Object Modeling Applied to CTAS

by

SeungYong Albert Lee

Submitted to the Department of Electrical Engineering and Computer Science in partial fulfillment of the requirements for the degrees of

Bachelor of Science in Computer Science

and

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 1999

© SeungYong Albert Lee, MCMXCIX. All rights reserved.

The author hereby grants to MIT permission to reproduce and distribute publicly paper and electronic copies of this thesis document in whole or in part.

Author

Department of Electrical Engineering and Computer Science

May 21, 1999

Certified by

Daniel Jackson
Associate Professor
Thesis Supervisor

Accepted by

Arthur C. Smith
Chairman, Department Committee on Graduate Students
Object Modeling Applied to CTAS

by

SeungYong Albert Lee

Submitted to the Department of Electrical Engineering and Computer Science on May 21, 1999, in partial fulfillment of the requirements for the degrees of Bachelor of Science in Computer Science and Master of Engineering in Electrical Engineering and Computer Science

Abstract

The Center-TRACON Automation System (CTAS) is an air-traffic control system developed by NASA for the FAA. It has been deployed in five major airports, and it is being prepared for more widespread deployment. In the fall of 1998, we redesigned a component of CTAS, the communication manager. We have successfully demonstrated the integration of the new component with other existing components.

In this thesis, new object modeling techniques are applied to the new component of CTAS to capture its design and subject it to analysis. These techniques and tools involved are also evaluated for further improvements.

Thesis Supervisor: Daniel Jackson
Title: Associate Professor
Acknowledgments

I would like to thank my thesis advisor, Daniel Jackson for his endless support and generosity.

I thank Room 511 crew, Eric Bothwell, Ben Self, Allison Waingold and Istvan Molnar for their help around the office.

Thanks to Sungtae Kim, Songjoon Park for their friendship which made MIT much more enjoyable place. Thanks to Ji Young Lee for her support and love.

Many thanks to my family who made my stay at MIT and this thesis possible.

SeungYong Albert Lee,

May 21, 1999
Contents

1 Introduction 7
  1.1 Objective .............................................. 7
  1.2 Object Modeling ....................................... 7
  1.3 Organization of This Report .......................... 8

2 Center TRACON Automation System (CTAS) 9
  2.1 CTAS Concept Development Tools ..................... 9
  2.2 National airspace .................................... 10
  2.3 TMA ..................................................... 10
  2.4 FAST ................................................... 15
  2.5 Communication Manager ............................... 17
  2.6 6.894 - Java version of CM .......................... 18

3 Object Modeling Tools: Womble, Alloy and Fox 21
  3.1 Womble .................................................. 21
    3.1.1 Overview ......................................... 21
    3.1.2 Example ......................................... 22
  3.2 Alloy .................................................... 22
    3.2.1 Overview ......................................... 23
    3.2.2 Example ......................................... 23
  3.3 Fox: Fast Object Constraint Solver ................ 25
    3.3.1 Checks consistency of constraints ............... 25
    3.3.2 Checks claimed properties ..................... 27
4 Object Model of CTAS

4.1 Object model extracted by Womble ........................................ 30
4.2 Translating into Alloy Model .................................................. 33
4.3 Using Fox ............................................................................. 35

5 Analysis

5.1 Receiving track data from ISM .................................................. 39
  5.1.1 Problem  ............................................................................ 39
  5.1.2 Solution ................................................................. 41
5.2 Closing connection ............................................................... 41
  5.2.1 Problem ................................................................. 41
  5.2.2 Solution ................................................................. 42
5.3 Deleting RA ................................................................. 42
  5.3.1 Problem ................................................................. 42
  5.3.2 Solution ................................................................. 43

6 Evaluation and Conclusion

6.1 Womble ................................................................. 44
6.2 Alloy and Fox ................................................................. 47
6.3 Redesign of CM ................................................................. 47
6.4 Future Work ................................................................. 47
List of Figures

