Formal model of IEC 61499 execution trace in FBME IDE

Tatiana Liakh [*], Radimir Sorokin [†], Daniil Akifev [†], Sandeep Patil[*], Valeriy Vyatkin[*] [‡]
[*] Department of Computer Science, Computer and Space Engineering, Lulea Tekniska Universitet, Sweden
[†] Independent researcher
[‡]Department of Electrical Engineering and Automation, Aalto University, Espoo, Finland

Email: tatiana.liakh@ltu.se, rad.sorokin@gmail.com, akifev.ru@gmail.com, sandeep.patil@ltu.se, vyatkin@ieee.org

***Abstract*—With increase in use formal verification tools and methods in distributed systems, it is becoming more challenging to analyse the execution traces generated by formal verification tools. This paper presents a method for unification of execution traces of industrial automation systems, based on IEC 61499 standard. Execution trace of a system is a sequence of events, where each event represents a change in the state of the system. Execution traces allow developers to explore safely behavior of control software. Execution traces can be obtained several ways, including monitoring of a real system (or its simulator), or as a counterexample build by model checker. In the paper we explore unification of execution traces for debug task in FBME - modular IDE for IEC 61499 applications. We present the formal model of the execution trace representation and show the working on a simple example.**

***Index Terms*—industrial automation, IEC 61499, model checking, testing, dynamic verification, IDE**

I. Introduction

IEC 61499 is a programming paradigm [1] for development of industrial control software. It is a standard that supports event-driven execution model. This models makes it easier to develop distributed control software, especially suited for Industry 4.0 applications [2].

The widespread use of the IEC 61499 standard has led to the need to create user-friendly and functional development tools, such as FBME (Function Blocks Modelling Environment), an integrated development environment (IDE) for IEC 61499 systems [3], [4]. FBME is based on IntelliJ and MPS platforms [5] and is designed modular and cross-platform. FBME modularity allows to integrate third-party tools and flexibly expand functionality.

Control software interacts with the real environment. Errors in such software can lead to hardware failures. This makes the cost of errors in such programs extremely high. Thus of particular interest are debugging and verification tools for developing IEC 61499 control programs. However, the specifics of industrial control software (interaction with environment, event-driven execution) complicates the use of traditional software quality control methods, such as testing, debugging, formal verification etc.

In previous works, we developed adaptation of formal verification techniques, such as model checking [6], and integrated them into FBME [7], [8]. Based on these studies, FBME uses NuSMV verifier in model checking module [9]. This module manages translation of IEC 61499 application into NuSMV model [10], runs NuSMV verifier and shows results of verification as counterexample. FBME also has debugger module that runs the explored IEC 61499 application in 4diac FORTE runtime and monitors the behaviour of the application, stores an execution traces and visualises it on module GUI.

These modules have similar features: they show counterexamples or execution traces, which describes system evolution over time. However, this approach to design industrial automation IDE design raises a number of problems:

- Execution trace from different tools may store different types of information. For example, runtime doesn't provides any information which line of the algorithm is executing right now;
- Execution traces are in not unified format, each tool stores trace in its own format. These differences confuse users when trying to analyse the behavior of the system;
- Textural representation of execution trace is not illustrative. Some studies have shown that textural representation obstructs adaptation of formal verification methods to the software development cycle [12];
- Modules developers have to do routine operations to display execution trace on GUI.

Thus, the task of developing a unified format of execution traces for IEC 61499 applications is relevant.

In this paper, we present unified formal model of IEC 61499 model execution trace. We analysed the formal model of IEC 61499 function block and built on its basis formal model of execution trace. Also we used this formal model of execution trace for modification of FBME architecture. FBME modules translates execution traces and counterexamples obtained from different third-party tools unto unified format and displays them on unified GUI. Finally we offer technical solutions for IntelliJ MPS based IDEs.

II. Related Facts

A. *IEC 61499 Standard*

IEC 61499 program is a graphical network of function blocks (FB). IEC 61499 is based on the object-oriented programming paradigm. Each function block belongs to a certain class. An example of an IEC 61499 application is in Fig. 1. Each function block (1) has interface defined by FB type. FB interface contains input event ports (3), output event ports (2), input data ports (4), and output data ports (5).

