Modeling Enhanced Scenarios for Automated Instrumentation

Dave Arnold and Jean-Pierre Corriveau
School of Computer Science
Carleton University, Ottawa, Ontario, Canada
{darnold, jeanpier}@scs.carleton.ca

Abstract

There is a resurgence of research in model-based testing, especially in the automated generation of test cases from abstract models. However, this work largely remains theoretical: industrial adoption is low. This is partly due to the dominance of state-based approaches that often rely on global states that are problematic with respect to scalability and traceability. Developers and testers alike significantly prefer the intuitive nature, traceability and user-friendliness of scenarios, to the semantics of formal approaches. Proposals for scenario-driven testing exist but, as is the case for the vast majority of existing work on model-based testing, there is a considerable gap between the generated test cases and their corresponding IUT instrumentation. It is this problem we address here. In this paper we focus on modeling responsibilities and scenarios within a scenario-driven testing framework that generates fully-instrumented test cases. Our work proceeds from the scenario contracts proposed by Nebut et al.

1. Introduction

In her seminal review of the state-of-the-art in software testing, Bertolino [1] remarks: “A great deal of research focuses nowadays on model-based testing. The leading idea is to use models defined in software construction to drive the testing process, in particular to automatically generate the test cases. The pragmatic approach that testing research takes is that of following what is the current trend in modeling: whichever be the notation used, say e.g. UML or Z, we try to adapt to it a testing technique as effectively as possible [...] The idea of model-based testing has been around for decades [...] but it is in the last few years that we have seen a groundswell of interest in applying it to real applications. Nonetheless, industrial adoption of model-based testing remains low and signals of the research-anticipated breakthrough are weak.” In his explanation of the very limited adoption of his industrial-strength state-based model-testing tools at Microsoft, Grieskamp [2] proposes several possible reasons. In particular, he remarks that developers and testers alike significantly prefer the intuitive nature of scenarios [3] to the semantics underlying more formal (not only logic-based and set theoretic, but also state-based) approaches.

Indeed, the use of scenarios (e.g., UML use cases) as a method for requirements capture is commonly accepted [4]; several software design paradigms are grounded in the notions of responsibilities and scenarios [5]. This is the case, in particular, for object-oriented reactive systems, which are the ones we consider here. Thus, not surprisingly, scenarios (such as Use Case Maps [6] and Binder’s extended Use Cases [7]) have also recently been used (sometimes in combination with state machines [3]) for model-based testing. Few, however, consider inter-scenario relationships [8] in the process of generating test cases. Among those who do, Nebut et al. [9] propose algorithms for test case generation from use-case like scenarios enhanced with contracts, that is, with pre- and post-conditions [10]. They then argue that, contrary to diagrammatic approaches to the sequential ordering of scenarios, such as Briand and Labiche’s use of UML activity diagrams [11], contracts are scalable (especially with respect to production and readability). These authors also demonstrate that coverage strategies akin to those of state-based approaches are readily available for their scenario contracts. We can therefore conclude that the automated generation of test cases from scenario-based models is possible (though difficult problems, such as the probable combinatorial explosion of the number of states, sequences of scenarios, and test cases, remain). But it is essential to remark that in the vast majority of existing work on model-based testing, there is a huge gap between the test cases generated and their corresponding instrumentation in an IUT. It is this problem we address here. To the best of our knowledge, with the notable exception of Grieskamp’s work at Microsoft [2], very few have succeeded at
building a model-driven testing framework that generates fully-instrumented test cases. In this paper, we focus on the modeling of responsibilities and scenarios within a scenario-driven testing framework that generates fully-instrumented test cases.

Our work proceeds from the scenario contracts proposed by Nebut et al. [9]. Whereas they simply augment scenarios with pre- and post-conditions, we further enhance scenarios and responsibilities with static checks, dynamic checks, and metrics in order to define a system’s contracts1. Most importantly, the modeling additions we propose do not modify the algorithms of Nebut et al. for test case generation, as discussed elsewhere [12]. Consequently, here, we focus instead on the semantics our validation framework (VF) can instrument automatically. That is, in this paper, we do not address test case generation per se but rather overview the types of modeling elements for which we have realized automated instrumentation. We emphasize that, in contrast to most model-driven testing toolsets [13]2, such automated instrumentation eliminates the need for glue code. Let us briefly elaborate. Because in most existing work on model-driven testing, the generated test cases are totally disconnected from an IUT, it is necessary for some programmer to express these test cases and their underlying checks into code. In other words test drivers for procedures, instances and scenarios must be built and may not correspond to the generated test cases. Automated instrumentation eliminates this risk and enforces traceability between test cases and their implementation.

