Formal Verification of ST Programs using CSP

Qibin Zhou¹, Fangda Cai²*, Yang Yang² and Changshun Wu³

¹ Shanghai Jianqiao University, Shanghai, 201306, China
² East China Normal University, Shanghai, 200062, China
³ Shanghai Formal-tech Information Technology Co., Ltd., Shanghai, 200333, China

*Corresponding author’s e-mail: fdcai@sei.ecnu.edu.cn

Abstract. Programmable Logic Controllers (PLCs) are embedded devices that are widely used in industrial control systems. PLC programs are written in special languages that are defined in the IEC 61131-3 standard, which includes two textual and three graphical programming languages. Structured text (ST) language is one of IEC 61131-3 standards, which is a high-level text language similar to PASCAL language. ST language features the following advantages: freely programming style, compact structure and strong data processing capabilities. Using ST language can be more convenient to realize the complex control programs. In this paper, we research and discuss formal verification of PLC program based on ST language, and show the conversion rules from ST language to the formal modeling language named CSP# (short for Communicating Sequential Programs). Finally, a case study is presented to demonstrate the procedure that PLC programs written in ST language are translated into CSP#, consequently, and the result of verification is given using the Process Analysis Toolkit (PAT).

1. Introduction

PLC (Programmable Logic Controller) [1] is widely used in industrial control systems from small applications such as elevator control systems to larger applications such as fission control in nuclear power plants. PLC programming languages are defined by the IEC 61131-3 [2] standard, which is the first international standard for a programming language of industrial control systems. This standard provides five different programming languages, namely, IL (instruction list), ST (structured text), LD (ladder diagram), FBD (function block diagram), and SFC (sequential function chart). Among them, ST is a high-level text language with an easy programming style and good portability, which can greatly improve the programming efficiency for PLC. ST programs are widely accepted and supported by various industrial manufacturers, including some well-known companies, such as Beckhoff and 3S (Smart software solution GmbH). With the growing complexity and reduced development time for control systems, the demand for high-quality software and safety-critical application need validation procedures.

Traditionally, verifications of PLCs programs are carried out through exhaustive testing. This way is very time-consuming and may not always detect all bugs. In recent years, the PLC community in industry as well as in academia has started to look for cost-effective and reliable solutions for bug detection. Formal verification is a proofing method that uses mathematical methods to rigorously prove the correctness of programs. It is often used in developing high-quality software for critical applications. In recent years, there have been many studies on the verification of PLC programs. PLC programs are mapped into formal models, which could be verified by some available checker tools, such as UPPAAL,
In addition, there is also a few research on the intermediate language of PLC. The research shows that ST may serve as a pivot to unify the multiple PLC languages [3]. Previous research work focused on the safety of PLC programs, and rarely considered race between multiple tasks or POUs.

CSP# is a formal modeling language that is based on the classic process algebra Communicating Sequential Processes (CSP) [4]. CSP enjoys unique advantages in interaction and concurrency, which can be applied to detect concurrent and real-time issue. CSP# maximally keeps the original CSP including high-level modeling operators like choices, parallel composition, interleaving and so on. In addition, CSP module (written in CSP#) is supported by PAT (Process Analysis Toolkit) [5, 6, 7]. PAT has a modularized feature and can provide the function of simulation and verification. In this paper, we use model checker PAT to detect the race of PLC programs with multiple POUs.

To perform the formal verification of ST programs, we provide a set of conversion rules from ST programs to CSP#, then, the verification of programs is conducted by PAT. The main contributions of this paper are as the following:

1. We propose the mapping rules to implement conversion from ST programs to CSP# programs, which includes mapping rules of variable declaration and program body. Basic elements of ST are mapped CSP# one-to-one. In addition, we set mappings in a top level of program structure.

2. Furthermore, we present a case study for conversion and verification to demonstrate the procedure in detail. Finally, the result of verification is shown and also analyzed according to the counterexample path given by PAT.

The rest of this paper is organized as the following: Section 2 discusses related work. Section 3 introduces the background. Section 4 describes the conversion rules between PLC programs based on ST and CSP#. Section 5 gives a case study to demonstrate the procedure of conversion and verification. Section 6 summarizes our work.