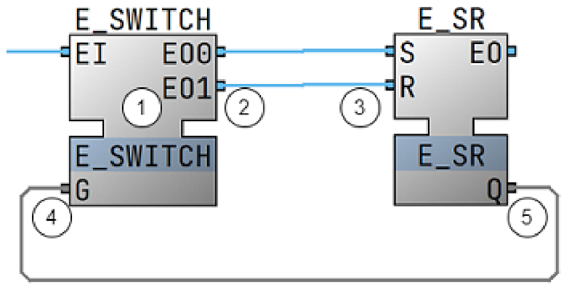


Fig. 1. IEC 61499 example application.

Execution Control Chart (ECC) describes the internal operational logic of the function block (Fig. 2).

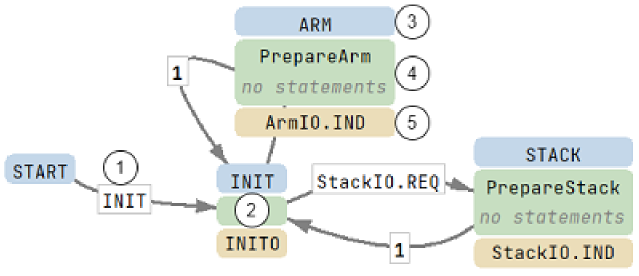


Fig. 2. IEC 61499 Execution Control Chart (ECC) diagram.

ECC is an executable event-driven state machine. ECC transitions (1) are managed by input events and logical boolean conditions. Each ECC state (2) has a unique identifier (3) and any number of algorithms (4). Standard uses Structured text (ST) language (from IEC 61131 standard) to express ECC algorithms. Also, ECC states can emit output messages (5). FBs are connected with event and data connections. Events manage the execution of IEC 61499 software.

B. IntelliJ Meta Programming system (MPS)

FBME is based on the IntelliJ Meta Programming system (MPS). MPS is a powerful workbench to design IDEs for domain-specific languages (DSL). MPS provides tools to express custom DSL grammar. Also, it provides tools to create executable code generators. MPS is based on Intellij IDEA IDE. This allows language developers to get all Intellij IDEA features at once, including advanced language editor, code completion, syntax highlighting, navigation, terminal, integration with Git, etc.

C. IEC 61499 Execution Trace Provider

Due to the specifics of IEC 61499 applications, FBME uses several types of IEC 61499 execution trace providers: model checkers, IEC 61499 runtime, and virtual simulator of IEC 61466 runtime.

- *Model checking* [6]is a powerful formal verification approach used to evaluate software quality. Unlike testing, formal verification proves whether the system meets a given formal specification. Model checking explores a finite-state model of a verified system. Specifications of the system are expressed with temporal logic formulas [11]. If the property is not satisfied, model checker generates a counterexample. The counterexample is an execution trace of the system formal model (sequence of system states of the explored model) which leads to the violation of the formal specification. FBME uses NuSMV and SPIN [13] verifies.
  At the same time, model checking cannot completely replace system testing and debugging for several reasons. First, model checkers use their own DSLs to describe the verified model. Also, verifier checks not the final system, but its formal model. Finally, a text representation of counterexamples makes it difficult to analyse the reasons for the specification violation.
- *IEC 61499 runtimes*. There are a lot of runtime environments for IEC 61499 compliant applications, such as NxT Forte, EcoRT, 4diac FORTE, and FBRT. Some of them provide real-time execution monitoring. Currently, FBME uses open 4diac FORTE runtime. 4diac FORTE allows to monitor the current state of the system, that is, the values of the event counters, the values of the variables, and the current state of ECC of the basic functional blocks at a certain point in time.
  At the same time, traditional testing and debugging application approaches are not suitable for industrial automation IDE, since it is impossible to ”pause” a cyber-physical distributed system.
- *IEC 61499 virtual simulator*. 4diac FORTE runtime allows monitoring system state updates at a specified frequency. However, if there are multiple system state changes between requests, the runtime does not allow to restore the order in which these changes occurred. This problem is especially relevant when restoring the order of IEC 61499 events emission. To solve this, we developed a runtime simulator in FBME. It allows users to analyse the execution of applications without running them in the runtime or on real devices by triggering events, setting the values of variables in functional blocks, and watching their reactions to changes. Simulation breeds a sequence of emitting events and ECC transitions of the basic functional blocks. The simulator collects this information into an execution trace.

