Bulletin of the *Transilvania* University of Braşov • Vol 7(56), No. 1 - 2014
Series III: Mathematics, Informatics, Physics, 131-138

SOFTWARE CORRECTNESS VERIFICATION BY CONTRACT

Corina - Ştefania NĂNĂU[1]

Abstract

Scenarios are ways to reflect the daily activity of a software system. In the life cycle of such a system, scenarios occur at different levels. One of their utilities is to facilitate verification of the correctness of the application functionality. This article presents a method for checking the specifications in case of component-oriented applications as verification is an important issue of the life cycle of the software application.

Key words: specification, design by contract, scenario, software component, state, action

1 Introduction

Component - oriented software development and verification of software are two sub-disciplines of software engineering, and even more, the action of checking the software is part of the life cycle of an application. Component - oriented programming is a way to create new applications based on other independent pieces of existing applications, called components.

To make a correct assessment of the implementation of a software application - whether based on components, objects or aspects - there are checks of contract's methods of application that are called in a scenario at runtime. Moreover, in aspect oriented programming issues, scenarios (preconditions, postconditions and invariants) can be injected into the application as aspects.

The next sections will present some general details on what specifications, scenarios, states and operations of a system mean, and in the last part of this article a case study outlining the previous theoretical questions is presented. The case study will be to achieve a component - based application on the activity at a petrol pump. It will identify the application components, the connections between these interfaces, the actions each component performs for carrying out some specific activity scenarios of the system and it will also identify the states of each component, as a result of performing the various operations. Graphical representation of the components will be made according to UML 2.0 [6] standards.

[1]Faculty of Mathematics and Informatics, *Transilvania* University of Braşov, Romania, e-mail:
cory2512@yahoo.com

2 What do specifications mean?

A specification generally indicates what a software system does or should do, but not how to do it. If we are dealing with the specifications in object oriented programming, these are actually represented by preconditions, postconditions and invariants. In case of components, we can attach invariants to the interface that specifies properties of objects that implement the interface. Therefore, the preconditions offer the contractual requirements of the interface client (called "caller") and the postconditions offer the corresponding contractual requirements of the operation provider, namely the component instance that implements the interface [1]. The contract based design extends the usual definition of abstract data types preconditions, postconditions and invariants. These specifications are called *contracts* according to a conceptual metaphor of the conditions and obligations for service contracts. The contract summarizes how the specifications cooperate with elements of a software system based on mutual obligations and benefits.

A specification of a component interface should include a few ways to describe the interface behavior. There are specified conditions that help in a correct use and implementation of interface modules.

Design by contract is a methodology that allows the addition of semantic information to the application interface by specifying monitored assertions about the status of the program at runtime [4]. These assertions form the contract (or specification) of an application.

We can say that there are two types of contracts: *usage contracts* and *realization contracts* [5]. The one implementing a client application using one component uses a *usage contract* (a contract between the objects of component interfaces and the client). The object of each instance of a component is created on the basis of installed components; it is an object with its own unique identity and data and provides the behavior implemented. An installed component may have one or more component objects. A usage contract is a running contract and it is defined by an interface specification. Interface specifications actually contain operations or methods provided by the interface and may also contain pre-and post conditions. *The realization contract* is defined between component specifications and implementation. This is a contract at design level and it is used by the one constructing a component following its specification. Component specifications are primarily interface groups that may also contain constraints on how their interfaces implementation is made and how the implementation will have to interact with other components. This type of information is not important for the client, and as long as such information is not part of interface specifications, the customer has no reason to know about their existence. If the interface specifications contain an information model and a component can support two or more interfaces, the elements of this information model of one of the interfaces may correspond mostly with the information elements of other interfaces supported and they must always have the same values. It is important to specify constraints between the information models of the interfaces as part of component specifications. These, together with interface specifications supported, represent the realization contract.

To ensure that an application works according to the specifications, different ways of verifying the desired application contract code are available. Correct execution of a program can only be seen by checking the contract code, but an automatic check of the contract during program execution would be made.

One way to check the contracts of programs written in Java is to use jContractor library that allows adding and checking contracts in the form of methods, as follows [4]: for each method we want to specify we will have to write two methods - one for preconditions and one for postconditions. These methods will be written in Java. When checking the contract of such a method, the invariant method and the precondition method will be called before the method body, and after the method itself execution, the postcondition method and again the invariant method will be called. jContractor provides support for the specification of methods rewritten in a specified chain of inheritance or interface implementation methods.

3 What do scenario mean?

In general, a scenario represents predictable interactions between users of the system and the system itself. Scenarios include information about expectations, motivations, actions and reactions of a system [7]. Scenarios attempt to reflect how the system is used in the context of its current activity. These are frequently used as part of the system development process and are often written in plain language with minimal technical details so that those involved in the process can have a common example on which to focus their discussions. Moreover, scenarios are used to describe the behavior we want the system to have, replacing or complementing the traditional "*functional requirements*". In case of "*agile*" methodologies, scripts are actually represented by the user stories, and for normal development of software, scenarios are written using structured use cases.

