User's Cheat Sheet for YRI_SD_RUNTIME_VERIF : A C++ Functional Library for Specifying "SDMM" (State Diagram Mealy Machine)

AUTHOR: Xavier Noundou [Pr. Prof. Dr.–Ing.]
Contact: YERITH.xavier@gmail.com

Contents

List of Figures

List of Tables

Table 1: STATE DIAGRAM MEALY MACHINE SPECIFICATION KEYWORDS IN YERITH_QVGE. **'AUTO'** KEYWORDS SPECIFIES ALSO SQL QUERY FOR GOING OUT AUTOMATICALLY FROM A FAIL (FORBIDDEN) STATE. ("SEE SECTION **??**.")

N°	scientific keywords	engineering keywords
1.	in_set_trace	in_sql_event_log
2.	not_in_set_trace	not_in_sql_event_log
3.	recovery_sql_query	recovery_sql_query
4.	STATE	STATE
5.	START_STATE	BEGIN_STATE
6.	FINAL_STATE ("**FINAL_STATE_AUTO**")	END_STATE ("**END_STATE_AUTO**") / ERROR_STATE ("**ERROR_STATE_AUTO**")
7.	IN_PRE	IN_BEFORE
8.	IN_POST	IN_AFTER
9.	**IN_POST_NOP**	**N / A**
10.	NOT_IN_PRE	NOT_IN_BEFORE
11.	NOT_IN_POST	NOT_IN_BEFORE
12.	**NOT_IN_POST_NOP**	**N / A**

Figure 1: A motivating example, as previous bug found in YERITH−ERP−9.0.
$\overline{Q0}$:= NOT_IN_BEFORE(YRI_ASSET, department.department_name).
$\overline{Q1}$:= IN_AFTER(YRI_ASSET, stocks.department_name).

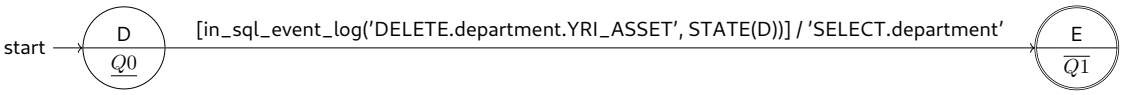


Figure 2: YERITH−ERP−9.0 administration section displaying departments ($\neg\overline{Q0}$).

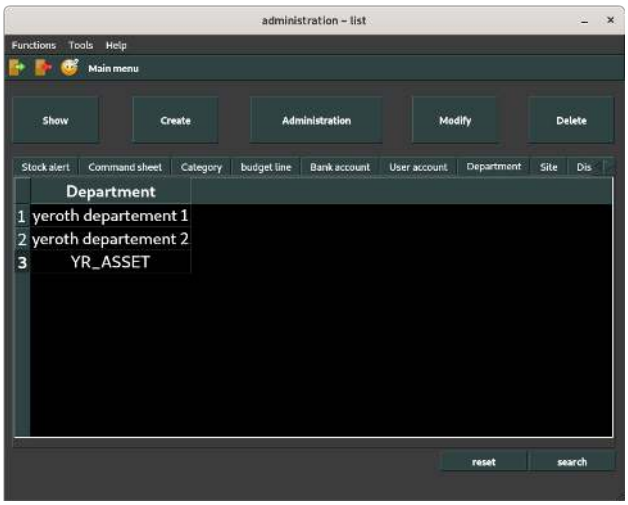
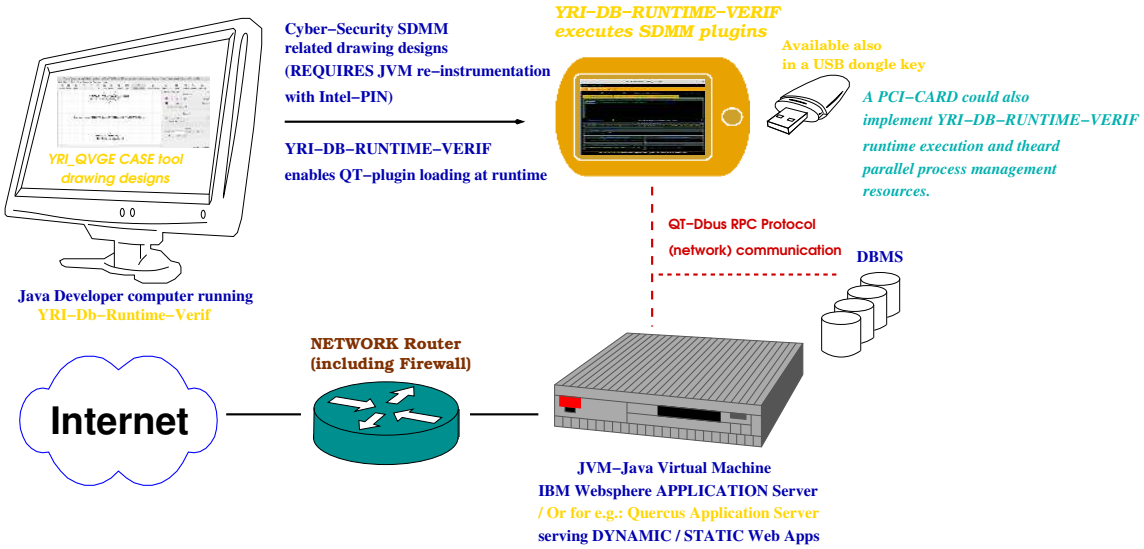


Figure 3: YERITH−ERP−9.0 stock asset window listing some assets ($\overline{Q1}$).

Figure 4: **A SAMPLE state diagram mealy machine file. KEYWORDS belonging both to 'engineering ("ERROR_STATE_AUTO")', and 'science (START_STATE)' can be intermingled in the same SDMM specification file.**

```
1. yr_sd_mealy_automaton_spec yr_missing_department_NO_DELETE
2. {
3.  START_STATE(d):NOT_IN_BEFORE(YRI_ASSET,department.department_name)
4.   ->[in_sql_event_log('DELETE.departement.YRI_ASSET',STATE(d))]/'SELECT.department'->
5.    ERROR_STATE(e):IN_AFTER(YRI_ASSET,stocks.department_name).
6. }
```

Figure 5: SAMPLE USE CASE SCENARIO OF "SDMM ".



Listing 1: ***Sample real world "C++" code as opposed to PSEUDO–CODE "C++" code; as modified by a developer after automatic generation of*** **YRI-DB-RUNTIME-VERIF.**

```
1  bool YERITH_QVGE_sample_PAPER_extended_version_PROPERY::DO_VERIFY_AND_or_CHECK_ltl_PROPERTY(
2      QString sql_table_ADDED_with_file_AND_line_number,
3      uint sql_record_qty_MODIFIED,
4      YRI_CPP_UTILS::SQL_CONSTANT_IDENTIFIER cur_SQL_command)
5  {
6      QStringList sql_table_ADDED_with_file_AND_line_number_LIST = sql_table_ADDED_with_file_AND_line_number.split(";", Qt::KeepEmptyParts);
7      QString sql_table_name = sql_table_ADDED_with_file_AND_line_number_LIST.at(0);
8      QString CPP_FILE_NAME = sql_table_ADDED_with_file_AND_line_number_LIST.at(1);
9      QString cpp_line_number = sql_table_ADDED_with_file_AND_line_number_LIST.at(2);
10
11     switch(cur_SQL_command)
12     {
13     case YRI_CPP_UTILS::INSERT:
14         break;
15
16     case YRI_CPP_UTILS::SELECT:
17         if (YRI_DB_RUNTIME_VERIF_Utils::isEqualsCaseInsensitive(sql_table_name, "departements_produits")) {
18             return YRI_SQL_SELECT_departements_produits();
19         }
20         break;
21
22     case YRI_CPP_UTILS::UPDATE:
23         break;
24
25     case YRI_CPP_UTILS::DELETE:
26         break;
27
28     default:
29         break;
30     }
31
32     return false;
33 }
```

Figure 6: A SCREENSHOT OF YERITH_QVGE.

