Runtime Verification of Timing and Probabilistic Properties using WMI and .NET

Jane Jayaputera,
Iman Poernomo, Heinz Schmidt
SCSSE, Monash University, Australia
{janej,ihp,hws}@csse.monash.edu.au
Objectives:

• to generalize design-by-contract to assertions involving time and probabilities,
  – contracts are no longer Boolean propositions,
  – using a version of Probabilistic Computational Tree Logic (PCTL).

• to use contracts for runtime verification of reliability and availability properties of distributed architectures.

• to implement contract monitoring system within .NET
  – built upon the Windows Management Instrumentation (WMI) framework.
  – demarcation of contracts and code – any .NET application can be associated with contracts and monitored, without the need for plumbing code within the application.
Design-by-contract.

- Meyer’s general definition of a contract: a formal agreement between a system entity and its clients, expressing each party’s rights and obligations.

- Runtime verification and software design process:
  - contracts are specified during design of a system, expressing required rights and obligations
  - during the implementation phase, check for contract violations using a runtime contract monitoring system
  - this facilitates a way of testing that design requirements are met by the implementation
  - through specifying contracts and monitoring contract fulfillment, greater trust between a component and clients is achieved.
Design-by-contract for components.

- Design-by-contract has been effectively employed at a fine-grained level, expressing required functional properties (as boolean valued pre- and post-conditions) on individual methods of objects.

- Over the past 5 years, component-oriented adaptations of design-by-contract have been proposed – e.g.,
  - required relationships between provided and required interfaces of single components (Reussner, Poernomo and Schmidt)
  - contracts involving required usage profiles of component methods (Hofmeister and DePrince)

- Component-oriented design-by-contract:
  - contracts focus on public properties of black box components (not private properties of objects),
  - coarser grained constraints are defined at a higher level architectural design phase,
  - monitoring achieved via component-based instrumentation (rather than purpose built compiler).
Probabilistic, timed design-by-contract.

- We adapt the principles of design-by-contract to checking nonfunctional properties of systems.

- Our contracts define required availability and reliability rights and obligations of components.

- Availability and reliability constraints often involve probability and timing requirements – e.g., “after a request for a service, there is at least 98% probability that the service will be carried out within 2 seconds”.

- Difficult to express using simple Boolean functions – we therefore express our contracts as formulae written in a version of Probabilistic Computational Tree Logic.
PCTL

- Probabilistic Computational Tree Logic (PCTL) was originally devised as a specification language for probabilistic model checking (Hansson and Jonsson).

- It is used to specify that boolean properties hold within a given time and given likelihood.

- The language takes time to be discrete step units.

- For instance, PCTL can specify a property such as: “there is a 99% probability within 20 time steps that a requested service will be carried out”.

Syntax

- PCTL formulae are built from atomic propositions, the usual logical connectives (implication, conjunction and negation) and special operators for expressing time and probabilities.

- We adapted the grammar of PCTL to better suit our purposes (can encode within ordinary PCTL)

\[
F ::= \text{atom} \mid \text{not } F \mid F \text{ and } F \mid F \text{ or } F \mid F \text{ until } F \text{ steps: } \text{int} \mid F \text{ leadsto } F \text{ steps: } \text{int} \\
Q ::= F \text{ until } F \text{ steps: } \text{int prob: float} \mid F \text{ leadsto } F \text{ steps: } \text{int prob: float}
\]
Semantics

• Formulae are assertions about the possible execution of a system, made at the point of initial invocation by a client.

• The truth of a formula is determined according to the actual evolution of the system, and, in the case of probabilistic formulae, the number of times the formula has held true after previous invocations.

• Each execution run is considered as a series of discrete states. The transition from one state to another is determined by some system activity. The transition from one state to another should be considered as a discrete time step.
Semantics (2)

- Nonprobabilistic formulae consist of two kinds of formulae:
  
  - *Ordinary propositional formulae* are assertions about the state of a system, can be verified to be true or false at any point in an execution.

  - *Formulae with time.* These formulae make statements about the way a system may evolve, given certain assumptions hold at a state.

  - E.g., the statement

    \[ A \leadsto B \text{ steps: } s \]

    holds when, assuming \( A \) is true at a state, \( B \) will become true within \( s \) steps.
Semantics (3)

- *Probabilistic formulae* are understood in terms of corresponding nonprobabilistic formulae with truth averaged over a number of system runs.

- For instance,

  \[ A \text{ leadsto } B \text{ steps: } s \text{ prob: } p \]

  is true for a number of runs, if the corresponding nonprobabilistic formula

  \[ A \text{ leadsto } B \text{ steps: } s \]

  is true over these runs with a probability of \( p \).
Generic time steps.

- We are generic over what basic, atomic propositions are to be made about a system and also what a time step should be.

- It is the job of the specification expert to choose the system activity corresponding to a time step.

- The specification expert should also provide a set of atomic propositions, to stand a vocabulary of important system properties that can be verified at each state.

