Proceedings of the Operational Research Society Simulation Workshop 2014 (SW14)
B.Tjahjono, C. Heavey, S. Onggo, and D-J. van der Zee, eds.

AN INVESTIGATION ON TEST DRIVEN DISCRETE EVENT SIMULATION

Shahriar Asta, Ender Özcan, and Peer-Olaf Siebers

University of Nottingham
School of Computer Science
Nottingham, NG8 1BB, UK
{sba,exo,pos}@cs.nott.ac.uk

ABSTRACT

This paper deals with the application of modern software development tools on simulation development. Recently, Agile Software Development (ASD) methods enjoy an increasing popularity. eXtreme Programming (XP) techniques, one of the techniques which belong to the ASD group of methods is a software development method which improves software quality and responsiveness of software projects through introducing short development cycles and a Test Driven Development (TDD) philosophy throughout the development. In this paper, we particularly pay attention to the application of the TDD by approaching simulation development from a test-first perspective. This study consists of a feasibility study of applying the TDD technique in simulation development in its various levels, say, acceptance and unit testing. Moreover, a simulation case study of a surgical ward has been considered, designed and implemented using the AnyLogic simulation toolkit. Our study differs from the mainstream in many ways. It addresses the feasibility of Test-Driven Simulation Development in Visual Interactive Modelling and Simulation (VIMS) environments as well as providing an insight into how the test-first concept can further help with the choice of components and acceptance testing.

Keywords: Test-Driven Development, Discrete Event Simulation, Extreme Programming, Modelling

1 INTRODUCTION

Agile Software Development (ASD) has been a recent revolution in program design. ASD combines a set of software development methods and is based on iterative and incremental improvement (Beck et al 2001). Many modern software development methods fall into the ASD category, one of which is the eXtreme Programming (XP). XP itself consists of short development cycles where at the end of each cycle the software product is *released* with the goal of absorbing new customer requirements, resulting in an adaptive development cycle. XP is consisted of various elements such as pair programming, various code review methods as well as unit testing. Unit testing in XP is particularly interesting considering that XP intends to promote the software product quality by employing a test-first concept which gives rise to Test Driven Development (TDD) methodology (Beck 2002). TDD requires a programmer to take a different view of the system by writing (and thinking about) the test cases before writing the actual unit code. Instead of being a method to test the software, TDD is the methodology of helping the developer and customer with specifying "*unambiguous requirements*" which are expressed in form of tests (Newkirk 2004). The TDD development cycle is shown in Figure 1. As well as being heavily utilized in industrial scale (Bhat and Nagappan 2006; Larman and Basili 2003; Gelperin and Hetzel 1987), TDD has been the subject of numerous research publications (Newkirk 2004; Erdogmus et al 2005; Müller and Padberg 2003). Simulation development has no major difference with developing any other software

product and therefore developing techniques such as XP or TDD could be considered as tools which secure some level of accuracy in the implementation phase of simulation development.

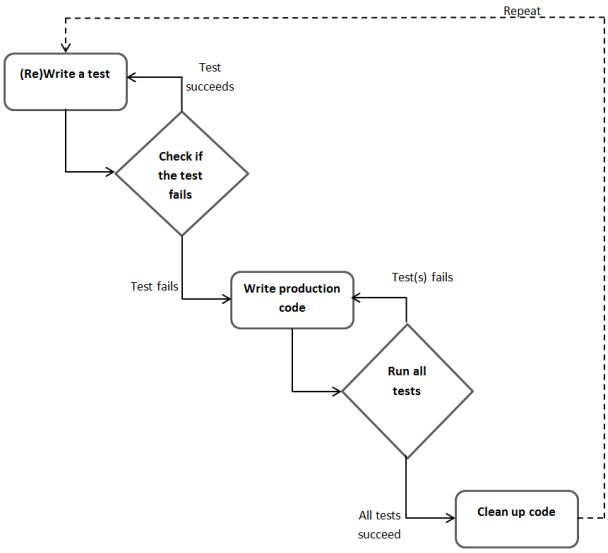

Figure 1 *Test-Driven Development Cycle*

In this study, we are particularly interested in the application of the TDD in Discrete Event Simulation (DES) on micro/macro levels. To this effect, we utilize a multi-paradigm visual interactive modelling and simulation (VIMS) environment, AnyLogic[1]. To the best of our knowledge this kind of approach has never been applied to developing Operations Research component based simulation models where researchers/practitioners often work with predefined components (design patterns) rather than coding methods.

