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

Xavier Noumbissi Noundou

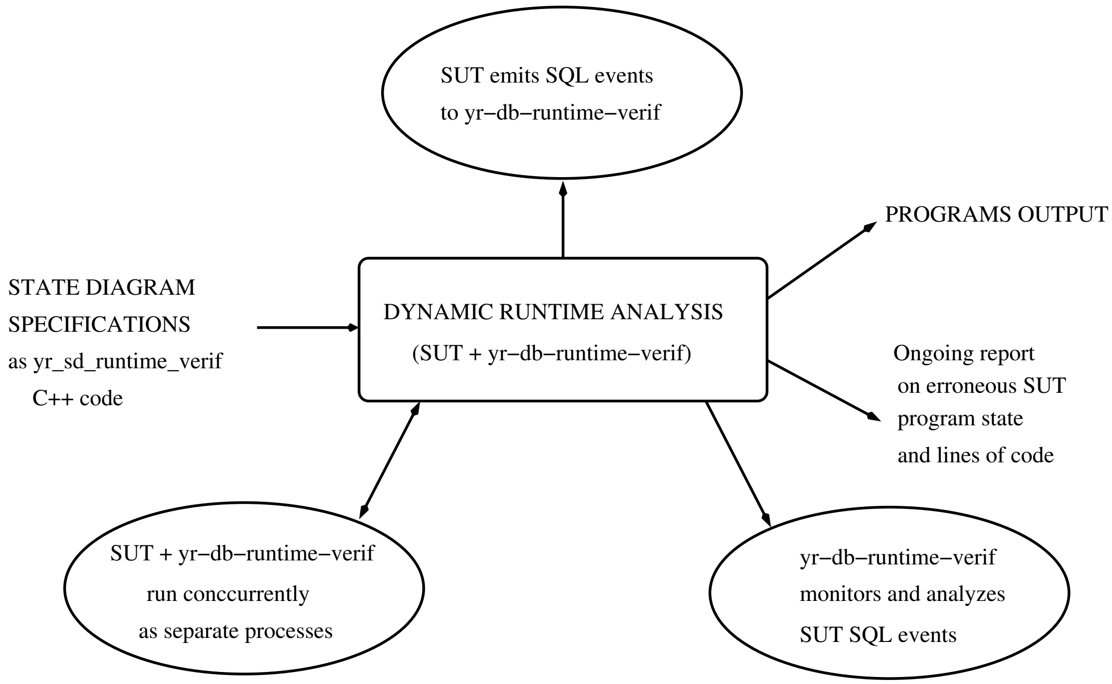
National Advanced School of Engineering, Yaoundé, Center Region, Cameroon
yeroth.d@gmail.com

Abstract. Software design properties are essential to maintain quality by continuous and regressive integration testing. This paper presents an effective and lightweight C++ program verification framework: **`YR_DB_RUNTIME_VERIF`**, to check SQL (Structure Query Language) [23] software design properties specified as temporal safety properties [9]. 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 [22]; 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 ($\exists$, $\in$, $\notin$), 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 [20] 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. 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 [21]***.

Keywords: computer software program analysis · computer software dynamic program analysis · software integration testing with SQL and GUI · integration testing with SQL and Qt–Gui

1 Introduction

Fig. 1: **YR_DB_RUNTIME_VERIF** operation.



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. *EventRaceCommander* [2] repairs, by dynamic runtime analysis, event race errors, a kind of temporal safety error, in web applications (thin clients). The testing of combinations between GUI windows (as thick–client) 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 authors of this paper.

Unit testing for GUI widgets is available by use of "NUnit" test frameworks like e.g. `Qt-Test` [1],`CppUnit` [14], etc.. Software test across GUI widgets (and MySQL queries) is however limited in support by these "NUnit" framework ! To the best of the knowledge of the authors of this paper, `DejaVu` [4] provide some support for `Java`'record and replay' testing while `FROGLOGIC` [10] provide support for `C++` GUI software 'record and replay' technology for testing thick–client GUI. '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.

As we will see in the related work, section 7, of this paper, most software design property checking framework 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 system with an impressive number of database tables: FOSS **YEROTH−ERP−3.0** for instance has about $300\,000$ lines of physical source code, 34 used SQL tables, and around 290 MariaDB SQL table columns. 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. Former work that uses runtime monitoring assumes for a sequential program, or an abstraction of the program as one single source code, on which program analysis is performed [19,3,16,5,8,6].

The program analysis technique the authors of this paper present 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. Figure 1 shows a high level overview of **YR_DB_RUNTIME_VERIF** operation.