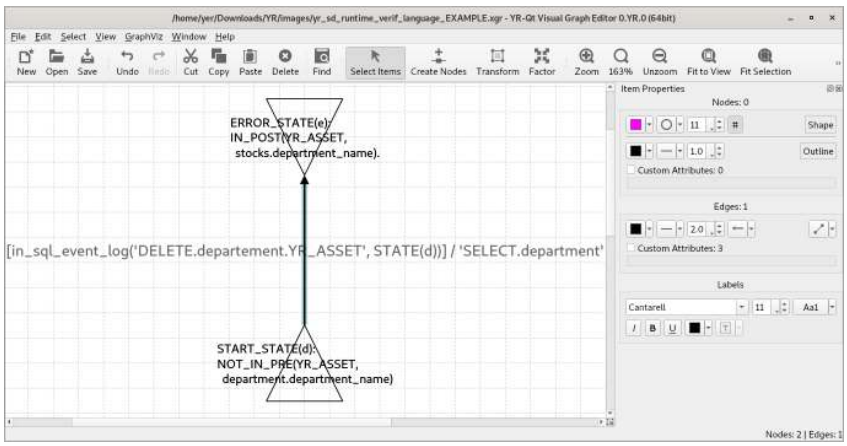
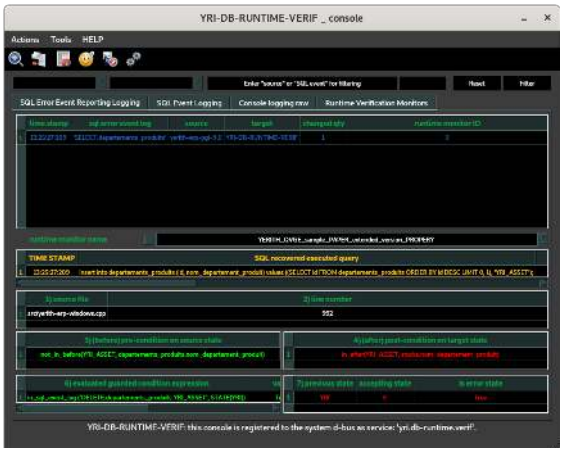


Figure 7: A SCREENSHOT OF YRI-DB-RUNTIME-VERIF SQL EVENT LOG.



1 Motivation for SDMM's Runtime Monitoring Verification Library "YRI_SD_RUNTIME_VERIF"

1.1 *A Sample Use–Case Scenario of "SDMM"*

1.2 *WHY DO I NEED FORMAL METHODS*

I HAVE interest in developing mathematical definitions and demonstrations for formal methods for following reasons :

1. any developer and / or BSC student can write code that compiles, run, and has JUnit classes testing code that passes.
2. BUT what demonstrates that the written and running program does what the primordial first intent of the developer and of his stakeholders was !?
3. THIS is exactly what formal methods (mathematics for computer science oriented towards testing and verifying; making sure, that a software program does what its intent is) does.

1.2.1 *"C++ library YRI_SD_RUNTIME_VERIF" : Expressing of sequencing of actions in time (temporal usage rules for system safety)*

Specifying temporal usage rules for system safety is primordial for ensuring a proper quality assurance of a system that is aimed to be more productive and safer in its usage by users, and / or maintainers.

THIS is why we created the C++ library YRI_SD_RUNTIME_VERIF. "**YRI_SD_RUNTIME_VERIF** " is an acronym for ***YERITH_State_Diagram_Runtime_Verification***.

1.3 Comparison with Unit Testing

1.3.1 Unit Testing

UNIT TESTING [Zel20]; [Nou09] (https://www.zenodo.org/record/8052444) consists in creating System Under Test program execution sequences with certain data & certain test criteria so to assert following statements over a program under test :

1. a specific input data in a unit test module method generates a pre-defined certain output data; *Thus developers must provide test input data for each test case runtime execution.*

UNIT TESTING is a kind of what is called :

- ***White-box testing [Zel20, MYE20]***; meaning a developer knows intrinsics (inside details) of the system-software under test;

  ⇒ Another kind of testing is called ***BLACK-box testing [Zel20, MYE20]***; meaning a developer doesn't need any inside source code details knowledge of the system-software under test.

Sample unit testing frameworks are for instance :

a) JUnit testing framework (https://www.junit.org) for JAVA programs;

b) Qt Test testing framework (https://doc.qt.io/qt-5) for Qt Graphical User Interface programs;

YRI-DB-RUNTIME-VERIF [NN25] (https://github.com/yerithd/yri-db-runtime-verif) is a simpler and more offering alternative; especially for software integrating testing (See Subsubsection 1.3.3) !

However, it is more appropriate for people with a strong mathematical background (set algebra & formal methods);

c) CppUnit testing framework (https://freedesktop.org/wiki/Software/cppuni) for C++ software.

1.3.2 Automated Unit Testing

Automated Unit Testing consists in automatically generating a System Under Test program execution sequences with certain data & certain test criteria so to assert following statements over a program under test :

1. a specific input data in a unit test module method generates a pre-defined certain output data; *Thus developers receive automatically test data & test cases for each test case runtime execution.*

Such generator systems like for example RT-Tester MBT (https://www.verified.de/products/model-based-testing) from "Verified System International GmbH" (https://www.verified.de), generally require a specification of the software and / or program under test; So to be able to perform the automatic code source generation for the Embedded Unit Tester "RT-Tester MBT (Model Based Testing)".

RT-Tester MBT, for example, inputs a "David Harel Statechart" (See Subsubsection 1.6.1) specification as for instance generated by a plug–in for "Borland Together 6", delivered as a by–product by "Verified System International GmbH".

A detailed algorithm and function description of RT-Tester MBT could be found in Subsection 1.6 !

1.3.3 Runtime Monitoring Verification

RUNTIME MONITORING Verification with* state diagram mealy machine *on the other side consists in :

1. specifying a wrong system under test program logical sequence using a mealy automaton as defined for instance in Subsection 1.5;
2. it is possible to specify conditions that trigger event between SDMM states : *guarded condition*; A guarded condition is placed between squared brackets ***('['guarded_condition']')***.

   Example :

   ```
   ->[in_sql_event_log('guardedevent',STATE(d))]/'event1'->
   ```

   The previous example guarded condition (***in_sql_event_log('guardedevent',STATE(d))***) transition means that : 'event1' occurs only if event 'guardedevent' has previously occurred and is an element all events that lead to mealy machine state STATE(d).

   The arrows signs ***('->')*** around depicts this is a transition.

3. YRI-DB-RUNTIME-VERIF observes monitors the SUT and emits all events as described and specified by developers; *Thus developers don't provide test cases and / or test data* !

Developers execute normally the System Under Test, by providing a normal system execution time run.

1.4 State Diagram Mealy Machine : Usages & Advantages

1.4.1 Usages

- Software code program testing & verifying requires means of expressing requiring in a form that can be used and executed by a computer program.

  "SDMM " is a textual (See Figure 4) and graphical (See Figure 1) representation of software system requirements; HOWEVER, requirements expressed using "SDMM " are fail (forbidden) requirements : I.E., software system execution run that shall never happen !

- During software development, developers need to characterize effects of program statements onto software program behavior.

  For instance, what causes the deletion of a database table column department value at the level of the software user–scale (Example in Figure 2). Figure 2 could also be represented by a *state diagram mealy machine* as illustrated in Figure 1.

1.4.2 Advantages

- ***WHITE-box*** as Well As ***Black-box testing*** is possible and / or available at once for any developer in testing; Depending on how YOU specify your runtime monitoring diagram & code (See Section 4.1).
- Software developers could implement code source checking of LTL–model checking formulas (https://www.wikipedia.org/) for their fail (forbidden) program requirements;

  I hope to also have sample ***LTL–model checking formulas*** implemented in future releases of YRI-DB-RUNTIME-VERIF, which is a runtime monitoring verification container provided to work with the ***"YRI_QVGE Design & Verification System"***.
- Software developers that create & maintain fail (forbidden) program requirements mustn't have a perfect knowledge of code sources of the System Under Test.

1.4.3 Cases of Practical Usages of "SDMM"

- Automobile industry "car–automobile–breaking safety & security usage rules" on–the–fly to avoid thousands of car–recall because of a bug (E.g.: Toyota–Camry–2009 (https://www.wikipedia.org));

  A car defect could only just be fixed at any Toyota registered auto-car-dealer by inputting a "SDMM" fix specification into an in–car placed YRI–QVGE–PC–Tablet device; Instead of sending it back to a Toyota car factory !

1.5 State Diagram Mealy Machine : Brief Summary Explanation

"YRI_SD_RUNTIME_VERIF" has following usages and advantages:

I.) ***Formalism STATE DIAGRAM mealy machine*** for system specification: Mealy Machines are kind of state machines where output states depend only on input on a source state.

STATE DIAGRAM mealy machine as defined by myself is a finite state automaton that only has following characteristics:

a.) Each automaton / machine only has 1 start state S_0, and 1 final state S_f that is an accepting / error state.

A final, accepting, or error state is a state that shows a defect status of the system under analysis (SUA / PUA / SUT).

b.) Each state S_i only has 1 outgoing state transition to an output state S_0.

c.) Each transition T, except a start transition "$start$", could have a **pre-condition, and post-condition**; That both are called or named state-edge-condition (**pre-condition on T_1: "$\underline{T_1}$"; post-condition on T_1: "$\overline{T_1}$"**).

A state S_i has a state–condition either as pre-condition "$\underline{T_1}$" on a state transition Or as a post–condition "$\overline{T_1}$" of a state transition T_1.

In Figure 1, state–conditions are for instance "$\underline{Q_0}$", and "$\overline{Q_1}$".

