

# Chapter 3

## The Use of Dynamic Temporal Assertions for Debugging

Ziad A. Al-Sharif, Clinton L. Jeffery and Mahmoud H. Said

### 3.1 Introduction

The growth in the software industry is rapid and the size of programs is becoming larger and larger. In contrast, the rate of advances in the debugging literature is relatively slow. Most debuggers are well suited for a specific class or set of bugs. Program bugs can be caused by numerous circumstances and revealed long after their root cause. Understanding the source code and the execution behavior of the program is essential to locate and find the cause of most bugs. This understanding can be achieved by different means; one is to employ different debugging sessions that capture, depict, analyze, and investigate the state of the program at, and in between, different points of execution.

A typical interactive source-level debugger is one of the most valuable debugging tools, but it relies heavily on the user's ability to conduct a live investigation. It helps programmers locate and find the root cause of bugs by stepping through the source code and examining the current state of execution.

Source-level debugging techniques such as conditional breakpoints and watch-points are dynamically inserted during the debugging session. They can check execution properties and stop the execution whenever a condition is satisfied. Even though such breakpoints may have the advantage of being conditional and dynamic with on-the-fly *insertion*, *deletion*, and *modification*, they are still bounded to their locations; the exact line number in the source code of the target program and the current state of the referenced variables and objects at that location on that execution time. For instance, a class variable may be assigned a bad value in a method that is not on the stack when the bug that caused the crash or core dump is revealed. This may force

---

Z.A. Al-Sharif (✉) · M.H. Said  
Software Engineering Department, Jordan University of Science and Technology,  
P.O. Box 3030, Irbid 22110, Jordan  
e-mail: zasharif@just.edu.jo

M.H. Said  
e-mail: mhsaid@just.edu.jo

C.L. Jeffery  
Computer Science Department, University of Idaho, Moscow, ID 83844, USA  
e-mail: jeffery@uidaho.edu

© Springer Science+Business Media, LLC 2017  
D. Lettmin and M. Winterholer (eds.), *Embedded Software Verification and Debugging*, Embedded Systems, DOI 10.1007/978-1-4614-2266-2\_3

a user to run multiple debugging sessions on the same bug before it is understood. Typically, a user can investigate the current state. If there is no evidence of the bug's root cause, he/she may restart the execution hoping to stop at an earlier point where the cause of the bug is still accessible [6]. In contrast, Temporal Assertions are logical expressions that use Temporal Logic (TL) in order to validate, not one state, but a sequence of execution states, such as a sequence of variable values changed within a block of code [8–10].

In order to introduce Dynamic Temporal Assertions (DTA) into conventional source-level debugging sessions, for this research a source-level debugger named UDB [1, 4] was extended with on-the-fly temporal assertions (made from within the live debugging session). UDB is the source level debugger for the Unicon programming language; it is packaged with the Unicon language distribution on Source Forge and downloaded from *unicon.org*. Aside from the temporal assertions extension, UDB's command set is that of GDB. UDB was used instead of GDB for this research because its higher level execution monitoring abstractions allow UDB to be more easily extended than is GDB [2, 3].

The new DTA assertions that UDB supports are not bounded by the limitations of ordinary breakpoints such as *locality* and *temporality*. UDB's DTA assertions serve three purposes:

1. Extend the usability of conventional source-level debuggers' conditional breakpoints and watchpoints. This simplifies the ability to validate relationships that may extend over the entire execution and check information beyond the state of evaluation.
2. Reduce the number of times a user has to stop and single step the execution for state-based investigation.
3. Augment a traditional breakpoint-based debugging session with testing and verification capabilities [7].

### 3.1.1 DTA Assertions Versus Ordinary Assertions

Standard in-code assertions are inserted into the source code to validate pre- and post-conditions or to check the value of some variables and expressions. In general, typical ordinary assertions suffer from three limitations:

- *Locality*: An ordinary in-code assertion is bounded by its location (scope); it cannot reference a variable from another scope even if it is live based on the current execution state. Assertions live in one of the functions; each can reference local and global variables. If the scope is a method, it can reference any of the class variables. In fact, typical assertions cannot check or validate local variables in other functions or methods, even if that foreign local is static or still live somewhere on the stack of the current program's execution state. For example, what if a user needs to check the value of variable `x` from `procedure foo()` against variable `y` from `procedure bar()`? see Figs. 3.1 and 3.2.

```

10 procedure foo( )
11   local a, b, c
12   x := 10
13   .....
16   bar()
17   .....
19 end
20 procedure bar( )
21   local a, b, c
22   y := 20
23   .....
26   y := ( y * a ) / b - c
27   // virtually assert always() { y >= foo:x }
28   .....
30 end
    
```

Fig. 3.1 The possibility of a temporal assertion over two live procedures

```

10 procedure foo( )
11   static x := 0
12   x += 1
13   .....
30 end
31 procedure bar( )
32   static y := 0
33   y += 1
34   .....
60 end
61 procedure baz( )
62   foo()
63   .....
74   bar()
75   // virtually assert always() { bar:y >= foo:x }
76   .....
90 end
    
```

Fig. 3.2 The possibility of a temporal assertion over two sibling procedures

- Temporality*: An ordinary in-code assertion is bounded by the current state of execution. It can check only the current value of the referenced variables. For example, what if a user needs to check the value of variable  $x$  against the previous or even the initial value of  $x$ ? Ordinary assertions are found to be useless once more.