We have conducted a feasibility study to investigate the opportunities of Test-Driven Simulation Development (for DES) within a VIMS environment. In our feasibility study, we simulate a surgical ward where the patients should spend no more than 30 minutes within the system. Our results show that it is possible to apply TDD to DES development in VIMS environment (combining coding the tests with the use of modelling elements in form of design patterns). Currently TDD deals with verification (comparing the conceptual model in form of requirements to the implemented model in terms of code) while the task description requires some form of validation (comparing the conceptual model with the real world). The rest of the paper is organized as the following. In Section 2 a brief survey of the researches conducted in this area is given followed by a description of our toolkit of choice (AnyLogic) in Section 3. Section 4, describes the case study (the NHS surgical ward problem) along with a detailed account of the application of TDD in the implementation phase of the simulation. Finally, conclusions are provided in Section 5.

2 RELATED WORK

This section provides the reader with a brief literature survey on the use of TDD concept in simulation. This research area is considerably new and the volume of work which can be referred to directly is scarce. One of the early studies on this topic was conducted in (Kleijnen 1995). In this study, the verification phase is divided into four steps: i) general good programming skills, ii) controlling intermediate simulation output by tracing, iii) comparing final output of the simulation with analytical results and iv)

[1] http://www.anylogic.com/

animation. The first step which was briefly described in (Kleijnen 1995) falls in the scope of our study. The author recommends a module-by-module testing of the simulation software development phase instead of *spaghetti* programming and point out that code analysts should *divide-and-conquer* the code verification process. Collier et al (2007), propose a TDD approach towards developing simulation. In this work, relying on the declarative structure of the simulation modelling toolkit that the authors have employed for their study, they show that a TDD approach is quite feasible. The toolkit in use is Repast[2] and it is argued that the reason why a TDD approach is possible is the fact that the toolkit maintains a strict separation between model and simulation concerns. That is, the coding can be performed isolated from the framework. An example of writing a code for the game Iterated Prisoner's Dilemma is given where a TDD approach has been taken to develop the model.

Zhang (2004), proposes the application of Test Driven Modelling to enhance the Model Driven Development process. The ideas from Test Driven Development are transformed and adapted for simulation modelling purposes. Our approach differs to the one proposed by Zhang (2004) due to the fact that instead of transferring the TDD ideas and adapt them for simulation modelling, we take a rather more direct approach by implementing the actual test units first instead of creating test models and replace the test units with them. Dietrich et al (2010), introduces a Model-Driven Simulation based on UML models. The unit testing method has been applied on UML simulation models, taking unit testing a level higher to the level of simulation models described in UML. The availability of automatic UML model transforming tools has reportedly enabled the authors to propose the idea. A new testing method is proposed in which UML is employed to specify model-level unit tests in order to validate simulation models defined with UML. The translation, execution and evaluation of the unit tests is described within the framework Syntony (Syntony is a standardized graphical modelling language, based on UML 2 which is proposed by Dietrich et al 2007). The proposed method allows the compilation and execution of the test code in combination with the simulation code. It has been shown that the proposed method provides a fully functional unit testing framework where the modeler relies on automated test case generation and execution.

3 THE ANYLOGIC TOOLKIT

AnyLogic is a recent and well known simulation toolkit. This toolkit consists of several libraries of objects which covers a wide spectrum of components necessary for designing different kinds of simulation paradigms (system dynamics, discrete event, and agent based). In AnyLogic, a model can be represented graphically while some Java code snippet can be added to implement the objects, component interactions and dynamics of the system. Though the implementation and modelling are not totally separate as it is the case with the Repast toolkit, the Anylogic toolkit is equipped with debugging and tracing utilities. An API library is also provided for users who wish to take a more advanced programming approach. Essentially, what the toolkit does is to assist the modeller to *write the minimum amount of Java code*. Based on these observations, and due to the fact that Anylogic follows an object oriented approach, one can easily conclude that TDD is feasible within the Anylogic framework. However, this claim is yet to be validated.