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 ($\exists$, $\in$, $\notin$) for state diagram program state specification as pre- and post-conditions.

   `yr_sd_runtime_verif` only enables the specification of states diagrams specifications as **unfeasible behavior specifications**. 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

Fig. 2: A motivating example, as current bug in **YEROTH–ERP–3.0**.
$\underline{Q0}$:= NOT_IN(YR_ASSET, product_department.department_name).
$\overline{Q1}$:= DB_IN(YR_ASSET, stocks.department_name).

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

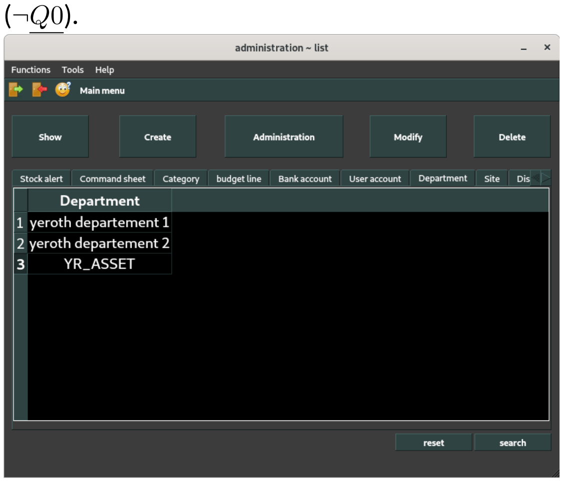
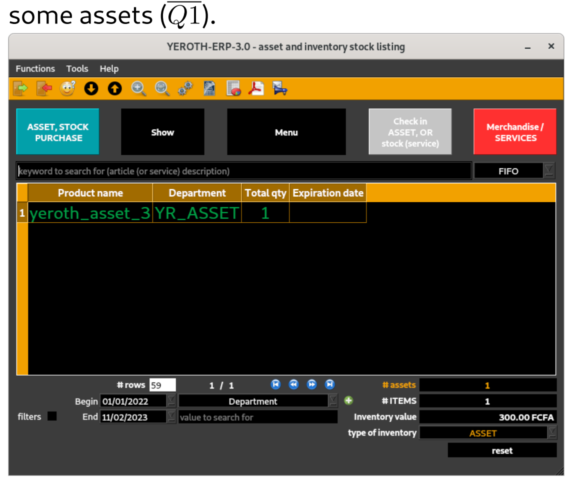


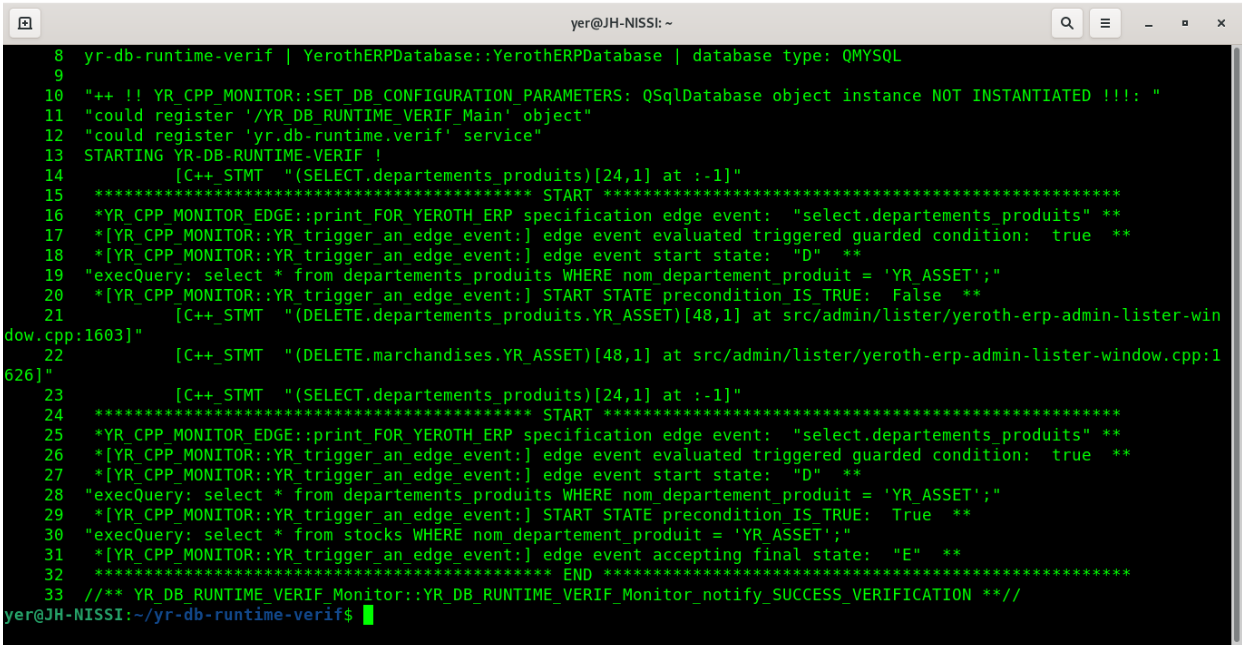
Fig. 4: **YEROTH–ERP–3.0** stock asset window listing some assets ($\overline{Q1}$).