- *Dynamicity*: An ordinary in-code assertion is bounded to the source code, where it is written and compiled; any change or modification requires the ability to recompile and rebuild the executable. If the ordinary assertion evaluates to `false`, it may provide a warning statement or terminate the execution. If the user wants to investigate, he/she may modify the assertion by tightening or loosening the condition, or adding nearby assertions.

### 3.1.2 DTA Assertions Versus Conditional Breakpoints

Conditional breakpoints and watchpoints are dynamically inserted during the debugging session. They can check execution properties and stop the execution whenever a condition is satisfied. Even though such breakpoints may have the advantage of being conditional and dynamic with on-the-fly *insertion*, *deletion*, and *modification*, they are still bounded to their locations; the exact line number in the source code of the target program and the current state of the referenced variables and objects at that location on that execution time. Whereas, DTA assertions are able to reference variables that are not accessible (not active in the current execution state) at evaluation time. This feature solves the problem provided in Fig. 3.2, which shows that procedure `foo()` and `bar()` are siblings in `baz()`.

## 3.2 Debugging with DTA Assertions

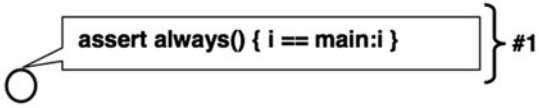
In general, users of source-level debuggers suffer from:

1. The limited information provided about the execution history, and
2. The lack of automated trace-based and analysis-based debugging techniques, which may help users validate various execution states.

DTA assertions, within a typical source-level debugger, provide an extension of conditional breakpoints and watchpoints. They employ agents that implement temporal logic operators, each with an automatic tracing mechanism. Traced data are *assertion-driven*; relevant information is gathered and analyzed in real time. Different DTA assertions can be applied on different execution properties with dynamicity and flexibility. Each assertion is capable of validating program properties that may extend over a sequence of execution states [14].

For example, a debugging process may include checking variable values from different scopes. Figure 3.3 shows a program that prints out the prime numbers from 1 to some `x`. The procedure `main()` calls `isPrime()`, which returns `true` when the passed argument is a primary number. The temporal assertion provided in #1 of Fig. 3.3 shows how to check the current local value of variable `i` against the last value of variable `i` of the procedure `main()` (denoted by `main:i`). This DTA assertion assumes that the value of parameter `i` should not change during the

```

1  procedure main()
2      local x, i
3
4      writes(" Please enter a positive integer number : ")
5      x := read()
6
7      write("\n The following are the primary numbers <= ", x)
8      every i := 1 to x do
9          if isPrime(i) then
10             write(i, " is a primary number ")
11         end
12
13     procedure isPrime( i )
14         local k
15
16         k := i
17         i -= 1
18         while (i > 1) do
19             {
20                 if k % i = 0 then
21                     fail
22                     i -= 1
23                 }
24             return k
25         end
26


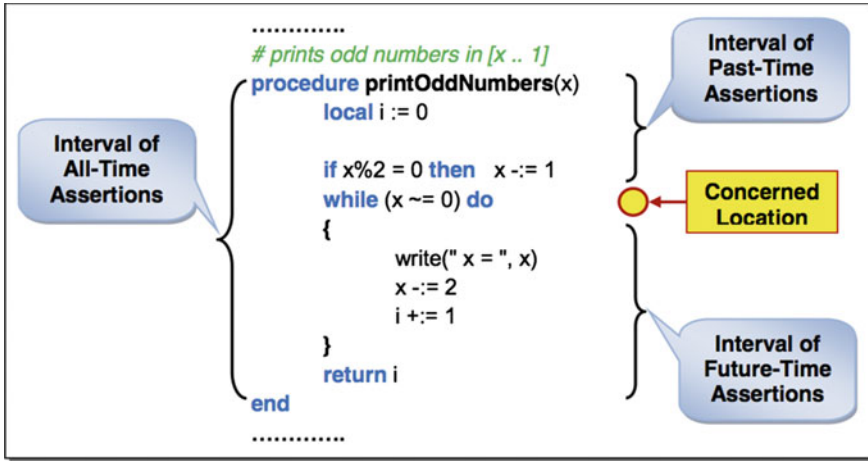
```

Fig. 3.3 Using temporal assertions to check variables from various scopes

execution of `isPrime()`. However, because the program is modifying the value of `i`, this assertion will evaluate to `false` at every change to `i` (temporal-state) in this `isPrime()` function, and it will evaluate to `false` at every return from this `isPrime()` function (temporal-interval).

### 3.3 Design

DTA assertions do not replace traditional breakpoints or watchpoints, instead they provide a technique to reduce their number, which means they are used to reduce the number of execution stops and improve the overall process of investigation. These temporal assertions advance breakpoints with agents of temporal logic operators (temporal agents). At a stop, besides the source-level debugging functionalities, the user can *delete*, *enable*, *disable*, and *modify* existing assertions, or even *insert* new



**Fig. 3.4** Temporal assertions: scope and interval

**Table 3.1** Atomic data related agents

Agent name	Return type	Description
<i>initial(x)</i>	Any	The initial value of <i>x</i>
<i>final(x)</i>	Any	The final value of <i>x</i>
<i>old(x)</i>	Any	The previous value of <i>x</i>
<i>current(x)</i>	Any	The current value of <i>x</i>
<i>new(x)</i>	Any	The next value of <i>x</i>
<i>max(x)</i>	Numeric	The maximum of all <i>x</i> values
<i>min(x)</i>	Numeric	The minimum of all <i>x</i> values
<i>newmax(x)</i>	True/false	Evaluate <i>True</i> if <i>x</i> has new max, <i>False</i> otherwise
<i>newmin(x)</i>	True/false	Evaluate <i>True</i> if <i>x</i> has new min, <i>False</i> otherwise
<i>sum(x)</i>	Numeric	The sum of all <i>x</i> values
<i>avg(x)</i>	Numeric	The average of all <i>x</i> values

DTA assertions at any location in the buggy program source code; all without the need to recompile the target program source code or to reload it under the debugger.

