Appeared in Journal of System Architecture, Elsevier B.V., volume 51, number 5, p. 331--345, 2005

Run-time Analysis of Time-critical Systems

Shikun Zhou [a] Hussein Zedan [b,1] Antonio Cau [b,*]

[a] *Electronic & Computer Engineering Department, Faculty of Technology, University of Portsmouth, Anglesea Road, Portsmouth PO1 3DJ, England*

[b] *Software Technology Research Laboratory, School of Computing, Faculty of Computing Sciences & Engineering, De Montfort University, The Gateway, Leicester LE1 9BH, England*

Abstract

AnaTempura is a tool based on Interval Temporal Logic (ITL). It is used to analyse time-critical systems at run-time. It validates code (implementation) against a formal specification. In this paper, we will describe a tool, AnaTempura and its supporting logic, ITL. A small but illustrative case study is presented.

Key words: Interval Temporal Logic, Real-time System, Run-time Analysis & Verification, Timing Diagrams

1 Introduction

The engineering of time-critical systems poses significant challenges to their *'correct'* specification, design and development. In such systems, the time, at which each input is processed or output is produced, is critical. Most of the time-critical computer systems are special-purpose and complex, require

* Software Technology Research Laboratory, School of Computing, Faculty of Computing Sciences & Engineering, De Montfort University, The Gateway House, Tel: ++44 (0)116 257 7937, Fax: ++44 (0)116 257 7936, The Gateway, Leicester LE1 9BH, England

Email addresses: Shikun.Zhou@port.ac.uk (Shikun Zhou), zedan@dmu.ac.uk (Hussein Zedan), cau@dmu.ac.uk (Antonio Cau).

[1] The author wishes to acknowledge the funding received from the U.K. Engineering and Physical Sciences Research Council (EPSRC) through the Research Grants GR/M/02583 and GR/M/32474

a high degree of fault tolerance, and are typically embedded in a larger system [1]. Avionics, robotics and process control are all examples of time-critical computing [1]. It has been recognised that the use of formal methods, in the development of such systems, is fundamental if "correctness" is to be assured. The use of formal approaches increases our understanding of a system by revealing inconsistencies, ambiguities, and incompleteness that might otherwise go undetected.

An important aspect in the development of a time-critical system is how to cope with its "evolution". The evolution of a software system could be due to changes in the original requirements, adopting a different hardware platform or to improve its efficiency. Rapid development causes continuous changes in the software life-cycle. These kind of changes bring even more troubles to developers in time-critical computing than in other applications. Because of its complexity, the likelihood of subtle errors is much greater and some of these errors could have catastrophic consequences such as loss of life, money, time or damage to the environment. Hence, we have developed a sound and practical approach to handle changes and analyse their effect in the development of time-critical applications [2,3].

This paper aims to present its supporting tool, AnaTempura, for handling change in time-critical system development and maintenance. Using this tool, we can validate and analyse system's behaviours of interest at run-time. The validation and analysis are performed within a *single* logical framework using Interval Temporal Logic (ITL [4,5]). The tool is based on the mature executable subset of ITL, Tempura [6]. The very nature of the execution mechanism of Tempura and the properties of its underlying logic, distinguish it from other temporal languages. Behavioral properties such as safety and timeliness, are easily expressed in ITL as theorems which can be validated and tested. These properties include invariants that may be required to be valid before and after the change is made. This facilitates the modular design and maintenance of complex systems that have both sequential and parallel sub-components. The approach allows us to validate various interesting properties. By compositional approach, we include any method by which properties of a system can be inferred from properties of its components, without additional information about the internal structure of those components.

The technique presented here is language independent. The source code could be in any language (e.g. C, C++, or Ada).

Methods and their supporting tools for the formal verification and analysis of system exist. For example, HyTech [7] is a symbolic model checker for linear hybrid automata [8], a subclass of hybrid automata that can be analysed automatically by computing with polyhedral state sets. A key feature of HyTech is its ability to perform parametric analysis, i.e. to determine the values of design parameters for which a linear hybrid automaton satisfies a temporal-logic requirement. HyTech has been most successful when applied to systems that involve an intricate interplay between discrete and continuous dynamics.

Mok et. al (within the SARTOR project) [9] developed a technique based on a logic for real-time systems (Real Time Logic (RTL)) and a specification language (Modechart). Real Time Logic first appeared in [10], and was inspired by Harel's statechart [11]. A method for verifying properties of systems specified in modechart was described in [12].

Both HyTech and Modechart are only suitable for the formal verification process during development. They cannot handle source code-level analysis against given properties. In addition, both formalisms are not compositional, which makes them hard for large-scale system evolution. The recent work on real-time constraints monitoring using RTL [13] and interval model checking [14] is based on Linear Time Logic (LTL). Although these analysers and their underlying logic are suitable for expressing real-time properties they both are non-compositional and not able to handle source code-level analysis. As we will see later on our tool is able to perform the necessary formal analysis at source code-level.

Analysers based on Anna [15] and PLEASE [16] are amongst early developed analysers. Anna is a language extension of Ada to include facilities for the formal specification of the intended behaviour of Ada programs. It augments Ada with precise machine-processable annotations so that well established formal methods of specification and documentation can be applied to Ada programs. Like Anna, PLEASE allows software to be annotated with formulae written in predicate logic; annotations can be used in proofs of correctness and to generate run-time assertion checks. As the logic in PLEASE is restricted to Horn clauses, specifications can be also transformed into prototypes which use Prolog to 'execute' pre- and post-conditions. Anna and PLEASE, however, do not deal with timing properties and are not compositional.

Temporal Rover [17] is a tool for the specification and verification/validation of protocols and reactive systems. It can automate the verification of real-time and relative temporal properties. The formal specification is written using a combination of Temporal Logic [18] and language of choice, such as C and

Java. Temporal-logic assertions are inserted into the body of executable code in conjunction with pieces of formal specification. Temporal-logic assertions can be simulated using the Temporal-Rover simulator. However, Temporal Rover verifies the systems based on simulation and is not compositional as well. It offers simple pre and post conditions that are not enough for handling complex parallelism. The tool has no graphical output yet.

