YR_DB_RUNTIME_VERIF: A FRAMEWORK FOR VERIFYING SQL CORRECTNESS PROPERTIES OF GUI SOFTWARE AT RUNTIME

Xavier Noumbissi Noundou

yeroth.d@gmail.com

Abstract. Software correctness properties are essential to maintain quality by continuous and regressive integration testing, as well as runtime monitoring the program after customer deployment. This paper presents an effective and lightweight C++ program verification framework: **YR_DB_RUNTIME_VERIF**, to check SQL (Structure Query Language) [29] software correctness properties specified as temporal safety properties [11]. A temporal safety property specifies what behavior shall not occur, in a software, as sequence of program events. **YR_DB_RUNTIME_VERIF** allows specification of a SQL temporal safety property by means of a very small state diagram [28]; such a state diagram, would only be specified by a start and an accepting state, and by a pre- and post-condition on the state diagram transition between them. In **YR_DB_RUNTIME_VERIF**, a specification characterizes effects of program events (via SQL statements) on database table columns by means of set interface operations ($\in$, $\notin$), and, enable to check these characteristics hold or not at runtime. Integration testing is achieved for instance by expressing a state diagram that encompasses both Graphical User Interface (GUI) states and MySQL [25] databases queries that glue them. For example, a simple specification would encompass states between 'Department administration' and 'Stock listing' GUI interfaces, and transitions between them by means of MySQL databases operations. **YR_DB_RUNTIME_VERIF** doesn't generate false warnings because its specifications are *not desirable (erroneous) specifications*. This paper focuses its examples on MySQL database specifications, labeled as states diagrams events, for the newly developed and FOSS (Free and Open Source Software) Enterprise Resource Planing Software YEROTH–ERP–3.0 [26].

Keywords: model–based testing · reactive system analysis · computer software program analysis · computer software dynamic program analysis · software integration testing with SQL and GUI · runtime monitoring

Table 1: YEROTH–ERP–3.0 RELEVANT SOFTWARE SYSTEM METRICS

Software System Metric	Value
User Interface (windows, dialog) number	60
MariaDB SQL table number	38
MariaDB SQL table column number	320
Source lines of code (SLOC)	300,000

1 Introduction

Fig. 1: **YR_DB_RUNTIME_VERIF** WORKFLOW (diagram inspired from operation diagram in [12]).
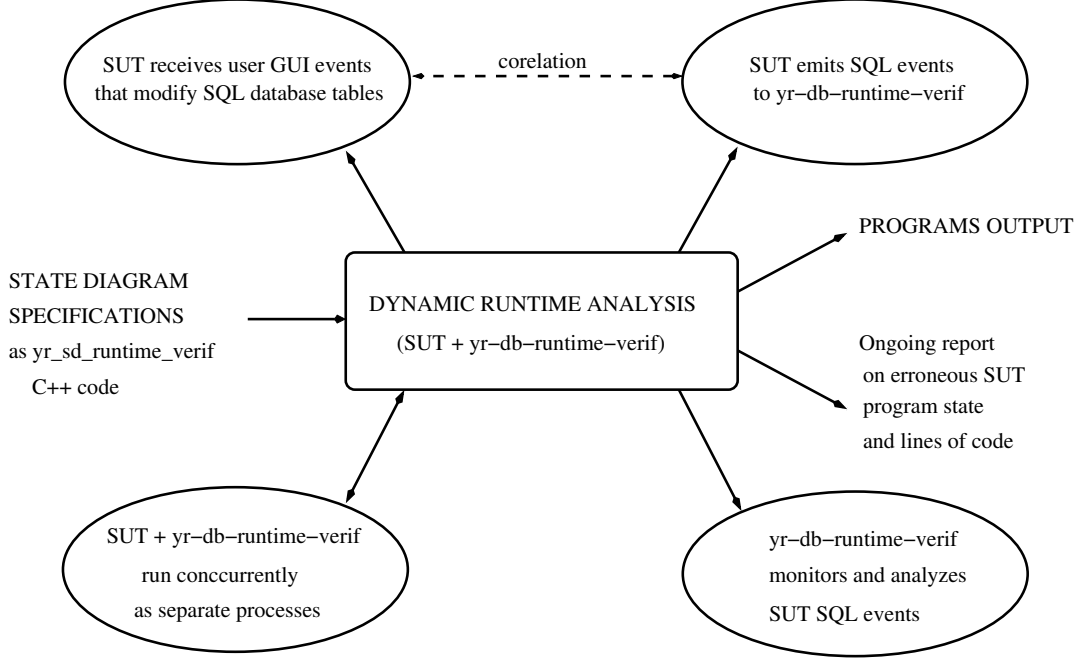
1.1 Motivations

This paper describes an effective dynamic analysis framework, based on runtime monitors specified in C++ programs (implemented in the software library **yr_sd_runtime_verif**), to perform software temporal safety property checking of GUI (Graphical User Interface) based software.

GUI based software are very comfortable and handy to use. However, tools to perform temporal safety property verification of GUI software are allmost not available as FOSS. The testing of combinations between GUI windows and database queries that glue them to make sense to the user, is allmost unavailable as FOSS, or at all to the best of the knowledge of the author of this paper. The FOSS C++ library **libfsmtest** [5] provides test suite generation support for source code behavior specifications as mealy automata. However, **libfsmtest** only allows for *desirable* correctness properties, and doesn't provide GUI (interaction) support or as plugin-based.

Unit or integration testing for GUI widgets is available by use of "NUnit" testing frameworks like e.g. **Qt-Test** [1], **CppUnit** [19], etc.. Software testing across GUI widgets (and MySQL queries) is however limited in support by these "NUnit" framework. To the best of the knowledge of the author of this paper, **DejaVu** [4] provides some support for **Java**'record and replay' testing while **FROGLOGIC** [13] provides support for C++ GUI software 'record and replay' testing technology. 'Record and replay' testing means a user performs a sequence of events that are recorded by testing infrastructure and automatically replay later on to see if expected events thereof occur. However,

none of this 'record and replay' technology tool enable temporal safety property specification as FOSS, with SQL as plugin.

As we will see in the related work, section 7, of this paper, most of software correctness property checking frameworks don't put an emphasis on checking temporal safety property of GUI software. Characterizing the effects of program statements (via SQL statements) on database table columns, and to check that these characteristics hold or not, is of predominant importance for large software systems with an impressive number of database tables. Table 1 illustrates for instance FOSS YEROTH–ERP–3.0 relevant software system metrics.

It means it can be very difficult for developers to keep application related logical requirements between the tables without appropriate software testing or analysis tools.

A large amount of former work on runtime monitoring assumes for a sequential program, or an abstraction of the program as one single source code, on which program analysis is performed [9,24,3,21,6,10,7].

The program analysis technique the author of this paper presents here abstract SQL events, GUI events, or sequences of them, as a state diagram, and enables developers to run them sequentially against a runtime monitor specified as a C++ program. In particular, the example presented in Section 2 specifies results of GUI windows events as SQL database pre-conditions on state diagram transitions; SQL events are specified as state diagram transition events. Figure 1 shows a high level overview of **YR_DB_RUNTIME_VERIF** workflow.