2-1 CTAS Airspace Definition ........................................ 11
2-2 TMA Scheduling Reference Points ................................. 12
2-3 CTAS TMA Configuration ........................................... 14
2-4 CTAS FAST Configuration .......................................... 16
3-1 Alloy model of a family structure ................................. 24
3-2 invariant schema: Basics .......................................... 26
3-3 configuration found from running Fox on Basics ................. 28
3-4 condition schema: nonempty ...................................... 28
3-5 configuration found from running Fox on nonempty .......... 28
3-6 assertion schema: HusbandsWife ................................. 29
3-7 configuration found from running Fox on HusbandsWife ....... 29
3-8 new invariant added to inv schame Basics ..................... 29
4-1 object model of CM created by Womble ......................... 31
4-2 clustered object model of CM .................................. 32
4-3 RA Manager ....................................................... 38
6-1 original object model from Womble ............................. 45
6-2 modified object model .......................................... 46
Chapter 1

Introduction

1.1 Objective

I have two goals to achieve in this project. One is to experiment with object modeling tools in context of industrial applications. The other is to evaluate the design of the application I analyzed and find code errors.

1.2 Object Modeling

An object model is an representation of the abstract state of a program. It takes the form of a graph whose nodes represent sets of objects, and whose edges represent either subset relationships or associations between objects.

Object-oriented programs often exhibit a code structure that corresponds closely to an object model, with subclassing implementing subset relationships and fields implementing associations. An object model is a 'micro architecture' of a program, showing its essential components and how they interact.

Traditional design representations, such as dependency diagrams and call graphs, are based more on the control structure of the code. Since software architectures are increasingly dynamic, with components being assembled at runtime, control structure is becoming less significant. An object model, in contrast, expresses the constraints that characterize the possible runtime configurations of the system. [6]
1.3 Organization of This Report

This report will be organized in 6 subsequent chapters explained in detail below.

- **Chapter 2. Center TRACON Automation System (CTAS)**
  Center TRACON Automation System (CTAS) is explained in this chapter. The system’s design and functionalities are described in some detail. This chapter also explains the new design of the Communication Manager, a central component of CTAS, developed in 6.894: *Workshop in Software Design* class in fall of 1998.

- **Chapter 3. Object Modeling Tools: Womble, Alloy and Fox**
  Three object modeling tools used in this project are explained. The first one is *Womble*, a tool for extracting object models from Java bytecode. The second one is *Alloy*, a new lightweight language for object modeling. The last one is *Fox*, a fast object constraint solver.

- **Chapter 4. Object Model of CTAS**
  In this chapter, object modeling process and the generated object model is explained in detail.

- **Chapter 5. Analysis**
  This chapter describes analysis performed on the object model generated in the previous chapter. A few flaws of the design discovered during the analysis are discussed as well as other claims shown to be true by code analysis.

- **Chapter 6. Evaluation and Conclusion**
  Object modeling tools are evaluated. Deficiencies, difficulties and usefulness of tools are discussed for further development of tools. Wraps up this report with review on the redesign of CM and future work.
Chapter 2

Center TRACON Automation System (CTAS)

The Center TRACON Automation System, or CTAS, is a set of tools designed to help air traffic controllers manage the increasingly complex air traffic flows at large airports. The tools in CTAS benefit air traffic controllers by reducing stress and workload, and benefit air travelers by reducing delays and increasing safety.[1]

CTAS is more than a set of tools. By combining the skill of controllers with computer generated advisories, the CTAS environment inaugurates a new approach to air traffic control, referred to as human-centered automation. In human-centered automation, controllers increase both their effectiveness to manage complex traffic situations and their awareness of the evolving traffic situation, while retaining their traditional responsibilities for control and safety.[1]

2.1 CTAS Concept Development Tools

The CTAS software is considered a “Concept Development Tool” and continues to be developed at the National Aeronautics and Space Administration (NASA) Ames Research Center in Moffett Field, CA. Currently, there are four major tools under development for CTAS. Among them, following two are the tools identified by the Federal Aviation Administration (FAA) for Build 2 deployment. [2]
1. The Traffic Management Advisor (TMA) is used by the Center’s Traffic Management Coordinator (TMC) to optimize the arrival flow and create a plan. TMA schedules aircraft to the metering fix and to the runway threshold by using the date from the HCS. This information gets updated as the aircraft is tracked through the system. [2]

2. The Final Approach Spacing Tool (FAST) is used by the approach controllers to sequence and schedule aircraft on the final runway approach. [2]

2.2 National airspace

Figure 2-1, CTAS Airspace Definition, illustrates the twenty areas, each approximately four hundred miles across, that comprise the airspace of the forty eight contiguous states. These areas are known as Air Route Traffic Control Centers (ARTCCs) or “Centers”, and are further divided into sectors for ATC purposes. CTAS helps the Center’s TMC optimize the arrival traffic flow and create a plan for managing traffic. CTAS provides ATC advisories to carry out the TMC’s plan using the TMA tool.