Since Anylogic is basically a graphical modelling environment, applying model-level unit tests as proposed by Dietrich et al 2010, seems to be well promising. Dietrich et al 2010 proposes to incrementally construct a simulation model by implementing model-level unit tests within the UML. However, it has been assumed that the underlying framework has the ability to automatically convert the UML model into executable code. But, utilization of this method, puts the development process into a new category, say, Model Driven Development (MDD). In this study, we apply TDD method for implementing a simulation model instead of deriving it. AnyLogic model is a tree of active objects encapsulating each other. The root of the tree is called the main object. The main object represents the highest abstraction level of the model. It embodies *Embedded Objects* (components), functions, variables

[2] http://repast.sourceforge.net/

and user defined objects. There are various libraries within the AnyLogic IDE which provide a wide range of components. We specially use the Enterprise Library as it contains the objects necessary to create simulation models. Each component comes with a set of events. For instance, events such as `onEnter`, `onExit` are triggered when an entity enters or leaves the component respectively. Each event has a field which gives the designer the opportunity to write a piece of code to determine the necessary action to be taken when the event occurs. This can be a single line of code (like increasing a counter), a function call or a class instantiation.

Each component processes an entity. The user can define an entity class with various attributes and associate it to components. However, AnyLogic associates a default entity class (with no extra attributes) to each component. Indeed, AnyLogic provides various variables of a component with meaningful default values. This enables the users to use the components in a plug-and-play fashion without initializing them.

4 A CASE STUDY: NHS PROBLEM

4.1 Problem Description

The average waiting time in various wards within the National Health System (NHS) is nowadays a critical and much argued topic. In this problem domain, DES is a commonly used tool to help finding alternative solutions that (at least in principle) allow to maintain service standards while keeping costs low. For our case study we have chosen to build a model of a (fictitious) NHS surgical ward that could be used for improving the operation of the ward.

The Case: The NHS has recently opened a new walk-in centre. There have been several complaints about long waiting times for non-urgent patients in the surgical ward (which is part of the walk-in centre). In response to the complaints the surgical ward management has set itself the target to improve the service so that the overall time from arriving at the ward to the beginning of the treatment does not exceed 30 minutes for at least 95% of non-urgent patients. There are always at least 5 doctors present in the ward which has a capacity of 8 operating rooms. In order to achieve the targets regarding non-urgent patients, the surgical ward management is happy to employ additional doctors whenever necessary. Once patients arrive at the entrance of the surgical ward they have to register with the front desk. Then they can move to the waiting area. The surgical ward has to deal with non-urgent and urgent cases. If urgent cases arrive they will be allowed to jump the queue in the waiting area and will be treated as soon as a doctor is available (we assume that only one doctor is required for treating each patient). There is no problem with the availability of nurses to assist the doctors. In this study, we can safely assume that there are a sufficient number of nurses on the surgical ward to support any solution.

4.2 Simulation Implementation Using TDD

One of the objectives of DES is to keep track of the statistics of the system under simulation. That is, we are interested in tracking the changes of the system status, triggered by occurrence of an event, right from the moment when an entity enters the system, until the moment it leaves the system (Nance 1993; Shannon 1998). Based on this definition and rephrasing the NHS ward example accordingly, an event is triggering the arrival of a new patient and the statistic that we are interested in collecting the average waiting time of many such patients (events). Thus, this definition suggests that the two major components we should consider as the starting point in system modelling are events, entities and systems entry and exit. We start with the entity, without which having a system is pointless.

It is obvious that an entity (a patient in our case study) needs to be represented as an object (a class). Based on TDD approach, the first thing that needs to be done is to design a test. This test can be in form of a test class or a test function. Judging by the simplicity of our case study, a test function is chosen to be

written instead of a test class. The test function `testPatientClass` (located in the main class) can be seen in Program 1. It tests the `patientClass` (which currently does not exist because it is not written yet) by checking the registration and urgency status of the patient. The type of the variable `patient` in Program 1 is `patientClass`. Please note that, this is our strategy throughout the paper that prior to implementing a class a test function or a test class is written first. As a standard, the name of this test function/class is the same name for the intended class with an additional *test* prefix.

Running the simulator produces an error due to the fact that first, the class `patientClass` does not exist and boolean parameters `isUrgent` and `isRegistered` are not available. In order to resolve the issue the class is designed with the two boolean variables mentioned above. This time simulator is run without errors. The `patientClass` in its current form can be seen in Program 2. From this point onward, the input/output of every component which is added to the model is set to the instance of the `patientClass` instead of the default value which is `Entity`. In AnyLogic, the input/output of a component determines the type of the entity which can be a user defined class or the default type `Entity`. Of course, in case this is forgotten, running the simulation will result in run-time errors.