Furthermore, we require that our modeling be decoupled from any particular implementation so that a single set of contracts may be tested against several candidate implementations. (Clearly, an implementation-specific approach to validation lacks abstraction and generality and thus decreases the possibility of test case reuse across implementations.) The question then is how can our implementation-independent model be used to generate tests that are automatically instrumented into an IUT? The answer lies in the idea of binding elements of a system’s contracts to actual procedures in the IUT. More specifically, some elements of a contract must be explicitly linked by the provider of an IUT to corresponding procedures of this IUT. For example, a responsibility of a scenario may have to be explicitly bound to a procedure in the IUT. If such mapping does not exist, then, alternatively, the IUT provider may define two unique events (observable_start_event and observable_end_event) associated with this responsibility and then bind these events to procedures in the IUT3. Unlike glue code, no programming is involved in such bindings, which merely build explicit traceability links that enable the automated instrumentation of the tests generated from a system’s contracts in the IUT.

In our work, each scenario is specified as a grammar of responsibilities. Each responsibility represents a simple functional unit. A responsibility is either:

1. Bound to a procedure within an IUT.
2. Bound (via observable events, as hinted above) to some procedures that through their interactions implement the responsibility.
3. Decomposed into a sub-grammar of responsibilities. In turn, each responsibility within the sub-grammar is then bound or further decomposed.

Without going in to detail, test case generation is a two-step process: how we deal with inter-scenario relationships is rooted in the algorithms of Nebut et al. [9]. Then, within each scenario, specific coverage of its grammar of responsibilities4 is achieved through traditional state-transition coverage algorithms [7]. (For more details, see [12]).

As a first modeling enhancement, static checks are restricted to a contract addressing a responsibility or a set of responsibilities (respectively bound to procedure(s) and class(es) in the IUT). Static checks do not impact test case generation. Conversely, both scenarios and responsibilities can refer to dynamic checks (inspired from Design-by-Contract (DbC) [9, 10]). Such checks (e.g., pre- and post-conditions) are typically used to express constraints on the state of one or more objects (existing during a particular execution) before and after the execution of a scenario or responsibility. And, as previously mentioned, they pertain to the generation of test cases accounting for inter-scenario relationships. Let us briefly elaborate. Even simple software systems are composed of multiple scenarios. As such, validation includes not only the execution of individual scenarios, but also the

---

1 Static checks correspond to structural queries on an implementation, whereas dynamic checks and metric evaluation query an execution of an implementation. Such queries can be associated with scenarios and responsibilities (see [12]).
2 The state-based work of Grieskamp [2] being the noticeable exception we are aware of.
3 It is possible that the provider of the IUT may have to add to this IUT procedures corresponding to responsibilities and observable events. In other words, a system’s contracts do impose observability requirements that must be implemented in the IUT. Also, please note that the cumbersomeness of creating such bindings is largely alleviated through the use of a ‘smart’ binding tool, as explained elsewhere [12]. This tool, in particular, notifies its user of missing bindings after having attempted to infer automatically as many as possible.
4 This does not suffer from combinatorial state explosion as it involves a few instances in a few specific states.
execution of multiple, possibly interleaving, scenarios. Such scenario integration requires inter-scenario operators to support temporal ordering, concurrency, and distribution. Theoretical work on such operators is absent from many scenario-based testing work but is addressed at length in SCENT by Ryser and Glinz [3, 8]. Our proposed modeling approach and its corresponding VF make operational portions of their work, as explained at length elsewhere [12].

Beyond verifying the static and dynamic checks of scenarios and responsibilities, the grammar of responsibilities of each scenario, and inter-scenario relationships, we require that our model also address non-functional requirements (such as performance, security, and usability). In our work, such non-functional requirements are test-enabled by gathering metrics during the execution of scenarios. Once a metric has been gathered, it needs to be analyzed, and ultimately, trade-off analysis must take place. Metric and trade-off analysis is an extremely domain specific activity. To support multiple domains within our VF, our framework supports pluggability (i.e., user-defined functional extensions). In other words, it is possible to define in our VF domain-specific extensions to our testable model, such extensions (called plug-ins) still enable automated instrumentation in an IUT (see [12]).

In the next section, we summarize the mode of operation of our framework. Then, in section 3, we walk through the details of a contract that is part of a large case study we have carried out. Two more contracts of this case study are provided in the appendix, after the conclusions found in section 4.

