A THREAD-SAFE IMPLEMENTATION OF A META-PROGRAMMABLE DATA MODEL

By

Daniel Balasubramanian

Thesis

Submitted to the Faculty of the

Graduate School of Vanderbilt University

in partial fulfillment of the requirements

for the degree of

MASTER OF SCIENCE

in

Computer Science

May, 2008

Nashville, Tennessee

Approved:

Professor Gabor Karsai

Professor Sandeep Neema

PREFACE

This work was partially supported under the DARPA/IXO MOBIES program, Air Force Research Laboratory under agreement number F30602-00-1-0580 and NSF ITR on "Foundations of Hybrid and Embedded Software Systems."

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER I

INTRODUCTION

As computer based systems of all types have grown in complexity, the use of models has become common for design, analysis, and testing. The term "model" is vague concept and can refer to a variety of things depending on the context. In general terms, a model can be defined as an abstraction of some real-world concept. By abstraction, we mean that not all characteristics or features of the actual entity are expressed in the model. For instance, a mathematical model describing the forces acting upon a skyscraper may not contain information about the aesthetics of the building, while a model developed by an architect may contain those sorts of details.

The type of model that one needs depends primarily on how the model will be used. For instance, if one is working on a suspension bridge and wants to know what the finished product will look like, then an actual miniature replica of the bridge, built to scale, may be what is desired, whereas if the weight capacity of the bridge needs to be determined, then most likely a precise physical and mathematical model is more relevant. In the domain of software systems, models can often represent the entire system under development, or at least everything that is needed to generate a significant portion of the system. For instance, a finite-state machine can be used to represent

a behavioral view of a system, while a UML Class Diagram [15] can be used to represent the various components of a system from a structural point of view.

In general, there may be several different types of models that can be used for the same purpose; the only requirement is that the model must be able to express all relevant information. If the model is not capable of holding all of the necessary information, then its usefulness is greatly diminished, or, in the worst case, the model may be useless. On the other hand, a model that is overly expressive may impose too much overhead or be too cumbersome to be used effectively and efficiently. What is particularly useful is a modeling formalism that is customizable, in the sense that the amount of detail that can be captured in a model can be specified by the user. Thus, what is desirable is not only a way of defining models, but a way of specifying what can be defined in models.

One can accomplish this task of describing what can be captured in a particular model through the use of models: one first defines a model that describes the details and features that can be captured in other models, and then uses these models to define the particular system. The models that are used to describe the entities and relationships found in other models are known as meta-models: models used to describe models (the prefix "meta" means one level of description higher). When the modeling language being used to develop a system consists of only the relevant concepts found in that particular domain, then we refer to the modeling language as a

"domain-specific" modeling language (DSML). DSML-s have the advantage of expressing precisely the concepts and relationships present in a particular domain, and thus have the benefits listed above: they can be used effectively and efficiently, and they meet the requirement listed above of being able to capture and express all necessary information. The disadvantage of using DSML-s is that their development is often time-consuming [5]. Fortunately, several tools have been developed in order to reduce the development time of DSMLs. One such tool is the Generic Modeling Environment (GME) [2], which helps facilitate the rapid development of DSML-s.

Model integrated computing (MIC) [13] is a design methodology for computer based systems which advocates the use of domain-specific models during all stages of system design, from initial design to testing to implementation. The process begins by first studying the target domain and understanding the entities and their relationships. After the concepts are well-understood, then a DSML is defined, which allows domain-specific models to be created. After domain-specific models have been created, one needs a way of transforming them into other things. In order to generate any sort of useful artifact (e.g., code, documentation) from a domain-specific model, there must be some way of accessing the information contained in the model and transforming it into another form. Tools that access models to produce some other sort of artifact are known as *model interpreters*. Without model interpreters, we cannot generate any form of useful artifact from our models, and we are very limited in what we can do with our models.

The Universal Data Model (UDM) [10], is a meta-programmable framework for realizing all steps of the MIC process: meta-modeling, modeling, and writing model interpreters. That is, UDM is a single framework that allows one to define DSML-s by creating meta-models , create instance models using the DSML, and also write model interpreters for translating the domain-specific models into other artifacts. The term "meta-programmable" is used because the user can define their own meta-model, and then configure UDM to build and access (domain-specific) models that conform to this meta-model.

To define a meta-model, a language for defining meta-models is necessary. In other words, a *meta*-meta-model is needed. UDM uses a simplified version of UML Class Diagrams [15] as its meta-meta-model. In this manner, a UDM compatible meta-model is an instance of a UML class diagram. From this point of view, UDM can also be viewed as a way of quickly implementing user-defined data-structures: each class in the UML class diagram, along with the associations (relationships) between these classes, form a data structure. UDM also has the benefit of providing multiple ways of persisting these data structures, including persistent XML, a native memory format, and a binary format compatible with the GME tool-suite (described above). Another approach would be to use a different meta-meta-model, such as the Meta Object Facility (MOF) meta-model [16], but the UDM approach is

simpler. For instance, MOF lacks the ability to have any sort of state information directly incorporated into an association, and UML is generally considered to be more intuitive [14].

UDM consists of two major components: a code generator that generates a programmatic description of a meta-model (i.e., it creates a data structure definition of the meta-model), and a set of base libraries. The process works the following way: the user defines a meta-model, usually in a visual manner (though not necessarily), and uses the UDM code generator to produce compilable code, currently either C++ or Java, that reflects the structure of the meta-model. Figure I.1 shows a high level overview of this process. The generated code reflects the structure of the user's meta-model (i.e., their UML class diagram) by producing either a C++ or Java "class" definition for each class in the UML class diagram, and also generates methods and attributes inside these classes to reflect the structure of attributes and relationships between the classes in the UML class diagram. Each generated class inherits a set of generic methods from a class defined inside the UDM base libraries, and in this manner, the UDM libraries allow the generated classes to be treated in a generic manner. For instance, this base functionality inherited by each class includes methods to get attributes and access other objects with which an object has a association.

The current implementation of UDM allows one to use models only in single-threaded code. Assuming that a meta-model has been defined, new

models that conform to (i.e., are described by) this meta-model can be created and existing models can be accessed in a programmatic way using the core UDM libraries. When we say that UDM can only be used in single-threaded code, we mean that one cannot perform multiple operations on a model concurrently in a deterministic manner. In the context of programming languages and executable code, this mean that the UDM core libraries, which are used to build new models and access existing models, are not "thread-safe" (this term is defined in detail in Chapter 2).

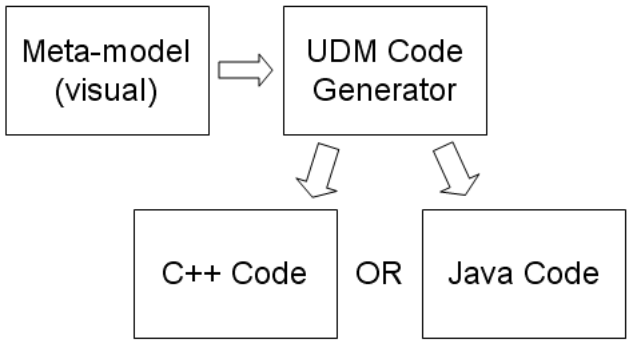

Figure I.1: High-Level Overview of Using UDM

This issue of thread-safety limits the ways in which UDM can be used. As computing technology advances, this becomes a large bottleneck in terms of performance. This may be especially true if UDM is being used as a way of implementing data-structures, as we may want to perform multiple operations on these structures in parallel. For instance, if we need to create several objects that will be "contained" inside another object, the ability

to do this from multiple threads could greatly improve performance if the processor being used has multiple cores. However, we cannot manipulate the data structures being used to hold the newly created objects from multiple threads unless they are guaranteed to be thread-safe and ensure protection from data corruption.

Problem Statement

The existing implementation of UDM has been used successfully in several projects (e.g., [3], [20]). However, as explained above, the current machinery does not allow UDM to be used in a multi-threaded environment. What we desire is a way of implementing the UDM core libraries in a thread-safe manner that is transparent to users: locking and unlocking of data structures happens automatically without any extra effort on the part of the user.

The goal of this work is to understand what thread-safety mechanisms will be sufficient to allow the UDM framework to be used in a multi-threaded environment. Specifically, we want to determine the exact synchronization and locking mechanisms that are powerful enough to provide mutual exclusion, and yet still allow a high performance gain over a non multi-threaded transformation. A correct solution will be one in which UDM models can be created and modified from multiple threads deterministically without any explicit synchronization mechanisms written by the user.

The work described in this thesis consists of two main parts:

1. A code generator that will generate user code in the C# programming language in a manner consistent with the existing UDM C++ and Java code generators.

2. A set of core UDM libraries, implemented in the C# programming language, that are thread-safe and that can be used by the code generated by the new code generator.

The rest of this thesis is outlined as follows. Chapter 2 gives background information about UDM and describes some similar object models. Chapter 3 gives details about the implementation of the new code generator and core UDM libraries. Chapter 4 describes the results of some tests using the new code with regard to efficiency, and Chapter 5 gives a summary of the work along with future plans.

CHAPTER II

BACKGROUND

This chapter gives background information on the structure of UDM, the meaning of thread-safety, and the ways in which the previous implementation of UDM was not thread-safe.

UDM Background

UDM, the Universal Data Model [1], is a meta-programmable framework that is used for both modeling and model transformations. The typical use case scenario of UDM involves the following steps.

1. Using UML Class Diagrams [15], define the domain. The entities of the domain are modeled as UML classes, which may contain attributes of the following types: string attributes, integer attributes, real-valued attributes, and boolean attributes. The relationships between elements of the domain are modeled using UML associations. These relationships can be arbitrary associations (in which the two classes involved share some sort of relationship with one another), or can be compositions (in which one class can be used to contain the other classes in a parent-child relationship).

2. Use the UDM code generator to generate a programmatic description (in either C++ or Java) of the classes and associations in the UML class diagram. If a visual notation was used to build the Class Diagram, then this is first translated to an XML description for use as input to the UDM Code Generator (the reason for this is so that one can use different visual tools, each of which usually has its own binary format, to create the diagram, as long as the chosen tool can generate a UDM Code Generator-compatible XML file). For each class in the UML Class Diagram, a corresponding C++ or Java class is generated. This generated class contains methods to create objects of this type, as well as methods to retrieve and set children objects (compositions) and associations, as well as retrieve and modify attribute values.