Several techniques have been proposed to assure the correctness of systems. The most widely used is based on extensive *testing* which provides powerful and more accessible tools for rapid prototyping. However, such techniques can easily miss critical errors when the number of possible tests is very large.

Another technique is the use of formal methods with possible mechanical support like theorem provers and proof checkers [19]. The advantage of this technique is that it gives a mathematical proof of correctness. However, it is very time consuming and often requires a great deal of manual intervention.

Model checking [20] has been proposed to overcome the limitations of testing and formal verification methods. In this approach correctness requirements are expressed in a logic and systems are modelled as state-transition systems. In the first phase the basic behaviour of the system being studied is explored and an abstract representation of that behaviour is created. In a second phase this abstract representation is used to prove or disprove the correctness requirements using an efficient search procedure. The obtained answer is either true or a counter example that shows why the requirement is not satisfied. The most important advantage over interactive theorem provers or proof checkers is that it is completely automatic. The main disadvantage of this technique is the state explosion problem.

A related verification technique is *on-the-fly verification* [21]. As opposed to a traditional logic model checker, an on-the-fly verifier works with a single-pass verification procedure. It stores in memory as little information as is necessary for the correct completion of the verification process. The actual verification will use the smallest possible fragment of system's behaviour. The advantage of an on-the-fly system is often that it can handle larger problem sizes, measured in terms of the basic time and space requirements that are minimally needed to solve the problem, and generally can discover errors in a design faster than a conventional model checker.

Runtime verification differs from these techniques in that the system generates a behaviour while it is running and the runtime verification system checks whether this behaviour satisfies a property. Depending on the failure model one can stop and "repair" the system once an error is found or emit an error message and continue. The difference with testing is that testing is used *before* the system is employed. The difference with model-checking and on-

the-fly verification is that these check every possible behaviour while runtime verification *only* checks the runtime ones. Of course when a system generates all the possible behaviours during its lifetime then these techniques are equivalent. But the crucial difference is that for runtime verification one has only to check the runtime behaviours of the system which can be done in much more efficient way (no state explosion).

1.2 Organisation

The paper is organised as follows. Section 2 introduces our formal model. Section 3 describes the prototype of AnaTempura. Section 4 is devoted to a case study to illustrate the use of the tool. We conclude in Section 5 with some remarks and future work.

2 Formal Model

2.1 Computation

We begin by establishing a computational model which is suitable for modelling time-critical systems. At any instant in time a system can be thought of as having an unique *state*. The system state is defined by the state variables of the system and, for a concurrent system, by the values in the communication links. A *computation* is a sequence of states. A system should be tightly related to its *environment*. A system should realise correct computations to meet requirements from its environment and supply feedbacks to the environment. The environment of a system embraces all external factors or forces, which include surrounding entities, conditions or influences, especially affecting the existence or development of the system. For example, for a robot control system, its environment includes all sensors and actuators of the robot.

A *behaviour* in our model is defined as a sequence of states, i.e., an interval σ. Hence, a behaviour could be finite or infinite. A property is a set of behaviours. A general classification of properties are readily available: **safety** (*something bad does not happen*) and **liveness** (*something good will eventually happen*) properties.

In addition, six timing parameters can be used to express timing properties. They are [22]:

- *Start time:* the time instant when a computation is activated.

- *Computation (or execution) time:* the time interval between the start time and the termination time of a computation.
- *Deadline:* the upper limit for the termination of a computation.
- *Activation period:* the interval between two successive start times.
- Any *communication delays* incurred per message transferred.
- *average times* spent in queues.

By means of time, from the programmer's perspective, access to time can be provided either by a clock primitive in the language or via a device driver for the internal clock, external clock or radio receiver. We use processor's time in this case, i.e. the amount of processor (CPU) time (normally in microseconds) used since the first call in the calling process.

2.2 Interval Temporal Logic

Interval Temporal Logic (ITL) is a flexible notation for both propositional and first order reasoning about periods of time found in descriptions of hardware and software systems. It can handle both sequential and parallel composition unlike most temporal logics [23] since assumption/commitment paradigm and a set of compositional guidelines [24] are applied in ITL. There is a very powerful and practical compositional proof system for ITL [23]. That is, much of the proof of a system specified in ITL can be decomposed into proofs of its parts. It offers powerful and extensible specification and proof techniques for reasoning about properties involving safety, liveness and timeliness.

2.2.1 Syntax and Semantics

The key notion of ITL is an *interval*. An interval σ is considered to be a (in)finite sequence of states $\sigma_0,\ \sigma_1 \ldots$, where a state σ_i is a mapping from the set of variables Var to the set of values Val. The length $|\sigma|$ of an interval $\sigma_0 \ldots \sigma_n$ is equal to n (one less than the number of states in the interval [2], i.e., a one state interval has length 0).

Expressions

$$e ::= \quad \mu \mid a \mid A \mid g(exp_1, \ldots, exp_n) \mid \imath a{:}\, f$$

Formulae

$$f ::= \quad p(e_1, \ldots, e_n) \mid \neg f \mid f_1 \wedge f_2 \mid \ \forall v \bullet f \mid \mathsf{skip} \mid f_1 \,;\, f_2 \mid f^*$$

Fig. 1. Syntax of ITL

[2] This has always been a convention in ITL

The syntax of ITL is defined in Fig. 1 where μ is an integer value, a is a static variable (doesn't change within an interval), A is a state variable (can change within an interval), v a static or state variable, g is a function symbol and p is a predicate symbol.

The informal semantics of the most interesting constructs are as follows:

- skip: unit interval (length 1, i.e., an interval of two states).
- f_1 ; f_2: holds if the interval can be decomposed ("chopped") into a prefix and suffix interval, such that f_1 holds over the prefix and f_2 over the suffix, or if the interval is infinite and f_1 holds for that interval. Note the last state of the interval over which f_1 holds is shared with the interval over which f_2 holds. This is illustrated in Figure 2.

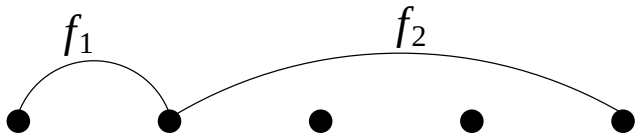

Fig. 2. Chop

- f^*: holds if the interval is decomposable into a finite number of intervals such that for each of them f holds, or the interval is infinite and can be decomposed into an infinite number of finite intervals for which f holds. Figure 3 illustrates the chopstar operator.

