the welding stations. The preparation time to set up the model of the system was about 3 man-hours. With the deadlock in place, the verification run took well under one second. After removing the source of the deadlock, the verification itself took about 2 minutes. For this setup the optional compiler optimizations made the verification 11% more expensive in terms of CPU-time.

The deadlock of the welding station was due to a protocol error in the request of the locking tokens. The robot control programs requested two tokens before waiting for the arrival of the first one. As can be seen in figure 5, this can lead to a rare deadlock. The request for token A from robot *1* arrives first, which the PLC grants. Then the requests for token A and B from robot *2* arrive. The PLC has already handed out token A but token B is given to robot *2*. As both robots will block waiting for the arrival of both tokens, a classic deadlock situation occurs (see figure 5). To resolve this problem, both robots need to first wait for the arrival of token A before requesting token B.

# of robots	2				9			
# of processes (robot/PLC)	3 (2/1)				10 (9/1)			
# of Variables (bit/byte)	214 (210/4)				936 (918/18)			
Contains deadlock	yes		no		yes		no	
Compiler optimizations	off	on	off	on	off	on	off	on
Number of transitions	179	179	45083	42742	2677	2677	24891214	30837984
CPU-time [sec]	0.030	0.061	0.077	0.140	0.061	0.108	119.640	132.718
Memory footprint [MByte]	2.501	2.501	3.184	2.989	4.161	3.868	909.786	909.883

Table 6. Benchmarks of analyzed industrial robot systems

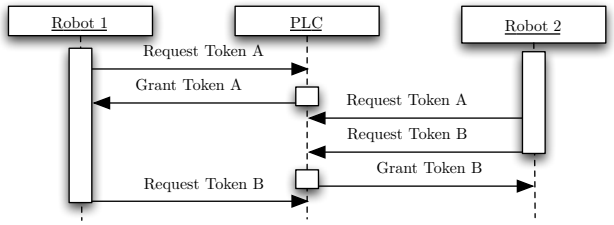

Fig. 5. Robot 1 and 2 deadlock each other trying to obtain two tokens concurrently

6 Conclusion and future work

This paper proposed the use of model checking techniques to verify the correctness of industrial robot system with respect to certain properties. We presented a compiler to transform robot programs into extended PROMELA models. Furthermore the tool allows to map an error-trace created by SPIN back into the original context.

Using the suggested approach we could validate original robot control programs from Audi car body welding stations. In this process we found a protocol error in the robot programs that could eventually lead to a deadlock. Based on the generated error-trace it was possible to find and remove the error. We could show that model checking is a viable option for analyzing the control programs of large industrial robot system even with 9 robots.

In contrast to the simulation-based approach which is currently state-of-the-art in industry, the approach can guarantee the correctness with respect to certain properties, such as freedom of deadlocks and absence of collisions. The required effort to set up the modeling tool chain is also considerably smaller than the effort currently spent in testing these properties.

We are planning several improvements for the current tools to increase their applicability:

- Multi-language support; we want to analyze industrial robot system using robots from other manufacturers that use different programming languages.
- Higher degree of automation in setup process; if the barrier to use model checking can be lowered even further, the applicability will be higher.

- Replay-renderer for an arbitrary number of robots/processes
- Include support for more devices; we want to analyze if the robot programs handle external errors correctly, e. g. a welding problem

The case study presented in this paper showed that for current industrial robot system the performance scales well. However if further details are added to the formal model state space explosion can get a problem. Hence future work must also take into account techniques to limit the state space.

