`YERITH_QVGE` : A Framework for Verifying SQL Correctness Temporal Properties of [GUI] Software at Runtime

Xavier Noundou[1]

[1]Yaounde, Center region, Cameroon.

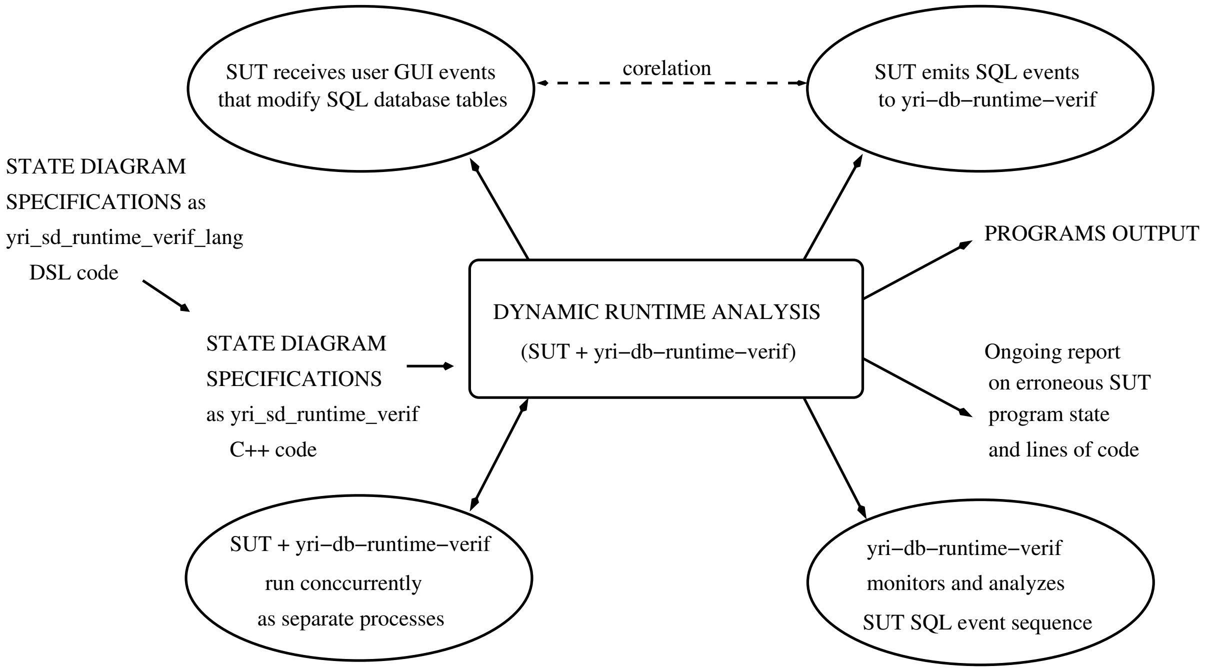
Contributing authors: yerith.xavier@gmail.com;

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: **`YRI-DB-RUNTIME-VERIF`**, to check SQL (Structure Query Language) [1] software correctness properties specified as temporal safety properties [2]. A temporal safety property specifies what behavior shall not occur, in a software, as sequence of program events. **`YRI-DB-RUNTIME-VERIF`** allows specification of a SQL temporal safety property by means of a state diagram mealy machine [3]. In **`YRI-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 [4] 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. **`YRI-DB-RUNTIME-VERIF`** doesn't generate false warnings; **`YRI-DB-RUNTIME-VERIF`** specifications are *not desirable (forbidden) specifications (fail traces)*. 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 YERITH–ERP–3.0 [5].

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

Fig. 1: **YRI-DB-RUNTIME-VERIF** WORKFLOW (diagram inspired from operation diagram in [6]).



1 Introduction

Table 1: YERITH–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

1.1 Motivations

This paper describes an effective dynamic analysis framework, based on runtime monitors specified in `C++` programs (implemented in the software library `yri_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` [7] 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` [8], `CppUnit` [9], 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` [10] provides some support for `Java`'record and replay' testing while `FROGLOGIC` [11] 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 YERITH–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 [12–16].

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 $\texttt{C}^{++}$ program. In particular, the example presented in Section 3 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 **`YRI-DB-RUNTIME-VERIF`** workflow.

1.2 Main Contributions

This paper presents 3 original main contributions:

- an industrial level quality framework (**`YRI-DB-RUNTIME-VERIF`**: https://github.com/yerithd/yri-db-runtime-verif), that solves temporal property verification by dynamic program analysis. **`YRI-DB-RUNTIME-VERIF`** makes use of the $\texttt{C}^{++}$ `Qt-Dbus` library, to input a *runtime monitor specification (`yri_sd_runtime_verif`)* as $\texttt{C}^{++}$ program code, that also enables software–library–plugin checks;

- a $\texttt{C}^{++}$ library: `yri_sd_runtime_verif` (https://github.com/yerithd/yri_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.

  `yri_sd_runtime_verif` only enables the specification of states diagrams specifications as ***not desirable (forbidden) behavior specifications (fail traces)***. Thus, **`YRI-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.

- An application of **`YRI-DB-RUNTIME-VERIF`** to check 1 temporal safety property error, found in the ERP FOSS YERITH–ERP–3.0.

Previous version of this paper

This paper extends a previous version [17], currently in conference proceedings SPLASH–ICTSS 2023 submission, with state diagram with more than 2 states, guarded conditions specifications, 2 new keywords for state diagram transition trace specification ("**in_sql_event_log**", "**not_in_sql_event_log**"), and **`YRI-DB-RUNTIME-VERIF`** binaries with more than 1 runtime monitor.