3. Use the generated programmatic description together with the UDM base libraries to build and/or transform models of your system. This is accomplished by instantiating objects of the classes described above and then setting attributes and associations between the elements.

For the second step listed above (defining the domain), and when working with models of software in general, one often finds that a visual notation is more convenient than a textual one. For instance, in order to represent a multi-dimensional relationship between elements (e.g., a binary association), a visual notation is easier and more compact than a textual one, and is also more quickly understood by humans. For this reason, UDM allows the user to

define their meta-model (that is, their UML class diagram) using a graphical modeling environment. UDM provides supporting tools that allow one to use GME to visually define the class diagram and then automatically translate this visual description to an XML-description. In fact, any tool that can produce an XMI [17] compatible description of the UML Class Diagram can be used with UDM [1].

To build complex models formed of many small, individual objects, UDM takes the approach of having a relatively small number of generic concepts that form the building blocks for models. This allows distinct, domain-specific objects to be treated in a uniform manner, thus easing the burden on the programmer in many regards. The three main generic concepts around which UDM is layered are the following.

1. *UDM Objects*: All of the classes that the user defines in their UML class diagram are translated into a programmatic representation, currently either C++ or Java classes. Each of these user defined classes are derived from a generic **UDM::Object** class, which provides base functionality such as the ability to retrieve children and attribute values. These classes do not define the specific implementation of how to perform domain-specific operations on objects, such as retrieve children objects of a certain type. Rather, this class can be considered as a "wrapper" class that contains an inner "implementation" object that does handle the domain-specific concepts, such as setting a specific attribute of a class. These methods found in these "wrapper" classes

forward the work to the methods of the implementation object. This is similar to the handle-body idiom described in [12].

2. *UDM Datanetworks*: Datanetworks are used to store UDM objects. A Datanetwork is organized in a tree-like structure, with one "root" Object, from which all other UDM Objects in the Datanetwork can be accessed. There are three different types of implementations for Datanetworks, also known as "backends", which allow networks of objects to be stored in three different types of formats, including an XML format, a native binary format, and a binary format compatible with GME. Each of these different types of Datanetworks is derived from an abstract base Datanetwork class, so that from the user's point of view, the specific type of Datanetwork is transparent, in the sense that the same methods are called regardless of the type of Datanetwork.

3. *UDM ObjectImpls*: These provide the actual implementation of each user defined class. These implementation objects contain the data structures that are used to store both attributes and relationships with other implementation objects. From the user's point of view, UDM Objects are used primarily when working with models, and the implementation objects are encapsulated "behind the scenes" to implement domain-specific functionality. Corresponding to the three different types of Datanetworks, there are three different types of ObjectImpls: one for UDM Objects stored in an XML Datanetwork, one for

UDM Objects stored in the native binary format Datanetwork, and one for UDM Objects stored in a GME-compatible Datanetwork. Each of these different types of ObjectImpls is derived from a base ObjectImpl class, so that from the user's point of view, all ObjectImpls are treated in the same manner (only their internal representation, which is hidden from the user, varies).

The advantage of this generic approach is that UDM base libraries are kept as general as possible, meaning that they can be used with any domain-specific language the user defines. In a sense, the user customizes UDM to be used with their particular domain. In other words, UDM provides a *meta-meta* model to the user (which is the modeling language for creating UML class diagrams), and by using this language to define a UML class diagram that is generated into a programmatic representation, the user defines the *meta*-model. In this sense, UDM is a meta-programmable data model: the user defines the meta-model, and the UDM framework provides the functionality that allows models to be built, stored, and accessed.

In order to support the generic architecture described above, in particular the use of an inner "implementation" ObjectImpl class which has the data structures and functionality to store and retrieve both attributes and relationships with other objects, UDM must be able to describe the structure of the meta-model programmatically. The reason for this is because the ObjectImpl class, and also the UDM Object class, are defined by the UDM authors, and are not modified by the UDM code generator; rather, the

code must be generated in such a way that it can describe the structure of the meta-model so that the ObjectImpl class "knows" what functionality a particular ObjectImpl provides.

Representing Meta-Models Programmatically

What the user desires is a way to build domain-specific models programmatically, or in other words, a way to create instance models of a given meta-model using code. However, if the instance models are to be domain-specific, then there must be a programmatic description of the meta-model, so that the objects in the instance model are "described" by the meta-objects in this meta-model. In a similar manner, if one is going to create meta-models programmatically, one needs a programmatic description of the meta-meta-model. Fortunately, only two "meta" levels are needed. There are two main reasons for this. First, the meta-meta-model does not change frequently, and thus does not need to be reprogrammed often. Second, the concepts found in UDM's meta-meta-model, UML Class Diagrams, are sufficiently expressive to be able to describe themselves, so that the programmatic description of the meta-meta-model uses elements from itself.

UDM solves this problem by generating code that automatically constructs (i.e., builds and initializes) "meta"-Datanetworks, which are Datanetworks whose Objects are used to describe Objects in other Datanetworks. The Objects in the UML Class Diagram Datanetwork (i.e., the meta-meta-model) are used to describe the Objects in a Datanetwork that corresponds

to a meta-model, and the Objects in the Datanetwork for a particular meta-model are used to describe the Objects in instance models of that meta-model.

It was explained above that UDM uses UML Class Diagrams as its meta-meta-model. This means that from UDM's perspective, a meta-model can be viewed as an instance of a UML Class Diagram. In order to build a Datanetwork that will describe a meta-model (which can be seen as an instance of a UML Class Diagram), one first needs a programmatic description of UML Class Diagrams. In other words, a programmatic description of the meta-meta-model is needed.

This programmatic description of UML Class Diagrams (the meta-meta-model) is generated automatically in the following way. First, a UML Class Diagram that describes UML Class Diagrams is created. In other words, UML Class Diagrams are modeled using UML Class Diagrams (they have sufficient expressiveness to describe themselves). The primary concept in a UML Class Diagram that is expressive enough to describe the concepts found in multiple domains is the UML Class. For this reason, all concepts in UML, including associations and inheritance, are represented by a UML Class. Even the concept of a UML Class is represented by a UML Class! This self-referential description of UML Classes is the "base" of the UML self-referencing. Next, this hand-created UML Class Diagram is transformed into the corresponding XML representation.

When the XML representation of the visual self-representation of UML Class Diagrams is processed by the UDM Code Generator, a programmatic description (in C++ or Java) is generated. For every UML Class in this visual UML Class Diagram, a C++ (Java) class is defined in the generated code which has the same name as this UML Class (an uppercase "Class" refers to a UML Class, and a lowercase "class" will be used to refer to the generated programmatic C++ or Java class definition).

Each generated C++ or Java Class contains a static object [23] of type **UML::Class**. The purpose of this static object is to serve as a "meta-object": objects that are found in models one-level lower in the hierarchy contain a reference to this meta-object to describe that they are of this type. For instance, the meta-objects in the programmatic description of the meta-meta-model are used to describe the objects in the meta-model, and the meta-objects in the programmatic description of the meta-model are used to describe the objects found in an instance model. In this way, UDM conforms to the classic four-metalayer meta-modeling framework in which each level consists of instances of elements of the next higher level. Figure II.1 shows the four-metalayer framework and its corresponding UDM concepts.

As was stated earlier, every UDM Object contains an inner implementation object. The main role of this implementation object is to provide the specific functionality required by different types of domain-specific objects. For instance, the implementation object contains the data structures used to hold information about the associations an object has with other objects.

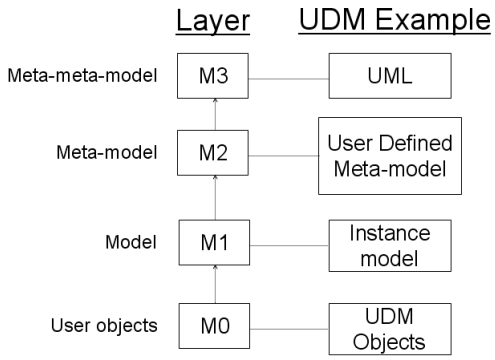

Figure II.1: Four Level Meta-Modeling Framework

Building a Model

In order to build an instance model corresponding to a meta-model, the user constructs a Datanetwork of objects whose type objects are found in the meta-Datanetwork. Obviously, the meta-Datanetworks need to be constructed before an instance model is created. Thus, the following steps are executed before accessing any user-level model.

- The Datanetwork of the meta-meta-model is initialized. This is the initialization of the generated code that describes UML Class Diagrams.
- The Datanetwork of the meta-model is initialized. This meta-Datanetwork is comprised of objects whose type-object is found in the meta-meta-Datanetwork, so that this meta-Datanetwork is an instance of a UML Class Diagram.

Both of these initialization functions are automatically generated by the UDM Code Generator, so that these two steps are performed transparently

by the UDM framework. After this initialization is completed, the user can access their model and use the generated domain-specific methods.

Related Tools

The Eclipse Modeling Framework

The Eclipse Modeling Framework (EMF) [6] is a Java framework for generating Java code from models. EMF takes as input a model defined using the XML Metadata Interchange (XMI) format [17] and outputs customizable Java code that corresponds to the model. The XMI definition of a model can be created in several ways, including defining the XMI document directly, using a graphical modeling tool that will export to the XMI format, and by annotating Java interfaces with model properties. After the input model has been defined, the EMF code generator creates a corresponding set of Java implementation classes.

EMF offers many similar features of UDM. EMF generates Java code, while UDM can generate either C++ or Java code. EMF takes as its input a model in XMI format. One can translate an XMI-compliant model into the input form necessary for the UDM code generator by using a tool provided with the UDM framework [1], so that UDM can generate C++ or Java code from an XMI-compliant model. Also, both EMF and the current version of

UDM can be used safely only in single-threaded code. That is, neither tool automatically provides of way of using its generated code in a multi-threaded environment. Both tools can also persist models in various formats. EMF supports XMI and schema-based XML serialization, while UDM supports XML persistence, a native binary format representation, and a binary format compatible with GME [2].

