



Template Based Transformation

Complete Control Over Generated Code

Version 2.0
May 24, 2004

PathMATE™ Series

Pathfinder Solutions LLC
33 Commercial Street, Suite 2
Foxboro, MA 02035 USA
www.PathfinderMDA.com
888-543-7222

Table Of Contents

PathMATE Overview	iii
1. Introduction	1
2. Development Process Overview	2
Analysis	2
Design and Translation	3
3. Implementation Code Templates.....	5
Example 1: Class Header	6
Example 2: Class Data Members	7
Example 3: State Action Function Declaration	8
Example 4: Event Generate Action Language.....	9
4. Summary.....	10
5. References	11

PathMATE Overview

This overview introduces Model Driven Architecture (MDA) and the PathMATE™ tools that make MDA work. MDA and PathMATE move you from writing and debugging code to developing and testing the logic of a high performance system. Over years of rigorous refinement in several industries, PathMATE tools have proven their value in rapid and effective software systems development.

PathMATE Toolset

The PathMATE Model Automation and Transformation Environment includes all the tools required to transform your MDA models into high-performance systems (Figure 1).

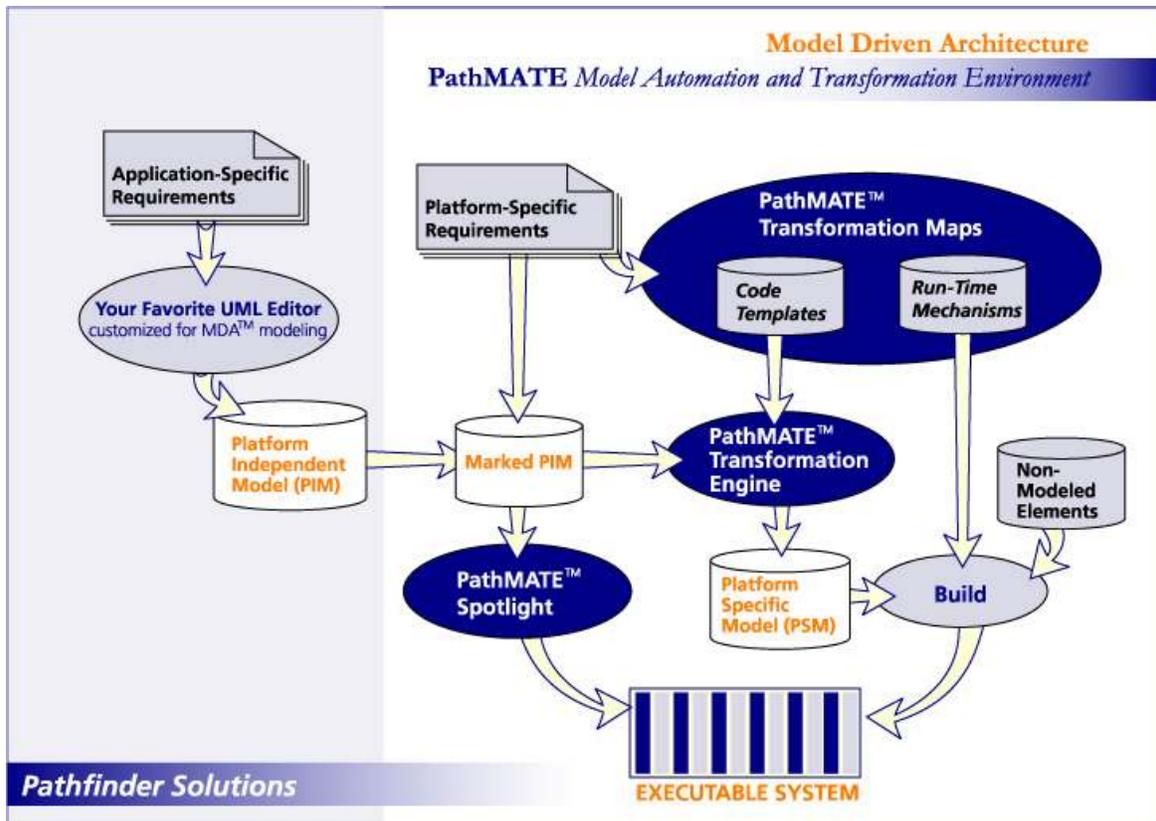


Figure 1. PathMATE Toolset

The three parts of the PathMATE toolset cooperate to turn your models into executable systems:

- *Transformation Maps* – Generate C, C++, or Java software with off-the-shelf Transformation Maps, or create custom maps to drive output for other languages or specific platforms.
- *Transformation Engine* – The Engine transforms platform-independent models into working, embedded software applications.
- *Spotlight* – Verify and debug your application logic with Spotlight, the most advanced model testing environment available.