1.3 Overview

This paper is organized as follows: Section 2 presents formal definitions of the principal concepts used in this paper. Section 3 presents a motivating example that will be used throughout this paper to explain the presented concepts of this paper. Section 4 presents the software architecture of **`YRI-DB-RUNTIME-VERIF`**, our GUI dynamic analysis framework. Section 5 introduces the $\texttt{C}^{++}$ software library `yri_sd_runtime_verif` to model states diagrams, and reused by **`YRI-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 Formal Definitions

yri_sd_runtime_verif's formal description of the state diagram formalism follows ***Mealy machine*** [3] added with ***accepting states (final or erroneous states), and state diagram transition pre- and post-conditions***: "*state diagram mealy machine*". Another excellent, detailed with proofs and theory presentation of mealy automata [18] is available. In comparison to statechart [19], which is a ***visual formalism*** for states diagrams, yri_sd_runtime_verif doesn't support at time 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
- $\mathbf{S_0} \in S$: 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 *guarded condition.*
- $\mathbf{\Sigma}$: an input alphabet, $\mathbf{\Sigma}$:= $\{False, True\}$.
  $'False'$ means no input from SUT into **YRI-DB-RUNTIME-VERIF**.
  $'True'$ means any input could come from SUT.
- $\mathbf{\Lambda}$: an output alphabet (of program events $e_n (n \in \mathbb{N})$), ϕ the no program event. A program event generally corresponds to a function or method call at a SUT source code statement (or program point).
- $\delta : S \times C$: 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 ($\overline{c}$).
- $\mathbf{T} : S \times \Sigma \rightarrow S \times \Lambda$: a transition function that maps an input symbol to an output symbol and the next state.
- : a $2-ary$ relation that maps a state diagram transition to a guarded condition expression.
- $\mathbf{\Gamma}$: a set of accepting states; $\mathbf{\Gamma} \in S$.

For instance, for the motivating example described in Figure 2 we have:

- $\mathbf{S} = \{\mathrm{D}, \mathrm{E}\}$;
- $\mathbf{S_0} = \mathrm{D}$;
- $\mathbf{C} = \{\underline{Q0}, \overline{Q1}\}$;
- $\mathbf{\Sigma} = \{False, True\}$;
- $\mathbf{\Lambda} = \{\phi, \text{'SELECT.department'}\}$;
- $\delta = \{(\mathrm{D}, \underline{Q0}), (\mathrm{E}, \overline{Q1})\}$;
- $\mathbf{T} = \{((\mathrm{D}, False), (\mathrm{D}, \phi)), ((\mathrm{D}, True), (\mathrm{E}, \text{'SELECT.department'}))\}$;
- $\mathbf{\Gamma} = \{\mathrm{E}\}$

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:

- $\underline{Q0}$:= IN_PRE(X, Y) that means value "X" is in ($\in$) database column value set "Y".
- $\underline{Q0}$:= NOT_IN_PRE(X, Y) that means value "X" is not in ($\notin$) database column value set "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:

- $\overline{Q1}$:= IN_POST(A, B) that means value "A" is in ($\in$) database column value set "B".
- $\overline{Q1}$:= NOT_IN_POST(A, B) that means value "A" is not in ($\notin$) database column value set "B".

For state diagram mealy machines with more than* 2 *states, only the first transition has a pre-condition specification (**IN_PRE**, or **NOT_IN_PRE**). Each other transition only has a post-condition specification (**IN_POST**, or **NOT_IN_POST**). Since each state only has 1 outgoing (edge) state transition, the post-condition of the previous (incoming) state transition acts as the pre-condition of the next transition.

Definition 4.

A trace $T_n =< e^0, e^1, .., e^n >$ is a sequence of SUT events (or SUT program points) $e^{i, i \in \{0,..,n\}}$ of length n. $trace(D)$ is the trace of SUT events up to state D. For instance, for the motivating example described in Figure 2 we have: $trace(E) = trace(D), <$ 'SELECT.department' $>$.

Proposition 1: NO FALSE WARNINGS.

`yri_sd_runtime_verif` only allows 1 outgoing edge or transition for a state in its specifications, and for *not desirable (forbidden) behavior*, as illustrated in Figure 2. There is no need to specify the red colored edge in Figure 2 because it represents runtime cases where no input events arrive from SUT into **YRI-DB-RUNTIME-VERIF**. These 2 properties, together with algorithm '`YRI_trigger_an_edge_event(QString an_edge_event)`' (Listing 3) of `yri_sd_runtime_verif`, ensures that there are no false warnings during **YRI-DB-RUNTIME-VERIF** analyses. For example, the runtime monitoring or verification systems [12–16] may give false warnings.

2.1 Guarded Condition Expression Specification in `yri_sd_runtime_verif`

Guarded conditions expressions can be specified using one of the `yr_create_monitor_edge` method and a boolean expression of type `YR_CPP_BOOLEAN_expression`. An edge without an explicit guarded condition has an ***implicit*** '`[True]`' guarded condition on it. The implicit guarded condition '`[True]`' mustn't be identified as an implicit input event '$True$', as specified in Definition 1.

Guarded conditions are meant to be trace set specification on program events. For instance in Figure 2 (motivating example): "`[`**`in_set_trace`** `('DELETE.department.YRI_ASSET', STATE(D))]`"means that a SQL 'DELETE' event removing a department named 'YRI_ASSET' from MariaDB SQL table '`department`' must have occurred in the trace leading to state 'D', before event '`SELECT.department`' can be triggered. A guarded condition could have two practical forms:

- "[**in_set_trace** ('event', STATE(D))]" is equivalent to: 'event' $\in$ trace(D).
- "[**not_in_set_trace** ('event', STATE(D))]" is equivalent to: 'event' $\notin$ trace(D).

where 'event' is an input event (event $\in \mathbf{\Sigma}$) and 'D' a state diagram state ($D \in \mathbf{S}$).

Fig. 2: A motivating example, as current bug in YERITH–ERP–3.0.
$\underline{Q0}$:= NOT_IN_PRE(YRI_ASSET, department.department_name).
$\overline{\overline{Q1}}$:= IN_POST(YRI_ASSET, stocks.department_name).

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

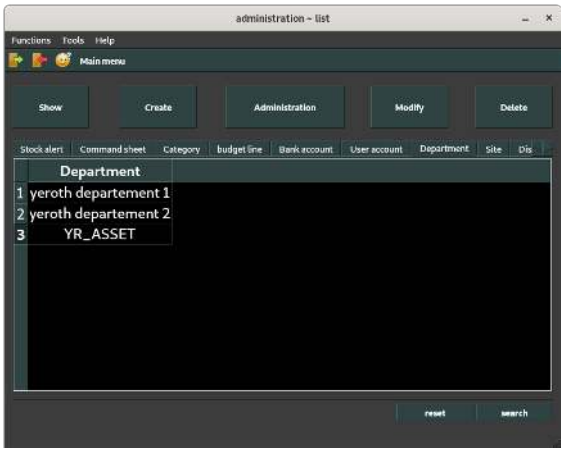


Fig. 4: YERITH–ERP–3.0 stock asset window listing some assets ($\overline{Q1}$).