The two tools differ in the meta-meta-model that they use. As stated previously, UDM uses a simplified version of UML Class Diagrams as its meta-meta-model, EMF uses a subset of the MOF API [16] known as the "ECore" as its meta-meta-model. MOF is described in more detail below.

MOF

The Meta Object Facility (MOF) [16] is an Object Management Group standard for meta-data (i.e., data about data). The goal of standardizing the form for meta-data is that applications that interchange data with one another can be developed and integrated with one another more quickly. MOF is used as the meta-data standard for a number of other standards, including the Unified Modeling Language.

While the Eclipse Modeling Framework (described above) uses a modified subset of MOF as its meta-model, UDM relies on a simplified version of UML Class Diagrams as its meta-model. In practice, UDM's use of UML Class Diagrams instead of MOF as its meta-meta-model is not a limitation, and the ability to use models that are XMI compliant with UDM (see above)

increases the number of tools one can use to generate meta-models that can be used with UDM.

Multi-Threaded Programming

As computers have evolved throughout the years, multi-threaded programming has become common. In a loose sense, when a program has multiple threads, it means that the program can execute multiple instruction streams simultaneously. Multi-threaded programming differs from single-threaded programming because care must taken to synchronize access to shared data that can be accessed from multiple-threads. The canonical example that is often used to show how multi-threaded programming can lead to unexpected or incorrect results is usually a variant of the following situation. Suppose that a program has a single integer variable, x, initialized to zero, along with two concurrently executing threads. Suppose that both threads perform only the following code:

```
for(int i=0; i<10; i++)
{
   x++;
}
```

The user would most likely expect the value of the variable x to be 20 when these threads are finished executing. However, the code to increment the value of x may be something such as the following:

- Read the current value of x from memory and store it in a register.
- Increment the value of the register.
- Store the value of the register back into the memory location of x.

We can easily see that if these sequences of instructions are interleaved between the two threads, the value of x may be less than the expected value of 20 when these two threads are done executing.

This sort of scenario, often known as the mutual exclusion problem, can happen in any program which shares data between multiple threads if access to the data is not synchronized between the threads. For instance, consider the following simple scenario, in which UDM Objects are shared between multiple threads.

Suppose a program has two threads, each with access to the same UDM Object (call it y). As described above, UDM gives the ability to define relationships between various Objects. Suppose our Object y can have a "parent" relationship with multiple other UDM Objects, and that the two threads in our program try to add new objects to this relationship: thread one wants to make an Object (call it v) a child of y, and thread two wants to make another Object (call it w) a child of y. Without knowing anything about the internal mechanisms for storing these relationships, one can already see the possibility for erroneous results. Obviously, in order for UDM to implement the functionality for Objects to have relationships with other Objects, the UDM core libraries need some sort of data structure to hold information about this

relationship, and this is currently a limitation for UDM: its core libraries are not thread-safe. Specifically, one cannot use the functionality provided by the UDM core libraries concurrently from multiple threads without the risk of erroneous results (such as the one described in the above scenario).

The need for a set of UDM core libraries that are thread-safe (in the sense described above) grows as more and more cores are included in single CPUs. While specific use cases for a multi-threaded UDM are particular to individual users, experience shows that it would be very helpful to have automatic locking mechanisms built into the included libraries so that the user does not have to put forth any extra effort in their coding. From the users' point of view, the way they use the UDM API should not change if thread-safety is built into the core libraries.

Synchronization Primitives

When programming with multiple threads, shared resources (e.g., data structures that can be accessed from more than one thread) must be "locked" before being accessed to ensure consistency [9]. This means that before a shared resource is either read-from (i.e., only observed and not modified) or written to (i.e., modified), some sort of "locking" object associated with the resource must be informed that the resource is being accessed. There are several ways to approach the problem depending on the desired semantics of execution; for instance, the following are possible execution semantics associated with locking mechanisms.

1. Single-reader, single-writer. This means that either one reader or one writer is allowed to access the resource at a time.

2. Multiple-reader, single-writer. This means that multiple threads can read from the resource at the same time, but only one writer can access and manipulate the resource at time.

3. Single-reader, single-writer (asynchronous). This has the same semantics as single-reader, single-writer listed above, with the difference that the call to read or write from/to the resource always returns immediately, even if the object is currently being accessed by another thread; the desired read or write will happen sometime in the future, and allows the thread which made the call to continue executing without blocking.

4. Multiple-reader, single-writer (asynchronous). This has the same semantics as asynchronous version of the single-reader, single-writer scenario listed above, with the difference that multiple-readers can access the resource at the same time.

A variety of synchronization mechanisms exist to provide mutual exclusion while still providing a reasonable amount of performance. All of these mechanisms aim to eliminate the problem of "busy-waiting". Busy-waiting occurs when a thread that cannot immediately access a shared resource because another thread currently has exclusive access simply "wastes" CPU cycles doing nothing other than waiting for the other thread to relinquish control of the resource.

The C# programming language [7] provides various synchronization primitives that can be used to provide mutual exclusion. These are described presently.

Locks

The "lock" keyword in C# ensures that a section of code is executed to completion by only one thread. The "lock" keyword provides mutual exclusion by obtaining the mutual-exclusion lock for a given object, executing the critical region code, and then releasing the lock. If one thread attempts to enter the protected section of code while another thread is already executing in this region, the calling thread will block until the first thread exits the protected region.

Monitors

Monitors are similar to the "lock" keyword in that they also prevent a section of code from being executed concurrently by more than one thread. One obtains a mutual exclusion lock on an object by passing it to the "Enter" method of the Monitor class, and releases that mutual exclusion lock by passing the object to "Exit" method of the Monitor class. In fact, using the lock keyword to obtain the mutual exclusion lock on an object is precisely equivalent to using the Enter and Exit methods of the Monitor with a "try-finally" block [7]. Generally, however, the lock keyword is preferred, because

it is more concise, and it also ensures that the underlying monitor is released by using the "finally" keyword.

Semaphores

A semaphore is an object used to limit the number of threads that can concurrently access a resource. A semaphore is initialized with an integer value that determines how many threads can access the protected resource concurrently. Then, whenever a thread needs access to the resource protected by a semaphore, it makes a request to the semaphore. If the number of threads currently accessing the resource is less than the amount allowed by the semaphore, then the semaphore's internal count of how many threads are accessing the resource is increased, and the thread is granted access to the resource. When the thread is done accessing the resource, it releases the semaphore, and the semaphore's internal thread access count is decreased. In the case that a thread requests access to a resource whose semaphore count is full (i.e., the maximum number of allowed threads are already accessing the resource), the thread blocks until a later time when the semaphore count is not at its maximum. There is no guaranteed order in which blocked threads are allowed to access the resource after they become un-blocked.

Mutexes

A mutex is very similar to a semaphore, the main difference being that a semaphore can be initialized to allow any number of threads concurrent access to a protected resource, while a mutex allows only one thread to access a protected resource at a time. If a thread tries to access a protected region through a mutex that is already held by another thread, the calling thread is suspended until the first thread releases the mutex.

There are two major differences between the first two concepts listed above and the latter two. The first is that the lock keyword and monitors do not allow one thread to communicate an event to another thread. The second major difference is that only semaphores and mutexes can be used for inter-process synchronization. However, the C# documentation states that for intra-process synchronization, the first two methods are computationally less expensive than the latter two and also make better use of resources.

ReaderWriterLock

The ReaderWriterLock class, provided with C#, is a locking mechanism that implements multiple-reader, single-writer synchronous semantics. Multiple readers alternate with single writers, so that neither reader nor writer threads are blocked for long periods. While this class offers the execution

semantics desired for the thread-safe implementation of UDM, the implementation section (below) describes reasons why this class is not the best choice for a locking mechanism.

CHAPTER III

IMPLEMENTATION

This chapter describes the implementation of the C# code generator along with the thread-safe UDM libraries.

Code Generator Implementation

The code generator, currently implemented in C++, operates in two phases. The first phase generates a C# class definition for every UML Class found in the input file. The second phase generates the C# code to initialize a meta-Datanetwork of this class diagram. For the second phase, there are two cases to consider:

1. The class diagram describes a meta-meta-model (e.g., UML).

2. The class diagram describes a meta-model that conforms to a meta-meta-model (usually UML).

The reason for the first case is so that the programmatic description of the meta-meta-model (UML) used by UDM can be automatically generated. In this case, there is no existing automatically generated code that can be used to instantiate objects of the meta-meta-model, so the generated code must use hard-coded initialization functions to "boot-strap" the meta-meta-model.

In other words, there must be hand-coded functions that are responsible for the creation of the meta-objects contained inside the meta-meta-model.

Phase One - Class Definition Generation

For every Class in the UML Class Diagram, a C# class with the same name is generated. Contained within this C# class is the following information:

1. A single static object of the **Uml Class**. This object serves as the "type" object for this class. A reference to this object is used to describe Udm Objects in instance models of this meta-model.

2. One static object of the Uml Attribute class for each attribute contained by the Class. This static object is used to describe the attribute. Also generated is a C# property [7] with the same name as the attribute that allows one to retrieve the value of the attribute and to set the value of the attribute.

3. One static object of the Uml AssociationRole class for each association contained by the Class. This static object is used to describe one end of a two-way association. Also generated is a C# property with the same name as the association role on the other side of the association. This property allows one to retrieve or set the object(s) associated with this object.

4. One static object of the Uml CompositionChildRole class for each Class that can be contained by the current Class. A corresponding C# property is also generated that allows one to retrieve or set the object(s) that will be the children of the current object.

5. One static object of the Uml CompositionParentRole class for each different Class that can contain this Class as a child (i.e., in a composition relationship). The corresponding C# property allows one to retrieve or set the parent of this object.

Phase Two - Initialization Code Generation

The purpose of the initialization code is to initialize the values of all of the static objects defined in the C# class definitions. It was described above that each generated class has some number of static objects that are all instances of classes defined in the meta-meta-model. These static objects must be initialized in a way that reflects the structure of the UML Class Diagram. For example, suppose we have a class named "X" in our class diagram. When the corresponding C# class definition is generated, it will contain one static object of the Uml Class class that is used to describe objects in an instance model whose meta-type is "X." This static object must be initialized and its attributes set to correspond its description in its UML Class Diagram.