UDB supports three kinds of DTA assertions, see Fig. 3.4. Each of these kinds has its own set of temporal agents. All these DTAs can reference execution properties and other internal extension agents such as the atomic data agents described in Table 3.1 and the behavioral agents described in Table 3.2.

**Table 3.2** Atomic execution behavior-related agents

Agent name	Return type	Description
<i>call(proc)</i>	Integer	The number of times <i>proc</i> is been called
<i>return(proc)</i>	Variable	The current value returned by <i>proc</i>
<i>initialized(x)</i>	True/false	<i>True</i> if <i>x</i> was assigned at first reference, <i>False</i> otherwise
<i>dead(x)</i>	True/false	<i>True</i> if <i>x</i> is never referenced at least once, <i>False</i> otherwise
<i>reference(x)</i>	Integer	The number of times <i>x</i> is been read and written
<i>assign(x)</i>	Integer	The number of times <i>x</i> is been assigned
<i>read(x)</i>	Integer	The number of times <i>x</i> is been read only
<i>alias(x)</i>	List	All current <i>x</i> aliases
<i>iterations(loop)</i>	Integer	The number of actual iterations of <i>loop</i>

### 3.3.1 Past-Time DTA Assertions

This category consists of four Past-Time Operators. These operators utilize information retained between entering an assertion's scope and a reaching assertion's source code location. At insertion time, the debugger starts retaining relevant information to be used during the assertion's evaluation. When the execution reaches the virtual execution point, where the assertion is hooked in the buggy program space, the assertion temporal interval is evaluated. If the evaluation is not able to complete due to some missing information (maybe out-of-scope referenced data is never used during an assertion's lifetime), the assertion evaluation is tagged with `Not Valid`. These four DTA assertions are:

1. `alwaysp() {expr}`: asserts that an expression must always hold (evaluate to true) for each, temporal state, temporal interval, and during the whole execution.
2. `sometimep() {expr}`: asserts that an expression must hold at least once for each temporal interval, and during the whole execution.
3. `previous() {expr}`: asserts that an expression must hold right at the last state before the end of the temporal interval.
4. `since() {condition ==> expr}`: asserts that an expression must hold right after condition is true up until the end of the temporal interval and for each interval.

### 3.3.2 Future-Time DTA Assertions

This category consists of four Future-Time Operators. These operators utilize information retained between reaching an assertion's source code location and leaving an assertion's scope. The agents of those operators start watching for referenced

objects when the evaluation is triggered, where the debugger starts retaining relevant information until the assertion's temporal interval is evaluated completely. If the execution is terminated before the assertion's interval is complete, the user is able to check temporal states in that incomplete temporal interval. These four DTA assertions are:

1. `alwaysf() {expr}`: asserts that an expression must always hold (evaluate to true) for each, state, temporal interval, and during the whole execution.
2. `sometimef() {expr}`: asserts that an expression must hold at least once for each temporal interval, and during the whole execution.
3. `next() {expr}`: asserts that an expression must hold right at the very first state in the temporal interval.
4. `until() {condition ==> expr}`: asserts that an expression must hold from the beginning of the temporal interval up until `condition` is true or the end of the temporal interval and for each interval.

### 3.3.3 All-Time DTA Assertions

This category consists of two All-Time Operators. These two operators are based on the time interval between entering an assertion's scope and exiting an assertion's scope. When the assertion scope is entered, the assertion starts retaining relevant information and evaluates its temporal states. When the execution exits the assertion scope, the assertion temporal interval is evaluated. These two DTA assertions are:

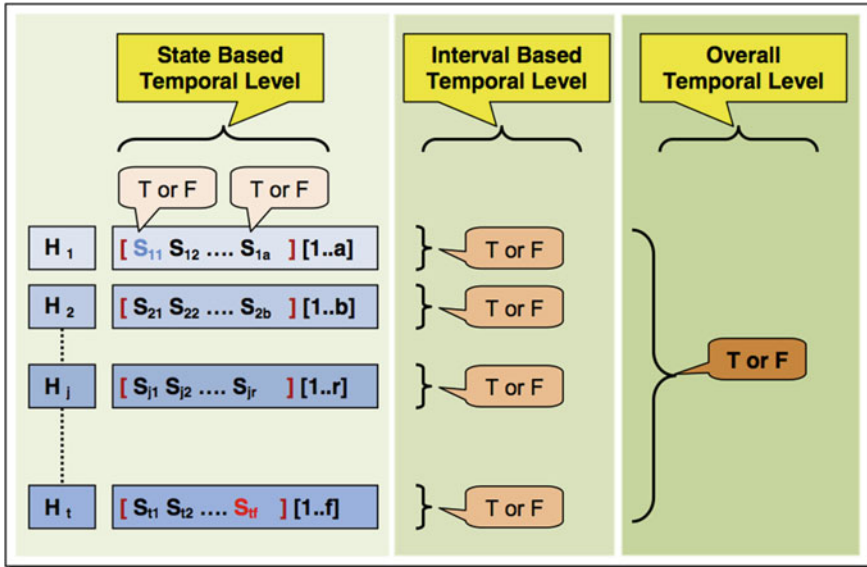
1. `always() {expr}`: asserts that an expression must always hold (evaluate to true) for each, state, temporal interval, and during the whole execution.
2. `sometime() {expr}`: asserts that an expression must hold at least once for each temporal interval, and during the whole execution.

