Test-Driven Modeling for Model-Driven Development

Yuefeng Zhang, *Motorola*

Extreme Programming is a code-centric, lightweight software development process.[1–3] One of XP's main goals is achieving agility—that is, delivering working products to customers and quickly adapting to changes. XP's strategy is to minimize software development effort (by doing "the simplest thing that could possibly work"[1]) and documentation. Testing plays a key role in this because XP developers write test cases before code and the code is only complete once it has successfully passed its test cases.[3]

This article presents a new software development process based on model-driven development (MDD), called *test-driven modeling*. (The basis of this article is a project in Motorola's iDEN division that is extending and migrating a large legacy telecommunication system to new platforms using TDM.) This process involves automatic testing through simulation and using executable models as living software system architecture documents. In TDM, we use the same message sequence charts (MSCs) for both system analysis (or design documents) and unit test cases. Similarly, we use the same high-level modeling diagrams for both automatic code generation and living software architecture documents to guide the system's detailed implementation in later phases. Thus, we can save a significant amount of time by reusing MSCs and modeling diagrams compared with traditional plan-driven methods.

We all know that testing is important for any software development, and it's challenging to figure out how much testing is enough. Coverage analysis in TDM addresses this difficulty.

A new software development process called test-driven modeling applies the Extreme Programming test-driven paradigm in a model-driven development environment. Practical results show that developers can effectively apply TDM to large projects with high productivity and quality in terms of the number of code defects.

Test-driven modeling

Figure 1 outlines the TDM workflow. TDM's main idea is to apply the XP test-driven paradigm to an MDD process.[4,5] We can divide the TDM process into six phases:

- System requirements specification
- High-level modeling (HLM)
- Low-level modeling (LLM)
- Modeling-quality control
- Automatic code generation
- Target platform testing

This article focuses on TDM's modeling aspect and thus discusses only the HLM, LLM, modeling-quality control, and code generation phases. For convenience, the modeling diagrams I use in this section are based on the Specification and Description Language (SDL) modeling language[6] (ITU-T Z 100, www.itu.int/rec/recommendation.asp?type=products&lang=e&parent=T-REC-Z) and the commercial modeling tool Telelogic TAU SDL Suite (www.telelogic.com/products/tau/sdl). The Object Management Group (OMG; www.omg.org) adopted SDL's main modeling features in UML 2.0 (www.omg.org/uml), which is supported by the next-generation modeling tool Telelogic TAU G2/Developer. Thus, we should be able to use TDM with TAU G2 with minimum change. To avoid disclosing Motorola's proprietary information, I used the SDL diagrams for a demo model in this article. Normally, MSCs are part of a model.[4,5] However, for the convenience of discussion, I treat MSCs as separate entities in the rest of this article.

High-level modeling

HLM starts with system requirements. HLM's two main goals include identifying

- The system's use case scenarios (that is, MSCs) from the system requirements specification
- The software system architecture that can realize the identified use case scenarios (that is, MSCs)

MSCs represent the system's dynamic behavior, and the software system architecture represents the system's static behavior.

A system MSC (see Figure 2) in HLM treats the system as a black box—that is, it only indicates message exchanges between the system and the system environment (UML actors). The software system architecture only indicates which subsystems are used in the system (see Figure 3), how they are connected to each other to form a system, and which active and passive classes and objects are used and how they are connected within a subsystem (see Figure 4). An active/passive object is an instance of an active/passive class. HLM doesn't define an active or passive object's behavior. This definition is one of the major tasks in LLM. However, we can define a dummy state-transition chart for each active object in HLM for architecture-verification purposes. These dummy state-transition charts just need to be good enough to reproduce the identified MSCs. At this point, the MSCs focus on signals and ignore the data associated with signals. We replace the dummy state-transition charts with real state-transition charts in LLM. After that, we can refine the system MSCs with signal data and finish the system unit testing by executing the refined system MSCs.

HLM is test driven because we create the system MSCs first and then define the software system architecture against them. The MSCs are automatically executed by simulation to verify whether the software system architecture meets system requirements (MSCs). Finally, we refine and execute the MSCs for system unit testing.

