Source-level runtime validation through interval temporal logic

Karlston D'Emanuele and Gordon Pace

University of Malta

Abstract. The high degree of software complexity achievable through current software development practices makes software more prone to failure. A number of work and work practices has evolved in order to reduce risks related to software correctness and reliability. One of which is validation, which monitors the system execution at runtime and verifies that the system states entered are valid according to the behavioural specification.
This paper describes a framework providing an assertion like validation environment for integrating software properties specified in interval temporal logic. The framework consists in three parts. The first part provides a mechanism for converting the human readable assertion to a symbolic automata, which is then used by the second part of the framework that performs the validation.

1 Introduction

A common problem in software development or maintenance is that the software behaviour is not what expected. The main cause is that it is very difficult to map behavioural specifications in the system's code. A solution can be to fully test the system but with the complexities in software that are being reached with current development techniques, it is infeasible.

Validation is another solution, which through logic based monitors the system is checked for correctness during runtime. Another important aspect of validation is that opposing to verification the system specifications do not need to be abstracted, especially if the monitors are placed local to the area effected. Validation monitors can be run synchronously with the system, in order to provide a more reliable solution for capturing errors on occurrence.

This paper presents a solution for integrating interval temporal logic inside a system core in order to perform runtime validation. The solution emphasises on guaranteeing that the memory and time necessary for the validation process can be estimated before commencing evaluation. These guarantees are attained by converting logical-based formulas into their equivalent Lustre [6] symbolic automata, where the memory and time required for evaluation are constants. Then a simple framework for the integration of the solution in a development environment is outlined.

The paper is organised as follows, in the next section quantified discrete duration calculus (QDDC) notation together with its semantics is introduced.

Section 3 deals with the framework which provides validation mechanisms for integration into the system code. Finally concluding by mentioning some work that has been performed in the validations area.

2 QDDC notation

Quantified discrete duration calculus [9] is an interval temporal logic that checks satisfiability of properties within an interval rather than the more common techniques of sampling values or that of checking satisfiability at the end of intervals.

2.1 Syntax and Semantics

On the assumption that the system state has finite variability and that changes to the system state are performed sequentially, let σ be a non-empty sequence of state variables evaluations,

$$\sigma \mathrel{\widehat{=}} (\text{state variable} \mapsto \mathbb{B})^{+}.$$

A state variable is a proposition and its syntax is defined as

$$P ::= 0 \mid 1 \mid p \mid P \ op \ P \mid \neg P \mid -P \mid +P$$

where p is a proposition variable and $op \in \{\wedge, \vee, \Rightarrow, \Leftrightarrow\}$.

The QDDC syntax is defined as,

$$QDDC ::= \lceil\!\lceil P \rceil\!\rceil \mid \lceil P \rceil^{0} \mid D_1 \widehat{} D_2 \mid D_1 \ b\_op \ D_2 \mid \neg D \mid \exists p \cdot D \mid \eta \ c\_op \ c \mid \Sigma P \ op \ c$$

where c is a constant, $b\_op \in \{\wedge, \vee, \Rightarrow, \Leftrightarrow\}$ and $c\_op \in \{<, \leq, =, \geq, >\}$.

Let $\sigma[b, e] \models D$ to denote that a finite non-empty sequence σ satisfies the QDDC formula D between the two discrete time clocks noted as b and e. Leaving propositions to have their standard mathematical definition and $\#(\sigma)$ denote the number of evaluations in the sequence σ, then the QDDC notation semantics is as follows

