ABSTRACT

ADEPT: A TOOL TO SUPPORT THE FORMAL ANALYSIS OF SOFTWARE DESIGN

by Sherrie Campbell

Formal specification languages can be used to support the rigorous development of complex software systems when these systems must be of high quality. Unfortunately, writing formal specifications and refining them into designs can be a challenging activity. Use of design patterns, which are a widely accepted design activity, helps create quality designs, but adds further complexity to the design activity.

We have developed a tool ADEPT, Advanced Design Employing Pattern Templates, that aids designers in using both formal specifications and design patterns. The software developer will use the ADEPT tool to guide them through the process of choosing a design pattern that is related to their formal system specification for the purpose of automatically supporting refinement. The user is guided through refining the specification and creating a design that not only incorporates one of the design patterns but also meets the given system specification.
ADEPT: A TOOL TO SUPPORT THE FORMAL ANALYSIS OF SOFTWARE DESIGN

A Thesis

Submitted to the
Faculty of Miami University
in partial fulfillment of
the requirements for the degree of
Master of Computer Science
Department of Computer Science and Systems Analysis

by
Sherrie Campbell
Miami University
Oxford, Ohio
2009

Advisor______________________________________
Dr. Ann E. Kelley Sobel

Reader_______________________________________
Dr. James D. Kiper

Reader_______________________________________
Dr. Gerald Gannod
# TABLE OF CONTENTS

1. Introduction ................................................................................................................. 1
2. Background .................................................................................................................. 3
   2.1 Formal Logic ........................................................................................................... 3
   2.2 Design Patterns ...................................................................................................... 4
   2.3 Formal Specification Languages ............................................................................ 9
   2.4 Tools ....................................................................................................................... 10
3. Implementation ............................................................................................................ 12
   3.1 Research Problem .................................................................................................. 12
   3.2 Environment .......................................................................................................... 14
   3.3 Design Patterns ..................................................................................................... 15
   3.4 Parsing .................................................................................................................. 22
   3.5 Specification and Design Merging ......................................................................... 29
   3.6 SPEC# Program .................................................................................................... 33
4. Example ...................................................................................................................... 38
5. Future Work ............................................................................................................... 43
6. Conclusion .................................................................................................................. 43
7. References .................................................................................................................. 44
8. Appendix ..................................................................................................................... 47
   8.1 Installation Manual ............................................................................................... 47
   8.2 Class Diagram of the System of the Design Objects ............................................. 48
   8.3 Flex & Bison input Grammar ................................................................................. 49
   8.4 User Manual .......................................................................................................... 59
   8.5 Implementation Code ............................................................................................. 75
LIST OF TABLES

Table 1. Behavioral design patterns................................................................................................. 9
Table 2. Symbols that will be used in the first-order logic................................................................. 24
LIST OF FIGURES

Figure 1. Formal logic example .................................................................................................................. 3
Figure 2. Example of ensures clause ............................................................................................................. 10
Figure 3. Using class diagrams for choosing the design pattern ................................................................. 16
Figure 4. Pattern revision tab ..................................................................................................................... 17
Figure 5. Adding instance variables to a class ............................................................................................ 18
Figure 6. Initializing Non-Null types ........................................................................................................ 19
Figure 7. Adding an operation to a class .................................................................................................... 20
Figure 8. Adding a method with parameters to a class .............................................................................. 21
Figure 9. Modifying relationships ............................................................................................................. 22
Figure 10. Example written as a single formula and parsed ...................................................................... 23
Figure 11. Acceptable grammar in Backus-Naur Form ............................................................................. 26
Figure 12. Precedence order ....................................................................................................................... 27
Figure 13. Uploading system specification ............................................................................................... 28
Figure 14. Clarifying specifications .......................................................................................................... 29
Figure 15. Match specifications ................................................................................................................. 30
Figure 16. Editing of matched specifications ........................................................................................... 31
Figure 17. Merged specifications ............................................................................................................... 32
Figure 18. Edited specifications .................................................................................................................. 33
Figure 19. SPEC# project with sort source code added .............................................................................. 34
Figure 20. SPEC# code that is ready for implementation code .............................................................. 35
Figure 21. Some common SPEC# keywords ............................................................................................ 36
Figure 22. Expose and modifies conditions ............................................................................................. 37
Figure 23. Example of modify and expose .............................................................................................. 37
Figure 24. Exception handling .................................................................................................................. 38
Figure 25. Adding Non-Null instance variable ......................................................................................... 39
Figure 26. Adding parameters with Non-Null ......................................................................................... 39
Figure 27. Parsed specifications for chat room ......................................................................................... 40
Figure 28. Specifications for the ForwardAll method in the AbstractChatroom class ......................... 41
Figure 29. Merged chat room program design with specifications ............................................................ 42
Figure 30. Raw SPEC# code created from ADEPT ............................................................................... 42
Figure 31. Start page of ADEPT ............................................................................................................... 59
Figure 32. Step 1 Choosing a pattern ....................................................................................................... 60
Figure 33. Strategy pattern ....................................................................................................................... 60
Figure 34. Step 2 Revising the pattern ...................................................................................................... 61
Figure 35. Editing a class .......................................................................................................................... 62
Figure 36. Changing the class name .......................................................................................................... 62
Figure 37. Adding an instance variable .................................................................................................... 63
Figure 38. Adding a method to a class ...................................................................................................... 64
ACKNOWLEDGEMENT

I would like to thank my committee members, Dr. Gerald Gannod and Dr. James Kiper for proofreading my work and suggesting programming changes. I would like to thank my advisor Dr. Ann E. Kelley Sobel for continuing to push me and all her advice. I also want to thank my husband Dr. Scott Campbell not only for his immense moral support but also for proofreading and debugging.
1. Introduction

Software engineering has not benefitted from the rigor and formality offered by strong design processes that other engineering disciplines commonly utilize. This fact increases the difficulty of designing systems that are provably correct and of high quality. Designing such systems has been a problem that has plagued programmers since large systems started being developed. A problem that is exacerbated with the growth in size and complexity of systems in today’s society. The design of these systems has become more difficult and costly.

The basis of this thesis is the belief that developers will develop better designs if the design can be verified early in the process. Discovering that the design is not correct and needs to be changed can be extremely costly if it is found at the end of the design cycle or after the system is in production. As McMillan states in (McMillan, 2006) “In general, there is a clear need for complete mechanical checking that the implementation of a processor or protocol matches the intended architecture (user model). This requires first of all a definitive model of the architecture – something that is currently lacking even for standardized architectures in the public domain. Second there must be a well defined criterion for determining that is a valid implementation of that architecture.” McMillan gives this justification for verification and the use of models to aid in verifying designs. But creating good models and their subsequent verification is difficult, even for experienced designers. In this work we aim to aid designers in choosing designs from basic patterns and then let them incorporate the requirements from the system specifications directly into the code. We hope to be able to aid designers in their task of creating quality, verifiable code by creating a tool that aids in this process.

In this work we assume that a set of system specifications exist. These specifications would include the system requirements and detailed functionality. The designer will pick a design pattern based on these specifications and will write a formal specification in first order logic based on these specifications as one of the steps. The tool aids the designer in capturing the specifications, selecting a design, and then in checking that the design meets the given specifications. The first step is to create and capture the design in a fashion that allows design testing and/or verification. The second step is to create a model. Modeling the design is a necessary task since the actual design is frequently too complex for verification. The final task is to use the model to verify or prove the design meets the stated specification. There are a number of techniques that can aid in these steps; techniques such as design patterns, modeling, formal methods and verification techniques. Some of the techniques are better suited to certain kinds of systems than others, but all have validity in their own realm.

The system specifications play a very important role in the design of any system. They detail all the requirements, both functional and nonfunctional, of what the system will do. In order to create a system that is provably correct some system specifications are written in formal mathematical terms. In this case, the mathematical terms will be in the form of first-order logic formulas. Trying to prove such formulas by hand can be a difficult, error-prone process, so ADEPT will aid in this by parsing the specifications and giving possible insertion points into the final SPEC# model (Barnett, Rustan, Leino, & Schulte, 2005). SPEC# is an extension of C# developed by Microsoft that allows the insertion of conditions directly into a programming
language. The benefit of this is that SPEC# uses a verifier to statically and dynamically check the truth of these conditions.

In order to create this model we need a good design. We chose to use the approach of design patterns so each user can build a design based upon a well known and highly utilized design. The use of design patterns as was brought to light by Gamma et al., a group of researchers who, due to their stature, came to be known as the Gang of Four. As the Gang of Four state in (Gamma, Helm, Johnson, & Vlissides, 2005), experienced programmers do not start a program from scratch, they use designs that have worked for them on similar problems. The Gang of Four created a series of patterns that solve standard problems that are frequently encountered in system designs. These can be used as the starting point for creating a design that can be tailored to meet the user’s specific needs.

Modeling is a necessary tool in verification. Systems are frequently so large that the whole system cannot be checked for every aspect, so models are created that abstract out the important aspects of the design. These design models can be checked for consistency and correct behavior. There are a number of techniques for verifying models that will be discussed in more detail in the next section. However, there are some flaws with these standard modeling techniques. Uchitel (Uchitel, Chatley, Kramer, & Magee, 2003) indicates that when creating a model it can be a long complex process that does not give any feedback until the very end of the process, when the model is completely finished. While the model eventually discovers potential problems, the changes are more expensive than if they had been found earlier in the process. Uchitel also indicates that creating the model in the abstraction step is critical to ensure the model exactly represents the original problem.

In this work we are creating a tool that ties these important steps in the design cycle together. Tool support can be very beneficial, especially to a beginning design or in creating more robust code. The tool that we have created, ADEPT, takes the user through many steps of the design process as indicated by the following basic steps: 1) The user will select a design pattern from a set of templates and their accompanying descriptions. The user can pick an appropriate design pattern for the basis of their model based on the system design specifications. 2) The class specifications and relationships for the chosen design pattern are used to create the basic design model. From this basic model the user can change the class names to fit their system and add new classes. They can also refine each class by adding instance variables and methods. Once all the classes have been finalized, relationships can be added and changed to further refine the system. 3) The user will upload their system specification using first-order logic. In this step a utility is run to ensure that the specifications are well formed. This helps in checking that the mathematical formulas are valid. 4) The user’s specifications are broken into clauses and matched with possible insertion points into the model. This is one of the most critical and difficult steps. We match clauses based on the class, instance variable and names. At the current time it is not possible to detect if the clause should be an invariant, pre or postcondition so the user must make that judgment in an editing section. 5) After all the specifications have been merged into the model; it is ready to be transformed into the actual specification language SPEC#. By coding the specification statements into the program, the model can then be checked using SPEC#’s static and dynamic verification processes. Thus the tool takes the user through
the multi-step process that takes them from their system specification to a model that is verifiable and ready to be programmed into a full system.

In this work we have defined a process that incorporates the use of the four existing methodologies of design patterns, modeling, formal logic and verification. To instantiate this process we created the ADEPT tool, Advanced Design Employing Pattern Templates, which follows the above design flow. The tool allows a user to take a formally specified system and create a model that is mathematically correct according to these design specifications. Some background about each of the logics, design patterns, specifications and tools will be discussed in Section 2. Details about the environment and workings of the ADEPT system will be covered in Section 3. A detailed example will be covered in Section 4.

2. Background

To fully discuss how we can prove systems to be correct we need to have some background information about how those systems are specified. Some programs that are not critical may not need the rigor of formal specification and verification. However, programs that are part of critical systems, are time sensitive, or work with sensitive information need to be held to a higher standard. It is systems such as these in which a more formal approach is needed. To be able to specify and verify such systems, knowledge of formal logic and system specification possibilities are needed. The system specification is a high level specification that needs to be translated into a system design specification so information on design techniques and modeling is also needed.

2.1 Formal Logic

Formal logic is one method for proving software correct. In formal logic, the requirements of the system are all expressed in mathematical terms. In the past it has been a standalone technique that is often performed by hand. As technology has advanced, many of the theories from the formal logic have been incorporated into other verification approaches. To express a system in formal logic, each function of the system needs to be analyzed to determine what the effect of changing state should be. A formula can then be stated to ensure the behavior is correct. For a very simple example, if you were adding a single element to an array the count of elements in the array when finished should be one greater than before and the count should be less than the total length allocated for the array. This can be expressed as the formulas in Figure 1. In this formula old refers the value of a variable before a method began which can be used for comparison purposes.

\[
\begin{align*}
\text{Array.Count} &= \text{old(Array.Count)} + 1 \\
\text{Array.Count} &\leq \text{Array.Length}
\end{align*}
\]

Figure 1. Formal logic example
One such formal logic system is Linear Temporal Logic (LTL). Sistla and Clark were trying to find a way to reason about parallel programs and proposed that LTL would be able to accomplish this (Sistla, A. P. and Clarke, 1982). LTL uses propositional variables but does not need to have a specific state or time. Instead it uses temporal operators such as always, eventually and until. This allows checks to be made for mutual exclusion, deadlock and starvation problems. The key to LTL is coming up with a formula that is based on program axioms which detail the specific processes in the program (Sistla, A. P. and Clarke, 1982).