III. Formal Model of IEC 61499 Program Execution Trace

Formal model of execution trace is based on IEC 61499 function block formal model. Formal semantics of IEC 61499 function blocks is based on Abstract State Machines [15]

Execution trace ET of a IEC 61499 system is a tuple:

$$ET = < s_1, s_2, s_3, ... > \quad (1)$$

Each element s_i of the tuple *ET* (1) presents an update of the whole IEC 61499 system state. s_i is a pair:

$$s_i = (TimeStamp_i, SystemStateUpdate_i) \quad (2)$$

Where:

SystemStateUpdate$_i$ (3) – tuple of events, shows state change of the state of the whole system. It is *not* an event in terms of IEC 61499.

TimeStamp$_i$ – moment of time, when the state update happened.

$$SystemStateUpdate_i =< e_{i1}, e_{i2}, ..., e_{iN} > \quad (3)$$

e_{ij} (4) is a system state change (event) detected between TimeStamp$_{i-1}$ and *TimeStamp$_i$*.

$$e_{ij} = (Fbd, SystemStateEventType, Data) \quad (4)$$

Fbd – id of the function block;

Data – tuple of arguments of arbitrary length;

The following is the event format depending on the event type. The semantics of the events arguments and their update rules is also explained.

SystemStateEventType possible values:

- *Q_update* – state Q of ECC of the specified FB was changed.

  $$e_{ij} = (Fbd, Q\_update, < Q\_fbd >) \quad (5)$$

  Q_fbd is a variable representing the current ECC state of function block Fbd.
- *VV_update* – New value of internal function block (FB) variable *var* from *VV_fbd*, where *VV_fb* - set of internal FB variables

  $$e_{ij} = (Fbd, VV\_update, < variable\_id, new\_value >) \quad (6)$$
- *VI_update* – New value of an input function block (FB) variable *var* from *VI_fbd*, where *VI_fb* - set of input FB variables

  $$e_{ij} = (Fbd, VI\_update, < variable\_id, new\_value >) \quad (7)$$
- *VO_update* – New value of a output function block (FB) variable *var* from *VO_fbd*, where *VO_fbd* - set of output FB variables

  $$e_{ij} = (Fbd, VO\_update, < variable\_id, new\_value >) \quad (8)$$
- *Alpha_update* – *alpha_fbd* flag shows that FB started its turn

  $$e_{ij} = (Fbd, Alpha\_update, < alpha\_fbd\_value >) \quad (9)$$
- *Beta_update* – *beta_fbd* flag shows that FB finished its turn

  $$e_{ij} = (Fbd, Beta\_update, < beta\_fbd\_value >) \quad (10)$$
- *Select_transition_event* – New value of *selectE_{ik}* – this function shows that input event E_{ik} was chosen by a FB from a buffer of input messages. The chosen event will be further used to choose the next enabled ECC transition.

  $$e_{ij} = (Fbd, Select\_transition\_event, < selectE_{ik} >) \quad (11)$$
- *Emits_event* – FB have emitted output event *event_id*

  $$e_{ij} = (Fbd, Emmits\_event, < event\_id >) \quad (12)$$
- *Ecc_transition_enabled* – transition *trn_id* of ECC was enabled

  $$e_{ij} = (Fbd, Ecc\_transition\_ennabled, < trn\_id >) \quad (13)$$
- *NA_event* – new value of *NA* pointer (counter) to the current ECC action

  $$e_{ij} = (Fbd, NA\_event, < NA >) \quad (14)$$
- *NI_event* – new value of *NI* pointer (counter) to the current step of running algorithm

  $$e_{ij} = (Fbd, NI\_event, < NI >) \quad (15)$$

This formal model of the IEC 61499 system execution trace was used in the modification of FBME architecture as a basis for a unified execution trace format. Based on the formal model of IEC 61499 function block, the formal model of execution trace fully describes any possible trace of the evolution of a system.