$\sigma_i \models P$	**iff**	P is true at time clock i.
$\sigma_i \models -P$	**iff**	$i > 0 \wedge \sigma_{i-1} \models P$.
$\sigma_i \models +P$	**iff**	$i < \#(\sigma) - 1 \wedge \sigma_{i+1} \models P$.
$\sigma[b, e] \models \lceil\!\lceil P \rceil\!\rceil$	**iff**	$\forall i \in [b, e) \cdot \sigma_i \models P$.
$\sigma[b, e] \models \lceil P \rceil^{0}$	**iff**	$b = e \wedge \sigma_b \models P$.
$\sigma[b, e] \models D_1 \ b\_op \ D_2$	**iff**	$\sigma[b, e] \models D_1 \ b\_op \ \sigma[b, e] \models D_2$.
$\sigma[b, e] \models \neg D$	**iff**	$\sigma[b, e] \not\models D$.
$\sigma[b, e] \models \eta \ c\_op \ c$	**iff**	$(e - b) \ c\_op \ c$.

$$\sigma[b,e] \models \Sigma P \ c\_op \ c \quad \textbf{iff} \quad \sum_{i=b}^{e} \begin{cases} 1 & \sigma_i \models P \\ 0 & otherwise \end{cases} \ c\_op \ c.$$

$$\sigma[b,e] \models \exists p \cdot D \quad \textbf{iff} \quad \exists \sigma' \cdot \sigma'[b,e] \models D \text{ and } \forall i \in [b,e], \forall q \in P \cdot q \neq p \wedge \sigma'_i(q) = \sigma_i(q).$$

$$\sigma[b,e] \models D_1 \hat{} D_2 \quad \textbf{iff} \quad \exists i \in [b,e] \cdot \sigma[b,i] \models D_1 \wedge \sigma[i,e] \models D_2$$

A number of derivable operators simplify the use of the notation. The table below defines four of the mostly used operators.

$$\begin{aligned} \diamond D &\ \widehat{=}\ true \hat{} D \hat{} true. \\ \Box D &\ \widehat{=}\ \neg \diamond \neg D. \\ P \xrightarrow{\delta} Q &\ \widehat{=}\ \Box((P \wedge \eta \geq \delta) \Rightarrow ((\eta = \delta) \hat{} Q)). \\ P \xhookleftarrow{\delta} Q &\ \widehat{=}\ \Box(\neg - P \hat{} (P \wedge \eta < \delta) \Rightarrow Q). \end{aligned}$$

2.2 Syntactic Sugar

In order to interweave the QDDC formulas within the code, it is necessary to define the formulas semantics. Gonnord *et. al* [5] showed that some QDDC operators are non-deterministic and hence have to be removed or substituted in order to be evaluated in runtime. Non-determinism arise from the use of the next evaluation in propositions, $+P$, the existential ($\exists p \cdot D$), and the chop operator ($D_1 \hat{} D_2$). While the first two has to be complete removed since they are completely non-deterministic. While the chop operator is restricted to its wide used deterministic version, in other words, the subintervals are bounded with deterministic occurrences of events.

As with the QDDC notation propositions are used as the underlying driving mechanism. Let $A_P(\mathcal{I})$ be the evaluation of P for the interval $\mathcal{I}$.

P	$A_P(\mathcal{I})$
p	p
$\neg P$	$A_P(\mathcal{I})$
$P_1 \wedge P_2$	$A_{P_1}(\mathcal{I})$ and $A_{P_2}(\mathcal{I})$
$P_1 \vee P_2$	$A_{P_1}(\mathcal{I})$ or $A_{P_2}(\mathcal{I})$
$P_1 \otimes P_2$	$(\neg A_{P_1}(\mathcal{I})$ and $A_{P_2}(\mathcal{I}))$ or $(A_{P_1}(\mathcal{I})$ and $\neg A_{P_2}(\mathcal{I}))$
$P_1 \Rightarrow P_2$	$\neg A_{P_1}(\mathcal{I})$ or $A_{P_2}(\mathcal{I})$
$P_1 \Leftrightarrow P_2$	$A_{P_1}(\mathcal{I}) \Rightarrow A_{P_2}(\mathcal{I})$ and $A_{P_2}(\mathcal{I}) \Rightarrow A_{P_1}(\mathcal{I})$

However, propositions alone are not enough to define the QDDC notation semantics. Therefore, a number of methods over propositions are introduced to aid in QDDC evaluation [5].