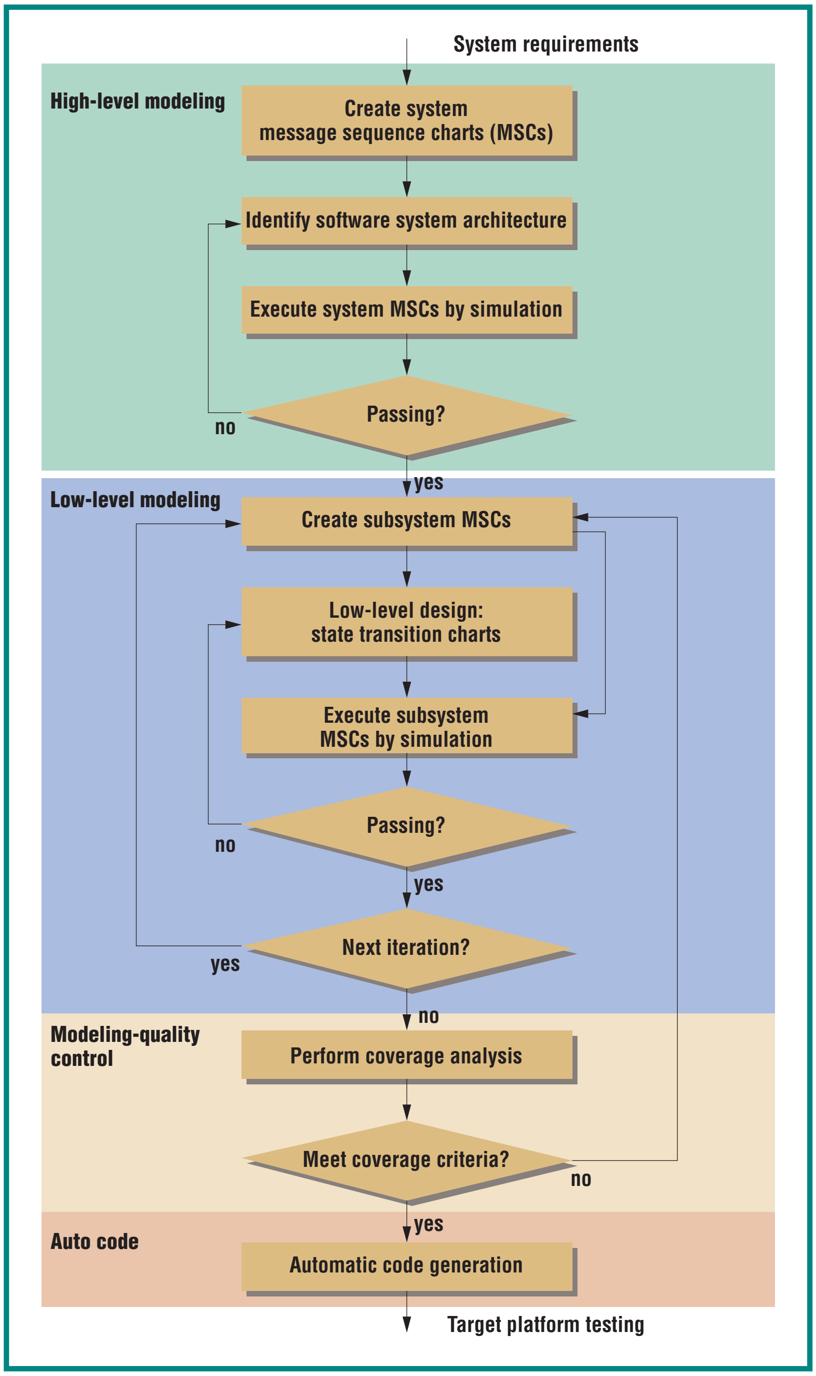

System requirements
High-level modeling
Create system message sequence charts (MSCs)
Identify software system architecture
Execute system MSCs by simulation
Passing?
no
yes
Low-level modeling
Create subsystem MSCs
Low-level design: state transition charts
Execute subsystem MSCs by simulation
Passing?
no
yes
Next iteration?
yes
no
Modeling-quality control
Perform coverage analysis
Meet coverage criteria?
no
yes
Auto code
Automatic code generation
Target platform testing

Figure 1. Test-driven modeling (TDM) workflow.