3 Motivating Example: missing department definition

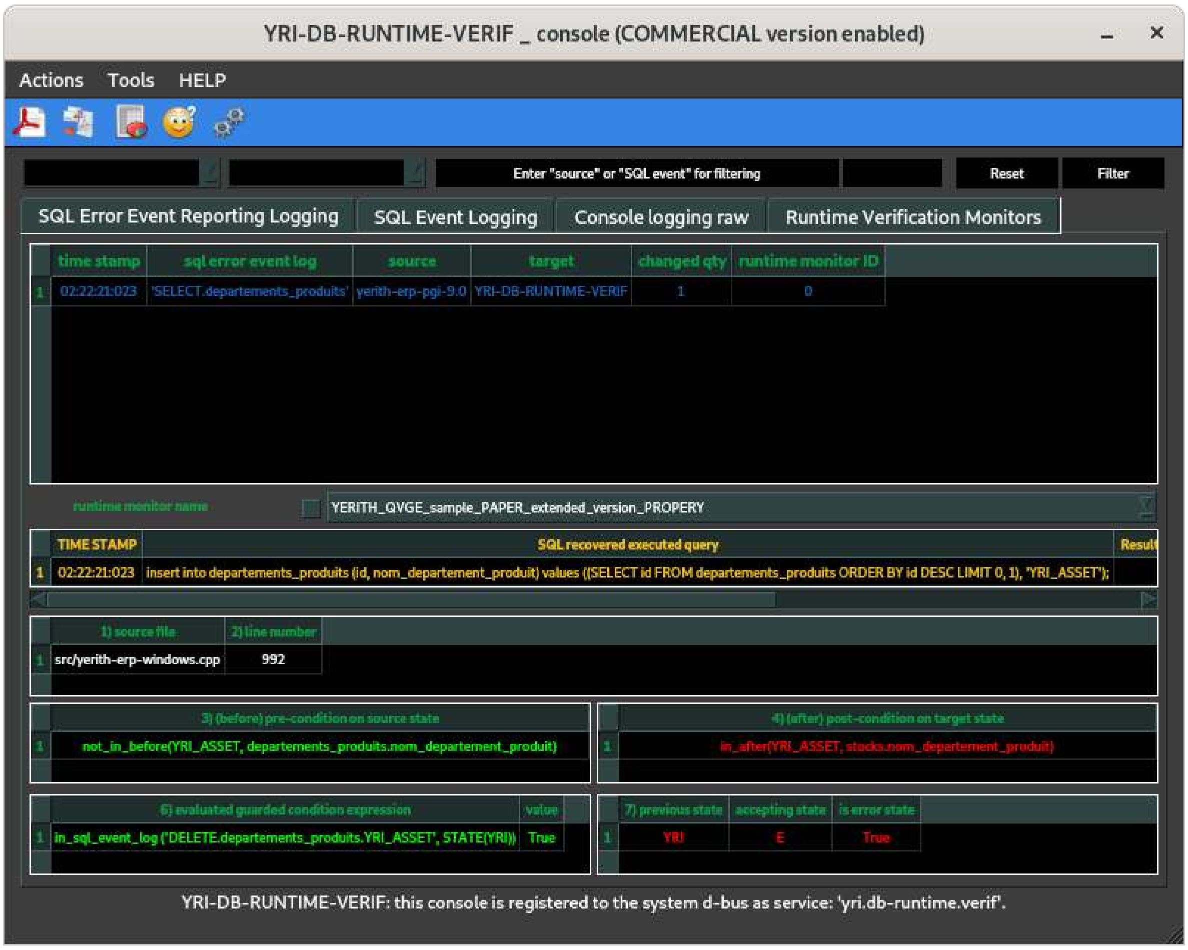
3.1 The Enterprise Resource Planing Software YERITH–ERP–3.0

YERITH–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. [5] for very small, small, medium, and large enterprises. YERITH–ERP–3.0 is developed using `C++` by means of the `Qt` development library. YERITH–ERP–3.0 is a large software with around 300 000 (three hundred thousands) of physical source lines of code. **`YRI-DB-RUNTIME-VERIF`** could be used for integration testing of YERITH–ERP–3.0, among different software modules.

3.2 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 'YRI_ASSET' in Figure 3 because there are still a stock asset listed within department 'YRI_ASSET', as illustrated in Figure 4. Figure 2

Fig. 5: **`YRI-DB-RUNTIME-VERIF`** graphical EDITOR viewing interface demonstrating that a final state has been reached (Section 6 analyzes these results).



illustrates the above temporal safety property as a simple state diagram.

3.2.1 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 $\underline{Q0}$ (as a predicate) in state 'D': **"NOT_IN_PRE(YRI_ASSET, department.department_name)"** means:

- ***a department named*** 'YRI_ASSET' ***is not in column*** 'department_name' ***of MariaDB SQL database table*** '`department`'. This might happen whenever button '`Delete`' in Figure 3 is pressed when item 'YRI_ASSET' is selected.

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

- ***a department named*** 'YRI_ASSET' ***is in column*** "`department_name`" ***of MariaDB SQL 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***.

Guarded Condition Expression

The guarded condition expression "[**in_set_trace** ('DELETE.department.YRI_ASSET', STATE(D))]"means a SQL 'DELETE' event removing a department named 'YRI_ASSET' from MariaDB SQL table '**department**' must have occurred in the trace leading to state 'D'.

Yri_sd_runtime_verif Specification Code

The source code specified in Listing 2 also illustrates a specification in C^{++} using software library yri_sd_runtime_verif of the state diagram specification above.