Thread-Safe Libraries

The following methods, which are part of the implementation object definition (i.e., the ObjectImpl class and its derived classes), had to be modified in order to protect the class's internal data structures from corruption when being accessed concurrently from multiple threads:

1. Setting and retrieving an object's parent
2. Creating and retrieving children
3. Creating associations between objects
4. Modifying and retrieving attribute values

The implementation of the thread-safe UDM base libraries involved the addition of a locking mechanism to the object implementation class (i.e., the ObjectImpl class). This locking structure is described below.

The Locking Mechanism

Because the core methods invoked on the implementation class objects are clearly separated into reads or writes, the locking semantics that we desire are multiple-reader, single-writer in a synchronous manner. This means that we want multiple threads to be able to read from a single implementation object concurrently, and we want at most one thread to be able to write to an object at one time. Synchronous means that if a thread attempts to read

or write an implementation object and cannot immediately complete because another thread is currently invoking the opposite action on the object, then the thread blocks and does not return until it is able to complete its read or write request.

As was mentioned in the background section, the C# programming language includes a ReaderWriterLock class that supports single writers and multiple readers. However, as described in [19], this class has several drawbacks associated with it. The biggest drawback is its poor performance. For instance, according to [19], even when there is no contention for a ReaderWriterLock, a method call to acquire the lock takes approximately five times longer to complete than a call to acquire a lock on a Monitor object. Additionally, the ReaderWriterLock class supports recursion. This means that it remembers which thread currently owns the lock, and if the thread that owns the lock attempts to acquire it again, the class increments a count on the lock and allows the thread to acquire the lock again. This requires additional memory to keep track of this count, and also requires additional time to update the counter, both of which contribute to its poor performance. The C# implementation of UDM does not require a locking mechanism that supports recursion, so we do not need this in our lock.

Described in [19] is the design of a very fast lock with multiple-reader, single-writer semantics known as the "OneManyResourceLock". The OneManyResourceLock class offers better performance than the ReaderWriterLock class because it does not support recursion and also minimizes transitions to kernel-mode [22]. Put simply, user-level programs, such as those that make use of UDM, execute primarily in user-mode. When a user-level program needs to manipulate a data-structure that is owned by the kernel (which is the "core" part of an operating system), it must transition to kernel-mode. These transitions are computationally expensive and can hurt performance. For instance, invoking a method to have a thread sleep (which means we are requesting that the thread simply not run for a certain amount of time) requires a transition to kernel mode. [18] gives a small comparison of the performance of operations which must transition to kernel mode against other operations that do not transition to kernel mode, including empirical results that indicate there are thread-safe methods that do not transition to kernel model that can be used to build locking mechanisms that are 10x-100x faster than mechanisms that must transition to kernel mode. For these reasons, the C# implementation of UDM uses the OneManyResourceLock as its underlying locking mechanism.

The C# UDM implementation was structured so that the time that an implementation object is locked is kept to a minimum. The next section addresses the issue of deadlocks.

Ensuring Deadlock Freedom

In general, proving that a multi-threaded program is free from deadlocks is very difficult. However, in the case of the UDM core libraries, the problem is simplified, and it can be proven that the libraries are free from deadlocks. A deadlock can occur in only one way: an implementation object (call it "X"), while holding its own lock, attempts to obtain a lock on a different implementation object (call it "Y") in an arbitrary manner. If we allow this to happen, then the following sequence of actions can occur:

1. Thread 1 obtains a lock on X.
2. Thread 2 obtains a lock on Y.
3. Thread 1, which still has a lock on X, tries to obtain a lock on Y.
4. Thread 2, which still has a lock on Y, tries to obtain a lock on X.

which results in a deadlock. The core libraries were implemented so that the following two assumptions hold:

1. The only method that holds locks on two different implementation objects simultaneously is the method to set a bi-directional association between two objects.
2. Every implementation object has a unique, immutable integer ID associated with it.

With these two assumptions, the task of ensuring deadlock freedom becomes much simpler: in the method to set a bi-directional association, simply compare the unique IDs of the two objects that must be locked before attempting to obtain the lock on either one. Obtain the lock on the object with the smaller ID first, obtain the other lock second, and release the locks in the reverse order. This technique of ensuring deadlock freedom is known as, "resource ordering" [9], and is the method used to prevent deadlocks in this implementation of UDM. A proof by contradiction (below) shows that this does indeed provide freedom from deadlocks.

Proof of Deadlock Freedom

Assume that there is a deadlock. By the assumptions given in the preceding section, the situation causing the deadlock is one in which two implementation objects, call them "X" and "Y", each in a different thread, both attempt to obtain a lock on the other object while holding a lock on themselves. Further, the assumptions guarantee this code must be in the method to set an association between two objects (as this is the only code in which two different locks are held by the same thread time concurrently). However, both X and Y have a unique ID, and one of these IDs must be smaller than the other:

- If the ID of X is smaller, then the thread in which Y is attempting to obtain a lock on X should have first obtained the lock on X.

- If the ID of Y is smaller, then the thread in which X is attempting to obtain a lock on Y should have first obtained the lock on Y.

With either case, we have a contradiction to the guaranteed assumption that the lock on the object with the smaller unique ID is obtained first. Thus, our initial assumption that there is a deadlock is incorrect, and we do not have a deadlock. Q.E.D.

CHAPTER IV

RESULTS

The end of the last section presented a proof that the resource-ordering method of obtaining locks guarantees freedom from deadlocks. This section provides empirical results from tests that use the generated code as the underlying data structure for a recursive algorithm that lends itself well to parallelism. This provides motivation for having a thread-safe version of the UDM core libraries and also gives a comparison showing the performance gain that can be given using multiple threads.

The Algorithm Used for Testing

The performance of the multi-threaded implementation of UDM was tested using a recursive algorithm to generate a fractal pattern known as the, "Sierpiński triangle" [21]. The fractal is generated as follows:

1. Start with a single triangle (for example, the leftmost image of IV.1).
2. Shrink the triangle to 1/2 its height and 1/2 its width, make two copies, and position the three shrunken triangles so that each touches the other two triangles at a corner (the second image of IV.1).
3. Repeat the second step with each of the smaller triangles.

The figure below shows the progression of the algorithm starting with a basic equilateral triangle.

Figure IV.1: Evolution of the Sierpiński Triangle

UDM was used as the underlying data structure holding information about the triangles. To do this, first a meta-model capable of describing an instance of a Sierpiński triangle was created. This was a very simple meta-model, consisting of three elements:

1. Nodes to represent the endpoints of a triangle.

2. Connections between nodes to represent the sides of a triangle.

3. A container object to hold the nodes and connections.

The newly designed UDM Code Generator was then used to generate a C# programmatic description of this Class Diagram. Finally, a recursive version of the Sierpiński algorithm using the data-structures defined in the generated C# code was implemented. The implementation of the algorithm allowed the user to specify how many different threads should be used to generate the fractal pattern. When multiple threads were used, both nodes and the container object used to hold nodes and connections were shared between different threads. Both the meta-model and the C# code used to implement the algorithm are given in the appendix.

The algorithm was parallelized in the following way. We start with a single triangle, and after performing the second step of the algorithm (see above) for the first time, the resulting computations on the three resulting triangles are performed in different threads. For instance, if the user desires to run the transformation with three threads, then each resulting triangle from the first step is passed to a different thread, and this thread begins to run the algorithm on only that triangle. For the case of four threads, the fourth thread is not created until after the second iteration of the algorithm, at which time one of the triangles produced during that iteration is passed to the fourth thread and the algorithm continues as before. Thus, in order to distribute the load between the threads evenly, the number of threads needs to be a power of 3. This is the primary reason that using four threads instead of three did not significantly decrease the running time of the algorithm, even on a quad core processor.

Explanation of Results

The three graphs below show the running time using various numbers of threads with different type of processors. The first graph shows the running times with a single core processor, the second graph shows the running times with a dual core processor, and the third shows the running times with a quad core processor.

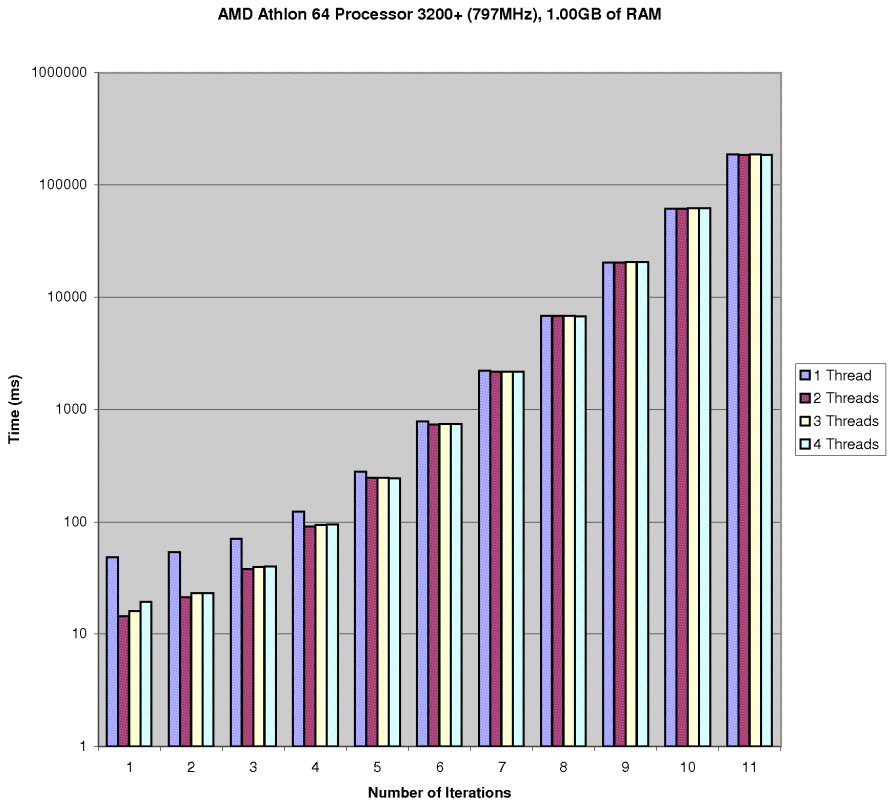

Figure IV.2: Results with Single Core Processor

Figure IV.2 seems to indicate that even for a single core processor, using multiple threads decreases the running time with a small number of iterations. This goes against what intuition would lead one to believe: that on a single core machine, using more than one thread will decrease the performance of such a program, because all code is executed on a single core, and adding multiple threads simply introduces overhead, such as context switches between the threads. However, there is an explanation for this behavior. Each additional thread that a program creates also means that the

