1. INTRODUCTION
Since the arrival of general-purpose computers, there has been a constant trend in making such machines programmable as easily and cost-effectively as possible. Initially raw binary (hexa-decimal) represented machine code was used to program the computers. However, the hardware-specific details increased the complexity of the programs significantly. The first remedy to this problem was the introduction of assembly languages that are symbolic abstraction very close for the machine code instructions, but much more readable for human being. The assembly programs are automatically converted to machine codes using programs called assemblers. While it was very effective, the assembly languages did not solve the problem totally, since their level of abstraction was still close to the machine code. That means there is a lot of explicit machine-specific detailed information an assembly programmer needs to know. In addition to increasing the complexity, such details make a program specific to a hardware platform (i.e. instruction set architecture). High-level programming languages (e.g. Algol, COBOL, and Pascal) were introduced to cope with this problem. Such languages provided a level of abstractions close to the mathematical notations for writing algorithms. Also, they provide mechanisms for manipulating abstract data types and modularizing programs. Programs written in these languages are converted to machine code using compilers. Since such programs contain almost no machine-specific signature, they must have the same meaning no matter what machine they are compiled for. It is the responsibility of the compiler to automatically translate such languages that are closer the human being model of thought to the detailed and complex machine code. Decades-long history of building such compilers has proved success for this approach. Using high-level languages, people can describe problems much more complex than what they could do with machine code or assembly programs. Moreover, having compilers of a language on different platform, it is possible in to port a program from one hardware platform to another one almost with no effort.

With easily programmable computers, there has been a constant trend in using them in new applications, resulting in building more and more complex application programs, possibly running on large networks of computers and involving thousands of people. Such applications are divided into numerous communicating components to overcome the increasing complexity and also to satisfy different needs of various people. As a result, merely relying on the mechanisms embedded in high-level languages for managing complexity and abstraction has not been effective.

There have been several approaches to solve this problem. Both formal and empirical software engineering methodologies have been introduced to somewhat standardize the process of software development by using notations and abstractions higher level than programming language. Such methodologies also provide engineers with guidelines and rules that should be followed in order to fulfill software project goals and reach to quality software.

Another approach was to build application programming platforms that consist of a enormous set of useful software components as well as well defined interface to enable reusing these components in custom applications. Sun J2EE [1] and Microsoft .NET [2] are examples of such platforms. In addition to such application development platforms, there are a large number of individual specific-purpose software components that can be reused in building a system. For instance there are numerous types of libraries that implement specific functionality such as mathematical operations, encryption, graphical user interface (GUI), etc. Also there are various types of middleware that can be used for different purposes such as information integration and distributed computing.

A major problem with these approaches is that as the complexity of applications grows, the amount of information programmers and designers must keep in their
mind grows as well. Such information mainly includes the details that are specific to each notation, knowledge about the components with the right functionality and their interface to each other. Moreover, there has to be a lot of tedious manual effort to develop software that follows some repeating patterns. Finally, applications are bound to the platforms from which all of the underlying components used in building them are taken. It defeats the purpose of portability that was one of the main goals of programming languages. This situation is very much similar to that early assembly programmers were in, in the sense that there is too much platform specific information to memorize, there is too much manual work, and there is little portability.

Generative Programming (GP) is a software engineering practice that addresses this problem. The major idea in GP is similar to the one that led people to move from assembly languages to high-level languages. That is raising the level of abstraction a programmer or designer must deal with, and automate all the translations from the high-level abstractions to implementation components. In GP techniques, programmers define a model of the system that just defines the data contents and logic of the applications. Then programs similar to compilers translate such models to the actual programs or machine codes automatically. As of traditional compilers, there can be several model compilers that translate a single model to various platforms, depending on the needs of different users (Figure 1).

Finally, translation of a model to the conventional software development platforms must be possible most of the time. In some possibly rare cases in which such translation is not computable, application programmers have to write the programs in the current programming languages, and link such components with the rest of the system that is automatically generated.

In this report, we provide a overview of techniques, tools, and methodologies that are used in building systems using GP. The focus of the report is more on model-driven GP. The rest of this report is organized as follows. We first provide a closer look at the current GP techniques. Then, describe the desired characteristics of a model for building general-purpose applications. Then, we review Executable UML as an example of such model and analyze its capabilities and weaknesses. Next, we describe the requirements for building model compilers, and analyze the existing tools. Finally, we describe techniques that use GP in building middleware software in particular.