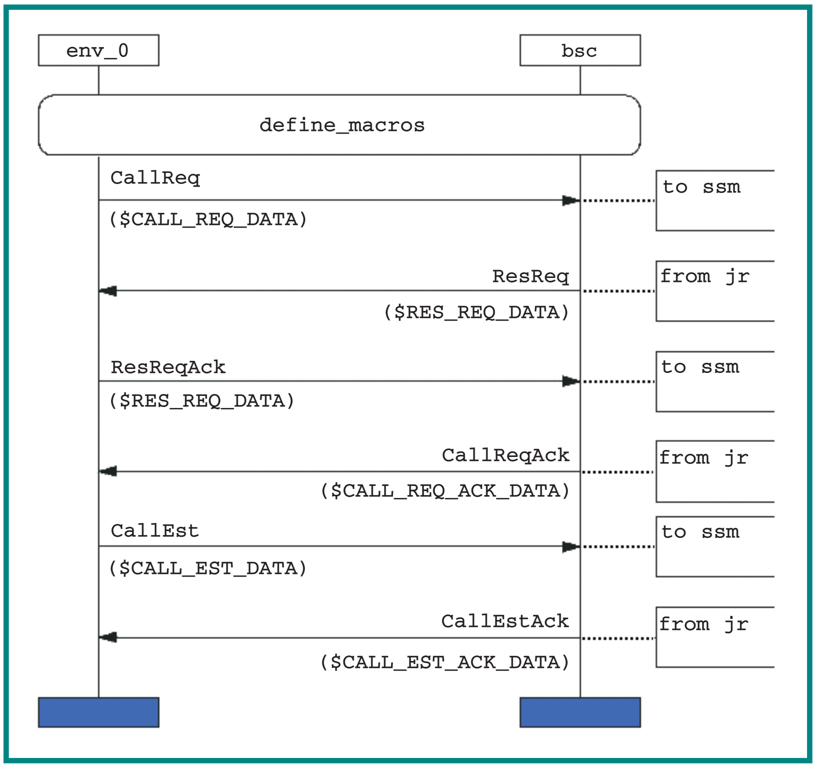


Figure 2. The system MSC call_establishment for a demo SDL model. The rounded rectangle represents an MSC reference, indicating that the MSC call_establishment refers to the MSC define_macros for reuse. Bsc is the subsystem, and ssm and jr are active classes. Env_0 represents the subsystem bsc's environment.

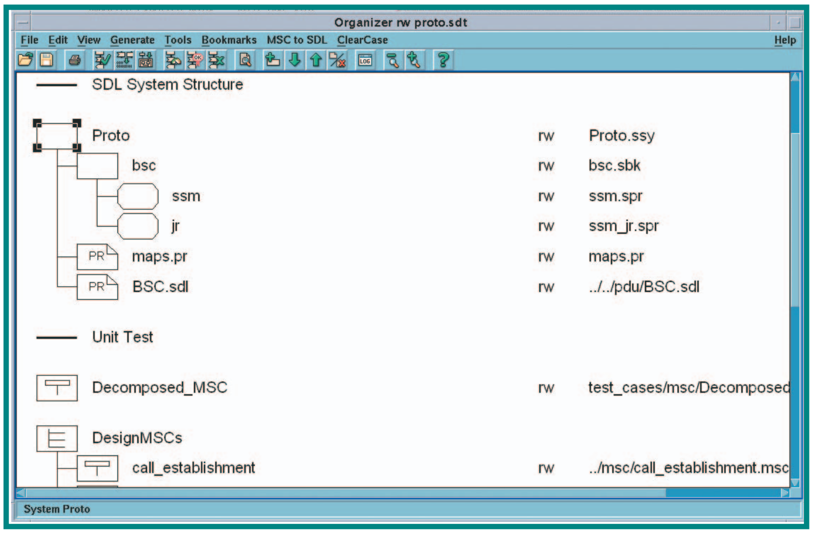
Low-level modeling

LLM's main goal is to define the subsystems' dynamic and static behavior. Similar to the system definition, subsystem MSCs define the subsystem's dynamic behavior. Additionally, a subsystem's static behavior is defined by the static behavior of the sub-subsystems, and/or possibly by the static behavior of the active or passive objects within the subsystem. An active object has its own thread of execution, and it calls passive objects for services. A state-transition chart in SDL defines an active object's static behavior. For example, Figure 5 shows the state-transition chart of the active object *ssm* from the demo model in Figure 3.

