Runtime Validation Using Interval Temporal Logic

Karlston D'Emanuele
kema001@um.edu.mt
Dept. of Computer Science and AI
University of Malta

Gordon Pace
gordon.pace@um.edu.mt
Dept. of Computer Science and AI
University of Malta

ABSTRACT

Formal specifications are one of the design choices in reactive and/or real-time systems as a number of notations exist to formally define parts of the system. However, defining the system formally is not enough to guarantee correctness thus the specifications are used as execution monitors over the system. A number of projects are around that provides a framework to define execution monitors in Interval Temporal Logic (ITL), such as Temporal-Rover [3], EAGLE Flier [1], and D^3CA [2] framework.

This paper briefly describes the D^3CA framework, consisting in the adaptation of Quantified Discrete-Time Duration Calculus [7] to monitoring assertions. The D^3CA framework uses the synchronous data-flow programming language Lustre as a generic platform for defining the notation. Additionally, Lustre endows the framework with the ability to predetermine the space and time requirements of the monitoring system. After defining the notation framework the second part of the paper presents two case studies - a mine pump and an answering machine. The case studies illustrate the power endowed by using ITL observers in a reactive or event-driven system.

Categories and Subject Descriptors

[**Formal Methods**]: Validation

General Terms

Formal validation, duration calculus, QDDC, D^3CA , interval temporal logic.

1. INTRODUCTION

The question "Does this program do what it is supposed to do?" and "Will the program work under any environment changes?" are frequently asked when developing a software. A number of techniques have been proposed and adopted during the years, one of which is formal validation.

The major validation tools around concentrate on temporal logics as they provide a means for time measurement. The temporal logic branch of Interval Temporal Logic (ITL) provide means to measure correctness in intervals, which facilitates the scoping of tests and reduce the total impact of the monitors on the system.

We illustrate this by building a basic framework in which the user can, alongside the program, specify properties using interval temporal logic which are automatically woven as monitors into the source code, thus producing the single monitored application. The monitored application at certain intervals checks whether the application state is valid according to the mathematical model, enabling runtime validation of ITL properties.

In this paper we concentrate on showing how the framework can be used in a normal application using two simulated environments.

The rest of the paper is organised as follows: the next section briefly describes some of the validation tools around. Section 3 outlines the syntax and semantics of the interval temporal logic notation used in the paper. In section 4 we describe a framework for the Interval Temporal Logic (ITL) monitors generation and weaving. Finally we conclude by presenting two scenarios where the framework was applied.

2. VALIDATION

The size and complexity of software developed today result in debugging and testing not being sufficiently effective. Formal methods go some way towards tackling this problem. Using model-checking, one test all execution paths of a system to be checked for correctness. Nevertheless, model-checking is expensive to run and does not scale up, even when systems are reduced via abstraction. Validation is a light-weighed formal method, which checks the system correctness for a single path of execution. The path verification is performed by checking at runtime that the formal specifications weaved into the system source code constantly hold. A number of projects have been undertaken in order to find a suitable validation technique for different logics and scenarios. Some projects are Temporal Rover [3], Java-MaC [6], EAGLE [1] and RTMAssertions [8].

Temporal Rover. is a proprietary validation tool by Time-Rover. It integrates properties specified in Linear Temporal Logic (LTL) and Metric Temporal Logic (MTL) into an annotated source code [3]. The integration is performed by a pre-compiler. The formal properties are checked for consistency on every cycle, determined by a call to a method "assign". On a false evaluation of a property an exception is raised, which the system handles in order to return to a safe state. One of the aims of Temporal Rover is to validate embedded system, which are limited in resources. Hence, the system provides another tool, DBRover, which allows properties to be checked remotely on a host PC.

Java-MaC. is a prototype implementation of a more generic architecture named Modelling and Checking (MaC). The MaC architecture consists of three modules, a filter, event recogniser and run-time checker. The properties used by the filter and event recogniser are written in Primitive Event Definition Language (PEDL), which provides a means of defining the formal specifications in relation to the implementation level. When an event (state transition) is identified by the event recogniser, the new state information is sent to the run-time checker, whose properties are written in Meta Event Definition Language (MEDL). The run-time checker in MaC is executed remotely and the information is transmitted using TCP sockets, leading to less strain on the monitored system resources.

