Supporting the Formal Analysis of Software Systems

Sherrie Campbell
Ann E. Kelley Sobel
Computer Science and Systems Analysis
Miami University
Oxford, OH USA

Abstract— **The formal analysis support environment, Advanced Design Employing Pattern Templates (ADEPT), is outlined. The use of ADEPT will alleviate the software engineer from the minute details that conducting formal analysis by hand requires. The developer will be able to experiment with a variety of derivations from specifications, architectural structures, and mixtures of quality attributes; thereby, gaining new insight into potential versions of high-quality, verifiable code.**

Keywords- ADEPT, design patterns, specification, Spec#, verification

I. INTRODUCTION

Although often confused with computer programming or regarded as a subset of computer science, software engineering is quite distinct from both subjects in that software engineers formally apply science, mathematics, and design analysis to solve practical problems while developing complex software systems. While software is pervasive throughout our society, the growth in its size and complexity continues to cause severe problems with the scheduling and cost of software development and the quality of the software systems produced.

Software engineering has not seen the benefits from the strong design processes that other engineering disciplines commonly utilize. This fact increases the difficulty in designing systems that are provably correct and of high quality. This design problem has plagued programmers since large systems started being developed. It becomes highly desirable to verify the design early since discovering that the design is flawed at the end of the design cycle or after the system is in production can be extremely costly. McMillan states: "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 it is a valid implementation of that architecture."[7]. In this work we aim to aid software developers in choosing designs from basic patterns and creating formal specifications that are translated into a language that will do automatic verification thereby reducing the need to perform the formal verification entirely by hand.

II. BACKGROUND

There has always been a common belief that the use of formal analysis during software development produces "better" programs in terms of both correctness and quality. An educational experiment has demonstrated that software engineering students who have been taught formal analysis skills had increased complex problem solving skills [10]. While the benefits of using formal analysis methods have been established, performing this analysis requires the finesse of succinct expression and proof demonstration that one learns over time from multiple applications. Given formal analysis's complexity, automated support is needed to increase the ease of applying it during software construction.

For the past decade, design patterns have been used to assist in the definition of the architectural structure of software [4]. Design patterns are a set of solutions that have been shown to work for given problems. These patterns can play a vital role in creating sound software system designs as they are utilizing the accumulated knowledge and past experience of others.

Coupling formal specification analysis with design patterns increases and enhances the developers' experiences with good design principles. This approach should increase the rigor of analyzing the design and therefore increase the quality of the software. Given the existence of formal design pattern specifications, a developer could assert that the desired functionality of the software is maintained at the design level as well as check that their design specification satisfies the original software specification. Pushing the analysis closer to the start of the software development process significantly reduces the exponential costs of error mitigation during construction. In this work we intend to start each design with a design pattern and support refinement of the design specifications into high-quality, verifiable code that is written in Spec# [1]. Our specification environment is referred to as ADEPT, Advanced Design Employing Pattern Templates.

III. COMPONENTS OF ADEPT

ADEPT is a software design specification tool that will aid in the creation of correct software. The tool is based on formal specifications of standard design patterns that are then expanded for each individual program and translated into verifiable code.

The software system specification notation that is used in ADEPT was created by Sobel [8] and has been used in a wide variety of settings from the classroom to industrial-strength software. The notation is founded on a verification model that captures the externally visible behavior of a software system. System specifications are written as first-order assertions involving the contents of an interaction sequence where each element of the sequence represents a method invocation. The system specification will be written in terms of an interaction sequence for the entire system. Individual process/class specifications are written in terms of an interaction sequence for that particular process/class. This specification notation is relatively easy to use due to its operational nature and due to its use of only first-order logic. It has been used in multiple undergraduate courses during the course of an educational experiment which additionally supports the claim of ease of use [10].

Using this specification notation, Sobel has written specification design pattern templates for all the design patterns in the behavior category. These templates are based on the known classes and their relationships defined by a particular design pattern. Given a software system specification, a user of ADEPT will be presented with a UML diagram of a basic version of a design pattern and aided in breaking their system specification into individual class specifications based upon that diagram (see [9] for an example of such a decomposition). To highlight the complexity of this task, the ADEPT tool must direct the user in transforming assertions on the structure of a system interaction sequence into assertions on individual class interaction sequences.