For systems that deal with time, Interval Temporal Logic (ITL) (Cau, Moszkowski, & Zedan, 2007), which uses propositional and first-order reasoning to model systems, can be used. ITL is a more specific form of LTL that is best used when checking for safety, liveness and projected-time. An interval is the key to ITL. Intervals are a sequence of states in which the states have mapped a set of variables to values. There are several tools which help in using ITL. Tempura, which is an interpreter, is used to evaluate the temporal logic. The temporal formulas are inserted into the code as assertions and are used to check the sequence of states while the program is running. To be valid, the sequence generated needs to be a valid sequence for the desired property. Theorems can be proved using a system called PVS (Owre, 2006) which stands for "Prototype Verification System." PVS uses higher order logic for the writing and checking of formal proofs and is used in a number of model checking systems such as Tempura.

Sometimes first-order logic has been dismissed as not being expressive enough to capture design modeling requirements. However, in (Halpern & Weissman, 2008), Halpern uses a subset of first-order logic to create a system to reason about policies. For a given set of policies and a particular environment you can use first-order logic to make sure each action is either permitted or forbidden and also that the policies are consistent. He permitted the use of function symbols in his language and was able to create a large set of possibilities that were easily managed and proved consistent. We are expanding on work done by Sobel (Sobel, 1999) that used first-order logic and transaction processing to create a specification and verification model. By using predicate and existential variables, we can specify what the states of the program should be in before and after method calls. This is indeed expressive enough to show correctness in state based systems.

### 2.2 Design Patterns

Creating a robust reusable object-oriented design is a task that is very difficult. Once a designer has created a good solution to a particular problem he tends to use that solution over and over. Others also can use an existing design as a basis for their work. As (Soundarajan & Hallstrom, 2004) points out, designs are the collective wisdom of a large number of people. New designers can then learn from this collective wisdom by using existing designs. Actual solutions that are started from design patterns lead to new different designs depending on the current conditions and requirements, but the basic design will be the same. Recognizing and cataloging these designs and creating patterns were pioneered by the Gang of Four (Gamma et al., 2005). In their book, they take many common problems that software engineers have to solve and give the basic pattern for the solution. The beauty of this method is that these solutions have been tested on many different systems and have been proven to be effective. They have been revised over the
years so that they are solid and efficient and they make the resulting code portable and reusable. Gamma et al. (Gamma et al., 2005) state that

“A design pattern names, abstracts, and identifies the key aspects of a common design structure that make it useful for creating a reusable object-oriented design. The design pattern identifies the participating classes and instances, their roles and collaborations, and the distribution of responsibilities. Each design pattern focuses on a particular object-oriented design problem or issue. It describes when it applies, whether it can be applied in view of other design constraints, and the consequences and trade-offs of its use.”

The design pattern’s information allows a designer to determine if the pattern will solve the problem at hand and whether or not the consequences of its use will be worth it. For programmers new to the field, these patterns can be extremely valuable in creating good designs. There are three basic types of design patterns that are described in (Gamma et al., 2005): Creational Patterns, Structural Patterns and Behavioral Patterns. The ADEPT system currently implements the original Behavioral Patterns. Creational Patterns deal with the creation of objects so that they are created in the best possible way for the particular environment. Structural Patterns focus on the relationships between different objects. The Behavioral Patterns as summarized in Table 1 are used mainly for characterizing the behavior between objects in a system. Behavioral patterns deal with how the objects communicate with each other and how control flows through a program. The table gives information about the pattern, when it should be used and a class diagram.¹

**Chain of Responsibility**

Avoids coupling of sender and receiver by allowing more than one object to handle the request. Objects are put in a chain and one of them handles request.

- Multiple objects can handle request
- Multiple object can receive requests dynamically
- The handling objects is specified dynamically

¹ Copyright © 2001 - 2009 Data & Object Factory, LLC. All rights reserved
**Command**

Make a request of an object so that requests can be queued, logged, and sent as parameters.

- Want to parameterize object by action
- When undo of command is needed
- When logging of commands is needed
- Have primitive operations like in transaction processing

**Interpreter**

Given a language and a grammar, the interpreter will interpret strings in the language

- Grammar is simple
- Do not need highly efficient algorithm

**Iterator**

Provide sequential access to a collection of objects without exposing their underlying structure.

- Want to access objects and maintain encapsulation
- Want multiple traversals of structure
- Want to iterate through various kinds of objects
**Mediator**

The mediator decouples dependencies between classes by having a single class act as the point of communication between all classes

- The objects communications patterns are complex
- Want to communicate with different objects
- Have a behavior that you want to split between several classes

**Memento**

Without exposing object, take a snapshot of the current state of the object for future restoration

- Need a snapshot of object to be able to restore to AND
- Getting the information directly from the object would expose object details

**Observer**

Create a one to many relationship so that all dependents are notified if the original object changes state

- When multiple aspects of class are dependent on each other
- When changing one aspect requires changes to an unknown number of other aspects
- Being able to notify others about changes without knowing who the objects are
State
Allow an object to change behavior after a state change. Object appears to change classes
- Objects behavior depends on its state
- Operations have complex conditional statements that depend on its state

Strategy
Allows a family of algorithms to be used interchangeably. Algorithm can change independently from client.
- Have related classes that only differ in behavior.
- Have different variations on an algorithm
- When an algorithm uses data that should be encapsulated
- Class has multiple behaviors that can be used instead of conditions

Template Method
Create the skeleton of a method and let subclasses redefine certain portions of the algorithm without changing the outcome.
- Have algorithm with invariant parts and varying parts
- Have behavior that can be factored out to avoid duplication
- Want to control extensions
Visitor

Allows operations on a structure without changing the classes of the structure.

- Many classes that have different interfaces
- Have many operations that need to be performed on objects
- Need to change operation to be performed on objects

For example, the Chain of Responsibility pattern decouples the requester from the object that actually performs the requests. Multiple requests can be chained together. The Client sends one or more requests to the handler that can then chain the requests together. The Client is decoupled from the actual handlers, and the handlers do not need to have any knowledge of each other. The Handler is the only object that needs all the specifics. Class diagrams such as those shown above will be used in the ADEPT system to help the designer choose which pattern will best fit their needs.

**2.3 Formal Specification Languages**

To aid in the design process, formal specification languages were developed. These languages allow the design to be specified in a precise, consistent manner. The design then iterates through a series of checks and refinements to help ensure the correctness of the design. Some of the specification languages include actual coding languages and diagramming techniques.

JML, Java Modeling Language, (Leavens & Cheon, 2006), iContract (Kramer, 1998), Eiffel (Meyer, 1987) and SPEC# (Barnett et al., 2005) are some of the formal specification languages that use design by contract. The contract is between the method and the caller and guarantees that certain conditions exist. Preconditions must be satisfied by the calling programming construct in order for the method to be called. Postconditions must be satisfied before the method can complete. The nice thing about JML and SPEC# are that the conditions are inserted directly into the code so they are always present. Other benefits that specification languages provide are
documentation, showing where the blame lies (caller or method), efficiency, and supporting modular reasoning. For example, to implement the array adding function that was specified in Figure 1 we will show how the conditions would look in a specification language as shown in Figure 2.

```java
class Array
{
    public virtual void Add(object o)
    ensures Array.Count = old(Array.Count) + 1;
    ensures Array.Count <= Array.Length;
    ...
}
```

Figure 2. Example of ensures clause

The ensures clause is a postcondition that will not allow a legal exit from the method unless the statement is true. By using statements such as these, program correctness is built into the program.

PVS is a specification language that is used for proving theorems (Owre, 2006). It includes a type checker, predefined theories and an interactive theorem checker that creates code and documentation. It is written in LISP and has a testing function and formalized libraries. The PVS language can define new types or use standard predefined ones and also has a number of standard abstract data types like lists and trees. The prover portion of PVS uses primitive inference procedures and sequent calculus in an interactive environment.

A slight variation from the languages is a technique that was used by France and Ghosh to specify patterns using UML (France, Kim, Sudipto Ghosh, & Song, 2004). UML has been growing in use for modeling and the authors wanted to describe the patterns and their interactions using this model. Both class diagrams and sequence diagrams are used to fully describe each pattern. The class diagrams are used to specify the classes that participate in the pattern and the associations between them. The sequence diagram is used more for describing the solution of how the classes actually interact. By using the combination of these two parts of UML they were able to specify patterns such that they could be easily implemented using a tool.

### 2.4 Tools

Many tools have been designed to ensure various aspects of system design. Many are based on one or more of the logics and languages already stated. Some of the tools are generic design tools that could be used with any implementation of a system, and others are implementation dependent on languages such as Java and C. JML iContract and SPEC#, which were already mentioned, are language dependent tools such as this.

One way of specifying a design is to use scenarios, which are good at describing the behavior of a system. The Labelled Transition System Analyzer (LTSA) is a tool that enables the use of scenario specification (Uchitel et al., 2003). Modeling concurrent systems can sometimes be very difficult and results of correctness are frequently not achieved until the design is complete. LTSL provides feedback during the design process. It also provides four different kinds of
output to aid in the design. First it creates a specification detailing the aspects of the system. It then also creates three models – an architecture model, a trace model that describes behavior and a constraint model. The models create message sequencing charts that let you see if scenarios are missing and provide models that can be reasoned about.

The Symbolic Model Checker, which is known as SMV, is used as part of a number of tools and packages that were found during the background research (McMillan, 2006). SMV is less formal than formal verification methods. It allows you to build a model and checks finite systems against specifications using a temporal logic called computational tree logic. SMV is good for analyzing synchronous and asynchronous nondeterministic processes but only supports finite data types such as Boolean scalars and fixed arrays. While state explosion was originally a problem with this technique, McMillan developed a traversal technique called Binary Decision Diagrams which greatly increased the number of states that could be checked (McMillan, 2006).

The departments of Information Technology at Uppsala University, Sweden, and Computer Science at Aalborg University, Denmark, have developed a tool named UPPAAL (Department of Information Technology at Uppsala University (UPP) in Sweden & Department of Computer Science at Aalborg University (AAL) in Denmark, 2007). The tool is designed for modeling, simulation and verification of real-time systems. UPPAAL has three distinct parts: the description language, simulator and model checker. The description language models system behavior for non-deterministic systems using various data types. The simulator goes through possible execution paths and can provide fault detection early in the design process. Whereas the model checker does an exhaustive dynamic behaviors check of the system. It employs the use of invariants and explores state spaces for verification.

Spin is a popular model checker that has been around since the early 1980’s (On-the-fly, LTL model checking with SPIN 2007). Spin uses LTL in its automata based checker that uses Promela (Process Meta Language). There are 3 ways in which to use SPIN. The simulator provides rapid prototyping while allowing random, guided or interactive simulations. The verifier proves the correctness of the programs. The proof approximation system validates systems with small to large state spaces. Though SPIN did not initially support any language it now supports embedded C code in the specifications.

SPEC# is a tool that has been designed by Microsoft to insert specifications directly into C# programs (Barnett et al., 2005). The designers propose that programming would be better if more assumptions about the program were actually recorded and enforced. By putting these assumptions in the actual code, the intent of the programmer is clear and these intentions can be checked at compile and run time. The SPEC# package already contains a language, compiler and verifier so a separate translation into another verification system is not needed. SPEC# allows reasoning about objects and does both static and dynamic verification.

