Test Driven Development of UML Models with SMART modeling system accepted for UML 2004, October, Lisbon

Susumu HAYASHI[1], PAN YiBing[1], Masami SATO[1],
Kenji MORI[1], SUL Sejeon[1], and Shusuke HARUNA[2]

[1] Kobe University, Nada, Kobe, Japan,
{shayashi,pybing,satoman,moriken,sejeon}@cs33.scitec.kobe-u.ac.jp
[2] Matsushita Electric Industrial Co., Ltd., Kadoma, Osaka, Japan,
haruna@isl.mei.co.jp

Abstract. We are developing a methodology of Test-Driven Development of Models (TDDM) based on an experimental UML2.0 modeling tool SMART. Our experience shows that TDDM is quite useful for agile model developments. SMART provides guidance how to build models based on compiler errors of testcases, something similar to what Quick Fix of Eclipse does. It also provides such guidance even from failures of testcases, which seems difficult in the case of TDD of programs.

1 Introduction

The integration of Agile methods and Modeling is attracting considerable attention [1, 2, 5–7, 12]. Test-Driven Development (TDD) is a central notion of Agile developments. In this paper, we present a method of TDD of Models with with an experimental UML2.0 modeling tool called SMART.

Boger et al. have introduced an agile modeling approach and its tool support in [6]. Their method is also test-driven and the tool supports it. However our methods differs from this pioneering work in some critical respects and extends it. The tool in [6] depends on Java in order to make model developments and code developments interact. We use an action language for Action Semantics instead of JAVA. Thus, our method is completely independent from any programming languages. This would be important for AgileMDA [12].

However, the most important feature which distinguishes our method from other agile methods would be the full scale realization of tool support of test-driven *guidance*. By “guidance”, we mean “a suggestion telling us the next steps to be achieved and/or aid to achieve the next steps to be achieved.”

In the original TDD, compiler error messages are utilized as suggestions to tell us the classes and methods that we should write next [3]. Quick Fix of Eclipse [16, 8] and some other tools aid us to write them by automatically generating stubs.

Thanks to UML architectures, we could make the idea of guidance advance forward even more than the original TDD for codes. Quick Fix guides which and

how program elements such as classes and methods should be introduced. But guidance by Quick Fix is restricted to *syntactical* aspects, since it is made based on compiler errors (see [8, 16]).

Failures of testcases can provide us with behavioral information how to fix codes. SMART system supports even TDD of behavioral aspects based on failures of testcases and other execution information. Quick Fix introduces stubs but our method can introduce fakes as well. (A stub does nothing. It exists only for compilation. A fake returns a fixed correct value for a particular testcase. See p. 169 of [2].)

This kind of technology is possible thanks to abstractness of UML modeling architecture. It might be difficult to realize similar tool supports for realistic programming languages. It is likely that TDD fits to model-based development such as AgileMDA even more than developments of programs. This seemingly paradoxical fact was realized by a simple technique of logging which we call *omnipresent log*, which is also possible by the abstractness of UML standard.

In 2, we explain TDDM-method by examples. In 3, we give an overview of SMART system. In 4, we describe the architecture of TDDM and discuss an advantage of modeling based developments to direct program developments for test-driven developments.

2 TDD of Models by example

In this section, we explain TDDM methodology by examples. The examples are on a room air conditioner with a remote controller.

2.1 TDD of State Machine

A development in our methodology starts with writing some requirements such as stories or usecases. For illustration, we start with a very simple requirement "the air conditioner may be turned on and off by pushing a power button."

Next step is to write a *testsuite tables*, a table of testcases, which could be understood as a formalized story or scenario instances. The following is a testsuite table for the requirement.

Name	Setup	Test Subject	Verification
testcase1	`aircon^PowerOff`	`aircon.power`	`aircon^PowerOn`
testcase2	`aircon^PowerOn`	`aircon.power`	`aircon^PowerOff`

`aircon` signifies the room air conditioner. "air-con" is a common Japanese word for room air conditioners. `aircon^PowerOff` refers to a state of the state machine of `aircon`. `aircon.power` refers to an event in its state machine. The first row of the table reads "when `aircon` is in the state of `PowerOff`, triggering of `power` event (a modelization of pushing power button) changes the current state to `PowerOn`". The second line reads similarly.