As discussed earlier, based on the DES definition, we need to consider the second component. Entrance and exit points of the system are marking points in which the system status changes. That is why, in the next step, we realize that the system requires a source (system entry) and a sink (system exit) where the number of incoming customers has to be equal to the number of the customers leaving the system. This constraint is again inferred from the DES definition where it highlights the fact that an entity which enters the system is bound to leave it at some point. On the other hand TDD spirit suggests that given a constraint (or any other design component for that matter) a test has to be written first. Thus, a test function located in the main class is written (Program 3), which examines this constraint. The test, namely `test_inOutMatch` calls the function `inOutMatch` which checks for a match between the number of customers in and out the system. Calling the test results in error, obviously because the function `inOutMatch` does not exist. This function is then written as in Program 4. Again, running the simulation results in an error since there is no source and sink components in the system. Consequently, a source and a sink are added from the Enterprise library and connected to each other. This time the simulator runs smoothly.

Program 1 testPatientClass

```
boolean ir = patient.isRegistered;
boolean iu = patient.isUrgent;
return ir&&ru;
```

Program 2 patientClass

```
public class patientClass
{
        public boolean isUrgent;
        public boolean isRegistered;

    public patientClass()
    {
        isUrgent = false;
        isRegistered = false;
    }
    @Override
    public String toString()
    {
        return super.toString();
    }
}
```

Program 3 test_inOutMatch

```
if (inOutMatch() == false)
        error("in-out count does not match");
```

Program 4 inOutMatch

```
if (source.count() == sink.count())
        return true;
else
        return false;
```

Now that the simulation model is initialized through the addition of a source and sink, we can move on to the next step, that is, measuring the *time in the system* of all the patients. This is a hard constraint and we always need to check it to see if it exceeds the 30 minutes (0.5 hour) threshold for 0.95% of the patients in the system. This is why the test function `testOverwaiting` (Program 5) is written. and called in the *on enter* event of the sink component. `testOverwaiting` is the acceptance test. The

validity of the entire system depends on passing this test. Running the simulation produce an error since there is no time measurement component called `tmEnd` in the system. Running the simulator produces yet another error. This time the reason is that there is no component which indicates the start of the time measurement. Such a component (`tmStart`) is added subsequently. The model in its current form is shown in Figure 4.

The very existence of a registration desk suggests that the subsequent departments/units within the surgical ward require some initial information from a patient which should be acquired during the registration process. That information includes specifics, such as, whether a patient is urgent or not. Thus, in this step we add the registration desk to our model and write some tests to check the registration of a patient as well as his/her urgency. The registration desk consists of a service component connected to time tracking components from both ends. A third connection is also provided by which the registration desk is connected to a resource unit, say, the registrar. What we need to make sure is to assert that every patient is unregistered prior to entering the registration desk and otherwise after leaving the desk. Thus two test functions are written in the main body of the simulation project to check such conditions (Program 6 and Program 7). Please note that the first test (`testDeskEnteringPatient`) is checked in the *on enter* event of the registration desk service component while the second test (`testDeskLeavingPatient`) is checked in the *on exit* event.

Program 5 testOverwaiting

```
int overwaitingCounter = 0;
int totalCounter = 0;
double sum = 0;
for(int i=0;i<tmEnd.dataset.size();i++)
{
        if(tmEnd.dataset.getX(i) > 0.5)
                overwaitingCounter ++;
        totalCounter ++;
}

if ((double)overwaitingCounter/
    (double)totalCounter > 0.95)
{
        error("over waiting !!!");
        return false;
}
return true;
```

Program 6 testDeskEnteringPatient

```
if (patient.isRegistered() == true)
{
        error("A patient is registered
               without attending the
               registration desk!");
        return false;
}
return true;
```

Program 7 testDeskLeavingPatient

```
if (patient.isRegistered() == false)
{
        error("A patient is leaving
               the registration desk
               unregistered!");
        return false;
}
return true;
```

Program 8 inOutMatch

```
int s=0;
s+=registrationDesk.queueSize();
s+=registrationDesk.delaySize();
s+=sink.count();
if (source.count() == s)
        return true;
else
        return false;
```

Running the simulation results in an error which is triggered in the second test (Program 7). The reason is that while serving the patients at the registration desk, their registration status is not changed. Therefore the necessary code is written to pass the test. That is, the code line `patient.registered=true;` is added to the *On enter delay* event of the `registrationDesk`. Running the simulation once more triggers another error in the function `test_inOutMatch`. The reason being that, the service component introduces delay to the system by putting some entities in its internal queue. Hence the number of entities in the sink is always less or equal to that of the source. At

this stage we need to amend the `inOutMatch` to pass the `test_inOutMatch` (Program 8). After this extension to our test function, the simulation runs without producing any errors (Figure 5).