1.2   Main Contributions

This paper presents 3 original main contributions:

1. an industrial level quality framework (**YR_DB_RUNTIME_VERIF**: `http://github.com/yerothd/ yr-db-runtime-verif`), that solves temporal property verification by dynamic program analysis. **YR_DB_RUNTIME_VERIF** makes use of the C++ `Qt-Dbus` library, to input a *runtime monitor specification (`yr_sd_runtime_verif`)* as C++ program code, that also enables software–library–plugin checks;

2. a C++ library: `yr_sd_runtime_verif` (`http://github.com/yerothd/yr_sd_runtime_ verif`); modeling a state diagram runtime monitoring interface using only set algebra inclusion operations ($\in$, $\notin$) for state diagram program state specification as pre- and post-conditions.
   `yr_sd_runtime_verif` only enables the specification of states diagrams specifications as *not desirable (erroneous) behavior specifications*. Thus, **YR_DB_RUNTIME_VERIF** doesn't generate any false warning. A violation of a safety rule has been found whenever a final state could be reached. On the other hand, not reaching a final state doesn't mean that there is not a test case (or test input) that cannot reach this final state.

3. An application of **YR_DB_RUNTIME_VERIF** to check 1 temporal safety property error, found in the ERP FOSS YEROTH–ERP–3.0.

1.3   Overview

This paper is organized as follows: Section 2 presents a motivating example that will be used throughout this paper to explain the presented concepts of this paper. Section 3 presents formal definitions of the principal concepts used in this paper. Section 4 presents the software architecture of **YR_DB_RUNTIME_VERIF**, our GUI dynamic analysis framework. Section 5 introduces the C++ software library `yr_sd_runtime_verif` to model states diagrams, and reused by **YR_DB_RUNTIME_VERIF**. We evaluate our dynamic runtime analysis in Section 6. Section 7 compares this paper with other papers that achieve similar work or endeavors. Section 8 concludes this paper.

2   Motivating Example: missing department definition

Fig. 2: A motivating example, as current bug in YEROTH–ERP–3.0.
$Q0 := \text{NOT\_IN\_PRE}(\text{YR\_ASSET}, \text{department.department\_name})$.
$\overline{\overline{Q1}} := \text{IN\_POST}(\text{YR\_ASSET}, \text{stocks.department\_name})$.

Fig. 3: YEROTH–ERP–3.0 administration section displaying departments ($\neg \underline{Q0}$).
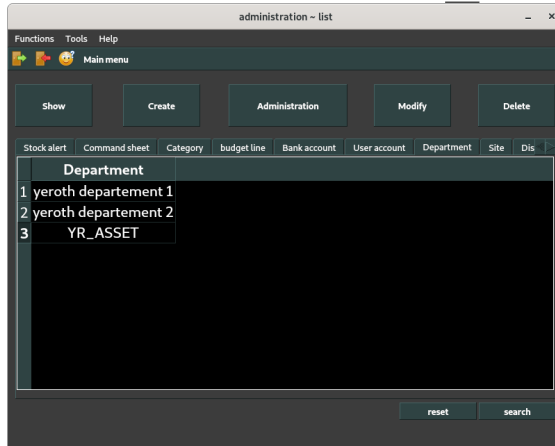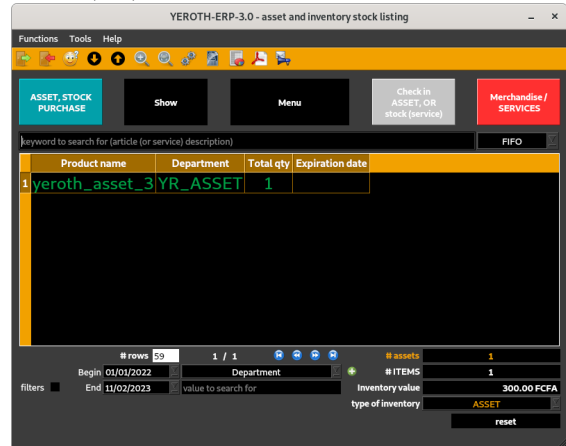

Fig. 4:   YEROTH–ERP–3.0 stock asset window listing some assets ($\overline{Q1}$).

2.1   The Enterprise Resource Planing Software YEROTH–ERP–3.0

YEROTH–ERP–3.0 is a fast, yet very simple in terms of usage, installation, and configuration Enterprise Resource Planing Software developed by Noundou et al. [26] for very small, small, medium, and large enterprises. YEROTH–ERP–3.0 is developed using C++ by means of the Qt development library. YEROTH–ERP–3.0 is a large software with around 300 000 (three hundred

Fig. 5: **YR_DB_RUNTIME_VERIF** command line shell output demonstrating that a final state has been reached (Section 6 analyzes these results).

```
         yer@JH-NISSI: ~                                    Q  ≡  _  □  ×
     8 yr-db-runtime-verif | YerothERPDatabase::YerothERPDatabase | database type: QMYSQL
     9
    10 "could register '/YR_DB_RUNTIME_VERIF_Main' object"
    11 "could register 'yr.db-runtime.verif' service"
    12 STARTING YR-DB-RUNTIME-VERIF !
    13        "[C++_STMT (SELECT.departements_produits)[24,1] at src/yeroth-erp-windows.cpp:967]"
    14 ************************************** START **************************************************
    15 *YR_CPP_MONITOR_EDGE::print_FOR_YEROTH_ERP specification edge event:  "select.departements_produits" **
    16 *[YR_CPP_MONITOR::YR_trigger_an_edge_event:] edge event evaluated triggered guarded condition:  true  **
    17 *[YR_CPP_MONITOR::YR_trigger_an_edge_event:] edge event start state:  "D"  **
    18 "execQuery: select * from departements_produits WHERE nom_departement_produit = 'YR_ASSET';"
    19 *[YR_CPP_MONITOR::CHECK_PRE_CONDITION_notIN:] precondition_IS_TRUE: FALSE    **
    20 *[YR_CPP_MONITOR::YR_trigger_an_edge_event:] START STATE precondition_IS_TRUE:  False  **
    21        "[C++_STMT (DELETE.departements_produits.YR_ASSET)[48,1] at src/admin/lister/yeroth-erp-admin-lis
    21 ter-window.cpp:1603]"
    22        "[C++_STMT (DELETE.marchandises.YR_ASSET)[48,1] at src/admin/lister/yeroth-erp-admin-lister-windo
    22 w.cpp:1659]"
    23        "[C++_STMT (SELECT.departements_produits)[24,1] at src/yeroth-erp-windows.cpp:967]"
    24 ************************************** START **************************************************
    25 *YR_CPP_MONITOR_EDGE::print_FOR_YEROTH_ERP specification edge event:  "select.departements_produits" **
    26 *[YR_CPP_MONITOR::YR_trigger_an_edge_event:] edge event evaluated triggered guarded condition:  true  **
    27 *[YR_CPP_MONITOR::YR_trigger_an_edge_event:] edge event start state:  "D"  **
    28 "execQuery: select * from departements_produits WHERE nom_departement_produit = 'YR_ASSET';"
    29 *[YR_CPP_MONITOR::CHECK_PRE_CONDITION_notIN:] precondition_IS_TRUE: True    **
    30 *[YR_CPP_MONITOR::YR_trigger_an_edge_event:] START STATE precondition_IS_TRUE:  True  **
    31 "execQuery: select * from stocks WHERE nom_departement_produit = 'YR_ASSET';"
    32 *[YR_CPP_MONITOR::CHECK_post_condition_IN:] postcondition_IS_TRUE: True    **
    33 *[YR_CPP_MONITOR::YR_trigger_an_edge_event:] edge event accepting final state:  "E"  **
    34 ************************************** END ****************************************************
    35 /** YR_DB_RUNTIME_VERIF_Monitor::YR_DB_RUNTIME_VERIF_Monitor_notify_SUCCESS_VERIFICATION **//
:
```

thousands) of physical source lines of code. **YR_DB_RUNTIME_VERIF** could be used for integration testing of YEROTH–ERP–3.0, among different software modules.