2. GP: A CLOSER LOOK

A widely accepted definition for GP is that it is the automatic selection, customization, and assembly of pre-built templates and components on demand. This process is in contrast with the current practice of manually searching, adopting, and integrating components [3].

There are some skepticism about that GP will not widely be used since it requires skills that are not mainstream, which could be true, in one sense. Actually, most developers prefer not to use GP in their design and implementation since they want control on the development process. Probably, we need to wait for a few years to see whether many vendors move to GP software engineering. However, there are clear incentives in using techniques such as GP. As hardware and network technologies become faster and cheaper at a predictable pace, software, particularly distributed software, is becoming slower, buggier, and more expensive and it is hard to predict how long it takes to build. A key reason for these different trends is that hardware and networks are now heavily built based on components off the shelf (COTS). While, as for software, current practice is to build the custom components for each product specifically. The interface of hardware COTS are usually standard, whereas distributed software components are usually built in a custom manner, or follow standards that are specific to a company.

Applying COTS standards to distributed software is not easy, and many tough R&D issues must be resolved [4]. GP

---

1 Due to our time limits, such an overview is neither complete nor accurate. However, we tried to cover some of the most techniques and technologies that are around.
as an automatic, reliable way of generating components and bring the Software COTS into reality may be a popular strategy for software vendors soon.

In GP the programmer states what he wants in abstract terms (a model) and the generator produces the desired system or component. The generator, which is a sophisticated piece of software, works as a model compiler and produces the desired system or component.

For domain modeling and code generation, three main items are needed. The first one is a modeling language. The second part is the generator that closes the gap between the domain model and the code world. The third part is a component library that facilitates the code generation. Component libraries usually consist of templates. The code generator uses component templates to emit the source code. This approach is language and domain model neutral.

Figure 2 shows a template that can be taken from component library and can be used by the code generator [5].

```
public class Person {
    //for-each-field--/
    private String mField = "";
    public String getField() {
        return mField;
    }
    public void setField( String pField ) {
        mField = pField;
    }
    //end--/
}
```

Figure 2 A class template.

To summarize, the GP process utilizes the above three items to generate the desired code as the result Figure 3. This magic is possible by direct mapping from each modeling element to the generation of a set of intentionally equivalent source code statements. When a library of components is available, the model interpreter can leverage a larger granularity of reuse by generating configurations of the available components [5].

3. MODELING REQUIREMENTS

In this section we describe the requirements of a model that can be used in GP. First, we describe abstract models that are built for a specific purpose. Then, we enumerate a set of criteria that are necessary for a model in order to express general-purpose applications. We should admit that we are not in a position to make any claim about the completeness of such a list of criteria. Instead, we just review the empirical criteria that are widely believed to be necessary by the software engineering community. We should also mention such a list of criteria is highly influenced by the object-oriented technologies literature.

3.1 DOMAIN SPECIFIC MODELS

There are several languages and notations that are used for modeling specific applications. Structured Query Language (SQL) is a fairly high-level and abstract language that has been widely used for expressing database data and query definition. SQL is a declarative language, i.e. it just defines the requirements and constraints on the desired data content. It does not specify the control structures necessary to implement the query. A query compiler translates each query into a set of functions whose execution would result in having the requested data. While very successful for database queries, it cannot be used to express the interactions and events in a complex system.

Another successful example of domain specific modeling languages is extensible markup language (XML) [6]. XML defines the content of the data in terms of its structures and constraints on the values of each data item. A program needs to translate XML into a more computable model in order to manipulate data efficiently.

Also, there are numerous tools and environments that help application programmers to design GUIs visually. Such environments provide programmers with a set of predefined widgets. Each widget can be easily translated to program code. The programmers can compose the application user interface by combining a set of such widgets together. The system generates a template code for the user interface and let the application programmer fills up the custom program event handling logic manually.

All of the models mentioned above have proven to be effective within their domain. However, they do not provide a uniform means to model the whole system. Next, we describe the major criteria for a modeling language to be able to completely model a general-purpose system.

3.2 MODELING REQUIREMENTS

To model a complex general-purpose application a modeling language must be able to model the following items.

3.2.1 Data Objects Content

For expressing data objects content, we first need a type system for defining the types of each data items within each
data object. Moreover, the modeling language must support hierarchical aggregation of sets of data items. Also, the relationships among the various data objects in the system must be explainable by the model. There are two major types of relationships: Specialization/Generalization, and Association (Linkage).