SPEC#, for instance, is based on C# but then has a number of additional functions in order to add requirements checking. An ensures clause indicates a postcondition that must be true before a method is permitted to exit. Invariants can be declared that state what condition must be true in the steady state of any object. More information about the syntax and examples of this will be given in Section 3.6.
In addition to the tools listed above there are many tools that do model checking such as

- Kronos (Yovine, 2002) – uses timed automata and correctness in modeling real-time systems.
- TRIO (The TRIO home page.2005) – is a specification language and modeling tool for critical real-time systems.
- Murphi (University of Utah, School of Computing) – is a descriptive language using guarded commands (condition/action rules) for concurrent systems.
- SGM (State Graph Manipulators) (National Chung Cheng University, Embedded Systems Lab, Department of Computer Science and Information Engineering, 2008) – is a tool to create state graphs. The modeling portion does safe chart pre-processing and it includes other verification techniques as well.
- Verisoft (Godefroid, 2002) – is a model checker developed to do unit testing and system testing for concurrent, reactive and real-time systems.
- CADP (Construction and Analysis of Distributed Processes) (VASY research team of INRIA, 2008) – is a toolbox that offers compilers, parallel-model checking and verification algorithms.
- SMI (Symbolic Model Interface) (Bozga) – uses decision diagrams to build a symbolic representation of the finite state systems. It was designed for communication protocols.
- InVeSt (Kramer, 1998) – verifies the invariants of infinite state systems using the PVS prover.

3. Implementation

3.1 Research Problem

The goal of this work is to create a process and a tool supporting the process, that aids the designer in 1) picking a design pattern to help in articulating the program design, 2) capturing the system specifications, and 3) using the system specifications with the design to check the conditions as the model is executed. The designer will be led through several tasks – entering of the pattern selection, system specification, and modeling. This process will integrate existing approaches such as formal methods, design patterns, and modeling.

Design patterns can play a vital role in the developer’s environment; Soundarajan (Soundarajan & Hallstrom, 2004) shows that when trying to find a solution, experience is a very important factor in creating a correct solution. Design patterns are set of solutions that have been shown to work for a given set of problems and thus they will be the starting point for each design. The beauty of design patterns are that they include information about what kind of problem the pattern is best used for, what information will be needed for interfaces, and the cost of using the patterns such as runtime, size, and so on.
While attractive, design patterns have a problem in that many people do not know how to implement them. Though students in software engineering classes may learn about design patterns, they frequently do not get the opportunity to try hands-on development of a project with them. This tool helps inexperienced (and experienced) designers by starting with the basic template of the design pattern and then guiding the user as they expand and individualize the design pattern to meet the user’s needs. The tool helps the user to set up the classes and the relationships, thus creating the basis of system, which helps conceptualize how the design goes together.

There are several existing research projects that have looked at the use of design patterns. These projects include tools to help develop code from existing patterns, code templates and specification templates. The projects are:

- Design Pattern Toolkit http://www.alphaworks.ibm.com/tech/dptk (IBM, 2007) – an Eclipse add-on with a Model-View-Controller architecture. Its main use is for web applications but can be used for text output that can be patterned.
- Tool support for object-oriented (design) patterns http://www.serc.nl/people/florijn/work/patterns.html (Florijn, 2003) – a tool designed to take a pattern from a template and integrate it into a program design and then check invariants.
- http://www.hillside.net/patterns/tools/ (Hillside.net, 2007) – has a pattern template library of C++ code that includes templates for Lists, Collections, Dynamic array, Pointer array, Composite, Flyweight and a Dynamically reconfigurable finite state machine.
- http://patterns.projects.cis.ksu.edu/ (SAnToS Laboratory, 2001) – gives specifications in standard patterns that can be used in various verification tools.

However just having a design pattern is not enough to complete a design. Design patterns do not include the system specification, which is a very important part of any system design. When designing a system, a developer will break the system up into various modules and elements that interact with each other but specifications are difficult to use. Wood discusses the inadequacies of the detailed specifications in (Wood, 1990). The documents that capture the requirements that specify how the system should function are usually fairly comprehensive. The details about the specifications are frequently lacking, and this frequently causes system inconsistencies and errors. Wood states one solution to this is the use of formal specifications, which allows the system to be specified in such a way, that it can be reasoned about and thereby removes some of the previous problems. Hence we decided the specifications need to be formal extractions about the system.

One of the reasons that specifications are so rarely used is because of their complex mathematical reasoning. In this system we use just first-order logic to create the system specifications. France et al. (France et al., 2004) state that people do not use system specifications in the design process because it frequently requires extensive high order
mathematic processing which many people do not possess. By using first-order logic, we are attempting to make mathematically correct programming more accessible to more people.

Requirements are stated in terms of what constraints need to be placed on the system. They describe what the system needs to do and under what conditions the specifications need to hold. Many of the tools and languages we have talked about require learning about complex math and specification languages. First-order logic, on the other hand, is taught in most Computer Science and Software Engineering degrees so there is no need to learn a new logic.

Eden indicates that there are multiple programming models that will satisfy the system design (Eden & Kazman, 2003). Any individual programming model needs to satisfy all the requirements in the specification. One of the difficult parts of the process is making the transition from the specification to the model. Specifications can get long and complex. Determining how they actually relate to the implementation is difficult. In addition, in most programs it is very difficult to prove that the program satisfies the requirements of the system specifications. Tests can be created to check that conditions are satisfied, but they do not always prove mathematical equivalence. In this work we attempt to make this transition a little smoother. The specification is first checked for mathematical correctness. It is then broken down into pieces that will fit into the code that is created in the modeling section. The tool aids the user through the design and verification process. By using this stepwise approach, it should make the process a little easier for someone not familiar with designing systems in this fashion.

Our work varies from other tools previously mentioned in that ADEPT takes the user’s specification and programmatically helps them break these specifications down so that they can be physically inserted into a specification language. Each clause in the specification is checked for matches in the classes and methods of the implementation. The user is walked through translating the clauses into class invariants, preconditions and postconditions. Class invariants make sure that that instance variables hold to the conditions at all stable times (after an “atomic” action). Preconditions ensure that certain conditions are true before a method is executed. When the method is exiting, the post conditions are checked to verify that the conditions are true. If any of these conditions are not met an error is thrown. Some conditions can be checked statically to determine if there are programming errors. These give errors similar to syntax errors indicating that a condition will not be met. The remaining conditions are checked dynamically during runtime to check that the program does indeed meet the mathematical system specifications. ADEPT facilitates the combination of the system specifications into a model, created from a design pattern, that can be proved through a verifier.

### 3.2 Environment

With the decision to create a system that combined specifications and the design we needed to use an environment that was based on a formal specification language. We decided on the SPEC# language (Barnett, Fähndrich, Leino, Schulte, & Venter, 2009). SPEC# is a specification language that extends the C# language with additional constructs that allow assertions about the code. This allows the programmer or designer to explicitly state conditions that should be true for the code to run, directly into the code. We chose SPEC# for a number of reasons. First, the
SPEC# package already contains a language, compiler and verifier so a separate translation into another verification system is not needed. Therefore, with some guidance, the user can insert their system specification into a model created from a design pattern, and then have the code do self verification. Second, SPEC# is an extension of C#. Users can convert previous projects into SPEC# and thus benefit from its verification capabilities. Lastly, SPEC# was created by the Microsoft designers and is therefore available on the widely used platform .NET and Visual Studio, and is easily accessible to many people. Anyone who has C# with Visual Studio can currently add SPEC# for free. The only limitation is that it can only be used for non-commercial uses.

SPEC# has several different checking capabilities. It has static checking that can check for non-null types, and can also perform exception checking. It has a dynamic capability that produces run-time checks for pre and post-conditions and object invariants. There is also an optional mathematical program verification system that can be used at design time, compile time or from the command line. SPEC# can be run via the command line but in this project everything will be set up in the Visual Studio environment. As a result of this design decision all other components of the system needed to be compatible with the Visual Studio environment as well.

One problem is the complexity of the system specification and its expression. The system specification is typically a long and complex mathematical formula. It is necessary to break this formula into smaller pieces that will correlate to sections of the design model. A parsing program will be used to do this. The parsing uses Flex (Flex for windows.2004) and Bison (Bison for windows.2009), which are based on the classic Lex and YACC parsing utilities. The original utilities were on a UNIX system but Flex and Bison can be run via windows command line options. The options will be executed from a system process within the ADEPT tool.

The User Interface portion of the tool was written in C#. This interface takes the class design that the user creates using UML, allows editing of the specifications that have been extracted during the parsing stage, and will create a final model that is created in SPEC#. The user will then compile their code in SPEC# to check their invariants. In the following sections we will describe each of the steps that need to be taken to complete the entire process.

### 3.3 Design Patterns

#### 3.3.1 Choosing a design pattern

The user will be presented with a list of design patterns with a description of the each pattern’s uses. Each pattern, when clicked, displays the class diagram for that particular design in the lower right hand portion of the screen. An example of this is shown in Figure 3. Here the Strategy pattern is chosen and the UML diagram displayed shows that there are three classes that are inherent to this design: a context that is requesting some kind of algorithm, a Strategy class and one or more ConcreteStrategy Classes. This pattern avoids coupling between the Client and the actual classes handling the requests. By having a Strategy Class, it can be used for actually calling any one of several different strategies to perform a particular algorithm.
When a pattern is chosen, the template of the classes and relationships for that pattern is used to create the basic design of the system. Each of the pattern templates is stored in an AbstractDesign object. The AbstractDesign contains the basic pattern information, a collection of DesignClass objects and a collection of Relationship objects. The DesignClass objects contain all the information about each class such as names, instance variables and methods. Each Relationship object indicates what kind of relationship is stored and contains pointers to each of the class objects in the relationship. The information in the AbstractDesign is used to build the new design object that contains all the information for building a new system. The AbstractDesign is cloned and current project information is added when a pattern is chosen to create the new Design object. This gives the user the basic information necessary to create the system model.

![Figure 3. Using class diagrams for choosing the design pattern](image)

### 3.3.2 Modifying the Design

Once a design pattern has been chosen, the process of customizing it to the particular application begins. The template provides the basic structure but this needs to be expanded to include classes specific to the user’s application. Additionally the user customizes the methods and instance variables that are associated with these classes. Though there are a number of tools that perform UML editing (Smart draw, 2009), (Smart draw, 2009; Altova, 2009), (SparkX Systems, 2009), we found that they had copyright restrictions, or that their output was not compatible with the code we were creating. We thus created our own editor for making changes to the classes and relationships. This gave more control over how the resulting classes could be created and modified.
The process of customization starts with the Pattern Revision. This step allows classes to be customized and new classes added and deleted. An example of this process is shown in Figure 4. As each class is renamed, the original class name from the pattern is maintained so that the user can relate the design back to the original pattern. This history is indicated by using a naming structure like template - newname. Each of the names must be distinct and the user should select names that make sense in the context of the final program. The example in Figure 4 shows that two of the classes have been renamed to reflect the names in the Sorting Program.

When editing a class, the user has many choices. They can change the name of the class, add, and/or edit instance variables. This process is depicted in Figure 5. Here the user has added the sortArray instance variable to the SortStrategy class. This variable is important because it is a key link to several of the clauses in the specification. This class will update the array and we need to make sure that the resulting array meets the requirements as captured in the specifications. The steps to add the necessary conditions to this class will be shown later in Section 3.5.1.
In SPEC#, instance variables can be declared to have a non-null value, which is indicated with an “!” after the object declaration type. The field initialization has to be done slightly differently in SPEC# than in C#. This is necessary so that the constructors can access all the fields before the object is actually used. An example in (Barnett et al., 2005), repeated in Figure 6 below, shows that this can be accomplished by changing the order of object creation and calling the base class. In the first part of the example, which is written in C#, the base class is created before enrollment information is created, so if the transcript information does not exist, the Student will be created with a null pointer to their transcript. In the second part of the example, which is written in SPEC#, the enrollment object is created first and therefore must exist before the student object is created. The constructors are assumed to be Delayed which means that only references that initialized fields in the objects may be used in the constructor. Calls to any methods within the constructor would have to be also of type Delayed. Fields would have to be declared NonDelayed. If you wanted to be able to refer to fields in the class the constructor. All fields need to be initialized first, before the base constructor is called. In order to accomplish this, the base class can be called from any point in the constructor allowing for the initialization of all fields as shown in the third example.
C# initialization of the student class
Class Student : Person {
    Transcript t;
    public Student (string name, EnrollmentInfo Ei)
        : base (name) {
        t = new Transcript(ei);
    }
}

SPEC# initialization of the student class
Class Student : Person {
    Transcript! t;
    public Student (string name, EnrollmentInfo! Ei)
        : t = new Transcript(ei),
        base (name) {
    }

SPEC# initialization of the student class with NonDelayed
Class Student : Person {
    Transcript! t;
    [NonDelayed]
    public Student (string name, EnrollmentInfo! Ei)
        {
        t = new Transcript(ei);
        base (name);
    }

Figure 6. Initializing Non-Null types

Null references are a frequent error in many programming systems and this provides a convenient and easy way to check for null values. Using this approach, the C# compiler will check to make sure that initialization code has been called to initialize values before objects are actually used; otherwise, a compile error will be thrown and the null value detected.
After users have entered any class instance variables, they can enter operations for the class, as shown in Figure 7. There are five basic fields in the method class that need to be entered. The class name is entered using standard C# variable naming and the class will be checked to make sure that no methods or instance variables by this name already exist in the current class. The accessibility option is required, which indicates whether the method is public, private or protected. Typing indicates if the class has any special designations such as override, abstract or virtual. This is an optional field that can be left blank. The Pure option can be chosen if the method will have no side effect on the objects in the class (i.e. no changes are made to the object). Only Pure methods can be called from an ensures clause. Last, one of the four return types must be chosen. If there are no parameters to add, the method can be saved, which will add the method and close the Add Method group box. If there are parameters to add, clicking on the Add Parameter button can be done as soon as all the method values are valid.