Each state–condition "S_i", and / or pre–condition; And / or post–condition is an algebra set specification that uses set inclusion operations : $\in, \notin$.

d.) A set inclusion operation as state–condition for a state S_i could for instance be :

1.) $\underline{\mathbf{Q0}}$:= IN_BEFORE(y, D) ***is to read*** "BEFORE next event, variable y is IN set D ($y \in D$)". THIS boolean first–order preposition is assigned into variable $\underline{\mathbf{Q0}}$; the "underlining" illustrates that this is a pre–condition : "***meaning a condition That shall hold before (PRE) next event to occur in software system.***"

2.) $\overline{\mathbf{Q0}}$:= IN_AFTER(y, D) ***is to read*** "AFTER next event, variable y is IN set D ($y \in D$)". THIS boolean first–order preposition is assigned into variable $\overline{\mathbf{Q0}}$; the "over-lining" illustrates that this is a post–condition : "***meaning a condition That shall hold after (POST) next event to occur in software system.***"

3.) IN_BEFORE_NOP() : THIS boolean first–order preposition means that this pre–condition (**'True'**) holds before (BEFORE) next event to occur in software system."

4.) IN_AFTER_NOP() : THIS boolean first–order preposition means that this post–condition (**'True'**) holds before next event to occur in software system, after (POST) previously occurred event."

5.) NOT_IN_BEFORE(y, D)

6.) NOT_IN_BEFORE_NOP()

7.) NOT_IN_AFTER(y, D)

8.) NOT_IN_AFTER_NOP()

II.) A C++ library that implements runtime monitoring and fail–state recovery as a static library : (https://www.github.com/yerithd/yri_sd_runtime_verif).

III.) A Free and OPEN SOURCE CODE SOFTWARE (Foss) implementation of a runtime monitor for using *state diagram mealy machine* specifications; BY means of a QT–dbus software communication stack with your own software : **"YRI-DB-RUNTIME-VERIF"** (https://www.github.com/yerithd/yri-db-runtime-verif).

YRI_SD_RUNTIME_VERIF's formal description of the state diagram formalism follows ***Mealy machine*** [Wik22] added with ***accepting states (final or erroneous states), and state diagram transition pre- and post-conditions*** : "*state diagram mealy machine*" ("SDMM").

Another excellent, detailed with proofs and theory presentation of mealy automata [PlH21] is available. In comparison to statechart [Har87], 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.*

A sample *state diagram mealy machine* is pictured in Figure 1. D is the start state,

1.6 Related State Diagram Formalisms

We here cite sample state diagram formalisms and the theory behind them for checking and / or verifying ***software design & programming properties***.

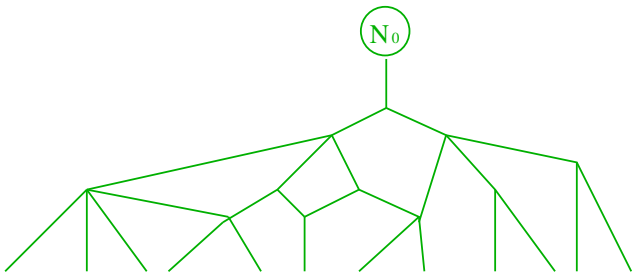
- Statechart by David HAREL : A statechart is here a visual formalism–description to enable a description of a complex software system that can also have states with substates; And / or timing conditions on states entries, and / or transition triggering conditions.

- "Kripke structure" with THEORY & formalism called "MODEL Checking"; mainly by DAVID–Emerson & CLARKE–Edmund [CGK+18] (https://mitpress.mit.edu/books/model-checking-second-edition)

  Software tools to perform model checking are called ***model checkers***. Sample model checkers are NuSMV (https://www.); SMV (https://www.); Spin-model checker (https://www.); etc..

1.6.1 David Harel Statechart : A Visual Formalism for State Diagram

Figure 9: A Sample David HAREL–Statechart.



Statechart [Har87], as defined and proposed by David HAREL; Represent diagrams, sometimes composite with internal states that are used to describe reactive systems, meaning systems that react and act based on environmental input and interaction.

Timed Discrete Input / Output Hybrid System (TDIOHS) as defined by Peleska et al. [BFPT06] implements David-HAREL statechart full compatibility incorporating the following elements for software system specification & elicitation :

1.) A software system behavior and actions can be represented as finite set of configurations; A configuration is an assignment of variables to values that situates a software system evolution over time;

2.) A set of configurations of a software system as a finite states graph representation where nodes represent software system states, And edges represent transition between software system states;

3.) Start– & End– (Final–, Accepting–, Erroneous–) state to represent INITIAL & FINAL software system state.

A reactive system has a particularity that it doesn't define a final-accepting state as it continuously reacts with its environment : it loops around its start state;

4.) Software system states transition defined as graph–edge between software system configuration–states;

5.) Software system state transition "Event" as label between graph–nodes;

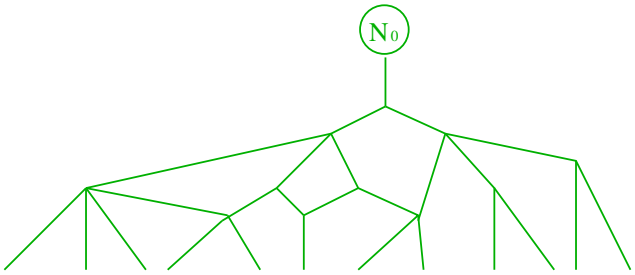
6.) Guarded condition as behavioral condition that must hold in order for a state transition event to trigger & thus send the software system into a new configuration;

7.) Timing condition

8.) hierarchical states (composite states, submachine state)

1.6.2 Timed Discrete Input / Output Hybrid System (TDIOHS)

Figure 10: A STCT–symbolic test case tree randomly generated by manual drawing for explanation purposes.



TDIOHS–Timed-Discrete-Input-Output-Hybrid-System [BFPT06] is a statechart compatible state diagram formalism defined by "Peleska et al." at the University of Bremen in Germany.

"Peleska et al." define a test mechanism to generate test cases from TDIOHS statechart visual description; The generated test cases abide to "Marie–Claude Gaudelle" algorithm defined in [Gau06] to statistically and uniformly-distribute test cases around a tree that describe all potential program code execution paths : STCT (Symbolic Test Case Tree).

A System Under Test (SUT) tree execution paths could then be traversed based on the real test data generated by a symbolic test data generator component as illustrated for instance in [nf10].

A sample STCT is illustrated in Figure 10; "N_0" is a start node of the program tree.

1.6.3 TDIOHS in Action within "Borland Together 6" with RT–Tester of 'verified.de'

Xavier NOUNDOU, myself, wrote together with Verified Systems International GmbH a plugin that enables developers to automatically create C++ testing code for design drawings created in the CASE-Computer Aided Software Engineering Design tool "Borland Together 6" (https://en.wikipedia.org/wiki/Micro_Focus_Together) .

I performed this task under supervision and advising of "Jan Peleska", as a student in Computer Science at the University of Bremen in Bremen–Germany in years "2004 / 2005".

Based on the automatically generated C++ module code, a developer could then use the UNIT Testing Framework for embedded systems RT-Tester of Verified.de (https://www.verified.de/products/model-based-testing) to create software module unit testing code with following criteria :

1.) ***MCDC (Modified Condition Decision Coverage) : test cases & data for checking any outcome of any boolean assignment of variables involved & used in a conditional IF–Then–Else branching statement;***

2.) ***Uniformly statistically distributed test cases [NN07] & test data [nf10] from all possible test runtime execution of the Program Under Analysis (PUA) [NN07, BFPT06]***;

3.) ***Myself I only received "2,000 Euros" as fund collected as money–rewards–intellectual property out of this work since "May 2007" when I delivered program code software as part–deliverable to acquire my "Diplom–INFORMATIKER" qualification from the department of Mathematics & Computer Science ("Fachbereich 3") of the "UNIversity–racist of Bremen–Germany".***

4.) etc.

Cite Tatiana Mangels, etc.

1.6.4 TDIOHS in Action by "Automatic Test Cases / Data Generation"

Master Thesis [NN07] in Computer Science of "Xavier N. NOUNDOU", at the University of Bremen in Bremen–Germany; demonstrates an implementation of uniform distributed statistically algorithm for selecting paths for creating test cases from a Statechart designed in "Borland–Together 6".

The statechart is first of all transformed into a Symbolic Test Case Tree (STCT) before "MARIE-Claude Gaudelle" [Gau06] modified algorithm by [NN07] is applied.

Cite paper of PELESKA, HELGE, & Tatiana.

More information and contacts for buying and / or trying this software product can be found at following URL : https://www.verified.de/products/model-based-testing, from the German society "Verified Systems International GmbH".

2 Mathematical Formal Definition of SDMM

THIS section gives a mathematical theoretical definition of state diagram mealy machine (abbreviated "SDMM"), as conceived originally by us for our project YERITH–ERP–9.0 [Nou22] runtime monitoring verification started in year 2015 in Yaounde Cameroon.