No other MDA transformation environment offers a more open or configurable set of development tools, designed to meet the requirements of systems engineers.

How PathMATE Works

Use Model Driven Architecture to build complex embedded systems that meet rigorous standards for speed and reliability. MDA works because it separates what the system does from its deployment on a particular platform. PathMATE adds these advantages:

- *Greatest architectural control* – A highly configurable Transformation Engine enables you to optimize output for resource-constrained platforms.
- *Clean separation of model and code* – Conforming to the MDA paradigm, PathMATE models contain no implementation code. That gives you fast and flexible deployment and migration capabilities.
- *Configurable, target-based model execution and testing* – Preemptively eliminate platform-specific bugs, minimize quality assurance resources, and accelerate development.
- *Lowest cost of ownership* – Integrate PathMATE with your existing UML editor. Build on your previous investment in training and software.
- *Speed* – Even large transformations take just seconds with PathMATE. That enables highly iterative model development, and rapid transformation and test cycles.

Try the demonstration software available at www.PathfinderMDA.com to get started quickly and easily.

1. Introduction

Model-based software development using UML has been applied for all types of software, from business applications to embedded systems. As modeling approaches and their supporting tool technologies have matured, the ability to execute models and generate production-grade code have spread as well. However the generation of implementation code for embedded systems has not always resulted in a system with the high level of run-time performance required.

Embedded software development teams often work in challenging execution environments, striving for high levels of system performance. Template-based transformation of UML models to code provides the control these teams need to ensure they will achieve required levels of performance. The heart of this technology is a transformation engine that reads UML analysis models and forms the output based on a separate set of rules – in the form of code templates. A single transformation engine can take a project's models, and generate different implementations with widely varying characteristics – even different implementation languages – just by using different template sets.

By choosing an appropriate off-the-shelf template set a project can attain reasonable efficiency, but sometimes this is not enough. By modifying implementation patterns in code templates, or creating new patterns of their own, the team can utilize the implementation architecture and project-specific optimizations they need to get the job done. A template-based approach to code generation results in an implementation with optimized performance, and still retains the simplicity, maintainability and extensibility of a modeled system.

2. Development Process Overview

The effective application of UML models to software engineering for challenging applications – especially in the embedded context – requires a development process that will ensure:

- models are unambiguous and complete
- the resulting system implementation can be optimized without impacting the models
- the overall architecture of the system is maintained by the process through multiple releases and requirements evolution

A transformational approach is used to maintain the separation of models from implementation, which facilitates the achievement of these goals. Specific aspects of this type of process are introduced below.

Analysis

The process of modeling a solution to a problem in terms of the problem itself is called *Analysis*. Effective Analysis models are rigorous and complete, and largely free of implementation bias. The Unified Modeling Language™ (UML) is currently the most popular notation for software Analysis. The work products produced during Analysis are:

- *Domain Model*: This is a UML class diagram showing the highest level decomposition of the system into areas of separate subject matter, called *Domains*. Domains are shown as Packages, and Dependency arrows show *Bridges* - the flow of requirements between Domains. A Domain can be modeled, or it can be developed via other means: hand-written code, legacy code, generated from another source, imported from a library, etc. Key system-level scenarios are captured with UML interaction diagrams to show interactions between Domains.
- *Class Model*: For each Domain that is to be Analyzed, a UML class diagram is used to define the Classes that form the structure of the Domain. Classes have associations with other Classes, and inherit from other Classes.
- *Scenario Model*: Key scenarios for this specific Domain are captured with UML interaction diagrams to show interactions between Domain services (operations), Class services (methods), Class event messages, and services of outside Domains used in this Domain.
- *State Model*: For each Class that receives events, a UML State Diagram is used to capture the Class lifecycle, defining state-dependent behavior for that Class.
- *Actions*: For each Domain service, Class service, and State action, a detailed, unambiguous behavioral description is created. This is expressed in an Action Language that conforms to the UML Action

Semantics, an analysis-level “programming” language that provides a complete set of Analysis-level execution primitives without biasing the implementation. By expressing behavioral detail in Action Language, considerable freedom is retained until the transformation phase for how each analysis primitive is implemented – critical for optimization.