SMART translates the testsuite table into a testsuite consisting of xUnit style testcases of an action language SAL (SMART Action Language) based on

UML Action Semantics. SAL for SMART is used to program actions in models as OAL for Executable UML [11]. The version of SAL used to write testcases is enhanced for testing and is called Extended SAL. The following is the Extended SAL testcase translated from the first row of the testsuite table presented above:

```
begin
setCurrent(aircon^PowerOff);  /* Setup part */
callEvent aircon.power;       /* Execution part */
isCurrent(aircon^PowerOn);    /* Verification part */
end
```

The setup part sets the current state of `aircon` to `PowerOff`. The execution part triggers the event `power`. The verification part is an assertion to check if the new current state of `aircon` is `PowerOn`. This kind of xUnit style testcases are automatically generated, when the contents of testsuite tables are entered. More exactly speaking, all tests are written in and executed by Extended SAL. Testsuite tables serve as the user interface in order to make the input of testcases easier. Although SAL programs must be directly written for complicated cases, testsuite tables are sufficient for most cases.

Next, we compile the testsuite. Then some compilation errors are reported, since no model has been defined. Just as other xUnit-style test tools, SMART Test Tool has a bar which indicates if there is an error or not (see Figure 1 in 2.4 below). When we compile or run testcases, the bar turns green (no error) or red (some errors). We got a red bar this time. SMART tells us there are no objects called `aircon`, no state called `PowerOff`, `PowerOn`, and no event called `power`.

Besides the error reports, SMART also provides suggestions of possible fixes of the errors just as Quick Fix of Java development environment of Eclipse does [16]. For example, it asks if an object `aircon` shall be created. If we accept the suggestion, then SMART prompts us to choose one of the existing classes or enter a new class name. Since no class has been defined yet, we enter a new class name, say, `Air-Conditioner`. Then, SMART declares a new class with the name. Similarly, we are guided to declare other model elements. This kind of suggestions by the tool is called *guidance*.

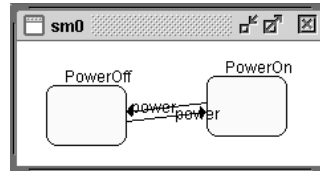
We will be guided and so declare a class, an object, an event and two states. What is actually entered from a keyboard is only the name of the class.

When all compilation errors are gone, we get the green bar. Then we run the testsuites created from the testsuite table. We have a red bar again, but this time the error is a failure. The assertion `isCurrent(aircon^PowerOn)` of **testcase1** failed.

Failure is behavioral errors. Some conventional development tools fix structural error such as lack of `aircon` object, but they do not fix behavioral errors. The guidance mechanism of SMART fixes even behavioral errors. SMART recognizes that the execution of the testcase took three steps:

setup: set the current state to `PowerOff`,
execution: trigger the event `power`,
verification: verify if the current state is `PowerOn`.

Since all the three steps happen with the state machine of `aircon`, SMART infers that we would intend a transition from `PowerOff` to `PowerOn` triggered by `power`. Thus it asks us if we would like to make such a transition. This is called *transition guidance*. Answering to this guidance "yes" to create a transition, the failure of the first testcase is gone. Similarly, we make the reverse transition driven by the failure of the second testcase. When it is done, the state machine **sm0** is obtained. What we have input from the keyboard are only the name of the class `Air-Conditioner`, although we have accepted many guidance.



Note that there are other possibilities of the solution for the failure of **testcase1**. An air conditioner may enter into a warming up state, when the power button is pushed. On the event of "warming up finished" triggered by a sensor or something, it would make a transition to `PowerOn`. This kind of scenario is possible. Thus guidance of SMART must be inspected and guaranteed by users to make models respect intention of users.

2.2 Scenario test and testsuite

The testcases we have considered are all "unit tests". How about scenario tests? Scenario tests can be written as a sequence of unit tests. The following is a scenario test to change status of the the air conditioner from Off to On and again to Off.

```
begin
setCurrent(aircon^PowerOff);  /* setup part */
callEvent aircon.power;       /* Execution part */
isCurrent(aircon^PowerOn);    /* Verification part */
callEvent aircon.power;       /* Execution part */
isCurrent(aircon^PowerOff);   /* Verification part */
end
```

This testcase can be represented by the following testsuite table:

Name	Setup	Test Subject	Verification
testcase3	`aircon^PowerOff`	`aircon.power`	`aircon^PowerOn`
		`aircon.power`	`aircon^PowerOff`