YRI_SD_RUNTIME_VERIF's formal description of the state diagram formalism follows ***Mealy machine*** [Wik22] 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 [PlH21] is available. In comparison to statechart [Har87], 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.*

2.1 Definition 1 : A state diagram (for mealy machine).

A state diagram is an 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.: $\overline{Q1}$). A pre-condition is comparable to a Harel-statechart *guarded condition*.
- Σ: an input alphabet, Σ:= $\{False, True\}$.
  $'False'$ means no input from SUT into YRI-DB-RUNTIME-VERIF.
  $'True'$ means any input could come from SUT.
- Λ: 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).
- δ : $S \times C$: 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 1 we have:

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

2.2 Definition 2 : A pre-condition.

A pre-condition of a state diagram transition is a predicate hat 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".

2.3 Definition 3 : A post-condition.

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.

IT is also for user interest to have following NO–OPeration post-conditions when working with state diagram mealy machines with more than 2 states :

- **IN_POST_NOP**
- **NOT_IN_POST_NOP**

OUR experience, not reported at time anywhere, shows that *state diagram mealy machine* with more than 2 states are really more for parallel system modeling; I.E. systems that work in GUI with timers and several threads of work at anytime.

"In such cases, subsequent linearly placed states may not belong to same thread of execution: this is kind of what is called in CSP (Communicating Sequential Processes) []; Parallel Composition."

Listing 2: 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).

```
1  bool MONITOR::YRI_trigger_an_edge_event(QString an_edge_event)
2  {
3      MONITOR_EDGE cur_OUTGOING_EDGE = _cur_STATE.outgoing_edge();
4
5      if (cur_OUTGOING_EDGE.evaluate_GUARDED_CONDITION_expression() &&
6          (an_edge_event == cur_OUTGOING_EDGE.edge_event_token()))
7      {
8          bool precondition_IS_TRUE = cur_OUTGOING_EDGE
9                          .CHECK_SOURCE_STATE_PRE_CONDITION(_cur_STATE);
10
11         if (precondition_IS_TRUE)
12         {
13             set_current_triggered_EDGE(cur_OUTGOING_EDGE);
14
15             MONITOR_STATE a_potential_accepting_state =
16                 cur_OUTGOING_EDGE.get_TARGET_STATE();
17
18             if (CHECK_whether__STATE__is__Final(a_potential_accepting_state))
19             {
20                 CALL_BACK_final_state_FUNCTION(a_potential_accepting_state);
21             }
22             return true;
23         }
24     }
25     return false;
26 }
```

2.4 Definition 4 : A trace.

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 1 we have: $trace(E) = trace(D), <$ 'SELECT.department' $>$.

2.5 *SUT Event Processing Method YRI_trigger_an_edge_event*

Listing 2 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 shall have only 1 outgoing edge (transition), by specification and construction, as explained in Proposition 2.5.1 in Section 2.

The algorithm in Listing 2 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.

2.5.1 *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 1. These 2 properties, together with algorithm 'YRI_trigger_an_edge_event(QString an_edge_event)' (Listing 2) of YRI_SD_RUNTIME_VERIF, ensures that there are no false warnings during YRI-DB-RUNTIME-VERIF analyses.

For example, the opponents runtime monitoring and / or verification tools–systems [BH12, BRBY00, AAC+05, Bod05, CR07] may give false warnings.

We need to also report that if a developer doesn't well specify to the runtime verifier tester where to emit event, as for instance in "Listing 1", false warnings (false positives) may occur.

2.5.2 *Explanation on HOW to avoid code that creates False Warnings (False Positives)*

2.6 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 1 (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 $\in \Sigma$) and 'D' a state diagram state ($D \in \mathbf{S}$).

2.7 SDMM for modeling parallel-concurrent software system

2.8 SDMM in Action within YRI_QVGE by 'Yerith R&D'

A screenshot of a fail (forbidden) *state diagram mealy machine* is illustrated in Figure 6.

This fail fail (forbidden) *state diagram mealy machine* is the same described in Figure 4 with

"YRI_QVGE Design & Verification System" domain specific language (DSL) "YRI_SD_RUNTIME_VERIF_LANG".

The program–DSL code source illustrated in Figure 4 was automatically generated by myself by ***"Saving a design as a DOT/GraphVIZ"*** document.

"Saving a design drawing as a DOT/GraphVIZ" creates source code in domain specific language ***"YRI_SD_RUNTIME_VERIF_LANG"*** within a file ending with ***'.sd_mealy'***.

2.9 SDMM in Action by automatic "Runtime Monitors Automatic Generation"

I created YRI-DB-RUNTIME-VERIF so to allow automatic generation of runtime execution time monitoring modules for *state diagram mealy machine* defined using a Domain Specific Language (DSL) called YRI_SD_RUNTIME_VERIF_LANG.

YRI_SD_RUNTIME_VERIF enables expression and description of state diagram mealy machine specification.

3 HOW TO Setup C++ Library "YRI_SD_RUNTIME_VERIF" for Usage in A C++ PROGRAM SOURCE CODE

3.1 Development Toolchain

4 METHODS of C++ Library "YRI_SD_RUNTIME_VERIF"

Table 2: Sample important Classes (prefix YRI_CPP_ to class name) & Methods of C++ Library "YRI_SD_RUNTIME_VERIF".

N°	CLASSES	METHODS	UTILITY
1.	MONITOR	CREATE_MONITOR	
2.	MONITOR	YRI_register_set_final_state_CALLBACK_FUNCTION	
3.	MONITOR	RESET_RUNTIME_MONITOR	
4.	MONITOR	YRI_trigger_an_edge_event	
5.	MONITOR	create_yri_monitor_state	
6.	MONITOR	create_yri_monitor_edge	
7.	MONITOR	find_yri_monitor_state	
8.	MONITOR	set_RUNTIME_MONITOR_NAME	
9.	MONITOR	IS_in_TRACE_LOG	
10.	MONITOR	TRACE_LOG_current_RECEIVED_EVENT_TOKEN	
11.	MONITOR	GET_root_edge	
12.	MONITOR	DELETE_yri_monitor_edge	
13.	notinset_inset_TRACE_expression	YRI_CPP_notinset_inset_TRACE_expression	
14.	notinset_inset_TRACE_expression	set__USE_SQL_SYNTAX_event_logging__FOR_PRINTING	

We place and explain sample methods of C++ Library "YRI_SD_RUNTIME_VERIF" that WE deem of major interest for developers :

1.)

4.1 Generated Method YOU need to code

Table 3: YERITH_QVGE Toolchain

PROJECT	Required Program / Library
1) YRI_SD_RUNTIME_VERIF_LANG	
2) YRI_SD_RUNTIME_VERIF_LANG_COMP	1)
3) YRI_SD_RUNTIME_VERIF_UNIT_TESTS	1)
4) YRI-DB-RUNTIME-VERIF	2)

Table 3 illustrates for each library project, which others it depends on.

5 Detailed Scientific and Engineering Presentation Document on 'zenodo.org'

Detailed formal scientific and engineering contributions of design and testing system YERITH_QVGE can be found in ***JOURNAL ARTICLE "Runtime Verification Of SQL Correctness Properties with YRI-DB-RUNTIME-VERIF"*** [nN23].

6 Conclusion

The graphical drawing tool YERITH_QVGE (Figure 6) costs only 2,500 EUROS. WE ONLY SUPPORT DEBIAN-LINUX (https://www.debian.org).

References

[AAC+05] Chris Allan, Pavel Avgustinov, Aske Simon Christensen, Bruno Dufour, Christopher Goard, Laurie J. Hendren, Sascha Kuzins, Jennifer Lhoták, Ondrej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, Julian Tibble, and Clark Verbrugge. abc the aspectbench compiler for aspectj a workbench for aspect-oriented programming language and compilers research. In Ralph E. Johnson and Richard P. Gabriel, editors, *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*, pages 88–89. ACM, 2005.

[BFPT06] B. Badban, M. Fränzle, J. Peleska, and T. Teige. Test automation for hybrid systems. In *Third International Workshop on Software Quality Assurance (SOQUA 2006)*, pages 14–21, 2006.

[BH12] Eric Bodden and Laurie Hendren. 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.

[Bod05] Eric Bodden. J-LO - A tool for runtime-checking temporal assertions. Diploma thesis, RWTH Aachen University, November 2005.

[BRBY00] Sergey Butkevich, Marco Renedo, Gerald Baumgartner, and Michal Young. Compiler and tool support for debugging object protocols. In *SIGSOFT '00/FSE-8*, 2000.

[CGK+18] Edmund M. Clarke, Orna Grumberg, Daniel Kroening, Doron A. Peled, and Helmut Veith. *Model checking, 2nd Edition*, 2018.