2.2   Example Temporal Safety Property

The motivating example of this paper consists of the temporal safety property stipulating that *"A DEPARTMENT SHALL NOT BE DELETED WHENEVER STOCKS ASSET STILL EXISTS UNDER THIS DEPARTMENT"*. This statement means that a user shall be denied the removal of department 'YR_ASSET' in Figure 3 because there are still a stock asset listed within department 'YR_ASSET', as illustrated in Figure 4. Figure 2 illustrates the above temporal safety property as a simple state diagram.

State Diagram Explanation 'D' is a *start* state as illustrated by an arrow ending on its state shape. 'E' is a *final* (error, or accepting) state as illustrated by a double circle as state shape.

The pre-condition $Q0$ (as a predicate) in state 'D':
'NOT_IN_PRE(YR_ASSET, department.department_name)' means:

1. *a department named* 'YR_ASSET' *is not in column* 'department_name' *of database table* 'department'. This might happen whenever button 'Delete' in Figure 3 is pressed when item 'YR_ASSET' is selected.

Similarly, the post-condition $\overline{Q1}$ (as a predicate) **'IN_POST(YR_ASSET, stocks.department_name)'**, in accepting state 'E', means:

1. ***a department named*** 'YR_ASSET' ***is in column*** 'department_name' ***of database table*** 'stocks'.

The **state diagram event transition** in Figure 2: 'select.department' denotes that when in 'D', a SQL 'select' on database table 'department' has occurred; 'E' is then reached as an ***accepting state***. The source code specified in Listing 1.4 also illustrates a specification in C++ using software library `yr_sd_runtime_verif` of the state diagram specification above.

2.3 `YR_DB_RUNTIME_VERIF` Analysis Report

The motivating example automaton in Figure 2 is analyzed by `YR_DB_RUNTIME_VERIF` as follows:

1. Whenever department 'YR_ASSET' is deleted in YEROTH–ERP–3.0, as done in Figure 3, the runtime monitor state 'D' with a state condition $Q0$ is entered

2. when MySQL library (plugin) event 'select.department' occurs, in Figure 3 because of YEROTH–ERP–3.0 displaying the remaining product departments, the guarded condition for edge event 'select.department' is automatically evaluated to 'True' by C++ library `yr_sd_runtime_verif`, because no other guarded condition was specified by the developer

3. `yr_sd_runtime_verif` enters the runtime monitor state to 'E' and state condition $\overline{Q1}$ via method `YR_trigger_an_edge_event(QString an_edge_event)` because there are still assets (**yeroth_asset_3**) left within product department 'YR_ASSET', as illustrated in Figure 4. 'E' is then an accepting (or final or error) state.

Figure 5 illustrates an analysis result of the afore described process, which gets evaluated and described in Evaluation Section 6.

2.4 Runtime Analysis Interpretation Of `yr_sd_runtime_verif` Models By `YR_DB_RUNTIME_VERIF`

The framework `YR_DB_RUNTIME_VERIF` assumes the following characteristics of a specification automaton in order to enable proper software integration testing:

1. the state diagram automaton only has 2 states: a start and a final state;
2. at most 1 state diagram transition pre-condition on the start state
3. exactly 1 post-condition on the final state, ***that must hold***, when the state diagram automaton reaches this final state
4. exactly one state diagram transition between the start and the final state
5. no edge guard condition

Future releases of `yr_sd_runtime_verif`***, as well as*** `YR_DB_RUNTIME_VERIF` ***will handle composition of states diagrams with*** 2 ***states in order to have states diagrams with more than only*** 2 ***states.***

3   Formal Definitions

`yr_sd_runtime_verif`'s formal description of the state diagram formalism follows **_Mealy machine_** [28] added with **_accepting states (final or erroneous state), and state diagram transition pre- and post-conditions_**. In comparison to statechart [15], which is a **_visual formalism_** for states diagrams, `yr_sd_runtime_verif` doesn't support for instance the following features: _hierarchical states (composite state, submachine state), timing conditions._

Definition 1. A state diagram is a 8–tuple $(S, S_0, C, \Sigma, \Lambda, \delta, T, \Gamma)$ where:

- **S**: a finite set of states
- **S_0** $\in S$: a start state (or initial state)
- **C**: a set of predicate conditions; pre-conditions are underlined (e.g.: $\underline{Q0}$), and post-conditions are overlined (e.g.: $\overline{Q1}$). A pre-condition is comparable to a Harel-statechart _guard condition_.
- **Σ**: an input alphabet, $\Sigma := \{False, True\}$.
  $'False'$ means no input from SUT into **`YR_DB_RUNTIME_VERIF`**.
  $'True'$ means any input could come from SUT.
- **Λ**: an output alphabet, ϕ the no output element
- $\delta : S \times C$: a 2-ary relation that maps a state s to a state-condition c as either a state diagram transition pre-condition ($\underline{c}$), or as a state diagram transition post-condition ($\overline{c}$).
- **T** $: S \times \Sigma \to S \times \Lambda$: a transition function that maps an input symbol to an output symbol and the next state.
- **Γ**: a set of accepting states.

For instance, for the motivating example described in Figure 2 we have:
S = {D, E}; **S_0** = D; **C** = $\{\underline{Q0}, \overline{Q1}\}$; **$\Sigma$** = $\{False, True\}$; **Λ** = $\{\phi, $'select.department'$\}$; δ = $\{(D, \underline{Q0}), (E, \overline{Q1})\}$; **T** = $\{((D, False), (D, \phi)), ((D, True), (E, $'select.department'$))\}$; **$\Gamma$** = {E}.

Definition 2. A pre-condition of a state diagram transition is a predicate that must be true before the transition can be triggered. A pre-condition $\underline{Q0}$ could have 2 forms:

- $\underline{Q0} := \text{IN\_PRE}(X, Y)$ that means value "X" is in ($\in$) database column value set "Y".
- $\overline{Q0} := \text{NOT\_IN\_PRE}(X, Y)$ that means value "X" is not in ($\notin$) database column value set "$\overline{Y}$".

Definition 3. A post-condition of a state diagram transition is a predicate that must be true after the transition was triggered. A post-condition $\overline{Q1}$ could have 2 forms:

- $\overline{Q1} := \text{IN\_POST}(A, B)$ that means value "A" is in ($\in$) database column value set "B".
- $\overline{Q1} := \text{NOT\_IN\_POST}(A, B)$ that means value "A" is not in ($\notin$) database column value set "B".