## 3.4 Assertion's Evaluation

Each reached (evaluated) assertion has at least one temporal interval. This interval consists of a sequence of temporal states. Temporal interval is defined by the assertion scope and kind. Assertion's scope is defined based on the source code location provided in the `assert` command. This scope is the procedure or method surrounding the assertion location. Figure 3.4 shows the temporal interval for all three kinds of temporal assertions in reference to the provided location. Together, the assertion's scope and kind define the temporal interval. In particular:

- *Temporal Intervals of Past-Time* DTA assertions start at entering the assertion's scope (calling the scope procedure) and end at reaching assertion's source code location for the very first time after entering the scope.



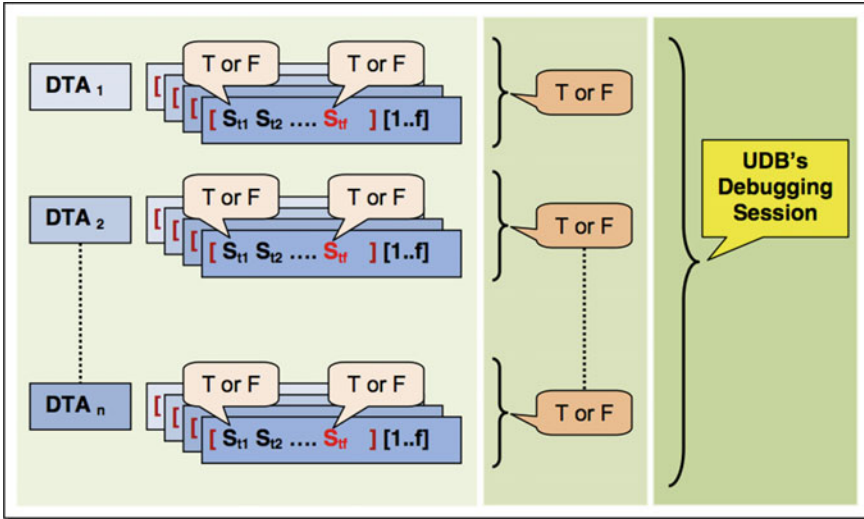


**Fig. 3.5** Sample Temporal Assertion’s Evaluation: An assertion is hit  $t$  times  $[H_1..H_t]$ . Each hit represents a Temporal Interval, which consists of a various number of states; each state is evaluated to `True` or `False`. Each Temporal Interval is evaluated based on the conjunctive normal form of its state-based evaluations (on that particular hit  $H_i$ ). Finally, on the overall temporal level, the assertion is evaluated once more based on the conjunctive normal form of all previous interval-based evaluations

- *Temporal Intervals of Future-Time* DTA assertions start at reaching an assertion’s source code location for the very first time after entering the assertion’s scope and end at exiting the assertion’s scope (returning from the scope procedure). In this kind of temporal assertions, the source code location can be hit more than once before the interval is closed.
- *Temporal Intervals of All-Time* DTA assertions start at entering assertion’s scope and end at exiting that scope; regardless of the provided source code location.

During a debugging session, it is possible for a user to have multiple assertions, each with multiple temporal intervals, and each interval with multiple temporal states. See Figs. 3.5 and 3.6. Each DTA assertion runs through three levels of evaluations:

1. *State-based:* temporal level (single state change). This evaluation is triggered by any change to the assertion referenced objects.
2. *Interval-based:* a sequence of consecutive states. This evaluation is triggered by reaching the end of assertion’s temporal interval (exiting the assertion scope).
3. *Overall execution-based:* a sequence of consecutive temporal intervals. This evaluation is triggered by the end of execution.



**Fig. 3.6** Sample evaluation of various temporal assertions ( $n$  DTA assertions) during a debugging session

UDB's DTA assertions are evaluated in the debugger side. By default, whenever an assertion evaluates to `false`, the source-level debugger stops execution in a manner similar to a breakpoint. The debugger transfers control to the user with an evaluation summary.

### 3.4.1 Temporal Cycles and Limits

A temporal `cycle` defines the maximum number of consecutive temporal intervals (maximum number of temporal level evaluation times), which defines the overall evaluation. The default value of `cycle` is unlimited number of evaluations. Temporal `limit` defines the maximum number of temporal states considered in each temporal interval. The definition of temporal limit is changed based on the kind of temporal assertion in reference. In particular:

1. In Past-Time DTA assertions: `limit` defines the maximum number of consecutive states before reaching assertion's source code location and after entering the assertion's scope.
2. In Future-Time DTA assertions: `limit` defines the maximum number of consecutive states after assertion's source code location is reached and before exiting the assertion's scope.

3. In All-Time DTA assertions: `limit` defines the maximum number of states before and after an assertion's source code location is reached, all within the assertion's scope.

The default limit is defined by whatever temporal states (temporal-interval) are encountered during the execution of an assertions' scope and based on its temporal interval. The user can reduce the number of temporal states considered in each temporal interval by setting this limit using the `limit` command.