Eagle. framework distinguishes from the other validation projects mentioned in this report. EAGLE performs validation over an execution trace and on a state-by-state basis that frees the monitoring system from using additional memory for storing the trace. The framework provides its own logic, named EAGLE, which is enough powerful to allow other logic notation to be specified in the EAGLE logic. A disadvantage in EAGLE is that the type of properties, that is whether it is a liveness or safety property, has to be explicitly specified.

RTMAssertions. uses Aspect-Oriented Programming (AOP) to separate between LTL properties and the program implementation. The LTL properties are first converted into an Abstract Syntax Tree (AST) which together with the RTM framework is weaved into the code by the aspect weaver. The LTL properties are evaluated by subdividing the LTL formula into a number of atomics, the tree leaves, which are evaluated first. Then by following the nodes path to the root, the actual formula result, is calculated recursively.

This section overviewed some of the projects in the validation field. Next is the logic notation used in the paper

3. DISCRETE AND DETERMINISTIC DURATION CALCULUS

Discrete and deterministic Duration Calculus is a descendant of Quantified Discrete-Time Duration Calculus by Pandya [7]. The logic assumes that the future is determined by the past events thus it can be easily implemented using any programming language.

3.1 Syntax and Semantics

The most basic expression in discrete and deterministic Duration Calculus are propositions, referred to as state variables. Let P be a state variable than its syntax is

$$P \ ::= \ \mathbf{false} \mid \mathbf{true} \mid p \mid P \ op \ P \mid \neg P$$

where p is a propositional variable and

$op \in \{\wedge, \vee, \Rightarrow, \Leftrightarrow, \otimes \text{ (xor)}\}$.

On the assumption that system states have finite variability. Let σ be a non-empty sequence of state variables

$$\sigma =_{df} (\text{state variable} \mapsto \mathbb{B})^{+}$$

where, the length of the sequence is given by $\#(\sigma)$.

Discrete and deterministic Duration Calculus is an Interval Temporal Logic, in other words the expressions defined using this notation require an interval. Defining first the concept of time as

$$\mathbb{T} =_{df} \mathbb{N}$$

Then an interval is defined as

$$\mathit{Intv} =_{df} \{(b, e) \mid b, e \in \mathbb{T}\}$$

Let $\sigma_{\mathcal{I}} \models D$ mean that the finite sequence σ satisfies the duration formula D within the interval, $\mathcal{I} \in \mathit{Intv}.$[1].

$$\begin{array}{lll}
\sigma_i \models P & \textbf{iff} & i \in \mathbb{T} \wedge P(i) = \mathbf{true} \\
\sigma_i \models P_1 \leftarrowtail P_2 & \textbf{iff} & i \in \mathbb{T} \wedge \sigma_{i-1} \models P_1 \wedge \sigma_i \models P_2 \\
\sigma_{\mathcal{I}} \models \text{begin}(P) & \textbf{iff} & \sigma_{\mathcal{I}_b} \models P \\
\sigma_{\mathcal{I}} \models \text{end}(P) & \textbf{iff} & \sigma_{\mathcal{I}_e} \models P \\
\sigma_{\mathcal{I}} \models \llbracket P \rceil & \textbf{iff} & \forall i \in \mathcal{I} \cdot i < \mathcal{I}_e \wedge \sigma_i \models P \\
\sigma_{\mathcal{I}} \models \llbracket P \rrbracket & \textbf{iff} & \forall i \in \mathcal{I} \cdot \sigma_i \models P \\
\sigma_{\mathcal{I}} \models \eta \leqslant c & \textbf{iff} & (\#(\sigma) = \mathcal{I}_e - \mathcal{I}_b) \leqslant n \\
\sigma_{\mathcal{I}} \models \Sigma(P) \leqslant c & =_{df} & \sum_{\mathcal{I}_b}^{\mathcal{I}_e - 1} P(i) \\
\sigma_{\mathcal{I}} \models \text{age}(P) \leqslant c & =_{df} & \text{The state variable } P \text{ is true for} \\
 & & \text{the last part of the interval and} \\
 & & \text{it is not constantly true for} \\
 & & \text{more than } c \text{ time units.} \\
\sigma_{\mathcal{I}} \models D_1 \text{ then } D_2 & \textbf{iff} & \exists m \in \mathcal{I} \cdot \mathcal{I}_b \leqslant m \leqslant \mathcal{I}_e \wedge \\
 & & \sigma_{[\mathcal{I}_b, m-1]} \models D_1 \wedge \\
 & & \sigma_{[\mathcal{I}_b, m]} \models \neg D_1 \wedge \\
 & & \sigma_{[m, \mathcal{I}_e]} \models D_2 \\
\sigma_{\mathcal{I}} \models D_1 \overset{\delta}{\hookleftarrow} D_2 & \textbf{iff} & \text{The duration formula } D_2 \text{ must} \\
 & & \text{be true for the first } \delta \text{ time units} \\
 & & \text{that formula } D_1 \text{ is true.} \\
\sigma_{\mathcal{I}} \models D^{*} & =_{df} & \exists n \in \mathbb{N} \cdot D_1 \text{ then } D_2 \ldots \text{ then } D_n
\end{array}$$