Design and Translation

Design is the creation of a strategy and mechanisms supporting the mapping of analysis constructs to a run-time environment – implementation. Design is conducted in a different concept space from analysis. The separation of Analysis from implementation supports the development of the Design independently of the analysis activities, retains the simplicity of the Analysis models, and allows the tuning and optimization of the implementation without impacting the analysis.

Transformation is the process where the UML models for each analyzed domain are mapped to implementation through Design strategies – code templates. Design is conducted at two levels:

- *Structural Design*: Identify the execution units – the threads, tasks, and processes – of the system, and allocate them to processors. Also allocate various subsets of the models to each of the units. At this level, the overall *implementation strategy* is defined.
- *Mechanical Design*: Develop detailed patterns (expressed in code templates) to map analysis to the required implementation, and build base mechanisms to support this implementation. At this level, specific constructs are developed to support the overall strategy defined during Structural Design.

The greatest benefits to be gained from transformation – flexibility and simplicity – are derived from the fundamental separation of Analysis from Design. The Analysis is free of any specific implementation complications, and the Design is focused on how to achieve the execution capabilities required – especially performance – in a specific execution environment.

This paper outlines a template-based approach for Mechanical Design where a transformation engine takes the semantic information in the analysis models and produces a set of executable source code through the application of templates. This generated code is a complete implementation for its Analyzed domain - it requires no further creative input beyond the models and implementation code templates. The generated code for the Analyzed domain is combined with other domains that may not be modeled: code generated from other sources, such as GUI IDEs, hand-written code, off-the-shelf libraries, or third-party developed elements.

By untangling the normal interweaving of the different concerns of problem space subject matter, implementation architecture, and execution platform, a transformational approach allows a simpler and

more effective treatment of each separate concern. This type of approach affords the development team flexibility at a strategic level:

- Freedom to decide which components (domains) are modeled, and which are not
- Complete control over generation of implementation code from modeled components
- Flexibility to change implementation strategy separate from models at any time
- Complete Design layer reuse by applying the same transformation to different applications
- Complete application reuse by translating unchanged analysis models to new execution environment (platforms, languages, technologies, etc.)