3.3 YRI-DB-RUNTIME-VERIF Analysis Report

The motivating example automaton in Figure 2 is analyzed by **YRI-DB-RUNTIME-VERIF** as follows:

- whenever department 'YRI_ASSET' is deleted in YERITH–ERP–3.0, as done in Figure 3, the runtime monitor state 'D' with a state condition $\underline{Q0}$ is entered
- when MySQL library (plugin) event 'SELECT.department' occurs, in Figure 3 because of YERITH–ERP–3.0 displaying the remaining product departments, the guarded condition for edge event 'SELECT.department' is automatically evaluated to '**True**' by C^{++} library yri_sd_runtime_verif, because no other guarded condition was specified by the developer
- yri_sd_runtime_verif enters the runtime monitor state to 'E' and state condition $\overline{Q1}$ via method YRI_trigger_an_edge_event(QString an_edge_event) because there are still assets (**yerith**_**asset**_3) left within product department 'YRI_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.

Fig. 6: **YRI-DB-RUNTIME-VERIF**: simplified software system architecture.

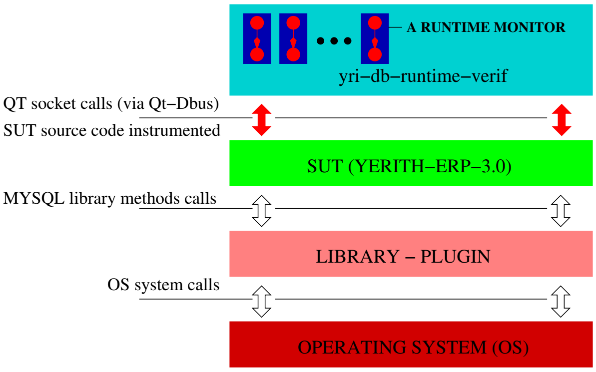
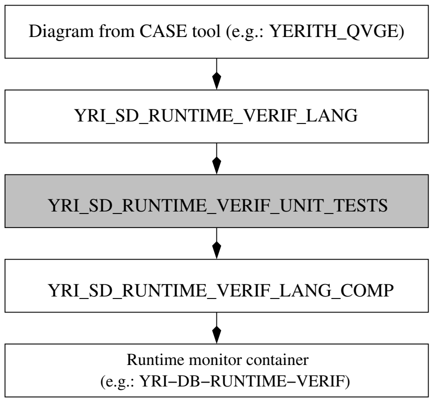


Fig. 7: `YERITH_QVGE` software library dependencies.



4 The Software Architecture of **YRI-DB-RUNTIME-VERIF**

4.1 Dynamic Analysis

4.1.1 SUT Source Code Instrumentation.

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

4.1.2 Debugging Information.

Each GUI manipulation of YERITH–ERP–3.0 in its instrumented source code part could generate a state transition within the analyzed runtime monitor state diagram in **YRI-DB-RUNTIME-VERIF**. Visualize "line 35" of Figure 5 to observe that a specific analysis message is sent to the console of **YRI-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

YRI-DB-RUNTIME-VERIF currently only processes the 4 SQL events in Table 2.

4.3 A Runtime Monitor (An Analysis Client)

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

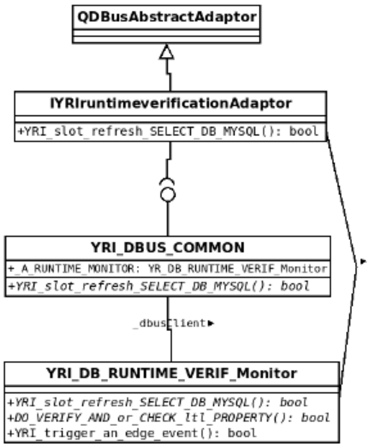
```
<!DOCTYPE node PUBLIC "-//freedesktop//DTD D-BUS Object
      Introspection 1.0//EN"
        "http://www.freedesktop.org/standards/dbus/1.0/introspect.
           dtd">
<node name="/YRruntimeverification">
  <interface name="com.yerith.rd.IYRruntimeverification">
    <method name="YRI_slot_refresh_SELECT_DB_MYSQL">
      <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>
```

An user (an analysis client) of **YRI-DB-RUNTIME-VERIF** needs to subclass class `YR_DB_RUNTIME_VERIF_Monitor`. The UML class diagram in Figure 8 displays the class structure of **YRI-DB-RUNTIME-VERIF**. `Qt-Dbus` communication adaptor `IYRruntimeverificationAdaptor`

Table 2: SQL Event Dbus Method Interface

SQL Event	Dbus Method Interface
DELETE	YRI_slot_refresh_DELETE_DB_MYSQL(QString, uint)
INSERT	YRI_slot_refresh_INSERT_DB_MYSQL(QString, uint)
UPDATE	YRI_slot_refresh_UPDATE_DB_MYSQL(QString, uint)
SELECT	YRI_slot_refresh_SELECT_DB_MYSQL(QString, uint)

Fig. 8: **`YRI-DB-RUNTIME-VERIF`**: simplified class diagram in UML [20].



shall be generated by the user of this library (on **`YRI-DB-RUNTIME-VERIF`** side) using `Qt-Dbus` command `qdbusxml2cpp` and an XML file, similar to the one displayed in Listing 1:

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 '**`YRI_trigger_an_edge_event(QString an_edge_event)`**' (Listing 3) of class '`YR_CPP_RUNTIME_MONITOR`' of `C`$^{++}$ library **`yri_sd_runtime_verif`** for each corresponding state diagram transition event.

Fig. 9: Class diagram in UML [20] to model a State Transition Diagram.

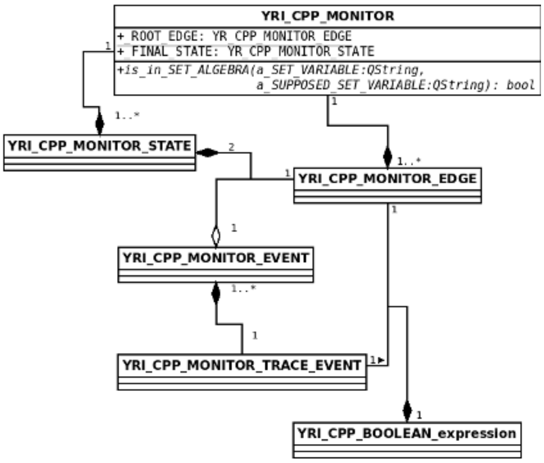
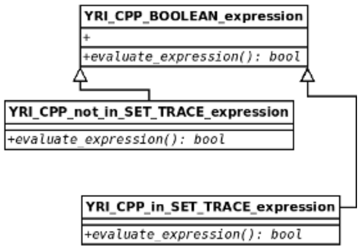


Fig. 10: Class diagram in UML [20] to model state diagram transition trace conditions in `yri_sd_runtime_verif` code.