Some ways in which scenarios can be used [7]:

- as viewing parts, providing a clearer picture of the system or product concerned
- shows the advantages offered by the system

Usually when we talk about scenarios, we have to do with model programs. The model - based development and the use cases - based development have inspired the proposal of a variety of software engineering approaches that synthesize state - based models from scenario - based models. These models describe the dynamic behavior of reactive systems.

A *scenario based model* represents the behaviors "inside the objects", behaviors described on the basis of interactions between multiple objects. In contrast to this modality, a *state - based model* is often used to represent the entire behavior of a reactive system. This behavior can be exemplified by a global state machine or a set of communicating state machines where each state machine describes the complete behavior "inside the object".

Scenario - based modelling and state - based modelling offer two different views on reactive systems. Scenarios are used both to help to achieve the functional requirements and for understanding and validating these requirements. Thus, for processing the requirements, scenario - based models are much more useful, thus the participants in a project can communicate more easily with customers. On the other hand, the code can be generated automatically from state - based models, because those who design the software application believe that state - based models are closer to the design and implementation. Actually, scripts can be used to control the entire lifecycle of the software development process [3]. In other words, scenarios are useful not only in the requirements analysis phase, but also in the design and implementation phases.

4 Abstract state machine, actions and states

An abstract state machine is a formal way of describing the algorithm steps. This offers a mathematical vision of the software state. Using the abstract state machine the software system model can be achieved.

The abstract state machine is characterized by the fact that it provides a simple and practical framework where the system engineer can adopt a "divide et impera" way of working. This method offers simplicity and flexibility in choosing the combination of concepts, the appropriate notation and technique for every problem in part, these being integrated into a framework with an uniform mathematical background.

If the system is component - oriented, the abstract state machine describes the correct usage scenarios. Based on the abstract state machine, a diagram of the application scenarios can be built. This diagram may represent the starting point in building the application model.

The state represents the information stored in the program or system at a certain time. Each program or system begins with an initial state. It is usually an empty or inactive state. The behavior continues until the system reaches an acceptance state. An acceptance state is a state in which the program objectives were achieved, where some work units have been completed. The system may stop in a state of acceptance. Alternatively it can continue working from a new work point. A sequence of actions that begins in the initial state and ends in accepted state is called *run* or *trace*. A run should not end in a state that is not an acceptance state. If the model program does not identify any acceptance state, the run can end in any state. Runs (executions) of a model program are sequences of method calls.

The actions are units of the system behavior. According to [2], an *action* may be composed of several small activities, the action itself being atomic, ie: once it starts, it completely runs without being interrupted or replaced by another action. For each type of action of implementation there is a method in the model program. When running the model program, each call method is an action for implementation. The relations between implementation actions and methods of

the model program are always one to one.

Scenario diagrams or sequence diagrams describe the interactions that occur between the objects of the described system (if our system is object - oriented) or between its components (if the system is component - oriented).

5 Case study

To illustrate how scripts are specified to a component - based application we have chosen as a case study the supply at a petrol pump.

The first step is the preparation of the application whose scenarios we wish to specify, more precisely it consists in carrying out the component diagram. This gives the application designers an easy to understand format on which the solution modelling can begin. The component diagram is also a very useful communication tool for different categories of participants in the project: clients, designers, developers, system administrators.

In UML 2, the notation for representing a component is consistent with Fig. 1. In graphical representation of a component the provided and the re-

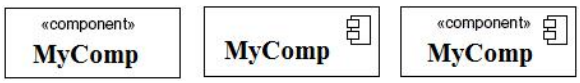

Figure 1: Ways of graphical representation of a component

quested component interfaces may appear too, as it shows in Fig. 2.

As we have communicated the basics for achieving the component diagram

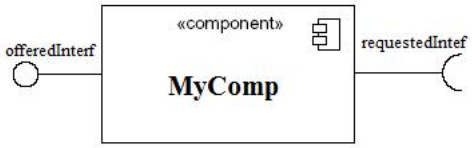

Figure 2: Graphical representation of a component with its requestes and provided interface

using UML 2, we can now create the diagram itself for the application chosen as case study. Fig. 3 shows the existence of three components: Client, PetrolPump and PayOffice. The Client component provides interfaces TankInterf and MoneyInterf and has as required interface FuelInterf. The latter, in turn, is the provided interface for the PetrolPump component, which has as a required interface RefuelInterf. CashInterf interface is required by the PayOffice component and provided by the Client component.