IV. Modification of FBME Architecture

In FBME we propose to use execution traces of an IEC 61499 application to prove the correctness of its code. Users may run the saved execution trace step by step in both directions and thus analyse the reasons for the application behavior. Also, modular architecture of FBME allows the integration of automatic trace analysis tools. For debugging purposes, we use three types of sources to produce execution traces: IEC 61499 compatible runtime, model checking tools, and a step-by-step virtual simulator of IEC 61466 runtime.

In the simplified form, the old version of FBME architecture is in Fig. 3.

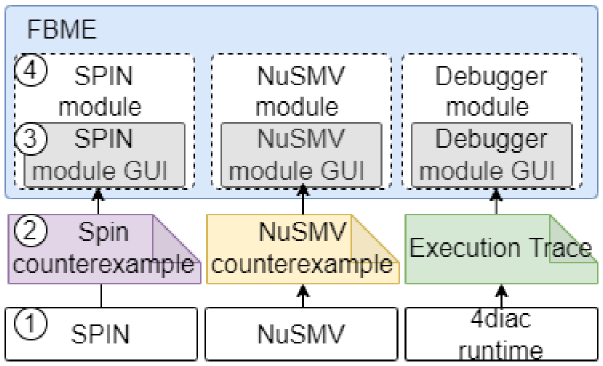


Fig. 3. Current FBME architecture.

Due to the lack of common execution trace model, FBME obtained three modules for analysing system execution of different sources. Each of these modules have their own GUI (3) and run asynchronously. Each module translates IEC 61499 application into a format compatible with the trace provider tool and manages the execution of the trace provider with this data. Trace provider creates execution trace (3). FBME module parses the execution trace and displays it on the module's GUI.

This architecture design gave rise to the problems mentioned above: a large number of different track formats. Each module had a separate GUI, which led to the need to duplicate code. Differences in the description of execution traces confused users. Also sometimes trace providers, e.g. model checkers, use their internal identifiers to describe the model of IEC 61499 application. All of this led to additional efforts both in the development of the FBME module and in the execution of trace analysis by the user.

The modular nature of FBME architecture allows to change shapes of its modules and their interactions with a comfort effort. We used the unified execution trace format to modify the FBME architecture (Fig. 4). Some amount of third-party trace provides, including model checkers, IEC 61499 runtimes, and simulators (1), produced execution traces in their specific, non-unified formats (2). Then FBME translates these execution traces into inner unified execution trace format (3) and presents them on unified GUI (4).

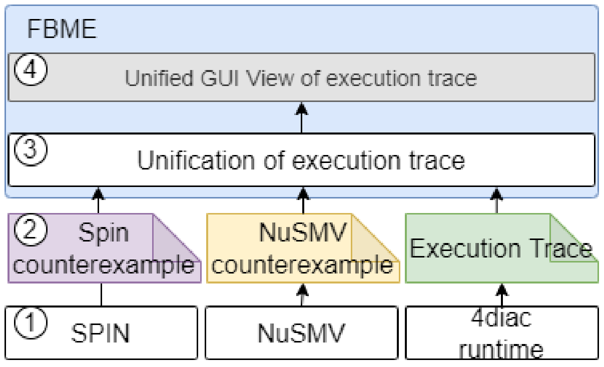


Fig. 4. The modified FBME architecture.

The modified architecture is presented in Fig. 5 on the example of the NuSMV integration plugin. TraceVisualizerPlugin is a MPS plugin responsible for the unified GUI representation of execution traces. This plugin contains a tool window that displays the list of available trace sources - model checkers, simulators, and IEC 61499 runtimes. Users choose which tool they are going to use to get the execution trace of a system. Also in this tool window, a user may set some input parameters for the chosen trace source tool, if it is needed. For example, for model checkers a user specifies the requirement to be checked. TraceVisualizerPlugin obtains the list of all integrated into FBME tools from TraceFactory (2) execution point interface, declared in TraceVisualizerPlugin. Execution point is a powerful MPS mechanism to exchange data between plugins. Other plugins providing system execution traces implement the TraceFactory interface, and TraceVisualizerPlugin may obtain the list of all implementations and display them on GUI. Also, users may choose to load an existing trace.