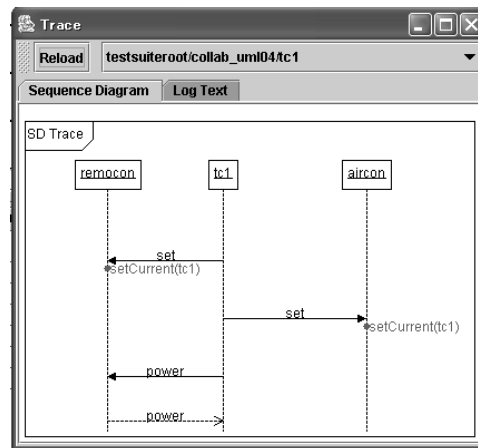
2.3 TDD of collaboration

Let us consider an example of TDD of collaboration. We have built `aircon` and then we consider the following testsuite:

Name	Setup	Test Subject	Verification
tc1	`remocon^PowerOff`	`remocon.power`	`remocon^PowerOn`
tc2	`aircon^PowerOff` `remocon^PowerOff`	`remocon.power`	`aircon^PowerOn`

The intention of the testsuite is "we can turn `aircon` on by `remocon`". `remocon` signifies a remote controller. "remo-con" is a Japanese word for remote controller. In the testcases of the previous subsection, only one object is concerned. Now two parties are involved. Compiling this, we create `aircon`, `remocon` and other elements. Then we run the compiled testcase.

The assertion `isCurrent(aircon^PowerOn)` of **tc2** failed, since `aircon` is not turned on by the trigger `remocon.power`. The transition guidance is not provided this time. This is because `remocon.power` is targeted to another machine `aircon`. But we do not know it. So we display a trace of the execution in a sequence diagram shown below to see what happened. This is called *sequence diagram guidance*.



The sequence diagram clearly shows what happened. The testcase set the initial test states of the two models but models do not communicate at all during the send/return of `power` event. We have to make them collaborate.

We do it by putting SAL code `callEvent(aircon.power)` in the effect of the transition of `remocon` triggered by `remocon.power`. (Effect is the activity executed with the transition [13].)

Since the models `remocon` and `aircon` have started to collaborate, it would be a good idea to have a collaboration diagram for it. We make a new model and declare a collaboration in its composite structure compartment. Let `aircon` and `remocon` be parts of the collaboration. Run the testcase again. The bar turns green, of course. We have not added any elements which may change the behavior of the model.

Although the bar is green, there might be some errors with the collaboration diagram. We may have SMART check if the diagram is correctly drawn by pushing the button of "communication test". Then, it reports that although `remocon` communicated with `aircon` but a connector between them is lost in the diagram, and SMART asks if it should draw a connector from `remocon` to `aircon`. It seems that we forgot to draw a connector between them. But SMART knew such a communication actually took place in the last executions of testcases. Accept the guidance, execute and check. The communication check passes this time. This kind of guidance is called *connector guidance*.

2.4 TDD of actions: change summary guidance

As the last example, we present a more elaborated developments with TDD of actions (This example is from [14]). Automatic inferences of actions are unrealistic. Thus, guidance is restricted to presentations of information collected from testcase executions. Although decisions are entirely left for users, this kind of information is useful enough.

We enhance the aircon-remocon model with controls of temperatures. The controller has an up-button, a down-button and a liquid crystal display indicating the room temperature. By up- and down-buttons, the temperature could be controlled by one degree. The highest and lowest limits of temperatures memorized by the controller are respectively 26 and 16 degrees centigrade. The memorized temperature does not change beyond the limits. When up- or down-button is pushed, the memorized temperature is transmitted to the air conditioner and the temperature memorized by the air conditioner is set according to it.

Firstly, we model `remocon`. The testsuite table is as follows:

Name	Setup	Test Subject	Verification
power_on	`remocon^PowerOff`	`remocon.power`	`remocon^PowerOn`
power_off	`remocon^PowerOn`	`remocon.power`	`remocon^PowerOff`
temp_up	`remocon.temp:=20`	`remocon.up`	`21=remocon.temp`
temp_up_max	`remocon.temp:=26`	`remocon.up`	`26=remocon.temp`
temp_down	`remocon.temp:=20`	`remocon.down`	`19=remocon.temp`
temp_down_min	`remocon.temp:=16`	`remocon.down`	`16=remocon.temp`

Figure 1 is a picture of the modeler, in which a state machine of `remocon` has been built by fixing all the structural errors of the testsuite given above. Then we run the testsuite. Some testcases fail. This is because the effects (actions) attached to transitions have not been written yet. For example, **temp_up** testcase tells us the temperature (20) does not rise. So we add the command `remocon.temp:=remocon.temp+1` in the effect of the transition triggered by `remocon.up`. The error is fixed and we do the same thing for **temp_down**. **temp_down** and **temp_up** run with the green bar. Now the entire model must be fine. Thus we run the entire testsuite. But, we have a red bar of failure again. What's wrong?