Similar to a system MSC, a subsystem MSC in LLM treats the subsystem as a black box—that is, it only indicates message exchanges between the subsystem and its environment. Subsystem MSCs are derived from system MSCs through MSC decomposition. A decomposed MSC shows the system's internal behavior—that is, message exchange between subsystems and active objects. The system and subsystem MSCs are suitable for system and subsystem unit (black box) testing, while the decomposed MSCs are useful for system integration testing.

The amount of work for LLM can be huge, depending on a project's size. Thus, typically we subdivide LLM into multiple iterations. Each iteration implements a subset of subsystem MSCs. In this manner, like XP and other model-driven processes,[4,5] TDM is incremental and iterative because the system functionality is developed in multiple iterations. Each iteration adds more system features on top of previous iterations incrementally.

LLM is test driven in that we create subsystem MSCs first and then define the detailed design (state-transition charts) of subsystems against subsystem MSCs. Finally, we enhance the subsystem MSCs with signal data and then automatically execute them by simulation to verify whether the subsystems' design meets subsystem requirements (that is, MSCs).


Organizer rw proto.sdt
File Edit View Generate Tools Bookmarks MSC to SDL ClearCase Help
System Proto

```
SDL System Structure

Proto                     rw    Proto.ssy
  bsc                     rw    bsc.sbk
    ssm                   rw    ssm.spr
    jr                    rw    ssm_jr.spr
  maps.pr                 rw    maps.pr
  BSC.sdl                 rw    ../../pdu/BSC.sdl

Unit Test

Decomposed_MSC            rw    test_cases/msc/Decomposed

DesignMSCs
  call_establishment      rw    ../msc/call_establishment.msc
```

Figure 3. A demo SDL model in Telelogic TAU SDL Suite, indicating an application system Proto that contains subsystem bsc, which contains two active classes, ssm and jr. The model includes two data type and SDL signal definition files, maps.pr and BSC.sdl.

Modeling-quality control

As I described earlier, MSCs are used as test cases in TDM. An MSC can refer to other MSCs (see Figure 2 for an example) and thus we can build an MSC hierarchy. So, a test case can include other test cases. A set of related MSCs can be organized and executed by the modeling tool together as a test suite; such a suite is called a module in Telelogic TAU SDL Suite. Like traditional test cases, multiple suites of MSCs are allowed and can be executed by the Telelogic TAU SDL Suite together in a batch mode. Once an MSC execution is completed, a summary report will be automatically generated by the modeling tool to indicate which MSCs passed or failed the testing. If an MSC execution fails, then the modeling tool will automatically generate and display an MSC graphically to indicate the exact failure point.

The challenge with testing is to determine how much is enough. For manual coding, the answer to this question is typically fuzzy. However, in TDM, the answer is intuitive—that is, we hope to achieve a high coverage percentage (ideally 100 percent). We obtain the answer by looking at graphical coverage trees that the modeling tool automatically generates. A coverage tree shows which design elements and state transitions have been covered by the executed MSCs (see Figure 6 as an example). By looking at the uncovered design elements and state transitions in the coverage trees, we can easily identify additional MSCs (typically exception-handling use case scenarios) to cover them. These additional MSCs are normally directly executed without model changes (see the LLM phase in Figure 1). In this manner, we can effectively increase the coverage percentage. One project goal is to agree on a coverage percentage amount that, when reached, indicates we have completed testing.

A coverage tree for a reasonably large SDL model tends to be too big for easy analysis. In this case, I recommend showing only the SDL symbols that MSC execution doesn't cover, as Figure 6 shows.

The model quality is test driven in that the coverage tree is generated by executing MSCs and it's in turn used to identify more MSCs to be executed.