In addition, CTAS assists controllers handling arrival air traffic within forty miles of a major airport. Within these areas, known as TRACONs, the CTAS FAST aids approach controllers in assigning aircraft to runways, as well as to sequence and schedule aircraft on the final runway approach.[1]

2.3 TMA

The Center air traffic controllers and Traffic Management Coordinators (TMCs) control arriving aircraft that enter the Center from an adjacent Center or depart from feeder airports within the Center. On the basis of the current and future traffic flow, the TMC creates a plan to deliver the aircraft, safely separated, to the TRACON at a rate that fully subscribes, but does not exceed, the capacity of the TRACON and destination airports. The TMC’s plan consists of sequences and scheduled times of arrival (STAs) at the meter fix, published points that lie on the Center-TRACON
Figure 2-1: CTAS Airspace Definition
Figure 2-2: TMA Scheduling Reference Points

boundary. The Center air traffic controllers issue clearances to the aircraft in Center so that they cross the meter fixes at the STAs specified in the TMC’s plan. Near the TRACON, the Center controllers hand the aircraft off to the TRACON air traffic controllers.[1]

The Traffic Management Advisor (TMA) assists, but does not replace, the Center TMCs and air traffic controllers in several ways. TMA increases situational awareness through a myriad of graphical displays and alerts. TMA also generates statistics and reports about the traffic flow. In addition, TMA computes the underlayed estimated time of arrival (ETA) to the outer meter arc, meter fix, final approach fix and runway threshold for each aircraft. Furthermore, TMA computes the sequences and scheduled
times of arrival (STAs) to the outer meter arc, meter fix, final approach fix, and runway threshold for each aircraft to meet the sequencing and scheduling constraints entered by the TMC. The TMA also assigns each aircraft to a runway to optimize the STAs. Although the TMA computes the STAs and runway assignments for each aircraft in Center, FAST may overrule these STAs and runway assignments when the aircraft enters the TRACON airspace. TMA continually updates its results at a speed comparable to the live radar update rate in response to changing events and controller inputs.[1]

The TMA software suite consists of the following CTAS processes.[1]

1. **Communications Manager (CM)** The CM provides the communication and common database between each of the processes.

2. **Radar Daemon** The Radar Daemon serves as the interface between CTAS and the air traffic control facility’s host computer.

3. **Weather Daemon** The Weather Daemon receives and processes live weather information from the National Oceanic and Atmospheric Administration (NOAA) or the National Weather Service (NWS).

4. **TMA Graphical User Interface (TGUI)** The TGUI receives input from and displays information to the TMCs.

5. **Planview Graphical User Interface (PGUI)** The PGUI receives input from and displays information to the controllers and TMCs.

6. **Route Analyzer (RA)** The RA computes the projected horizontal route that each aircraft will follow. This information is then passed to the Trajectory Synthesizer (TS).

7. **Trajectory Synthesizer (TS)** The TS computes the ETAs and 4-D trajectory, including both the horizontal route and descent profile, for each aircraft.
Figure 2-3: CTAS TMA Configuration
8. **Dynamic Planner**(DP) The DP computes the runway assignment for each aircraft and computes the sequences and STAs to the outer meter arcs, meter fixes, final approach fixes, and runway thresholds. [NASA]

### 2.4 FAST

The Final Approach Spacing Tool (FAST) is a CTAS decision support tool for the terminal area (TRACON) air traffic controllers. The TRACON typically encompasses the airspace within approximately 40 nautical miles of a major airport. FAST provides landing sequences and landing runway assignments, as well as speed, and heading advisories that help controllers manage arrival traffic and achieve and accurately spaced flow of traffic on final approach.[1]

The FAST software suite consists of the following CTAS processes.[1]

1. **Communications Manager** (CM) The CM provides the communication and common database between each of the processes.

2. **Radar Daemon** The Radar Daemon serves as the interface between CTAS and the air traffic control facility’s host computer.

3. **Weather Daemon** The Weather Daemon receives and processes live weather information from the National Oceanic and Atmospheric Administration (NOAA) or the National Weather Service (NWS).