2. The Framework

Our VF provides an open architecture for the specification and execution of our testable model. The VF is being integrated into Microsoft’s Visual Studio for ease of use and integration with existing development processes. The VF accepts three elements as input. The first element is the contracts specification that corresponds to our testable model. We have defined a high-level general purpose contract language known as Another Contract Language (ACL) [12]. The ACL includes constructs for the capture of static checks, dynamic checks, responsibilities, scenarios, metric capturers, and metric evaluators. Additional domain-specific constructs can also be added to the ACL, via VF plug-ins [14].

The second input element is one or more candidate IUTs against which the ACL contracts will be tested. Our VF accepts implementations in binary form. That is, source code for the IUT is not required. Through the use of Microsoft’s Phoenix Research Development Kit [15], the VF is able to open .NET managed executables, as well as C++ binaries that contain debugging information.5

Bindings represent the third and final input element to our framework. Each IUT has a unique set of bindings. Before a contract can be verified by the VF, some of its elements must be bound to implementation artifacts located within the IUT. Such bindings are essential for the automated instrumentation and execution of tests. The binding process begins with Phoenix being used to obtain a structural representation of the IUT. The VF’s binding tool then allows for the specification of a mapping between each responsibility defined within a contract to an actual procedure (or group of procedures) within the IUT (as previously mentioned). Most importantly, bindings allow contracts to be independent of implementation details, as specific procedure names, and parameter types/orders used within the IUT do not have to correspond to a similarly-named contract artifact. In addition, we repeat, such a binding step allows several candidate IUTs to be verified against a single testable model.

Details of how a contract is compiled, instrumented and executed can be found elsewhere [14].

3. A Brief Example

In this brief example we will present in detail one contract of a testable model representing a physical grocery store. Two more contracts can be found in the appendix.

Our grocery store can be viewed as a set of customers who each enter the grocery store and select one or more items for purchase. Each of the items selected by a customer is placed into a shopping cart. Once the customer has finished selecting the items that s/he wishes to purchase, the customer proceeds to the checkout where s/he selects an open cash, and then joins the corresponding (possibly empty) queue. Finally the customer purchases the selected items and leaves the store. Furthermore, we will make the following assumptions about the grocery store:

- Stores always remove all food items and cash registers when closed.
- Each food item is unique (i.e. two cans of soup are different items).
- Cashes cannot be closed if they have customers waiting.
- Customers cannot leave a queue once they enter it.

5 The debugging information is required to resolve symbol names during the binding process.
• Customers enter only if they are buying something.