As soon as a user specifies the execution trace source, TraceVisualizerPlugin gets the corresponding implementation of the TraceFactory execution point. It asynchronously invokes generateTrace method from TraceFactory and receives java.util.concurrent.Future object. We use Future object because constructing an execution trace may take some time. The operation of industrial control systems is often associated with physical delays in a plant, and model checkers may also spend considerable time investigating a system model. The generateTrace method receives tuning parameters as input arguments for the specified tool. For NuSMV GenerateTrace receives a requirement to be checked by the verifier.

SMVPlugin manages the interaction with third-party tools via IntegrationService objects. IntegrationService objects manage calls of the Fb2SMV tool and NuSMV verifier. Execution trace, obtained from the verifier, has to be converted to a unified format. We have developed a TraceProviderToolkit library (4) for this purpose. It contains the necessary classes to build a unified execution trace of a system. Each plugin implements its successor of the CountereExampleParser abstract class – a translator from the specific format into the counterexample format by the unified formal model of the execution trace.

Execution trace is stored into a dynamic array of SystemStateUpdate objects, which describes state change of the state of the whole system, that has been detected in the timeStamp moment. Each SystemStateUpdate object contains a time stamp and array of SystemStateEvent objects. SystemStateEvent objects describe system events.

All possible SystemStateEvent types are stored in SystemStateEventType enumeration. Note that data in the execution trace may not include all types of SystemStateEventType - it may be related to the restrictions of concrete trace provider tool or restrictions of a system formal model. For example, Fb2SMV builds a formal NuSMV model which does not simulate system time. Or 4diac FORTE Runtime cannot provide ECC transitions inner data. Also, CountereExempleParser may use some additional data obtained during translation from IEC 61499 AST into a formal model - it may be file mapping identifiers from the formal model into identifiers from the IEC 61499 application. Additionally, CountereExempleParser may store the trace in a file.

When the execution trace is ready, CountereExempleParser publishes it into the Future object for TraceVisualizerPlugin. TraceVisualizerPlugin visualizes the obtained trace on the unified GUI.

With this approach, FBME remains modular and open to the integration of other third-party tools.

V. Use case example: A blinky app

Let's demonstrate the results of the work on a Blinky test system. The debugging process of this example in FBME is shown in Fig. 6. It is a simple IEC 61499 application, which switches between 0 and 1 every 1 second. The block diagram of the application is shown on panel 1. E_CYCLE block