### 3.4.2 *Evaluation Log*

Furthermore, the assertion's log gives the user the ability to review the evaluation behavior of each assertion (evaluation history). The debugger maintains a hash table for each assertion. It maps assertions' intervals into lists with information about their temporal state base evaluation. Each list reflects a temporal interval, which maintains the evaluation order and result for each temporal state. Each list reflects one temporal interval, also maintained based on their order. Completely evaluated intervals are tagged with `True` or `False`. If the evaluation process is already started, but the final result is still incomplete, perhaps the end of the interval is not reached yet, these intervals are tagged with `Pending` until they are complete. This will convert `Pending` into `True` or `False`. However, some assertions may never be triggered for evaluation; this may occur because the execution never reached the assertion's insertion point during a particular run. These assertions have the hit counter set to zero.

### 3.4.3 *DTA Assertions and Atomic Agents*

Atomic agents are a special kind of extension agents (nontemporal logic agents) [1, 3, 4]. They expand the usability of DTA assertions and facilitate the ability to validate more specific data and behavioral relationships over different execution states, see Tables 3.1 and 3.2. When an atomic agent is used within a DTA assertion, it retains and processes data and observes behaviors in relevance to the used assertions. The assertion scope is what determines when the agent should start to work and what range of data it should be able to retain and process. For example, if the assertion uses the `max(var)` or `min(var)` atomic agents, the agent always retains the maximum or minimum respectively over the assertion temporal interval.

Those atomic agents add more advancement and flexibility to the usefulness of DTA assertions and their basic temporal logic operators. In particular, DTA assertions that reference atomic agents can easily check and compare data obtained by these atomic agents, which encapsulate simple data processing such as finding the minimum, maximum, sum, number of changes, or average. For example, suppose

1. (udb) **assert** test.icn:50 sometimep() { x < y }
2. (udb) **assert** test.icn:50 alwaysp() { old(x) != current(x) }
3. (udb) **assert** test.icn:50 alwaysf() { return(foo) > 0 }
4. (udb) **assert** test.icn:50 always() { iteration(while) < 100 }
5. (udb) **assert** test.icn:50 always() { call(baz) < 1000 }

Fig. 3.7 Sample of different UDB's temporal assertions

that a static variable is changed based on a conditional statement where it is incremented when the condition is `true` and decremented when the condition fails. What if the user is interested in the point at which this variable reaches a new maximum or minimum? DTA assertions provide a simple solution for such situations.

For example, the assertion number 1 of Fig. 3.7 will pause the execution when variable `x` becomes greater than or equal to `y`. As another example, suppose the user is interested in the reasons behind an infinite recursion; perhaps a key parameter in a recursive function is not changing. DTA assertions provide a mechanism to retain the parameter value from the last call and compare it with the value of the current call, see assertion number 2 of Fig. 3.7. If `old(x) == current(x)`, the assertion will stop the execution and hand control to the debugger where the user can perform further investigation. Of course, there are other reasons that may cause infinite recursion, such as the key parameter value changing in the opposite directions on successive calls.

Moreover, DTA assertions simplify the process of inserting assertions on program properties such as functions' return values, and loops' number of iterations. For example, a user may insert a breakpoint inside a function in order to investigate its return value, or place an in-code assertion on the value of the returned expression. A DTA assertion provides a simpler mechanism; see assertion number 3 of Fig. 3.7. Assertion number 4 of Fig. 3.7 states that the while loop at line 50 in file `test.icn` always iterates less than 100 times. Finally, assertion number 5 of Fig. 3.7 shows how to place a DTA assertion on the number of calls to a function; the assertion will stop execution at call number 1000. This particular assertion is difficult to accomplish using conventional source-level debugging features such as breakpoints and watchpoints.

### 3.5 Implementation

DTA assertions are virtually inserted into the buggy program source code on-the-fly during the source-level debugging session. UDB’s static information is used to assist the user and check the syntax and the semantic of the inserted assertion. Each assertion is associated with two sets of information (1) *event-based* and (2) *state-based*. The debugger automatically analyzes each assertion at insertion time in order to determine each set. It finds the kind of agents that are required to be encountered in the evaluation process. If any extension agent is used, the debugger establishes an instance of that agent and associates it with its relevant object.

The host debugger maintains a hash table that maps each assertion source code location into its related object (agent). The assertion object is responsible for maintaining and evaluating its assertion. It contains information such as (1) the parsed assertion, (2) a list of all referenced variables (3) a list with all temporal intervals and their temporal states, and (4) the assertion event mask: a set of event codes to be monitored for each assertion; this event mask includes the event masks for any of the referenced agents. Execution events are acquired and analyzed in real time. Some events are used to control the execution whereas others are used to obtain information in support of the state-based technique [2, 5].

Each assertion has its own event and value masks, which are constructed automatically based on the assertion, see Fig. 3.8. A union set of all enabled assertion event masks is unified with the debugging core event mask. The result is a set of events requested by the debugging core during the execution of the buggy program. This set is recalculated whenever an assertion is added, deleted, enabled, or disabled. On-the-fly, UDB’s debugging core starts asking the buggy program about this new set of events. A change on any assertion event mask alters the set of events forwarded by the