The next section describes the design of a system that translates formulae written using the notation introduced into run-time monitors for .NET systems.

4. THE TOOL: D^{3}CA

D^{3}CA is a prototype implementation of a validation engine for properties defined in deterministic QDDC. The programming language used for implementation is C#.

D^{3}CA consists of two modules: the validation engine and the weaving of validation with the system. The validation engine grabs a collection of properties and using a simulated Lustre environment checks the each property with the current state. The weaving process can be performed using Aspect Oriented programming (AOP) tools. Nevertheless, for better understanding of the communication process between the validation engine and the monitored system, a weaver is discussed.

Figure 1 illustrates the architecture of a D^{3}CA monitored system.

[1]Due to lack of space, some operators are defined informally. For full formal definitions, refer to [2].

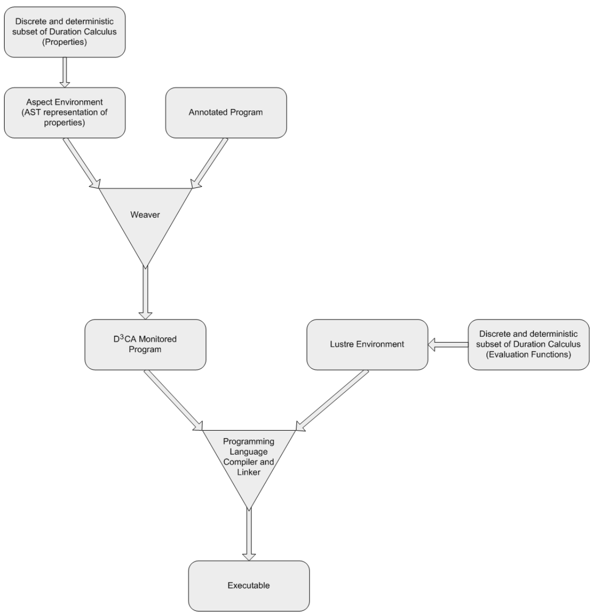

Figure 1: D^{3}CA Architecture Overview.

4.1 Lustre

Lustre is a dataflow synchronous language [4], that is it applies the synchrony hypothesis over execution of code. The hypothesis states that all computations are performed atomically and take no time to execute. Lustre implements a restricted subset of dataflow primitives and structures, that makes it applicable to reactive and real-time systems.

Lustre provides three data types: integer, real and boolean. The variables used inside Lustre must be explicitly declared and initialised to the appropriate data type. The expression variables have the form "$X = E$" where X is associated with the expression E. Therefore, E is taken to be the only possible value for X.

The Lustre operators set consists in the basic data type operators, conditional branching and two special operators, `pre` and "followed by" ($\rightarrow$). The `pre` operator enables the access to a variable history. While the "followed by" operator is used to concatenate two streams together, where at time zero the second stream is equal to the first value of the first stream.

In D^{3}CA the Lustre environment is simulated, that is, rather than using a Lustre compiler the data types are implemented as classes. Using boolean variables with deterministic QDDC leads the value of **false** to be ambiguous. The Lustre environment is extended with 3-valued logic data type, which extends the boolean data type with the additional value of **indeterminate** for expressions whose truth value has yet to be determined.