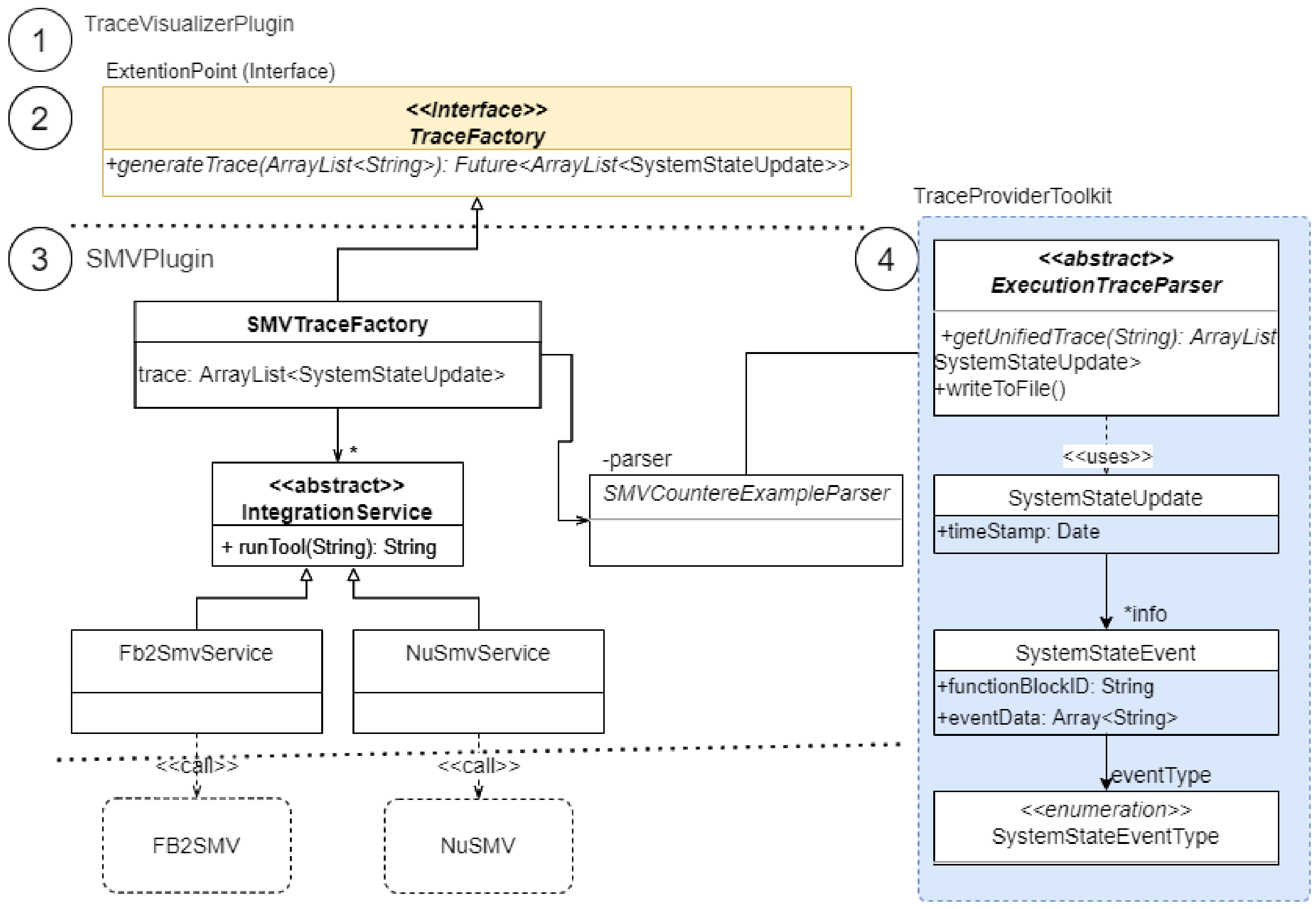


Fig. 5. The modified FBME architecture.

activates the application every 1 second. E_SWITCH function block switches between output events E0 and E1 depending on the value of an input variable G. E_SR is a Set/Reset function block that produces output boolean value Q.

In Fig. 6 execution trace is obtained from the 4diac FORTE runtime. Panel 2 shows execution trace steps. Panel 3 shows the current state of the system.

The previous FBME version didn't support such functionality. Different plugins for integration execution trace providers had separate non-unified GUIs and the user had to switch between them and repeat routine operations.

VI. Conclusion

In the paper, we introduce the formal model of execution traces based on the formal model of the IEC 61499 function blocks. We present the design approach of MPS-based IDE for IEC 61499 applications. The use of the unified execution trace format allows for standardization of the integration of various execution trace providers in IDEs for industrial automation. We don't have to deal with the textual representation of execution traces which makes it difficult to integrate formal methods into the development cycle. Integration of different trace providers, such as verification and debugging tools into FBME allows us to enhance the safety and reliability of developed IEC 61499 control applications. Also, the modified FBME architecture may be used as a pattern for MPS-based IDEs for industrial automation purposes.

Acknowledgment

This work was sponsored, in part, by the H2020 project 1-SWARM co-funded by the European Commission (grant agreement: 871743).

References

[1] "Programmable Logic Controllers — Part 3: Programming Languages, IEC Standard 61131-3", Third ed, 2013.

[2] V. Vyatkin, "IEC 61499 as Enabler of Distributed and Intelligent Automation: State-of-the-Art Review," in IEEE Transactions on Industrial Informatics, vol. 7, 2011, pp. 768-781.