Proposition 1: NO FALSE WARNINGS. `yr_sd_runtime_verif` only allows 1 outgoing edge or transition in its specifications, as illustrated in Figure 2. There is no need to specify the red colored edge of Figure 2 because it represents runtime cases where no input event arrives from SUT into **`YR_DB_RUNTIME_VERIF`**. This property, together with algorithm 'YR_trigger_an_edge_event(QString an_edge_event)' (Listing 1.5) of `yr_sd_runtime_verif`, ensures that there are no false warnings during **`YR_DB_RUNTIME_VERIF`** analyses. For example, the runtime monitoring or verification systems [] may give false warnings.

4   The Software Architecture of `YR-DB-RUNTIME-VERIF`

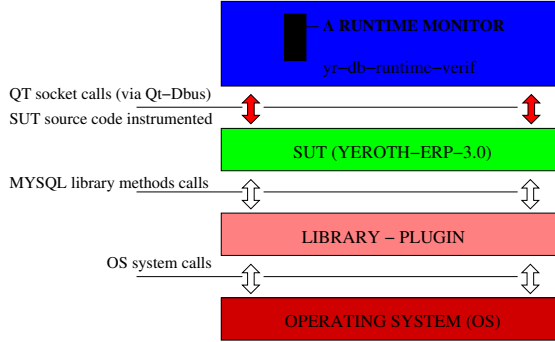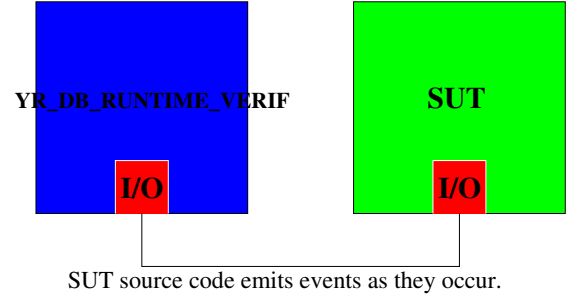Fig. 6: `YR_DB_RUNTIME_VERIF`: simplified software system architecture.

Fig. 7: `YR_DB_RUNTIME_VERIF` and SUT socket communication (diagram inspired from Jan Peleska diagram–work).

QT socket calls (via Qt–Dbus)
SUT source code instrumented

MYSQL library methods calls

OS system calls

SUT source code emits events as they occur.

4.1   Dynamic Analysis

SUT Source Code Instrumentation. `YR_DB_RUNTIME_VERIF` runs as a separate Debian Linux process from the application to dynamically analyze (YEROTH–ERP–3.0 in this case). Figure 6 illustrates a software system architecture layer of a software system that uses `YR_DB_RUNTIME_VERIF`. Figure 6 and Figure 7 illustrate how YEROTH–ERP–3.0 is instrumented to send MySQL database events, as they occur on due to the GUI of YEROTH–ERP–3.0, to process `YR_DB_RUNTIME_VERIF`, so it can perform runtime analysis of the monitor implemented within it.

Debugging Information. Each GUI manipulation of YEROTH–ERP–3.0 in its instrumented source code part could generate a state transition within the analyzed runtime monitor state diagram in `YR_DB_RUNTIME_VERIF`. Visualize "line 35" of Figure 5 to observe that a specific analysis message is sent to the console of `YR_DB_RUNTIME_VERIF` in cases where a final state has been reached; the message at "line 33" is for an accepting (final) state of the state diagram specification of the motivating example presented in Figure 2.

4.2   SQL Events

4.3   A Runtime Monitor (An Analysis Client)

Listing 1.1: **"XML file adaptor for YEROTH–ERP–3.0 test cases (reduced from 4 to only 1 SQL event for paper)."**

```xml
<!DOCTYPE node PUBLIC "−//freedesktop//DTD D−BUS Object Introspection 1.0//EN"
    "http://www.freedesktop.org/standards/dbus/1.0/introspect.dtd">
<node name="/YRruntimeverification">
  <interface name="com.yeroth.rd.IYRruntimeverification">
    <method name="YR_slot_refresh_SELECT_DB_MYSQL">
      <annotation name="org.qtproject.QtDBus.QtTypeName.In0" value="QString"/>
      <annotation name="org.qtproject.QtDBus.QtTypeName.In1" value="uint"/>
      <annotation name="org.qtproject.QtDBus.QtTypeName.In2" value="bool"/>
      <arg type="QString" direction="in"/>
      <arg type="uint" direction="in"/>
```
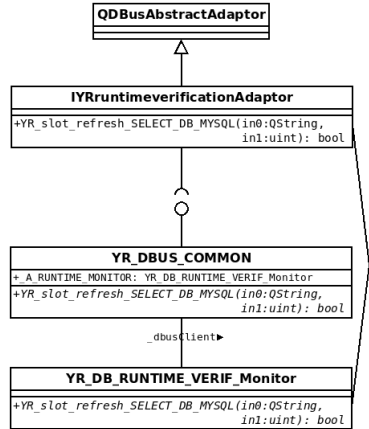
```
      <arg type="bool" direction="out"/>
    </method>
  </interface>
</node>
```

Fig. 8: **`YR_DB_RUNTIME_VERIF`**: simplified class diagram in UML [8].

A user (an analysis client) of **`YR_DB_RUNTIME_VERIF`** needs to subclass class `YR_DB_RUNTIME_VERIF_Monitor`. The UML class diagram in Figure 8 displays the class structure of **`YR_DB_RUNTIME_VERIF`**. Classes in the class diagram in Figure 8 only display a subset of their methods and interfaces so the diagram could fit in this paper.

`Qt-Dbus` communication adaptor `IYRruntimeverificationAdaptor` shall be generated by the user of this library (on **`YR_DB_RUNTIME_VERIF`** side) using `Qt-Dbus` command `qdbusxml2cpp` and an XML file, similar to the one displayed in Listing 1.1:

Listing 1.2: Command to generate `Qt-Dbus` adaptor on **`YR_DB_RUNTIME_VERIF`** side

```
qdbusxml2cpp −a YRruntimeverification_adaptor yr.db−runtime.verif.xml
```

Then, `Qt-Dbus` communication adaptor `IYRruntimeverificationAdaptor` must be generated on System Under Test (SUT) side (YEROTH–ERP–3.0 in this case):

Listing 1.3: Command to generate `Qt-Dbus` adaptor interface on SUT side (YEROTH–ERP–3.0 in this case)

```
qdbusxml2cpp −c IYRruntimeverificationAdaptor_Interface \
        −p src/IYRruntimeverificationAdaptor_interface.h:src/IYRruntimeverificationAdaptor_interface.cpp \
        yr.db−runtime.verif.xml
```

5 `yr_sd_runtime_verif`: A C++ Library to Model States Diagrams

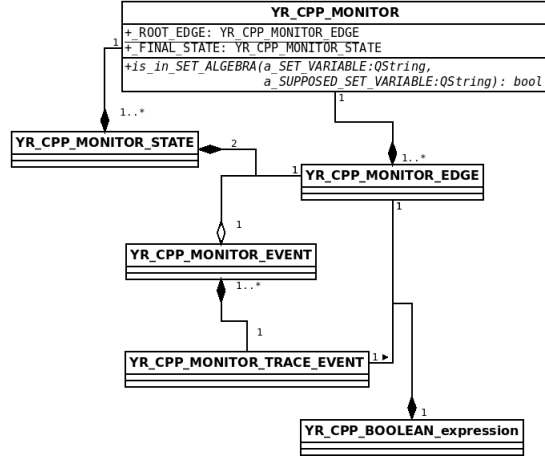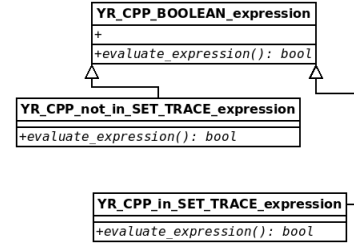Fig. 9: Class diagram in UML [8] to model
a State Transition Diagram.