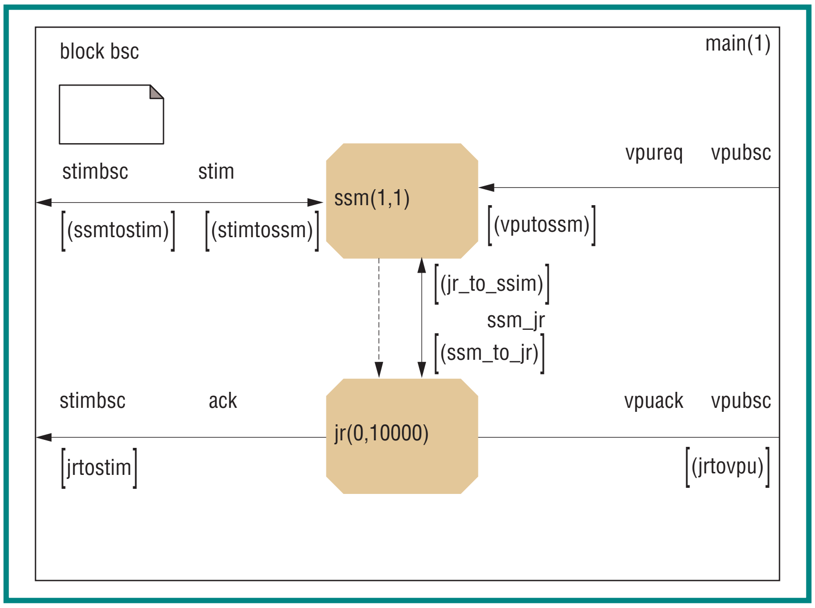


Figure 4. The architecture of the subsystem bsc, indicating that two active classes ssm and jr are used within bsc. The dashed arrow represents that an active object ssm creates the instances of the active class jr.