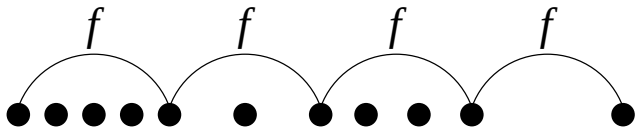

Fig. 3. Chopstar

- $\imath a$: f: choose a value of a such that f holds. If there is no such an a then $\imath a$: f takes an arbitrary value from a's range.

  Example: $\imath a$: skip ; $(x = a)$ which when evaluated over an interval of length at least one will give the value of x in the second state. If the interval is less than one it will give an arbitrary value.

2.2.2 Derived constructs

Following is a list of some derived constructs which are useful for the specification of systems:

- $\bigcirc f \;\widehat{=}\;$ skip ; f: next f, f holds in the next state. Example: $\bigcirc X = 1 \;\widehat{=}\;$ skip ; $X = 1$: Any interval such that the value of X in the second state is 1 and the length of that interval is at least 1. See Figure 4 for an interval satisfying $\bigcirc X = 1$.

- more $\widehat{=} \bigcirc$true: non-empty interval, i.e., any interval of length at least one.
- empty $\widehat{=} \neg$more: empty interval, i.e., any interval of length zero (just one state).
- inf $\widehat{=}$ true ; false: infinite interval, i.e., any interval of infinite length.
- finite $\widehat{=} \neg$inf: finite interval, i.e., any interval of finite length.
- $\Diamond f \widehat{=}$ finite ; f: sometimes f, i.e., any interval such that f holds over a suffix of that interval. Example: $\Diamond X \neq 1 \widehat{=}$ finite ; $X \neq 1$: Any interval such that there exists a state in which X is not equal to 1. See Figure 4 for an interval satisfying $\Diamond X \neq 1$.
- $\Box f \widehat{=} \neg\Diamond\neg f$: always f, i.e., any interval such that f for all suffixes of that interval. Example: $\Box X = 1 \widehat{=} \neg($finite ; $X \neq 1)$: Any interval such that the value of X is equal to 1 in all states of that interval. See Figure 4 for an interval satisfying $\Box X = 1$.
- fin $f \widehat{=} \Box($empty $\supset$ $f)$: final state, i.e., any interval such that f holds in the final state of that interval.
- $\bigcirc exp \widehat{=} \imath a: \bigcirc(exp = a)$: next value, i.e., the value of exp in the next state of the interval.
- fin $exp \widehat{=} \imath a:$ fin $(exp = a)$: end value, i.e., the value of exp in the last state of the interval.

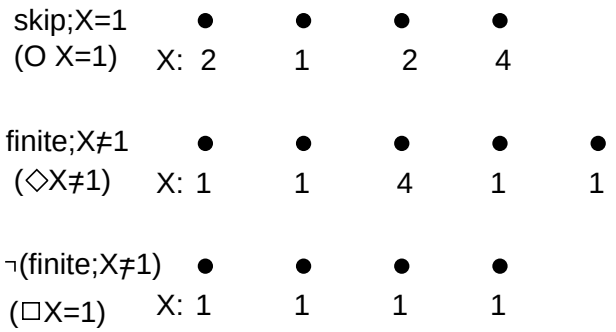

Fig. 4. Some sample ITL formulae

Not only can we specify systems at a high level of abstraction we can also specify systems at a lower level using programming constructs:

- if f_0 then f_1 else $f_2 \widehat{=} (f_0 \wedge f_1) \vee (\neg f_0 \wedge f_2)$: 'if then else' programming construct.
- while f_0 do $f_1 \widehat{=} (f_0 \wedge f_1)^* \wedge$ fin $\neg f_0$: the 'while loop' programming construct.
- $A := exp \widehat{=} \bigcirc A = exp$: 'assignment' programming construct, i.e., the value of A in the next state will be the value of exp in the first state of the interval.
- stable $A \widehat{=} \Box($more $\supset$ $A := A)$: 'stability' programming construct, i.e., the value of A doesn't change in the interval.

There are two basic builtin types in ITL (which can be given pure set-theoretic definitions). These are integers $\mathcal{N}$ (together with standard relations of inequality and equality) and Boolean (*true* and *false*).

Further types can be built from these by means of $\times$ (tuple operator) and the power set operator $\mathcal{P}$ (in a similar fashion as adopted in the specification language Z).

For example, the following introduces a variable x of type T

$$(\exists x : T) \cdot f \mathrel{\widehat{=}} \exists x \cdot (type(x, T) \wedge f)$$

Here $type(x, T)$ denotes a formula describing the desired type. For example, $type(x, T)$ could be $0 \leq x \leq 7$ and so on. Although this might seem to be a rather inexpressive type system, richer types can be added.

2.4 Tempura

An important reason of choosing ITL is the availability of an executable subset, known as Tempura [6], of the logic. A formula is executable if

- it is deterministic,
- the length of the corresponding interval is known.
- the values of the variables (of the formula) are known throughout the corresponding interval.

The Tempura interpreter takes a Tempura formula and constructs the corresponding sequence of states, i.e., interval. For more technical details of the interpreter, we refer the reader to [6] which is available from the ITL homepage [25].

The syntax of Tempura resembles that of ITL. It has as data-structures integers and booleans and list construct to built more complex ones. Tempura offers a means for rapidly developing, testing and analysing suitable ITL specifications. As with ITL, Tempura can be extended to contain most imperative programming features and yet retain its distinct temporal feel. The use of ITL, together with its subset of Tempura, offers the benefits of traditional proof methods balanced with the speed and convenience of computer-based testing through execution and simulation. The entire process can remain in one powerful logical and compositional framework.

3 The Prototype of AnaTempura

AnaTempura (available from [25]) is designed as a semi-automatic tool which aims to support the step-by-step methodology. This tool aims at helping engineers in handling time-critical systems evolution. However, it adopts a semi-automation architecture since human intervention is crucial and unavoidable in handling time-critical systems evolution due to difficulty to understand a program automatically. AnaTempura is an integrated workbench for ITL that offers

- specification support,
- validation and verification support in the form of simulation and run-time testing in conjunction with formal specification.