Fig. 10: Class diagram in
UML [8] to model state diagram
transition trace conditions in
`yr_sd_runtime_verif` code.

5.1 Structure Of `yr_sd_runtime_verif`

Table 2: Runtime Monitor Specification Classes

State Diagram Feature	Class
State	YR_CPP_MONITOR_STATE
Transition	YR_CPP_MONITOR_EDGE
Event	YR_CPP_MONITOR_EVENT
Trace at state level	YR_CPP_MONITOR_TRACE_EVENT
Guard Condition	YR_CPP_BOOLEAN_expression
Set Trace Inclusion at edges	YR_CPP_in_SET_TRACE_expression
Set Trace non Inclusion at edges	YR_CPP_not_in_SET_TRACE_expression
Runtime Monitor	YR_CPP_MONITOR

`yr_sd_runtime_verif` is a state diagram C++ library the author of this paper created to work with the dynamic analysis program **YR_DB_RUNTIME_VERIF**. Figure 9 and Figure 10 represent the class structure, in UML, of `yr_sd_runtime_verif`. Listing 1.4 shows the C++ code that models the motivating example in Figure 2, and that uses runtime monitoring C++ state diagram library `yr_sd_runtime_verif`.

Listing 1.4: `yr_sd_runtime_verif` C++ code modeling a current bug in YEROTH–ERP–3.0 (Figure 2).

```
1  YR_CPP_MONITOR_EDGE *a_last_edge_0 = create_yr_monitor_edge("D",
2                                        "E",
3                                        "select.departements_produits");
4
5  a_last_edge_0->get_SOURCE_STATE()->set_START_STATE(true);
6
7  a_last_edge_0->get_TARGET_STATE()->set_FINAL_STATE(true);
8
9  a_last_edge_0->set_PRE_CONDITION_notIN("YR_ASSET",
10                                  "departements_produits.nom_departement_produit");
11
12 a_last_edge_0->set_POST_CONDITION_IN("YR_ASSET",
13                                  "stocks.nom_departement_produit");
14
15 YR_register_set_final_state_CALLBACK_FUNCTION(&YR_CALL_BACK_final_state);
```

There is no need to write C++ code for the red specified edge of Figure 2; this represents runtime cases where no input event arrives from SUT into `YR_DB_RUNTIME_VERIF`.

Table 2 specifies which class is in `yr_sd_runtime_verif` code for each runtime monitor/state diagram element.

5.2   Methods for Pre- and Post-Condition Specifications

Table 3: `yr_sd_runtime_verif` Methods for Pre-/Post-Condition Specification

Runtime Monitor State (class YR_CPP_MONITOR_EDGE) Methods	Utility
set_PRE_CONDITION_notIN(QString DB_VARIABLE, QString db_TABLE_db_COLUMN)	sets a NOT IN DATABASE pre–condition
set_PRE_CONDITION_IN(QString DB_VARIABLE, QString db_TABLE_db_COLUMN)	sets an IN DATABASE pre–condition
set_POST_CONDITION_notIN(QString DB_VARIABLE, QString db_TABLE_db_COLUMN)	sets a NOT IN DATABASE post–condition
set_POST_CONDITION_IN(QString DB_VARIABLE, QString db_TABLE_db_COLUMN)	sets an IN DATABASE pre–condition

Table 3 illustrates methods for specifying pre– and post–conditions of a runtime monitor state diagram transition. Each method takes in 2 arguments:

– QString DB_VARIABLE
– QString db_TABLE__db_COLUMN

The first method argument "DB_VARIABLE" specifies which variable is to be expected as value for the specification of the second variable argument "db_TABLE__db_COLUMN". The second variable gives in a string to be specified in format "DB_table_name.DB_table_column"; and its supposed value is the returned value of the first variable argument "DB_VARIABLE".

These 4 pre- and post-conditions methods make assumptions that a **program variable value** "DB_VARIABLE" is in set "DB_table_name.DB_table_column" or not; if the value of "DB_VARIABLE" is in the database table column, it means it is **in the set** ($\in$) of values "DB_table_name.DB_table_column"; and not being in the table column means it is **not in the set** ($\notin$).

Example from the motivating example in Section 2 Listing 1.4 of the runtime monitoring specification stipulates for instance in its "line 12", as post-condition:

```
a_last_edge_0->set_POST_CONDITION_IN("YR_ASSET",
                                     "stocks.nom_departement_produit");
```

that 'YR_ASSET' shall be a value in the value set ($\in$) of SQL table 'stocks' column 'nom_departement_produit'.

5.3   SUT Event Processing Method `YR_trigger_an_edge_event`

Listing 1.5: C++ Pseudo-code for `YR_trigger_an_edge_event(QString an_edge_event)`: `yr_sd_runtime_verif` method for triggering state diagram events (edges or transitions).

```
1   bool YR_CPP_MONITOR::YR_trigger_an_edge_event(QString an_edge_event)
2   {
3       YR_CPP_MONITOR_STATE cur_edge_SOURCE_STATE = _current_STATE;
4
5       YR_CPP_MONITOR_EDGE *cur_edge = cur_edge_SOURCE_STATE.outgoing_edge();
6
7       if (0 == cur_edge) { return false; }
8
9       if (true == cur_edge->evaluate_GUARDED_CONDITION_expression() &&
10         (an_edge_event == cur_edge->edge_event_token()))
11      {
12          bool precondition_IS_TRUE =
13                  cur_edge->CHECK_SOURCE_STATE_PRE_CONDITION(cur_edge_SOURCE_STATE);
14
15          if (precondition_IS_TRUE)
16          {
17              set_current_triggered_EDGE(cur_edge);
18
19              YR_CPP_MONITOR_STATE a_potential_accepting_state =
20                  cur_edge->get_TARGET_STATE();
21
22              if (CHECK_whether__STATE__is__Final(a_potential_accepting_state))
23              {
24                  CALL_BACK_final_state_FUNCTION(a_potential_accepting_state);
25              }
26
27              return true;
28          }
29      }
30
31      return false;
32  }
```

Listing 1.5 illustrates the pseudo–code of `yr_sd_runtime_verif` SUT event processing method `YR_trigger_an_edge_event(QString an_edge_event)`. '`YR_trigger_an_edge_event(QString an_edge_event)`' is responsible for interpreting a monitor at runtime, based on its current state, and on the current event received from SUT. Each state in `yr_sd_runtime_verif` states diagrams shall have only 1 outgoing edge (transition), by specification and construction, as explained in Proposition 3 in Section 3.