This is a good point to use *change summary guidance*. A sequence diagram guidance displays a particular execution of a testcase. On the other hand, change summary guidance displays change of attributes and state of models caused by an event or activity under test. In this case, change summary guidance for the testsuite looks like this

```
Created automatically when running TestCases:
From: testsuiteroot\remocon\temp_up
remocon.temp[0]: 20 ==> 21 true
remocon: PowerOn
```

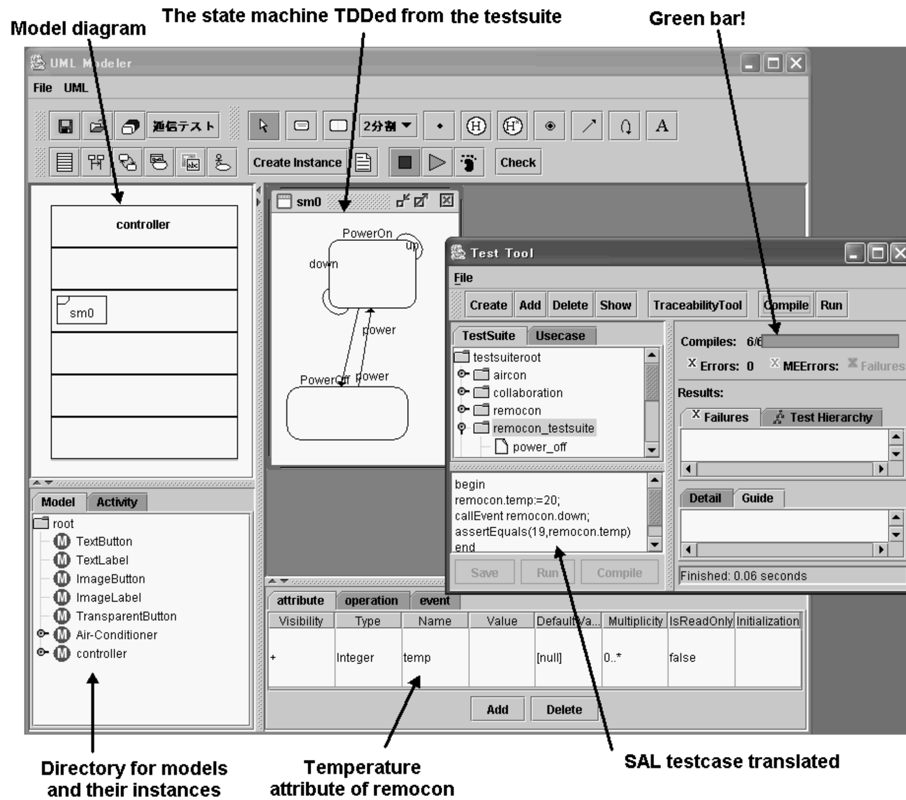


Fig. 1. A state machine made by TDD with SMART

```
From: testsuiteroot\remocon\temp_up_max
remocon.temp[0]: 26 ==> 27 false
remocon: PowerOn
```

The first unit for `testsuiteroot\remocon\temp_up` shows that **temp_up** is ok. The temperature `temp` changed from 26 to 27 as expected by the test-case. But the second unit `testsuiteroot\remocon\temp_up_max` shows that **temp_up_max** for `remocon` failed. The temparature changes from 26 to 27, but the expected value by the testcase is 26. Oops! We have forgotten to limit the temperature rise. We have to prevent the rise over 26 by putting a guard to the transition trigger. We have to do the same for "down". Now, we regained the green bar!

We do similar thing for `remocon` model. Then we make these two collaborate (communicate). A testsuite for the communication looks as follows:

Name	Setup	Test Subject	Verification
power_on	`remocon^PowerOff;` `aircon^PowerOff;`	`remocon.power`	`remocon^PowerOn;` `aircon^PowerOn`
temp_up	`remocon.temp:=20;` `aircon.temp:=20`	`remocon.up`	`21=remocon.temp;` `21=aircon.temp`

The rest of developments are essentially the same as the one in 2.3 and we omit it.