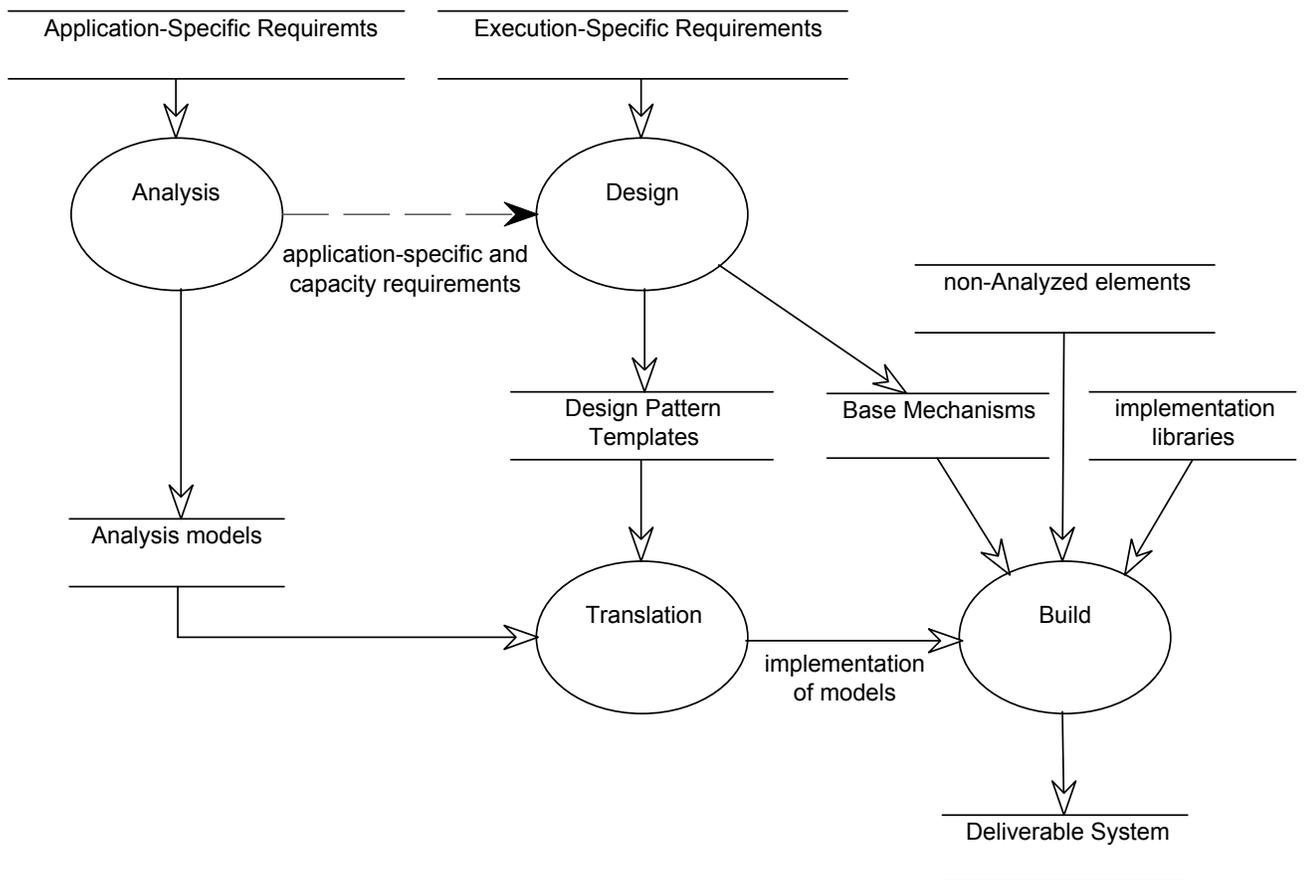


Figure 2. Analysis, Design, and Translation Processes

3. Implementation Code Templates

A template is a Design pattern captured in a form understandable to a transformation program. A complete set of code templates is provided to a transformation program which reads the semantic information from UML Analysis models, and then produces a set of target documents – code.

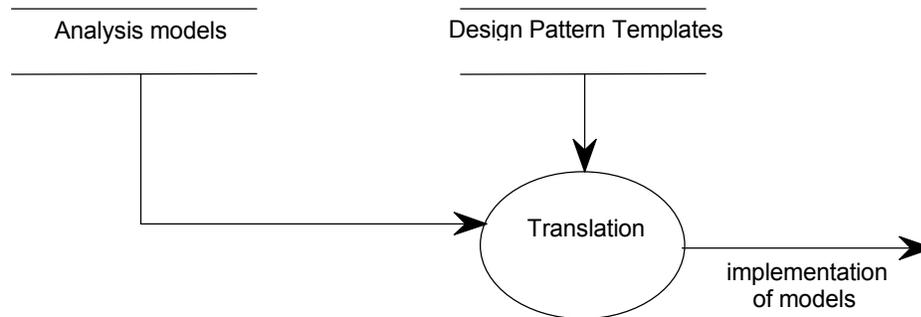


Figure 3. Translation Process

To be effective for code generation, a template notation must support two key capabilities:

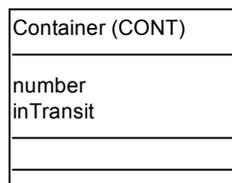
- The presentation of model information through substitution fields
- Facilitate the navigation and processing of UML model elements so implementation strategy decisions can be made based on an investigation of actual model information.

When the term “template” is used, generally the first capability is assumed. However for proper generation of efficient implementation code, the capability to easily gather diverse elements of model information and derive semantic conclusions is critical.