The algorithm in Listing 1.5 demonstrates that, given correct trace and event information from SUT, `yr_sd_runtime_verif` always exactly matches the user specification. Thus never giving false warnings.

6 Evaluation

The main experimental results in this paper demonstrate the efficacy of our tool to find errors in the SUT (YEROTH–ERP–3.0), presented in Subsection 2.2.

6.1 Qualitative Results

Table 4: SUT (YEROTH–ERP–3.0) Trace Output (Figure 5).

CONSOLE OUTPUT LINE	SQL EVENT	SUT PROGRAM POINT (TRACE)
21	"DELETE.department.YR_ASSET"	"src/admin/lister/yeroth-erp-admin-lister-window.cpp:1603"
22	"DELETE.merchandise.YR_ASSET"	"src/admin/lister/yeroth-erp-admin-lister-window.cpp:1626"
23	"SELECT.department"	"src/yeroth-erp-windows.cpp:967"

SUT (YEROTH–ERP–3.0) TRACING. Table 4 illustrates SUT source code trace information as presented in `YR_DB_RUNTIME_VERIF` console output in Figure 5. We have translated from French to English the MariaDB SQL table names.

SQL EVENT CALL SEQUENCE. Listing 1.4 illustrates the C++ code that we created to model and generate `YR_DB_RUNTIME_VERIF` binary executable that generates output in Figure 5 after deletion of department 'YR_ASSET' in Figure 3 of our motivating example. A careful observation of the output in Figure 5 illustrates the following sequence:

1. **line 23:** at state D, execution of the state diagram event "select.department " (SUT button 'Delete' has been pressed at **line 21**) :

   ```
   select * from departements_produits WHERE
       nom_departement_produit = 'YR_ASSET';
   ```

2. **line 28, line 29:** evaluation of the pre–condition $Q0$ of state D stating that product department 'YR_ASSET' is not existent evaluates to 'TRUE' (triggering of event "delete.department.YR_ASSET " by pressing of SUT button 'Delete' at **line 21** has removed any asset department name 'YR_ASSET').

   ```
   *[YR_CPP_MONITOR::CHECK_PRE_CONDITION_notIN:] precondition_IS_TRUE: True    **
   ```

3. **line 31, line 32:** checking post–condition $\overline{Q1}$ in state E (there are still stocks in stock department 'YR_ASSET') evaluates to 'TRUE', thus state E is reached as an accepting state, because department name 'YR_ASSET' still exists in SUT SQL table "stocks", as illustrated in Figure 4 of the motivating example:

   ```
   "execQuery: select * from stocks WHERE nom_departement_produit = 'YR_ASSET';"
   *[YR_CPP_MONITOR::CHECK_post_condition_IN:] postcondition_IS_TRUE: True    **
   ```

6.2 Runtime Performance

`YR_DB_RUNTIME_VERIF` and `yr_sd_runtime_verif` don't incur a runtime supplemental overhead to the SUT, apart from emitting SQL events from SUT to `YR_DB_RUNTIME_VERIF` as they occur, because no hand–shaking mechanism is used between `YR_DB_RUNTIME_VERIF` and the SUT. The emission of an SQL event from SUT to `YR_DB_RUNTIME_VERIF` doesn't cost more than 2 statements execution time (getting a pointer to the DBUS server, and calling a method 'YR_slot_refresh_SELECT_DB_MYSQL' or other similar 3 methods (for INSERT, UPDATE, and, DELETE) on it).

7   Related Work

1. **SUT source code instrumentation with specification.** "Clara" [7] enables to express software correctness properties using `AspectJ` and *dependency state machines*, both as instances of the typestate formalism, a formalism that is merely used for checking correctness of programs by a static compilation (analysis) technique called ***typestate checking***. The Clara framework weaves (instruments), and annotates a program with runtime monitors using `AspectJ`, then tries to optimize the weaved program by static analysis. The "residual program", meaning the weaved statically optimized program is then executed and runtime monitored by developers to detect runtime errors. Runtime monitoring tools [9,24,3,21,6,10] work as similar as the Clara framework does.

   `YR_DB_RUNTIME_VERIF` doesn't instrument the System Under Test (SUT) with any specification. It runs the runtime monitor concurrently from the analyzed SUT, but not with hand–shaking mechanism, thus not augmenting runtime execution of the SUT. `YR_DB_RUNTIME_VERIF` specifies the runtime monitor as a state diagram, a subset of typestate, specified as a C++ program, and augmented with accepting states and state transition pre- and post-condition.

2. **Specification as set interface operations.** "Hob" [22,23] is a program verification framework that enables to: characterize effects of program statement on data structures by means of all ($\forall$, $\exists$, etc.) algebra abstract set interface operations; and to check that these characteristics hold or not, using static analyses.

   `YR_DB_RUNTIME_VERIF` is a program verification framework that enables to: characterize effects of program statements (via SQL [25] (Structure Query Language) on database table columns by means of set interface operations ($\in$, $\notin$); and to check that these characteristics hold or not, using dynamic runtime analysis.

3. ***Offline* analysis Vs. *Online* analysis.**
   "`DejaVu` " [18] enables to check safety temporal property expressed in **first-order past linear-time temporal logic (FO-PLTL)** for events that carry data. `DejaVu` inputs a trace log (*offline*) and a FO-PLTL formula, and outputs a boolean value for each position in the inputted trace. "LogScope" [17] checks, *offline*, software systems correctness properties expressed using a rule–based specification language over state machines. It is not very precise what type of state machine is created and processed. "LogScope" translates specifications into C++ monitors (that could carry data). "`EventRaceCommander`" [2] repairs in web applications (*online*), event race errors, a kind of safety error. "`Purify`" [16] doesn't allow for custom SUT specifications. It has built-in memory access safety properties to check *offline* on program execution, after instrumentation of the SUT, its third-party, and vendor object-code libraries.

   `YR_DB_RUNTIME_VERIF` inputs a SUT correctness property specification as a state diagram (as a subset of `LTL` [11]). States diagrams specifications are implemented as C++ program monitors using C++ library `yr_sd_runtime_verif`. `YR_DB_RUNTIME_VERIF` outputs a developer given (by means of a callback function, as seen in 'line 15' in Listing 1.4) string message [1] in case an accepting state was entered, and a trace event of YEROTH–ERP–3.0 leading to it. `YR_DB_RUNTIME_VERIF`'s monitors need not store data, as `DejaVu` monitors must. `YR_DB_RUNTIME_VERIF` events also carry data (database table and column name, records quantity modified by current SUT event). Runtime monitors could be checked against programs written in any programming language or framework, as long as they emit necessary SQL events to `YR_DB_RUNTIME_VERIF`.

---

[1] **'YR_DB_RUNTIME_VERIF_Monitor_notify_SUCCESS_VERIFICATION'** in this paper motivating example in Figure 5.

8 Conclusion And Future Work

This paper has presented a lightweight C++ Qt-Dbus [20] tool to check a program against a runtime monitor using set interface operations ($\in$, $\notin$) on program statement: **YR_DB_RUNTIME_VERIF**. **YR_DB_RUNTIME_VERIF** doesn't generate false warnings because its specifications are *not desirable (erroneous) specifications*.