[CR07] Feng Chen and Grigore Rosu. Mop: an efficient and generic runtime verification framework. In Richard P. Gabriel, David F. Bacon, Cristina Videira Lopes, and Guy L. Steele Jr., editors, *Proceedings of the 22nd Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 569–588. ACM, 2007.

[Gau06] M.-C. Gaudel. ???–test automation for hybrid systems–??? In *Third International Workshop on Software Quality Assurance (SOQUA 2006)*, pages 14–21, 2006.

[Har87] David Harel. Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, 8(3), 1987.

[MYE20] Andrew MYERS. Software testing ..., 20..

[nf10] Serges ACHILLES nono fopoussi. Automatisierte testdatengenerierung hybrider diskret-kontinuierlicher eingebetteter systeme. https://www.deutsche-digitale-bibliothek.de/person/gnd/141875240, 2010. DOCTORATE THESIS IN COMPUTER SCIENCE (Dr.–Ing.), University of Bremen, Bremen, Bremen, Germany.

[NN07] Xavier Noumbissi Noundou. Statistical test cases generation for reactive systems. https://www.informatik.uni-bremen.de/agbs/qualifikationsarbeiten/diplomarbeiten_e.html, 2007. Integrated Bachelor & Master's Degree Thesis in Computer Science (B.Sc. & M.Sc.), University of Bremen, Bremen, Bremen, Germany.

[nN23] Xavier noumbissi Noundou. A Framework for Verifying SQL Correctness Temporal Properties of GUI Software at Runtime. https://zenodo.org/records/10976659, October 2023.

[NN25] Xavier Noumbissi Noundou. A C++ functional library for specifying "SDMM" (state diagram mealy machine). https://www.zenodo.org/records/10474033, 2025. A C++ Functional Library for Specifying "SDMM".

[Nou09] Xavier Noundou. Junit 4 tutorial. https://www.zenodo.org/record/8052444, Oct. 2009. Text–tutorial "Junit 4", University of Waterloo, Waterloo, Ontario, Canada.

[Nou22] Xavier Noundou. YERITH–ERP–PGI–3.0 Doctoral Compendium. https://archive.org/download/yerith-erp-pgi-compendium_202206/JH_NISSI_ERP_PGI_COMPENDIUM.pdf, June 2022. Accessed last time on January 21, 2023 at 23:24.

[PlH21] Jan Peleska and Wen ling Huang. Test automation; foundations and applications of model-based testing. https://www.informatik.uni-bremen.de/agbs/jp/papers/test-automation-huang-peleska.pdf, July 2021. Accessed last time on May 06, 2023 at 12:00.

[Wik22] Wikipedia.org. Mealy machine. https://en.wikipedia.org/wiki/Mealy_machine, December 2022. Accessed last time on Dec 15, 2022 at 12:00.

[Zel20] Andreas Zeller. Software testing ..., 20..

Index

User's Guide for the Design and Testing System YERITH_QVGE (YRI_QVGE)

AUTHOR: Xavier Noumbissi Noundou [Pr. Prof. Dr.–Ing.]
Contact: YERITH.XAVIER@gmail.com

Contents

List of Figures

List of Tables

Table 1: STATE DIAGRAM MEALY MACHINE SPECIFICATION KEYWORDS IN YERITH_QVGE. **'AUTO'** KEYWORDS SPECIFIES ALSO SQL QUERY FOR GOING OUT AUTOMATICALLY FROM A FAIL (FORBIDDEN) STATE. ("SEE SECTION 9.")

N°	scientific keywords	engineering keywords
1.	in_set_trace	in_sql_event_log
2.	not_in_set_trace	not_in_sql_event_log
3.	recovery_sql_query	recovery_sql_query
4.	STATE	STATE
5.	START_STATE	BEGIN_STATE
6.	FINAL_STATE ("**FINAL_STATE_AUTO**")	END_STATE ("**END_STATE_AUTO**") / ERROR_STATE ("**ERROR_STATE_AUTO**")
7.	IN_PRE	IN_BEFORE
8.	IN_POST	IN_AFTER
9.	**IN_POST_NOP**	**N / A**
10.	NOT_IN_PRE	NOT_IN_BEFORE
11.	NOT_IN_POST	NOT_IN_BEFORE
12.	**NOT_IN_POST_NOP**	**N / A**

Figure 1: A motivating example, as previous bug found in YERITH−ERP−9.0.
$\underline{Q0}$:= NOT_IN_BEFORE(YRI_ASSET, department.department_name).
$\underline{\overline{Q1}}$:= IN_AFTER(YRI_ASSET, stocks.department_name).

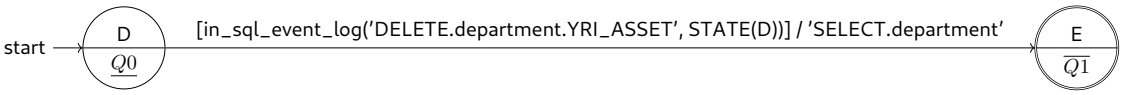


Figure 2: YERITH−ERP−9.0 administration section displaying departments ($\neg\underline{Q0}$).

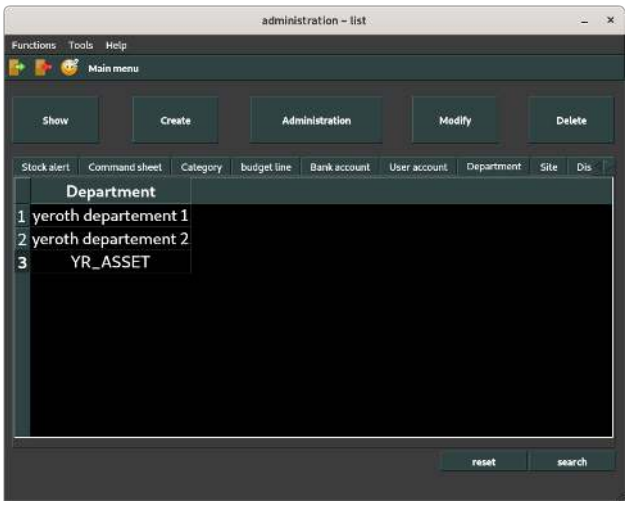


Figure 3: YERITH−ERP−9.0 stock asset window listing some assets ($\underline{\overline{Q1}}$).

Figure 4: **A SAMPLE state diagram mealy machine file. KEYWORDS belonging both to 'engineering ("ERROR_STATE_AUTO")', and 'science (START_STATE)' can be intermingled in the same SDMM specification file.**

```
1. yr_sd_mealy_automaton_spec yr_missing_department_NO_DELETE
2. {
3.  START_STATE(d):NOT_IN_BEFORE(YRI_ASSET,department.department_name)
4.   ->[in_sql_event_log('DELETE.departement.YRI_ASSET',STATE(d))]/'SELECT.department'->
5.    ERROR_STATE(e):IN_AFTER(YRI_ASSET,stocks.department_name).
6. }
```

Figure 5: A SCREENSHOT OF YERITH_QVGE.

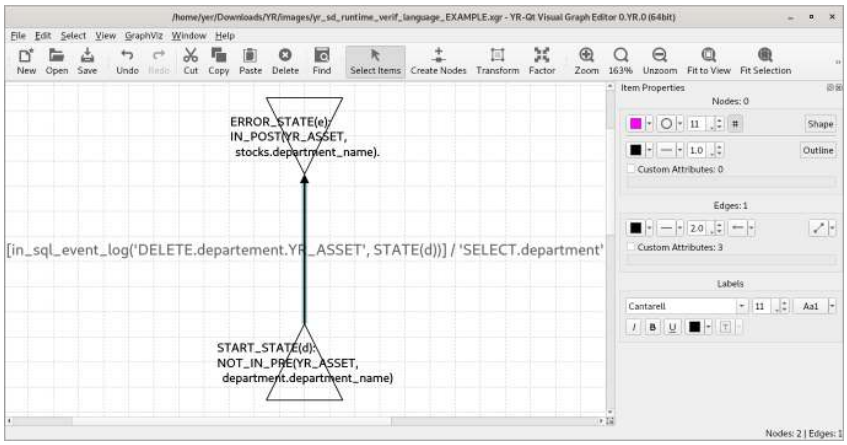


Figure 6: A SCREENSHOT OF YRI-DB-RUNTIME-VERIF SQL EVENT LOG.