Example 1: Class Header

This example shows some of the basic elements of applying Design patterns through templates. A subset of the two transformation inputs are shown: a UML Analysis Class, and the pattern that maps it to an implementation C++ class. The result is a C++ class header. Please note how the Analysis elements - domain prefix, class name, class description - are mapped into code. (Please note that in this template fragment, a variable "object" is used to refer to UML Class information.)

Analysis: UML Class



Description: A bowl, disk, pan, baking sheet, or other mixing or cooking vessel.

Template: UML Class definition -> C++ class header (segment)

```
//=====
class [domain.prefix]_[object.name] : public \
    [FOREACH parent IN object.superTypes SEPARATOR ", public "] \
    [object.domain.prefix][parent.prefix][ENDFOREACH]
{
    /* [object.description] */

public:
    // This is the list of all instances of objects of this type
    static PfdBaseList instanceList;
    . . .
```

Generated Code: Container C++ class header (segment)

```
//=====
class FP_CONT : public          FP_PA
{
    /* A bowl, disk, pan, baking sheet, or other mixing or cooking vessel.
    */

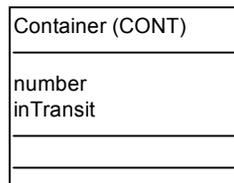
public:
    // This is the list of all instances of
    //objects of this type
    static PfdBaseList instanceList;
    . . .
```

In the example above, a simple substitution (lexical replacement) is done in the template to build a C++ class definition for the example UML Class.

Example 2: Class Data Members

This example shows the generation of member declarations for attributes of a UML class.

Analysis: UML Class



Template: UML Class Attributes -> C++ class data members (segment)

```

. . .
public:
    // Attributes:
    [FOREACH attribute in object.attributes]
        /* [attribute.description] */
        [attributedataType] [attribute.name];
    [ENDFOREACH /*attribute */ ]
. . .

```

Generated Code: Container C++ class header (segment)

```

public:
    // Attributes:
    /* External identification number. */
    Integer number;
    /* Flag indicating if the container is moving towards
       its location, or if it has reached it. */
    Boolean inTransit;

```

In the example above, an iteration is required over the attribute information – this shows a simple form of model information navigation. In combination with the template segment from example 1 (and other segments), this template segment builds a part of the class definition. There are other templates which emit no code at all, and just provide the executive control over the complete template set, building the generated code base from the required pieces in the appropriate order.

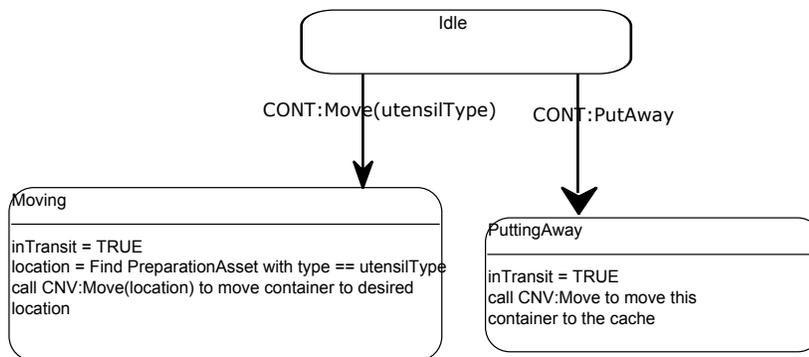
The bulk of the complexity in a product-grade template set for high performance code actually is dedicated to the *selection* of the proper implementation coding pattern to be applied in a given modeling situation. As the translator works through the models to generate code, the analysis modeling context is considered in conjunction with specific transformation hints or “coloring” - provided as project-specific properties on modeled elements – and a determination is made as to which specific pattern is to be applied. The bulk of project-specific

template customizations involve supplementing the available pattern set with a new pattern template, and then augmenting the selection logic to know when to apply the new pattern. All of this selection logic is expressed in “template” notation, as well as the patterns themselves. *Design Pattern Templates* offers a more detailed look into this aspect of template-based transformation (see *References* below).

Example 3: State Action Function Declaration

This example shows the generation of a declaration of a state entry action for a UML state model.