Since the concurrent communication between **YR_DB_RUNTIME_VERIF** and a program occurs over the RPC instance Dbus, a runtime monitor could be checked against programs written in any programming language or framework, as long as they emit necessary SQL events to **YR_DB_RUNTIME_VERIF**.

Runtime monitors specifications in yr_sd_runtime_verif, modeled as state diagram mealy machines, augmented with accepting states, and pre-/post-condition on state diagram transitions, as introduced in Section 3 of this paper, don't allow more than 2 states and 1 state diagram transition currently.

Future work would be an algorithm within yr_sd_runtime_verif, to compose single 2–states state diagram mealy machines. The author of this paper is also planing to create a compiler for the specification language grammar in Appendix B Figure 13 so to automatically generate yr_sd_runtime_verif C++ state diagram specification code.

Fig. 11: 'YR_QVGE' model for the running example in Figure 2.
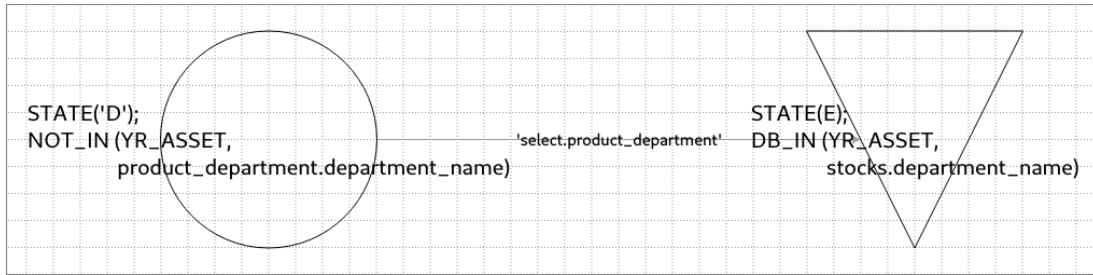
Also, the author of this paper has started, as future work, creating a graphical drawing tool (YR_QVGE) for in Section 3 defined state diagrams. A model is shown in Figure 11. It is an extension of the FOSS (Free and Open Source Software) Graphviz [14] drawing tool QVGE [27].

9 Acknowledgments

References

1. doc.qt.io/qt 5: Qt 5.15 (Dec 2022), accessed last time on Dec 22, 2022 at 12:40
2. Adamsen, C.Q., Møller, A., Karim, R., Sridharan, M., Tip, F., Sen, K.: Repairing event race errors by controlling nondeterminism. In: Proceedings of the 39th International Conference on Software Engineering, ICSE (2017). `https://doi.org/10.1109/ICSE.2017.34`, `files/ICSE17Repairing.pdf`
3. Allan, C., Avgustinov, P., Christensen, A.S., Dufour, B., Goard, C., Hendren, L.J., Kuzins, S., Lhoták, J., Lhoták, O., de Moor, O., Sereni, D., Sittampalam, G., Tibble, J., Verbrugge, C.: abc the aspectbench compiler for aspectj a workbench for aspect-oriented programming language and compilers research. In: Johnson, R.E., Gabriel, R.P. (eds.) Companion to the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2005, October 16-20, 2005, San Diego, CA, USA. pp. 88–89. ACM (2005). `https://doi.org/10.1145/1094855.1094877`, `https://doi.org/10.1145/1094855.1094877`
4. Alpern, B., Ngo, T., Choi, J.D., Sridharan, M.: DejaVu: deterministic Java replay debugger for Jalapeño Java virtual machine. In: Addendum to the Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA) (2000). `https://doi.org/http://doi.acm.org/10.1145/367845.368073`, `files/dejavu_demo.pdf`
5. Bergenthal, M., Krafczyk, N., Peleska, J., Sachtleben, R.: libfsmtest an open source library for fsm-based testing. In: Clark, D., Menendez, H., Cavalli, A.R. (eds.) Testing Software and Systems. pp. 3–19. Springer International Publishing, Cham (2022)
6. Bodden, E.: J-LO - A tool for runtime-checking temporal assertions. Diploma thesis, RWTH Aachen University (Nov 2005), `https://www.bodden.de/pubs/bodden05jlo.pdf`
7. Bodden, E., Hendren, L.: The clara framework for hybrid typestate analysis. International Journal on Software Tools for Technology Transfer (STTT) **14**, 307–326 (2012), `https://www.bodden.de/pubs/bl2010clara.pdf`, 10.1007/s10009-010-0183-5
8. Booch, G., Rumbaugh, J., Jacobson, I.: Unified Modeling Language User Guide, The (2nd Edition) (Addison-Wesley Object Technology Series) (2005)
9. Butkevich, S., Renedo, M., Baumgartner, G., Young, M.: Compiler and tool support for debugging object protocols. In: SIGSOFT '00/FSE-8 (2000)
10. Chen, F., Rosu, G.: Mop: an efficient and generic runtime verification framework. In: Gabriel, R.P., Bacon, D.F., Lopes, C.V., Jr., G.L.S. (eds.) Proceedings of the 22nd Conference on Object-Oriented Programming, Systems, Languages and Applications. pp. 569–588. ACM (2007). `https://doi.org/10.1145/1297027.1297069`
11. Clarke, E.M., Grumberg, O., Kroening, D., Peled, D.A., Veith, H.: Model checking, 2nd Edition (2018), `https://mitpress.mit.edu/books/model-checking-second-edition`
12. Eyolfson, J., Lam, P.: Detecting unread memory using dynamic binary translation. In: Qadeer, S., Tasiran, S. (eds.) Runtime Verification. pp. 49–63. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)
13. froglogic.com: Home • froglogic. `http://www.froglogic.com/home` (Dec 2022), accessed last time on Dec 18, 2022 at 20:00
14. graphviz.org: DOT Language | Graphviz. `http://graphviz.org/doc/info/lang.html` (2022), accessed last time on JUNE 8, 2022 at 12:30
15. Harel, D.: Statecharts: a visual formalism for complex systems. Science of Computer Programming **8**(3) (1987)
16. Hastings, R.O., Joyce, B.A.: Fast detection of memory leaks and access errors (1991)
17. Havelund, K.: Specification-based monitoring in c++. In: Margaria, T., Steffen, B. (eds.) Leveraging Applications of Formal Methods, Verification and Validation. Verification Principles. pp. 65–87. Springer International Publishing, Cham (2022)
18. Havelund, K., Peled, D., Ulus, D.: Dejavu: A monitoring tool for first-order temporal logic. pp. 12–13 (04 2018). `https://doi.org/10.1109/MT-CPS.2018.00013`
19. http://freedesktop.org/wiki/Software/cppunit: cppunit (Dec 2022), accessed last time on January 01, 2023 at 12:00
20. doc.qt.io/qt-5/qtdbus index.html: Qt D-Bus (Dec 2022), accessed last time on Dec 22, 2022 at 12:40

21. Krüger, I.H., Lee, G., Meisinger, M.: Automating software architecture exploration with m2aspects. In: Whittle, J., Geiger, L., Meisinger, M. (eds.) SCESM '06: Proceedings of the 2006 International Workshop on Scenarios and State Machines: Models, Algorithms, and Tools, Shanghai, China, May 27, 2006. pp. 51–58. ACM (2006). `https://doi.org/10.1145/1138953.1138964`, `https://doi.org/10.1145/1138953.1138964`