Entire system (gas station) states: open or closed

Client states: waiting, pending supply, pending payment or free

Gas pump states: active - with sub-states *available* or *occupied* - or inactive

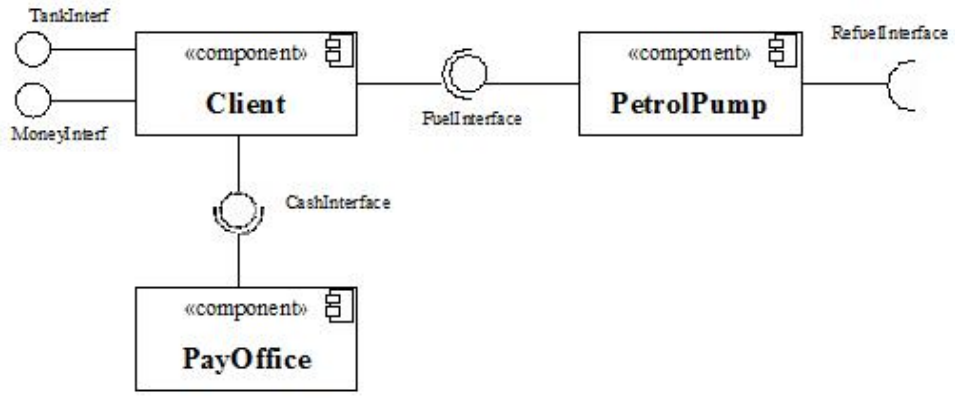

Figure 3: Component diagram of the application

Cash register states: available, free or occupied

Operations performed by the Client component: fuel supply, pay

Operations performed by the PetrolPump component: fuel tanks supply

Operations performed by the treasurer (represented by PayOffice component): collects money

The baseline scenario offered by the system is composed of the following steps: the customer sits at a pump, supplies successfully, pays successfully and is free. Of course, there may be several scenarios: the pump may not have petrol; the client might not have enough money on him, etc.

Find below specifications of operations performed by each component. Verification of system correctness consists in checking these specifications. For the *supply operation* executed by the client to be successful, the preconditions to be met are the following: the tank must have enough free space, there must be petrol in the pump, the amount of petrol removed from the pump must have a positive value. The postcondition in case of success is the fact that the client has successfully supplied the tank, meaning that he has extracted from the tank an amount of gasoline less or equal to the amount of petrol existing in the pump, otherwise, the client fails to supply (for non - compliance of at least one precondition). The invariant checked before and after implementation of any method is the system state check (the gas station should be open). Only in this case can operations be made. For the *payment transactions* to run successfully, the precondition is to verify that the customer has enough money on him, the postcondition means the client gets to the final state: free. For the *supply operation* executed by the component PetrolPump, the verified preconditions are that the pump be active and contain petrol and the postcondition is that the client can manage to fill the tank. For *money collecting operation*, the preconditions are that the customer must have enough money on him and have successfully achieved the supply operation, and the post-condition transfers the client to the final state - free - or in an unknown state (in case the prerequisite is not met).

In TankInterf interface provided by Client component we will write a *refuel*

method, with its specifications (the application will be written in Java, using jContractor for the specifications):

```
procedure REFUEL(fuelAmount)
    if ( thenpump.Available = true)
        pump.Content ← pump.Content + fuelAmount;
        tank.Content ← tank.Content + fuelAmount;
    end if;
end procedure;

function REFUEL_PRECONDITION(fuelAmount)
    ok ← true;
    if ( thenfuelAmount < 0)
        ok ← false;
    end if;
    if ( thentank.Size < tank.Content)
        ok ← false;
    end if;
    if ( thenpump.Content < 0)
        ok ← false;
    end if;
    return ok;
end function;

function REFUEL_PRECONDITION(fuelAmount)
    //it will be checked if the fuel amount extracted from the pump
    //is less or equal with the amount of fuel existent there
    ok ← false;
    if ( thenpump.Content < fuelAmount)
        ok ← true;
    end if;
    return ok;
end function;
```

6 Conclusions

One of the most important goals of this article is to highlight how the implementation of contracts (specifications) of an application and the verification of requirements that make up these contracts to obtain an application that operates as required. This highlighting is based on a very short example of an application method specification. Application was written in Java and the contract was written according to jContractor standards.

References

[1] Brger, E., Strk, R.F., *Abstract State Machines. A Method for High-Level System Design and Analysis*, Springer, 2003.

[2] Jacky, J., Veanes, M., Campbell, C., Schulte, W., *Model - Based Software Testing and Analysis with C#*,Cambridge University Press, 2008.

[3] Jacobson, I., Ng., P., *Aspect - Oriented Software Development with Use Cases*, Addison Wesley, Professional, 1st edition, 2004.

[4] Karaorman, M., Abercrombie, P., emphjContractor: Introducing Design-by-Contract to Java Using Reflective Bytecode Instrumentation, Formal Methods in System Design, Springer Science + Business Media, Inc. Manufactured in The Netherlands, 275-312, 2005.

[5] Nyholm, C., *Designing Component-Based System with UML Contract Specifications*, Mlardalen University, The Department for Computer Science and Engineering, 2002.

[6] Rumbaugh, J., Jacobson, I., Booch, G., *The Unified Modeling Language Reference Manual*, Second Edition, Addison-Wesley, 2005.

[7] http://en.wikipedia.org/wiki/Scenario_(computing).