The complete grocery store case study requires many additional aspects. However this short description will suffice to illustrate the modeling of a scenario, its interactions, and even of a non-functional requirement. We will focus here on the specific contract representing the grocery store proper, given in Figure 1.

```java
Contract Store {
    Value Integer openCashes;
    Value Integer totalOpenCashes;
    Structure {
        Belief HasCash {
            HasMemberOfType(tCashContainer);
        }
    }
    Observability Integer OpenCashes();
    Observability Boolean IsOpen();
    Observability Boolean HasCash(tCash x) {
        cashContainer.Contains(x);
    }
    Invariant OpenCashNumber {
        context.openCashes >= 0;
        context.openCashes == OpenCashes();
    }
    Responsibility new() {
        context.openCashes = 0;
        context.totalOpenCashes = 0;
        Post(foodContainer.IsEmpty() == true);
        Post(cashContainer.Contains(c) == true);
        Post(IsOpen() == true);
    }
    Responsibility finalize() { }
    Responsibility Open() { }
    Responsibility Close() { }
    Responsibility OpenCash(tCash c) {
        Pre(IsOpen() == true);
        Pre(c not= null);
        Pre(cashContainer.Contains(c) == true);
        Pre(c.IsOpen() == false);
        openCashes = openCashes + 1;
        totalOpenCashes = totalOpenCashes + 1;
        Post(cashContainer.Contains(c) == true);
        Post(c.IsOpen() == true);
    }
    Responsibility CloseCash(tCash c) { }
    Responsibility AddFood(tFoodItem i) { }
    Responsibility RemoveFood(tFoodItem i){ }
    Scenario MainStore() {
        Trigger(new()),
        { Open(),
            (AddFood(dontcare))+,
            OpenCash(dontcare),
            { OpenCash(dontcare)
                | CloseCash(dontcare)
            parallel{
                once Value tCash s;
                once Value tCustomer c;
                c = newInstance tCustomer,
                c.EnterStore(context),
                (c.AddFood(dontcare))+,
                s.AddCustomer(c),
                c == s.NextCustomer(),
                c.RemoveFood(),
                c.Pay(),
                c.LeaveStore(context)
            }+, (RemoveFood(dontcare))+, Close()
        }+, 
        Terminate(finalize());
    }
    Metric Integer TotalOpenCashes() {
        context.totalOpenCashes;
    }
    Reports {
        Report("Cost to open cashes: {0}",
            CashCost(TotalOpenCashes()));
    }
    Exports {
        Type tFoodItem conforms Item;
        Type tCash conforms Cash;
        Type tCustomer conforms Customer;
        Type tFoodContainer conforms 
            BoundedContainer<tFoodItem, 10000>;
        Type tCashContainer conforms 
            BoundedContainer<tCash, 10>;
        Field cashContainer tCashContainer;
        Field foodcontainer tFoodContainer;
    }
}

Figure 1. Grocery Store Contract Listing in ACL

The Contract keyword denotes the beginning of the contract named Store. The contract’s body begins with the declaration of two contract variables. Contract variables are used to store information during the execution of the contract. Such information can be used during the invocation of dynamic checks and for the capture of metric information. The openCashes contract variable will be used within a check, and totalOpenCashes will be used for metric capture. Both contract variables are declared using the Value modifier. The Value modifier indicates that the contract variable will contain a scalar value. The List modifier (not shown here) is used to denote an ordered list of values.

Each contract is composed of several contract sections. Each contract section has a predefined and independent semantics. The Store contract contains each of the supported contract sections. The first section is a structure section. The structure section contains any static checks that are to be performed before the IUT is executed. The Store contact contains a single static check named HasMemberOfType that determines if the IUT type the Store contract is bound to, defines a field matching the type denoted by the
Following the structure section, the Store contract defines three observability methods. An observability method is used to access state information about the IUT during execution. Observability methods are defined in one of two ways:

1. The observability method can be bound to an actual procedure within the IUT.
2. A body can be specified to calculate the return value using other existing observability methods.

Observability methods are invoked by responsibilities and scenarios. They are guaranteed to be side-effect free. Instrumentation is added to the IUT to ensure that the observability methods are indeed side-effect free.

A single invariant follows the observability methods. Invariants are evaluated before and after each responsibility (see below) to ensure that the IUT is in a valid state. The invariant defines two checks, the first ensures that there is never a negative number of open cashes, the second ensures that the contract variable representing the number of open cashes is in agreement with the value returned by the predefined observability method. That is, the number of open cashes reported by the executing IUT matches the value stored within the contract.

Next, a series of responsibilities are defined. A responsibility represents a unit of functionality that is required from the IUT. Responsibility bodies can define preconditions and post-conditions to be checked before and after the responsibility has been executed. Responsibility bodies can also invoke dynamic checks and perform calculations on contract variables. There are two special responsibilities: new and finalize. The new responsibility is automatically bound to any constructors that are defined within the IUT type bound to the containing contract. Likewise the finalize responsibility is automatically bound to any destructors. In the Store contract the new responsibility is used to initialize the contract variables, and uses post-conditions, denoted by the Post keyword, to ensure that the store has been created correctly. The body of the finalize responsibility is omitted here due to space constraints. However it would contain preconditions, denoted by the Pre keyword, to ensure that all resources have been released prior to destruction. The bodies of the Open and Close responsibilities have also been omitted; these responsibilities would be bound to IUT procedures that open and close the grocery store respectively. The body of the OpenCash responsibility uses preconditions to ensure that the store is open, the cash register is valid and is part of the store, and is not already open. Next, the body updates the contract variables, and then defines post-conditions to ensure that the given cash register has been opened. The omitted body of the CloseCash responsibility performs the opposite action ensuring that the given cash register has already been opened and is part of the store, updates the contract variables, and checks that the given cash register is now closed. Likewise the AddFood and RemoveFood responsibilities are used to add and remove food items from the store respectively. Their bodies would provide constraints to check the state of the food items being added and removed and to ensure that the correct add or remove action was performed correctly.