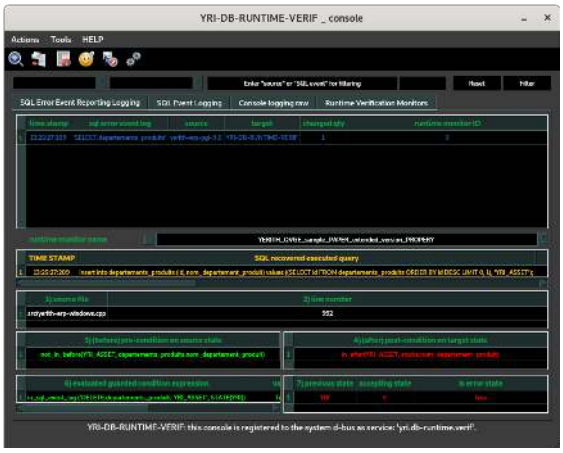
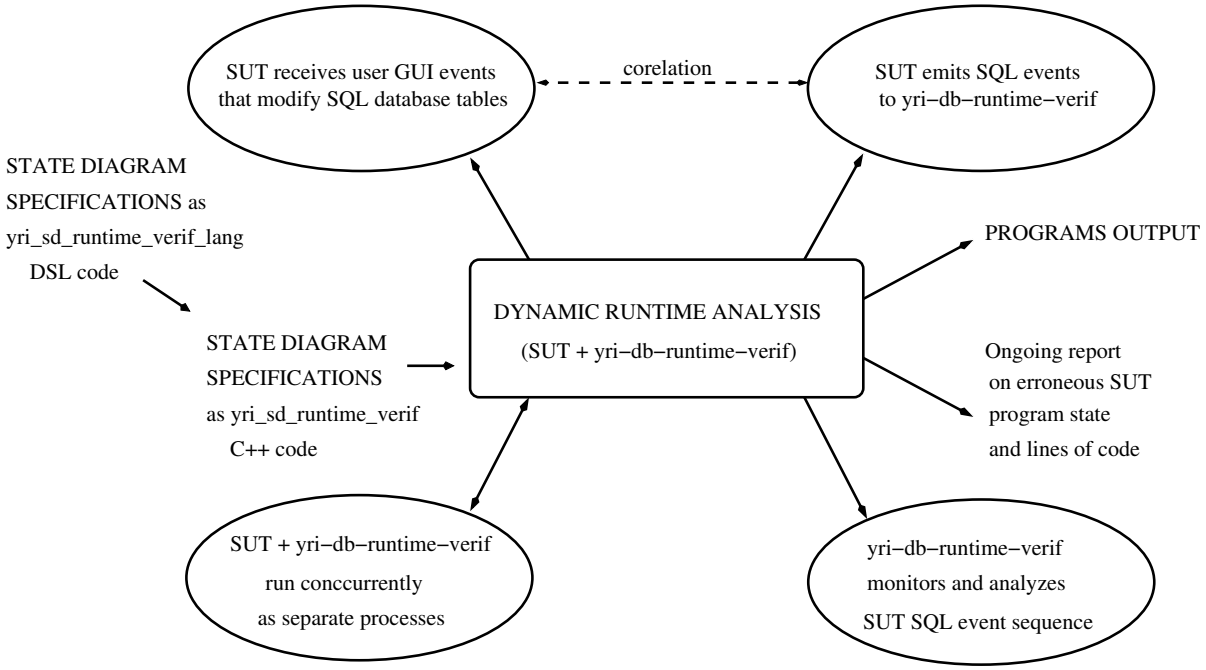
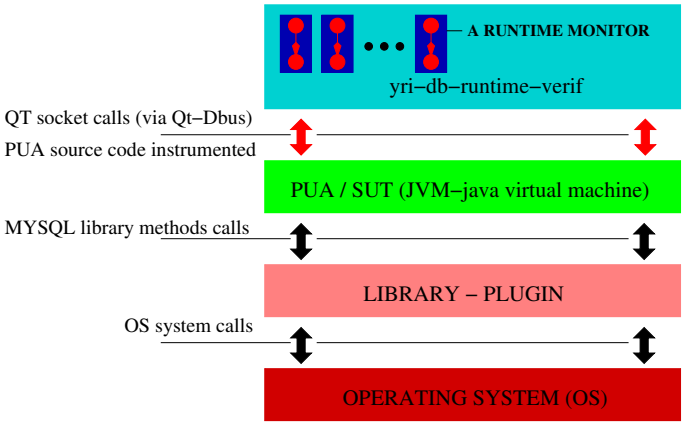


Figure 7: YRI-DB-RUNTIME-VERIF operation WORKFLOW (inspired from "Jon Eyolfson's RV'13 paper on TRACERORY" (https://hdl.handle.net/10012/6206)).



1 Introduction

Figure 8: SOFTWARE ARCHITECTURE OF YRI-DB-RUNTIME-VERIF.



This user's guide helps briefly and concisely how to create a binary executable of the runtime monitoring testing tool **YRI-DB-RUNTIME-VERIF** having user defined runtime monitors. The guide also specifies keywords allowed within runtime monitor specifications as State Diagram Mealy Machines.

YERITH_QVGE (YRI_QVGE) could be used for the following automatic generation, analysis, verification, and validation tasks:

1. Automatic generation of runtime monitoring module program to prove whether a test procedure, automated, or not, is correct with regards to a test and / or design STATE DIAGRAM MEALY MACHINE (formally described in [Nou23]).

   In effect, let the test execution be runtime monitored to watch whether accepting error states would be found.

   For instance, Junit testing environment could automatically integrate an automatically generated runtime monitor infrastructure for unit testing.

2. Automatic generation of runtime monitoring module program for any software that can emit DBus messages.

   "*Such runtime monitoring modules are for interest for special LTL model checking properties that cannot get a definite answer through use of a conventional model checker*".

3. Software design properties with SQL

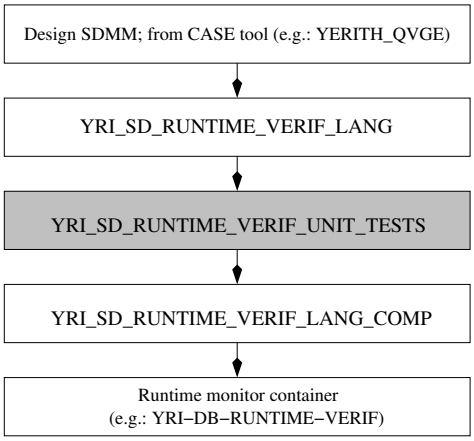
4. Software design properties including event sequences over different layers of software system architecture

5. Class diagram with sequence diagram.

[1] https://www.github.com/yerithd/yri_sd_runtime_verif
[2] https://www.github.com/yerithd/yri-db-runtime-verif
[3] Scientific: fail (forbidden) trace.
[4] Structure Query Language.

2 YERITH_QVGE (YRI_QVGE) Short Overview

Figure 9: YERITH_QVGE software library dependencies.



YERITH_QVGE is a CASE (Computer-Aided Software Engineering) design tool to generate "domain-specific language (DSL) YRI_SD_RUNTIME_VERIF_LANG [1]" files, to be inputted into the "compiler YRI_SD_RUNTIME_VERIF_LANG_COMP", so to generate C++ files for the "runtime verifier tester YRI-DB-RUNTIME-VERIF [2]" that allows for manual verification of SQL correctness properties of Graphical User Interface (GUI) software.

Figure 10 illustrates a workflow diagrammatically of the afore described process.

Figure 9 show a diagram of the afore described process; The step of the unit tests is colored in gray because it is only for developers of YERITH_QVGE intended.

YRI-DB-RUNTIME-VERIF inputs SQL correctness properties expressed using the formalism "*state diagram mealy machine* (YRI_SD_RUNTIME_VERIF_LANG)". Figure 8 illustrates a software system architecture of YRI-DB-RUNTIME-VERIF, together with the monitored program under analysis. The Free Open Source Code Software (FOSS) tool-chain of development testing is located as follows for free, EXCEPT for "YERITH_QVGE " that is a Closed Source Code Software (CSCS):

- COMPILER (i.e.: YRI_SD_RUNTIME_VERIF_LANG_COMP): https://www.github.com/yerithd/yri_sd_runtime_verif_lang
- RUNTIME VERIFIER TESTER (i.e.: YRI-DB-RUNTIME-VERIF): https://www.github.com/yerithd/yri-db-runtime-verif
- state diagram mealy machine UNIT TESTS CODE (i.e.: YRI_SD_RUNTIME_VERIF_UNIT_TESTS): https://www.github.com/yerithd/yri_sd_runtime_verif_UNIT_TESTS
- state diagram mealy machine (i.e.: YRI_SD_RUNTIME_VERIF_LANG): https://www.github.com/yerithd/yri_sd_runtime_verif

3 YERITH_QVGE (YRI_QVGE) Project Dependency

Table 2: YERITH_QVGE Design and Testing System Dependencies

PROJECT	Required Program / Library
1) YRI_SD_RUNTIME_VERIF_LANG	
2) YRI_SD_RUNTIME_VERIF_LANG_COMP	1)
3) YRI_SD_RUNTIME_VERIF_UNIT_TESTS	1)
4) YRI-DB-RUNTIME-VERIF	2)

Table 2 illustrates for each library project, which others it depends on.

4 Advantages of YERITH_QVGE

A sample state diagram mealy machine is shown in Figure 4.