2. Related Work

There is numerous research on transformation between PLC programming languages, which has been studied in the literature [3, 8, 9, 10]. The methods of transformation from LD to ST and LD to IL were given in the references [8] and [9] respectively. Lopes V E E. [10] developed a software tool to convert LD programs to SFC programs. Beremiz [11] is a free and open-source IEC 61131-3 automation IDE. Its PLC platform can support IEC61131-3 five development languages. Its PLCOpen editor has a built-in export filter that converts graphical languages to their equivalent textual, such as, LD and FBD can be converted into ST. Darvas et al. [3] adopted an intermediate model (IM) that ST was as a pivot to represent all five standard PLC languages. The PLC programs (SCL, SFC in Siemens) were transformed to IM. Finally, IM was translated to existing model checkers.

Moreover, there are some related studies on verification for PLC programs. Frey G et al. [12] reviewed the application of formal methods in PLC programs. Some approaches found in the literature [13, 14, 15, 16, 17,18, 19, 20, 21, 22, 23, 24] are limited to verification for a kind of PLC programming language. For example, Bender et al. [14] provided a metamodel for a subset of the LD language and applied a model-driven approach for formal verification of LD programs through Time Petri Net. In the literature [20], the author proposed a formal semantics of IL language, which covers a subset of IL. In this semantics context, the theorem prover Coq was used to prove some safety properties of a simple example of PLC programs. Huuck et al. [24] presented a semantic definition for safe SFCs and detected whether an SFC is safe or not. All of these approaches worked in a similar fashion by verifying only one of the five languages. Beyond that, as mentioned previously, Darvas et al. adopted a new idea verifying the PLC programs through an IM. In this paper, we try to convert the PLC program based on ST to CSP#, and use the concurrency feature of CSP# to detect whether there will be bugs when the program is concurrent.

3. Background

This section gives a brief overview of the concepts and terms relevant to this work, with a brief discussion of PLC programs and CSP#.
3.1. An overview of PLC programs
A PLC usually consists of a CPU (central processing unit), a memory, input and output devices. PLC has a characteristic execution mode. A PLC program is executed in a permanent loop. As is shown in figure 1(a), each scan cycle consists of 3 phases [25]: 1) It reads the input and stores the input in a specific area of the memory. 2) It executes PLC code response to the input, which subsequently generates output values. 3) It passes the output values to the actuators. The time to complete an entire cycle is known as the scan time.

The top level of software structure is called a configuration. Each configuration can contain one or more resources. Each resource can contain one or more declared tasks. A task can be used to control how the program is executed. A PLC program consists of encapsulated blocks of code called POUs (Program Organization Units) defined by the IEC 61131-3 standard. There are three types of POUs: Functions, Function Blocks, and Programs. The presentation in figure 1(b) shows one configuration consisting of one resource. A POU is programmed using either the textual programming languages IL and ST or the graphical languages LD, FBD and SFC. The following discussion mainly focuses on ST.

A POU mainly consists of variable declaration and program body. The part of variable declaration involves common elements defined in IEC 61131-3 standard. The declaration part of variables may contain various VAR sections. The program body can be written in ST, whose sketch is shown in figure 2(a). An ST program is constructed by using statements. Statements are composed of expressions and keywords. An expressions is a combination of operands and ST operators. There are four types of statements, namely assignment statement, function or function block control statement, selection statement and loop statement. Some common keywords in ST include “:=” “IF” “CASE” “FOR” “WHILE” “REPEAT” “EXIT” “RETURN” and so on. ST does not include a jump instruction (GOTO). Since jump instructions encourage the use of non-structured programs, it is not adopted. A simple example of ST is shown in figure 2(b).
3.2. Brief introduction to CSP#

CSP is a formal language that has been widely applied in industrial settings for the efficient specification and reliable verification of concurrency of systems. The application of CSP for formal verification has been extensively studied in the existing literature [26, 27, 28]. CSP# is the input language of CSP Module in PAT, which is mainly influenced by the classic CSP and extends CSP with various useful language features. CSP# in PAT supports low-level constructs like variables, arrays, if-then-else, while, etc. The relevant syntax structures of CSP# are listed as below:

(1) Global Definitions

This part mainly includes model name, constants, variables, channels, model inclusion and so on. Model name is optional and used as an ID for the simulator. A global constant is defined using keyword #define. Constant enumeration can be defined using keyword enum. A global variable is defined using keyword var. A channel is declared using keyword channel. Model inclusion supports importing other files from the local folder by using the include keyword.