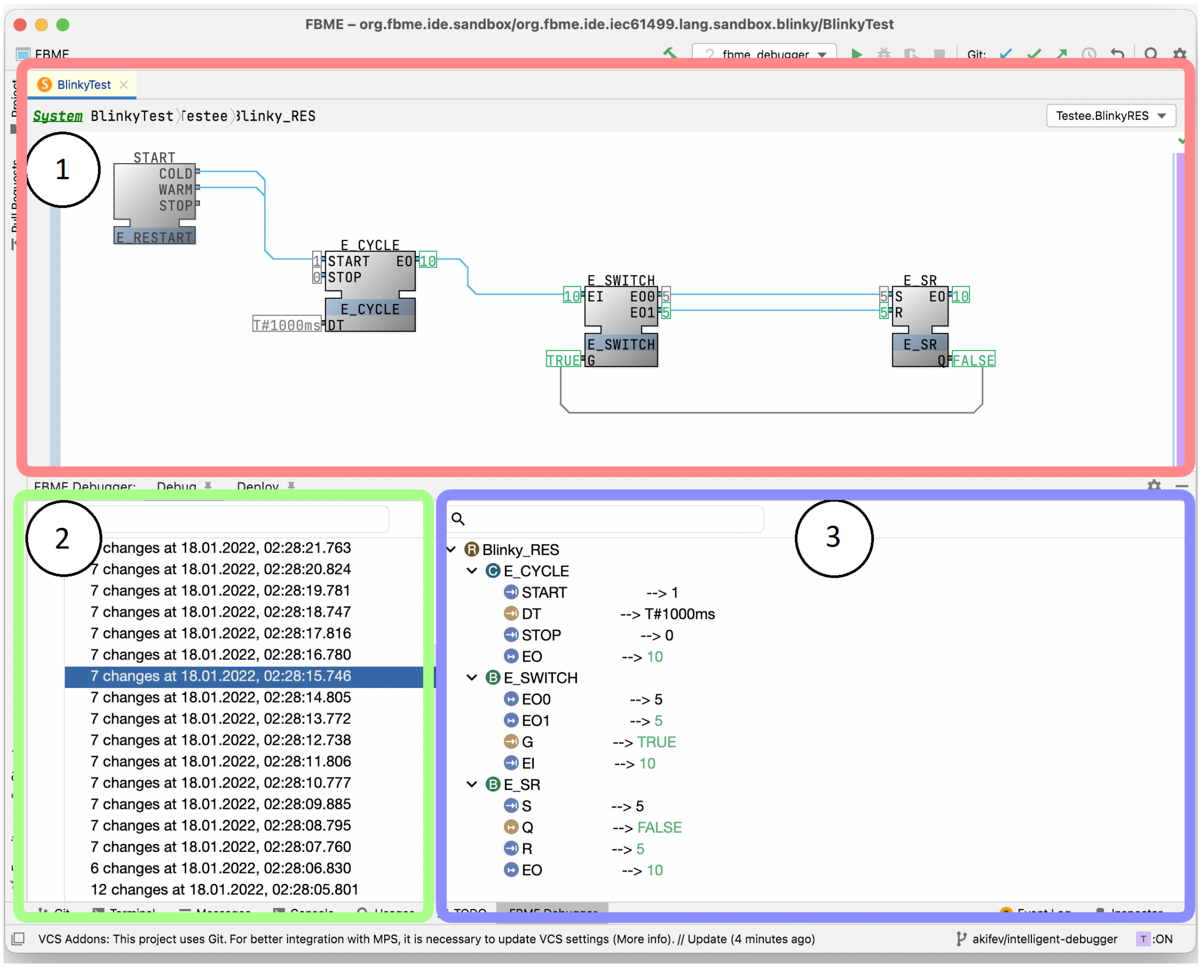
[3] Jet Brains, Jetbrains/fbme, 2020, [online] Available: https://github.com/JetBrains/fbme.

[4] R. Sorokin, S. Patil and V. Vyatkin, "Novel development tool for IEC 61499 based on domain-specific languages",in IFAC-PapersOnLine, vol. 55, no. 7, pp. 439-444, 2022.

[5] Jet Brains, Meta Programming System, 2020, [online] Available: https://www.jetbrains.com/mps/.

[6] E. M. Clarke, O. Grumberg, D. Kroening, D. Peled and H. Veith, Model checking, MIT press, 2018.