3 SMART modeling system

In this section, we will give details of SMART modeling system. SMART modeling system is a UML 2.0 modeling tool to realize our TDDM methodology. The main application targets of SMART are distributed embedded systems such as home electric appliances. Our development method, and the SMART system as well is based on state machines as Executable UML. The current version SMART 0.3 is based on UML 2.0 standard except its action language which is still based on UML 1.5.

3.1 Model concept of SMART

In SMART, a system is modeled as a behaviored and structured classifier with graphical user interfaces. A *SMART model* consists of its name, properties, operations, events, constructors, and the following five compartments (i) Composite Structure, (ii) User Interface, (iii) State Machine, (iv) Interaction (as Sequence Diagram), (v) Use Case. A SMART model is depicted by a table like diagram called *model diagram* (see Figure 1 in 2.4). The concept of SMART model and model diagrams were introduced and discussed in [15].

SMART models can be nested. A distributed system, such as communicating electric appliances is modelized by the composite structure compartment. Formally, a distributed system is a SMART model whose composite structure compartment consists of a collaboration (of UML 2.0) and each part of the collaboration is again a SMART model. The aircon-remocon model in 2 can be represented by a SMART model consisting of a collaboration of two SMART models of aircon and remocon. However, collaborations need not be written explicitly.

SMART is intended to be a simulator for homenetwork. Thus it has a graphical tool to build GUI's for home appliances such as a remote controller of an air conditioner or a TV. A GUI for the remote controller is easily made by putting some buttons and a text box on a picture of the controller body.

3.2 SAL: SMART Action Language

SAL (SMART Action Language) is our own action language based on UML Action Semantics. SAL compiles its programs into Actions and then Action

Executions for execution. The current SMART 0.3 uses a sequential version of SAL based on UML 1.5 Action Semantics. A new version of SAL supporting concurrent data flow computations with multi-valued functions based on UML 2.0 Action semantics is nearly completed and planned to replace the current SAL soon.

SAL is used to describe effects of transitions of state machines and operations of classes. Conforming to UML 2.0 standard [13], actions are handled independently from these model elements. Namely, SAL programs are written and managed separately from these model elements, and then they are associated with them. SAL is also a language for testcases as explained in the next subsection.

3.3 STT: SMART Test Tool

STT (SMART Test Tool) is a test tool for TDDM introduced in [14]. It is implemented on JUnit testing framework [4]. In xUnit framework, test subjects and testcases are both written in the same language "x" for xUnit. JUnit testcases are Java programs to test Java programs. In STT, test subjects are UML models and testcases are written in SAL programs enhanced with a test profile STP (SMART Test Profile). Since SAL is an action language for Action Semantics, this architecture conforms to UML except enhancements for STP.

In Extreme Modeling [6], test subjects are UML models, and testcases are sequence diagrams and Java programs. Their Java program testcases are also based on JUnit and so look very similar to our method. But, it does not seem that the "test-driven guidance" technique is used. A reason of this lack might come from the separation of testing framework from UML standards. In our framework, since tests are conducted by Action Semantics which drives the models, syntax error messages and semantical failures are easily utilized by TDDM-support tool.

Furthermore, since our testcases are in "native semantics" language of UML, they fit to AgileMDA. Testcases for each platforms should be generated from the abstract testcases of STT like our method. Such an approach is used in Telelogic TAU2 UML modeler, but the testcases of TAU2 are based on ITU TTCN3 which are for "test-after" style testing.

Since testing is not considered in UML standard, we introduces simple profile for testing called STP (SMART Test Profile). Exactly speaking STP is a metamodel extension, although we call it a profile. Firstly, the concept of configurations of executing state machines is introduced so that state machines in execution can be dumped and restored. Metaclasses for state machines are hard-extended by them. Secondly, the Action metaclass is enhanced with actions for testing. New metaclasses for actions are

1. *WriteStateMachineConfigurationAction* and *ReadStateMachineoConfigurationAction* for writing and reading configuration of state machines to set up and verify state machine configuration.
2. *DumpConfigurationAction* and *RestoreConfigurationAction* dump and restore executing models. At any stable moment, the entire models in execution

can be dumped out and restored back later. This make set up and tear down for testing extremely easy. This is quite useful also for usual developments of models. The implementation of these for SMART dumps/restores even the positions of model diagrams.
3. *AssertAction* extends Action Semantics with actions for xUnit-style assertions.