In addition to the declaration of the data items within each object, there has to be a language to define the semantics and constraints on manipulating the data items. For instance, in the object-oriented terminology that means each object must be accessed and modified only by using a set of accessor and mutator that define the constraints on each data item value.

3.2.2 Objects Lifecycle
Each data object in the system is created and destroyed at certain time spots. Moreover, during its lifetime, the data object attribute values might change in response to some certain events happening in the system. A useful way of modeling such patterns of change is to devise a state machine for each data object. The state abstraction usually reflects what actually takes place in the real world.

3.2.3 Systems Objects Interactions
Systems objects do interact each other to implement system’s scenarios. The interactions usually involve exchange of both data and control among several objects in the system. Each object interaction might result in the change of state of communicating objects. For each object there might be a procedure associated with each instance of communication that manipulates object attributes and possibly changes the state of the object.

3.2.4 Actions Semantics
Each procedure in the system comprises a set of actions. Each action carries out some functional computation, data access, and communication to various objects in the system. Actions are similar to program code, except that actions must be abstract, including no platform specific detail. Moreover, action semantics must be formal so that it can be verified independent of the implementation.

3.2.5 Concurrency and Synchronization
In most systems, the components live and act concurrently. That means at any period of time, there might be several active interactions and actions. A model must be able to precisely describe the concurrency and its related timing constraints in the system. Moreover, there must be a way to describe the synchronizations necessary for system to function correctly.

4. UNIFIED MODELING LANGUAGE
4.1 A BRIEF HISTORY
Unified Modeling Language (UML) [7] has been proposed by Object Management Group (OMG) as a merger to several successfully used object-oriented analysis and design methodologies. It contains many notations and diagrams for showing various aspects of a typical information system. Since its arrival, it has widely been used in the development of many large software projects. UML models are perceived to be comprehensible, accurate, and complete both by system designers and developers and also by non-technical people and managers. However, the translation from the UML models to the actual system code has been done manually.

4.2 MODEL-DRIVEN ARCHITECTURE
Model-Driven Architecture (MDA) [8] by OMG has been proposed as a framework for GP, in which UML has a central role as the modeling language. In MDA, system analysts and designers use UML to describe a platform independent model (PIM) of the system. Then, using appropriate translation tools, such PIMs are translated to platform specific models (PSMs) each of which is targeted for a particular architecture, operating system, and software development platform (e.g. CORBA, Web Services, and J2EE).

Although there are a lot of white papers envisioning various aspects of MDA and its potential advantages, there is little in the MDA community literature showing how actually such a nice architecture is to be implemented.

4.3 EXECUTABLE UML
Executable UML [9][10] is an effort to actually implement MDA. It borrows most of the required notations and language from UML. In particular, class diagrams are used for modeling the content of the system data objects. Also, UML state-charts are used to model system objects lifecycles, and for modeling system objects interactions both UML collaboration diagrams and state-charts are used with slight modifications.

However, in order to make UML executable, besides the standard UML diagrams, Executable UML introduces a set of syntax for formally describing class actions. It relies on the precise action semantics for UML [11] adopted as an integral part of UML in late 2001. These action semantics provide for the specification of actions, but they do not define the syntax of an action language. In this section, we briefly review the syntax used in Executable UML for describing action semantics.

There are four different types of syntax are used in Executable UML: object action language [12], selection expressions, link object actions, and a set of normal programming language constructs for describing selection and iteration. Next, we shortly review each of these notations.

4.3.1 Object action language
Object action language provides syntax mainly for describing the creation, deletion, and setting and getting attributes of the individual objects in the system. Here are some instances of such actions.
create object instance newPublisher of Publisher;
newPublisher.name = "Addison-Wesley";
x = newPublisher.name;
delete object instance newPublisher;

4.3.2 Selection expressions
Inspired by SQL, selection expressions are used to describe selection of a single object or a set of objects. Each select operation is applied on a class of objects, and returns a set of object references that match with the constraints specified in the expressions. The expression mentioned below is an example.

select many newBooks from instances of Book
where selected.copyright <= 2002 and
selected.copyright >= 1995

In the above expression selected refers to the current selected object and newBooks is an object reference set that contains the reference to the objects of class Book that satisfy the condition specified by where clause. It may or may not be empty after executing the expression.