(2) Process Definitions

\[ \text{Process ::= Stop} \mid \text{Skip} \mid e_0 \rightarrow P \mid c! \exp \rightarrow P \mid c? x \rightarrow P \mid P \parallel Q \mid P; Q \mid P \leftrightarrow b \leftrightarrow Q \mid P[]Q \]

- \( \text{Stop} \) denotes that process does absolutely nothing.
- \( \text{Skip} \) denotes that process terminates.
- \( e_0 \rightarrow P \) describes a process which performs an event \( e_0 \) first and then behaves as specified by process \( P \). An event can be attached with statements.
- \( c! \exp \rightarrow P \) and \( c? x \rightarrow P \) are used to describe that processes communicate with each other through channels. \( c \) is a channel which must be declared before it is used. \( c! \exp \rightarrow P \) represents sending a message and \( c? x \rightarrow P \) represents receiving a message.
- \( P \parallel Q \) expresses parallel composition of two processes with barrier synchronization.
- \( P; Q \) gives a sequential execution where \( P \) and \( Q \) are processes. In execution procedure, \( P \) starts first and \( Q \) starts only when \( P \) has finished.
- \( P \leftrightarrow b \leftrightarrow Q \) shows a conditional choice. If \( b \) evaluates to be true, then \( P \) executes, otherwise, \( Q \) executes.
- \( P[]Q \) states that either \( P \) or \( Q \) may execute. If \( P \) performs an event first, then \( P \) takes control. Otherwise, \( Q \) takes control.

PAT also supports the classic if-then-else. The relationship between \( P \leftrightarrow b \leftrightarrow Q \) and if-then-else is equivalent in semantics. Its specific structure is as follows:

\[ \text{if}\{\text{cond1}\}\{P\}\text{else if}\{\text{cond2}\}\{Q\}\text{else}\{M\} \]

(3) Assertions

An assertion is defined using keyword #assert. In PAT, a number of different assertions are supported including the full set of Linear Temporal Logic (LTL). For example, given a process \( P() \), the following assertion asks whether \( P() \) satisfies the LTL formula \( F \).

\[ \#assert P() \models F; \]

4. ST to CSP# Transformation

The transformation workflow from the PLC programming languages defined by the IEC 61131-3 standard to the formal language CSP# is depicted in figure 3. The language conversion in the IEC 61131-3 standard may be realized with the help of Beremiz. Our work mainly focuses on the conversion from ST to CSP# and verification using PAT. Next, we will give the mapping rules from ST to CSP#.
4.1. Variable Declaration Mapping

The declaration part of IEC 61131-3 standard contains different types of variables, including input variables, output variables, input and output variables, etc., which are defined by the keyword “VAR_INPUT” “VAR_OUTPUT” “VAR_IN_OUT”. Among them, variables have a variety of data types, which can be basic data types (BOOL, INT, REAL) or other data types (array, enumeration, structure). The variables in CSP# are divided into global constants, global variables, array variables and channel variables. The data types only support boolean and integer. In addition, PAT tool supports custom data structure types and function libraries through C#. Due to the differences between languages, some restrictions need to be made in the partial transformation of variable declaration. In this paper, only boolean and integer data types are considered. In most PLC systems, due to the characteristics of periodic execution, time-consuming calculations such as floating-point type are usually calculated by special equipment and communicated with PLC.

The mapping rules from IEC 61131-3 standard variables to CSP# variables are as follows:

- **Rule 1** Constant Mapping

  ST:  
  ```
  𝐴𝑛𝑦𝑉𝐴𝑅  
  (varName : dataType := constValue;)
  END_VAR
  ```

  CSP#:  
  ```
  #define (varName constValue;)
  ```

- **Rule 2** Variable Mapping

  ST:  
  ```
  𝐴𝑛𝑦𝑉𝐴𝑅  
  (varName : dataType := initialValue ;)
  END_VAR
  ```

  CSP#:  
  ```
  var (varName := initialValue ;)
  ```

Where `AnyVAR` is one of the keywords “VAR\VAR_INPUT\VAR_OUTPUT\VAR_IN_OUT”. `()` indicates one or more occurrences of content in brackets. In CSP#, `var` is defined without distinguishing data types. When converting, the default initial value is 0 for integer and false for boolean.