Extended SAL is an extension of SAL supporting STP. This is the language which was used for testcases shown in 2. For example, `setCurrent` command sets up current states of state machines, and `isCurrent` checks if a state is a current state. `assertEquals` is a counterpart of `assertEquals`-assertion of JUnit.

Errors of testcases are classified into three categories (i) Syntax error, (ii) ME error (Missing Element error) and (iii) Failure. A syntax error is an error of syntax on Extended SAL. This is not an error of a model, but an error of a testcase.

ME errors and failures are errors on models. ME errors are errors of missing model elements. If a token in a testcase refers to a model element but such an element does not exist, then ME error is raised. As syntax errors, this sort of errors is detected by the SAL compiler.

On the other hand, failures are behavioral errors. It is quite difficult to fix them automatically. It may be even dangerous to do so, since there are too many possible solutions and presenting small numbers of them to users may guide them into wrong directions which are not conform to their intention. The behavioral guidance supported by the current version of SMART are *transition guidance*, *change summary guidance*, *connector guidance*, and *sequence diagram guidance* used in the examples of 2.3. In 4, we will discuss how these guidance are realized and how they can be extended.

3.4 STWT: SMART Tractability Web Tool

STWT (SMART Tractability Web Tool) is a tool managing relationships documents and elements in SMART. Figure 2 shows the concept of STWT. SMART has a lot of documents such as testcases and model elements. All of them are related. For example, the testcase TC203 is an "instance" of the usecase UC32a, which is a refinement of the usecase UC32. The Comp0 presents the collaboration of `aircon` and `remocon` described in UC32.

When the designer of the system changes his mind so that the maximum of the temperature should be 28 degree and so changed the requirements. Then testcases must be accordingly changed. STWT is a web of STWT-documents to trace dependence of these documents in such a case. Testcases, usecases, and model elements can be declared as STWT-documents. Change summary guidances, which are plain texts generated by STT, are declared as STWT-documents, when they are generated.

STWT-documents are related by connectors and connectors has meta-documents called *explanatory note* (see Figure 2), in which we can write how

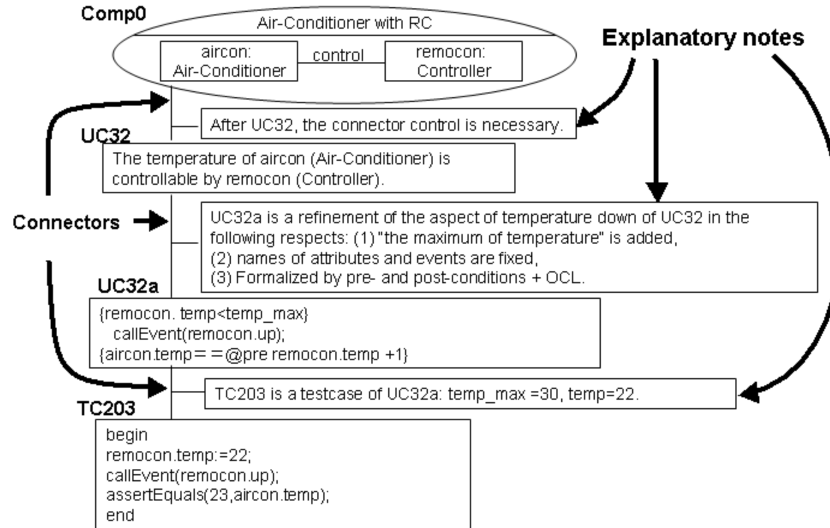


Fig. 2. The concept of STWT

the parties at connector ends are related. In the current version of SMART, explanatory notes are plain texts.

To browse STWT web, SMART now has two "view" mechanisms. *Local view* good for displaying each documents and *Global view* displaying a bird eye view of the web. STWT-documents may have "levels" and STWT-documents are deployed in positions specified by these levels.

4 Architecture of guidance and omnipresent log

In this section, the architecture of guidance is explained and extensions of guidance is discussed. Guidance is a suggestion or aid made to help modeling. Guidance is classified into behavioral guidance and structural guidance. Guidance is behavioral, when it is based on the results of executions of models and testcases. Guidance is structural when it is not behavioral. Since compilation takes place before executions, guidance based on compiler errors are structural. Actually, most structural guidance are driven by SAL compiler errors.

How are behavioral guidance realized? SMART records all executions of testcases and actions (activities in UML 2.0 terminology). These records are called *omnipresent log*, since the logging presents everywhere and anywhere. In reality, SMART does not keep records of all the executions, but they are cached and kept up to a certain limit.