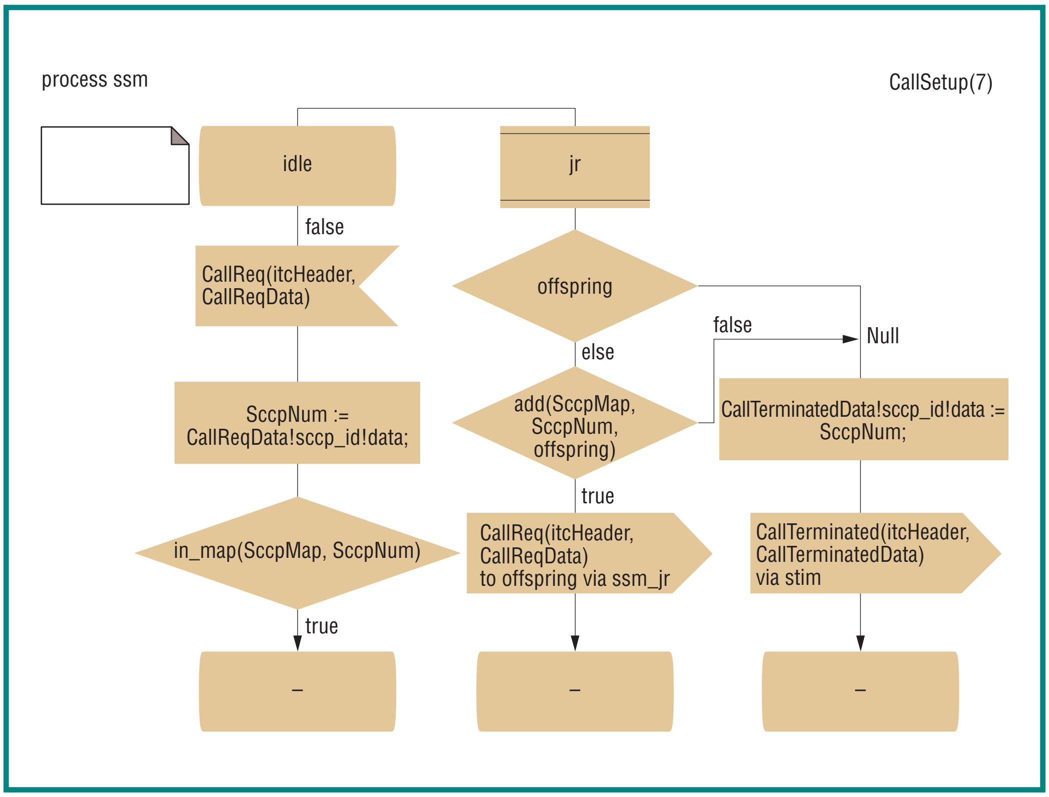

process ssm
CallSetup(7)
idle
jr
false
CallReq(itcHeader, CallReqData)
offspring
false
Null
else
SccpNum := CallReqData!sccp_id!data;
add(SccpMap, SccpNum, offspring)
CallTerminatedData!sccp_id!data := SccpNum;
true
in_map(SccpMap, SccpNum)
CallReq(itcHeader, CallReqData) to offspring via ssm_jr
CallTerminated(itcHeader, CallTerminatedData) via stim
true
–
–
–

Figure 5. Part of the state-transition chart for the active object ssm in the demo model.

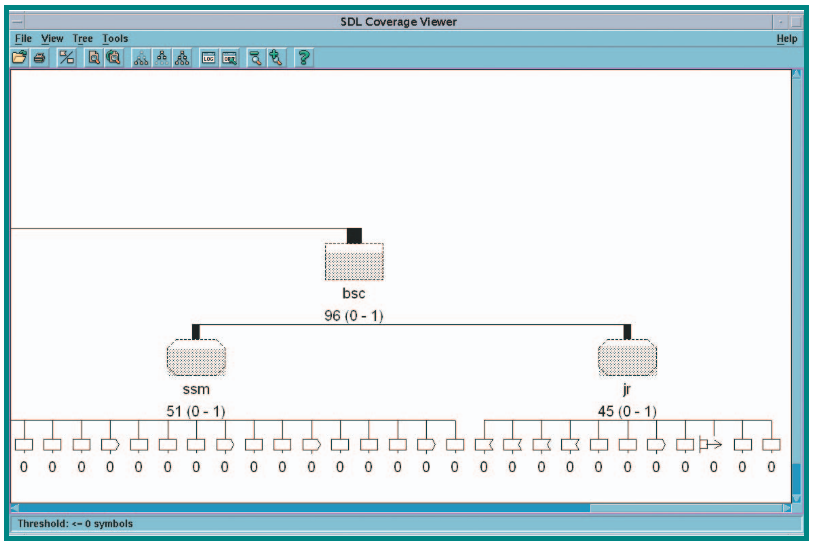


Figure 6. Part of an SDL symbol coverage tree for the subsystem bsc in the demo model. Leaf nodes represent SDL symbols, and different shapes represent different SDL symbols. White leaf nodes indicate the SDL symbols not covered, while grayed leaf nodes (not shown in this figure) indicate the SDL symbols covered. The number associated with a leaf node represents the number of times the corresponding SDL symbol was executed during MSC execution (zero indicates the symbol was never executed).

Automatic code generation

Like other MDD processes,[5] TDM is model-centric in that a modeling tool automatically generates the code from a design model.

Typically, we install and use a modeling tool to generate code on a generic machine (Solaris in this case), but the target machine is a different platform. In this case, the automatically generated code for simulation will differ from the automatically generated code for the target platform. In addition, we can't do performance testing through simulation. Thus, target platform testing is necessary even though the simulation results indicate that the design is correct in terms of meeting system and subsystem requirements—that is, passing MSC execution. Automatic testing tools such as the Telelogic TAU TTCN Suite are available for target platform test automation. By using TTCN tools and a related MSC-to-TTCN test case translation tool, the same set of MSCs used in the system and subsystem testing with simulation can be modified and then transformed into TTCN test cases for the system and subsystem testing on the target platform.

Automatically generated code is test driven because it's subject to testing on the target platform, and the issues (such as auto code defects and performance issues) revealed during target platform testing will trigger code regeneration after the issues have been resolved by tracing them back to either design (modeling) or code-generator issues (defects or enhancements).