Listing 2: `yri_sd_runtime_verif` C++ code modeling a current bug in YERITH–ERP–3.0 (Figure 2).

```
YRI_CPP_MONITOR_EDGE *a_last_edge_0 = create_yri_monitor_edge("D",
                                          "E",
                                          "select.departements_produits");

a_last_edge_0->get_SOURCE_STATE()->set_START_STATE(true);

a_last_edge_0->get_TARGET_STATE()->set_FINAL_STATE(true);

a_last_edge_0->set_PRE_CONDITION_notIN("YRI_ASSET",
                                          "departements_produits.nom_departement_produit");

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

YRI_register_set_final_state_CALLBACK_FUNCTION(&YRI_CALL_BACK_final_state);
```

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

5.1 Structure Of `yri_sd_runtime_verif`

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

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 `YRI-DB-RUNTIME-VERIF`.

Table 3 specifies which class is in `yri_sd_runtime_verif` code for each runtime monitor/state diagram element.

5.2 Methods for Pre- and Post-Condition Specifications

Table 4 illustrates methods for specifying pre– and post–conditions of a runtime monitor

Table 3: 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

Table 4: `yri_sd_runtime_verif` Methods for Pre-/Post-Condition Specification

Class YR_CPP_MONITOR_EDGE Methods	Utility
set_PRE_CONDITION_notIN(QString, QString)	sets a NOT IN DATABASE pre–condition
set_PRE_CONDITION_IN(QString, QString)	sets an IN DATABASE pre–condition
set_POST_CONDITION_notIN(QString, QString)	sets a NOT IN DATABASE post–condition
set_POST_CONDITION_IN(QString, QString)	sets an IN DATABASE pre–condition

state diagram transition. Each method takes in 2 arguments of string (`'QString'`) type: `'DB_VARIABLE'`, `'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 3

Listing 2 of the runtime monitoring specification stipulates for instance in its "line 12", as post-condition:

```
a_last_edge_0->
   set_POST_CONDITION_IN("YRI_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 `YRI_trigger_an_edge_event`

Listing 3 illustrates the pseudo–code of `yri_sd_runtime_verif` SUT event processing method `YRI_trigger_an_edge_event(QString an_edge_event)`. '`YRI_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 `yri_sd_runtime_verif` states diagrams

Listing 3: C++ Pseudo-code for `YRI_trigger_an_edge_event(QString an_edge_event)`: `yri_sd_runtime_verif` method for triggering state diagram events (edges or transitions).

```
bool MONITOR::YRI_trigger_an_edge_event(QString an_edge_event)
{
    MONITOR_EDGE cur_OUTGOING_EDGE = _cur_STATE.outgoing_edge();

    if (cur_OUTGOING_EDGE.evaluate_GUARDED_CONDITION_expression() &&
        (an_edge_event == cur_OUTGOING_EDGE.edge_event_token()))
    {
        bool precondition_IS_TRUE = cur_OUTGOING_EDGE
                        .CHECK_SOURCE_STATE_PRE_CONDITION(_cur_STATE);

        if (precondition_IS_TRUE)
        {
            set_current_triggered_EDGE(cur_OUTGOING_EDGE);

            MONITOR_STATE a_potential_accepting_state =
                cur_OUTGOING_EDGE.get_TARGET_STATE();