Analysis: UML State Model (Fragment)



Template: UML Class Attributes -> C++ class data members (segment)

```

// Now the state action functions.
[FOREACH state in object.states]
  [EXPAND act_pro (class_name, state)];
[ENDFOREACH /* state */ ]
    
```

Generated Code: Container C++ class header (segment)

```

// Now the state action functions.
void FPCONT::doMoving (PfdEvent *event);
void FPCONT::doIdle (PfdEvent *event);
void FPCONT::doPutting_Away (PfdEvent *event);
    
```

In the example above, the template *act_pro* is invoked to emit the action function profile.

Example 4: Generate Event Action Language

This example shows the generation of an event to a UML state model.

Analysis: Action Language Generate Statement

```
GENERATE AR:SetupComplete() TO (new_recipe);
```

This statement, from the FoodPrep (FP) domain, specifies that the ActiveRecipe (FP_AR) class's SetupComplete event is sent to the instance specified by the new_recipe reference. In implementation, an event is implemented as a class defined within the receiving class's definition. The action language statement above is translated to a call to the FP_AR::SetupComplete::generate() method. This example shows the generate() method itself.

Template: UML Event Generate Method

```
void [event_class_name]::generate([class_name] *dest\
[FOREACH evdi IN event.parameters]
, [EXPAND gen_type (evdi.dataType)] [evdi.langId]\
[ENDFOREACH /* params */]
)
{
    // Create a new instance of the event
    [event_class_name] *ev = new [event_class_name] (dest);
    [ENDIF /* create */]
    // Now fill in the parameters
    [FOREACH evdi IN event.parameters]
    ev->[evdi.langId] = [evdi.langId];
    [ENDFOREACH /* params */]

    // Send the event
    ev->send();
}
```

Generated Code: Event Generate Function Implementation

```
void FP_AR::SetupComplete::generate(FP_AR *dest)
{
    // Create a new instance of the event
    FP_AR::SetupComplete *ev = new FP_AR::SetupComplete (dest);
    // Now fill in the parameters

    // Send the event
    ev->send();
}
```

The static method generate() creates an instance of the SetupComplete event and places it on the system's event queue with the send() method.

4. Summary

The solution to the core application “problem” is expressed in UML analysis models. Complete, executable analysis models provide an excellent basis for developing a problem solution that is robust and durable yet simple and flexible. These models are translated to implementation code using template-based transformation. Off-the-shelf template sets can produce efficient implementations, but with templates the project team has complete control over the implementation. This separation of the implementation concerns from the problem-space concerns maintains the simplicity, maintainability and flexibility of the models, and allows the freedom to pursue a high-performance implementation.

This paper is intended as a brief overview of transformation of UML analysis models to implementation code. For an in-depth investigation of a complete, executable transformation example, contact Pathfinder Solutions to receive the Robochef sample model and accompanying C++ code templates and base mechanisms. For a paper discussing the determination and application of project-specific optimizations in the context of template-based transformation, including detailed examples, please request “Design Pattern Templates: A Strategy for Optimizing Embedded System Performance” from Pathfinder Solutions.

5. References

For more information about PathMATE, please call Pathfinder Solutions at 508-384-1392, e-mail us at info@pathfindermda.com, or visit us at www.pathfindermda.com. You may wish to refer to the following sources:

On PathMATE:

Model Based Software Engineering: Rigorous Software Development with Domain Modeling, Pathfinder Solutions, 2004 (this paper is available from www.pathfindermda.com)

Design Pattern Templates: A Strategy for Optimizing Embedded System Performance, Pathfinder Solutions, 2004; (this paper is available from www.pathfindermda.com)

On the UML™:

The Unified Modeling Language User Guide, Grady Booch, James Rumbaugh, Ivar Jacobson, Addison Wesley, 1999; ISBN 0-201-57168-4

UML Distilled, Martin Fowler, Addison Wesley, 1997; ISBN 0-201-32563-2

UML Summary Version 1.1, Object Management Group, Inc. 1997 (this paper is available from www.omg.org)

UML™ is a trademark of Object Management Group, Inc in the U.S. and other countries.