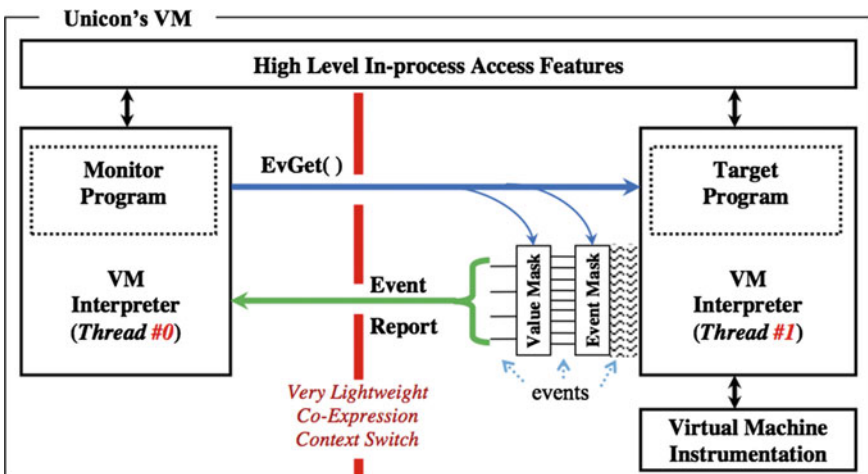


Fig. 3.8 UDB’s use of event mask and event value within the Unicon virtual machine

debugging core to that assertion object. Temporal logic agents automatically obtain the buggy program state-based information to evaluate DTA assertions. Each agent automatically watches assertion referenced variables and retains their information in the debugger space.

### 3.6 Evaluation

DTA assertions provide the ability to validate relationships that may extend over the entire execution and check information beyond the current state of evaluation. DTA assertions' temporal logic operators are internal agents. Those agents can reference other atomic agents, which provide access to valuable execution data and behavior information. UDB's DTA assertions have the following features:

- Dynamic insertion, deletion, enabling, disabling, and modification. Assertions are managed on-the-fly during the debugging session without source or executable code alteration.
- A nondestructive way of programming supported by an assertion-free source code. In general, debugging information is needed only during program development, testing, verification, validation, and debugging.
- Assertions are virtually inserted and evaluated as part of the buggy program source code. All assertions live in the debugging session configuration; each is evaluated by the debugger in the debugger execution space. The debugger automatically maintains state-based techniques to determine what information is needed to evaluate each assertion, and it uses event-based techniques to determine when and where to trigger each assertion evaluation process. Some program state-based information is collected before assertion evaluation, while other information is obtained during the evaluation process. All DTA assertions are evaluated as if they were part of the target program space
- Optional evaluation suite, where a user can specify an evaluation action such as `stop`, `show`, and `hide`. The `show` action enriches assertions with the sense of in-code tracing and debugging with print statements, where a user can ensure that the evaluation has reached some points and the referenced variables satisfy the condition.
- The ability to log the assertion's evaluation result. This lets the user review the assertion evaluation history for a specific run. Evaluated assertions are marked with `True` or `False`. Some DTA assertions may reference data in the future; those assertions are marked with `Not Valid` for that exact state-based evaluation. Assertions' intervals are marked with a counter that tracks their order in the execution. If an assertion has never been reached, it is distinguished by its counter value, which is zero in this case. Log comparison of different runs is considered in future works.
- Most importantly, DTA assertions can go beyond the scope of the inserted location. Each assertion may refer to variables or objects that were living in the past during

previous states, but not at evaluation point, and each assertion may compare previous variable values against current or future values. Each DTA assertion implicitly employs various agents to trace referenced objects and retains their relevant state information in order to be used at evaluation time.

### 3.6.1 Performance

In consideration of the performance in terms of time, the implementation of temporal assertions utilizes a conservative assertion-based event-driven tracing technique. It only monitors relevant events; the event mask and value mask are generated automatically for each assertion at insertion time. Temporal assertions are evaluated in three levels. First is the state-based level, which depends on any change to the referenced execution property. Second is the interval-based level, which is determined by the assertion scope and kind. Third is the overall evaluation level, which occurs once per each execution. Different assertions can reference different execution properties. For this reason various assertions will differ in their cost.

However, in order to generally assess the role of the three evaluation levels in the complexity of these temporal assertions, let us assume that  $E_s$  is the maximum cost of monitoring and evaluating a state change within a temporal assertion. Furthermore, let us assume that  $n$  is the maximum number of state changes during a temporal interval and  $m$  is the maximum number of temporal intervals during an execution. See Figs. 3.5 and 3.6. This means, the maximum cost of evaluating a temporal interval for this assertion is  $E_s * n$  and the maximum cost of an assertion during the whole execution is  $(E_s * n) * m$  which is equal to  $E_s * n * m$ . However,  $E_s$  includes the cost of event forwarding. This means that part of  $E_s$  is  $(2E_p + 2E_c)$ , where  $E_c$  is the cost of reporting an event to UDB and  $E_p$  is the cost of forwarding an event to the temporal logic agent (internal agent). This means the  $E_s$  dominates both  $n$  and  $m$ ; state change is the main performance issue in temporal assertions.

Furthermore, retained information is limited and driven by assertions' referenced execution properties. Assertions are virtually evaluated because they are in another execution space. The evaluation occurs in the debugger space with data collected and obtained from the buggy program space. The assertion log gives the user the ability to review the evaluation behavior of each assertion. Temporal assertions use in-memory tracing. A table is allocated for all assertions; it maps each assertion source code location to the instance object of the actual assertion. Another table is allocated for each assertion; it tracks temporal intervals, each of which is a list (stack) with each of the state-based evaluation result. A third table is used to map assertion temporal intervals with their evaluation result, each of which is one value `True`, `False`, or `Not Valid`. Then one variable is holding the up to the point result which is either true or false. The dominating part in the used space is the number of state changes,  $E_s$ . Each state base evaluation is tracked with a record that keeps information about the line number, file name, and the result.