Figure 7. Adding an operation to a class
Methods can have 0 or more parameters added. When add parameter is chosen, if the method has not been saved, the object will be created in order to add the parameters to it. It will not be actually saved to the class until the Save button is hit in the Add Method group box. The parameters use the same editing section as the instance variables with the added field of parameter type which is IN, OUT, REF or RETURN as shown in Figure 8. The specialization that is added in these editing diagrams will eventually be used to create the code for the program itself and will also be used in the matching of the specifications to the design of the program.

Figure 8. Adding a method with parameters to a class

Relationships between classes are crucial to the design process. The initial relationships between the classes in the pattern are set in the template and cannot be changed or deleted. If new classes are added, relationships between these classes should be added to help enhance the design. In order to modify the design’s relationships, the Add Relationship option is chosen from Figure 4 and the relationship window will appear as shown in Figure 9. The class that is at the start of the relationship goes in the first class box. The class that is the related class or the one that will have the arrowhead goes in the related class box. When save is clicked the relationship will be added to the relationship window. These relationships are used to set up the base classes for inheritance. They also help the user see how classes interact with each other.

After the relationships have been completed, the basic design of the system is complete. The design can be printed, to help in the subsequent steps. The printed version shows the original UML design, and the new design listed out by class, including instance variables and methods. The next step in the process is to upload the system specifications. This is done in the SystemSpec tab and will be discussed in the next section.
3.4 Parsing

As indicated previously, system specifications are very important in creating systems that meet their requirements. These specifications, written in formal logic, are critical to this design tool. We assume that the specifications for the system exist but they need to be entered into ADEPT and transformed such that they can be used at the class and method level. To do this, a parser will be used to break the specifications into smaller distinct provable parts.

3.4.1 First-Order Logic

The specification is expressed in first-order logic and must be written as a well formed formula (WFF). As in any WFF, it is made up of one or more other WFFs. The specification can possibly be a quite long formula that will evaluate to true or false. We can tell if the whole formula is well formed by parsing the whole system specification into each of these individual formulas. We can also take each individual piece and use it in the model for verification purposes. The specification as a whole is quite complex, but each individual formula actually pertains to just a small subset of the system and it is that subset that we identify.

As a simple example to demonstrate the system, we will define a simple sorting routine and then take this through the system. The system specifications for a sort are shown in Figure 10. These
specifications ensure that the sorted array is the same length as the unsorted array and that every element of the new array is larger than the previous element.

**Original Logic formula**

\[
\text{sortArray.Length} = \text{old(sortArray.Length)} \land \\
\text{Permutation(old(sortArray))} \land \\
\forall i \ (0 \leq i < \text{sortArray.Length}-1(\text{sortArray}[i] \leq \text{sortArray}[i+1]))
\]

**Parsed clauses**

\[
\text{sortArray.Length} = \text{old(sortArray.Length)} \\
\text{Permutation(old(sortArray))} \\
\text{sortArray.Length} = \text{old(sortArray.Length)} \land \text{Permutation(old(sortArray))} \\
\text{sortArray}[i] \leq \text{sortArray}[i+1] \\
\forall i \ (0 \leq i < \text{sortArray.Length}-1(\text{sortArray}[i] \leq \text{sortArray}[i+1]))
\]

Figure 10. Example written as a single formula and parsed

The system specifications can quickly become quite complex and breaking them down into components that can be implemented programmatically by hand would be a difficult task. The ADEPT package takes the specifications and parses them into clauses that will then be inserted directly into the programming model. A clause is an individual piece of the whole formula that is itself a WFF. In Figure 10, the parser would take the original formula and parse it into all the individual formulas shown at the bottom of the figure. This process thus removes the user’s burden of translating the specifications into code. The basics of the specification and the related syntax will be described in Section 3.4.2. How this relates to the SPEC# programming language will be described in 3.6.1.

### 3.4.2 Entering System Specifications

The next step in the design process is to enter the system specifications into the package. This is accomplished by uploading a text file in the System Spec tab. The specifications themselves are first-order logic that is written as a single formula. When creating the specifications we will be using the symbols in Table 2 and grammar in Figure 12 for the syntax. Note that the Or and And symbols are created using the || and && respectively. So the previous example would be written as:

\[
\text{sortArray.Length} = \text{old(sortArray.Length)} \land \\
\text{Permutation(old(sortArray))} \land \\
\forall i \ (0 \leq i < \text{sortArray.Length}-1(\text{sortArray}[i] \leq \text{sortArray}[i+1]))
\]

Figure 11. Sort specification in first-order logic
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>=&gt;</td>
<td>Implication $\Rightarrow$</td>
</tr>
<tr>
<td>⇔</td>
<td>If and Only IF $\iff$</td>
</tr>
<tr>
<td>A</td>
<td>ForAll $\forall$</td>
</tr>
<tr>
<td>E</td>
<td>There Exists $\exists$</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>And $\land$</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>!</td>
<td>Not</td>
</tr>
<tr>
<td>(</td>
<td>Grouping Start</td>
</tr>
<tr>
<td>)</td>
<td>Grouping End</td>
</tr>
<tr>
<td>[</td>
<td>Array subscript start</td>
</tr>
<tr>
<td>]</td>
<td>Array subscript end</td>
</tr>
<tr>
<td>&lt;</td>
<td>Less than</td>
</tr>
<tr>
<td>&gt;</td>
<td>Greater than</td>
</tr>
<tr>
<td>=</td>
<td>Equality</td>
</tr>
<tr>
<td>!=</td>
<td>Not equal</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Less than or Equal $&lt;$</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Greater than or Equal</td>
</tr>
<tr>
<td>,</td>
<td>Separate terms in a list</td>
</tr>
<tr>
<td>.</td>
<td>Separate parts of a identifier name</td>
</tr>
<tr>
<td>+ - / * %</td>
<td>Standard mathematical operators</td>
</tr>
<tr>
<td>null</td>
<td>Null value</td>
</tr>
<tr>
<td>constant</td>
<td>Integer or Real number constants will be accepted</td>
</tr>
<tr>
<td>identifierName</td>
<td>String constant using standard C# naming conventions, except cannot start with A or E</td>
</tr>
<tr>
<td>identifierName()</td>
<td>Predicate or function</td>
</tr>
</tbody>
</table>

Table 2. Symbols that will be used in the first-order logic.

With an identifier, a single identifier name could be used such as Employee or the “.” notation can be used to create identifier combinations. For example, if you had the method Hired in the Employee class, Employee.Hired would be a valid symbol. This also allows for the use of standard special functions which are inherent to SPEC#. While these functions are not normally
used in dot notation in SPEC#, we will use them in that fashion in the specification in order to preserve the formula flow. The functions will then be translated into the actual SPEC# syntax during the merging phase.

- **result** denotes method return value (use as `methodName.result`)
- **old(E)** denotes E evaluated in method’s pre-state

The following are all functions that can be performed on arrays.

- **sum, product, min, max, count** (use as `arrayName.functionName`)
  for example `array.min`

These functions have been added to SPEC# to help express conditions so that the verifier can check them more easily. *Result* is good for checking to make sure that methods have return values that are within expected ranges or values. The *old* function is imperative in many state checking conditions because the current state depends on the previous state. The various array functions can also be useful for checking various conditions. These array quantifications can be used to find a value, such as min, and then use it in other parts of the specification.

The parsing takes a string and breaks it into tokens. To accomplish this, the string needs to be composed of regular expressions that form a well formed formula. The complete grammar for this system is shown in Figure 12 in Backus-Naur Form (BNF). In BNF there is a set of production rules that are used to define a grammar. In these rules any term on the left can be replaced by one of the symbols or equations on the right hand side of the “:=” statement. On the right hand side alternatives are separated by “|”. You continue doing replacements until you get to a terminal symbol which is a symbol that does not exist on the left hand side. BNF allows statements to be evaluated recursively by using the same symbol on the right and left side of the equation.

This grammar was derived by taking a subset of the properties of first-order logic and defining how the grammar would be recognized in ADEPT. Each formula under the formula symbol is considered a clause and will be parsed into a statement for insertion into the SPEC# program.

```plaintext
formula  ::= ( formula )
|  expr equalop null
|  expr logicop expr
|  pred
|  ! formula
|  formula && formula
|  formula || formula
|  formula <=> formula
|  formula ==> formula
|  quant var_expr ( formula )
|  quant var_expr ( range formula )

range  ::= expr logicop expr logicop expr
```
Formulas are built up using the symbols and using the acceptable grammar to build longer formulas. Parenthesis may be added around any expression to make the order of processing
more apparent. Parentheses are required when using quantifiers as in \(\text{Ai} \ (0 \leq i < \text{SortArray}.\text{Length} - 1 \ (\text{SortArray}[i] <= \text{SortArray}[i+1]))\). The processor uses standard left order processing in the precedence order indicated in Figure 13 but the identifiers are not checked for type.

Multiplication Division
Addition Subtraction
\(<\>\=\neq\leq\geq\)
EXISTS
FORALL
NOT
AND
OR
IMPLICATION
IFONLYIF

Figure 13. Precedence order

For example, the previous first-order logic formula given in Figure 11 is made up of three main formulas that are connected by && and a fourth formula that is found within the third formula. How this is broken down is shown in Figure 14. The first formula is an expression being compared to another expression. The second one is a predicate which is an identifier followed by an expression list. The third one is a quantification with a range, with forall being the quantification and \(i\) being the range variable. This particular formula, by definition, contains a formula. In this particular case, the sub formula is a comparison of the form expr logicop expr.

1) \(\text{sortArray}.\text{Length} = \text{old}(\text{sortArray}.\text{Length})\)  \hspace{1cm} expr equalop expr
2) \(\text{Permutation}(\text{old}(\text{sortArray}))\) \hspace{1cm} pred
3) \(\text{Ai} \ (0 \leq i < \text{sortArray}.\text{Length} - 1 \ (\text{sortArray}[i] <= \text{sortArray}[i+1]));\) \hspace{1cm} quant var ( range (formula ))
4) \(\text{sortArray}[i] <= \text{sortArray}[i+1]);\) \hspace{1cm} expr logicop expr

Figure 14. Sort specification broken by BNF grammar

The basis of the first-order logic is to be able to establish the truth of the formula. We want to be able to prove that the program satisfies all the conditions set forth in the specifications. As such, each of the individual WFF in the whole specification needs to be able to evaluate to true or false. Predicate logic naturally fits this condition and so predicates can be used and are recognized in the identifier (expression_list) notation. A function name would also be legal according to the parsing notation, and indeed are legal as long as they can be used to evaluate the truth of the statement. Our implementation of the parser does not perform checking on the return type, so it has no way of knowing if a function returns a Boolean or a value. We considered using a special notation to differentiate functions from predicates, such as fn, but that notation seemed arbitrary and still did not solve the function evaluation problem without implementing type checking.
The specifications that are entered will be broken into clauses by the ADEPT system. These clauses will be used to match variables and classes in the design in order to insert the specifications into the code. To break apart the specifications, the utilities Flex and Bison are used. Flex takes an input file and breaks it into a series of tokens using regular expressions. It does this by using a rule set that is based on the symbols in Table 2. The tokens are then passed to the Bison program that will parse them into a tree based on the grammar in Figure 12. The actual files for these can be found in Section 8.3. In testing the program, it was found that ( ) are extremely important in the formula. The parsing functions have a limited amount of look ahead features and are breaking the specification into manageable pieces, so very complex formulas without parenthesis may be parsed incorrectly.

Once the specification has been parsed, the parse tree is then traversed to extract the various clauses of the specification and these are translated into conditions. If the tokens can be parsed into a valid tree structure then the specification is valid, otherwise an error will be given. These errors can sometimes be misleading depending on where the error was found. The conditions that were found up to the point where the error is found are still output, which helps in finding the error in the specification formula.