In the next step, we need to check if the patients are split according to the urgency of their cases. Whether a patient is in need of urgent attention or not is decided upon by the registration desk. While urgencies are dealt with without a delay, non-urgent patients need to wait in the ward. A queue presents this waiting area for non-urgent patients. This is why a queue should be added to the model. However, prior to adding the queue, a selector (`urgencySelector`) is needed to split the patients into urgent and non-urgent categories. At this point, based on our TDD approach, we implement a test in which proper checks are made to verify the existence of the components as well as their functionality. The first test, `testSelector`, would be regarding the first component we intend to add, that is the `urgencySelector`. It simply checks whether the selector distributes the entities according to the stated requirements (Program 9). Running the simulator triggers an error (via `testSelector`) since we have not added the selector component. Thus, in order to pass the test we add this component (`urgencySelector`) and set its probability to 0.05. The `true` port of the selector corresponds to urgent cases whereas the `false` port corresponds to non-urgent ones. Both ports are connected to the `tmEnd` component. The probability 0.05 addresses the probability of entities which are assigned to the `true` port (urgent patients). Running the simulator at this point does not generate any errors.

The next entity is the waiting queue (`waitingArea`) for which we need to write a test. However, this time, instead of writing a new test, we decide to extend an existing test function `test_inOutMatch`. This test function relies on the function `inOutMatch` which is now modified as in Program 10 and includes the number of entities in the queue and the delay service of the `waitingArea`. Running the simulator produces an expected error, notifying that the component `waitingArea` does not exist. This component is added between the `false` port of the `urgencySelector` and the `tmEnd` component. Figure 6 shows the latest modifications on the model.

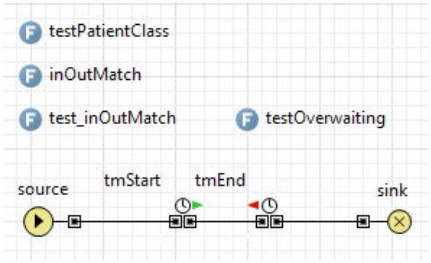

Figure 4 *Addition of time tracking components*

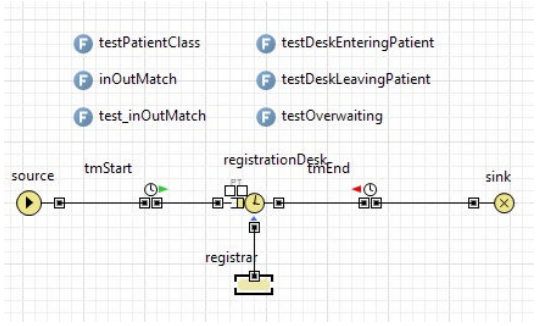

Figure 5 *The registration desk*

In our next step, we need to proceed with the implementation by adding a resource component to the system which represents the available doctors by which both urgent and non-urgent patients are served. However, the urgent patients always have higher priority than the non-urgent patients. That is why a secondary queue which is a priority queue (`priorityQueue`) should be added to the model. To implement a test for this new component, once again the `inOutMatch` function is modified to test the functionality of the new component by adding the number of entities in the `priorityQueue` to the overall number of entities within the system. Running the simulator produces an error indicating the absence of the `priorityQueue` component which is added subsequently and placed before the `tmEnd` component. The priority of each entity is set to be the value of the Boolean variable `isUrgent`. Subsequently, and according to the problem description where we initially have 5 doctors and 8 rooms, the service component (`ward`) should be added to the model. Similar to the process of adding the `waitingArea` and `priorityQueue` components, we extend the `inOutMatch` function to test the

new component (Program 11). After running the simulation and receiving the error, notifying the absence of the component we add the `ward` component between `priorityQueue` and `tmEnd` (Figure 7).