            if (CHECK_whether__STATE__is__Final(a_potential_accepting_state))
            {
                CALL_BACK_final_state_FUNCTION(a_potential_accepting_state);
            }
            return true;
        }
    }
    return false;
}
```

shall have only 1 outgoing edge (transition), by specification and construction, as explained in Proposition 2 in Section 2.

The algorithm in Listing 3 demonstrates that, given correct trace and event information from SUT, `yri_sd_runtime_verif` always exactly matches the user specification. Thus never giving false warnings.

Table 5: SUT (YERITH–ERP–3.0) "**`YRI-DB-RUNTIME-VERIF`** graphical EDITOR" error states (Figure 5).

SQL EVENT	SUT PROGRAM POINT (TRACE)
"SELECT.department"	"src/yerith-erp-windows.cpp:992"

6 Evaluation

The main experimental results in this paper demonstrate the efficacy of our tool to find errors in the SUT (YERITH–ERP–3.0), presented in Subsection 3.2.

Qualitative Results.

*SUT (*YERITH–ERP–3.0*) TRACING.*

Table 5 illustrates SUT source code trace information as presented in **`YRI-DB-RUNTIME-VERIF`** console output in Figure 5. We have translated from French to English the MariaDB SQL table names.

SQL EVENT CALL SEQUENCE.

A careful observation of the output in Figure 5 illustrates the following sequence:

- **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 = 'YRI_ASSET';`
- **line** 28**, line** 29**:** evaluation of the pre–condition $\underline{Q0}$ of state D stating that product department 'YRI_ASSET' is not existent evaluates to 'TRUE' (triggering of event "'`DELETE.department.YRI_ASSET`' " by pressing of SUT button '`Delete`' at **line** 21 has removed any asset department name 'YRI_ASSET').
  `*[YRI_CPP_MONITOR::CHECK_PRE_CONDITION_notIN:] precondition_IS_TRUE: True   **`
- **line** 31**, line** 32**:** checking post–condition $\overline{Q1}$ in state E (there are still stocks in stock department 'YRI_ASSET') evaluates to 'TRUE', thus state E is reached as an accepting state, because department name 'YRI_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 = 'YRI_ASSET';"`
  `*[YRI_CPP_MONITOR::CHECK_post_condition_IN:] postcondition_IS_TRUE: True   **`

Runtime Performance.

`YRI-DB-RUNTIME-VERIF` and `yri_sd_runtime_verif` don't incur a runtime supplemental overhead to the SUT, apart from emitting SQL events from SUT to **`YRI-DB-RUNTIME-VERIF`** as they occur, since no hand–shaking mechanism is used between **`YRI-DB-RUNTIME-VERIF`** and the SUT. The emission of an SQL event from SUT to **`YRI-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

- **SUT source code instrumentation with runtime monitor specification.** "Clara" [12] 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 [13–16] work as similar as the Clara framework does.

  `YRI-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 increasing runtime execution of the SUT. **`YRI-DB-RUNTIME-VERIF`** specifies the runtime monitor as a *state diagram mealy machine*, a subset of typestate, specified as a `C`$^{++}$ program, and extended with accepting states and state transition pre- and post-condition.

- **SUT binary code instrumentation with a runtime monitor.** With `tracerory` [6, 21]", **Jon Eyolfson and Patrick Lam** use runtime program binary code instrumentation technique in `INTEL-pin` [22] to instrument running programs for purposes of detecting unread memory. I.e., `tracerory` doesn't generate itself a runtime monitor, it uses `INTEL-pin` [22] to generate a runtime monitor for its verification purposes. "`Purify`" [23] doesn't allow for SUT user correctness property specification. 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.

  In contrast, with **`YRI-DB-RUNTIME-VERIF`**, the user instruments the source code of the analyzed `C`$^{++}$ program at compile time with SQL events emitting code. **`YRI-DB-RUNTIME-VERIF`** could also be used to perform memory write integrity check like `tracerory`. **`YRI-DB-RUNTIME-VERIF`** monitors program trace events at database level (also source code statement level by mapping to SQL query statements), and not at program counter level as `tracerory` does. **`YRI-DB-RUNTIME-VERIF`** inputs a SUT correctness property specification as a *state diagram mealy machine* (as a subset of `LTL` [2]); **`YRI-DB-RUNTIME-VERIF`** is a runtime monitor container; **`YRI-DB-RUNTIME-VERIF`** also generates itself a runtime monitor whereas `tracerory` doesn't.

- **Source code automated test case generation for `C`$^{++}$ with `TDIOHS` [24], or FOSS library `libfsmtest` [7].**

- **Specification as set interface operations.** "Hob" [25, 26] is a program verification framework that enables to: characterize effects of program statement on data structures by means of all (∀, ∃, etc.) algebra abstract set interface operations; and to check that these characteristics hold or not, using static analyses.

  `YRI-DB-RUNTIME-VERIF` is a program verification framework that enables to: characterize effects of program statements (via SQL [4] (Structure Query Language) on database table columns by means of set interface operations (∈, ∉); and to check that these characteristics hold or not, using dynamic runtime analysis.

- **Concurrent Event Stream Analysis.**

  "`DejaVu` " [27] 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" [28] 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" [29] repairs in web applications (*online*), event race errors, a kind of safety error.

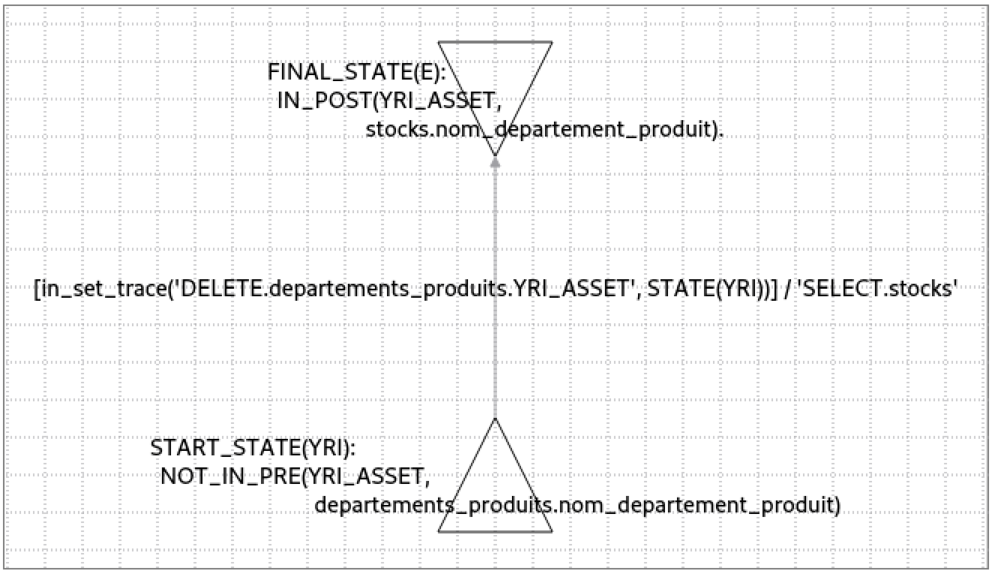
States diagrams specifications are implemented as C++ program monitors using C++ library yri_sd_runtime_verif. **YRI-DB-RUNTIME-VERIF** outputs a developer given (by means of a callback function, as seen in 'line 15' in Listing 2) string message [1] in case an accepting state was entered, and a trace event of YERITH–ERP–3.0 leading to it. **YRI-DB-RUNTIME-VERIF**'s monitors need not store data, as DejaVu monitors must. **YRI-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 **YRI-DB-RUNTIME-VERIF**.

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

Fig. 11: **A Mealy Machine State Diagram Specified Using `yri_sd_runtime_verif` Specification Language.**