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. [21] 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.



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 $\overline{Q0}$ (as a predicate) in state 'D': **'NOT_IN(YR_ASSET, product_department.department_name'** means:

1. ***a department named*** 'YR_ASSET' ***is not in column*** 'department_name' ***of database table*** 'product_department'.

Similarly, the post-condition $\overline{Q1}$ (as a predicate) **'DB_IN(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.product_department' denotes that when in 'D', a SQL 'select' on database table 'product_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.product_department`' occurs, in Figure 3 because of **YEROTH−ERP−3.0** displaying the remaining product departments, the guarded condition for edge event '`select.product_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 $Q1$ via method `YR_trigger_an_edge_event(QString an_edge_event, YR_CPP_BOOLEAN_expression *bool_GUARDED_CONDITION)`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

3 Formal Definitions

`yr_sd_runtime_verif`'s formal description of the state diagram formalism follows ***Mealy machine*** [22] added with ***accepting states (final or erroneous state), and state diagram transition pre- and post-conditions***. In comparison to statechart [12], which is a ***visual formalism*** for states diagrams, `yr_sd_runtime_verif` doesn't support for instance the following features:

1. hierarchical states (composite state, submachine state, etc.)
2. timing conditions
3. etc.

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

- $\mathbf{S}$: 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.: $\overline{Q1}$).
- $\mathbf{\Sigma}$: an input alphabet
- $\mathbf{\Lambda}$: an output alphabet
- $\delta : S \times C$: 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.
- $\mathbf{\Gamma}$: a set of accepting states.

For instance, for the motivating example described in Figure 2 we have:
$\mathbf{S} = \{\text{D}, \text{E}\}$; $\mathbf{S_0} = \text{D}$; $\mathbf{C} = \{\underline{Q0}, \overline{Q1}\}$; $\mathbf{\Sigma} = \{True\}$; $\mathbf{\Lambda} = \{\text{'select.product\_department'}\}$; $\delta = \{(\text{D}, \underline{Q0}), (\text{E}, \overline{Q1})\}$; $\mathbf{T} = \{((\text{D}, True),\ (\text{E}, \text{'select.product\_department'}))\}$; $\mathbf{\Gamma} = \{\text{E}\}$.

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(X, Y) that means value "X" is in ($\in$) database column value set "Y".
- $\underline{Q0} :=$ NOT_IN(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:

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

Definition 4. A trace $T_n =< e^0,\ e^1,\ ..\ e^n >$ is a sequence of SUT events $e^i (i \in \{0, .., n\})$ 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:

- $trace(D) =<>$, $trace(E) =<$ select.product_department $>$.

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

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

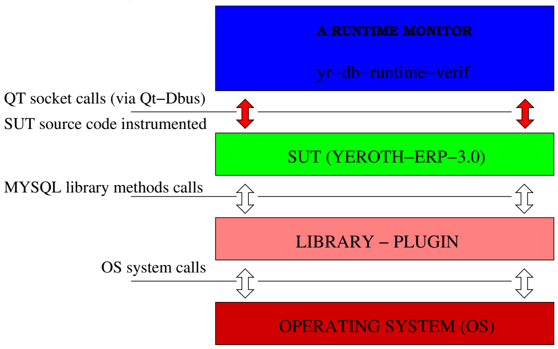
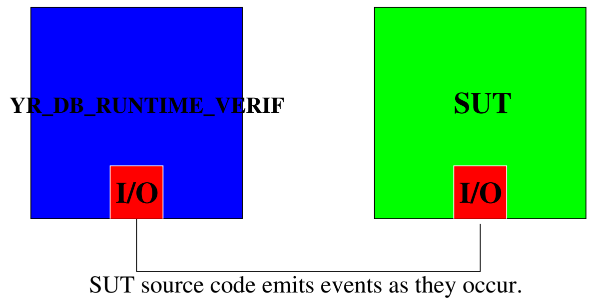


Fig. 7: **YR_DB_RUNTIME_VERIF** and SUT socket communication.



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 33 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 31 is for an accepting (final) state of the state diagram specification of the motivating example presented in Figure 2.