WITH manual drawing of SQL CORRECTNESS PROPERTY MODEL, you are freed from manually writing "state diagram mealy machine text files" that could be tedious and lengthy. Also, editing state diagram mealy machine files manually could be more error-prone than letting a compiler (YRI_SD_RUNTIME_VERIF_LANG) do it for you.

5 State Diagram Mealy Machine (SDMM)

TABLE 1 depicts scientific keywords and their engineering counterpart that can be used in describing NOT DESIRABLE [3] SQL [4] call sequence state diagram mealy machine in YERITH_QVGE Design and Testing System.

A STATE DIAGRAM mealy machine specification is compiled into C++ code that describes a runtime monitor to be executed in the runtime monitoring tester YRI-DB-RUNTIME-VERIF. Figure 4 depicts a sample State Diagram Mealy Machine specification on a NOT DESIRABLE SQL call sequence.

5.1 HOW TO READ A "SDMM"

Figure 1 shows a finite automaton representation of the mealy machine description in Figure 4. It shall be read as follows:

- The program is in a start state D; state D is a start state since there is incoming "START" arrow into it.
- (**Pre-**) Condition $\overline{Q0}$: "department name `'YRI_ASSET'` is not in table column `'department_name'` of database table `'department'`"; applies in state D.
- Whenever GUARD CONDITION : ***in_sql_event_log('DELETE.department.YRI_ASSET', STATE(d))***: "event `'DELETE.department.YRI_ASSET'` appears in SQL event log (trace) leading to state D"; applies in state D, system under test (SUT) event `'SELECT.department'` could occur.
- When SUT event `'SELECT.department'` occurs, SUT is now in state E; state E is an error state because the node that represents it in Figure 1 has 2 circles on it.
- (**Post**-) Condition $\overline{Q1}$: "department name `'YRI_ASSET'` is in table column `'department_name'` of database table `'stocks'`"; applies in state E.

  This shall not be the case since department `'YRI_ASSET'` is no more defined in SUT database table `'department'`.

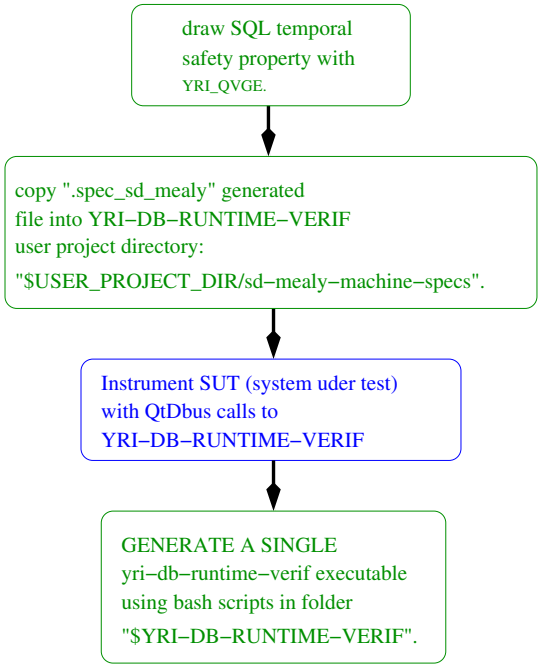
5.2 "SDMM" WITH MORE THAN 2 STATES

State Diagram Mealy Machines (SDMM) with more than 2 states have following characteristics, as detailed in scientific and engineering journal paper [Nou23] in preparation:

- Only the first transition has a pre-condition specification
- Each other transition only has a post-condition specification
- Since each state only has 1 outgoing state transition, the post-condition of the previous (incoming) state transition acts as the pre-condition of the next transition.

6 YERITH_QVGE (YRI_QVGE) Workflow

Figure 10: Workflow explanation.



The **"Design and Testing System"** YERITH_QVGE works with following workflow, as illustrated graphically in Figure 10, and in Figure 7:

1. Draw Structure Query Language (SQL) temporal safety property using drawing tool YERITH_QVGE;
2. copy the generated ".spec_sd_mealy" files into a user project directory in YRI-DB-RUNTIME-VERIF home development folder: "$YRI–DB–RUNTIME–VERIF";
3. follow the steps described in Section 7 so to gather a single executable that defines all specified runtime monitors.

7 Custom User Project (YRI–DB–RUNTIME–VERIF)

Table 3: YRI-DB-RUNTIME-VERIF Directories

Variable for illustration purposes	Meaning
$YRI–DB–RUNTIME–VERIF	root directory of YRI-DB-RUNTIME-VERIF
$YRI–DB–RUNTIME–VERIF/$USER_PROJECT	root directory of user project

Table 3 illustrates directories that will be used to describe a process to generate a single binary executable for a user's custom project with several runtime monitor specifications.

Figure 6 illustrates a screenshot of the Graphical User Interface (GUI) of YRI-DB-RUNTIME-VERIF. You can get a copy of YRI-DB-RUNTIME-VERIF using the following command:

git clone https://www.github.com/yerithd/yri-db-runtime-verif

Creating a binary executable for State Diagram Mealy Machine (SDMM) specifications consists of the following elements:

1. 'MariaDB' database connection configuration file: this file defines settings to connect to the system under test (SUT) application database; it is located in path: "$YRI-DB-RUNTIME-VERIF/YRI-DB-RUNTIME-VERIF-GUI-ELEMENTS-SETUP/yri-db-runtime-verif-database-connection.properties".

   A database connection to the SUT application database is required in order to check LTL property through the SDMM application library **YRI_SD_RUNTIME_VERIF_LANG.**

2. Property configuration file: this file defines environment variables necessary for building a binary executable for the user; it is located in path: "$YRI-DB-RUNTIME-VERIF/$USER_PROJECT/bin/configuration-properties.sh".

3. "$YRI-DB-RUNTIME-VERIF/$USER_PROJECT/sd-mealy-machine-specs": this directory contains user defined State Diagram Mealy Machine (SDMM) specifications to generate Corresponding runtime monitors within a single binary executable.

4. Generate an executable for a user defined runtime monitor:
   a) execute following command in directory "$YRI-DB-RUNTIME-VERIF":

   ```
   . ./YRI-create-executable-for-user-SDMM.sh -d $USER_PROJECT
   ```

   b) modify the LTL verification code part within the generated source code files.

   Then execute following command in directory "$YRI-DB-RUNTIME-VERIF":

   ```
   ./yr_db_runtime_verif_BUILD_DEBIAN_PACKAGE.sh
   ```

   c) uninstall YRI-DB-RUNTIME-VERIF with following command in directory "$YRI-DB-RUNTIME-VERIF":

   ```
   ./yr_DB_RUNTIME_VERIF_uninstall.sh
   ```

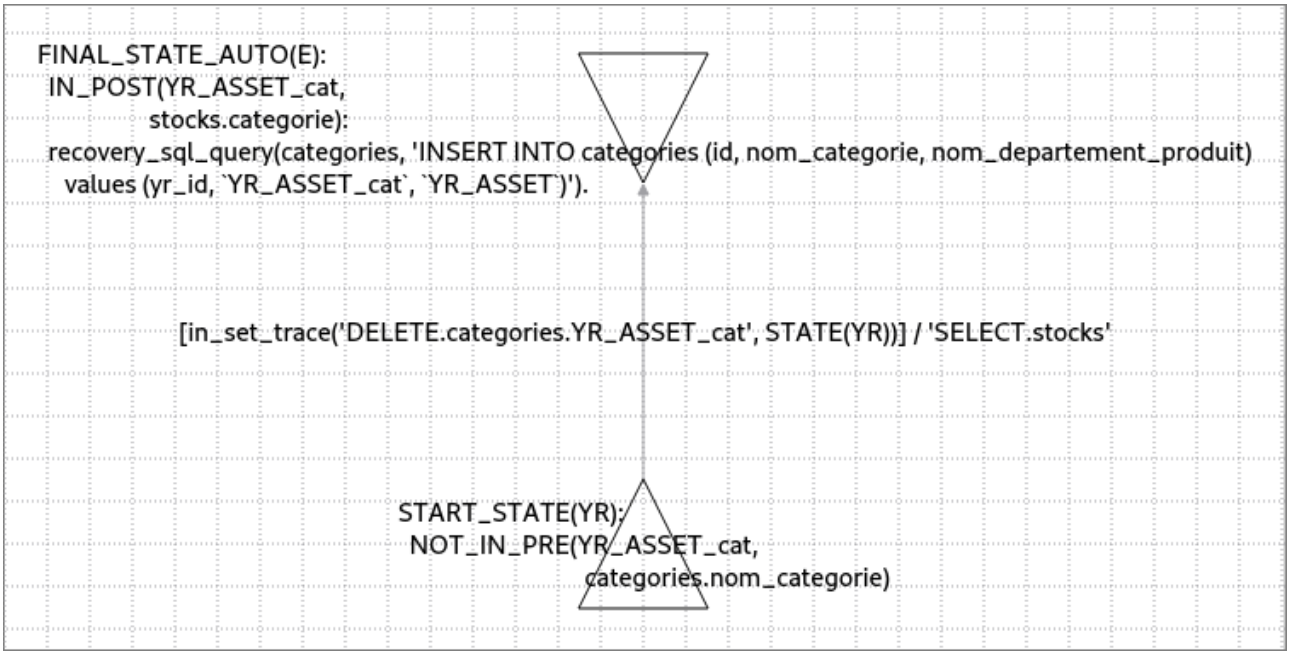
   d) re-install YRI-DB-RUNTIME-VERIF with following command in directory "$YRI-DB-RUNTIME-VERIF":