Ultimately, these design specifications must be refined into verifiable code and ADEPT uses Spec# for this purpose. Spec# is a tool that has been designed by Microsoft to insert specifications directly into C# programs. The creators of Spec# purport that programs would be "better" if more assumptions about the program were actually recorded and enforced [1]. These assumptions express reasoning about objects and verification of these assumptions can be performed either statically or dynamically.

IV. Mediator Example

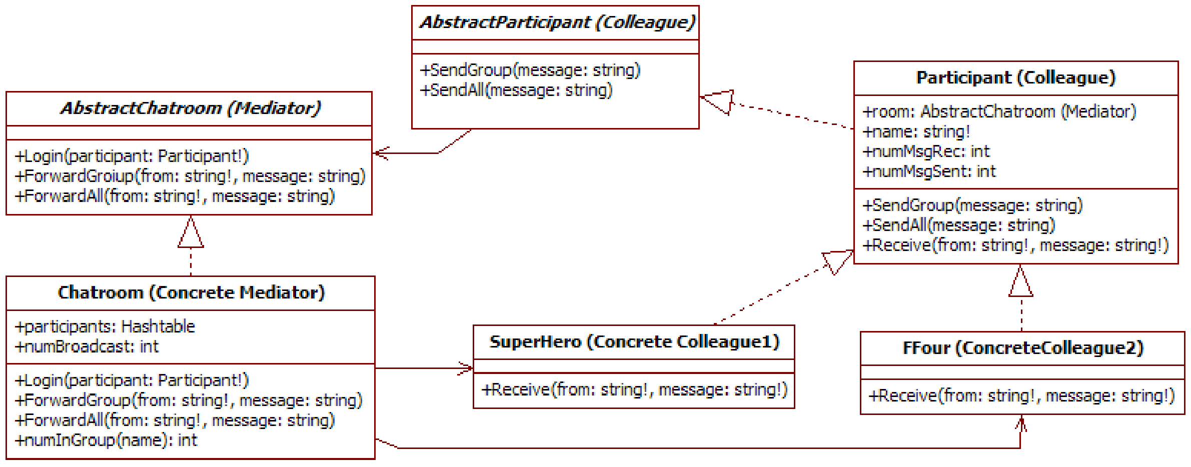
The following example illustrates the translation from a software system specification into the Mediator design pattern specification with its implementation given in Spec#. According to Gamma, Helm, Johnson and Vlissides, the mediator patterns are designed to "define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently." [4]. The mediator pattern requires one mediator and at least two colleagues. The example contains an implementation of a chat room where a group of participants in the chat room are of type SuperHero and Fantastic Four (FFour) as shown in Fig 1. The limitations on this presentation constrain the amount of detail provided; however details on the system specification can be found in [8] and on the Spec# notation in [1].

A. *Mediator implementation*

Elements of the interaction sequence for the mediator Chatroom (h_{MCR}) appear in (1). The first component identifies whether this element represents a receive (?) or send (!). The second component identifies the class that is either receiving the method invocation or is making it. The third component identifies the class object making the invocation. The fourth component identifies the method invoked. Lastly, the fifth component represents the parameter list and this list appears as a sequence.

$$<\text{? Or !, class, obj, method, } \mathbf{X}> \qquad (1)$$

The system specification for the mediator Chatroom is shown in Fig. 2. The system specification represents that if an element appears on the mediator Chatroom interaction sequence representing that a Participant received a message



AbstractParticipant (Colleague)
+SendGroup(message: string)
+SendAll(message: string)
AbstractChatroom (Mediator)
+Login(participant: Participant!)
+ForwardGroiup(from: string!, message: string)
+ForwardAll(from: string!, message: string)
Participant (Colleague)
+room: AbstractChatroom (Mediator)
+name: string!
+numMsgRec: int
+numMsgSent: int
+SendGroup(message: string)
+SendAll(message: string)
+Receive(from: string!, message: string!)
Chatroom (Concrete Mediator)
+participants: Hashtable
+numBroadcast: int
+Login(participant: Participant!)
+ForwardGroup(from: string!, message: string)
+ForwardAll(from: string!, message: string)
+numInGroup(name): int
SuperHero (Concrete Colleague1)
+Receive(from: string!, message: string!)
FFour (ConcreteColleague2)
+Receive(from: string!, message: string!)