The following shows parts of a real log:

```
begin-testcase:testsuiteroot\temp\testcase1,2004/1/22/13:28
write-attribute:controller.set\-temp,controller.set\-temp[0],24,24,24,...
setcurrent-state:aircon.poweron,testcase1
  :
transition-end:controller.transition1
message-send-return-event:push\-up,testsuiteroot\temp\testcase1,...
assert-equals:25,controller.set\-temp[0],25,24,false
```

The readers would see that it records call/return of messages, start/end of activities and testcases, call of each actions like write-attribute action, and errors, etc. etc.. Since practically all the executions of the models are recorded in this way, it is easy to make sequence diagram guidance and change summary guidance from the omnipresent log. By definition, *guidance is behavioral, when it is based on the omnipresent log.*

4.1 Transition guidance and some related guidance

Let's explain how the transition guidance is made from the omnipresent log. When a testcase is executed and a failure on current states occurs, the execution record of the failure is extracted from the omnipresent log and analyzed. If the test subject is an event for a model, the setup part sets a state to the current state of the model (a model has a state machine at most one), the verification part checks if a state is the current state of the same model, and finally the check fails, then the transition guidance is triggered. This triggering condition for the transition guidance is nicely represented by the following pre- and post-conditioned behavior by using a variable keeping the current state `current`:

{`current=PowerOff`} transition triggered by `power` {`current=PowerOn`}.

An obvious solution for "transition triggered by `power`" is `current:=PowerOn`, which means the transition to `PowerOn`. Note that this solution can be obtained by Dijkstra-Gries style top-down inference of programs from specifications by the axiom of assignment (e.g. [9]). The same consideration leads us to a guidance for the following behavior specification:

{`temp=22`} action triggered by `up` {`temp=23`}.

The solution after Dijkstra-Gries style top-down inference is

{`temp=22`} `temp:=23` {`temp=23`}.

The inferred code `temp:=23` is not correct, but it is the standard fake used in TDD. Putting it in the effect of the transition triggered by `up`, a fake for the testcase is automatically generated. This will be called *assignment fake guidance*. Stub generation is now supported by some programming tools Eclipse, IDEA, etc., but fake generation are not.

The same technique will produce fakes for communications. A fake for the testcase **tc2** in 2.3 could be generated by putting `setCurrent(aircon^PowerOn)`

in the effect of the transition triggered by `remocon.power`. Fakes for change of values of properties caused by communications can be generated in the same way.

There are no essential difficulties to install these fake guidances and they are planned to be done very soon. But they should not be used till an improvement of STWT is completed. Fake generation is useful but also dangerous. Such semi-automatically generated codes could be forgotten easily. When they are left forgotten, they are bugs which pass tests. Actually they are bugs designed after testcases! Fakes must be marked and managed by STWT.

A challenging research project is to use model checking to infer fakes. Model checking is used to infer the next steps of play-out from temporal specifications [10]. By such a method, we may infer fake codes automatically from *pointwise specifications* that are the specifications without parameters as the ones presented above.

4.2 User defined guidance, investigators

We "coded" the triggering conditions of guidance by pre- and post-conditions above. By introducing a script language to describe actions of guidance, scripting of guidance for customization would be possible.

Omnipresent log is an archive of complete execution traces. We may regard it a huge archive and let agents mine incidents from the archive. These agents are *investigators* of the archive and they could run even on other machines if the archive are shared through networks.

xUnit style testcases are not quite good for checking invariants. But we could ask investigators to check if `temp` attribute is kept between 16 and 26 for *all* executions of the air conditioner model recorded in the log. Note that the check can be done even after the execution finished. In a sense, we can test the past.

4.3 Modeling and omnipresent log

Note that the technique of omnipresent log is the same as the one of record/reply debugger often used for debugging of nondeterministic systems such as multi-threaded Java programs. For such applications, the huge size of log causes difficulty.

For our case, the size of log is remarkably small. This fact seems to come from the abstractness of behavioral semantics of UML models. UML models are abstract even for their executions. We do not need to record accesses to registers, stack, changes of program counters, etc. etc..

The current version of the omnipresent log records almost all actions made by SAL activities and state machines but not really the all executions. We expect recording all the execution would not make the log much bigger but it would definitely increase the size. Thus some techniques to reduce the size may be necessary. Techniques developed for record/reply debugger would be helpful. This would be an interesting research problem.