4. **TMA Graphical User Interface** (TGUI) The TGUI receives input from and displays information to the TMCs.

5. **Planview Graphical User Interface** (PGUI) The PGUI receives input from and displays information to the controllers and TMCs.

6. **Route Analyzer** (RA) The RA computes the projected horizontal route that each aircraft will follow. This information is then passed to the Trajectory Synthesizer (TS).
Figure 2-4: CTAS FAST Configuration
7. **Trajectory Synthesizer (TS)** The TS computes the ETAs and 4-D trajectory, including both the horizontal route and descent profile, for each aircraft.

8. **Profile Selector (PFS)** The PFS computes the runway assignment for each aircraft and computes the sequences and STAs to the final approach fixes, and runway thresholds.

### 2.5 Communication Manager

The CM, as its name implies, manages the communications among all of the CTAS processes shown connected to it in the figures above. The exceptions are ISM and WDPD, which act as servers to CM, and TS, which communicates via shared memory with its calling process. The other processes, namely, RA, PFS, PFS_C, DP, PGUI, and TGUI, are clients to CM and depend on it for all communications.[1]

CM receives all the aircraft tracks and flight plans from ISM, and distributes those data in a controlled manner to all the other processes. It performs a similar function for weather information from WDPD. CM uses the aircraft data to create and maintain its own aircraft database; its clients depend on CM to know when an aircraft has been added to or deleted from the CTAS.[1]

CM is the intermediary for any message passed from one client process to another, since all messages pass through CM on their way to their intended destination. Often, a message sent by one process is intended for several processes, and CM manages the transmission of each message to all intended recipients. In fact, CM manages the connection and initialization of all the client processes. If those processes are started and cannot establish communication with CM, they are programmed to exit. Once connected, CM completes their initialization, including sending all current aircraft and airspace configuration information of interest in the form of messages. It is able to do this because, since it receives all transmitted messages, it is able to extract information from the messages in order to maintain key parameters in its aircraft database. Then, if a new process is started, for example, a PGUI, CM is able to send it the latest information on each aircraft, along with any other global parameters that
the PGUI needs to become current with the rest of the system. [1]

Similarly, since CM receives all messages, it is able to record all data necessary for analysis and/or playback.

CM has a GUI that provides several categories of information to the user, as well as enabling key user inputs. The status information includes a panel showing all the aircraft in the system, their aircraft and flight plan types, their track status (tracked, coasting, planned, etc.), and their ETA status. For each aircraft, the user can bring up additional information such as the complete flight plan, current track data, and current ETA’s.[1]

The CM GUI permits the user to do the following: dynamically connect to and disconnect from all valid aircraft data sources, start and stop data recording or playback, initiate two-way communications with the Host, connect to the WDPD, configure the ground speed and heading data filters, specify which TGUI is in control of the DP, and create "fake" aircraft for testing purposes. Additional panels allow the user to filter which aircraft are sent to which processes by flight plan type (e.g., arrival, departure, overflight, etc.) and/or by current Host sector ownership. In addition, the CM GUI scrolling text window is used by the software to print important messages to the user.[1]

2.6 6.894 - Java version of CM

In the fall of 1998, we redesigned a component of CTAS. We reverse engineered the code of the component using a variety of tools, created a radical redesign, implemented it, integrated the new component into the remaining components of the existing system, and demonstrated that it could replicate the basic functionality of the existing system on a recorded radar feed.[5]

In our redesign, we decided not to be constrained by the current realities of the development. We did not ask what we might be done within the existing Communications Manager to improve its design in the short term. Rather, we decided to investigate how the CM might look if we were able to redesign it from scratch. On
the other hand, we were eager to preserve the interfaces between the CM and the rest
of CTAS, so that we could test our new design with the existing algorithmic and user
interface components. [5]

We implemented our design in Java for two reasons. First, we were eager to use
the most modern technology. Second, we felt that were we to use C++, we might
be more tempted to reuse large parts of the existing CM. Its main features, and the
respects in which it differs from the existing design, are:[5]

1. Data abstraction. The existing design is built in a traditional, procedure-
oriented style, in which procedures communicate by arguments and global vari-
ables that are bound to elaborate record structures defined in header files. Most
of the components of our design, in contrast, are abstract data types that en-
capsulate data structures and prevent direct access. We have abstractions for a
message, a message queue, a client proxy, a message handler, aircraft state, etc.