The use of a Lustre environment in D^{3}CA allows the space and memory requirements of the specified properties to be predetermined. Hence, providing a place for optimisation and a knowledge on the side-effects of the validation engine on the monitored system.

4.2 Weaver

The weaver module consists in transforming annotated control instructions to the actual validation code. The annotated control instructions can be of five types:

1. $\{$*validation_engine*: **bind** *variable_name variable_value*$\}$
2. $\{$*validation_engine*: **unbind** *variable_name*$\}$
3. $\{$*validation_engine*: **start** *assertion_name*$\}$
4. $\{$*validation_engine*: **stop** *assertion_name*$\}$
5. $\{$*validation_engine*: **synchronise** $\mathbb{B}\}$

In validation the state variables are mapped to the system variables. In software, variables are typically placed in their local scope, hence, they cannot be accessed from outside code. This raises a problem with interval temporal logic assertions that have crosscutting semantics. To surmount the problem the two variable un/binding annotations are provided. A variable can be bound to either a system variable while the variable is in scope or to a numeric value. The unbind annotation instructs the weaver that the value of the variable must be kept constant.

An important characteristic of the D^{3}CA is that it uses interval temporal logic. That is, the properties are expected to hold for intervals. To control the interval of an assertion the start and stop annotations are provided. When checking a stopped assertion, the **indeterminate** value is considered to be false as the property has not been full satisfied during the interval.

The last and most important annotation is "synchronise". This annotation instructs the weaver that the properties has to be checked for consistency. Nevertheless, before performing the runtime checking the variables are updated. The update also includes the reassignment of constant variables as to reflect their values according to the current state, Figure 2.

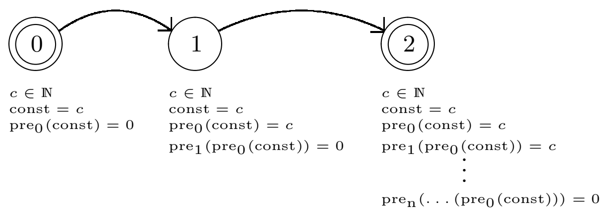

Figure 2: Lustre constant to system state relation

The "synchronise" annotation takes a boolean value. This value instructs the synchronisation method whether to abandon execution on trapping an error. However, independently of the value passed the runtime checker still reports the errors that has been encountered.

4.3 Monitoring

The monitoring process consists in two interdependent tasks. The core task is the evaluation of the properties, which is performed using the Lustre environment. The direct use of the evaluation process is cumbersome. Hence, the validation task is used to abstract the evaluation process and perform the repetitive task of evaluating each property.

4.3.1 Initialisation

The initialisation process consists in converting the monitoring properties from the mathematical notation to a collection Abstract Syntax Trees (ASTs). The conversion process is performed using a parser as the mathematical notation provides a suitable grammar representation.

The AST data structure is adopted since it provides a suitable visualisation of how complex properties are decomposed into smaller properties. Evaluating the smaller properties is assumed to be simpler, hence, by evaluating the lower nodes in the tree facilitates the process of obtaining the satisfiability value.

4.3.2 Evaluation

The evaluation process is a bottom-up traversal of the AST structure. The simplest nodes to evaluate are the leaf nodes that are either propositions or numeric variables. After mapping the system state value to the leaf nodes, the nodes at a higher level can be evaluated. The evaluation of the non-leaf nodes consists in calling the function related to the operator[2]. The property satisfiability value is then determined by the value obtained by evaluating the root node.

EVALUATE(*Symbolic Automaton*)

for each node starting from the leaf nodes
    do expression variable ← evaluation node expression
Symbolic automaton validity result ← root node value

4.3.3 Validation

The evaluation process described above is encapsulated in the validation process. The property ASTs are evaluated bottom-up, hence, the state variables has to be updated before the starting the evaluation. In algorithm VALIDATE line 1 suspends the system execution to perform the validation process. Therefore, ensuring that the system state is not corrupted during the validation process. When the system is stopped, line 2 updates the state variables to reflect the system variables. That is, performs a transition from the current state to the new state.