Comparing TDM and a test-driven process

As I mentioned earlier, TDM aims to use the test-driven idea underlying a typical test-driven software development process such as XP to enhance a typical model-driven software development process for agility and quality. Thus, we can compare many TDM process activities, artifacts, or characteristics with those of another test-driven process (see Table 1).

Table 1

Comparing test-driven modeling with a test-driven process (such as Extreme Programming)

Correspondence		**TDM**	**Test-driven process**
Corresponding process activities	Implementation	Modeling	Coding
	Testing	Simulation	Test-case execution against real application code
Corresponding process artifacts	Test case	MSC	Test-case implementation (for example, a set of test driver objects calling a set of application object methods)
	Test suite	A set of MSCs	A set of test-case implementations
Corresponding process characteristics	Test driven	Create MSCs first	Create test cases first
	Test automation	Automatic MSC execution using a tool (such as Telelogic TAU)	Automatic test-case execution using a tool (such as JUnit for Java)
	Regression testing	Execute all the MSCs for the current and previous iterations	Execute all the test-case implementations for the current and previous iterations
	Iterative	Multiple development iterations	Multiple development iterations
	Incremental	Each iteration implements more system functionality on top of previous iterations	Each iteration implements more system functionality on top of previous iterations

Practical experiences

Here we discuss our experiences in modeling-tools selection and our implementation of a project using TDM with the selected modeling tool.

Tools selection

A key factor in adopting TDM with success is the modeling tool's capability. Many commercial modeling tools are available, but not all of them will let us effectively implement TDM. The following were the major criteria in tools selection for the project:

- *Full code generation.* In TDM, modeling is coding—that is, developers only work with models (such as SDL or UML diagrams) and code will be automatically generated from models. This requires that the modeling tool be able to generate fully executable code rather than a skeleton of code as certain modeling tools do.
- *MSC verification.* To be test driven in TDM, MSCs must be used as test cases. This requires that the modeling tool we use can execute MSCs—that is, given an MSC, the tool can automatically verify whether or not the model (design) can reproduce the MSC. Certain modeling tools only support MSC tracing—that is, generating an MSC based on an interactive simulation's execution. Then, visual comparison is necessary to verify whether the generated MSC meets requirements. These tools can't implement TDM effectively.
- *Test automation.* TDM requires regression unit testing—that is, in each iteration, not only the MSCs assigned to that iteration but also all the MSCs implemented in previous iterations must be executed. Thus, there can be numerous MSCs. This requires that the modeling tool we use can execute a set of MSCs without human intervention.
- *Standards compliance.* To avoid being locked into a particular tool vendor, the modeling tool must support internationally standardized modeling languages (such as SDL and UML). In addition, the tool vendor must support the latest modeling language, UML 2.0, in the near future.
- *CPU performance.* Our project has specific CPU performance requirements from customers. The code that the modeling tool generates must meet these requirements. Motorola's historical data indicates that tool vendors tend to fail to enhance their modeling tools in time if a CPU performance problem occurs with the code that their tools generated. In some cases, tool vendors can't enhance their modeling tools to meet our CPU performance needs. We avoid this difficulty by using a Motorola proprietary automatic code generator. To this end, the modeling tool we use must be able to interface with the Motorola code generation tool so we can replace the modeling tool's built-in code generator with the Motorola code generator when necessary.

Based on these criteria, we found that Telelogic TAU SDL Suite was the only tool that met all the requirements.

We observed one limitation with Telelogic TAU SDL Suite. Currently, the simulator generated from an SDL model by Telelogic TAU SDL Suite doesn't let the data value associated with a signal or message in an MSC change without breaking the MSC execution. Hence, MSCs are sensitive to model changes (especially the changes to data types) because the execution of some MSCs can fail due to those changes. It can be time consuming to fix all broken MSCs if the number is large. We can resolve or alleviate this problem by defining macros in MSCs to centralize the modification of data values.

Telelogic TAU SDL Suite was the only tool that met all the requirements.