Method	Description
`after`(P, b)	Returns true if P was true at the start of the interval (denoted as b). Equivalent to $\lceil P \rceil^{0\frown} true$.
`strict_after`(P, b)	Returns true if P was true at the start of the interval. However, on the start of the interval it returns false. Equivalent to $\lceil true \rceil^{0\frown} \lceil -P \rceil \frown true$.
`always_since`(P, b)	Returns true if P has been constantly true for the interval, starting at clock cycle b. Formally, $\Vert P \lceil$.
`nb_since`(P, b)	Returns the number of occurrences of P. In QDDC equivalent to ΣP.
`age`(P, b)	Returns the number of times P was true from the last time it evaluated to false in the interval.
`first`(P, b)	Returns true on the first occurrence of P.

Given the above methods, the fragment of QDDC that can evaluate to false only as time passes is defined as

G	$A_G(\mathcal{I})$
`begin`(P)	after($A_P(\mathcal{I})$ and `b`)
$\Vert P \rceil$	strict_after(`b`) and pre(always_since($A_P(\mathcal{I})$, `b`))
$\eta \leq c$	nb_since($true$, `b`) $\leq c$
$\Sigma P \leq c$	nb_since($A_P(\mathcal{I})$, `b`) $\leq c$
`age`(P) $\leq c$	age($A_P(\mathcal{I})$, `b`) $\leq c$
$G_1 \wedge G_2$	$A_{G_1}(\mathcal{I})$ and $A_{G_2}(\mathcal{I})$
$G_1 \vee G_2$	$A_{G_1}(\mathcal{I})$ or $A_{G_2}(\mathcal{I})$

Note the introduction of the QDDC operator `age`. The operator is useful in order to provide alternatives to some of QDDC derived operators. For example, let `then` operator to be the concatenation of two subintervals, then $P \xrightarrow{\delta} Q$ is expressed equivalently as,

$$P \xrightarrow{\delta} Q \equiv \text{age}(P) < \delta \text{ then } P \wedge Q.$$

Finally, the full fragment of QDDC that can be evaluated in runtime, is

F	$A_F(\mathcal{I})$
G	$A_G(\mathcal{I})$
`end`(P)	after(`b`) and $A_P(\mathcal{I})$
G then F	A_F(first(not $A_G(\mathcal{I})$
$F_1 \wedge F_2$	$A_{F_1}(\mathcal{I})$ and $A_{F_2}(\mathcal{I})$
$\neg F$	not $A_F(\mathcal{I})$

The `then` operator denotes the deterministic version of chop, which state that the entire interval is satisfied if on the first failure to satisfy the first subexpression, the second subexpression is immediately satisfied.

2.3 Examples

Before commencing with further formalisms, this section provides two simple examples of how systems can be expressed in QDDC formulas.

Guess a number example. Consider the simple guess a number game, where the number of attempts are limited. Whenever the user attempts to guess the number, he or she can either try a number smaller or higher than the target or the target number. The game can easily be specified using QDDC logic, as

$$((\llbracket \text{Less xor Greater} \rrbracket \text{ and } \eta < \delta)^\frown(\eta \leq \delta \Rightarrow \lceil \neg\text{Less and } \neg\text{Greater} \rceil^0))^*.$$

where delta is the number of attempts the user has.

The first subexpression of the chop operator specifies the condition that the user is trying to guess the number within the allowed attempts, while the second subexpression verifies that the user guessed the number within the provided limit. The Kleene closure is used to allow the formula to be used repeated whenever a new game is started.