The actual validation process consists in performing the evaluation process on the collection of ASTs, lines 3–6. Each AST is checked for validity and one of the 3 logic states is returned. When a property is violated the system reports the error together with a the property trace. Note that, the monitoring system uses symbolic automata to represent the system, hence, it is not possible to depict the entire state according to the execution path, without keeping a history.

VALIDATE

1 Stop system execution. // *R*equired for variable integrity
2 Update non-expression variables
3 **for** each symbolic automaton
4   **do** Valid ← Evaluate(Symbolic automaton)
5   **if** Valid == **false**
6   **then** Error(Symbolic automaton)
7 Resume system execution. // *O*n the assumption that the system was not aborted due to errors.

[2]Refer to [2] for the actual deterministic QDDC execution semantics.

4.4 Design review

D^3CA implements a solution for monitoring systems using interval temporal logic. The validation process is performed on a Lustre environment, which allows memory and space requirements to be predetermined.

Properties are cleaner if written in mathematical notation. The validation mechanism provided by D^3CA includes a parser that on initialisation of a property the mathematical notation is converted into symbolic automata. The symbolic automata are then stored in an Abstract Syntax Tree data structure as it provides a suitable representation for the evaluation process.

The architecture of the monitored program during runtime is illustrated in Figure 3.

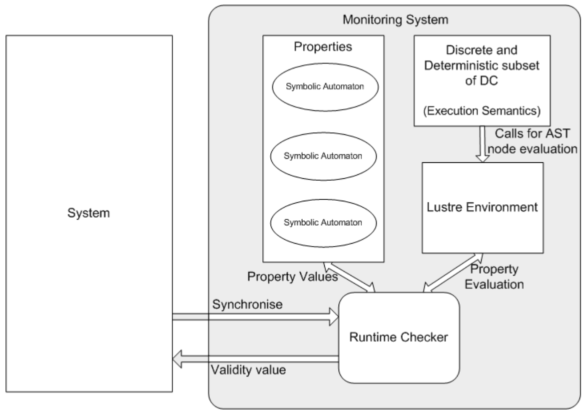

Figure 3: System composition diagram

5. CASE STUDIES

The framework is applied to two simulators, a mine pump and an answering machine.

5.1 Mine Pump

The first case study consists in the adaptation of a commonly used example in Duration Calculus literature [7, 5]. The case study consists in simulating the behaviour of a water extraction pump employed in a mine to lower the level of the water collected in a sump.

A mine has a water (H_2O) and methane (CH_4) leakage. The water leaked is collected in a sump which is monitored by two sensors signalling when the water level is high or dangerous. When the water level is high a pump is started to pump water out. Nevertheless, the pump cannot be started is the methane level is high. Using the notation introduced earlier the property for starting the pump can be defined as,

$$(\llbracket LowH_2O \rrbracket \text{ then } (\text{age}(HighH_2O \wedge \neg HighCH_4) <= \delta \text{ then } \llbracket PumpOn \rrbracket))^*$$

where, δ is the time required for the pump to start operating.

When the pump is operating it takes ϵ time to lower the water level to acceptable level. This property is defined as

$$(\llbracket PumpOn \rrbracket \wedge \eta \leqslant \epsilon \text{ then begin}(LowLowH_2O))^*$$

The last property related to the pump operation is to check that when the water level is low or the methane level

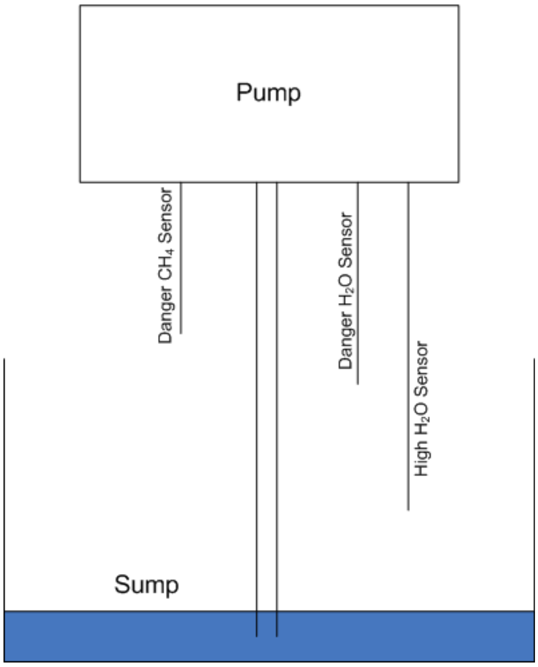

Figure 4: Mine Pump Environment

has rose to the level where it is dangerous to operate machines.