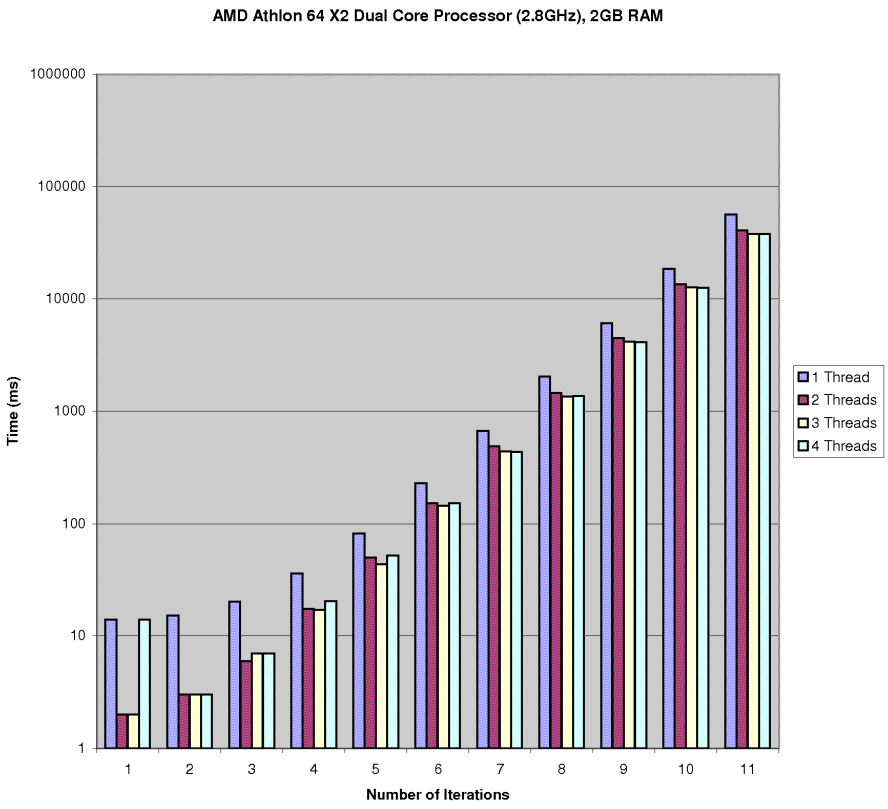

Figure IV.3: Results with Dual Core Processor

underlying operating system gives more scheduling time to the program. For executions of the program with fewer than 8 iterations, the additional time that the operating system allocates to the program outweighs the overhead associated with switching between multiple threads. However, for 8 or more iterations, the overhead of switching between multiple threads outweighs the additional time allocated to this process. For example, with 8 or more iterations, using 2 threads as opposed to 1 thread offers an average speedup in total running time of only 0.5%, while using 3 or 4 threads increases the

average running time. The full results for the average percent speedup in running time using multiple threads are shown in Figure IV.5.

Figure IV.3 displays the running time of the algorithm using a dual core processor and various numbers of threads, and shows that using multiple threads increases performance regardless of the number of iterations. This is in agreement with what one would expect: because there is more than one physical core, if the work can be divided between the cores, then the total running time should decrease. The average speedup in running time using 2 threads vs. 1 thread is approximately 45%, the average speedup using 3 threads vs. 1 thread is approximately 48%, and the average speedup using 4 threads vs. 1 thread is approximately 39%. With 8 or more iterations, the average speedup in running time using multiple threads was 31%, as opposed to 0.5% for using a single core processor. Figure IV.6 shows the full results for average percent speedup using a dual core processor.

Figure IV.4 displays the running time of the algorithm using a quad core processor. The results in this case are similar to those of the dual core processor. However, with a four core processor, there is an even greater relative speedup in average running time than with dual core processor, as shown in Figure IV.7. With 8 or more iterations, using 3 or 4 threads offers an average decrease in running time of over 50% verses using 1 thread (as opposed to a 31% decrease with a dual core processor and 0.5% using a single core processor).

Using Amdahl's Law [11], which is a model for the relationship between expected speedup of parallelized implementations of an algorithm relative to a serial version of the algorithm, we find that our empirical results are close to the ideal speedup predicted by Amdahl's Law. If P is the proportion of a program that can be made parallel, and *1-P* is the proportion that cannot be parallelized, then according to Amdahl's Law, the maximum speedup that can be achieved by using N processors is:

$$\frac{1}{(1-P)+\frac{P}{N}}$$

For the case of a dual core processor (which essentially acts as two processors) and using the parallelized implementation described above, we have parameters *P=1/2* (the amount of work done by one of the threads is 1/2 of the amount done by the other thread) and *N=2*, which gives an ideal speedup of:

$$\frac{1}{(1-\frac{1}{2})+\frac{\frac{1}{2}}{2}} = \frac{4}{3} \approx 1.33$$

Referring to Figure IV.6, we find that our actual speedup for more than seven iterations was an average of approximately 27%, which is very close to the ideal 33% speedup predicted by Amdahl's Law. The difference can be attributed to the overhead of switching between multiple threads and also

the shared underlying data structures to which access had to be synchronized by the locking mechanism described in Chapter III.

For the case of a quad core processor and using three threads with our parallelized implementation, the ideal speedup should be 66%, which is easy to see intuitively because each of the three triangles generated by the first iteration of the algorithm are subsequently processed by different threads on different cores, meaning that each thread is now doing 1/3 of the work that had to be done by 1 thread in the single thread case. Figure IV.7 shows that for five or more iterations, the average speedup offered by our implementation was approximately 50%. This is lower than the ideal speedup, but the difference between the actual speedup and ideal speedup can be attributed to the same factors as in the dual core case (e.g., context switches between threads). Also, when using three threads, there are on the average more concurrent requests to the underlying data structures, so the overhead imposed by locking mechanism affects the speedup more.