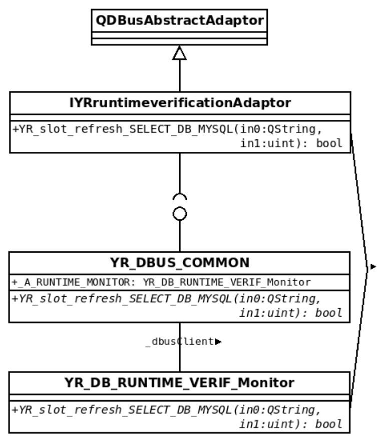
4.2 A Runtime Monitor (An Analysis Client)

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

```
<!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 [7].



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 [7] to model a State Transition Diagram.

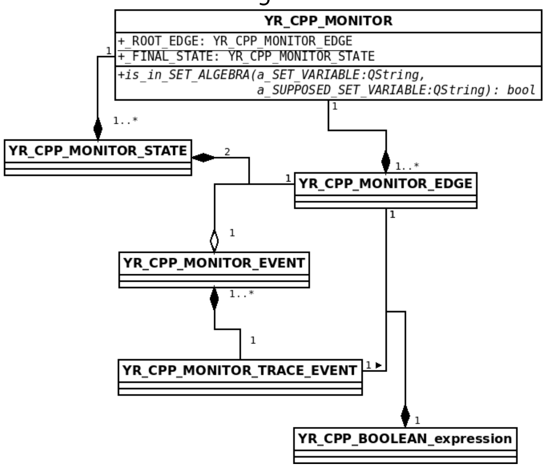
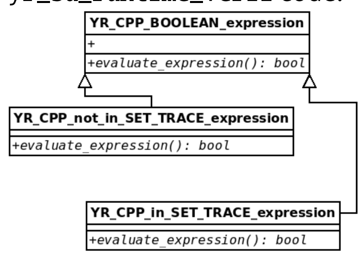


Fig. 10: Class diagram in UML [7] to model state diagram transition trace conditions in `yr_sd_runtime_verif` code.



5.1 Structure Of `yr_sd_runtime_verif`

`yr_sd_runtime_verif` is a state diagram C++ library the authors 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`.

Table 1 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 2 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

Listing 1.4: `yr_sd_runtime_verif` C++ code modeling a current bug in **YEROTH−ERP−3.0**.

```
if (0 != _a_runtime_monitor)
{
    YR_CPP_MONITOR_EDGE *a_last_edge_0 = _a_runtime_monitor->create_yr_monitor_edge("D", "E");

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

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

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

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

    YR_CPP_MONITOR_EVENT *a_last_edge_event_0 = a_last_edge_0->set_EDGE_EVENT("select.departements_produits");

    _a_runtime_monitor->YR_register_set_final_state_CALLBACK_FUNCTION(&YR_CALL_BACK_final_state);
}
```

Table 1: 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 2: `yr_sd_runtime_verif` Methods for Pre-/Post-Condition Specification

Runtime Monitor State (class YR_CPP_MONITOR_STATE) 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

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 11, as post-condition:

```
a_last_edge_0->get_END_STATE()->
  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 Triggering of SUT Events During Runtime Analysis

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.

The analysis client then calls method '**YR_trigger_an_edge_event(QString an_edge_event, YR_CPP_BOOLEAN_expression *bool_GUARDED_CONDITION)**' of class 'YR_CPP_RUNTIME_MONITOR' of C++ library yr_sd_runtime_verif for each corresponding state diagram transition event. Method '**YR_trigger_an_edge_event(QString an_edge_event, YR_CPP_BOOLEAN_expression *bool_GUARDED_CONDITION)**' first evaluates a state diagram transition guarded condition before it can trigger the corresponding state diagram transition event.

6 Evaluation

6.1 Qualitative Results

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.

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

- **line** 23**, line** 28**:** at state D, execution of the state diagram event (SUT button 'Delete' has been pressed at **line** 21) "select.product_department ":

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

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

  ```
  precondition_IS_TRUE:  True
  ```

- **line** 30**, line** 31**:** 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:

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

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 checking 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 (Listing 1.2) on it).

7 Related Work

1. **Event stream processing.** "Beep Beep 3" [11] is a Java framework enabling developers to analyze data coming as event from sources; the analysis takes place in so called *processors*, that are computations on data, specified as Java code. "Beep Beep 3" could also define a mealy machine as a processor (or also called runtime monitor).

   YR_DB_RUNTIME_VERIF's runtime monitor specification is done in the C++ language; but a program written in any programming language can be verified against the same runtime monitor as long as it emits necessary events via the RPC Qt-Dbus interface.