Parsing creates possible clauses for insertion into the design. This provides a lengthy list of possible conditions which can have some repetitions because of the detailed parsing. The user must select the set of needed clauses for their design and can delete any unnecessary ones. They then edit the initial raw set of clauses before inserting them in the design code.

Figure 15. Uploading system specifications
To continue with our sorting example, the user has uploaded their specification and ADEPT parsed the specifications for valid grammar, as can be seen in Figure 15. The possible clauses are then presented in their entirety. The user has deleted clauses that are not needed and Figure 16 shows the user deleting the final line. The last line is usually the entire formula, which will be needed in some programming designs and not others. This process narrows the list down into a more realistic set. Once the specification has been entered for the system the system, the system specification can be merged with the design that was created from the design pattern.

Figure 16. Clarifying specifications

3.5 Specification and Design Merging

The system specification is inherently supposed to be an abstract representation of the system gives requirements about functionality. The system design however needs to create a more detailed version of that specification that can be later implemented. In our system, we create a system design by using descriptive modeling techniques to describe the class and relationship structure of the system. We then match the system specifications with elements in the design and merge them together in order to be able to proceed to the verification state.

3.5.1 Creating Possible Requirements From model and System Specifications

The next step is to match the parsed clauses with the system design. This is done in the Spec Edit Tab as shown in Figure 17. When the user selects the Match option, ADEPT will attempt
to match the clauses from the system specification with the design that has been entered. The identifierNames in each clause are searched for class names, method names and instance variables. If any of the identifiers in the clause, match any of the information from the class, the clause is put in that classes’ possible requirements list.

Figure 17. Match specifications

All the possible matched clauses for each class are added to the class objects. These conditions have not yet been identified as invariant, pre or postconditions. In fact, there are multiple possibilities for each clause. A clause might be needed as an ensures condition and as an invariant. To make insertion of the clauses into the model easier, each class’s clauses need to be edited to indicate whether they are an invariant, pre and/or postcondition. These assertions are defined in the following paragraphs. How the clauses should be marked depends on what is being proved by that particular statement. Whether it is a condition that must be true before a method runs or after it finishes executing, or whether the condition is being placed on the state of the object, determines its type. A corresponding condition is added in the merged code for each condition or invariant that is indicated for each clause. If no conditions are indicated, then that clause will not be added for the class.

When the design is merged with the specifications, the preconditions and postconditions are encoded using SPEC# requires and ensures statements. These statements must be side effect free statements that evaluate to true or false. These statements can call methods, but those methods must also be side effect free which are denoted as [Pure] in SPEC#. The requires statement is a precondition that puts the responsibility on the caller of the method to make sure that all the conditions will be true before the method is called. The ensures statement is a postcondition that makes sure the method itself is satisfying all the requirements. An exception
will be thrown if any of the \textit{ensures} statements are not true when the method has finished executing.

Class \textit{invariants} are conditions that are placed on class instance variables to be sure that the class satisfies the stated conditions. These invariants must be true whenever the class is in a steady state. These fields must be fields that have an owner. This is ensuring that when one class is using another class, that the other class is also meeting its specifications. In order to be able to change these attributes there are some other aspects of SPEC# that are very important that will be discussed in more detail in Section 3.6.3.

To show how a clause is added, look at the example in Figure 18. This example illustrates the editing of \texttt{SortStrategy}. The specification requires that each time a new array is sorted, the length of the new array needs to be the same as the length of the unsorted array. That specification is captured via an \textit{ensure} condition. Next, the specification requires the sorted array to have all the same elements as the old array, so the clause that contains the \texttt{Permutation} is marked with the \textit{ensure} condition. Finally, the clause that contains the forall is used to check that each element of the sorted array is smaller than or equal to the next element. This clause is also marked with the \textit{ensure} condition to validate this requirement. To help with its verification, it is necessary to mark it as a loop \textit{invariant} condition. Loop \textit{invariants} are used to aid in checking clauses that contain quantification. They help to check that the clause is true for each iteration of the loop. In this example, the loop \textit{invariant} is written in a very similar fashion to other clauses containing ensures conditions.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure18.png}
\caption{Editing of matched specifications}
\end{figure}

3.5.2 Modifying Specifications

After marking all the clauses for each class, the specifications are finally ready for the final merge. The result is the initial SPEC# code based upon the user’s specification. However, it is possible that not all clauses were properly matched with their respective classes. The user will need to edit this initial SPEC# code to manually associate these clauses with the appropriate class. ADEPT places any clause, that cannot be programatically matched, at the bottom of the screen. The user can cut and paste these clauses into the code and manually make them into ensure, require or invariant conditions. Other edits can be made to move conditions from one method to another.

![Figure 19. Merged specifications](image)

If there is anything that has been missed, the user can make modifications to any of the conditions in the code, but changes to the code in the DesignSpec tab is not reflected in the objects from the other tabs. This code information is saved in an internal file so that it can be edited at a later date but at the current point in time you cannot go back and forth between the Spec Edit and the DesignSpec tabs. If you go back to the Spec Edit tab, or one of the other tabs, and do the merge again, the original merge will be lost.
3.6 SPEC# Program

The final SPEC# program can be created and the whole functionality of the SPEC# system can be used once the user has completed the finalization of the specifications in the merged text. The final step in the process entails creating a project in Visual Studio and including the code that has been created in ADEPT so that the full functionality of SPEC# can be realized. SPEC# includes static and dynamic verification to check for design problems in the model. Static verification is similar to that done by the compiler; it checks modular properties, possible null conditions and logical properties. Warnings and errors that are similar to compiler errors will be given during the static check. Runtime checks occur during execution for more temporal properties, overflow and full functional correctness.

3.6.1 Creating SPEC# Source From Classes and Specifications

Each class that has been created by ADEPT contains all the information about the instance variables, attributes and operations that the user defined, in addition to the conditions that were added as part of the specification merge step. When the option is chosen to Create SPEC#, the classes are translated into SPEC# source code and saved in a file with the .ssc extension.
3.6.2 Creating the SPEC# Project

This source code needs to be associated with a SPEC# project. This is a manual process. The user launches Visual Studio and creates a new project that is a SPEC# project. The .ssc source file is then added manually to the new SPEC# project via the Add Existing Item option in Visual Studio. The user can at this point continue with the next step. Figure 21 shows the SortTest project that was created in Visual Studio. SortTest, which was created from ADEPT, was then added to the project.

![Figure 21. SPEC# project with sort source code added](image)

3.6.3 Finalizing the SPEC# Program

Once the project has been created in Visual Studio, the user will need to continue to modify the SPEC# program by adding the implementation code. The conditions that have been added are just statements to help validate that the modeling code executes as expected. It is best to write the implementation in the SPEC# environment where compiler feedback can help the user determine if their conditions are being met. SPEC# will indicate problem areas of a program by underlining the code. In Figure 21 the Permutation method reference is underlined because an ensures clause can only call methods that do not have side effects. To handle this, the Permutation method needs to be marked as [Pure]. Once these changes are made the project will build. The code itself will still need to be filled in as shown in Figure 22.
Figure 22. SPEC# code that is ready for implementation code

There are some other important key aspects to SPEC# that are important to get the validation to work correctly. To use SPEC# effectively, go through the SPEC# tutorial found in the documentation section of the SPEC# homepage (Rustan, Leino, & Monahan, 2008). It more completely discusses each of these aspects. An overview is given in this section.

There are a number of quantifications that can be performed on arrays as has been mentioned previously. These capture quite a bit of information and assertions about arrays. These can be inserted in the program in a number of different ways depending on the need. They can be inserted without a range and the range of the whole array will be assumed. They can also be given with an inclusive and non inclusive range. The type of insertion will be created from the surrounding information in the specification and inserted in the syntax as shown below.

\[
\text{quantification } \{\text{datatype } i \text{ in (low:high); } E(\text{ArrayName } [i]) \}; \quad // \text{ range excluding high}
\]

\[
\text{quantification } \{\text{datatype } i \text{ in (low..high); } E(\text{ArrayName } [i]) \}; \quad // \text{ inclusive range}
\]

\[
\text{quantification } \{\text{datatype } i \text{ in } \text{ArrayName}; } E(i) \}; \quad // \text{ all of array}
\]

Where:
quantification is: \textit{forall} or \textit{exists} or \textit{count}
datatype will be determined from the array type declared in the program
expression will come from the specification associated with the quantification
for example \textit{forall} \{ \text{int } j \text{ in } (0..k); \ a[j]>0\}
min, max, product, and sum are specified in a similar manner except that instead of having an expression at the end of the phrase they will just have a reference to the array with the subscript. They can also be used as an attribute method depending on context. For example sum{int i in (0:a.Length); a[i]}

There are two statements, assert and assume that should be used in the body of the code to help in the verification of the code. Assert E and Assume E will check the condition E during runtime. If the condition is true, then nothing happens. If it is false, then it will throw an exception. The difference between the two statements is found during static verification. The static verifier attempts to prove that E is true for an Assert statement whereas for an Assume it accepts its truth as fact without static verification. These are shown in Figure 23 along with other keywords that have been covered so far.

- assert E; in-line assertion in a method
- assume E; in-line assertion in a method. Not verified in static verification.
- ensures E; declares postcondition
- requires E; declares precondition
- invariant E; condition that must be true during the steady state
- old (E) value of E before method
- result the result that will be returned from the method.

Figure 23. Some common SPEC# keywords

SPEC# works on a basis of ownership and when objects can be modified. If an object is an aggregate object there are two possibilities for ownership, [REP] and [PEER]. A field declared as [REP] is an object that is used within another object but each has its own owner. It is used for building hierarchical structures. The object that is using the [REP] field is just using a representation of it. If the object is declared as [PEER] however, then it is owned by an object that also owns the object that is declaring it. If you were to think of it as a tree the owner would be the parent field and the peer objects would be the leaves. Peer fields are the default for these objects. Objects need to keep a state of consistency. For this to happen, their owner needs be mutable and their state needs to be valid. When objects have peer fields, they usually need peer consistency. This means for every peer object, each peer has the same owner as the current object and the peer and the current object are consistent. This makes intuitive sense. If we have a car that is made up of various parts such as seats and wheels, the Car needs to be changeable to be able to add the parts to it and each of the parts needs to be valid when they are added.

Objects are not mutable by default, which helps preserve the above qualities. To modify an object that has an invariant at the object level, the object has to be exposed in the method where it is changed. The modifies condition is used to declare exactly what is going to be modified. The expose condition then encloses the block of code that does the actual modification. The object’s invariant is then checked at the end of the expose condition. A peer field may not need to be exposed if the object is peer consistent. A rep field will use the expose to get access to the information. Classes that have subclasses can also have an [Additive Expose] if they are going to
expose more information than was already exposed in the enclosing class. If the object’s preconditions are met, then it can call objects to which it has reference to, as long as it is not in an expose block. The syntax of the expose and modifies conditions are shown in Figure 24.

- `expose (this);` exposes current object
- `modifies w;` declares what a method is allowed to modify
- `modifies w` where w is a list of:
  - `p.x` field x of p
  - `p.*` all fields of p
  - `p.**` all fields of all peers of p
  - `this.*` default modifies clause, if this-dot-something is not mentioned in modifies clause
  - `this.0` disables the “this.*” default
  - `a[i]` element i of array a
  - `a[*]` all elements of array a

**Figure 24. Expose and modifies conditions**

In Figure 25 there is an example of the expose and modify conditions. The transcript has been declared as a Rep field of the class and always needs to remain consistent. In order to be able to change it, the `UpdateTranscript` method has to declare that it will modify t. Then inside the actual code that does the modification an `expose t` lets t be temporarily mutable and therefore be inconsistent. At the end of the expose block all the invariants must hold and the object will be consistent again.

```java
Class Student : Person {
    [Rep] Transcript! t;

    public void UpdateTranscript(String major) modifies t;
    {
        expose (t);
        t.NewMajor(major);
        ...
    }
}
```

**Figure 25. Example of modify and expose**

The Otherwise statement indicates what the program should do when a program throws an exception as part of a condition. Exceptions can be either checked or unchecked. If they are checked then they need to be declared in the method declarations and the caller would be expected to catch the exception. An unchecked exception will be propagated up through the
program until it is caught somewhere or the program terminates. Examples of these two exception types are shown in Figure 26.