- In this way, the scope and application of PCTL as a specification language can be made to suit a particular domain.
Example

Consider a system in which PCTL time steps are taken to correspond to method calls to components within the system. The important system properties are “the system is in a failed state” and “the system is in a healthy state”. We denote these two properties by the propositions $Failed$ and $Healthy$, respectively.

Then a fault-tolerance constraint can be specified as

\[ Failed \xrightarrow{\text{leadsto}} Healthy \text{ steps: 2 prob: .999} \]

This states that, with a probability of .999, if the system is in a failed state after a client invocation, it will take two time steps to become healthy again.
Runtime verification – basic idea.

- The semantics defines how our formulae can be checked at runtime over the execution of a system.

- We instrument architectures to monitor and build a semantic model of system runs.

- At the end of each run, PCTL assertion formulae are checked against the total set of runs so far obtained.

- When the average behaviour falls below required probabilities, the assertion is violated and system tester is notified.
The WMI and .NET

- We implemented our monitoring system using the Microsoft Windows Management Instrumentation (WMI) framework in .NET.

- WMI is the core management enabling technology built into the Windows operating systems.

- The framework enables instrumentation and plumbing through which Windows resources can be accessed, configured, managed, and monitored.

- Because WMI is built into Windows, any .NET application can be instrumented very easily, without the need to write any extra code internal to the application.
Data and events

WMI enables the monitoring of data and events within an application:

- WMI events enable a subscription-based approach to monitoring important changes in an application. Windows and .NET provide a large base set of important events that can be monitored.
  
  - for example, component activation, method calls and exceptions are available for monitoring as WMI events without the need for manual instrumenting.
  
  - in addition, the developer can extend the WMI event model to accommodate domain-specific events.

- WMI can also collect a range of important data views of an application (e.g., public variables of components within the application).
Propositions and time steps.

Our assertion monitoring framework is generic over how we choose to associate WMI events with time steps and what basic PCTL propositions are to be made about data views.

- Time steps are generic over some chosen event that occurs regularly and can be monitored by the WMI. Possible WMI events could be method calls, ping heartbeat protocols, specific method call executions, or user-defined WMI time step events.

- PCTL formulae can be checked against a .NET system according to a semantic mapping that must be defined between atomic propositions and .NET Boolean valued functions about system data gathered by WMI. This provides the ability for the user to define the semantics of their atomic PCTL assertions.
Verification (1) Verification of formulae can be done according to the formal semantics, given a set of runs for the system. To construct such a set, we monitor atomic property truth values using WMI over multiple system invocations.

The monitoring process proceeds as follows for a single run:

1. We take an empty list, which will represent the run’s states.

2. The monitor waits for the arrival of the WMI event chosen to be associated with a time step.

3. Upon arrival of the event, the monitor should check truth values of each atomic proposition using the semantic map. Mappings between atomic propositions and truth values are stored in a hashtable, which is added as a node in the run’s list.

4. Steps 2 and 3 are repeated until program execution finished and the list for the run is completed.

We can check compound nonprobabilistic formulae with respect to a run, in the sense determined by the semantics.
Probabilistic assertion checks

- We begin to check probabilistic formulae after a certain number of runs.

- This number is configured by the administrator. When this number has been reached, a probabilistic formula is checked against the average truth value of its corresponding nonprobabilistic subformulae for the runs.

- This is then checked again after each new run has been built up.

- The monitor notifies the system administrator in the case that an assertion is false.
Domain specific libraries.

We envisage our monitoring approach to be used with libraries to be defined over particular domains, with a deployment package to be reusable for many applications in that domain.

A given system deployment in our framework involves associations between WMI events, monitored data views and elements of a PCTL assertion. In particular, a monitored system involves

- a library of WMI events and components. At a minimum, this consists of the standard WMI events that can monitor .NET applications.

- a vocabulary of atomic Boolean propositions (to be used in a PCTL grammar for assertions about the system).
Implementations of how to evaluate the atomic propositions for a system.

These libraries and semantic mappings between them are defined using an administration interface, using reflection to obtain information about domain specific WMI libraries. The user can specify contracts programmatically, as custom attributes of .NET classes, and also externally, using the administration interface.
Further work and conclusions.

It is envisaged that approaches similar to ours will become increasingly useful to coarse-grained system design, implementation and maintenance.

- **Design.** In previous work, the authors have defined a method for design of architectures with PCTL contracts. Using a state-based semantics of components, we used decidable model checking of PCTL contracts to improve the quality of designs.

- **Implementation and testing.** Used in conjunction with that approach, our runtime verification methods enable a second layer of trust by making certain that implementations conform to designs. Contract violation at the testing stage can lead to reimplementation or to refactoring of design.
- Maintenance. Runtime verification of non-functional properties has potential for application simpler maintenance and administration of deployed applications. Large-scale service-oriented software often has cost tariffs associated with failure to meet availability constraints. Accurately specifying and monitoring such constraints is therefore an important issue. A semiformal approach such as ours goes some way to increasing trust in this area.