Results

TDM was applied in the project to develop the core components of a large telecommunication system in Motorola using SDL and Telelogic TAU SDL Suite. The system consists of nine subsystems that are deployed as separate operating system processes. Six of these subsystems are automatically generated, and the other three (including user interfaces) aren't suitable for SDL modeling. More than 60 people work on this project, divided into groups. Some external groups are in different geographical locations. A big group is subdivided into multiple teams, and each team consists of two to 10 developers. Different teams are responsible for developing different subsystems. The members in a typical team sit close to each other and often work together. Weekly face-to-face group meetings foster communication among teams within a group as do weekly conference call meetings among groups.

A modeling and code-generation coordinator helps coordinate modeling and code-generation activities across teams and groups.

Practical experiences demonstrate

- TDM can improve productivity significantly compared with a traditional plan-driven development process by automatic code generation. For instance, some metrics data from Motorola's iDEN division indicated that this project achieved approximately five times productivity improvement in terms of KAELOC/Staff Month.
- TDM can improve quality level significantly by comprehensive simulation and coverage analysis. For example, some metrics data from Motorola's iDEN division indicated that this project achieved approximately 20 percent better phase containment in terms of the number of defects that escaped from the modeling phase into the target platform testing phase. For some projects in Motorola's iDEN division, we achieved from two to three times better phase containment up to zero defects in target platform testing.

Telelogic TAU SDL Suite supports simulation (MSC execution) at different levels (system, subsystem, and active object). This lets TDM apply the test-driven mechanism at both the system and subsystem level.

Telelogic TAU SDL Suite supports symbol and state-transition coverage trees. The symbol (that is, modeling element) coverage tree is similar to the traditional code path coverage, but the state-transition coverage tree is unique for MDD. A state-transition coverage tree shows which combinations of states and signals are covered. With this project, we can typically achieve close to 100 percent symbol coverage by performing coverage analysis if we don't use third-party components (such as SDL abstract data types). However, the achievable state-transition coverage percentage level can be low in certain circumstances. For example, SDL lets asterisk states and input signals indicate any states and signals. An asterisk state immediately followed by an asterisk input signal in SDL means that any state can handle any input signals in a state-transition chart. This can make the state-transition coverage percentage extremely low for a reasonably large model with many states and signals. We can resolve this problem by temporarily replacing an asterisk state or signal with a concrete representative state or signal during coverage analysis.

In theory, we can use TDM to develop projects of any size. However, like other MDD processes,[4,5] from a return-on-investment perspective, the major benefit of adopting TDM normally comes from reusing the process to develop multiple releases of an application system or related systems and subsystems. It might not make economic sense to adopt the process for a one-time small project due to process start-up overheads (such as a modeling tool license fee and learning curve).

The practical experiences of the project show that the Telelogic TAU SDL Suite meets our needs in implementing TDM. The TDM process and the modeling tool Telelogic TAU SDL Suite helped us successfully start the development of the first release of our product. Two new product releases are already underway using the same development process and modeling tool. To maintain the current product releases and to develop new releases, we expect to use the same process and tool in the foreseeable future. A similar process has also been successfully used in other organizations within Motorola.

Acknowledgments

I thank Associate Editor-in-Chief Stan Rifkin for his recommendation, careful review, and proofreading. I also thank the reviewers for their constructive comments and Motorola managers for approving this article's publication.

References

1. K. Beck, *Extreme Programming Explained*, Addison-Wesley, 2000.
2. G. Succi and M. Marchesi, *Extreme Programming Examined*, Addison-Wesley, 2001.
3. K. Auer and R. Miller, *Extreme Programming Applied*, Addison-Wesley, 2002.
4. I. Jacobson, J. Rumbaugh, and G. Booth, *The Unified Software Development Process*, Addison-Wesley, 1999.
5. B. Douglass, *Real-Time Design Patterns*, Addison-Wesley, 2003.
6. J. Ellsberger, D. Hogrefe, and A. Sarma, *SDL—Formal Object-Oriented Language for Communicating Systems*, Pearson Education, 1997.

About the Author

Yuefeng Zhang is a distinguished member of the technical staff at Motorola. His research interests include model-driven development, agile software development process, and modeling tools and their applications to automatic code generation. He received his PhD in computer science from the University of Western Ontario, London, Ontario, Canada. Contact him at iDEN BSC, iSD/GTSS/Motorola, Arlington Heights, IL; yzhang1@email.mot.com.