Now, lets try to specify the same formula using the restricted logic to interweave it with the game code. Due to the constraints placed by the way a program is executed, a small modification is required. The modification is required since whenever the chop starts a new subinterval, the length of the entire interval is lost. Hence while leaving the limit of attempts to be checked programmatically and placing both Less and Greater variables are to true whenever the limit is exceeded to ensure that the chop operator fails, the formula to be integrated inside the code is:

$$(\llbracket \text{Less xor Greater} \rrbracket \text{ then end}(\neg\text{Less and } \neg\text{Greater}))^*.$$

Memory allocation example. Now consider a slightly more useful example. One of the major glitch in developing software in C/C++ is memory allocation. Freeing memory that was not allocated or already freed, or never freeing allocated memory give rise to misbehaviour in software which is very difficult to trigger or notice. Through the use of two simple QDDC formulas integrated with the memory de/allocation methods one can easily check for these problems.

Let $P \lessdot\sim Q$ stand for P must hold one clock cycle before the occurrence of Q. Given that the event of memory allocation is labelled as Alloc and that of memory deallocation labelled as Free, then, the formulas required for the simple memory check program are,

$$\Sigma\text{Alloc} > \Sigma\text{Free} \lessdot\sim \Sigma\text{Free}. \tag{1}$$

$$(\lceil \Sigma\text{Alloc} \ > \Sigma\text{Free} \rceil \frown \lceil \Sigma\text{Alloc} \ = \Sigma\text{Free} \rceil)^*. \tag{2}$$

Formula 1 is used to ensure that under no circumstances the number of freed memory exceeds the number of memory allocated. The second formula describes the behaviour in memory usage, that is, either some memory is still allocated or it is all free. The second subexpression in formula 2 ensures that at the end of program execution all memory has been freed. Defining these formulas for interweaving is straightforward.

$$\Sigma\text{Free} \Rightarrow \Sigma\text{Alloc} \geq \Sigma\text{Free}.$$

$$(\llbracket \Sigma\text{Alloc} > \Sigma\text{Free} \rrbracket \text{ then } \llbracket \Sigma\text{Alloc} = \Sigma\text{Free} \rrbracket)^*.$$

It is important to note that more complex scenarios can be similarly handled using the restricted QDDC for integrating the formulas in the code.

3 Encoding QDDC syntax

Using the syntactics presented in Section 2.2 and Lustre representation for symbolic automata, in this section we present the solution adopted for encoding the restricted QDDC syntax. The solution is adopted from Gonnord *et. al* [5].

Consider it is required to evaluate the QDDC formula $\lceil P \rceil$. The formula is evaluated as `strict_after(`b`)` and `pre(always_since(`$A_P(\mathcal{I})$`, `b`))`. In other words, we need to evaluate two methods because the `pre()` is an operator over QDDC variables that returns the value they contained one clock cycle before.

First, let define the algorithms for the two methods. The `strict_after(`b`)` states that when b is true the method returns false, but subsequently it must return true. Therefore, given tat each different method invoked has a QDDC variable associated with it, for example `after(`p`)` has a variable named `after_p` to store its value history and on the assumption that all variables are initialised to false then,

$$\texttt{strict\_after\_b} \ = \ \texttt{false} \rightarrow (\texttt{pre}(b) \ \vee \ \texttt{pre(strict\_after\_b)}).$$

The arrow sign after `false` is used to indicate that the variable `strict_after_b` is initialised to false. One must also note that the or operator is lazily evaluated left-hand parameter first because on the first clock cycle the `pre()` might not be initialised.