Fig. 5. The Analysis Process

AnaTempura is an open architecture that allows new tool components to be easily "plugged in". An overview of the run-time analysis process in AnaTempura is depicted in Figure 5. AnaTempura automatically monitors and analyses time-critical systems.

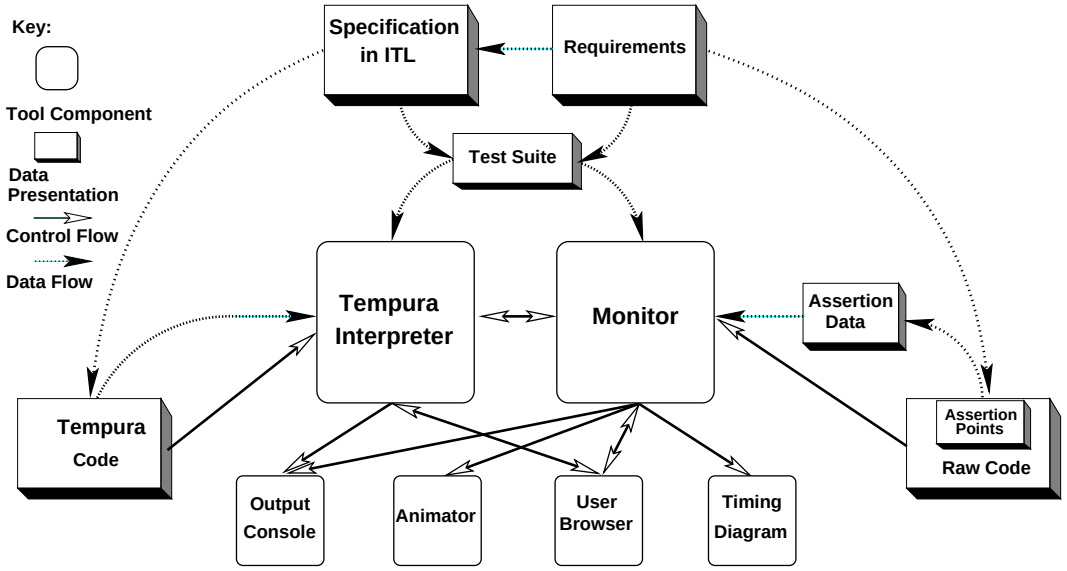

Fig. 6. General System Architecture of AnaTempura

The starting point is formulating all behavioral properties of interest, such as safety and timeliness. These are stored in a Tempura file. An information generating mechanism, namely, Assertion Points (Described in Section 3.1), will be inserted into the body of the raw code (e.g. C code). These Asser-

tion Points will generate run-time information (assertion data), such as state values, time stamps, during the execution of the program. The sequence of assertion data will be a possible behaviour of the system and for which we check the satisfaction of our property.

This checking is done as follows: start the *AnaTempura* and load the Tempura file. *AnaTempura* will then start the compiled raw code with Assertion Points. We then start the Tempura Interpreter to check whether the behaviours, generated by the Assertion Points, satisfy the properties. We note here that if the properties are not satisfied, AnaTempura will indicate the errors by displaying what is expected and what the current system actually provides. Therefore, the approach is not just a "keep-tracking" approach, i.e. giving the running results of certain properties of the system. There is also an facility to animate the received behaviours and to visualise timing properties.

Runtime verification offered by AnaTempura differs from other verification techniques in that the system generates a behaviour at runtime and the system checks whether this behaviour satisfies a given property. An appropriate action should be taken depending on the chosen failure model, one can stop and "repair" the system once an error is found or emit an error message and continue. The difference with testing is that testing is used *before* the system is employed. The difference with model-checking and on-the-fly verification is that these check every possible behaviour while runtime verification *only* checks the ones that are generated at runtime. Of course when a system generates all the possible behaviours during its lifetime then these techniques are equivalent. But the crucial difference is that for runtime verification one has only to check the runtime behaviours of the system which can be done in much more efficient way (no state explosion).

There are two main components in AnaTempura, *Monitor* and *Tempura Interpreter*. Monitor allows users to analyse the program at run-time with respect to a specification. Tempura Interpreter is used to execute Tempura files. AnaTempura also offers powerful visualisation function to enhance the ease of operation of the tool.

3.1 Assertion Points

Assertion points play a central role in AnaTempura. Assertion points are at the source-code level and divide a given code into chunks as depicted in Figure 7.

A typical automatic process of inserting assertion points is as follows:

(1) determine the set of variables V that is used to express the property,
(2) in the source code of the system for every variable in the set V we insert an assertion point directly after the program construct that assigns a value to this variable.

This can be easily performed by a simple preprocessor.

For example, the required information about at B_2 reflects the state of the system from B_1 to B_2, i.e., code chunk of C_1.

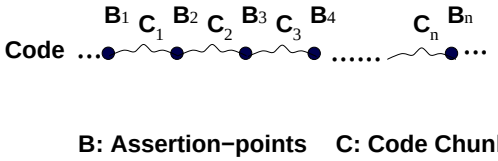

Fig. 7. Assertion points and Chunks

Assertion data consists of run-time information, including:

- *states of the system*: mappings between all variables and their values at a particular point, where assertion points are allocated, in the system's execution. In a state of the system, both the input to the system used during its execution and the value after system's computation are included. It will be generated as variable/value pairs. For example, the state:
  (input, 1), (Letter_Class, 1), (Air_Mail, undefined), (Tray, 1)
  tells that a mail sorter system gets input, "1", at the beginning of the execution, the variables, "Letter_Class" and "Tray", then have the value of 1 and the variable, "Air_Mail", remains undefined. Before system execution on the input begins, all variables are undefined.
- *time stamps:* mappings between a particular point, where Assertion Points are allocated and a value of a variable is being changed, and the time of change. Time stamps, obtained from the system clock, are additional information to states of the system. When using time stamps, the states of the system will be indicated by sets of triplets, (variable, value, time stamp). An example is as follows:
  (input, 1, 10), (Letter_Class, 1, 10), (Air_Mail, undefined, 10), (Tray, 1, 10)
  where "10" means 10 time units (for example, seconds). This data tells that the variables, "input", "Letter_Class", and "Tray" hold the value of 1 at time 10, and the variable, "Air_Mail" is undefined at that moment.
- *location:* either an assignment statement, an input statement, or an output statement, where an Assertion Point is inserted and gathers assertion data from this location and related neighbouring locations. A state error can be easily detected according the data of location. The location can be indicated by a name of a variable and its time stamp.