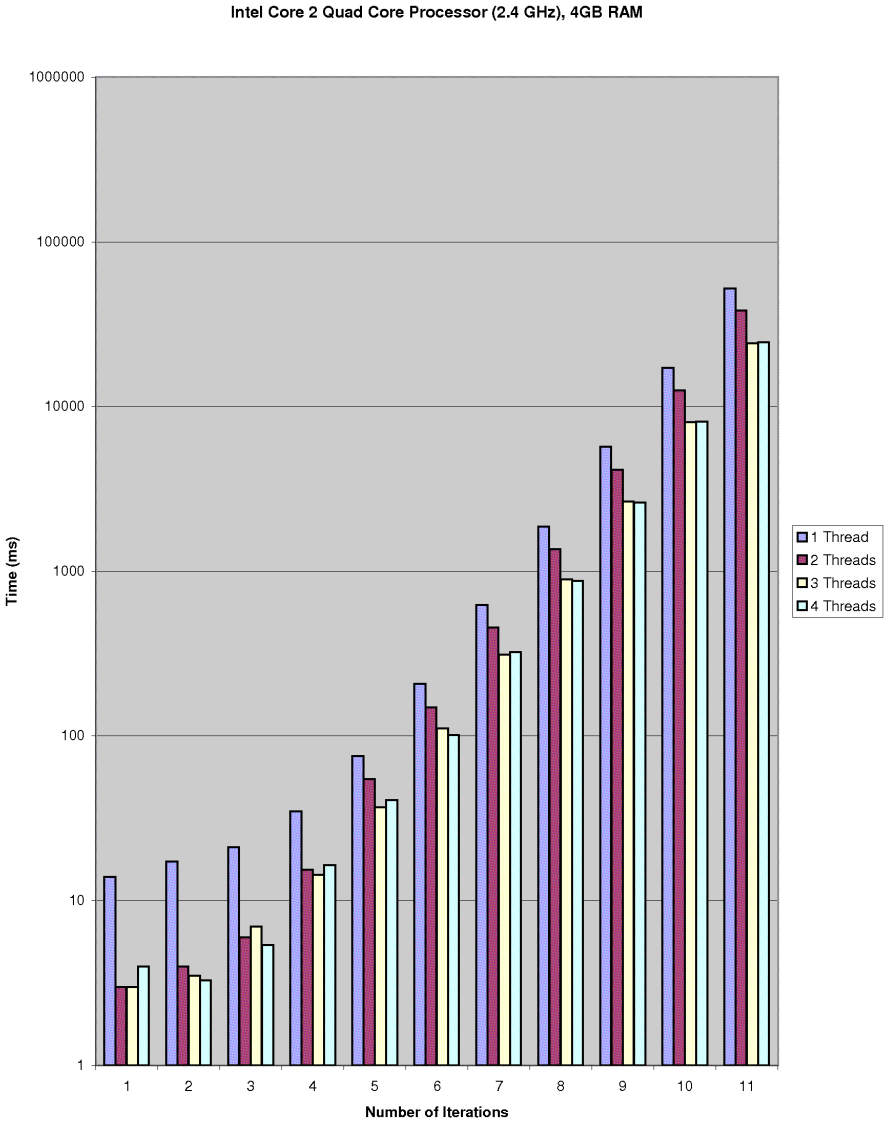

Figure IV.4: Results with Quad Core Processor

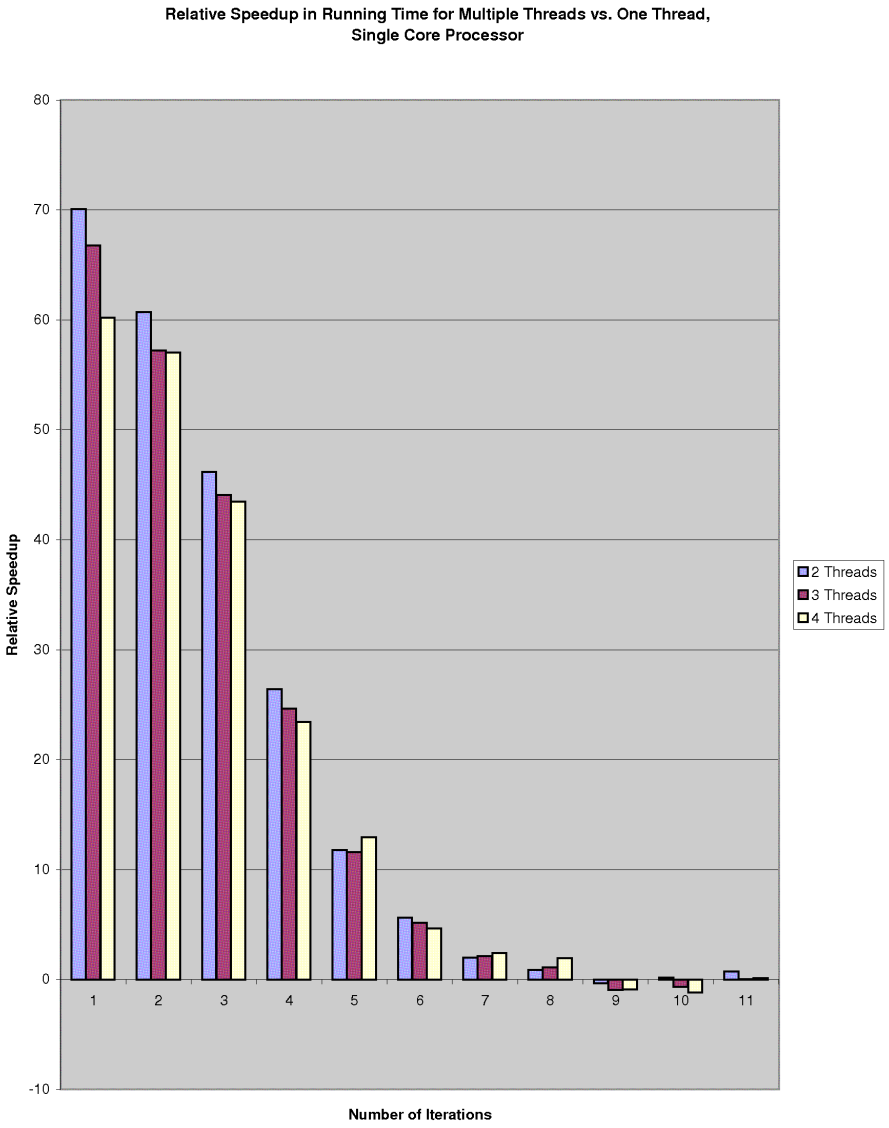

Figure IV.5: Relative Speedup in Running Time Using Multiple Threads on a Single Core Processor

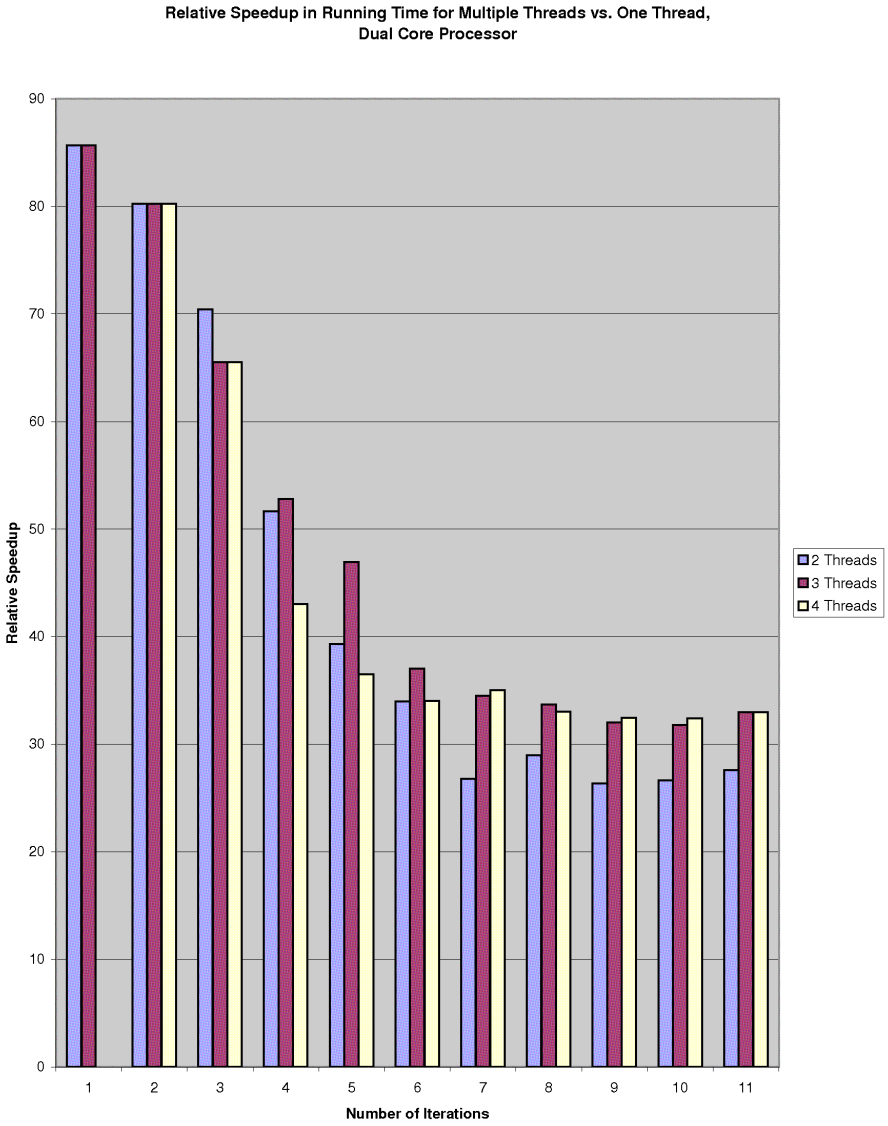

Figure IV.6: Relative Speedup in Running Time Using Multiple Threads on a Dual Core Processor

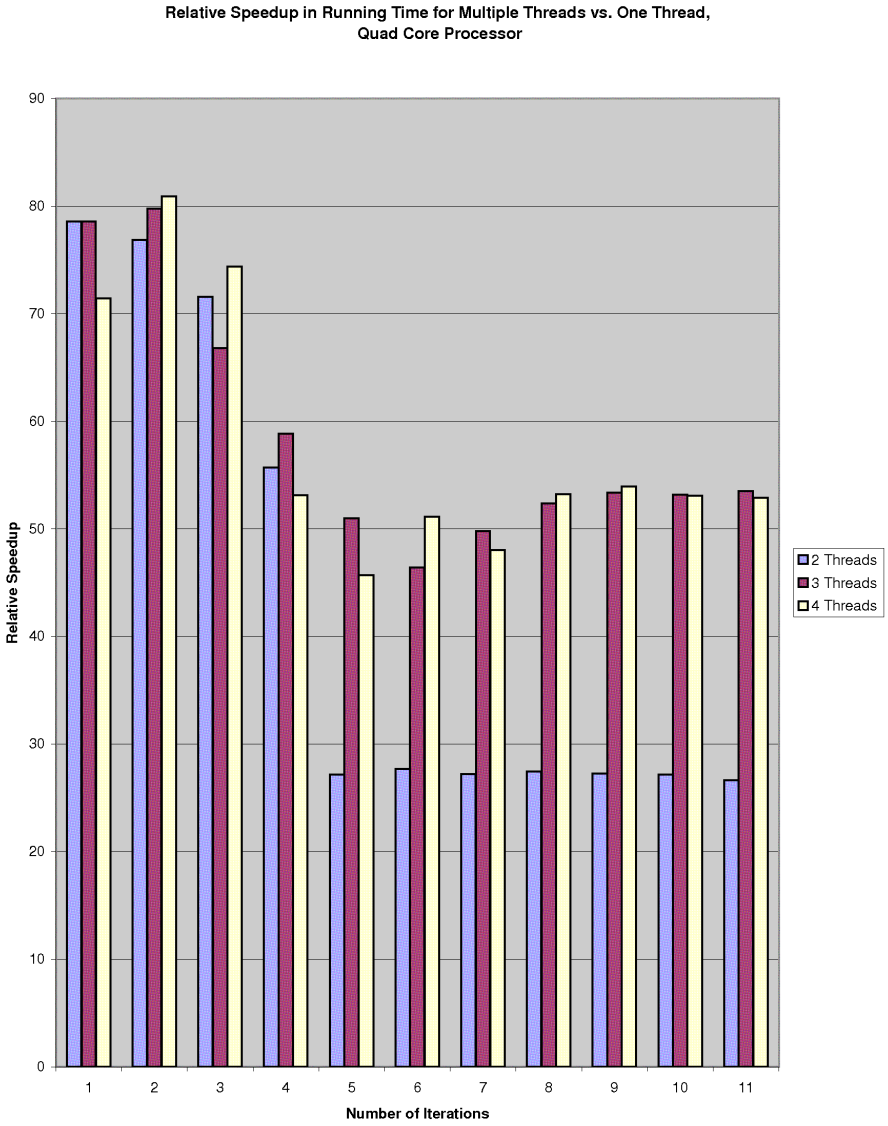

Figure IV.7: Relative Speedup in Running Time Using Multiple Threads on a Quad Core Processor

CHAPTER V

CONCLUSIONS

This thesis has described the implementation of a thread-safe, meta-programmable data model that can safely be used in a multi-threaded environment with no synchronization code written by the user. The locking mechanism provides multiple-reader, single-writer semantics, so that multiple threads can read from an object concurrently, while at most one thread can write to an object at a time. A proof that the program is deadlock-free was also provided.

The data model was tested in a program in which it provided the underlying data structures to a fractal generating algorithm. This sequential fractal generating algorithm was then parallelized and the speedup using various numbers of threads was measured on processors with one, two, and four cores. The measured speedup offered by using multiple threads on processors with multiple cores was less than the ideal speedup predicted by Amdahl's Law, but was still large enough to justify the implementation of thread-safety to the data model.