   YR_DB_RUNTIME_VERIF simply checks a post-condition as a set inclusion operation to verify final state acceptance; "Beep Beep 3" can define a more complex *processor computation* to check a final state acceptance.

2. **SUT source code instrumentation with specifications.** The Clara framework for hybrid typestate analysis [6] enables developers to express software design 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 cheking***. The Clara framework weaves (instruments), and annotates a program with runtime monitors using AspectJ, then tries to optimize he 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 [19,3,16,5,8] 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–checking mechanism, thus not augmenting runtime execution of the SUT as Clara does.

   YR_DB_RUNTIME_VERIF specifies the runtime monitor as a state diagram, a subset of typestates, specified as a C++ program, and augmented with accepting states and state transition pre- and post-condition.

3. **Specification as set interface operations.** The Hob system for verifying software design properties [17,18]: **Hob** is a program verification framework that enables developers to: characterize effects of program statement on data structures by means of all ($\forall$, $\exists$, etc.) 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 developers to: characterize effects of program statements (via SQL [20] (Structure Query Language) on database table columns by means of set interface operations ($\exists$, $\in$, $\notin$); and to check that these characteristics hold or not, using dynamic runtime analysis.

4. **POST–MORTEM vs ONLINE safety property analysis.** "DejaVu: a monitoring tool for first-order temporal logic [13]". DejaVu enables developers to check software systems safety temporal property expressed in **first-order past linear-time temporal logic (FO-PLTL)** for events that carry data. DejaVu inputs a trace log and a FO-PLTL formula, and outputs a boolean value for each position in the inputted trace. The drawback of DejaVu is that users must have a very good description formal language background, i.e., be expert in formal verification.

   YR_DB_RUNTIME_VERIF inputs a system unfeasible specification as a state diagram (as a subset of FO-PLTL) and outputs a 'yes', and a trace event of **YEROTH–ERP–3.0** leading to a final state.

   YR_DB_RUNTIME_VERIF events also carry data (database table and column name, records quantity modified by current SUT event).

8 Conclusion

This paper has presented a lightweight `C++` `Qt-Dbus` [15] tool to check a program against a runtime monitor using set interface operations ($\exists$, $\in$, $\notin$) on program statement: **`YR_DB_RUNTIME_VERIF`**. 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 MySQL events to **`YR_DB_RUNTIME_VERIF`**.

A current application of the type of analysis technique presented in this paper would be for instance testing the software for a control device of a simulator of a LINAC [1] (linear accelerator) in the radiotherapy.

Several runtime monitoring tools [19,3,16,5,8,6] have been presented in the past by researchers; they have the drawback to annotate and to instrument the System Under Test (SUT), thus augmenting the SUT runtime overhead, *or behavior*.

9 Acknowledgments

We would like to thank THOMAS NDIE DJOTIO, and JAN PELESKA for their support in continuing this research.

[1] A device used for external beam radiation treatment for cancer patients.

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. 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
6. 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
7. Booch, G., Rumbaugh, J., Jacobson, I.: Unified Modeling Language User Guide, The (2nd Edition) (Addison-Wesley Object Technology Series) (2005)
8. 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
9. 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
10. froglogic.com: Home • froglogic. http://www.froglogic.com/home (Dec 2022), accessed last time on Dec $18, 2022$ at $20{:}00$
11. Hallé, S.: Event Stream Processing with BeepBeep 3: Log Crunching and Analysis Made Easy (12 2018)
12. Harel, D.: Statecharts: a visual formalism for complex systems. Science of Computer Programming **8**(3) (1987)
13. 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
14. http://freedesktop.org/wiki/Software/cppunit: cppunit (Dec 2022), accessed last time on January $01, 2023$ at $12{:}00$
15. doc.qt.io/qt-5/qtdbus index.html: Qt D-Bus (Dec 2022), accessed last time on Dec $22, 2022$ at $12{:}40$
16. 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
17. 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)
18. Lam, P.: The Hob System for Verifying Software Design Properties (February 2007)
19. 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

20. MariaDB.org: MariaDB Foundation - MariaDB.org. http://www.mariadb.org (Jun 2022), accessed last time on June 24, 2022 at $12{:}20$
21. 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 $23{:}24$
22. Wikipedia.org: Mealy machine. http://en.wikipedia.org/wiki/Mealy_machine (Dec 2022), accessed last time on Dec 15, 2022 at $12{:}00$
23. Wikipedia.org: SQL - Wikipedia. http://en.wikipedia.org/wiki/SQL (Feb 2023), accessed last time on February 08, 2023 at $12{:}00$