Note that we may assume the *standard execution semantics* of UML for such research. Although the semantics is not completely defined, this would be a great advantage compared with record/replay debugger for programming languages and hardware.

Thus modeling has two great advantages abstractness and standardization for realization of omnipresent log compared to programming languages.

5 Conclusions

In this paper, a methodology of TDD of Models was introduced. The methodology is based on support with SMART UML modeling tool. The methodology was illustrated with examples and the architecture of the methodology and the system were explained. Possible extensions of TDDM were also discussed.

The novel feature of SMART system is test-driven behavioral guidance. It is possible by the abstract and standardized semantics of model executions. Thus this TDD methodology fits better to MDA than to code base developments.

For further developments of TDDM technology, improvements of STWT is inevitable. In the next version of SMART, STWT will be based on XML and Semantic Web like architecture.

An important feature of TDD which our system lacks for is supports for refactoring. A refactoring editor for UML models has introduced for Poseidon tool [7]. SMART should have such aids of refactoring. Refactoring such as replacing fake codes with real codes is highly intelligent and very difficult to automate. It has not been known if our architecture helps to solve these problems.

Now we input testsuite tables from keyboards. But contents of fields of testsuite tables are very simple and are easily specified graphically. By the play-in technology in [10], such an interactive and graphical input tool for testsuite tables would be realized without great difficulty. By such a tool, the entire developments would become very graphical, interactive and intuitive.

The concurrent version of SAL and SMART are planned to be launched very soon. This version of SMART is expected to be useful for simulation of homenetworks, but there are several problems to be solved for practical applications. The examples of `aircon-remocon` model assumed synchronous communication. However communications between most Japanese air conditioners and remote controllers are asynchronous. SMART should support such a communication and then there is a problem of timing of evaluation of testcases. An assertion verifying change of temperature set must be evaluated after the change of temperature attribute. Such timing mechanism does not present in xUnit-style testcases and it is possible only when SMART/SAL supports concurrent realtime computing.

We are planning to attack these problems soon to apply SMART to the problem of homenetwork simulations. In the long term, we are also thinking to relate STWT to formal methods including term rewriting system and proof checkers, and Semantic Web.

6 Acknowledgments

We express our deepest appreciation to Tomoharu Yachikami, who designed and built the first version of SMART. He also directed our attention to TDD. Without his ideas and effort, our project would not be materialized. We thank to Shingo Noguchi, Kazuto Aikou and Hideyuki Kushida for their efforts to building a state machine tool from which SMART evolved.

References

1. Scott W. Ambler, `http://www.agilemodeling.com`, Agile Modeling site.
2. David Astels, *Test-Driven Development: A Practical Guide*, Printece Hall, 2003.
3. Kent Beck, *Test-Driven Development: By Example*, Addison-Wesley, 2002.
4. Kent Beck, Erich Gamma, `http://www.junit.org`, Junit site.
5. Barry Boehm, Richard Turner, *Balancing Agility and Discipline, A Guide for the Perplexed*, Addison-Wesley, 2004.
6. Marko Boger, et al., Extreme Modeling, in Succi, G. and Marchesi, M. *Extreme Programming Examined*, 175–189, Addison-Wesley, 2000.
7. Marko Boger, et al., *Refactoring Browser for UML*, in LNCS 2591, 366–377, 2003.
8. Erich Gamma and Kent Beck, *Contributing to eclipse*, Addison-Wesley, 2004.
9. David Gries, *The Science of Programming*, Springer Verlag, 1981.
10. David Harel and Rami Marelly, Specifying and Executing Behavioral Requirements: The Play-In/Play-Out Approach, *Software and System Modeling* **2**, 82–107, 2003.
11. Stephen J. Mellor, Marc J. Balcer, *Executable UML*, Addison-Wesley (2002)
12. Stephen J. Mellor, *Agile MDA*, `www.projtech.com/pubs/current/agilemda.pdf`.
13. OMG, *UML 2.0 Superstructure Specification*, `http://www.omg.org` (2003)
14. Yibing Pan, *Test Driven Development of UML Modeling* (in Japanese), master thesis, Graduate school of Science and Technology, Kobe University, 2004.
15. Masami Sato, *Structural modeling based on UML2.0* (in Japanese), bachelar thesis, Faculty of Engineering, Kobe University, 2004.
16. Sherry Shavor et al., *The Java Developer's Guide to Eclipse*, Addison-Wesley, 2003.