4.3.3 Link object actions
Link actions are those that relate objects to each other. Here are some examples of such actions:
relate newBook to newPublisher across R1;
relate newBook to myPublisher across R2.
   'produces and markets';
select one bookPublisher related by
   newBook->Publisher[R1];
select many allPublisherBooks related by
   newPublisher-> Book[R1];
unrelate newBook from myPublisher across R1;

As shown in the above examples, the associations among objects are named (e.g. R1) so that they can be referred later in the selection expressions. Moreover, each association can be assigned with a verbal phrase (e.g. 'produces and markets'). Note that Publisher and Book are the classes from which bookPublisher and newBook are instantiated. Association of objects to each other may result in creating link objects that represent the abstraction for the properties of the relationship between the objects. For instance, in the following expressions:
relate newBook to myPublisher across R2
   creating ownership;
The object that represents the associations is ownership. It can be referred to by an expression like the following.
select one ownership that relates newBook to
   myPublisher across R2;

4.3.4 Iterations and Selection
In combination with the expressions mentioned above, normal programming language constructs, like if-then-else statements or loops can also be used in describing the semantics of actions. Here is an example:

select many newBooks from instances of Book
where selected.copyright <= 2002 and
selected.copyright >= 1995
if empty newBooks
    // no such books; do proper error handling
foreach b in newBooks {
    b.bookPrice *= 1.1;
}

While abstract and platform independent, such syntax is similar to the imperative programming language syntax. That means, it explicitly specifies the control and data flow of the program. Therefore, using GP technologies does not seem to raise the level of abstraction of the actions.

Next, we describe how the system model specified with abstract diagrams and notations is verified and automatically translated to a platform specific model (or code).

5. DEFECT REDUCTION
It is possible to verify the consistency and completeness of a model automatically. Since the relations and constraints in the system are defined precisely, a model checker would be able to identify the parts of the system that are inconsistent or undefined.

Also, having an executable model of the system that is independent from the target platform, it is possible to run the model to verify the correctness of the application logic and behavior far prior to the actual code generation and integration. This process reduces the number of defects in the resulting system by an order of magnitude[10].

Finally, since the code is generated automatically, the chance of introducing bugs in the implementation phase is significantly reduced (assuming model compilation process is functioning properly). Therefore, the transition from a correct model to a correct system is smooth. This automation reduces the cost of system debugging and maintenance.

6. MODEL COMPILERS
A model compiler is a program that converts a platform independent model of the system to an intermediate representation (e.g. codes in high level programming languages) that can be automatically translated to machine code using existing tools. Similar to the programming language compiler, a model compiler may run a syntax checking before the translation phase.

A model compiler generally accesses a repository of pre-designed components. Such components are either fully implemented reusable components, or abstract design patterns that captures the underlying semantic representation of an arbitrary model. The model compiler uses a set of mechanisms and translation tools to instantiate from the generic patterns, specialize them, and weave them
together with the pre-built components and manually written code.

Next, we first describe the characteristics of the reusable components. Then, we briefly talk about the structure of design patterns. Finally, we describe how translation rules can be used in generating code.

6.1 REUSABLE COMPONENTS
Each reusable component provides a useful and often simple functionality that can be called using a precisely defined interface. Moreover, it is usually expected that using a reusable components does not have any side effect on the global state of the system. Good and rather old examples of such components are UNIX environment tools such as sort, grep, and find. Such tools can be woven together using UNIX pipe mechanism to build more complex programs.

Although reusing pre-built components is extremely beneficial and cost-effective, it is hard in practice to build such components. There are many implementation issues such as the data representation, the component input/output semantics, and possible side effects on the system state. Often reusing a component needs careful customisation that is impossible if the source code is not available and very hard otherwise. Also, it is hard to anticipate and define the functionality that is the common denominator of many applications requirements much in detail, unless the prospective reusable component is fairly simple.

6.2 DESIGN PATTERNS
Design patterns are abstract templates for solving a well-known problem. Although there is an influential and widely known classification of design patterns [13], the definition and classification of such templates might be different from one system to another. However, the ultimate goal is to maximize reusability and customizability of the pre-components. A model compiler automatically maps the model components to such design patterns. Then, by using translation rules (described later), customizes the pattern to fit the logic specified in the model.

To meet system performance or constraints requirements, software projects sometimes require multiple pattern implementations for a given model element type. An example of such situation is developing a single application for various hardware platforms e.g. high-end desktop PC, notebooks, or handheld devices and PDAs.