As the `always_since(`P`, `b`)` name suggests the method returns true if the variable parameter has been constantly true from the start of the interval.

$$\texttt{always\_since\_P\_b} \ = \ \texttt{false} \rightarrow (b \vee \texttt{pre(always\_since\_P\_b)}) \ \wedge P.$$

Finally, the variable associated with the QDDC formula, `const_P`, is assigned the value,

```
const_P = strict_after(b) and pre(always_since(P, b)).
```

The use of the capital letter P within the algorithms is to indicate that the parameter value can either be a QDDC variable or an expression as in the case of evaluating ⟦Less xor Greater⟧ in the first example in Section 2.3. The methods in Section 2.2 are all evaluated using the same reasoning as in the above example-driven evaluation. For example, `after(P)` and `age(P, b)` are evaluated as:

```
after(P) = false→(P ∨ pre(after(P))).

nb_since(P, b) = if after(b) and P then
                   0 →pre(nb_since(P, b)) + 1
                 else
                   0 →pre(nb_since(P, b)).

age(P) = if P then
           0 →pre(age(P)) + 1
         else
           0.

first(P, b) = if after(b) then
                P ∧ ¬(strict_after(P)).
```

The basic operators in the algorithms above are the next operator, and the borrowed Lustre operators `pre()` and the initialisation operator (→). The latter consists in a simple initialisation process that assigns the formula variable, example after_P, with the value preceding the operator.

Each variable that requires to use its previous value(s) is required to create additional memory space, the size of the variable data type, in order to store the old values. The process for keeping historical data regarding a variable is outlined in the algorithm below.

Variable assignment

```
  Evaluate proposition
  If keeping history
    Assign variable old value to current value
  Assign variable current value to the value returned in step 1.
```

In the assignment algorithm lines 2 and 3 ensure that if the variable is keeping history of its history, example `pre(pre(P))`, the algorithm is called recursively to update all history memory locations.

Being able to evaluate the restricted QDDC notation, the next operator is used to advance the basic clock by one clock cycle and perform the validation process outline below.

```
validate = ∀ QDDC variables
              read data from the pointed memory location
           ∀ QDDC formulas
              evaluate formula
              if formula fails then
                stop process and report error.
```

4 Framework

The framework presented provides an assertion-like environment, which at user specified time intervals validates the system state with a set of interweaved QDDC formulas.

By leaving the user to perform the interweaving by placing the formulas as assertions within the code, the framework consists of three layers. The fundamental layer provides the validation engine that takes a set of automata representing the formulas and using the syntactic presented in Section 2.2, evaluates their satisfaction. To simplify the use of the engine, another layer is added on top of the engine. This layer allows the user to pass a well-formed QDDC formula and converts it into a symbolic automata, which is then fed to the underlying layer whenever the user requests the system to check its consistency. Finally in order to simplify the framework use and to abstract the user from the underlying system, an interface layer is provided.

Engine. The engine layer is responsible for validating the system with the formulas provided, as described in Section 3. The layer also provides a mechanism to report violations to the user because a formula might return false but will still be valid, like in the case of a `then` statement.

Conversion is achieved through the use of a parser, which checks for well-formed formula by analysing the string representing the formula with the grammar defined in Section 2.2. During the formula checking the parser also attempts to build a symbolic automaton for its representation. It is important to note out that since the conversion is performed at runtime, the symbolic automaton generated is not optimised but rather it consists of small automata concatenated together according to the formula.

5 Related work

The field of validation is gaining in importance, however, not a lot of work has been performed because the main focus is still on verification. Two of the major projects in the use of state variables for checking the system correctness are Bandera [1] and Temporal Rover [2, 4, 3].

The Bandera project is dedicated to the Java programming language and extracts a system model from the source code. The model extracted is minimised

through the use of abstraction and slicing techniques. The project also provides a visualisation tool for checking the model for consistency, and allows the user to place behavioural constraints and check them through a simulated execution.

Bandera use of constraints over a model of the system is similar to our work. That is, both works use a description of the expected behaviour to check the system correctness. However, the main difference is that Bandera checks are performed over an abstracted model of the system that might not fit well in the system. While our framework performs the checks on the real system and during its execution. An advantage the Bandera project has over our solution is that the user is not required to be knowledgeable of how systems can be described formally. Nevertheless, our solution provides an environment that allows errors to be captured following the destinated user feedback rather than using a preselected scenarios that the developer team thinks that might occur.