2. Infinite queues. The message queue is the most notable data abstraction. By
providing an illusion of an infinite queue with non-blocking reads and writes,
it allows client code to be written without any concerns for deadlock. We were
thus able to avoid all the complexities that arose in the existing design from
the need to avoid filling the outgoing buffers, such as the batching of flight plan
messages. This data abstraction is a little more intricate than the others.

3. Generic message processor. In the existing design, messages are handled in a
traditional style. A large case statement branches on the message type, and the
message is then deconstructed and copied into a local record by type-specific
code. The message handling code is long and complicated, and there is much
replication. In our redesign, a generic processor finds a handler appropriate to
an incoming message by looking up its type in a table. Code that is executed
for all messages regardless of message type is factored out, so that each handler
is simpler than a branch of the case statement in the existing system. Regis-
tration of message handlers is dynamic, so that it would be easy to change the
association between message types and handlers during execution. This makes
monitoring easy to add: to track occurrences of particular incoming message
type, for example, all that is required is the addition of a new handler that is
registered for the type. In our demonstration code, though, the handlers are all
registered at start-up.

4. Uniform external interfaces. For testing purposes, the existing CM has a playback mode in which it takes input from a recorded file of radar and flight plan information. In its standard mode, this information is received in messages from the Input Source Manager. The two modes use different external interfaces, one involving messages, the other a file. In our design, all input to the CM is via messages. To run the system from the recorded data, we implemented a process that masquerades as the Input Source Manager (ISM), reading the file of recorded data and generating messages that are indistinguishable to the CM from real ISM messages. This scheme makes the CM simpler, and makes playback mode a better predictor of real behaviour. Moreover, it gave us some confidence that our design would work not only for recorded data but also with live data from the ISM. Similarly, the CM has its own user interface that is used for setting basic configuration parameters; in our design this would also be a separate process, its messages handled like any others. We did not implement this user interface, as it is not needed for basic functioning.

5. Message handler language. Rather than writing the message handlers by hand in Java, we chose to generate the handling code from a domain-specific message handling language. This language was designed to accommodate the existing C header file descriptions of message formats, so that we only had to write small fragments of code to indicate, for example, how fields of messages should be mapped to database records.
Chapter 3

Object Modeling Tools: Womble, Alloy and Fox

3.1 Womble

Womble is a tool that extracts object models from Java bytecode. In short, Womble processes a Java program presented as a collection of class files. It applies a simple, lightweight analysis to the code to produce an object model: a graph whose nodes are Java classes, and whose edges represent either subclassing or associations. The object model may be visualized using Blob, a graph layout tool developed that is integrated into Womble, or using Bell Labs's Dot, for which Womble generates an appropriate input file. Womble can also generate module dependence diagrams.[6]

3.1.1 Overview

Womble employs a new approach to object model extraction whose key features are as follows:[6]

1. Associations are inferred by examining a class’s methods, in addition to its declared fields. How a field is used in the code of a class provides more useful clues to its role than its declaration alone. Multiplicity and mutability of associations can both be determined.
2. Container classes, such as `Hashtable`, are handled correctly most of the time: the container class does not appear as a node, but results in appropriate associations.

3. Almost all the analysis is performed locally, classfile by classfile. It is therefore insensitive to missing files, and scales linearly with the size of the system.

Womble is known to be very useful. Even the inventors of static analysis tools do not use them all the time on their own code. But Womble is used on many Java programs very frequently, even on Womble itself. Womble’s utility seems to be due to two factors.[6]

- It places practically no burden on the user at all, primarily because it works on bytecode. No makefiles or paths to header files or libraries are needed; one simply points the tool to some directories containing classfiles and it generates a picture. It is very tolerant of omissions; as more classfiles are presented, it simply extends the picture.

- Object models seem to be a natural and expressive representation of programs. They are generally much smaller than dependence diagrams or call graphs; the picture resulting from a program of 20,000 lines can usually be viewed without any filtering or reorganization.

3.1.2 Example

In this project Womble was the crucial tool to extract an object model from our Java version of CM. Since user interface of Womble is very simple, I will go right into the object model extracted from Java version of CM in next chapter.