**Checked Exception**

```java
public void OpenFile(String name)
throws FileNotFoundException;
```

**Unchecked Exception**

```java
public void ArrayListAdd (int index, object o)
ensures 0 <= index && index < ArrayList.Length
otherwise throws IndexOutOfRangeException;
```

**Figure 26. Exception handling**

Once all of the conditions have been modified and the regular programming statements for the model have been added, the project can be compiled. The SPEC# compiler will give C# syntax errors and SPEC# errors for items that do not pass the static verification model. Run time errors will be thrown for conditions that break the logic during actual execution.

### 4. Example

In order to test the system more fully, another example from a different pattern will be shown. This example is a simple chat room that was presented as an example in (Campbell & Sobel, 2008). The chat room is a console based system that allows users to join a room and send and receive messages with the people in the room. In this particular situation, there is group of people who want to be able to communicate with each other without keeping track of everyone who is on the system. The mediator pattern is a good design for this problem because the mediator can keep track of all the participants and handle the messages the participants send.

In this chat room, a person must be logged on to send or receive messages. This system allows sending a message to a particular group or to everyone in the chat room. To check that a person can receive a message the following conditions must be met:

1) The person must be logged on
   and either
2) A member of the chat room either sent a message to everyone in the chat room or
3) A member of the participants group sent a message to everyone in their group

In this example, to be able to send a message the sender needs to exist and they need to have a name, so we are going to use the non-null feature. In Figure 27, we check the nonnull check box for the name of the participant to insure that whenever a participant is created they must have a
name. They will only have a room when they actually login so it is acceptable for that instance variable to have a null value. In Figure 28, the non-null is being added to the parameters as well. In the Method box, the Login method has already been added, which takes a Participant object. This object cannot be null for the login method to work. We will not have to add ensures clauses to our project to ensure this, they are inherent to the language.

![Figure 27. Adding Non-Null instance variable](image1)

![Figure 28. Adding parameters with Non-Null](image2)
To be able to specify this in first-order logic we have the following formula

\[
\text{name.numMsgRec} = \text{old(name.numMsgRec)} + 1 \implies \\
\exists i (\text{Participants}[i] = \text{name}) \land \\
\exists j (\text{Participants}[j] = \text{from}) \land \\
((\text{NumBroadcast} = \text{old(NumBroadcast)} + \text{from.numInGroup}) \land \\
\text{ForwardGroup.result} = \text{from} \lor (\text{NumBroadcast} = \text{old(NumBroadcast)} + \text{participants.count} \land \\
\text{ForwardAll.result} = \text{from}));
\]

These conditions resulted in the parsed clauses shown in Figure 29. Seeing that this specification is more detailed, the number of possible clauses is greater than in the previous example. Though there is some duplication because of the breaking down of the individual formulas, many of them will be needed in their various forms. The clauses that contain \text{NumBroadcast}, will be needed in the methods that send messages to make sure that they send the correct number of messages. The clauses that contain \text{result}, specify that the message came from the sender. The final clause is needed in the Participant’s Receive method to check the entire condition as originally specified.

A number of the clauses match up for the ForwardGroup and ForwardAll methods in the AbstractChatroom class. For the ForwardAll method we have chosen the ones that specifically deal with that particular method as shown in Figure 30. The ForwardAll().result = from will ensure that the message always goes out from the person who actually sent the message. The
NumBroadcast = old(NumBroadcast) + count{ Participants i in participants; i} will make sure that the message is sent to everyone is logged in.

Figure 30. Specifications for the ForwardAll method in the AbstractChatroom class

After all the editing has been done to the classes and specifications, the merged program is created as shown in Figure 31. All of the clauses were able to be matched so there are no extras clauses to insert at the bottom. The SPEC# program was created from this code and added to a project resulting in the code in Figure 32. At this point the user will need to make some changes. The result keyword needs to be by itself so the method name needs to be removed from all occurrences of .result. Other programmatic changes will have to be made depending on how the Participants class is implemented. After that, the basic design can be checked statically and more code added to check it dynamically. The full code is shown in Section 8.5.
Figure 31. Merged chat room program design with specifications

Figure 32. Raw SPEC# code created from ADEPT
5. Future Work

Dr. Sobel has defined a specification notation that captures the externally visible behavior of a system using interaction sequences where elements of the sequence represent method invocations. Sobel has used this notation in the classroom as part of an educational experiment that demonstrated students who have been taught formal analysis skills had increased complex problem solving skills (Sobel, 1999). Past usage of this specification notation in both the classroom and on industrial-strength software has shown that it is relatively easy to use due to its operational nature and due to its prerequisite knowledge of only first-order logic (Sobel, 1999). Incorporated within the specification model is a set of rules for refining specifications into design pattern specifications. These design pattern specification “templates” would assist the user in refining their system specification into their chosen design structure.

6. Conclusion

In this work we have created a tool, ADEPT, that aids in the design of provably correct systems. This tool helps the user make sure that their system specification is grammatically correct. It then allows them to choose a design pattern that has classes and relationships setup and lets them embellish those to make the design fit their own particular needs. ADEPT will then show which classes and methods the original specification has been parsed into and let the user specify what kind of condition each clause is. Actual code written in the specification language is the created from the merged specifications and design. The user can then use the specification language’s IDE to further refine their model and check to make sure their specifications are mathematically correct.

It has been found in testing that though first-order logic is easier than some of the higher order logics it is still not “easy” and great care must be taken in creating the original specification. Also, ensuring that the model names match the specification names is very important or the matching section will be very ineffective.
7. References


8. Appendix

8.1 Installation Manual

The following steps are needed to set up the program

- Install Flex & Bison to C:\GnuWin32 (could be installed to a different path but path can NOT have any spaces in it. Same corresponding path would need to be used in all following instructions.)
- Put C:\GnuWin32\bin\ in System Path environment variable
- Have .NET 2008 installed on computer
- Install SPEC#
- Copy ADEPTParser file to C:\
8.2 Class Diagram of the System of the Design Objects
8.3 Flex & Bison input Grammar

Flex input Grammar

```c
%{
#include "parseradept.tab.h"
%

underscore []
lparen [()]
rparen [)]
lbracket [[]]
rbracket [[]]
and [&][&]
or [][[]]
comma [,]
dot [,]
not [!]
implication [=-][=-][>]
ifonlyif [><][=][=][>]
plusop [+]
minusop [-]
divop [/]
multop [*]
modop [%]
lt [<]
gt [>]
eq [=]
ne [!=]
le [<][=]
ge [><][=]
constant [0-9]+|[0-9]+|[0-9]+.[0-9]+|[0-9]+constant
var [B-DF-Za-z][A-Za-z0-9_]*

%\%
[\t\n]+ /* do nothing */
--.*\n /* comment */
[A] {return (FORALL);}
[E] {return (EXISTS);}
{lparen} {return (LPAREN);}
{rparen} {return (RPAREN);}
{lbracket} {return (LBRACKET);}
{rbracket} {return (RBRACKET);}
{comma} {return (COMMA);}
{dot} {return (DOT);}
{and} {return (AND);}
{or} {return (OR);}
{not} {return (NOT);}
{substitution} {return (SUBSTITUTION);}
{implication} {return (IMPLICATION);}
```
Bison input grammar

```c
#include <stdio.h>
#include <ctype.h>
#include <malloc.h>
#include "lex.yy.c"

#define YYSTYPE YYSTYPE

void finish();
char *write_one(char *);
char *write_four(char *, char *, char *, char *);
char *copy_four(char *, char *, char *, char *);
char *write_five(char *, char *, char *, char *, char *);
char *write_range(char *, char *, char *, char *, char *);
char *copy_var(char *);
char *write_exists(char *, char *, char *);
char *write_existsLong(char *, char *, char *, char *);
char *write_expr(char *, char *, char *);
char *write_arrayDefault(char *, char *);
```

```c
{ifonlyif} {return (IFONLYIF);} 
{plusop} {return (PLUSOP);} 
{minusop} {return (MINUSOP);} 
{divop} {return (DIVOP);} 
{multop} {return (MULTOP);} 
{modop} {return (MODOP);} 
{lt} {return (LT);} 
{gt} {return (GT);} 
{eq} {return (EQ);} 
{ne} {return (NE);} 
{le} {return (LE);} 
{ge} {return (GE);} 
[cC][oO][uU][nN][tT] {return (COUNT);} 
[mM][iI][nN] {return (MIN);} 
[mM][aA][xX] {return (MAX);} 
[pP][rR][oO][dD][uU][cC][tT] {return (PRODUCT);} 
[sS][uU][mM] {return (SUM);} 
[oO][lL][dD] {return (OLD);} 
[nN][uU][lL] {return (NULLX);} 
{constant} {return (CONSTANT);} 
{var} {return (VAR);} 
[^,] {printf("SYNTAX ERROR: '%s' IS ILLEGAL\n", yytext);} 
```
char *write_arrayRange(char *, char *, char *);
char *write_arrayCount(char *, char *, char *, char *, char *);
char *write_arrayCountDefault(char *, char *, char *, char *, char *, char *);
char *gen_infix(char *, char *, char *);
char *list_add(char *, char *); char vartype;
char temp[32];
FILE *fo;
%

%union {
    char sval1[1000];
    char sval2[100];
    char sval3[32];
}
%token LPAREN RPAREN LBRACKET RBRACKET COMMA DOT SUBSTITUTION NULLX
CONSTANT VAR
%token COUNT SUM PRODUCT MIN MAX OLD
%token END 0 "End of file"
%left IFONLYIF
%left IMPLICATION
%left OR
%left AND
%left FORALL
%left EXISTS
%left LT GT EQ NE LE GE
%left PLUSOP MINUSOP
%left MULTOP DIVOP MODOP
%nonassoc NOT

%type <sval1> formula
%type <sval1> expr_list
%type <sval1> range
%type <sval1> expr
%type <sval1> term
%type <sval1> var_expr
%type <sval1> array_expr
%type <sval1> count_expr
%type <sval2> var
%type <sval2> pred
%type <sval3> array_val
%type <sval3> constant
%type <sval3> quant
%type <sval3> dot
%type <sval3> old
%type <sval3> nullx
%type <sval3> count
%type <sval3> logicop
%type <sval3> equalop
%type <sval3> realop
%type <sval3> lparen
%type <sval3>rparen
%type <sval3>lbracket
%type <sval3>rbracket

%start system_goal
%%
valid_Form : formula END
    | error {printf("***** ; EXPECTED TO END FORMULA, BUT FOUND "%s"\n", yytext);} ; 

formula : lparen formula rparen {strcpy($$, write_expr ($1, $2, $3));}
    | expr equalop NULLX     {strcpy($$, write_expr ($1,$2,"NULL"));}
    | expr logicop expr     {strcpy($$, write_expr ($1,$2,$3)));}
    | pred                   {strcpy($$, write_one ($1));}
    | NOT formula           {strcpy($$, write_expr ("!", $2,""));}
    | formula AND formula   {strcpy($$, write_expr ($1,"&","",$3));}
    | formula OR formula    {strcpy($$, write_expr ($1,"||","",$3));}
    | formula IFONLYIF formula   {strcpy($$, write_expr ($1,"<==","",$3));}
    | formula IMPLICATION formula   {strcpy($$, write_expr ($1,"==>","",$3));}
    | quant var_expr lparen formula rparen   {strcpy($$, write_exists($1, $2,$4));}
    | quant var_expr lparen range formula rparen   {strcpy($$, write_existsLong($1, $2,$4, $5));}

range : expr logicop expr logicop expr {strcpy($$, write_range ($1,$2,$3,$4,$5))}
    | error {printf("***** range EXPECTED, BUT FOUND "%s"\n", yytext);} ;

pred : var_expr lparen expr_list rparen {strcpy($$, copy_four ($1,$2,$3,$4))};
  | var_expr lparen rparen  {strcpy($$, gen_infix($1,$2,$3));}

expr_list: expr   {strcpy ($$, $1);}
  | expr COMMA expr_list  {strcpy ($$, list_add($1, $3)));}

expr : term   {strcpy ($$, $1);}
  | expr realop expr   {strcpy($$, gen_infix($1,$2,$3)));}
  | lparen expr rparen      {strcpy($$, gen_infix($1,$2,$3)));}

term : var_expr   {strcpy ($$, $1);}
  | constant   {strcpy ($$, $1);}
  | error {printf("***** identifier or constant EXPECTED, BUT FOUND "%s"\n", yytext);} ;

var_expr: var    {strcpy ($$, $1);}
  | array_expr   {strcpy ($$, $1);}
  | count_expr   {strcpy ($$, $1);}
  | var_expr dot var    {strcpy ($$, gen_infix($1,"","",$3)));}
  | old lparen var_expr rparen   {strcpy($$, copy_four ($1, $2, $3 ,$4)));}
  | var lbracket expr rbracket   {strcpy($$, copy_four ($1, ", $3 ,"]")));}
\begin{verbatim}
rbracket : RBRAKCET {strcpy ($$, "]");}
    | error {printf("***** \] EXPECTED, BUT FOUND \%s\n", yytext);} ;

lbracket : LBRACKET {strcpy ($$, ";");}
    | error {printf("***** \[ EXPECTED, BUT FOUND \%s\n", yytext);} ;
dot : DOT {strcpy ($$, ".");}
;
old : OLD {strcpy ($$, "old");}
;
count : COUNT {strcpy ($$, "count");}
;
nullx : NULLX {strcpy ($$, "null");}
;
system_goal : valid_Form {finish() ;}
;
%%
void finish()
{
    printf("Halt\n");
}
/* Surround all identifiers with ?` for easy detection in program */
char *copy_var(char val[])
{
    static char tempname[50];
    sprintf(tempname,"?%s`", val);
    return tempname;
}