A major validation project is Temporal Rover by Time-Rover. Temporal Rover provides an assertion-based environment for the integrating specifications defined in temporal logic into programs [2, 4, 3]. The project consists in a precompiler that produces a file similar to the original with tangling code, representing the assertions and the validation engine.

Temporal Rover is far more powerful than the solution presented here. One of the advantages over our solution is that the system handles non-deterministic scenarios by creating instances for all possible solutions. Nevertheless, our solution lacking in non-determinism handling provides the user with a lightweight environment that reports errors as soon as they occur.

Another project is the integration of temporal assertions into a parallel debugger [8]. The parallel debugger project performs the validation by making use of a macrostep debugger, which while the program is being executed the assertions are checked for their validity. It is difficult to compare the project with our solution since there is lack of technical detail regarding the underlying work.

6 Conclusion and Future work

A major trouble in software development is verifying the system correctness. The concept of validation provides a solution for checking correctness during the execution. Through validation it is also possible to trigger errors before they propagate and report their occurrence in more depth than it is possible with traditional error handling.

Using temporal logic formulas for defining the system specifications and allowing their integration into the system source code provides a robust environment for providing more reliable systems. The framework presented in this paper provides an environment for performing synchronous monitoring over system execution.

The grammar provided in Section 2.2 is enough expressive to handle the majority of QDDC formulas. Nonetheless, there are some commonly used QDDC expressions that require to be expressed in equivalent expressions. Since the

framework does not handle optimisations and it is an inconvenience for the user, the framework can be enhanced to support common QDDC expressions directly.

One of the framework drawback is the that formulas are converted at runtime since they are manually interweaved. Using an approach similar to that of aspect-oriented programming [7, 11, 10], where the formulas are supplied through a separate file and their observational interval defined through markups in the system code, the formulas can be converted through a pre-compiler which also handles the generation of tangled code.

Finally, another enhancement is to allow the validation to be performed asynchronously. This is fruitful in situations where immediate error reporting and handling is not critical and also in situations where time response during testing is important, especially if the framework is disabled in the system final version.

References

1. James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Păsăreanu, Robby, and Hongjun Zheng. Bandera: extracting finite-state models from java source code. In *International Conference on Software Engineering*, 2000.
2. Doron Drusinsky. The temporal rover and the ATG rover. In *SPIN 2000*, 2000.
3. Doron Drusinsky and J.L. Fobes. Executable specifications: Language and applications. *CrossTalk - The Journal of Defense Software Engineering*, 17(9):15–18, September 2004.
4. Doron Drusinsky and Klaus Havelund. Execution-based model checking of interrupt-based systems. In *Workshop on Model Checking for Dependable Software-Intensive Systems. Affiliated with DSN'03, The International Conference on Dependable Systems and Networks*, pages 22–25, 2003.
5. L. Gonnord, N. Halbwachs, and P. Raymond. From discrete duration calculus to symbolic automata. In *3rd International Workshop on Synchronous Languages, Applications, and Programs, SLAP'04*, Barcelona, Spain, March 2004.
6. N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.
7. Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.
8. József Kovács, Gábor Kusper, Róbert Lovas, and Wolgang Schreiner. Integrating temporal assertions into a parallel debugger. In *Proceedings of the 8th International Euro-Par Conference*, pages 113–120, 2002.
9. P. Pandya. Specifying and deciding quantified discrete-time duration calculus formulae using DCVALID. Technical Report TCS00-PKP-1, Tata Institute of Fundamental Research, 2000.
10. Mario Schüpany, Christa Schwanninger, and Egon Wuchner. Aspect-oriented programming for .NET. In *First AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (AOSD-2002)*, March 2002.
11. Olaf Spinczyk, Daniel Lohmann, and Matthias Urban. AspectC++: an AOP extension for C++. *Software Developer's Journal*, pages 68–76, May 2005.