3.2 Alloy

Alloy is a lightweight, precise and tractable notation for object modeling. It attempts to combine the practicality of UML’s static structure notation with the rigor of Z, and
to be expressive enough for most object modeling problems while remaining amenable to automatic analysis.[4]

Alloy has a textual notation, of which a subset is also expressible graphically. It has a simple set-based semantics, and a type system that, by treating scalars as singleton sets, allows relations and functions to be treated uniformly, and sidesteps the problem of undefined expressions.[4]

### 3.2.1 Overview

An object model describes a state space, in which each state is viewed abstractly as a collection of objects and their inter-relationships. Object models are used primarily in the design of object-oriented programs. In the specification phase, the objects represent conceptual entities, often drawn from the problem domain. In the design phase, the objects correspond to members of classes, and the model as a whole may be viewed as an abstraction of the set of reachable heap states. Object models can also be useful in other applications that involve relational structure, e.g. for documenting the constraints of an architectural style or integration framework.[4]

Alloy is a new object modeling notation that was designed to meet three criteria: first, to be lightweight – small and easy to use, and capable of expressing common properties tersely and naturally; second, to be precise – that is, having a simple and uniform mathematical semantics; and, third, to be tractable – amenable to efficient and fully automatic semantic analysis.[4]

Alloy was designed hand-in-hand with its checker, Fox (Fast Object Constraint Solver), to be described in next section. Several features of the language were either motivated by the possibility of performing checks – for example, the distinction between invariant, condition and definition schemas – or were added only when a checking mechanism had been devised. [4]

### 3.2.2 Example

Figure 3-1 shows the Alloy model of a family structure. It consists of several
model Family {
  domain {Person, Name}
  state{
    partition Man, Woman : static Person
    Married : Person
    parents : Person -> static Person
    siblings : Person -> Person
    wife("husband") : Man ? -> Woman ?
    name : Person -> Name ?
  }
  def siblings {
    all a, b | a in b.siblings <-> (a.parents = b.parents)
  }
  inv Basics{
    all p | some p.wife <-> p in Man & Married
    no p | p.wife / in p.siblings
    all p | (sole p.parents & Man) && (sole p.parents & Woman)
    no p | p in p.+parents
    all p, q | p.name = q.name -> no (p.parents & q.parents)
  }
  op Marry (m Man!, w Woman!) {
    m not in Married && w not in Married
    m.wife' = w
    all p : Man - m | p.wife' = p.wife
    all p | p.name' = p.name
    all p | p.parents' = p.parents
    Person' = Person
  }
  assert HusbandsWife{
    all p : Married & Woman | p.husband.wife = p
  }
}

Figure 3-1: Alloy model of a family structure
schemas: domain, state, def, op, inv, assert. In domain and state schema, the objects and relations between them are declared. Domain schema defines the top level objects; Person and name in this example. These object declaration also defines types of object. Given these objects, state schema further defines subset of objects and relations between objects. For instance, objects Man and Woman are declared as objects of type Person. Partition tag adds that Man and Woman are partitioning Person to the declaration. [4]

3.3 Fox: Fast Object Constraint Solver

Fox is a new tool for software design that offers, for the first time, fully automatic analysis of object models. Current commercial tools offer only shallow, syntactic analysis—primarily that names are used consistently. Fox, on the other hand, can perform a deep semantic analysis of models that incorporate complex textual constraints. It can check the consistency of constraints, generate sample configurations, simulate execution of operations, and check that operations preserve constraints. [3]

An object model usually describes a huge set of possible configurations. Fox analyzes every configuration within some bounded size exhaustively. It routinely analyzes $10^{30}$ configurations in seconds. In the construction of an object model, Fox can give interactive response as the designer experiments with different constraints. [3]

Fox works by translating the constraints to be analyzed into a huge boolean formula that may then be presented to a variety of backend solvers. The current Fox prototype incorporates a Davis Putnam solver and WalkSAT, a new stochastic algorithm developed at Bell Labs. As boolean satisfaction technology continues to evolve, Fox will be able to exploit it for the analysis of object models. [3]

3.3.1 Checks consistency of constraints

Fox checks consistency of constraints. In the family example from previous section, inv schema (invariant schema) adds constraints to the model. These constraints or
invariants must be consistent. In other words, there must be at least one run time configuration which satisfying all the constraints. In the inv schema from the family example, there are five basic invariants.

\begin{verbatim}
inv Basics{
    all p | some p.wife <-> p in Man & Married
    no p | p.wife / in p.siblings
    all p | (sole p.parents & Man) && (sole p.parents & Woman)
    no p | p in p.+parents
    all p, q | p.name = q.name -> no (p.parents & q.parents)
}
\end{verbatim}

Figure 3-2: invariant schema: Basics

The first one says “every person who has a wife implies that the person is a married man and vice versa”. The second one says “nobody marries his or her sibling”. The third one says “everyone has only one father and one mother”. The fourth one says “nobody can be an ancestor of himself or herself”. The last one says “two people with the same name don’t have the same parents”.