   ```
   ./yr_DB_RUNTIME_VERIF_INSTALL.SH
   ```

8 HOW TO START YRI-DB-RUNTIME-VERIF

- The "ELF-x64" binary executable, in the source development directory is located in full path: "$YRI-DB-RUNTIME-VERIF/bin".
- The DEBIAN-LINUX icon ( ) of YRI-DB-RUNTIME-VERIF is located in "Applications" menu under section "Programming", and section "Accessories".
- The "ELF-x64" binary executable, after installation of the DEBIAN-LINUX package 'yri-db-runtime-verif.deb' is located in full path: "/opt/yri-db-runtime-verif/bin".

Figure 11: **SAMPLE sql recovery state diagram model in YERITH_QVGE**



9 SQL QUERY Recovery execution on demand

A user can specify which SQL command query to execute whenever a System Under Test (SUT) lands in an accepting error state. This is done using keywords ending with "AUTO", used for meaning "AUTO RECOVERY FROM FAIL STATE":

1. `recovery_sql_query`
2. `END_STATE_AUTO`
3. `FINAL_STATE_AUTO`
4. `ERROR_STATE_AUTO`.

The use of an "AUTO" keyword shall be accompanied with a use of keyword `recovery_sql_query`, that specifies a SQL command query to run when landing in this fail error accepting state.

9.1 Automatic SQL Command Query Generation

YERITH_QVGE implements an automatic SQL query generation strategy in case a user don't specify a SQL command query, since it could be leaved empty: Subsections 9.1.1, 9.1.2, 9.1.3, and 9.1.4 describe the strategy implemented.

9.1.1 ERROR ACCEPTING STATE for sdmm 1.

$$\frac{\texttt{not in\_before}\,(\texttt{YX},\,\texttt{YY}) \qquad \texttt{ACTION}\,(\texttt{V})}{\texttt{in\_after}\,(\texttt{DD},\,\texttt{YR})}$$

9.1.2 RECOVERY 1.

$$\frac{\texttt{in\_after}\,(\texttt{DD},\,\texttt{YR}) \qquad \texttt{ACTION}\,(\texttt{RECOVERY\_ND})}{\texttt{not in\_after}\,(\texttt{DD},\,\texttt{YR})}$$

9.1.3 RECOVERY 2 (Practical solution to be implemented in YRI-DB-RUNTIME-VERIF.

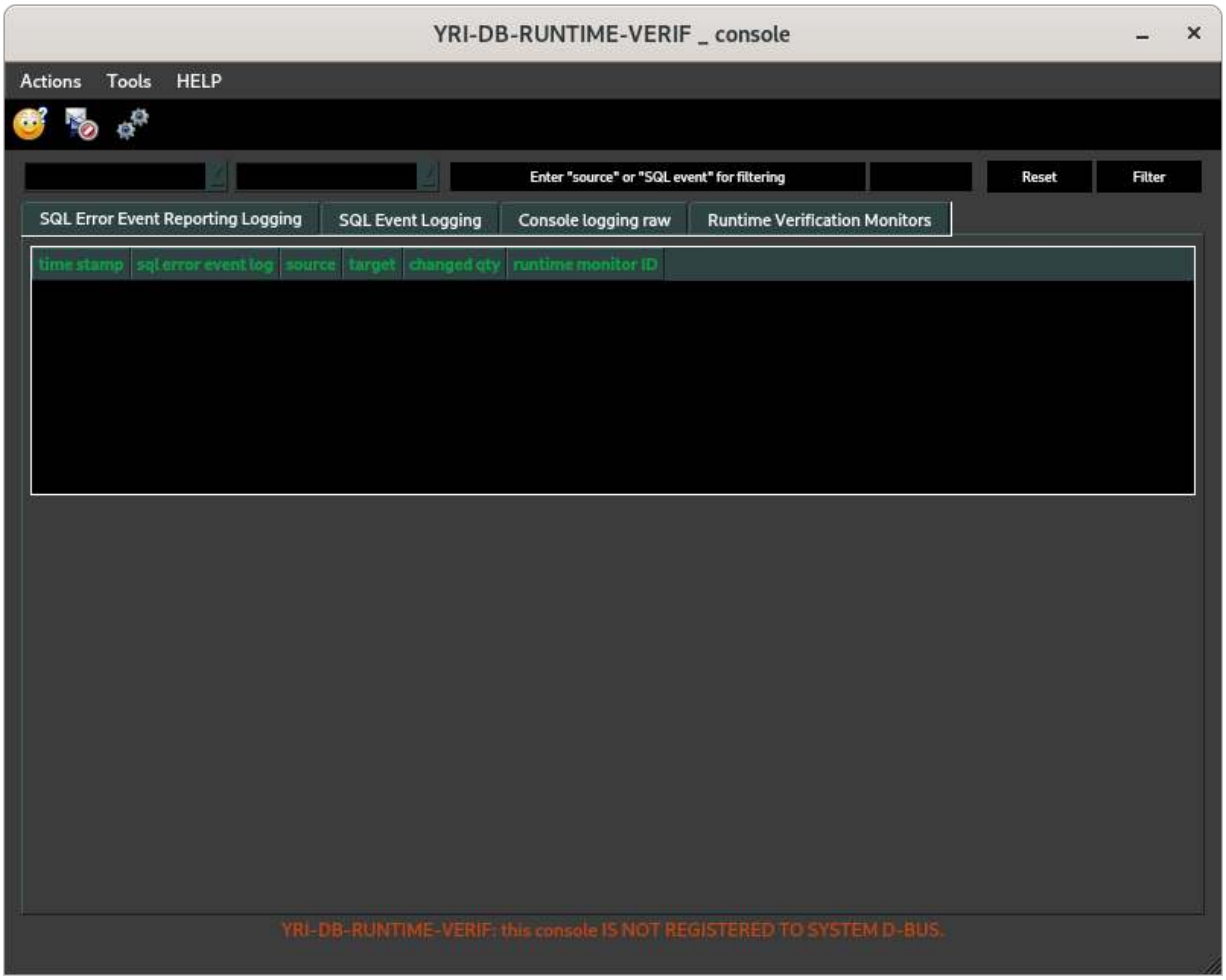
$$\frac{\texttt{in\_after}\,(\texttt{DD},\,\texttt{YR}) \qquad \texttt{ACTION}\,(\texttt{RECOVERY\_D})}{\texttt{in\_after}\,(\texttt{YX},\,\texttt{YY})}$$

9.1.4 Concrete RECOVERY 2 action.

$$\frac{\texttt{in\_after}\,(\texttt{YX},\,\texttt{YY}) \qquad \texttt{insert\_RECOVERY}\,(\texttt{YX},\,\texttt{YY})}{\texttt{in\_before}\,(\texttt{YX},\,\texttt{YY})}\ \bullet$$

10 HOW TO USE a user interface

Figure 12: YERITH_QVGE user interface screenshot.



11 YRI_SD_RUNTIME_VERIF SPECIFICATION LANGUAGE

Figure 13 illustrates a "Backus-NAUR form (BNF)" of our specification language for YRI-DB-RUNTIME-VERIF tool.

12 Formal Scientific and Engineering Project Description

Detailed formal scientific and engineering contributions of design and testing system YERITH_QVGE can be found in ***JOURNAL ARTICLE "Runtime Verification Of SQL Correctness Properties with YRI-DB-RUNTIME-VERIF"*** [Nou23].

13 Conclusion

The graphical drawing tool YERITH_QVGE (Figure 5) costs only 2,500 EUROS. WE ONLY SUPPORT DEBIAN–LINUX (https://www.debian.org).

References

[Nou23] Xavier Noundou. A Framework for Verifying SQL Correctness Temporal Properties of [GUI] Software at Runtime. https://zenodo.org/records/13232567, October 2023.

Figure 13: **Grammar in Backus–Naur Form (BNF) of YRI_SD_RUNTIME_VERIF_LANG 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*⟩ ')'
| ⟨*in-spec-nop*⟩ '(' ')'

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

⟨*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

⟨*in-spec-nop*⟩ ::= IN_POST_NOP

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

⟨*not-in-spec-nop*⟩ ::= NOT_IN_POST_NOP

⟨*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