Figure 1. UML Diagram of Chatroom

$$
\begin{array}{l}
\forall m \,.\, 0 < m < \# h_{MCR} \,. \\
\quad h_{MCR}[m] = \langle ?, \text{Participant}, \text{obj}, \text{Receive}, \mathbf{X} \rangle \\
\qquad (\exists\, n \,.\, 0<n<m \,.\, h_{MCR}[n] = \langle ?, \text{Chatroom}, \text{room}, \text{Login}, \langle \text{obj} \rangle \rangle \\
\qquad\quad \wedge\, (\exists\, j \,.\, 0< j< m \,.(h_{MCR}[j]=\langle ?, \text{Chatroom}, \text{obj}, \text{ForwardAll}, \mathbf{X} \rangle \\
\qquad\qquad \vee (h_{MCR}[j] = \langle ?, \text{Chatroom}, \text{obj}, \text{ForwardGroup}, \mathbf{X} \rangle)))
\end{array}
$$

Figure 2. System specification

from the Chatroom, then they must have previously:

a) logged into the Chatroom

and either

b) some member of the Chatroom broadcast the message to everyone associated with the Chatroom or

c) some member of the Chatroom broadcast the message to everyone associated with their particular group

The constraints on the mediator Chatroom interaction sequence are broken into assertions that must be satisfied by the individual classes in the implementation. These assertions are represented as ensures clauses in Spec#. Portions of the actual Spec# code follow.

B. *Transferring the specification to Spec#*

In this example the mediator is the Chatroom class and the colleagues are of the abstract class Participant in the Chatroom. Each participant is a member of either the SuperHero or FFour class and they can send messages to everyone in their group or to all participants logged into the chat room.

The individuals do not keep track of the other members of the group; the Chatroom, which is acting as a mediator, ensures messages are sent to the correct participants. By writing our class specifications in Spec#, we are able to verify that the code is working correctly by directly inserting class specification clauses into the C# code. This is demonstrated in the code in Fig. 3.

The numInGroup method is an enabling method to determine how many members there are in a particular group, ensuring that the correct number of messages have been sent. ForwardGroup will send a message to everyone in the same group and the ensures clause checks that the correct number of messages have been sent. In the ForwardAll method, the message is sent to all participants that are logged in and the ensures clause checks that the number of messages sent is the same as the number of logged in participants. These ensure clauses can be very helpful when programming. Even when doing this simple example, an ensure clause caught a simple programming error. The error would have been caught eventually, probably during testing, but was caught earlier because the ensure clause was not valid.

An excerpt of an application that uses our Chatroom example is also shown in Fig. 4. The Thing's message will only get sent to himself and Reed as they are the only people in the FFour group currently logged in. The mediator keeps track of this, alleviating the detail of keeping track of who is logged in from each individual participant.

```
Class Chatroom: AbstractChatroom
{  public Hashtable participants = new Hashtable();
   private int numBroadcast = 0;

// Chatroom login ensures we have added participant
   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;    }

// Mediator forwards message only to those in the
// same concrete colleague group and ensures that
// correct number have been broadcast
   public override void ForwardGroup( string! from,
      string message)
   ensures numBroadcast ==
        old(numBroadcast) + numInGroup(from);
  { Participant sender =
      (Participant)participants[from];
    Participant receiver;

    IDictionaryEnumerator en =
      participants.GetEnumerator();
   while (en.MoveNext())
   {   receiver = (Participant)en.Value;
       if (receiver.GetType() ==
               sender.GetType())
       { receiver.Receive(from, message);
         numBroadcast++; }  }  }

// Mediator forwards message to all participants
// ensures that correct number have been broadcast
   public override void ForwardAll( string! from,
      string message)
   ensures NumBroadcast ==
        old(NumBroadcast) + participants.Count;
  {   IDictionaryEnumerator en =
      participants.GetEnumerator();
     while (en.MoveNext())
    {     Participant p = (Participant)en.Value;
          p.Receive(from, message);
          numBroadcast++; }  }
}

// in Concrete Colleague class, receive message and
//  ensures that it was received correctly
   public override void Receive( string! from,
     string! message)
   ensures numMsgRec == old(numMsgRec) + 1;

  {   Console.Write("To {0}:  ", this.Name);
      base.Receive(from, message);      }
```