22. Kuncak, V., Lam, P., Zee, K., Rinard, M.: Modular pluggable analyses for data structure consistency. Transactions on Software Engineering **32**(12), 988–1005 (Dec 2006)

23. Lam, P.: The Hob System for Verifying Software Design Properties (February 2007)

24. Maoz, S., Harel, D.: From multi-modal scenarios to code: compiling lscs into aspectj. In: Young, M., Devanbu, P.T. (eds.) Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2006, Portland, Oregon, USA, November 5-11, 2006. pp. 219–230. ACM (2006). `https://doi.org/10.1145/1181775.1181802`, `https://doi.org/10.1145/1181775.1181802`

25. MariaDB.org: MariaDB Foundation - MariaDB.org. `http://www.mariadb.org` (Jun 2022), accessed last time on June 24, 2022 at 12:20

26. Noundou, X.N.: YEROTH–ERP–PGI–3.0 Doctoral Compendium. `http://archive.org/download/yeroth-erp-pgi-compendium_202206/JH_NISSI_ERP_PGI_COMPENDIUM.pdf` (Jun 2022), accessed last time on January 21, 2023 at 23:24

27. showroom.qt.io: QVGE; Qt Visual Graph Editor | Showroom (Jun 2022), accessed last time on Jun 27, 2022 at 12:40

28. Wikipedia.org: Mealy machine. `http://en.wikipedia.org/wiki/Mealy_machine` (Dec 2022), accessed last time on Dec 15, 2022 at 12:00

29. Wikipedia.org: SQL - Wikipedia. `http://en.wikipedia.org/wiki/SQL` (Feb 2023), accessed last time on February 08, 2023 at 12:00

A   Processing of SUT Event Stream By An Analysis Client

Listing   1.6:   'DO_VERIFY_AND_or_CHECK_ltl_PROPERTY':
YR_DB_RUNTIME_VERIF's overridden method for processing SUT event stream C++ pseudo-
code.

```
1   bool DO_VERIFY_AND_or_CHECK_ltl_PROPERTY(
2      QString  sql_table_NAME,
3      SQL_CONSTANT_IDENTIFIER cur_SQL_command)
4   {
5      switch (cur_SQL_command)
6      {
7      case SELECT:
8
9          if ("department" == sql_table_NAME))
10         {
11             return YR_trigger_an_edge_event("select.department");
12         }
13         break;
14
15     default:
16             break;
17     }
18
19     return false;
20  }
```

Listing 1.6 illustrates the pseudo-code of **YR_DB_RUNTIME_VERIF** SUT event processing
method 'DO_VERIFY_AND_or_CHECK_ltl_PROPERTY'. An analysis client must
first override method 'DO_VERIFY_AND_or_CHECK_ltl_PROPERTY' of class
'YR_DB_RUNTIME_VERIF_Monitor' so to implement a checking algorithm for each event received
from SUT, as for instance the events illustrated in Figure 2 of the motivating example.

The   analysis   client   then   calls   method   'YR_trigger_an_edge_event(QString
an_edge_event)' of class 'YR_CPP_RUNTIME_MONITOR' of C++ library yr_sd_runtime_verif
for each corresponding state diagram transition event.

B   YR_SD_RUNTIME_VERIF SPECIFICATION LANGUAGE

Fig. 12: **A Mealy Machine State Diagram Specified Using yr_sd_runtime_verif Specifi-
cation Language.**

```
1. yr_sd_mealy_automaton_spec yr_test_sd_mealy
2. {
3.  START_STATE(d);IN_PRE(v, table.column)
4.   -> [IN_SET_TRACE('delete.table.v', STATE(d))] / 'select.table' ->
5.    FINAL_STATE(e);NOT_IN_POST(v, table.column).
6. }
```

Fig. 13: **Grammar in Extended Backus Naur Form (EBNF) of `yr_sd_runtime_verif` Mealy Machine State Diagram Specification Language.**

⟨*specification*⟩ ::= **yr_sd_mealy_automaton_spec** '{' ⟨*mealy-automaton-spec*⟩ '.' '}'

⟨*mealy-automaton-spec*⟩ ::= ⟨*sut-state-spec*⟩
           | ⟨*sut-state-spec*⟩ '→' ⟨*sut-edge-state-spec*⟩

⟨*sut-edge-state-spec*⟩ ::= ⟨*sut-edge-mealy-automaton-spec*⟩ '→' ⟨*mealy-automaton-spec*⟩

⟨*sut-edge-mealy-automaton-spec*⟩ ::= ⟨*edge-mealy-automaton-guard-cond*⟩ ⟨*event-method-call*⟩

⟨*edge-mealy-automaton-guard-cond*⟩ ::= /* empty */ | '[' ⟨*trace-specification*⟩ ']' '/'

⟨*trace-specification*⟩ ::= ⟨*in-set-trace*⟩ | ⟨*not-in-set-trace*⟩

⟨*sut-state-spec*⟩ ::= ⟨*start-state-property-spec*⟩ ';' ⟨*algebra-set-specification*⟩
           | ⟨*state-property-spec*⟩ ';' ⟨*algebra-set-specification*⟩
           | ⟨*final-state-property-spec*⟩ ';' ⟨*algebra-set-specification*⟩

⟨*algebra-set-specification*⟩ ::= ⟨*in-algebra-set-spec*⟩ | ⟨*not-in-algebra-set-spec*⟩

⟨*in-algebra-set-spec*⟩ ::= ⟨*in-spec*⟩ '(' ⟨*prog-variable*⟩ '.' ⟨*db-table*⟩ '.' ⟨*db-column*⟩ ')'

⟨*not-in-algebra-set-spec*⟩ ::= ⟨*not-in-spec*⟩ '(' ⟨*prog-variable*⟩ **.** ⟨*db-table*⟩ '.' ⟨*db-column*⟩ ')'

⟨*in-set-trace*⟩ ::= **IN_SET_TRACE**'(' ⟨*event-method-call*⟩ ',' ⟨*state-property-specification*⟩ ')'

⟨*not-in-set-trace*⟩ ::= **NOT_IN_SET_TRACE**'(' ⟨*event-method-call*⟩ ',' ⟨*state-property-specification*⟩ ')'

⟨*in-spec*⟩ ::= **IN_PRE** | **IN_POST**

⟨*not-in-spec*⟩ ::= **NOT_IN_PRE** | **NOT_IN_POST**

⟨*start-state-property-spec*⟩ ::= **START_STATE**'(' AlphaNum ')'

⟨*state-property-spec*⟩ ::= **STATE**'(' AlphaNum ')'

⟨*final-state-property-spec*⟩ ::= **FINAL_STATE**'(' AlphaNum ')'

⟨*event-method-call*⟩ ::= String

⟨*prog-variable*⟩ ::= AlphaNum

⟨*db-table*⟩ ::= AlphaNum

⟨*db-column*⟩ ::= AlphaNum

C   YEROTH–ERP–3.0 MAINTENANCE VERIFICATION INTERFACE

Fig. 14: YEROTH–ERP–3.0 Maintenance Verification Interface.