```
yri_sd_mealy_automaton_spec yri_missing_department
{
 START_STATE(d):NOT_IN_PRE(YRI_ASSET,department.department_name)
  ->[in_sql_event_log('DELETE.departement.YRI_ASSET',STATE(d))]/'SELECT.department'->
   ERROR_STATE(e):IN_POST(YRI_ASSET,stocks.department_name).
}
```

Fig. 12: '`YERITH_QVGE`' model for the example specification in Figure 11.



8 Conclusion And Future Work

This paper has presented a lightweight C++ `Qt-Dbus` [30] tool to check a program against a runtime monitor using set interface operations ($\in$, $\notin$) on program statement: **`YRI-DB-RUNTIME-VERIF`**. **`YRI-DB-RUNTIME-VERIF`** doesn't generate false warnings; **`YRI-DB-RUNTIME-VERIF`** specifications are *not desirable (forbidden) specifications (fail traces)*. Since the concurrent communication between **`YRI-DB-RUNTIME-VERIF`** and a program occurs over the RPC (Remote Procedure Call) instance `Dbus`, a runtime monitor could be checked against programs written in any programming language or framework, as long as they emit the necessary SQL events to **`YRI-DB-RUNTIME-VERIF`**.

Future work would be a tool-chain to validate `yri_sd_runtime_verif` models as represented in this paper.

Also, the author of this paper has developed a graphical drawing tool (`YERITH_QVGE`) for in Section 2 defined state diagrams. A model of `YERITH_QVGE` is shown in Figure 12. It is an extension of the FOSS (Free and Open Source Software) Qt **`Graphviz`** [31] drawing tool `QVGE` [32]. `YERITH_QVGE` generates, from a model, an input file for the compiler `yri_sd_runtime_verif_lang_comp`.

Listing 4: '**DO_VERIFY_AND_or_CHECK_ltl_PROPERTY**': **`YRI-DB-RUNTIME-VERIF`**'s overridden method for processing SUT event stream `C`$^{++}$ pseudo-code.

```
bool DO_VERIFY_AND_or_CHECK_ltl_PROPERTY(
    QString sql_table_NAME,
    SQL_CONSTANT_IDENTIFIER cur_SQL_command)
{
    switch (cur_SQL_command)
    {
    case SELECT:

        if ("department" == sql_table_NAME))
        {
            return YRI_trigger_an_edge_event("'select.department'");
        }
        break;