Following the responsibility definitions, a single scenario is defined. Each scenario must contain a scenario trigger, denoted by the Trigger keyword, and a scenario termination condition, denoted by the Terminate keyword. (The trigger and termination conditions correspond to Nebut et al. pre- and post-conditions on use cases [9].) The MainStore scenario is triggered when the IUT type bound to the Store contract is instantiated and a constructor has been executed. The first responsibility the scenario grammar expects is Open. Once the store has been opened, one or more food items are added to the store via the AddFood responsibility. The individual food items are not of interest to our high-level store scenario and consequently are referenced using the dontcare keyword. Once the food items have been added to the store, the grammar specifies that a single cash must be opened. Next, the scenario grammar specifies that one of three actions can occur:

1. A new cash is opened via the OpenCash responsibility.
2. A cash is closed via the CloseCash responsibility.
3. The third action that can occur involves the introduction of the parallel keyword. The parallel keyword is used to denote that the scenario grammar specified within the brace brackets can be seen as a single responsibility. That is, multiple instances of the same scenario grammar section may be executing in parallel at any given time. A parallel section can also be viewed as a sub-scenario within the main scenario, where there can be any number of sub-scenarios active...
at any one time. In the case of the MainStore scenario, several customers could be in the store buying items. The parallel section begins with the declaration of two scenario variables named s and c. The s scenario variable will be of the IUT type bound to the tCustomer symbol. The c scenario variable will be of the IUT type bound to the tCash symbol. The first scenario grammar element within the parallel section is called when a new customer is created. The first scenario grammar element within a parallel section can be viewed as the sub-scenario trigger. The grammar element also introduces the newInstance keyword. The newInstance keyword specifies that the scenario grammar continues only when a new instance of the IUT type bound to the tCustomer symbol is created. Next, the customer enters the store and adds one or more food items to his/her cart via the Customer::AddFood responsibility. The customer then selects a cash and is added to the end of the queue. The scenario continues, when the customer gets to the front of the queue, and removes his/her food items from the cart via the Customer::RemoveFood responsibility. Finally, the customer pays for the items and leaves the store.

The MainStore scenario concludes with the removal of the food items, followed by the closing of the store via the Close responsibility. The outer plus (+) operator indicates that the store can be opened and closed one or more times during the execution of the IUT.

Following the scenario definition, a single metric method is defined. Metric methods are used to access metrics gathered by the contract. The metric is named TotalOpenCashes and returns the value stored in the totalOpenCashes contract variable. Metric methods can only be invoked from within a reports section. The reports section is evaluated by the VF following execution of the IUT. The reports section consists of metric evaluators being invoked to analyze the metric information gathered during execution. The reports section defined within the Store contract contains a single report statement that uses the metric evaluator plug-in named CashCost to determine the cost required with opening the given number of cashes. This value is obtained by using the previously defined TotalOpenCashes metric method. The result of executing the report statement is displayed on the contract evaluation report.

Finally, the Store contract contains an exports section. This section is used to specify the bindings required for the contract. The first export line, binds the tFoodItem symbol to an IUT type that conforms to an externally defined Item contract. Likewise, the second export line binds the tCash symbol to an IUT type that conforms to the Cash contract. The third export line, binds the tCustomer symbol to an IUT type that conforms to the Customer contract. Next, the tFoodContainer and tCashContainer symbols are bound to IUT types that conform to the BoundedContainer contract using the specified generic contract parameters. Finally, exports to the two internal container fields are specified so that the IUT’s containers can be referenced within the contract.

Two more contracts, a customer and a cash, are provided in the appendix.

4. Conclusions

As Bertolino observes [1], current work on model-based testing focuses on test case generation from a specific modeling notation but such test cases are generally disconnected from an implementation under test. As such, this work is seldom adopted in an industrial context. Following Grieskamp [2], we believe model-based testing must entail automated instrumentation of test cases in order to be relevant in practice. Industrial relevance also demands that a testable model be semantically rooted in the notions of responsibilities and scenarios that promote traceability between high-level modeling concepts and implementation constructs. In this paper, we have presented the basic elements of such a model. The point to be grasped is that, contrary to existing research in model-based testing, the semantics of our proposed model are established from what can be instrumented automatically in our validation framework. In other words, it is testability that drives modeling, not the other way around. Furthermore, our work is integrated into a widely-used development environment and, through plug-ins, enables domain specializations. This directly addresses two further challenges Grieskamp (Ibid.) identifies for industrial model-based testing.

5. Acknowledgements

Support from the Natural Sciences and Engineering Research Council of Canada (NSERC) is gratefully acknowledged.

6. References


7. Appendix

Here is the customer contract.