$$(\text{age}(LowH_2O \vee HighCH_4) \leqslant \delta \text{ then } \lceil PumpOff \rceil)^*$$

The discrete and deterministic Duration Calculus notation allows environment assumptions to be defined. The water level sensors are expected to report the water level in ascending or descending order. That is the water cannot go to dangerous level before it reaches the high level. The first sensor assumption is that the water is at high-level for some time before it reaches the dangerous level.

$$\lceil HighH_2O \rceil \overset{\omega}{\longleftarrow} \lceil \neg DangerousH_2O \rceil$$

We can also say that the water is in dangerous levels if it is also at the high level.

$$\lceil DangerousH_2O \Rightarrow HighH_2O \rceil$$

The other environment assumptions are related to the methane release. For the mine operations not to be interrupted on frequent intervals, the methane release occurs only at least ζ time after the last methane release. More formally,

$$(\text{age}(\neg HighCH_4) \leqslant \zeta \text{ then } \mathbf{true}) \leftarrow HighCH_4$$

When deploying the mine pump an assumption is made that methane releases are of short burst thus allowing the pump to be operated.

$$\text{age}(HighCH_4) \leqslant \kappa$$

where κ is the maximum time a methane release can take for the pump to be operatable.

Finally we equip the mine pump with an alarm system to notify the workers that there is possibility of danger. The alarm will go off when either the water reaches the dangerous level or there is a high level of methane in the mine.

$$(\text{age}(DangerousH_2O) \leqslant \delta \text{ then } \lceil AlarmOn \rceil)^*$$

$$(\text{age}(HighCH_4) \leqslant \delta \text{ then } \lceil AlarmOn \rceil)^*$$

$$(\text{age}(\neg DangerousH_2O \wedge \neg HighCH_4) \leqslant \delta \text{ then } \lceil AlarmOff \rceil)^*$$

5.2 Mine Pump Scenario

The scenario presented here consists in simulating a long methane release, which violates the methane release assumption.

The constant variables are initialised as follows: $\delta = 2$, $\epsilon = 7$, $\kappa = 2$, $\omega = 17$, $\zeta = 25$.

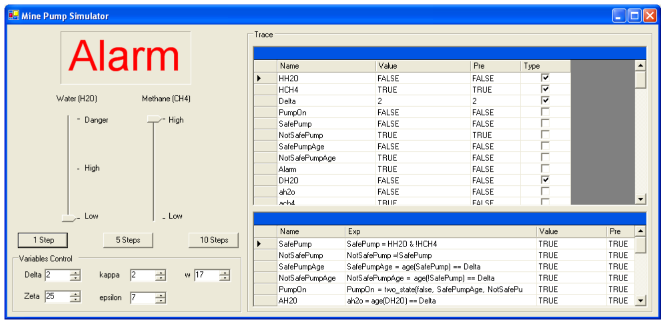

Figure 5: Mine Pump Screen Shot

The simulator has two controls, one for the water level and another for the methane level, which allow the simulation of environment changes. The simulation interface also provide three buttons one to signal a clock tick, another to simulate 5 clock ticks using the same system state and the last button to simulate 10 clock ticks.

To simulate the scenario where a methane release breaks the assumption

$$\text{age}(HighCH_4) \leqslant \kappa$$

the methane controller is set on the high mark. Then since $\kappa = 2$ the 5 clock ticks button is pressed. On the third clock tick the assumption breaks and an assumption violated error is reported back to the user. In order to track back the origin of the error the user is presented back with the assertion that failed and the system state values. In our case the system state is ($\text{age}(HighCH_4) = 3$, $\eta = 3$, $\kappa = 3$, $\text{age}(HighCH_4) \leqslant \kappa = \mathbf{false}$). When analysing the system state it is immediately clear that that the methane release was longer than what is expected.

5.3 Answering Machine

This section describes a simulated answering machine on which the D^{3}CA is applied. Figure 6 illustrates the answering machine states and the possible transition between the states.