Future Work

This work can be extended in a number of ways. The implementation object can be extended to provide different support for different types of persistent storage. For instance, direct support could be provided for XML and also for the binary GME format, as is provided in the non-thread-safe C++ version of UDM.

The non-thread-safe C++ version of UDM is used as the underlying data model for the graph transformation tool-suite GReAT [8]. A new version of the GReAT code generator [4] that targets C# UDM could be implemented, as preliminary tests indicate that even without multi-threading, the performance of the new version of UDM is comparable to the non-thread-safe C++ version. Also, the performance of GReAT could be increased by using this newly implemented thread-safe version of UDM if there were a way of automatically or semi-automatically generating parallel graph transformations that use multiple threads to execute. This would require an analysis of the transformation and then a re-implementation of the GReAT code generator to generate C# UDM code that runs in multiple threads. The underlying GReAT language might also need to be extended to include support for user provided "hints" as to which pieces of a GReAT transformation can be run in parallel.

APPENDIX A

SIERPIŃSKI ALGORITHM

Figure A.1 below shows the meta-model used in the Sierpiński Triangle algorithm. There are three elements:

1. Node: represents a vertex of a triangle or triangles.
2. Connection: represents an edge between two vertices of a triangle.
3. SierpinskiTriangleModel: a single container used to hold Nodes and Connections.

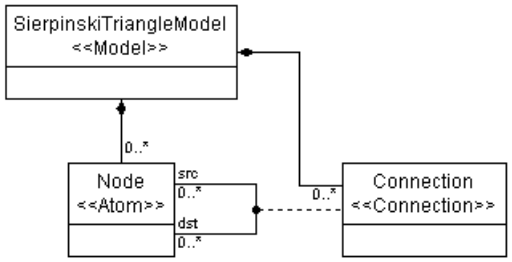

Figure A.1: UML Meta-model Used in the Sierpiński Triangle Algorithm

The following code was used to implement both the sequential and parallel versions of the Sierpiński Triangle algorithm.

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Threading;
using System.Diagnostics;

using TriangleModel;

namespace TriangleTest
{
    /* A simple struct to hold information passed to spawned threads */
    struct Holder
    {
        public Connection c1, c2, c3;
        public int numGen;
        public int eventNum;
        public bool doMore;
    }

    class Program
    {
        private UdmStatic.StaticDatanetwork sdn;
        private int numThreads, numGenerations;
        private AutoResetEvent[] myEvents;

        public Program()
        {
            numThreads = 0; //use no extra threads for the default case
            numGenerations = 5;  //5 iterations by default
        }

        public Program(int x)
        {
            numThreads = x;
            myEvents = new AutoResetEvent[x];
            numGenerations = 5;

            for (int i = 0; i < x; i++) {
                myEvents[i] = new AutoResetEvent(false);
            }
        }

        public Program(int x, int y)
        {
            numThreads = x;
            myEvents = new AutoResetEvent[x];
            numGenerations = y;

            for (int i = 0; i < x; i++) {
                myEvents[i] = new AutoResetEvent(false);
            }
        }

        public void WorkerThread(object data)
```

```
{
    Holder h = (Holder)data;

    if (h.doMore == false) {
        TransformTriangle(h.c1, h.c2, h.c3, h.numGen);
        myEvents[h.eventNum].Set();
    }

    else {

        Holder h3 = new Holder();
        h3.doMore = false;
        h3.numGen = h.numGen - 1;
        h3.eventNum = 2;

        {
            Node n1 = h.c1.srcConnection_end;
            Node n2 = h.c2.srcConnection_end;
            Node n3 = h.c3.srcConnection_end;

            string[] newPositions = GetPositions(n1, n2, n3);

            SierpinskiTriangleModel myParent =
                n1.SierpinskiTriangleModel_parent;

            //Create the new nodes
            Node NewNodeA = Node.Create(myParent, null);
            Node NewNodeB = Node.Create(myParent, null);
            Node NewNodeC = Node.Create(myParent, null);

            NewNodeA.position = newPositions[0];
            NewNodeB.position = newPositions[1];
            NewNodeC.position = newPositions[2];

            //Create 6 new connections
            Connection[] myConns = new Connection[6];
            for (int i = 0; i < 6; ++i)
            {
                myConns[i] = Connection.Create(myParent, null);
            }

            myConns[0].srcConnection_end = NewNodeA;
            myConns[0].dstConnection_end = n2;
            myConns[1].srcConnection_end = n2;
            myConns[1].dstConnection_end = NewNodeB;
            myConns[2].srcConnection_end = NewNodeB;
            myConns[2].dstConnection_end = NewNodeA;
            myConns[3].srcConnection_end = NewNodeA;
            myConns[3].dstConnection_end = NewNodeC;
            myConns[4].srcConnection_end = NewNodeC;
            myConns[4].dstConnection_end = NewNodeB;
            myConns[5].srcConnection_end = n3;
            myConns[5].dstConnection_end = NewNodeC;
```

```
            h.c1.dstConnection_end = NewNodeA;
            h.c2.srcConnection_end = NewNodeB;
            h.c3.srcConnection_end = NewNodeC;

            //insert the links in the new holder
            h3.c1 = h.c1;
            h3.c2 = myConns[3];
            h3.c3 = h.c3;

            Thread lastT = new Thread(
                new ParameterizedThreadStart(this.WorkerThread));
            lastT.Start((object)h3);

            TransformTriangle(myConns[0], myConns[1],
                myConns[2], h.numGen-1);
            TransformTriangle(myConns[4], h.c2,
                myConns[5], h.numGen-1);
        }

        myEvents[h.eventNum].Set();
    }
}

public void DoMultipleWork(int localNumGen)
{
    RootFolder rf = RootFolder.Cast(sdn.GetRootObject());
    SierpinskiTriangleModel stm =
        SierpinskiTriangleModel.Create(rf, null);

    Node n1 = Node.Create(stm, null);
    Node n2 = Node.Create(stm, null);
    Node n3 = Node.Create(stm, null);

    Connection conn1 = Connection.Create(stm, null);
    Connection conn2 = Connection.Create(stm, null);
    Connection conn3 = Connection.Create(stm, null);

    n1.position = "(502,124)";
    n2.position = "(845,733)";
    n3.position = "(166,733)";

    conn1.srcConnection_end = n1;
    conn1.dstConnection_end = n2;
    conn2.srcConnection_end = n2;
    conn2.dstConnection_end = n3;
    conn3.srcConnection_end = n3;
    conn3.dstConnection_end = n1;

    //go ahead and do one iteration so that we can
    //spawn multiple threads easier
    if (localNumGen > 0)
    {
        --localNumGen;
```

```
string[] newPositions = GetPositions(n1, n2, n3);

SierpinskiTriangleModel myParent = stm;

//Create the new nodes
Node NewNodeA = Node.Create(myParent, null);
Node NewNodeB = Node.Create(myParent, null);
Node NewNodeC = Node.Create(myParent, null);

NewNodeA.position = newPositions[0];
NewNodeB.position = newPositions[1];
NewNodeC.position = newPositions[2];

//Create 6 new connections
Connection[] myConns = new Connection[6];
for (int i = 0; i < 6; ++i)
{
    myConns[i] = Connection.Create(myParent, null);
}

myConns[0].srcConnection_end = NewNodeA;
myConns[0].dstConnection_end = n2;
myConns[1].srcConnection_end = n2;
myConns[1].dstConnection_end = NewNodeB;
myConns[2].srcConnection_end = NewNodeB;
myConns[2].dstConnection_end = NewNodeA;
myConns[3].srcConnection_end = NewNodeA;
myConns[3].dstConnection_end = NewNodeC;
myConns[4].srcConnection_end = NewNodeC;
myConns[4].dstConnection_end = NewNodeB;
myConns[5].srcConnection_end = n3;
myConns[5].dstConnection_end = NewNodeC;

conn1.dstConnection_end = NewNodeA;
conn2.srcConnection_end = NewNodeB;
conn3.srcConnection_end = NewNodeC;

//Tell the new threads which triangles to work with

Holder h1 = new Holder();
h1.c1 = conn1;
h1.c2 = myConns[3];
h1.c3 = conn3;
h1.doMore = false;
h1.numGen = localNumGen;
h1.eventNum = 0;

Holder h2 = new Holder();
h2.c1 = myConns[0];
h2.c2 = myConns[1];
h2.c3 = myConns[2];
h2.doMore = false;
h2.numGen = localNumGen;
h2.eventNum = 1;
```

```
Holder h3 = new Holder();
h3.c1 = myConns[4];
h3.c2 = conn2;
h3.c3 = myConns[5];
h3.doMore = false;
h3.numGen = localNumGen;
h3.eventNum = 2;

Stopwatch allCounter = Stopwatch.StartNew();

if (numThreads == 1)
{
    Thread t1 = new Thread(
        new ParameterizedThreadStart(this.WorkerThread));
    t1.Priority = ThreadPriority.Normal;
    t1.Start(h1);
    TransformTriangle(myConns[0], myConns[1],
        myConns[2], localNumGen);
    TransformTriangle(myConns[4], conn2,
        myConns[5], localNumGen);
    WaitHandle.WaitAll(myEvents);
}

else if (numThreads == 2)
{
    Thread t1 = new Thread(
        new ParameterizedThreadStart(this.WorkerThread));
    Thread t2 = new Thread(
        new ParameterizedThreadStart(this.WorkerThread));

    Thread mainThread = Thread.CurrentThread;
    mainThread.Priority = ThreadPriority.Lowest;

    t1.Start(h1);
    t2.Start(h2);
    TransformTriangle(myConns[4], conn2,
        myConns[5], localNumGen);

    WaitHandle.WaitAll(myEvents);
}

else if (numThreads == 3)
{
    Thread t1 = new Thread(
        new ParameterizedThreadStart(this.WorkerThread));
    Thread t2 = new Thread(
        new ParameterizedThreadStart(this.WorkerThread));

    Thread mainThread = Thread.CurrentThread;
    mainThread.Priority = ThreadPriority.Lowest;

    h1.doMore = true; //spawn one additional thread later
    t1.Start(h1);
```

```
            t2.Start(h2);
            TransformTriangle(myConns[4], conn2,
                myConns[5], localNumGen);