At an Assertion Point, the current system state information is sent to AnaTempura. These can then be used to check the validity of certain properties and generate a run-time report.

The use of assertion points makes it possible to monitor two components, X and Y. This monitoring process can be expressed in ITL as follows:

$$X \ \wedge \ Y$$

where X and Y could be two concrete programs, a concrete program (representation) and a property, or/and a requirement specification and a property. For example, we can check a concrete program (implementation) against its requirement specification (properties), i.e., we check whether its run-time behaviours satisfies the properties. Assertion points can help to realise automatic test generation, i.e., $\Box(input) \ \wedge \ Prop$, where *input* is a command to read the input data of the running program and *Prop* is the property we want to check.

When program control reaches an assertion point, the program will send the current state values to the monitor for validation. The sequence of state values generated by all assertion points forms a runtime behaviour of the system. AnaTempura will execute $\Box(input) \ \wedge \ Prop$ to validate whether this behaviour satisfies *Prop*.

3.2 The Monitor

The monitor is an interactive system with a user-friendly interface that allows the user to analyse the time-critical systems. The user can manually enter any predicate using the User Browser (Figure 6). We have used Tcl/TK [26] and Expect [27] to build the tool.

The main function of the monitor is to capture assertion data sent by assertion points. The monitor will automatically generate sequences of actions that fulfill simulation objectives (a particular state or series of actions) with respect to assertion data. The simulation results will be presented by the visualisation function (Section 3.3) in both textual manner (real-time report) and graphical manner (animation).

The monitor will execute the run-time analysing process described at the beginning of this section. The monitor captures the assertion data from the running program and then send it to Tempura interpreter. The Tempura interpreter executes the Tempura file and analyses the assertion data received and feedbacks judgements, *pass* or *fail*, in conjunction with where and how the failure happens, to the monitor. The monitor displays the received results

in the Output Console (Figure 6).

3.3 Visualisation in AnaTempura

The visualisation function of AnaTempura will give feedback on the execution of a program. It can visualise various program aspects, such as concurrency and timing information.

Three ways are used to visualise time-critical information.

- *Textual Representation* of the data, control flow information and so on. An *output console* has been set up for this purpose. For example, the timing values will be displayed on this console.
- *Timing Diagrams* present program data in a two-dimensional way with time information on one axis and individual task on the other.
- *Animation* of the program execution symbolically places each process, sometimes with portion of distributed data, on an individual point in a two dimensional display, corresponding to a single instant of time, i.e. snapshot. As time advances, the display changes like an animated movie.

The employment of the visualisation system will enable developers or users to:

- observe how computations are executed,
- visually develop and analyse systems
- monitor, analyse, and improve the performance of the system.

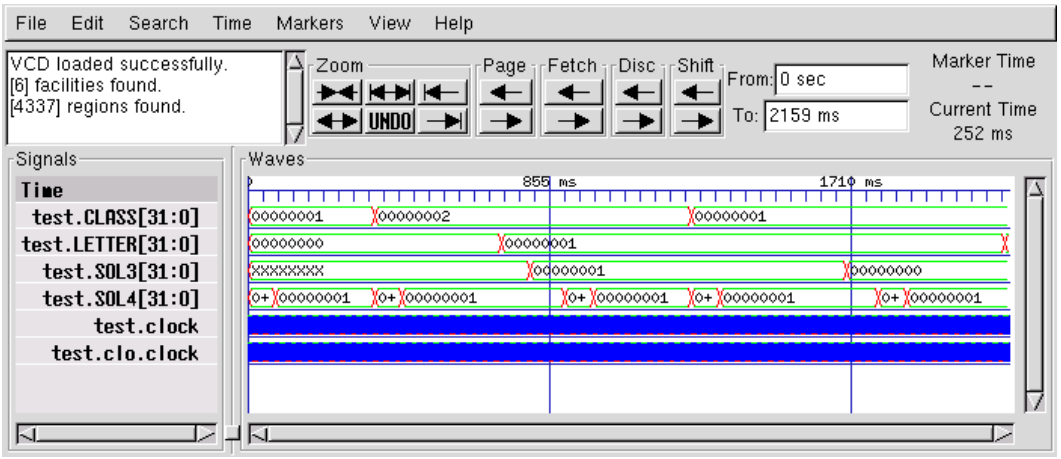

Fig. 8. Timing Information of State Variables

Timing information is shown using a timing diagram [28] (see Figure 8). This timing diagram will be helpful for analysing properties with respect to scheduling issues. The Veriwell [29] simulator and GTKwave [30] have been embedded

in AnaTempura to produce timing diagrams. Time stamps, generated by Assertion Points, are sent to the Veriwell simulator, who produces a *vcd* file, containing run-time information, such as names of variables and corresponding time stamps. This vcd file is then used by GTKwave to generate timing diagrams.

4 Case Study

In this section, we present a case study to illustrate the use of the tool.

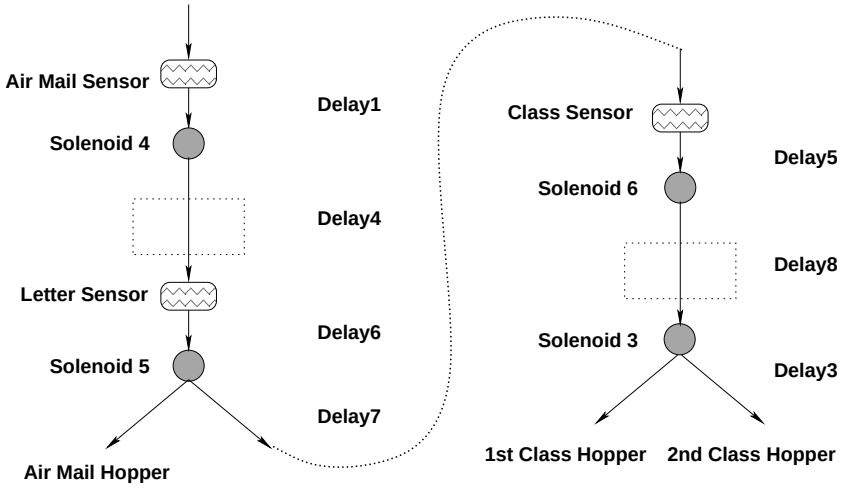

Fig. 9. The Structure of the Mail Sorter

Description This case study is extracted from a Post Office letter sorting system, which is used to sort First Class, Second Class and Air Mail letters.