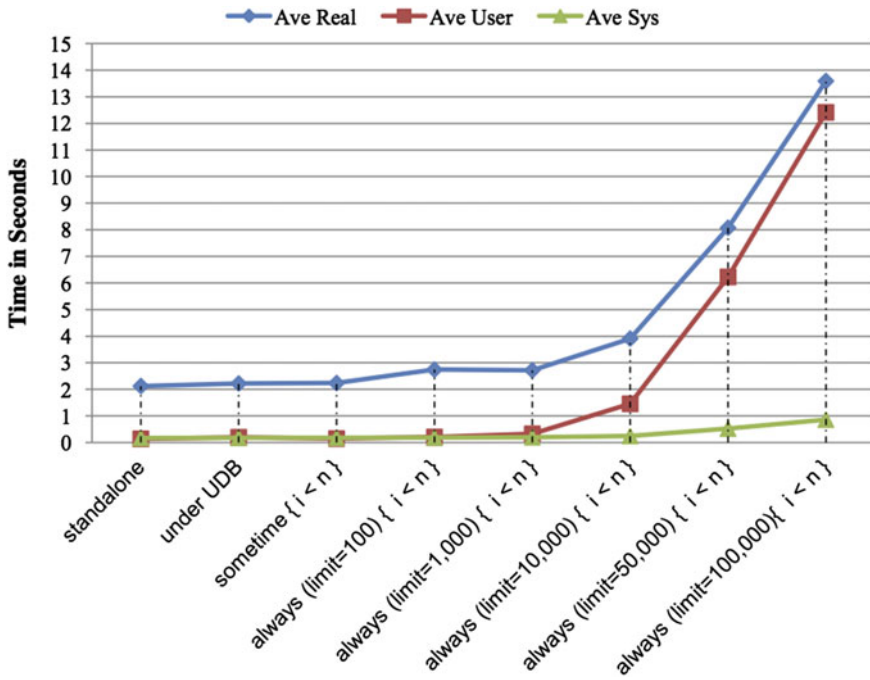


Fig. 3.9 UDB's temporal assertions evaluation time

In order to find the impact of temporal assertions on the execution of the target program and the debugging time, a simple temporal assertion is applied on a simple program. The program prints numbers between 1 and 100,000; see Fig. 3.9. The temporal assertion is applied with various sizes of temporal intervals. These intervals start at size 1, 100, 1000, 10,000, 50,000, and 100,000. The experiment is based on eight kinds of runs, each is observed for five times and the average of these times is reported. These kinds of runs range from measuring the time for the program in the standalone mode (no monitoring is involved), monitored under UDB with no assertion applied, then with an assertion that has various intervals. Figure 3.9 shows the impact of these temporal assertions on the execution time.

### 3.7 Challenges and Future Work

Debugging with DTA assertions provides advantages over typical assertions and conditional breakpoints and watchpoints. At the same time, it faces some challenges and limitations, some of which are based on associating assertions with the executable's source code, evaluating assertions in the debugger, and the source-level debugger's ability to obtain and retain relevant event-based and state-based information with reasonable performance.



First, if an assertion makes a reference to a variable, which is not accessible from within the assertion's scope, the debugger should automatically trace those variables and retain their relevant state information to be used at the assertion evaluation time. This allows a DTA assertion to access data that is not live at the assertion's evaluation time.

Second, what if the assertion source code location is overlapping with a statement? Which one should be evaluated first, the assertion or the statement? A conservative approach may consider the assertion evaluation after the statement only if the statement has no variables referenced by the assertion, or if the statement does not assign to any of the assertion referenced variables. However, if the statement will assign to any of the assertion referenced variable, the assertion can be evaluated before and after the statement evaluation. If the two evaluations are different such as one is `true` and the other is `false`, or both are `false`, the assertion will stop the execution and hand the control to the debugger and the user to investigate. The work presented in this paper, takes the simplest approach which is to evaluate the assertion before the statement. Furthermore, if an assertion is not overlapping with an executable statement, the AlamoDE framework cannot report a line number event from a nonexecutable line. A line number event is only reported when a statement in that line number is fetched to be executed. This is reached by checking the assertion source code location before confirming that the assertion is inserted successfully. It checks whether the line number is empty or it is commented out.

Finally, if a referenced variable is an object or a data structure such as a list, this can cause two problems. First, the object is subject to changes under other names because of aliasing. Second, if the object is local, it may get disposed by the garbage collector before the evaluation time. The implementation could be extended to implement trapped variables that would allow us to watch an element of a structure or utilize an aliasing tracing mechanism to retain all changes that may occur under different names. The implementation of temporal assertions presented in this paper does not go after heap variables, which is left for future work.

### 3.8 Conclusion

DTA assertions bring an extended version of in-code assertion techniques, found in mainstream languages such as C/C++, Java, and C#, into a source-level debugging session. These temporal assertions help users test and validate different relationships across different states of the execution. Furthermore, assertion evaluation actions such as `show` provide the sense of debugging and tracing using print statements from within the source-level debugging session. They give the user a chance to know that the execution has reached that point and the asserted expression evaluated to `true`; it also gives the user the ability to interrupt and stop the execution for more investigation. The ability to log the assertion evaluation result provides the user with the ability to review the evaluation process. A user can check a summary result of what went wrong and what was just fine.