```java
Contract Customer
{
    Value tStore store;
    Value Integer cartSize;
    Observability Boolean HasPaid();
    Observability Boolean InStore()
    {
        context.store == null;
    }
    Observability Boolean HasFood()
    {
        not foodContainer.IsEmpty();
    }
    Responsibility new()
    {
        context.store = null;
        context.cartSize = 0;
    }
    Responsibility EnterStore(tStore s)
    {
        context.store = s;
    }
    Responsibility LeaveStore(tStore s)
    {
        Belief SameStore
        {
            Pre[s == context.store];
        }
        context.store = null;
    }
    Responsibility AddFood(tFoodItem i)
    {
        Pre[i not- null];
        Pre(foodContainer.Contains(i) == false);
        Post(foodContainer.Contains(i) == true);
    }
    Responsibility RemoveFood()
    {
        Pre(HasFood() == true);
        context.cartSize = foodContainer.Size();
        Post(HasFood() == false);
    }
    Responsibility Pay()
    {
        Pre(HasPaid() == false);
        Post(HasPaid() == true);
    }
    Responsibility tCash SelectCash(tStore s)
    {
        Pre[s not- null];
        Pre[s.IsOpen() == true];
        Pre[InStore() == true];
        Pre[s == context.store];
        Post[value not- null];
        Post[s.HasCash(value)];
        Post[value.IsOpen() == true];
    }
    Scenario BuyItems
    {
        Pre[SelectCash(store)];
        exported once Value tCash cash;
        Trigger(new());
        EnterStore(store),
        (AddFood(dontcare)+);
        atomic
        {
            cash = SelectCash(store),
            Belief ValidCash
            {
                Check(cash not- null);
            },
            cash.AddCustomer(context),
            context == cash.NextCustomer(),
            RemoveFood(),
            Pay(),
            LeaveStore(store);
        }
        Terminate(finalize());
    }
    Metric Integer CartSize()
    {
        context.cartSize;
    }
    Metrics
    {
        ReportAll("The avg size of a cart is: [{0}]", AvgMetric(CartSize()));
    }
    Exports
    {
        CartSize;
    }
}
```
Figure 2. Customer Contract Listing in ACL

Here is the contract for the cash.

Contract Cash {
    List Integer customer_times;
    Value Integer processed_customers;
    Observability Boolean isOpen();
    Observability Boolean HasCustomers() { not queue.isEmpty(); }
}

Responsibility new() {
    customer_times.Init();
    processed_customers = 0;
}

Responsibility Open() {}

Responsibility Close() {}

Responsibility AddCustomer(tCustomer c) {
    Pre(queue.Contains(c) == false);
    Post(queue.Contains(c) == true);
}

Responsibility tCustomer NextCustomer() {
    Value tCustomer next;
    Pre(isOpen() == true);
    Pre(hasCustomers() == true);
    next = PreSet(queue.front());
    Post(value not= null);
    Post(next == value);
    Post(queue.Contains(value) == false);
}

Responsibility ProcessCustomer(tCustomer c) {
    Belief CustomerCheck {
        Pre(c not= null);
        Pre(queue.Contains(c) == false);
        Belief NotPaid {
            Pre(c.HasPaid() == false);
        }
    }
    Belief SelectedCash {
        Pre(c.buyItems().cash == context);
    }
    c.removeItem(),
    c.pay(),
    atomic {
        processed_customers = processed_customers + 1;
        Belief DoneCheck {
            Check(c.HasPaid() == true);
        }
        c.leaveStore(dontcare);
    };
}

Scenario RunCash {
    Value Integer count;
    Trigger(new());
    atomic {
        count = 0;
        Open();
    },
    atomic {
        count = count + 1;
        AddCustomer(dontcare)
    }
    atomic {
        count = count - 1;
        ProcessCustomer(NextCustomer())
    }*,
    Close()
}*,
    Terminate(finalize());
    Belief AllProcessed {
        Post(count == 0);
    }
}

Metric List Integer WaitingTimes() {
    context.customer_times;
}

Metric Integer ProcessedCustomers() {
    context.processed_customers;
}

Metrics {
    Report("Avg customer waiting time: {0}",
        AvgMetric(WaitingTimes()));
    Report("Max customer waiting time: {0}",
        MaxMetric(WaitingTimes()));
    Report("Min customer waiting time: {0}",
        MinMetric(WaitingTimes()));
    Report("Number of Customers: {0}",
        ProcessedCustomers());
}

Exports {
    Type tCustomer conforms Customer;
    Type tQueue conforms BoundedQueue<tCustomer, 100>:
    Field queue tQueue;
}

Figure 3. Cash Contract Listing in ACL