The system is defined as follows in ITL:

$$
\begin{aligned}
Sorter \;\widehat{=}\; (& \\
& \mathsf{fin}\,(Time) - Time \leq 470 \wedge \\
& (\\
& \quad AirMail\_Sensor = Air \;\supset\; \Diamond Solenoid5 = ON \\
&) \wedge \\
& (\\
& \quad AirMail\_Sensor \neq Air \wedge Class\_Sensor = 1 \;\supset \\
& \qquad \Diamond Solenoid5 = OFF \wedge \Diamond Solenoid3 = ON \\
&) \wedge \\
& (\\
& \quad AirMail\_Sensor \neq Air \wedge Class\_Sensor = 2 \;\supset \\
& \qquad \Diamond Solenoid5 = OFF \wedge \Diamond Solenoid3 = OFF \\
&) \\
)^* &
\end{aligned}
$$

A class sensor and an Air Mail sensor detect the different colours of stamps and send corresponding signals to the control program which in turn sends signals to solenoids. The solenoids will turn the sorting gates *on* or *off* so as to release the letters into the correct hoppers.

Delays in the system, which guarantee that sensors and actuators (solenoids) can accomplish their tasks in time, occur at different places: Delay1 (70ms) and Delay4 (250ms) for Air Mail sensor and Solenoid 4 respectively; Delay 6 (70ms) and Delay 7 (80ms) for Letter sensor and Solenoid 5 respectively; Delay 5 denotes the detection time for the Class sensor, Delay8 denotes the execution time for the Solenoid 6 to release letters and Delay 3 denotes the execution time for the Solenoid 3.

There are two properties of interests: timeliness and safety properties. The former involves the delay times for switching the solenoids and reading the sensors, whilst the later are safety requirements: "*no Air Mail letter in the tray of first class letters or second class letter and visa versa*" and "*no Second class letter in the tray of first class letters and visa versa*". These properties are expressed in terms of the Air Mail, Class and Letter Sensors, switches (Solenoid 3 ,Solenoid 4, Solenoid 5 and Solenoid 6), and the various time delays (Delay1, Delay4, Delay6, Delay7, Delay5, Delay8, and Delay3).

The main actuators, Solenoid 3 and Solenoid 5, drop the letter into the correct hopper. If Solenoid 3 is "ON", the system will release the letter into the First class hopper, and when "OFF", it will release the letter into the Second class hopper. All timing parameters are extracted from the real mail sorter system.

In order to define the timing properties we introduce the following ITL formula which described a phase in the sorting process with its corresponding delay.

$$phase(Loc, Delay) \mathrel{\widehat{=}} [\mathsf{skip} \wedge \mathsf{fin}\,(Time) - Time = Delay \wedge LetterState = Loc \wedge \mathsf{stable}\,(LetterState)]$$

The timing property for sorting Air Mail letters is defined in Figure 10, it shows the time phases of the sorting process.

$$\begin{aligned} Prop_{air} = {} & phase(at\_air\_sensor, Delay1)\ ;\ \mathsf{skip};\\ & phase(at\_Solenoid\_4, Delay4)\ ;\ \mathsf{skip};\\ & phase(at\_letter\_sensor, Delay6)\ ;\ \mathsf{skip};\\ & phase(at\_Solenoid\_5, Delay7) \end{aligned}$$

Fig. 10. Timing property for sorting Air Mail letters

The timing property for sorting 1st/2nd Class letters is defined in Figure 11.

$$\begin{aligned} Prop_{1st/2nd} = {} & phase(at\_air\_sensor, Delay1)\ ;\ \mathsf{skip};\\ & phase(at\_Solenoid\_4, Delay4)\ ;\ \mathsf{skip};\\ & phase(at\_letter\_sensor, Delay6)\ ;\ \mathsf{skip};\\ & phase(at\_Solenoid\_5, Delay7)\ ;\ \mathsf{skip};\\ & phase(at\_class\_sensor, Delay5)\ ;\ \mathsf{skip};\\ & phase(at\_Solenoid\_6, Delay8)\ ;\ \mathsf{skip};\\ & phase(at\_Solenoid\_3, Delay3) \end{aligned}$$

Fig. 11. Process for sorting 1st/2nd Class letters

All the timing properties will be checked at run-time against above formulae. Checking runs for sorting 1st Class and 2nd Class letters are presented in Figure 12.

The top window shows the animation. The bottom window is a terminal to show outputs and inputs of the running program. The middle window shows