The answering machine depicted above has four different interval measurements. The time spent in the “idle” and “receiver up” states cannot be determined. Therefore, when specifying the system in Duration Calculus the intervals can be considered as open. While the “ringing” and “recording” intervals that are fixed in length. The ringing interval is set to 10 rings, therefore, the answering machine must start playing the recorded message only if in the meantime the receiver has not been pulled up. The message “recording” interval allows the callee to leave a message for about 3 minutes and then the line is dropped. The last interval measure is determined by the length of the message recorded

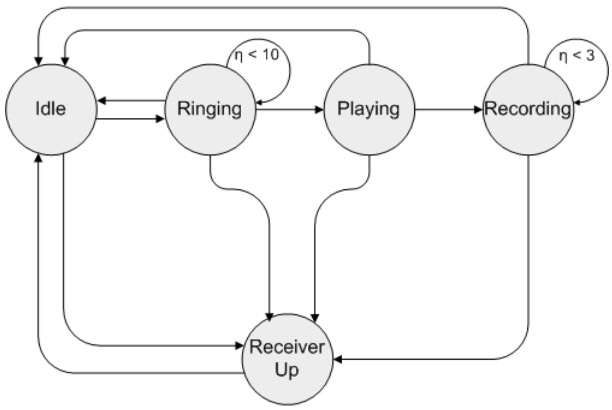

Figure 6: Answering Machine State Diagram

by the answering machine owner. This interval measurement is applied to the "playing" state. Therefore, the specification of the system is

$$
\begin{aligned}
&((\lceil \mathrm{idle} \rceil \text{ then} \\
&\quad \mathrm{age}(\mathrm{ringing}) \leqslant 10 \text{ then} \\
&\qquad \lceil \mathrm{playing} \rceil \text{ then } \mathrm{age}(\mathrm{recording}) \leqslant 3)^* \vee \\
&\qquad\quad \mathrm{end}(\mathrm{receiver\ up}))^*
\end{aligned}
$$

It can be noted that although the intervals are measured using different units, the specifications consider the length of interval independently of the measuring units. In the case of D^{3}CA and of this particular case study the different measuring units are handled by the placing of "synchronise" annotation in the system implementations.

The "idle" state reflects that the answering machine is doing no other operation. Hence, when the receiver is lifted up the answering machine is expected to be idle. The "receiver up" state in Figure 6 is included to show that the idle state of the answering machine and the idle state when the receiver is up are different.

The "idle" state property can be specified in terms of the answering machine states as

$$\mathrm{idle} = \neg\mathrm{ringing} \wedge \neg\mathrm{playing} \wedge \neg\mathrm{recording}.$$

The simplest assumption properties to verify are related to the transition from one state to the next, where the next state has only one entrance path.

$$\mathrm{idle} \lessdot\!\!- \mathrm{ringing}$$

$$\mathrm{age}(\mathrm{ringing}) \leqslant 9 \lessdot\!\!- \mathrm{playing}$$

$$\mathrm{playing} \lessdot\!\!- \mathrm{recording}$$

The answering machine under design assumes that when the receiver is up then no other calls can come in. That is, there is no multiplexing between different lines. When the receiver is up, the answering machine should be idle.

$$\mathrm{end}(\mathrm{receiver\ up}) \implies \mathrm{end}(\mathrm{idle})$$

The answering machine is then simulated in relation to the above operation properties and assumption properties. The properties specified above were able to trigger all the errors inserted in the simulation. Hence, they forced the behaviour of the system to the specifications.

5.4 Answering Machine Scenario

The mine pump scenario showed how the framework can trigger violations to state properties. In this scenario we show that the framework is also capable of detecting violations to state transitions. This is illustrated by a small simulation that violates the transition from ringing to playing the recorded message.