Given above five invariants, it is hard to see if there exists at least one configuration satisfies all five invariants. But, Fox can find a configuration(within the given scope) very quickly. When I ran Fox for above inv schema, it found an following configuration.

But, this configuration was not so interesting since every domain and component is empty. So I define a condition schema to have non empty configuration. Figure3-4 shows the cond schema added to the model.

Essentially, when we run this cond schema Fox will try to find a configuration satisfies both the invariants described in inv schemas and the condition described in the cond schema. The following is the result I got from Fox.

So, we now found a configuration which is not empty and consistent with all the invariants we added to the object model.
Another functionality of Fox is checking claimed properties defined in *assert schemas*. In the family example, the following assertion was inserted to be check.

This assertion says "for every married woman p, p's husband's wife is p." Basically, it’s claiming that if you are married woman, your husband’s wife is yourself. This claim is obvious, but we are not sure it is true for our family object model. So, we run Fox to check this *assert schema* and surprisingly, Fox found a configuration. This configuration Fox found, however, must be interpreted differently from the configurations found during consistency checking. When we ran Fox on *invariant schemas* and *condition schemas*, we wanted to find a configuration to show that those invariants and conditions are consistent. When we run Fox on *asset schemas*, “configuration found” means Fox found a counter-example for the given assertion. The following is the configuration found from running Fox on assert schema *HusbandsWife*.

This result shows that either the assertion is wrong or the family model is not correct. The obvious problem here is that we need a tighter definition of *Married*. As we defined *siblings* as relation between two *Persons* who shares the same parents, we define *Married* as *Persons* who has a husband or a wife.

After the insertion of *Married* schema, *HusbandWife* schema was consistent with the model.

Fox can also perform checks on *Operations* like *Marry* schema in the example. However, description on how to make use of *Op* schema is not discussed here.
Configuration found:
Domains:
Name = {}
Person = {}
Components:
Man = {}
Married = {}
Woman = {}
husband = {}
name = {}
parents = {}
siblings = {}
wife = {}

Figure 3-3: configuration found from running Fox on Basics

nonempty{
    some Person
    some Name
    some Man
    some Woman
    some Married
}

Figure 3-4: condition schema: nonempty

Configuration found:
Domains:
Name = \{N2\}
Person = \{P1, P2\}
Components:
Man = \{P2\}
Married = \{P2\}
Woman = \{P1\}
husband = \{P1 -> P2\}
name = \{P1 -> N2, P2 -> N2\}
parents = {}
siblings = {}
wife = \{P2 -> P1\}

Figure 3-5: configuration found from running Fox on nonempty

28
assert HusbandsWife{
    all p : Married & Woman | p.husband.wife = p
}
}

Figure 3-6: assertion schema: HusbandsWife

Configuration found:
Domains:
    Name = {}
    Person = {P2}
Components:
    Man = {}
    Married = {P2}
    Woman = {P2}
    husband = {}
    name = {}
    parents = {}
    siblings = {}
    wife = {}

Figure 3-7: configuration found from running Fox on HusbandsWife

def Married {
    all a : Married | some (a.wife + a.husband)
}

Figure 3-8: new invariant added to inv schame Basics
Chapter 4

Object Model of CTAS

The process of building an object model of CTAS is discussed in this chapter. In short, Womble extracts an object model from CM Java bytecode. The extracted object model is then translated into Alloy model and some modifications are added where necessary. Then, necessary invariants and claims are checked by Fox for further reshaping of the object model.

4.1 Object model extracted by Womble

First, Womble extracts an object model from CM Java bytecode. Figure 4-1 is the object model created. The source code for Java CM is about 6K lines. The object model extracted by Womble is very manageable size although it is not small enough to fit in one page. Figure 4-1 is only added to give an example. In most cases, multiple pages of a graph are printed out and pasted together for an analysis. The actual Dot file can be found in Appendix C.