// write a stream of four values to the output and save it
char *write_four(char val1[], char val2[], char val3[], char val4[])
{
    static char tempname[200];
    sprintf(tempname, "%s%s%s%s", val1, val2, val3, val4);
    fprintf(fo, "%s%s%s%s\n", val1, val2, val3, val4);
    return tempname;
}

// write a stream of four values to the output and save it
char *copy_four(char val1[], char val2[], char val3[], char val4[])
{
    static char tempname[200];
    sprintf(tempname, "%s%s%s%s", val1, val2, val3, val4);
    return tempname;
}
\end{verbatim}
return tempname;
}

// this writes the single expression to the output file and saves it
char *write_one(char form_desc[])
{
  static char tempname[200];
  sprintf(tempname, "%s", form_desc);
  fprintf(fo,"%s\n", form_desc);
  return tempname;
}

// this writes the 3 part expression to the output file and saves it
char *write_expr(char expr_Type[], char id_name[], char form_desc[])
{
  static char tempname[200];
  sprintf(tempname, "%s %s %s", expr_Type, id_name, form_desc);
  fprintf(fo,"%s %s %s\n", expr_Type, id_name, form_desc);
  return tempname;
}

// writes out an existential equation
char *write_exists(char expr_Type[], char id_name[], char form_desc[])
{
  static char tempname[200];
  sprintf(tempname, "%s {datatype %s in (0:.Length); %s}\n", expr_Type, id_name, form_desc);
  fprintf(fo,"%s {datatype %s in (0:.Length); %s}\n", expr_Type, id_name, form_desc);
  return tempname;
}

// writes out an existential equation with a range
char *write_existsLong(char expr_Type[], char id_name[], char range_val[], char form_desc[])
{
  static char tempname[200];
  sprintf(tempname, "%s {datatype %s in %s; %s}\n", expr_Type, id_name, range_val, form_desc);
  fprintf(fo,"%s {datatype %s in %s; %s}\n", expr_Type, id_name, range_val, form_desc);
  return tempname;
}

// writes out an array math equation
char *write_arrayDefault(char array_name[], char expr_Type[])
{
  static char tempname[200];
  sprintf(tempname, "%s {datatype %s in %s; %s}\n", expr_Type, array_name);
  return tempname;
}
// writes out an array math equation with range

char *write_arrayRange(char array_name[], char expr_Type[], char range_val[]) {
    static char tempname[200];

    sprintf(tempname, "%s {datatype i in %s; %s[i]}", expr_Type, range_val, array_name);
    return tempname;
}

// writes out an array math equation with range and qualification

char *write_arrayRangeExpr(char array_name[], char expr_Type[], char range_val[], char expr[]) {
    static char tempname[200];

    sprintf(tempname, "%s {datatype i in %s, %s; %s[i]}", expr_Type, range_val, expr, array_name);
    return tempname;
}

// writes out an array count equation with range

char *write_arrayCount(char expr_Type[], char range_val[], char expr1[], char op[], char expr2[]) {
    static char tempname[200];

    sprintf(tempname, "%s {datatype i in %s; %s%s%s}", expr_Type, range_val, expr1, op, expr2);
    return tempname;
}

// writes out an array count equation with default range

char *write_arrayCountDefault(char expr_Type[], char array_name[], char expr1[], char op[], char expr2[]) {
    static char tempname[200];

    sprintf(tempname, "%s {datatype i in (0:%s.Length); %s%s%s}", expr_Type, array_name, expr1, op, expr2);
    return tempname;
}

// this breaks a range up into its 2 component parts

char *write_range(char val1[], char comp1[], char val2[], char comp2[], char val3[]) {
    static char tempname[200];

    if (strcmp(comp2,"<")==0) {
        sprintf(tempname, "(%s:%s)", val1, val3);
        fprintf(fo,"%s %s %s & & %s %s %s; \n", val1, comp1, val2, comp2, val2, comp2, val3);
    } else if (strcmp(comp2,"<=")==0)
```c
if ( argc != 3 )
    printf("Usage: spec inputfile outfile
");
else
{
    FILE *fopen(), *fp, *fp2 ;

    fp = fopen( ++argv, "r" );
    if( fp == NULL )
    {
        printf("spec: Couldn't open %s\n",argv);
        exit( 1 );
    }

    fo = fopen( ++argv, "w");
    if( fo == NULL )
    {
        printf("spec: Couldn't open %s\n",argv);
        exit( 1 );
    }
```
yyrestart(fp);
yyparse();

fclose(fp);
fclose(fo);
### 8.4 User Manual

ADEPT starts on the screen shown in Figure 33. This start page outlines the steps in the design process. You can create a new design or open one that has been previously started. Designs that have been saved will have the .adpt extension.

![Start page of ADEPT](image)

**Figure 33. Start page of ADEPT**

**Step 1. Choosing a pattern:** The first step in the process is to choose a design pattern. The **Pattern** tab displays the list of behavioral patterns that are implemented in this system. To see the class diagram for any particular pattern, click on the pattern name and the diagram will show in the lower right hand corner. Figure 35 shows the Strategy pattern’s diagram. To choose a pattern click on the desired pattern and then click the **Start revising selected pattern** button. This will create a copy of all the classes, variables, methods and relationships that are part of that pattern to create the starting design. The **Pattern Revision** tab will automatically get focus at the completion of the creation of the basic design.
Figure 34. Step 1 Choosing a pattern

Figure 35. Strategy pattern

60
Step 2. Revising the pattern: The design pattern provides the basic template from which to start a program, but the design can be modified to meet the particular specifications of the intended program. Each class can be edited to change the class name, add instance variables and add methods. New classes can be added and deleted or an existing class can be copied. These can all be found on the Pattern Revision tab as shown in Figure 36. (Note that the original template classes cannot be deleted to preserve the pattern integrity.) Relationships can also be added, changed and deleted but again the original relationships cannot be changed or deleted. Each of these processes will be outlined in the following diagrams.

Figure 36. Step 2 Revising the pattern
**Editing a class**: To edit a class click on the desired class and click the Edit Class button as shown in Figure 36. Figure 37 shows the class that is not edited yet. This class is set up to be an abstract class and has an abstract `AlgorithmInterface` defined. To change the class name click on **Customize Class Name** and a text box that will allow a new name to be entered will appear as shown in Figure 38. The name should follow standard C# naming conventions. In the list boxes the original template name will appear with the newly defined name and separated by a dash. If a new name is not given the original template name will be used.

![Figure 37. Editing a class](image)

**Instance Variables**: To add a new variable click on the **Add New Variable** button. This will bring up the **Add Variable** box at the bottom of the screen. The variable name follows standard C# naming conventions and cannot be the same as any other variables or methods in the class.

![Figure 38. Changing the class name](image)
Accessibility is a required field that indicates whether the variable is public, private or protected. A form of Data type is required; either a Primitive can be chosen from the list or one of the classes. The other fields are optional. Non-null adds checks to make sure that an object is initialized before it is used, this option can only be chosen with classes and objects such as strings. Arrays can be chosen with either primitives or classes. The Create Accessor option will create Accessor get and set methods when the code is created in a later step.

To Edit an existing variable click on the variable in the Variable list box and click the Edit Selected Variable button. This will bring up the editing as described above.

To Delete an existing variable click on the variable in the Variable list box and click the Delete Selected Variable button.

Figure 39. Adding an instance variable
Methods: To add a new method click on the Add New Method button as shown in Figure 40. This will bring up the Method box. The required fields are Name, Accessibility, and a Return Data Type. Name follows standard C# naming conventions and cannot be the same as other methods or variables in the class. Accessibility is a required field that indicates whether the variable is public, private or protected. A form of Return Data Type is required, a void, constructor, primitive or class can be chosen. The data type can also have the additional declarations of Non-Null or Array. The Category field is an optional field that tells if the method is an override, abstract or virtual class. The Pure option can be chosen if the class does not have any side effects and is going to be called from an ensures condition. Ensures clauses can only call Pure methods. If the class has parameters they can be entered now by clicking on the Add Parameter button. Existing parameters can be edited or deleted by clicking on the desired parameter in the Parameters list box, and then clicking on Edit Parameter or Delete Parameter respectively. When all the information about the Method has been entered (including any optional parameters) click on Save. This will save all the Method information and close the Method box. Cancel will prevent any of the method changes from taking place and close the Method box.

Figure 40. Adding a method to a class
**Parameters:** The parameter edit box is very similar to the **Instance Variable** box. **Name**, **Accessibility** and **Data type** are required as is **Passing mode**. The **Passing mode** will default to *In* as most parameters are input parameters that are passed by value. **Ref**, **Out** and **Return** can also be specified. **Non-Null** and **Array** are optional fields in this area also. When **Save** is clicked the parameter will be added to the **Parameter** list box, the **Parameter** area will close and the **Method** area will become enabled again. **Cancel** will prevent additions or changes from saving, close the **Parameter** area, and enable the **Method** area.

![Figure 41. Adding a parameter to a method](image-url)
Figure 42 shows the design after several of the classes have been renamed. To copy a class that has a similar implementation, click on the class in the Pattern Revision tab and click on the Copy Class Button. A popup box will allow the entering of the class name for the copied class, shown in Figure 43. This will copy all the variables, methods and relationships of the original class. This is a good method for implementing Concrete Classes that need multiple similar class structures. Relationships for the pattern templates are stored in the basic design but if any classes have been added they need to have relationships set up using the Add Relationship button.
**Customizing Relationships**: New relationships can be added by choosing the two classes in **Class** and **Related Class**, and the **Type of Relationship**. If a relationship would have an arrow in a UML diagram then the class that would be at the arrow head, should be in the **Related Class** box. In Figure 44, ConcreteStrategyA – InsertionSort inherits from the Strategy – SortStrategy class. InsertionSort points to SortStrategy, so InsertionSort is in the **Class** column and SortStrategy is in the **Related Class** column. To delete a relationship double click on the row containing the relationship and click on **Delete Selected Row**. To edit a relationship, double click on the row containing the relationship and click on **Edit Selected Row**. The information for the row to be edited will be placed in the top editing portion of the screen. Change any of the 3 fields and click **Save** to keep the changes or **Cancel** to keep the relationship in its original form. An example of editing is shown in Figure 45. NOTE: Only relationships that have been added by the user can be edited or deleted.

![Figure 44. Modifying relationships](image)

![Figure 45. Adding a new relationship](image)
**Step 3. System Specifications**: The design can be printed, to help in the subsequent steps. The printed version shows the original UML design, and the new design listed out by class, including instance variables and methods. This is found under the file menu.

The system specifications should be entered into a text file with the extension .txt which will be uploaded using the **Upload System Specification** button. The specifications are written in first order logic using the grammar specified in Table 2 and Figure 12. ADEPT will take the file and parse the formulas into all valid combination of sub formulas that are all well formed formulas. In this example, the following formula was uploaded from the text file.

\[
\text{sortArray.Length} = \text{old(sortArray.Length)} \land \\
\text{Permutation(old(sortArray))) \land \\
\text{Ai (0} \leq i < \text{sortArray.Length-1} (\text{sortArray}\_i \leq \text{sortArray}\_i+1))}
\]

This formula is parsed into the clauses shown in Figure 46. Some of these clauses may not be needed because the specification will be covered by one of the other clauses. Deleting clauses that you know will not be needed in the program will reduce the number of actual matches. You can delete one or more lines at a time by using standard selecting techniques (Use Shift to get a block of text or Ctrl to select distinct lines). After selecting the lines click on the **Delete** button.