Figure 3. Chatroom code fragments

```
// Example of application
Chatroom room = new Chatroom();
Participant Ironman = new SuperHero("IronMan");
Participant Thing = new FFour("The Thing");
Participant Reed =new FFour("Reed");
room.Login(Ironman);
room.Login(Thing);
room.Login(Reed);

Thing.SendGroup ("It's clobberin time…");
```

Figure 4. Instantiation and call

C. *Specification decomposition methodology*

The system specification is written in a notation that captures the interactions between classes. As such, the names used in the system specification will most likely correspond to methods found in the Chatroom (mediator) and SuperHero & FFour (participant) classes. The user is prompted to embellish the code template ADEPT provides for a given UML design pattern diagram. The ADEPT system automatically inserts any system specification clause that contains a use of a method name as a potential ensures clause for that method. If there are names used in the system specification that do not appear in methods, the user is alerted that those definitions are missing (see [9] for a sample session).

In our Chatroom example, the system specification refers to the Chatroom methods Login, ForwardAll, and ForwardGroup and these clauses were inserted into the corresponding ensures clauses. The user can easily modify this specification fragment as needed. Given that Spec# does not permit references to method invocations, these specification clauses must be translated from method invocations into instance variable references. Once the user believes that their system specification transformation is complete, they will use the Spec# compiler to check their evolving implementation.

V. Related Works

There has been has been substantial research in the area of design patterns and specifications, see [2] [3] [5] [6] and [11]. Reference [2] and [5] are both tools that use design patterns to create code but do not emphasize formal specifications. Reference [11] is designed to take a pattern from a template and integrate it into a program design and then check invariants. This was written in Java and Smalltalk and is no longer being supported. Dwyer, Avrunin, and Corbett created a specification system for finite state systems and tested it with over 500 examples [3]. Konrad and Cheng [6] expanded on [3] by adding real-time specifications.

Our work is different from the aforementioned works in that ADEPT uses SPEC# for verification and both the system specification and SPEC# notation use first order logic. Most software engineers have had a discrete mathematics course so they have the background needed to understand these notations. It is our belief that this will make the tool easy to use and more people will be willing to try it.

VI. Future Work

The next step will be to complete the automation of this process within the ADEPT tool. We have currently shown that it is possible to iterate through a design from a design pattern, through design specifications and end up with a program that is verifiable in Spec#. ADEPT will allow the user to choose a basic design pattern, refine their specification and create a Spec# program that can verify that the design is correct.

VII. Summary

The use of formal analysis during software development can increase the quality of the code a software engineer generates. Despite this advantage, software engineers are reluctant to use formal analysis given the complexity of its application. ADEPT promises to help alleviate this burden by supporting the transformation of a system specification into design pattern specifications and then using Spec# to ensure that the corresponding implementation does indeed satisfy these specifications.

References

[1] Barnett, M., RM. Leino, M., & W. Schulte, "The Spec# Programming System: An Overview". LNCS 3362: Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS), Springer-Verlag, 2005.

[2] Design Pattern Toolkit http://www.alphaworks.ibm.com/tech/dptk (IBM, 2007).

[3] Dwyer, M. B., Avrunin, G. S., & Corbett, J. C. "Patterns in property specifications for finite-state verification" Software Engineering, 1999. Proceedings of the 1999 International Conference on, Los Angeles, CA. 411-420.

[4] Gamma, E., R. Helm, R. Johnson, & J. Vlissides, Design patterns elements of reusable object-oriented software. Boston: Addison-Wesley, 2005.

[5] http://www.hillside.net/patterns/tools/ (Hillside.net, 2007)

[6] Konrad, S., & Cheng, B. H. C. (2005). Real-time specification patterns. Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on, St. Louis, MO, USA. 372-381.

[7] McMillan, K. The cadence SMV model checker, http://www.kenmcmil.com/smv.html, November 2007.

[8] Sobel, A.E.K., "The model and methodology of a trace-based operational formal method", Technical Report, Miami University, 1999, 28 pages.

[9] Sobel, A. E. K., & S. Campbell,. Supporting the formal analysis of software designs. 20th Conference on Software Engineering Education & Training, 2007. CSEET '07., Dublin. pp. 123-132.

[10] Sobel, A.E.K. and M. Clarkson, "Formal Methods Application: An empirical tale of software development", IEEE Trans. On Software Engineering, Vol 28. No. 3, pp 308-320, March 2002.

[11] Tool support for object-oriented (design) patterns http://www.serc.nl/people/florijn/work/patterns.html (Florijn, 2003).