References

1. Nanette Bauer, Sebastian Engell, Ralf Huuck, Ben Lukoschus, and Olaf Stursberg, *Stursberg: Verification of PLC Programs given as Sequential Function Charts*, In: Integration of Software Specification Techniques for Applications in Eng., Springer, LNCS, 2004, pp. 517–540.
2. Dirk Beyer, Thomas Henzinger, Ranjit Jhala, and Rupak Majumdar, *The software model checker BLAST*, International Journal on Software Tools for Technology Transfer (STTT) **9** (2007), 505–525, 10.1007/s10009-007-0044-z.
3. Sébastien Bornot, Ralf Huuck, Yassine Lakhnech, and Ben Lukoschus, *Utilizing static analysis for programmable logic controllers*, ADPM 2000: The 4th International Conference on Automation of Mixed Processes: Hybrid Dynamic Systems, 2000, pp. 183–187.
4. G. Canet, S. Couffin, J. j. Lesage, A. Petit, and Ph. Schnoebelen, *Towards the automatic verification of PLC programs written in Instruction List*, IEEE International Conference on Systems, Man and Cybernetics, 2000, pp. 2449–2454.
5. James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Pasareanu, and Hongjun Zheng, *Bandera: Extracting Finite-state Models from Java Source Code*, Proceedings of the 22nd international conference on Software engineering, ACM Press, 2000, pp. 439–448.
6. Patrick Cousot and Radhia Cousot, *Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints*, Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages (New York, NY, USA), POPL '77, ACM, 1977, pp. 238–252.
7. Patrick Cousot and Radhia Cousot, *Verification of embedded software: Problems and perspectives*, Proceedings of the 1st International Workshop on Embedded Software (EMSOFT), USA. LNCS 2211, Springer-Verlag, 2001, pp. 97–113.
8. Pedro de la Cámara, Marìa del Mar Gallardo, and Pedro Merino, *Model extraction for ARINC 653 based avionics software*, Proceedings of the 14th international SPIN conference on Model checking software (Berlin, Heidelberg), Springer-Verlag, 2007, pp. 243–262.
9. Yifei Dong and C. R. Ramakrishnan, *An optimizing compiler for efficient model checking*, Proceedings of the IFIP TC6 WG6.1 Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols (FORTE XII) and Protocol Specification, Testing and Verification (PSTV XIX) (Deventer, The Netherlands, The Netherlands), FORTE XII / PSTV XIX '99, Kluwer, B.V., 1999, pp. 241–256.
10. Lucio Mauro Duarte, Jeff Kramer, and Sebastian Uchitel, *Towards faithful model extraction based on contexts*, Proceedings of the Theory and practice of software, 11th international conference on Fundamental approaches to software engineering (Berlin, Heidelberg), FASE'08/ETAPS'08, Springer-Verlag, 2008, pp. 101–115.

11. Klaus Havelund, *Java PathFinder, A Translator from Java to Promela*, Proceedings of the 5th and 6th International SPIN Workshops on Theoretical and Practical Aspects of SPIN Model Checking (London, UK), Springer-Verlag, 1999, pp. 152–.
12. Gerard J. Holzmann, *The Model Checker SPIN*, IEEE Transactions on Software Engineering **23** (1997), 279–295.
13. Gerard J. Holzmann and Margaret H. Smith, *An automated verification method for distributed systems software based on model extraction*, IEEE Transactions on Software Engineering **28** (2002), 364–377.
14. Mauro Mazzolini, Alessandro Brusaferri, and Emanuele Carpanzano, *Model-checking based verification approach for advanced industrial automation solutions*, Emerging Technologies and Factory Automation (ETFA), 2010, pp. 1–8.
15. Olivera Pavlovic, Ralf Pinger, and Mail Kollmann, *Automated formal verification of PLC programs written in IL*, Conference on Automated Deduction (CADE), 2007, pp. 152–163.
16. Bastian Schlich, Jörg Brauer, Jörg Wernerus, and Stefan Kowalewski, *Direct model checking of PLC programs in IL*, 2nd IFAC Workshop on Dependable Control of Discrete Systems (DCDS) (Bari, Italy), 2009.
17. Helmut Seidl and Vesal Vojdani, *Region analysis for race detection*, Proceedings of the 16th International Symposium on Static Analysis (Berlin, Heidelberg), SAS '09, Springer-Verlag, 2009, pp. 171–187.
18. Hai Wan, Gang Chen, Xiaoyu Song, and Ming Gu, *Formalization and Verification of PLC Timers in Coq*, Proceedings of the 2009 33rd Annual IEEE International Computer Software and Applications Conference - Volume 01 (Washington, DC, USA), IEEE Computer Society, 2009, pp. 315–323.
19. Mohammed Bani Younis and Georg Frey, *Formalization of existing PLC programs: A survey*, Proceedings of Computing Engineering in Systems Applications (CESA) (Lille, France), July 2003.