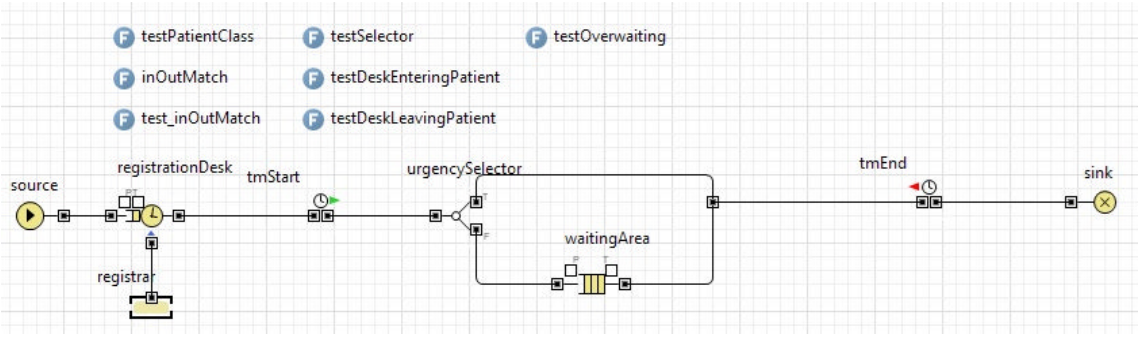

Figure 6 *Adding the urgency selector and the waiting area to the model*

Running the simulation produces a very interesting error. The error is generated by the test function `testOverwaiting`. The reason is that the time measurement components are placed in the wrong place where they also cover the time spent by the patient while he/she is being attended. That duration certainly should not be included in the waiting time. Hence, the `tmEnd` component which measures the time a patient leaves the waiting area is shifted leftwards and placed before the `ward` service component. Running the simulation after these modifications does not produce any error any more. Figure 8 shows the final model. Although our case study in this paper consists of a small DES project, one can easily note that the process is simple and rather repetitive. Once a set of *standard* unit tests are established for each component of a DES system, one can automatically run the *standard* unit tests prior to adding a component to the model. For instance, it is clear that `test_inOutMatch` is necessary for a pair of source and sink components regardless of the conceptual model.

Program 10 inOutMatch

```
int s=0;
s+=registrationDesk.queueSize();
s+=registrationDesk.delaySize();
s+=waitingArea.size();
s+=sink.count();
if (source.count() == s)
        return true;
else
        return false;
```

Program 9 testSelector

```
p=urgencySelector.probability(entity);
if (p != 0.05)
        error("wrong probability!");
```

Program 11 inOutMatch

```
int s=0;
s+=registrationDesk.queueSize();
s+=registrationDesk.delaySize();
s+=waitingArea.size();
s+=priorityQueue.size();
s+=ward.queueSize();
s+=ward.delaySize();
s+=sink.count();
if (source.count() == s)
        return true;
else
        return false;
```

5 DISCUSSIONS AND CONCLUSION

In this paper, we have demonstrated that a TDD approach can be used to develop and implement a DES model. We have described the development process step by step using a case study. Our approach can be summarized in the following guideline:

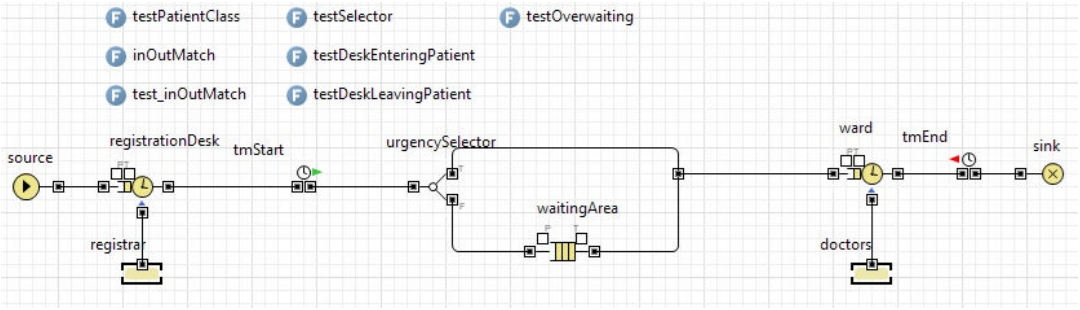

Figure 7 *The tmEnd component is in a wrong place causing testOverwaiting test function to fail.*