Source-level debuggers provide the ability to conditionally stop the execution through different breakpoints and watchpoints. At each stop, a user will manually investigate the execution by navigating the call stack and variable values. Source-level debuggers require a user to come up with assumptions about the bug and let him/her manually investigate those assumptions through breakpoints, watchpoints, single stepping, and printing. In contrast, DTA assertions require the user to come up with logical expressions that assert execution properties related to a bug's revealed behavior and the debugger will validate these assertions. Asserted expressions can reference execution properties from different execution states, scopes, and over various temporal intervals. Furthermore, unlike conditional breakpoints and watchpoints, which only evaluate the current state, DTA assertions are capable of referencing variables that are not accessible at evaluation time (not active in the current execution state).

DTA assertions do not replace traditional breakpoints or watchpoints, but they offer a technique to reduce their number and improve the overall investigation process. DTA assertions reduce the amount of manual investigation of the execution state such as the number of times a buggy program has to stop for investigation.

Finally, debugging with temporal assertions is not new. In 2002 Jozsef Kovacs et al. has integrated Temporal Assertions into a parallel debugger to debug parallel programs [12]. In 2005 Volker Stolz et al. used LTL over AspectJ pointcuts to validate properties during program execution that are triggered by aspects [11]. In 2008 Cemal Yilmaz et al. presented an automatic fault localization technique using time spectra as abstractions for program execution [13]. However, to the best of our knowledge, we are the first to extend a typical source level debugger's features of conditional breakpoints and watchpoints with commands based on temporal assertion that capture and validate a sequence of execution states (temporal states and temporal intervals). Furthermore, these assertions can reference out-of-scope variables, which may not be live in the execution state at evaluation time.

## References

1. Al-Sharif Z, Jeffery C (2009) UDB: an agent-oriented source level debugger. *Int J Softw Eng* 2(3):113–134
2. Al-Sharif Z, Jeffery C (2009) Language support for event-based debugging. In: *Proceedings of the 21st international conference on software engineering and knowledge engineering (SEKE 2009)*, Boston, July 1–3, 2009, pp 392–399
3. Al-Sharif Z, Jeffery C (2009) A multi-agent debugging extension architecture. In: *Proceedings of the 21st international conference on software engineering and knowledge engineering (SEKE 2009)*, Boston, July 1–3, 2009, pp 194–199
4. Al-Sharif Z, Jeffery C (2009) An agent-oriented source-level debugger on top of a monitoring framework. In: *Proceedings of the 2009 sixth international conference on information technology: new generations*, vol 00, April 27–29, 2009. ITNG. IEEE Computer Society, pp 241–247. doi:[10.1109/ITNG.2009.305](https://doi.org/10.1109/ITNG.2009.305)

5. Al-Sharif Z, Jeffery C (2009) An extensible source-level debugger. In: Proceedings of the 2009 ACM symposium on applied computing, Honolulu, Hawaii. SAC '09. ACM, New York, NY, pp 543–544. doi:[10.1145/1529282.1529397](https://doi.org/10.1145/1529282.1529397)
6. Boothe B (2000) Efficient algorithms for bidirectional debugging. In: Proceedings of the ACM SIGPLAN 2000 conference on programming language design and implementation, Vancouver, British Columbia, Canada, June 18–21, 2000. PLDI '00. ACM, New York, NY, pp 299–310
7. Drusinsky D, Michael B, Shing M (2008) A framework for computer-aided validation. *Innov Syst Softw Eng* 4(2):161–168
8. Drusinsky D, Shing M (2003) Monitoring temporal logic specifications combined with time series constraints. *J Univ Comput Sci* 9(11):1261–1276. [http://www.jucs.org/jucs\\_9\\_11/monitoring\\_temporal\\_logic\\_specification](http://www.jucs.org/jucs_9_11/monitoring_temporal_logic_specification)
9. Drusinsky D, Shing M, Demir K (2005) Test-time, run-time, and simulation-time temporal assertions in RSP. In: Proceedings of the 16th IEEE international workshop on rapid system prototyping, June 08–10, 2005. RSP. IEEE Computer Society, Washington, DC, pp 105–110. doi:[10.1109/RSP.2005.50](https://doi.org/10.1109/RSP.2005.50)
10. Koymans R (1990) Specifying real-time properties with metric temporal logic. *Real-Time Systems* vol 2, no 4, Oct 1990. Kluwer Academic Publishers, Norwell, MA, USA, pp 255–299. doi:[10.1007/BF01995674](https://doi.org/10.1007/BF01995674)
11. Volker S, Eric B (2006) Temporal assertions using aspect. *Electron Notes Theory Comput Sci* 144(4):109–124
12. Jozsef K, Gabor K, Robert L, Wolfgang S (2002) Integrating temporal assertions into a parallel debugger. In: Euro-Par'02, Monien B, Feldmann R (eds) Proceedings of the 8th international euro-par conference on parallel processing. Springer, London, UK, pp 113–120
13. Yilmaz C, Paradkar A, Williams C (2008) Time will tell: fault localization using time spectra. In: Proceedings of the 30th international conference on software engineering (ICSE '08). ACM, New York, NY, USA, pp 81–90. doi:[10.1145/1368088.1368100](https://doi.org/10.1145/1368088.1368100)
14. Al-Sharif ZA, Jeffery CL, Said MH (2014) Debugging with dynamic temporal assertions. In: IEEE international symposium on software reliability engineering workshops (ISSREW), 3–6 Nov 2014, pp 257–262. doi:[10.1109/ISSREW.2014.60](https://doi.org/10.1109/ISSREW.2014.60)