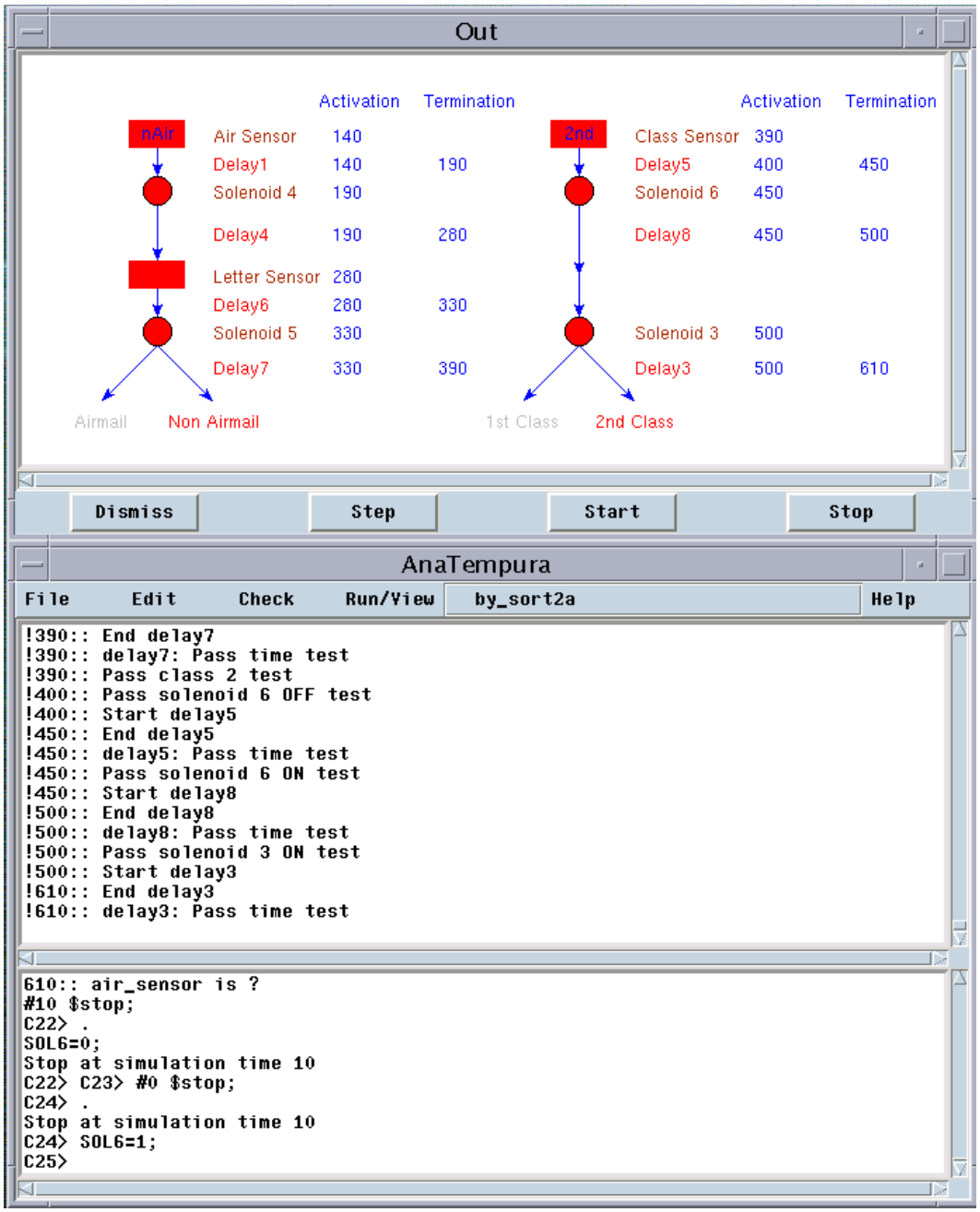

Fig. 12. Validation of Sorting Letters

the checking process. Assertion-points are placed (as described before) in the code to check the safety property and the timeliness property. Different designs of assertion-points will be used for different applications. For the mail sorter application, a parameterised C function `assertion` is introduced.

```
void assertion(char *aname, int val)
{ printf("!PROG: assert %s:%d:%d:!\n",aname, val, myclock());}
```

This function sends the state of the system (name of variable `aname` and the value `val` of that variable), and the current time (`myclock()` is a function which is used to capture the time). For example, `assertion("class",1)`, `assertion ("soloff",SolNo)` and `assertion ("solon",SolNo)` are used to check the safety property of the program, i.e., whether a First class can be released into the First class hopper. The following shows a fragment of C code with assertion points that is a part of the C program of the mail sorter:

```
void SolOn(int SolNo) { ...;assertion ("solon",SolNo); }
...
void SolOff(int SolNo) {...;assertion ("soloff",SolNo); }
...
if (class_sensor == 1) {
    assertion("class",1); SolOff(4); Delay(Dtime1,1);
    SolOn(4); Delay(Dtime4,4);
    scan_sensor ("letter sensor is ?",&letter_sensor);
    assertion("lsens",letter_sensor);
    if ( ! YellowSet){
      Delay(Dtime2,2); SolOff(3);
      Delay(Dtime3,3); YellowSet = 1;
    }
}
```

We note that

```
assertion("class",1);
assertion("lsens",letter_sensor);
```

are two assertion-points that generate run-time information. The tool checks and processes the sequence of information, compares it with the properties, and gives messages like "Pass Letter Sensor 1 test" and "Pass Class 1 test", which indicate that the safety and the timeliness properties are satisfied.

The part of the C program that is used to sort Air Mail letter is presented below. Corresponding assertion-points for sorting Air Mail letter have been inserted.

```
if (air_sensor == 1 ) {
  assertion("air",1); SolOff(4);
  Delay(Dtime1,1); SolOn(4); Delay(Dtime4,4);
  scan_sensor ("letter sensor is ?",&letter_sensor);
  assertion("lsens",letter_sensor);
  if( ! AirSet) {
    Delay(Dtime6,6); SolOff(5);
    Delay(Dtime7,7); AirSet = 1;
  }
```

```
}
```

where the line

```
assertion("air",1);
```

is used as an assertion-point of checking the process of sorting Air Mail letters. Three assertion-points are inserted for the relevant sensors and actuators, while three other assertion-points are inserted for the relevant timeliness property.

Checking runs for sorting Air Mail letters are also presented in Figure 12. They indicate that the safety property has been met for this particular behaviour. The program has sent correct control signals to the actuators, Solenoid 4 and Solenoid 5, and all Air Mail letters have been delivered to the Air Mail Hopper. At the same time, all timeliness properties have been satisfied. The delay times for the sensors and solenoids are correct.

5 Conclusion

We have presented a tool, based on a formalism known as Interval Temporal Logic. The tool allows us to monitor the program execution and capture possible behaviours of the system over which various interesting properties, such as timeliness and safety, can be validated at run-time. The tool also offers a powerful visualisation function to help users to understand the system. The tool was used on a small but illustrative case study of a mail sorter.

We believe that one of the strengths of the tool is its sound foundation. Interval Temporal Logic (ITL), with its rich axiomatic system and compositional proof rules, enables properties of interests to be verified and validated over intervals of time.

Unlike other tools and techniques, **AnaTempura** can be used for both sequential and parallel systems. In addition to the application domains it covers both timed and untimed applications. It is language independent, i.e., it can be used to process the legacy code in any language, such as C, C++, Ada and COBOL.