1. Pick a requirement, i.e., a story
2. Write a test case that will verify a small part of the story and assign a fail verdict to it
3. Write the code that implement particular part of the story to pass the test
4. Execute all tests
5. Rework on the code, and test the code until all tests pass
6. Repeat step 2 to step 5 until the story is fully implemented

There is some evidence that while developing a DES in a VIMS environment using a TDD approach, one can extract test patterns specific to the components that the project requires. Each DES component (such as sink, source, queue etc.) can be associated to a group of unit tests that should be called prior to adding the component to the project. While we do not necessarily assume an identical identity for programmers and modellers, the test patterns will help both groups to develop DES models faster and in a more reliable fashion. Furthermore, such test patterns help in employing the TDD approach in VIMS environments such as Simul8 and Witness where much less coding effort is foreseen.

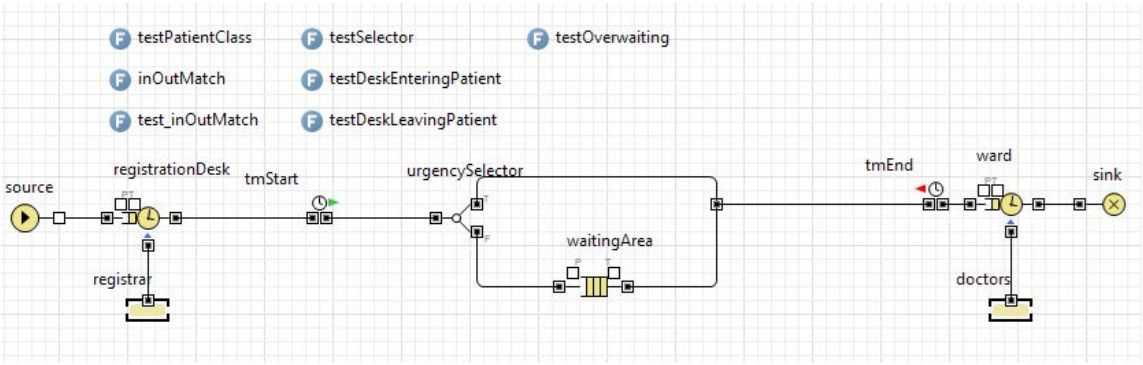

Figure 8 *The final system*

We have employed the TDD approach at two levels, namely unit testing and acceptance testing. We observed that applying TDD at unit testing level in relation to DES is straight forward, but not for acceptance testing. Considering the case study we have used, while the final model (see Figure 8) runs smoothly without producing any immediate error, the test function `testOverwaiting` may still generate an error during a longer run. This is due to the fundamental difference between developing a simulation model and an ordinary software model. In the latter case, the objective is to develop a product which is supposed to function perfect. However, in case of developing simulation models, the objective is to design a product where imperfections of the system under study are outlined.

Moreover, prior to constructing the model, no attempts were made to acquire information regarding the probability distribution of the arrival of the patients, the attendance duration and etc. This does not mean that no such distribution was used in the model constructed above. Such distributions exist and AnyLogic's default distributions (along with their default parameters) were used without any change. The reason for such an approach stems from our test first method. In TDD, minimal coding is essential to the success of the approach. Making an analogy, we decided to extend that idea for the Test Driven Simulation by employing minimal data/knowledge. The purpose is to check if the available test can help to expose the missing data/knowledge of the system. As it happens, it does indeed. It is interesting to point out that this test was actually the second test written for the system and when it was written the only components of the system were the source and the sink. It is also interesting to emphasize the fact that when one tries to resolve the issue, model modification and model parameters (instead of coding) is required to make the test pass. In other words, instead of writing minimal code to make the test pass, providing an additional, though minimal, knowledge about the system is essential to make the test pass. Evidently, it is possible to take our approach to a higher level of abstraction where the unit tests contribute to the specification of model parameters. If the simulation model is given sufficient time, it forces the designer to introduce some additional parameters (minimal knowledge to make the test pass) with which the simulation passes the test.

As a result, we have presented a method to apply the TDD philosophy to simulation development. Whether or not the discussions above hold in general is a part of what we intend to investigate further as a future work.

REFERENCES

Beck K, Beedle M, Van Bennekum A, Cockburn A, Cunningham W, Fowler M, Grenning J, Highsmith J, Hunt A., Jeffries R, Kern J, Marick B, Martin R C, Mellor S, Schwaber K, Sutherland J and Thomas D (2001). Manifesto for Agile Software Development.