4.2. Program Body Mapping

An ST program is mainly composed of statements. The basic elements of statements are expressions and keywords. Statements are divided into assignment statements, selection statements, iteration statements, etc. CSP# takes the process as the basic execution unit and supports the constructs of statement regarded as “syntactic sugar”. In this paper, a POU in ST program is equivalently mapped to a CSP module, which contains one or more processes. The process can contain different statements. CSP# supports if-statement and while-statement. The if-statement also can be used to represent the while-statement equivalently. The mapping rules of the basic elements in ST are given as follows.

1. Common Operators Mapping

   The mapping of operators from ST to CSP# is shown in the Table 1.

   ![Table 1. Common operators mapping.](image)

   (2) Assignment Statement Mapping

   Assignment statements in ST are divided into expression assignment and function assignment, and the mapping rules are as follows:

   - **Rule 3** Expression Assignment Mapping

     ST:  
     ```
     varName := expression;
     ```

     CSP#:  
     ```
     varName = expression;
     ```
Rule 4  Function Assignment Mapping
ST:  \( \text{varName} := \text{funcName}(\text{paramList}); \)
CSP#:  \( \text{varName} = \text{call(funcName, paramList}); \)

(3) Selection Statement Mapping
Selection statements include IF statements and CASE statements, and the mapping rules are as follows:

- Rule 5  IF Statement Mapping
  a) transform “IF” keyword into “if(“;
  b) transform “THEN” keyword into “)”;\n  c) transform “ELSF” keyword into “)”;\n  d) transform “ELSE” keyword into “)else{”;
  e) transform “END_IF” keyword into “)”.

- Rule 6  CASE Statement Mapping
  ST:  CASE \( \text{cond} \) OF
      Value1,Value2 : Statements1;
      Value3 : Statements3;
      ...
      ValueN : StatementsN;
      ELSE StatementsN + 1;
      END_CASE;
  CSP#:  \( \text{if}((\text{cond} == \text{Value1} || \text{cond} == \text{Value2})\{\text{Statements1};\)
         \( \text{else if}(\text{cond} == \text{Value3})\{\text{Statements3};\)
         \( \text{else if}(\ldots\ldots)\{\text{StatementsN};\)
         \( \text{else}\{\text{StatementsN + 1;}\} \);\n
(4) Iteration Statement Mapping
The iteration statements include FOR loop, WHILE loop and REPEAT loop. These statements are also converted to if statements in CSP#, the mapping rules are listed respectively below.

- Rule 7  FOR Statement Mapping
  ST:    FOR \( \text{ControlVar} := \text{FormExpr} \) TO \( \text{ToExpr} \)
        \( \text{IncExpr} \) DO
        \( \text{Statements;} \)
        END_FOR;
  CSP#:  \( \text{ForInit()} = \{ \text{ControlVar} = \text{FromExpr}; \} \)
         \( \rightarrow \text{ForBody()}\);
         \( \text{ForBody()} = \text{if} (\text{ControlVar} <= \text{ToExpr})\{
         \{ \text{ControlVar} = \text{ControlVar + IncExpr;} \}
         \rightarrow \text{ForBody()}\);\n
- Rule 8  WHILE Statement Mapping
  ST:    WHILE \( \text{CondExpr} \) DO
          \( \text{Statements;} \)
          END_WHILE;
  CSP#:  \( \text{WhileProcess()} = \text{if} (\text{CondExpr} == \text{true})\{
          \{ \text{statements;} \} \rightarrow \)
Rule 9: REPEAT Statement Mapping

ST:
```
REPEAT Statements;
UNTIL CondExpr
END_REPEAT;
```

CSP#:
```
RepeatInit() = \{ Statements; \} → RepeatProcess();
RepeatProcess() = if (CondExpr != true){
    \{ Statements; \} → RepeatProcess()
};
```

The above rules mainly cover the transformations of basic language elements in ST. And the combination rules of them are as below.

(5) Combination Mapping Rules

In general, ST program is composed of different statements in order. During the transformation, we treat each statement as an event, labeled $e_{\text{statement}_i}$. For the statement block of sequential execution, we map it to a process, in which each statement acts as an event to execute according to the sequence of statements. It looks like $e_{\text{statement}_i} \rightarrow e_{\text{statement}_{i+1}}$. When converting IF statements, the process $\text{Skip}$ is added to indicate that process terminates, as shown in figure 4. For iterative statements, they are mapped to related processes according to Rule 7, Rule 8, and Rule 9. Processes are executed in a certain relationship.