    default:
            break;
    }

    return false;
}
```

A Processing of SUT Event Stream By An Analysis Client

Listing 4 illustrates the pseudo-code of **`YRI-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 '`YRI_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 '**`YRI_trigger_an_edge_event(QString an_edge_event)`**' of class '`YRI_CPP_RUNTIME_MONITOR`' of `C`$^{++}$ library `yri_sd_runtime_verif` for each corresponding state diagram transition event.

B YRI_SD_RUNTIME_VERIF SPECIFICATION LANGUAGE

Fig. 13: **Grammar in Backus–Naur Form (BNF) of `yri_sd_runtime_verif` Mealy Machine State Diagram Specification Language.**

⟨*specification*⟩ ::= **yri_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-call*⟩

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

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

⟨*sut-state-spec*⟩ ::= ⟨*start-state-property-spec*⟩
  | ⟨*start-state-property-spec*⟩ ':' ⟨*algebra-set-specification*⟩
  | ⟨*state-property-spec*⟩ ':' ⟨*algebra-set-specification*⟩
  | ⟨*final-state-property-spec*⟩ ':' ⟨*algebra-set-specification*⟩
  | ⟨*final-state-auto-property-spec*⟩ ':' ⟨*algebra-set-specification*⟩ ':' ⟨*recovery-sql-query-spec*⟩

⟨*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-sql-event-log*⟩ ::= **in_sql_event_log**'(' ⟨*event-call*⟩ ',' ⟨*state-property-specification*⟩ ')'

⟨*not-in-sql-event-log*⟩ ::= **not_in_sql_event_log**'(' ⟨*event-call*⟩ ',' ⟨*state-property-specification*⟩ ')'

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

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

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

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

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

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

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

⟨*final-state-auto-property-spec*⟩ ::= **END_STATE_AUTO**'(' AlphaNum ')'
  | **FINAL_STATE_AUTO**'(' AlphaNum ')'
  | **ERROR_STATE_AUTO**'(' AlphaNum ')'

⟨*recovery-sql-query-spec*⟩ ::= **recovery_sql_query**'(' ⟨*db-table*⟩ ',' ⟨*sql-recovery-query*⟩ ')'

⟨*sql-recovery-query*⟩ ::= String

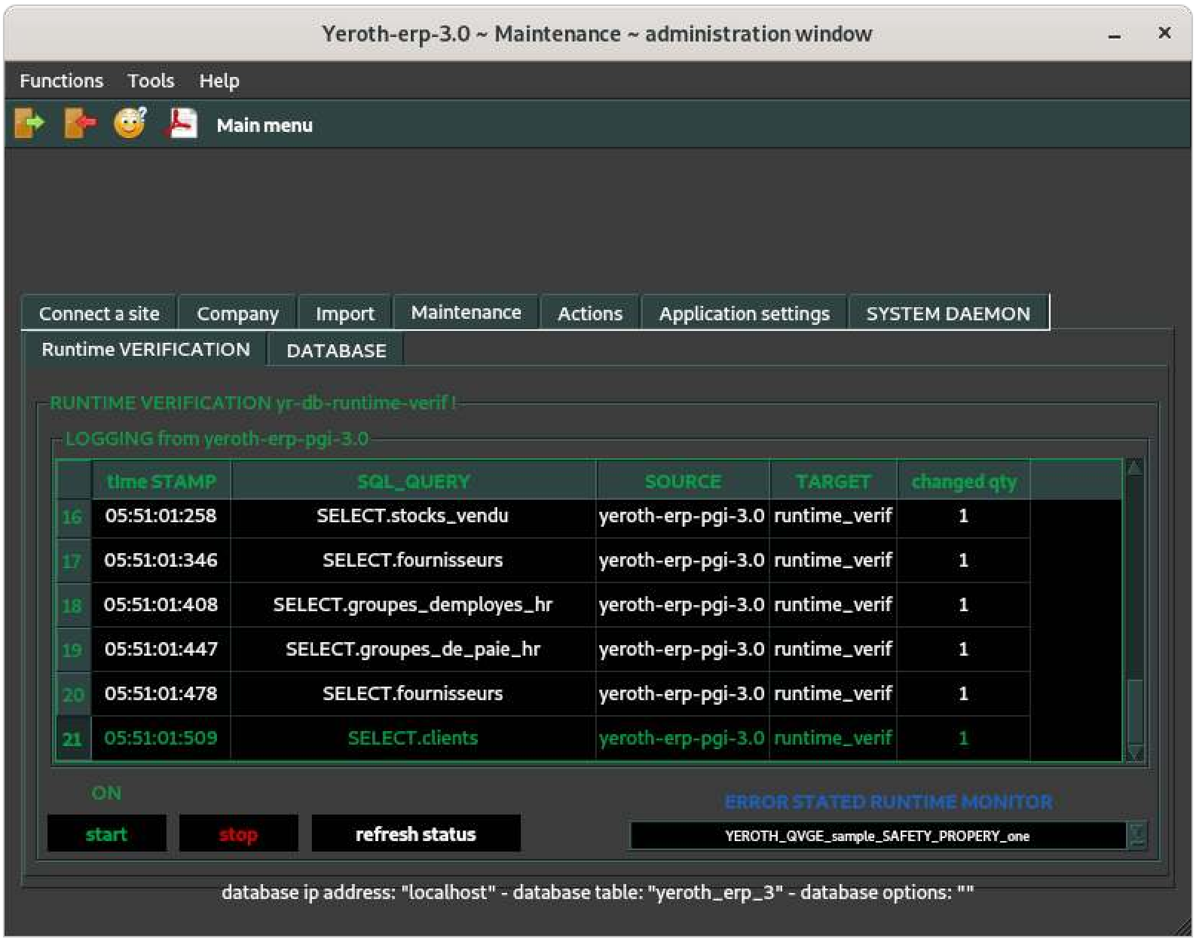
⟨*event-call*⟩ ::= String

⟨*prog-variable*⟩ ::= AlphaNum

⟨*db-table*⟩ ::= AlphaNum

⟨*db-column*⟩ ::= AlphaNum

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



C YERITH–ERP–3.0 MAINTENANCE VERIFICATION INTERFACE

References

[1] Wikipedia.org: SQL - Wikipedia. https://en.wikipedia.org/wiki/SQL. Accessed last time on February 08, 2023 at 12:00 (2023)

[2] 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

[3] Wikipedia.org: Mealy machine. https://en.wikipedia.org/wiki/Mealy_machine. Accessed last time on Dec 15, 2022 at 12:00 (2022)

[4] MariaDB.org: MariaDB Foundation - MariaDB.org. https://www.mariadb.org. Accessed last time on June 24, 2022 at 12:20 (2022)

[5] Noundou, X.: YERITH–ERP–PGI–3.0 Doctoral Compendium. https://archive.org/download/yerith-erp-pgi-compendium_202206/JH_NISSI_ERP_PGI_COMPENDIUM.pdf. Accessed last time on January 21, 2023 at 23:24 (2022)

[6] 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 (2013)

[7] 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, Cham (2022)

[8] doc.qt.io/qt-5: Qt 5.15. https://doc.qt.io/qt-5. Accessed last time on Dec 22, 2022 at 12:40 (2022)

[9] https://freedesktop.org/wiki/Software/cppunit: cppunit. https://doc.qt.io/qt-5/qtdbus-index.html. Accessed last time on January 01, 2023 at 12:00 (2022)

[10] 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/10.1145/367845.368073

[11] froglogic.com: Home • froglogic. https://www.froglogic.com/home. Accessed last time on Dec 18, 2022 at 20:00 (2022)

[12] Bodden, E., Hendren, L.: The clara framework for hybrid typestate analysis. International Journal on Software Tools for Technology Transfer (STTT) **14**, 307–326 (2012). 10.1007/s10009-010-0183-5

[13] Butkevich, S., Renedo, M., Baumgartner, G., Young, M.: Compiler and tool support for debugging object protocols. In: SIGSOFT '00/FSE-8 (2000)

[14] Allan, C., Avgustinov, P., Christensen, A.S., Dufour, B., Goard, C., Hendren, L.J., Kuzins, S., Lhoták, J., Lhoták, O., 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

[15] Bodden, E.: J-LO - A tool for runtime-checking temporal assertions. Diploma thesis, RWTH Aachen University (November 2005). https://www.bodden.de/pubs/bodden05jlo.pdf

[16] 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

[17] Noundou, X.: Yr_db_runtime_verif: a framework for verifying sql correctness properties of gui software at runtime (2023). https://archive.org/download/yri_ictss_2023/yri_ictss_2023.pdf

[18] Peleska, J., Huang, W.-l.: Test Automation; Foundations and Applications of Model-based Testing. https://www.informatik.uni-bremen.de/agbs/jp/papers/test-automation-huang-peleska.pdf. Accessed last time on May 06, 2023 at 12:00 (2021)

[19] Harel, D.: Statecharts: a visual formalism for complex systems. Science of Computer Programming **8**(3) (1987)

[20] Booch, G., Rumbaugh, J., Jacobson, I.: Unified Modeling Language User Guide, The (2nd Edition) (Addison-Wesley Object Technology Series). (2005)

[21] Eyolfson, J.: Tracerory; Dynamic Tracematches and Unread Memory Detection for C/C++. (2012). MASTER OF APPLIED SCIENCES (MASc). https://hdl.handle.net/10012/6206

[22] Luk, C.K., Cohn, R.S., Muth, R., Patil, H., Klauser, A., Lowney, P.G., Wallace, S., Reddi, V.J., Hazelwood, K.M.: Pin: building customized program analysis tools with dynamic instrumentation. In: PLDI '05 (2005)

[23] Hastings, R.O., Joyce, B.A.: Fast detection of memory leaks and access errors. (1991)

[24] Badban, B., Fränzle, M., Peleska, J., Teige, T.: Test automation for hybrid systems. In: Third International Workshop on Software Quality Assurance (SOQUA 2006), pp. 14–21 (2006)

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

[26] Lam, P.: The Hob System for Verifying Software Design Properties. (2007)

[27] Havelund, K., Peled, D., Ulus, D.: Dejavu: A monitoring tool for first-order temporal logic, pp. 12–13 (2018). https://doi.org/10.1109/MT-CPS.2018.00013

[28] 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, Cham (2022)

[29] 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

[30] doc.qt.io/qt-5/qtdbus-index.html: Qt D-Bus. https://doc.qt.io/qt-5/qtdbus-index.html. Accessed last time on Dec 22, 2022 at 12:40 (2022)

[31] graphviz.org: DOT Language | Graphviz. https://graphviz.org/doc/info/lang.html. Accessed last time on JUNE 8, 2022 at 12:30 (2022)

[32] showroom.qt.io: QVGE; Qt Visual Graph Editor | Showroom. https://showroom.qt.io/qvge-qt-visual-graph-editor. Accessed last time on Jun 27, 2022 at 12:40 (2022)