Beck K (2002). Test Driven Development: By Example. *Addison-Wesley Longman Publishing Co., Inc.*, Boston, MA, USA. 2002.

Bhat T and Nagappan N (2006). Evaluating the efficacy of test-driven development: industrial case studies. *In Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering (ISESE '06)*. ACM, New York, NY, USA, pp.356-363.

Borshchev A, Simulation Modeling with AnyLogic: Agent Based, Discrete Event and System Dynamics Methods.

Collier N T, Howe T R, and North M J (2007). Test-Driven Simulation Development using Repast Simphony, *Proceedings of the North American Association for Computational Social and Organizational Science (NAACSOS) 2007 Annual Conference*, Emory University, Atlanta, GA USA.

Dietrich I, Dressler F, Schmitt V and German R (2007). SYNTONY: network protocol simulation based on standard-conform UML 2 models. *Paper presented at the meeting of the VALUETOOLS.*

Dietrich I, Dressler F, Dulz W, and German R. (2010). Validating UML simulation models with model-level unit tests. *In Proceedings of the 3rd International ICST Conference on Simulation Tools and Techniques (SIMUTools '10)*. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), ICST, Brussels, Belgium, Belgium, , Article 66 , 9 pages.

Erdogmus H, Morisio M and Torchiano M (2005). On the effectiveness of the test-first approach to programming, *IEEE Transactions on Software Engineering*, vol.31, no.3, pp.226-237.

Gelperin D and Hetzel W (1987). Software Quality Engineering, *Proceedings of Fourth International Conference on Software Testing*, Washington D.C.

Kleijnen, Jack P C (1995). Verification and validation of simulation models, *European Journal of Operational Research*, Elsevier, vol. 82(1), pp.145-162.

Kleijnen Jack P C, Bettonvil B, and Van Groenendaal W (1998). Validation of Trace-Driven Simulation Models: a Novel Regression Test. *Manage. Sci*. 44, 6 (June 1998), pp.812-819.

Larman C and Basili V R (2003). Iterative and incremental developments. a brief history, *Computer* , vol.36, no.6, pp.47-56.

Müller M and Padberg F (2003), About the Return on Investment of Test-Driven Development, *In International Workshop on Economics-Driven Software Engineering Research EDSER-5*.

Nance R E (1993). A history of discrete event simulation programming languages. *In The second ACM SIGPLAN conference on History of programming languages (HOPL-II)*. ACM, New York, NY, USA, pp.149-175.

Newkirk J W and Vorontsov A A (2004). Test-Driven Development in Microsoft .Net. *Microsoft Press*, Redmond, WA, USA.

Shannon R E (1998). Introduction to the art and science of simulation, *Simulation Conference Proceedings*, 1998. Winter , vol.1, pp.7-14.

Zhang Y (2004). Test-driven modeling for model-driven development, *Software, IEEE* , vol.21, no.5, pp.80-86.

AUTHOR BIOGRAPHIES

SHAHRIAR ASTA received a BSc Computer Engineering from the University of Isfahan in 2003. In 2012 he received his MSc Computer Engineering from Istanbul Technical University. He is currently a PhD student in the Automated Scheduling, Optimization and Planning (ASAP) group at the University of Nottingham (http://www.cs.nott.ac.uk/~sba).

ENDER ÖZCAN is an Operational Research and Computer Science lecturer with the ASAP research group at the University of Nottingham (http://www.cs.nott.ac.uk/~exo). He was appointed to the EPSRC funded LANCS initiative and has been leading the Systems to Build Systems research cluster. He has been awarded grants as PI and co-I from a variety of funding sources. He has co-organised many workshops streams on hyper-heuristics/metaheuristics and has many publications in the area. Dr Ozcan is an associate editor of the International Journal of Applied Metaheuristic Computing and Journal of Scheduling.

PEER-OLAF SIEBERS is a lecturer in the School of Computer Science at the University of Nottingham in the United Kingdom (http://www.cs.nott.ac.uk/~pos/). His main research interest is the application of data driven computer simulation to study human-centric complex adaptive systems. This is a highly interdisciplinary research field, involving disciplines like social science, psychology, management science, operations research, economics and engineering. Other areas of interest include Risk Analysis and Systems Biology.