![Figure 4. Combination Mapping.](image)

4.3. Top Level Mapping

Considering the software model of IEC 61131-3 standard, a POU in ST programs is mapped to a CSP# module (i.e. xxx.csp file). Different POUs can call each other through the “# include” keyword. Global shared variables are mapped to the main file of the CSP module. The main process calls other POUs' processes according to the execution relationship between programs. In the next section, we will give a specific case study to show these transformations.

5. Verification Case Study

In this section, we present a case study of transformation and verification between multiple POUs. Each POU is mapped to a CSP module, and the competing relationships between the modules are validated by importing files between the modules. Figure 5 shows a simplified version of the ST program for robot motion control, which contains two POUs, namely ProgA and ProgB. Among them, the program ProgA is responsible for reading data from obstacle detection sensors and changing the direction of robot motion when encountering obstacles. The program ProgB is responsible for controlling the
forward motion of the robot. There are two global variables in the whole control program, namely, Obstacle and Forward.

![ST programs for robot motion control](image1)

Using the previous conversion rules, the equivalent CSP# programs are as shown in figure 6. The two POUs are mapped into two CSP modules respectively. They are concurrently executed by the main CSP module.

![Equivalent CSP# programs](image2)

In this case, there are two properties verified. Figure 7 and figure 8 show the verification results of control programs. The system satisfies the deadlock-free property as shown in figure 7. However, it
violates the property of "when an obstacle is detected, the robot will move in the opposite direction" as shown in figure 8, and the counterexample path also is given.
Figure 9. Analysis of counterexample path.

According to the counterexample path provided above, figure 9 marks the counterexample path in the simulation environment. Through analysis, it is found that if progA and progB are scheduled by task T1 and task T2 respectively since both programs operate on the global variable Forward, the following situations may occur.

- Assuming T2 first executes progB, there is no obstacle detected at this time. Before T2 executes Forward = 100, T1 task (higher priority than T2) arrives, T2 is interrupted;
- T1 task starts to execute progA. First, set Obstacle to false, then judge the input signal from sensors. At the same time, the obstacle is detected, set Obstacle to true, and then reverse the movement, that is, Forward = -100;
- After T1 task is completed, T2 task continues to execute Forward = 100.

Through analysis, it is found that when there is an obstacle in front, the Forward value should be kept at -100, but progB rewrites the Forward value to 100, that is, the robot still moves forward when encountering an obstacle. This situation is obviously not in line with the expected results of the system, and such errors are not easy to detect by conventional test methods. In this paper, the model checking method is used to traverse the whole system state, and all possible situations can be found.

6. Conclusion

This paper's goal is to apply formal methods to industrial control programs. ST can serve as a pivot to unify the multiple PLC languages, and the transformation workflow is sketched from PLC programs written in other languages to ST. On this foundation, we provide the conversion rules from ST programs to a formal language CSP#, which lays the foundation for the formal verification of PLC programs. In addition, a case study with multi POU programs is shown to illustrate the transformation of ST programs, and the verification is further conducted with PAT platform. For the unsatisfied property, the reason is analyzed by the generated counterexample. Formal methods can detect some subtle bugs by traversing all states in the system, which is a recommended practice to ensure the correctness of safety-critical software.
References