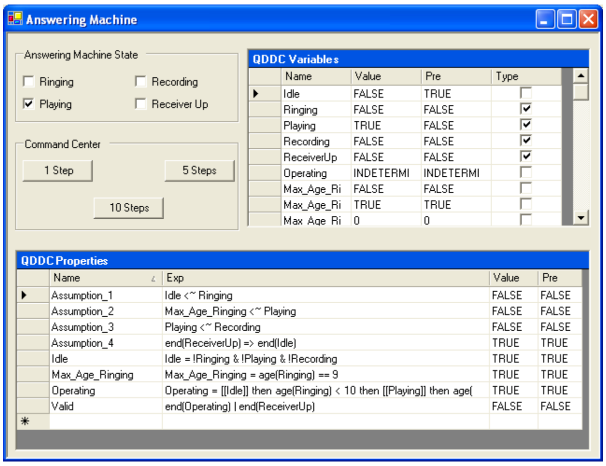

Figure 7: Answering Machine Screen Shot

The answering machine simulator has a number of control buttons, one for every state except for "idle". The state "idle" is represented by unmarking all the other controls. When the system starts the state is immediately set to "idle". A number of steps are performed to leaving the state as "idle". Then the phone starts ringing, so the "ringing" control is marked and a number of clock ticks are simulated. However, after the 5 ringing tone the "playing" state is marked. This violates the transition assumption

$$\mathrm{age}(\mathrm{ringing}) \leqslant 9 \lessdot\!\!- \mathrm{playing}$$

as only 5 ringing tones has been performed. The simulator immediately reports the error, figure 8. The error shows the property violated together with the values of each subexpression.

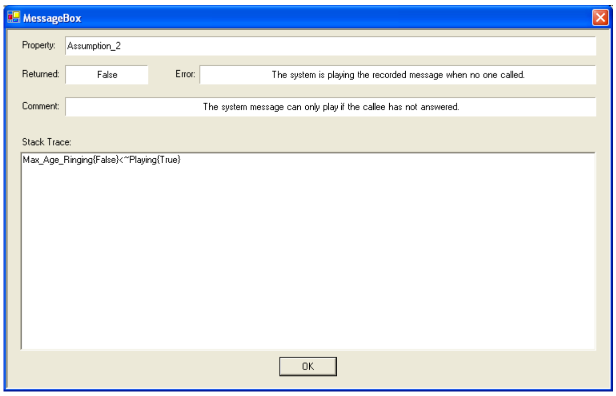

Figure 8: Answering Machine Error Report

From the two simple scenarios presented in this section it was shown that the framework has the potential to define different properties of the system. The framework hides all the complexities related to notation interpretation in programming language and in defining the monitoring system.

The benefit of using Interval Temporal Logic monitors is the increase in reliability of the system without the need to overwhelm the system with point-logic assertions.

6. CONCLUSION

The use of validation for testing software correctness is a well applied concept, and different scenarios lead to the use of different validation approaches. In this paper we showed how Interval Temporal Logic validation can be integrated with normal applications and in real-life scenarios. The integration is obtained through the use of a framework that allows to predetermine the space and time requirements for computing state satisfiability. The framework presented in this paper simplifies the migration from one scenario to another by freeing the validation part from environment and platform dependencies.

7. REFERENCES

[1] H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Eagle monitors by collecting facts and generating obligations. Technical Report Pre-Print CSPP-26, University of Manchester, Department of Computer Science, University of Manchester, October 2003.

[2] K. D'Emanuele. Runtime monitoring of duration calculus assertions for real-time applications. Master's thesis, Computer Science and A.I. Department, University of Malta, 2006. To be submitted.

[3] D. Drusinsky. The temporal rover and the ATG rover. In *SPIN*, pages 323–330, 2000.

[4] N. Halbwachs. Synchronous programming of reactive systems. In *Computer Aided Verification*, pages 1–16, 1998.

[5] M. Joseph, editor. *Real-time systems specification, verification and analysis*. Tata Research Development and Design Centre, Jume 2001.

[6] M. Kim, S. Kannan, I. Lee, O. Sokolsky, and M. Viswanathan. Java-MaC: a run-time assurance tool for Java prgrams. In *1st Workshop on Runtime Verification (RV'01), volume 55 of ENTCS*, 2001.

[7] P. Pandya. Specifying and deciding quantified discrete-time duration calculus formulae using DCVALID. Technical Report TCS00-PKP-1, Tata Institute of Fundamental Research, 2000.

[8] S. Thaker. Runtime monitoring temporal property specification through code assertions. Department of Computer Science, University of Texas at Austin, 2005.