6.3 TRANSLATION RULES
Translation rules are used to customize abstract patterns that are selected by the model compiler and to weave them together. Figure 4 shows an example of such translation rule that specifies how a Java class is generated based on objects that are defined in the model.

It is also possible to define some rules for automatic optimization and tuning of the system. However, the actual mechanisms have not been explored to the best of our knowledge.

![Figure 4 Highlights of a translation rule that creates a Java Class.](image)

7. CASE STUDY: USING GP IN BUILDING MIDDLEWARE
In this section we demonstrate how the efficiency of middleware software can be improved by taking advantage of the GP concept. What we can expect from synthesizing these two concepts is the automation of generating middleware for building the interoperability. The methodology of pursuing the interoperability of different models is established on the idea of GP [14].

7.1 MIDDLEWARE ELEMENTS
What we expect at the end is a generator that provides a framework, under which the components can be interchangeable parts. This way we can apply the software reuse systematically and strategically to achieve high productivity. We can automate the middleware generation required to compose the other components. These components could be the COTS components that have an important role in the software industry. GP is the key technique to automate the assembly of COTS components to realize a large gain of productivity in the ultimate component market [6]. Many components are already standard and some of them are still non-standard. Figure 5 shows that hardware components such as CPU and PowerPC chipsets are standard. Also many software components are standardized, such as TCP/IP, JVMs, CORBA ORBs and components, Ada, C, C++, etc. While COTS standards promote reuse, they limit design choices. Historically, COTS couples the functional and the QoS aspects.
7.2 MIDDLEWARE GENERATION
In order to get the system to work we need to glue up all the components together using our Middleware glue. What we need to do is to select the right components, configure them and assemble them, after adding the non-standard components. At the end we will expect a code that works for us in this specific one-of-a-kind system. It’s the ordinary way. Though, GP focuses on software families rather than one-of-a-kind systems. Given a product order specification, highly customized end products can be generated automatically from components based on the Generative Domain Model (GDM), a model of specifying a family of systems, namely, that particular domain [15].

We are going to need all aspects of this model, such as structure description, Real-Time description of the system, connector description, QoS Contract descriptions, etc. QoS is an important concern that is to ensure the generated product meets required and predictable quality. Having all these we can use a Generator to give us the delegate contracts that have all these aspects included in it. Figure 6, is showing us this idea.

7.3 MIDDLEWARE GENERATOR AT WORK
The ideal way of middleware generator usage is to utilize it in a system to generate code based on feedback.

The functional and QoS-related aspects of middleware can be improved greatly by advanced R&D on the following topics:

• A common internal representation (ideally auto-generated) for each middleware specification based on generalizing the middleware semantics. This would be in a Platform Specific Domain (PSD) such as CORBA, COM+.

• A generated implementation that is optimized automatically for each target platform and application use-case. These two can provide a feedback system to a Middleware Generator which provides a generated implementation that is optimized automatically for each target platform and application use-case. Based on reflective assessment of platform descriptions and application use-case.

First the generator receives the target platform description and application requirements, which is going to give the Middleware generator enough information to use discrimination network in order to analyze the optimization rules and opportunities.

Then it will generate middleware that is customized for a particular platform and application use-case.

8. CONCLUSIONS
Generative Programming (GP) is a novel programming technique that can improve both speed and quality of software development process, especially for building and maintaining large-scale and complex software. It helps developers find the system defects earlier. Moreover, new features can be integrated into the system automatically.

However, GP is in its infancy stages. That means there are not many tools that support the idea either in academia or industry. Unlike programming language compilers, there is no theoretical framework for how a model such as UML can be automatically translated to code. Descriptions about the model compilers are usually informal and advertising, rather than informative. Therefore, the feasibility of building full-scale model compilers is still under question.
Moreover, it is not clear how GP works for concurrent and real-time systems in which there are a lot of delicate timing constraints. Such constraints are usually satisfied by careful crafting the low level facilities of the underlying hardware and operating system.

Finally, the integration of manually and automatically generated code might become too complicated (if it’s not impossible). This might be specially a serious problem in the maintenance phase in which new features is expected to be added to the system that changes the system model, which in turn changes the generated code that might be a mixture of automatically generated code and manually developed software.

9. REFERENCES
[14] Zhao, W., A Product Line Architecture for Component Model Domains