            WaitHandle.WaitAll(myEvents);
        }

        allCounter.Stop();
        Console.WriteLine("Total time: " +
            allCounter.ElapsedMilliseconds + " ms.");
    }
}

public void StartDoMultipleWork()
{
    sdn = new UdmStatic.StaticDatanetwork(
        TriangleModel.Init.umldiagram);
    sdn.CreateNew("T1.mem", "", ref TriangleModel.RootFolder.meta,
        Udm.BackendSemantics.CHANGES_LOST_DEFAULT);

    if (numThreads == 0) {
        Stopwatch sw = Stopwatch.StartNew();
        BeginTransformation(numGenerations);
        sw.Stop();
        Console.WriteLine("Total time : " +
            sw.ElapsedMilliseconds + " ms.");
    }

    else {
        DoMultipleWork(numGenerations);
    }
    sdn.SaveAs("CSharpTriangleOutput.mem");
}

//the method to call if we only have one thread
public void BeginTransformation(int localNumGen)
{
    RootFolder rf = RootFolder.Cast(sdn.GetRootObject());
    SierpinskiTriangleModel stm =
        SierpinskiTriangleModel.Create(rf, null);

    Node n1 = Node.Create(stm, null);
    Node n2 = Node.Create(stm, null);
    Node n3 = Node.Create(stm, null);

    Connection conn1 = Connection.Create(stm, null);
    Connection conn2 = Connection.Create(stm, null);
    Connection conn3 = Connection.Create(stm, null);

    n1.position = "(502,124)";
    n2.position = "(845,733)";
    n3.position = "(166,733)";

    //connect the vertices of the initial triangle
```

```
    conn1.srcConnection_end = n1;
    conn1.dstConnection_end = n2;
    conn2.srcConnection_end = n2;
    conn2.dstConnection_end = n3;
    conn3.srcConnection_end = n3;
    conn3.dstConnection_end = n1;

    TransformTriangle(conn1, conn2, conn3, localNumGen);
}

public string[] GetPositions(Node n1, Node n2, Node n3)
{
    string one = n1.position;
    string two = n2.position;
    string three = n3.position;

    int x1, y1, x2, y2, x3, y3;

    x1 = System.Convert.ToInt32(
        one.Substring(1, one.IndexOf(',') - 1));
    y1 = System.Convert.ToInt32(one.Substring(
        one.IndexOf(',') + 1,
        one.Length - (one.IndexOf(',') + 2)));

    x2 = System.Convert.ToInt32(
        two.Substring(1, two.IndexOf(',') - 1));
    y2 = System.Convert.ToInt32(two.Substring(
        two.IndexOf(',') + 1,
        two.Length - (two.IndexOf(',') + 2)));

    x3 = System.Convert.ToInt32(
        three.Substring(1, three.IndexOf(',') - 1));
    y3 = System.Convert.ToInt32(three.Substring(
        three.IndexOf(',') + 1,
        three.Length - (three.IndexOf(',') + 2)));

    int NNAX = (x1 + x2) / 2;
    int NNAY = (y1 + y2) / 2;

    int NNBX = (x3 + x2) / 2;
    int NNBY = (y3 + y2) / 2;

    int NNCX = (x1 + x3) / 2;
    int NNCY = (y1 + y3) / 2;

    string pos1 = "(" + NNAX + "," + NNAY + ")";
    string pos2 = "(" + NNBX + "," + NNBY + ")";
    string pos3 = "(" + NNCX + "," + NNCY + ")";

    return new string[] { pos1, pos2, pos3 };
}

public void TransformTriangle(Connection a, Connection b,
    Connection c, int GenNum)
```

```
{
    if (GenNum > 0)
    {
        --GenNum;

        Node n1 = a.srcConnection_end;
        Node n2 = b.srcConnection_end;
        Node n3 = c.srcConnection_end;

        //get the positions for the new nodes
        string[] newPositions = GetPositions(n1, n2, n3);

        SierpinskiTriangleModel myParent =
            n1.SierpinskiTriangleModel_parent;

        //Create the new nodes
        Node NewNodeA = Node.Create(myParent, null);
        Node NewNodeB = Node.Create(myParent, null);
        Node NewNodeC = Node.Create(myParent, null);

        //set the positions of the new nodes
        NewNodeA.position = newPositions[0];
        NewNodeB.position = newPositions[1];
        NewNodeC.position = newPositions[2];

        //Create 6 new connections
        Connection[] myConns = new Connection[6];
        for (int i = 0; i < 6; ++i) {
            myConns[i] = Connection.Create(myParent, null);
        }

        //set the endpoints of the new connections
        myConns[0].srcConnection_end = NewNodeA;
        myConns[0].dstConnection_end = n2;
        myConns[1].srcConnection_end = n2;
        myConns[1].dstConnection_end = NewNodeB;
        myConns[2].srcConnection_end = NewNodeB;
        myConns[2].dstConnection_end = NewNodeA;
        myConns[3].srcConnection_end = NewNodeA;
        myConns[3].dstConnection_end = NewNodeC;
        myConns[4].srcConnection_end = NewNodeC;
        myConns[4].dstConnection_end = NewNodeB;
        myConns[5].srcConnection_end = n3;
        myConns[5].dstConnection_end = NewNodeC;

        a.dstConnection_end = NewNodeA;
        b.srcConnection_end = NewNodeB;
        c.srcConnection_end = NewNodeC;

        //recursively generate more triangles from
        //each of the new three triangles
        TransformTriangle(a, myConns[3], c, GenNum);
        TransformTriangle(myConns[0], myConns[1],
            myConns[2], GenNum);
```

```
            TransformTriangle(myConns[4], b, myConns[5], GenNum);
        }
    }

    static void Main(string[] args)
    {
        if (args.Length == 0)  //default case
        {
            Program p1 = new Program();
            p1.StartDoMultipleWork();
        }

        //if the user specified an additional number of threads
        else if(args.Length == 1)
        {
            int numThreads = Convert.ToInt32(args[0]);
            if ((numThreads >= 0) && (numThreads < 4))
            {
                Program p1 = new Program(numThreads);
                p1.StartDoMultipleWork();
            }

            else
            {
                PrintUsage();
                Environment.Exit(-1);
            }
        }

        else if (args.Length == 2)
        {
            int numThreads = Convert.ToInt32(args[0]);
            int numGenerations = Convert.ToInt32(args[1]);
            if ((numThreads >= 0) || (numThreads < 4))
            {
                Program p1 = new Program(numThreads, numGenerations);
                p1.StartDoMultipleWork();
            }

            else
            {
                PrintUsage();
                Environment.Exit(-1);
            }
        }

        else
        {
            PrintUsage();
            Environment.Exit(-1);
        }
    }

    static void PrintUsage()
```

```
        {
            System.Console.WriteLine("Usage: TransformTriangle.exe " +
            "[numThreads] where numThreads = 1, 2, or 3");
        }
    }
}
```

BIBLIOGRAPHY

[1] Bakay A. and Magyari E. *The UDM Framework*, November 2005. Available with the UDM Framework, http://repo.isis.vandebilt.edu.

[2] Ledeczi A., Maroti M., Bakay A., Karsai G., Garrett J., Thomason IV C., Nordstrom G., Sprinkle J., and Volgyesi P. The generic modeling environment. May 2001.

[3] Vizhanyo A., Agrawal A., and Shi F. Towards generation of high-performance transformations. In *Generative Programming and Component Engineering, Vancouver, Canada*, 2004.

[4] Vizhanyo A., Agrawal A., and Shi F. Towards generation of high-performance transformations. October 2004.

[5] Ákos Lédeczi, Árpád Bakay, Miklós Maróti, Péter Völgyesi, Greg Nordstrom, Jonathan Sprinkle, and Gábor Karsai. Composing domain-specific design environments. *Computer*, 34(11):44–51, 2001.

[6] Frank Budinsky, Stephen A. Brodsky, and Ed Merks. *Eclipse Modeling Framework*. Pearson Education, 2003.

[7] Microsoft Corporation. *Microsoft C# Language Specifications*. Microsoft Press, Redmond, WA, USA, 2001.

[8] Balasubramanian D., Narayanan A., Buskirk C., and Karsai G. The graph rewriting and transformation language: Great. September 2006.

[9] Lea D. *Concurrent Programming in Java, Second Edition*. Addison-Wesley, 2000.

[10] Magyari E., Bakay A., Lang A., Paka T., Vizhanyo A., Agrawal A., and Karsai G. Udm: An infrastructure for implementing domain-specific modeling languages. In *The 3rd OOPSLA Workshop on Domain-Specific Modeling*, October 2003.

[11] Amdahl G. Validity of the single processor approach to achieving large-scale computing capabilities. September 1967.

[12] Coplien J. *Advanced C++ Programming Styles and Idioms*. Addison-Wesley Publishing Company, 1992.

[13] Sztipanovits J. and Karsai G. Model-integrated computing. *Computer*, pages 110–112, April 1997.

[14] Emerson M., Sztipanovits J., and Bapty T. A mof-based metamodeling environment. *Journal of Universal Computer Science*.

[15] Object Management Group (OMG). *Unified Modeling Language Specification, Version 1.4.2*, January 2005.

[16] Object Management Group (OMG). *Meta Object Facility (MOF) Core Specification, Version 2.0*, January 2006.

[17] Object Management Group (OMG). *MOF 2.0/XMI Mapping, Version 2.1.1*, December 2007.

[18] Jeffrey Richter. Performance-conscious thread synchronization. *MSDN Magazine*, 20(10), October 2005.

[19] Jeffrey Richter. Reader/writer locks and the resourcelock library. *MSDN Magazine*, 21(7), June 2006.

[20] Neema S., Kalmar Z., Feng S., Vizhanyo A., and Karsai G. A visually-specified code generator for simulink/stateflow. In *IEEE Symposium on Visual Languages and Human-Centric Computing*, 2005.

[21] Waclaw Sierpinski. Sur une courbe dont tout point est un point de ramification. *C. R. Acad. Sci. Paris.*

[22] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts, 6th Edition.* Wiley, 2001.

[23] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.