In the future, we plan to integrate the **AnaTempura** with an ITL verifier/Proof Checker, such as PVS [31] and Mona [32]. Currently, we are undertaking larger case studies from various critical application domains, including manufacturing industry and finance.

Acknowledgement

The authors wish to thank colleagues at the Software Technology Research Laboratory for their helpful criticisms and discussion during this work.

References

[1] J. Stankovic, K. Ramamritham, Hard Real-Time Systems: Tutorial Text, IEEE Comp. Society Press, 1988.

[2] S. Zhou, H. Zedan, A. Cau, A Framework for Analysing the Effect of 'Change' in Legacy Code, in: IEEE Proc. of ICSM99, 1999.

[3] S. Zhou, Compositional framework for the guided evolution of time-critical systems, Ph.D. thesis, De Montfort University (2003).

[4] B. Moszkowski, A temporal logic for multilevel reasoning about hardware, IEEE Computer 18 (2) (1985) 10–19.

[5] A. Cau, H. Zedan, Refining Interval Temporal Logic specifications, in: M. Bertran, T. Rus (Eds.), Transformation-Based Reactive Systems Development, Vol. 1231 of LNCS, AMAST, Springer Verlag, 1997, pp. 79–94.

[6] B. Moszkowski, Executing Temporal Logic Programs, Cambridge University Press, Cambridge UK, 1986.

[7] T. A. Henzinger, P. Ho, H. WongToi, HyTech: A Model Checker for Hybrid Systems, Software Tools for Technology Transfer 1 (1997) 110–122.

[8] R. Alur, T. A. Henzinger, P. Ho, Automatic Symbolic Verification of Embedded Systems, IEEE Transactions on Software Engineering 22 (3) (1996) 181–201.

[9] A. K. Mok, SARTOR- a Design Environment for Real-Time Systems, in: Proceedings of 9th IEEE COMPSAC, 1985, pp. 174–181.

[10] F. Jahanian, A. K. Mok, Safety Analysis of Timing Properties in Real-Time Systems, IEEE Transactions on Software Engineering 12 (9) (1986) 890–904.

[11] D. Harel, Statecharts: A Visual Formalism for Complex Systems, in: Science of Programming 8, 1986.

[12] F. Jahanian, D. A. Stuar, A Method for Verifying Properties of Modechart Specifications, in: Proceedings of Real-Time Systems Symposium, 1988.

[13] F. Jahanian, A. Goyal, A Formalism for Monitoring Real-Time Constraints at Run-Time, 20th Int. Symp. on Fault-Tolerant Computing Systems (FTCS-20) (1990) 148–55.

[14] S. Campos, O. Grumberg, Selective Quantitative Analysis and Interval Model Checking: Verifying Different Facets of a System, in: Rajeev Alur, Thomas A. Henzinger (Eds.), Proceedings of the Eighth International Conference on Computer Aided Verification CAV, Vol. 1102 of LNCS, Springer Verlag, New Brunswick, NJ, USA, 1996, pp. 257–268.

[15] S. Sankar, D. Rosenblum, R. Neff, An Implementation of Anna, in: J. G. P. Barnes, J. G. A. Fisher (Eds.), Proceedings of the Ada International Conference on Ada in Use, ACM, Cambridge University Press, Paris, 1985, pp. 285–296.

[16] R. B. Terwilliger, PLEASE: a Language Combining Imperative and Logic Programming, SIGPlan Notices 23 (4) (1988) 103–110.

[17] D. Drusinsky, The Temporal Rover and the ATG Rover, in: Proceedings of SPIN2000, Time-Rover Corp., Springer-Verlag, 2000.

[18] Z. Manna, A. Pnueli, The Temporal Logic of Concurrent Systems: Specification, Springer-Verlag, New York, 1991.

[19] J. Rushby, A Tutorial on Specification and Verification Using PVS, in: P. G. Larsen (Ed.), In proc. of the First International Symposium of Formal Methods Europe FME '93: IndustrialStrength Formal Methods, Odense, Denmark, 1993, pp. 357–406.

[20] E. M. Clarke, E. A. Emerson, A. P. Sistla, Automatic Verification of Finite State Concurrent Systems Using Temporal Logic Specifications: A Practical Approach, in: Conference Record of the Tenth Annual ACM Symposium on Principles of Programming Languages, ACM SIGACT-SIGPLAN, Austin, Texas, 1983, pp. 117–126.

[21] G. J. Holzmann, Design and Validation of Computer Protocols, Prentice Hall, 1990.

[22] A. Burns, A. Wellings, Real-Time Systems and Programming Languages, 2nd Edition, Addison-Wesley, 1997.

[23] B. Moszkowski, Some very compositional temporal properties, in: E.-R. Olderog (Ed.), Programming Concepts, Methods and Calculi, Vol. A-56 of IFIP Transactions, IFIP, Elsevier Science B.V. (North–Holland), 1994, pp. 307–326.

[24] H. Zedan, A. Cau, S. Zhou, A calculus for evolution, in: Proc. of The Fifth International Conference on Computer Science and Informatics (CS&I'2000), 2000.

[25] The ITL homepage.
URL `http://www.cse.dmu.ac.uk/~cau/itlhomepage/index.html`

[26] B. Welch, Practical Programming in TCL and TK, 3rd Edition, Prentice Hall, 1999.

[27] D. Libes, Exploring Expect, O'Reilly UK, 1994.

[28] R. Schlör, W. Damm, Specification and verification of system level hardware designs using timing diagrams, in: Proc. European Conference on Design Automation, 1993, pp. 518–524.

[29] S. Inc., VeriWell User's Guide (1994).

[30] GTKwave Electronic Waveform Viewer.
URL `http://www.cs.man.ac.uk/apt/tools/gtkwave/`

[31] A. Cau, B. Moszkowski, Using PVS for Interval Temporal Logic Proofs. Part 1: The syntactic and semantic encoding, Technical monograph 14, SERCentre, De Montfort University, Leicester (1996).

[32] R. Gómez, H. Bowman, Pitl2mona: Implementing a decision procedure for propositional interval temporal logic, Journal of Applied Non-Classical Logics 14 (1-2) (2004) 107–150.