![Figure 46. Step 3 Uploading the system specifications](image-url)
Step 4. Matching the specification with the design: The next step is to match the parsed segments from the system specification with the system design. This is done by clicking on the Match Specification with Class Information button in the Spec Edit tab. This will enable the Edit Class Specifications and Create Merged Design Specification options as shown in Figure 47. The matching of specifications will look for matches of identifierNames in the parsed clauses with the class, instance variable and method names in each class. After this match has been completed EACH class that is expected to have conditions in it needs to be edited by selecting the class and clicking on the Edit Class Specifications button.
Step 5. Editing the clauses for each Class: When the class comes up clauses that matched the class name or instance variable names will be shown in the bottom of the screen. These clauses have been indicated as possible matches with this class but have not been given a particular condition type. Each clause has the possibility of being an invariant, precondition or postcondition.

Zero or more boxes can be checked on each line. If no boxes are checked that clause will not be added for the class. For each box checked, the corresponding condition will be added in the merged code. For example, Figure 48 shows the clauses for the InsertionSort being edited. The specification requires that each time a new array is sorted, the length of the new array needs to be the same as the length of the unsorted array. That specification is captured via an ensure condition. Next, the specification requires the sorted array to have all the same elements as the old array, so the clause that contains the Permutation is marked with the ensure condition. Finally, the clause that contains the forall is used to check that each element of the sorted array is smaller than the next element. This clause is also marked with the ensure condition to validate this requirement. To help with its verification, it is necessary to mark it as a loop invariant condition. Loop invariants are used to aid in checking clauses that contain quantification. They help to check that the clause is true for each iteration of the loop. The loop invariant is written in a very similar fashion to other clauses containing ensures conditions.

Figure 48. Step 5 Editing the matched specifications
Step 6. Merge the system specification and the design: Click on the Create Merged Design Specification button on the Spec Edit tab and all the code is created and inserted into a rich text box area in the Design Spec tab.

Step 7: Make changes to the Merged Design: The conditions for each class are inserted in red for easy detection in the text box. However, it is possible that not all clauses were properly matched with their respective classes. Also, as in this example, the conditions matched in the abstract class and did not have a method with which to associate the clauses. The final conditions need to be moved to the appropriate method as is shown in Figure 49. The user will need to edit this initial SPEC# code to manually associate these clauses with the appropriate class. ADEPT aids the user in doing this by placing any clause that cannot be programmatically matched at the bottom of the screen. The user can cut and paste these into the code and manually make them into ensure, require or invariant conditions. Other edits can be made to move conditions from one method to another.

Figure 49. Step 7 Editing the merged design
Figure 50. Editing the merged design after conditions have been moved to correct position

**Step 8: Create the SPEC# code:** To create the SPEC# source click on the **Create SPEC# Source** button. This will take the code in the text box and create a .ssc file. This can also be done using menu commands the **Design / Create SPEC#**. This source code needs to be associated with a SPEC# project. This is a manual process. Launch Visual Studio and create a new project that is a SPEC# project. Manually add the .ssc source file to the new SPEC# project via the **Add Existing Item** option in Visual Studio. Figure 51 shows the **SortTest** project that was created in Visual Studio. **SortTest.ssc**, which was created from ADEPT, was then added to the project in Figure 52.
Figure 51. Creating a SPEC# project

Figure 52. The sort source code added into the SPEC# project
Once the project has been created in Visual Studio, other modifications still need to be made to the SPEC# program. The conditions that have been added are just statements to help validate that the modeling code executes as expected. It is best to write the implementation in the SPEC# environment where compiler feedback can help determine if their conditions are being met. SPEC# will indicate problem areas of a program by underlining the code. In Figure 52, the Permutation method reference is underlined because an ensures clause can only call methods that do not have side effects. To handle this, the Permutation method needs to be marked as [Pure]. Once these changes are made the project will build. The code itself will still need to be filled in. This is shown in Figure 53. The final code with the Insertion sort implementation follows.

Figure 53. Source code after program dependent modifications
8.5 Implementation Code

// Implementation Code for the Sorting program with conditions
using System;
using Microsoft.Contracts;

public class Context
{
    private SortStrategy strategy;

    public Context(SortStrategy strategy)
    {
        this.strategy = strategy;
    }

    public void ContextInterface()
    {
        strategy.AlgorithmInterface();
    }
}

public abstract class SortStrategy
{
    private int[]! sortArray;

    public SortStrategy(int [] newArray)
    {
        this.sortArray = newArray;
    }

    public int[] SortArray
    {
        get {return sortArray;}
        set {sortArray = value;}
    }

    // Abstract interface has the conditions that will be upheld in any of the
    // override methods
    public abstract void AlgorithmInterface();
    ensures SortArray.Length == old(SortArray.Length);
    ensures forall {int i in (0:SortArray.Length-1); (SortArray[i] <= SortArray[i+1])};
    ensures Permutation (old(SortArray));
}

// One sort strategy for testing purposed. Keep inserting a value into
// the array and make sure that the ones inserted are sorted
public class InsertionSort : SortStrategy
{
    public InsertionSort(int []! a)
    {

}
public override void AlgorithmInterface()
{
    int numInserted;
    int t;
    if (SortArray.Length > 0)
    {
        numInserted = 1;
        assert forall(int j in (1:numInserted), int i in (0:j); SortArray[i] <= SortArray[j]);
        while (numInserted < SortArray.Length)
            invariant 1 <= numInserted && numInserted <= SortArray.Length;
        invariant forall(int j in (1:numInserted), int i in (0:j); SortArray[i] <= SortArray[j]);
        {
            for (t = numInserted; t > 0 && SortArray[t - 1] > SortArray[t]; t--)
            {
                int temp;
                temp = SortArray[t];
                SortArray[t] = SortArray[t - 1];
                SortArray[t - 1] = temp;
            }
            numInserted++;
        }
    }
    [Pure] // pure is needed to show that method won't change values
    // Permutation checked to make sure the old and new arrays are equivalent
    public bool Permutation(int[] oldArray)
    {
        bool returnVal = true;
        for (int i = 0; i < oldArray.Length && returnVal; i++)
        {
            if (count { int k in (0:SortArray.Length); SortArray[k] == oldArray[i]} ==
                count { int j in (0:oldArray.Length); oldArray[i] == oldArray[j]})
                returnVal = true;
            else
            {
                returnVal = false;
            }
        }
        return returnVal;
    }
}
public class MainApp
{

    static void Main()
    {
        Context context;
        int[] newArray = {8, 2, 9, 3, 7, 22, 2, 99, 3, 1};
        context = new Context(new InsertionSort(newArray));
        context.ContextInterface();
        writeArray(newArray);
        Console.Read();
    }

    static void writeArray(int[] a)
    {
        for(int i=0; i<10; i++)
        {
            Console.Write (a[i] + " ");
        }
    }
}

//Implementation Code for the Chat room program with conditions

using System;
using Microsoft.Contracts;
using System.Collections.Generic;
using System.Text;
using System.Collections;

// Chat room example
// Sherrie Campbell
namespace MediatorChatRoom
{
    class Program
    {
        static void Main(string[] args)
        {
            // Create chat room
            Chatroom room = new Chatroom();

            // Hardcode participants and log them into room
            Participant Ironman = new SuperHero("IronMan");
            Participant Spiderman = new SuperHero("Spiderman");
            Participant SilverSurfer = new SuperHero("Silver Surfer");
            Participant Hulk = new SuperHero("Incredible Hulk");
            Participant TheThing = new FantasticFour("The Thing");
            Participant Reed = new FantasticFour("Reed Richards");
            Participant InvisibleGirl = new FantasticFour("The Invisible Girl");
            Participant HumanTorch = new FantasticFour("The Human Torch");
        }
    }
}
room.Login(Ironman);
room.Login(Spiderman);
room.Login(SilverSurfer);
room.Login(Hulk);
room.Login(TheThing);
room/Login(Reed);
room.Login(InvisibleGirl);
room/Login(HumanTorch);

// Hardcode some chatter
SilverSurfer.SendAll("Hey, anyone doing anything interesting?");
Ironman.SendGroup("We have a new assignment supers.");
Spiderman.SendAll("We super heros need to do work together to beat the invaders.");
HumanTorch.SendGroup("Everything I do seems to go up in flames");
Reed.SendAll("Testing");

// Keep information on the screen
Console.Read();
}
}

// "Mediator"

abstract class AbstractChatroom
{
    public abstract void Login(Participant! participant);
    public abstract string ForwardGroup(string! from, string message);
    public abstract string ForwardAll(string! from, string message);
}

// "ConcreteMediator"

class Chatroom : AbstractChatroom
{
    private Hashtable participants = new Hashtable();
    private int numBroadcast = 0;

    // Login a valid participant as long as they don’t already exist in the hashtable
    public override void Login(Participant! participant)
    ensures Participants[participant.Name] == null =>
        Participants.Count == old(Participants.Count) + 1;
    {
        if (participants[participant.Name] == null)
            participants[participant.Name] = participant;

        participant.Room = this;
    }
// Given a particular participant, find out how many are logged in for
// their group
[Pure]
public int numInGroup(string name)
{
    int num = 0;
    Participant group = (Participant)participants[name];

    foreach (DictionaryEntry p in participants)
    {
        if (((Participant)p.Value).GetType() == group.GetType())
            num++;
    }
    return num;
}

public int NumBroadcast
{
    set{ numBroadcast = value; }
    get{ return numBroadcast; }
}

public Hashtable Participants
{
    get { return participants; }
}

//Send message to participants in same group
public override string ForwardGroup(
    string! from, string message)

    ensures NumBroadcast == old(NumBroadcast) + numInGroup(from);
    ensures result == from;
{
    Participant sender = (Participant)participants[from];
    Participant receiver;

    IDictionaryEnumerator en = participants.GetEnumerator();
    while (en.MoveNext())
    {
        receiver = (Participant)en.Value;

        if (receiver.GetTypeId() == sender.GetTypeId())
        {
            receiver.Receive(from, message);
            NumBroadcast = NumBroadcast + 1;
        }
    }
    return from;
}

//Send message to all participants
public override string ForwardAll(
    string! from, string message)

    ensures NumBroadcast == old(NumBroadcast) + Participants.Count;
    ensures result == from;
{ IDictionaryEnumerator en = participants.GetEnumerator();
 while (en.MoveNext())
 { Participant p = (Participant)en.Value;
   p.Receive(from, message); NumBroadcast++;
 } return from;
}

// "AbstractColleague"
class Participant
{
  private Chatroom room;
  private string! name;
  private int numMsgRec = 0;
  private int numMsgSent = 0;

  // Constructor
  public Participant(string name)
  {
    this.name = name;
    this.numMsgRec = 0;
    this.numMsgSent = 0;
  }

  // Methods
  public string Name
  {
    get{ return name; }
  }

  public Chatroom Room
  {
    set{ room = value; }
    get{ return room; }
  }

  public int NumMsgSent
  {
    set{ numMsgSent = value; }
    get{ return numMsgSent; }
  }

  public int NumMsgRec
  {
    set{ numMsgRec = value; }
    get{ return numMsgRec; }
  }
}
// Send a message to everyone in same group that is logged in
public void SendGroup(string message)
ensures numMsgSent == old(numMsgSent) + 1;
{
    string from = room.ForwardGroup(name, message);
    numMsgSent++;
}

// Send a message to everyone that is logged in
public void SendAll(string message)
ensures numMsgSent == old(numMsgSent) + 1;
{
    room.ForwardAll(name, message);
    numMsgSent++;
}

// Receive a message from the system
public virtual void Receive(
    string! from, string! message)
requires Room != null;
ensures NumMsgRec == old(NumMsgRec) + 1;
{
    Console.WriteLine("From {0}: '{1}'", from, message);
    numMsgRec++;
}

//" ConcreteColleague1"

class SuperHero : Participant
{
    // Constructor
    public SuperHero(string! name)
    {
        base(name);
    }

    // For testing prepend messages with participants name
    public override void Receive(string! from, string! message)
    ensures NumMsgRec == old(NumMsgRec) + 1;
    {
        Console.Write("To {0}: ", this.Name);
        base.Receive(from, message);
    }
}

//" ConcreteColleague2"

class FantasticFour : Participant
{
    // Constructor
    public FantasticFour(string! name)
    {
        base(name);
    }
}
public override void Receive(string! from, string! message)
ensures NumMsgRec == old(NumMsgRec) + 1;
{
    Console.Write("To {0}: ", this.Name);
    base.Receive(from, message);
}