[7] V. Shatrov and V. Vyatkin, "Promela Formal Modelling and Verification of IEC 61499 Systems with comparison to SMV", in IFAC-PapersOnLine, 2021, pp. 1-6.


FBME – org.fbme.ide.sandbox/org.fbme.ide.iec61499.lang.sandbox.blinky/BlinkyTest
FBME
fbme_debugger
Git:
BlinkyTest
System BlinkyTest 〉Testee 〉Blinky_RES
Testee.BlinkyRES
1
START
COLD
WARM
STOP
E_RESTART
E_CYCLE
START
STOP
EO
DT
T#1000ms
E_SWITCH
EI
EO0
EO1
G
TRUE
E_SR
S
R
EO
Q
FALSE
FBME Debugger:
Debug
Deploy
2
3
Git
Terminal
Messages
Console
Usages
TODO
FBME Debugger
Event Log
Inspector
VCS Addons: This project uses Git. For better integration with MPS, it is necessary to update VCS settings (More info). // Update (4 minutes ago)
akifev/intelligent-debugger
T:ON

```
7 changes at 18.01.2022, 02:28:21.763
7 changes at 18.01.2022, 02:28:20.824
7 changes at 18.01.2022, 02:28:19.781
7 changes at 18.01.2022, 02:28:18.747
7 changes at 18.01.2022, 02:28:17.816
7 changes at 18.01.2022, 02:28:16.780
7 changes at 18.01.2022, 02:28:15.746
7 changes at 18.01.2022, 02:28:14.805
7 changes at 18.01.2022, 02:28:13.772
7 changes at 18.01.2022, 02:28:12.738
7 changes at 18.01.2022, 02:28:11.806
7 changes at 18.01.2022, 02:28:10.777
7 changes at 18.01.2022, 02:28:09.885
7 changes at 18.01.2022, 02:28:08.795
7 changes at 18.01.2022, 02:28:07.760
6 changes at 18.01.2022, 02:28:06.830
12 changes at 18.01.2022, 02:28:05.801
```

```
Blinky_RES
  E_CYCLE
    START      --> 1
    DT         --> T#1000ms
    STOP       --> 0
    EO         --> 10
  E_SWITCH
    EO0        --> 5
    EO1        --> 5
    G          --> TRUE
    EI         --> 10
  E_SR
    S          --> 5
    Q          --> FALSE
    R          --> 5
    EO         --> 10
```

Fig. 6. Debugging of blinky test example in FBME.

[8] M. Xavier, S. Patil and V. Vyatkin, ”Cyber-physical automation systems modelling with IEC 61499 for their formal verification,” 2021 IEEE 19th International Conference on Industrial Informatics (INDIN), 2021, pp. 1-6.

[9] A. Cimatti, E. Clarke, F. Giunchiglia and M. Roveri, ”Nusmv: a new symbolic model checker”, International Journal on Software Tools for Technology Transfer, vol. 2, no. 4, pp. 410-425, 2000.

[10] FB2SMV: IEC 61499 function blocks xml code to smv converter, 2021, [online] Available: https://github.com/dmitrydrozdov/fb2smv.

[11] R. Gotzhein, ”Temporal logic and applications—a tutorial”, Computer Networks and ISDN Systems, vol. 24, no. 3, pp. 203-218, 1992.

[12] B. Johnson, Y. Song, E. Murphy-Hill and R. Bowdidge, ”Why don't software developers use static analysis tools to find bugs?,” 2013 35th International Conference on Software Engineering (ICSE), pp. 672-681, 2013.

[13] G. J. Holzmann, ”The model checker SPIN,” in IEEE Transactions on Software Engineering, vol. 23, no. 5, pp. 279-295, May 1997.

[14] 4diac FORTE – IEC 61499 Runtime Environment, 2022, [online] Available: https://www.eclipse.org/4diac/en_rte.php

[15] W. Dai, C. Pang, V. Vyatkin, J. H. Christensen and X. Guan, ”Discrete-Event-Based Deterministic Execution Semantics With Timestamps for Industrial Cyber-Physical Systems,” in IEEE Transactions on Systems, Man, and Cybernetics: Systems, vol. 50, no. 3, pp. 851-862, 2020.