[1] F. D. Petruzella, Programmable logic controllers. Tata McGraw-Hill Education, 2005.
[2] M. Tiegelkamp and K.-H. John, “Iec 61131-3: programming industrial automation systems,” 1995.
[3] B. F. Adiego, D. Darvas, E. B. Vinuela, J. C. Tournier, S. Bliudze, J. O. Blech, and V. M. G. Suarez, “Applying model checking to industrial sized plc programs,” IEEE Transactions on Industrial Informatics, vol. 11, no. 6, pp. 1400–1410, 2015.
[4] C. A. R. Hoare, “Communicating sequential processes,” Communications of the ACM, vol. 21, no. 8, pp. 666–677, 1978.
[5] J. Sun, Y. Liu, J. S. Dong, and J. Pang, “Pat: Towards flexible verification under fairness,” in International Conference on Computer Aided Verification. Springer, 2009, pp. 709–714.
[6] L. Yang, “Model checking concurrent and real-time systems: the pat approach,” Ph.D. dissertation, 2009.
[7] Y. Sun, “Toward a model checker for ambient logic using the process analysis toolkit,” Ph.D. dissertation, Bishop’s University, 2015.
[8] L. Huang, W. Liu, and Z. Liu, “Algorithm of transformation from plc ladder diagram to structured text,” in Electronic Measurement & Instruments, 2009. ICEMI’09. 9th International Conference on. IEEE, 2009, pp. 4–778.
[9] K. H. Moonga, Y. Linru, and L. Shaojun, “Algorithm for compiling unrestricted ladder diagram to iec 61131-3 compliant instruction list,” in Proceedings of the World Congress on Engineering (WCE) Vol. II, 2011.
[10] V. E. E. Lopes, “Converting ld to sfc (iec 61131-3),” 2017.
[11] https://beremiz.org/index.html.en.
[12] G. Frey and L. Litz, “Formal methods in plc programming,” in Systems, Man, and Cybernetics, 2000 IEEE International Conference on, vol. 4. IEEE, 2000, pp. 2431–2436.
[13] B. Zoubek, J.-M. Roussel, and M. Kwiatkowska, “Towards automatic verification of ladder logic programs,” in IMACS-IEEE” CESA’03”;” Computational Engineering in Systems Applications”, 2003, pp. CD–ROM.
[14] D. F. Bender, B. Combe male, X. Cregut, J. M. Farines, B. Berthomieu, and F. Vernadat, “Ladder metamodeling and plc program validation through time petri nets,” in European Conference on Model Driven Architecture-Foundations and Applications. Springer, 2008, pp. 121–136.
[15] F. Jouault and I. Kurtev, “Transforming models with atl,” Lecture Notes in Computer Science, vol. 3844, pp. 128–138, 2006.
[16] D. Soliman and G. Frey, “Verification and validation of safety applications based on plcopen safety function blocks using timed automata in uppaal,” in Proceedings of the Second IDAC Workshop on Dependable Control of Discrete Systems (DCDS), 2009, pp. 39–44.
[17] J. Li, “Automatic translation of fbd-plc-programs to nusmv for model checking safety-critical control systems.”
[18] R. Huuck, B. Lukoschus, and N. Bauer, “A model-checking approach to safe sfcs,” in Proc. CESA, 2003.
[19] B. Schlich, J. Brauer, J. Wernerus, and S. Kowalewski, “Direct model checking of plc programs in il,” IFAC Proceedings Volumes, vol. 42, no. 5, pp. 28–33, 2009.
[20] S. O. Biha, “A formal semantics of plc programs in coq,” in Computer Software and Applications Conference (COMPSAC), 2011 IEEE 35th Annual. IEEE, 2011, pp. 118–127.
[21] V. Gourcuff, O. De Smet, and J.-M. Faure, “Efficient representation for formal verification of plc programs,” in Discrete Event Systems, 2006 8th International Workshop on. IEEE, 2006, pp. 182–187.
[22] R. Wang, Y. Guan, M. Zhou, J. Zhang, and X. Song, “A component-based modeling and validation method for plc systems,” Advances in Mechanical Engineering, vol. 6, p. 127618, 2014.
[23] A. Basu, B. Bensalem, M. Bozga, J. Combaz, M. Jaber, T. H. Nguyen, and J. Sifakis, “Rigorous
component-based system design using the bip framework,” IEEE Software, vol. 28, no. 3, pp. 41–48, 2011.

[24] R. Huuck, B. Lukoschus, and N. Bauer, “A model-checking approach to safe sfcs,” Proc Cesa, 2003.

[25] M. Koskimies et al., “Applying model checking to analysing safety instrumented systems,” 2008.

[26] Y. Huang, Y. Zhao, L. Zhu, Q. Li, H. Zhu, and J. Shi, “Modeling and verifying the code-level osek/vdx operating system with csp,” in Fifth International Conference on Theoretical Aspects of Software Engineering, 2011, pp. 142–149.

[27] M. Kleine, “A csp-based framework for the specification, verification, and implementation of adaptive systems,” in International Symposium on Software Engineering for Adaptive and Self-Managing Systems, 2011, pp. 158–167.

[28] F. Zhang, Y. Zhao, D. Ma, and W. Niu, “Formal verification of behaviora aadl models by stateful timed csp,” IEEE Access, vol. PP, no. 99, pp. 1–1, 2017.