This object model is first divided into four clusters for easier organization: engine, aircraft, alviq, mesglang. This division actually follows the package declaration of the source code. Engine cluster consists of all the essential components of the communication manager. Aircraft cluster consists of aircraft, aircraft id and aircraft table objects. Alviq(Application Level Virtual Infinite Queue) consists of input and output queue objects. Lastly, mesglang cluster consists of objects created from the message
Figure 4-1: object model of CM created by Womble
Figure 4-2: clustered object model of CM
generation engine which translates a message script into appropriate objects automatically. Figure 4-2 shows the clustered object model. Some uninteresting objects are eliminated for clarity in this figure.

Engine cluster contains most number of objects and is also the most complicated. It can be further divided into number of functional parts: message processor, scheduler, RA manager, connection manager, message handlers, client filters and client groups.

4.2 Translating into Alloy Model

After the object model extracted from Womble is trimmed out and organized for an analysis, it is translated into Alloy notation. The following is the translated Alloy model of CM. The following is the translated Alloy model.

```alloy
model Communication_Manager {

domain {
   AircraftTable, AircraftEntry, Aircraft, AircraftId, MessageProcessor, Message, RAManager, AcidRAs_Entry, RALoads_Entry, Load, Schedulable, ClientGroup, ClientAddDelHandler, ClientFilter, ClientAddHandlers_Entry, ClientDelHandlers_Entry, HandlerManager, MessageHandler, HandlerTable, Alviq, Thread, Queue, OutputStream, InputStream, Object
}

state {
   DefaultMessageProcessor : MessageProcessor
   Client, TimedEvent, Scheduler, ConnectionManager : Schedulable
   OneClientFilter, ClientTypeFilter : ClientFilter
   Writer, Reader : Thread
   QueueOutputStream : OutputStream
   QueueInputStream : InputStream

   Cma_ac_st_obj, CTAS_flight_info_st_obj, Flight_plan_st_obj,
   Mft_st_obj, Blocked_slot_flight_info_st_obj,
   Hold_flight_info_st_obj, Meter_fix_id_st_obj : Object

   33
}
// from AircraftTable
aircrafts : AircraftTable ? -> AircraftEntry
aircrafts_key : static AircraftEntry ? -> AircraftId ?
aircrafts_value : static AircraftEntry ? -> Aircraft ?
id : static Aircraft -> AircraftId

track : Aircraft -> Cm_ac_st_obj ?
fp : static Aircraft -> Flight_plan_st_obj
fi : static Aircraft -> CTAS_flight_info_st_obj
crossed_fid : Aircraft -> Meter_fix_id_st_obj ?
fid_in_fp : static Flight_plan_st_obj -> Meter_fix_id_st_obj ?
metering_fix_time : static CTAS_flight_info_st_obj -> Mft_st_obj ?
blocked_slot_info : static CTAS_flight_info_st_obj -> Blocked_slot_flight_info_st_obj ?
hold_info : static CTAS_flight_info_st_obj -> Hold_flight_info_st_obj ?
fid_in_mft : static Mft_st_obj -> Meter_fix_id_st_obj ?

// from MessageProcessor
aircraftTable : MessageProcessor -> AircraftTable ?
raManager : MessageProcessor -> RAManager ?
messageProcessor : RAManager ? -> DefaultMessageProcessor ?
RAs : RAManager ? -> Client
RALoads : RAManager ? -> RALoads_Entry
acidRAs : RAManager ? -> AcidRAs_Entry
unassigned : RAManager ? -> AircraftId
acidRAs_key : static AcidRAs_Entry ? -> AircraftId ?
acidRAs_value : static AcidRAs_Entry ? -> Client ?
RALoads_key : static RALoads_Entry ? -> Client ?
RALoads_value : static RALoads_Entry ? -> Load ?

// from ClientGroup
clientGroup_in_MP : MessageProcessor -> ClientGroup ?
clients : ClientGroup ? -> Client
handlerMgr : static ClientGroup ? -> HandlerManager ?
clientDelHandlers : ClientGroup ? -> ClientDelHandlers_Entry
clientDelHandlers_key : static ClientDelHandlers_Entry ? -> ClientFilter ?
clientDelHandlers_value : static ClientDelHandlers_Entry ? ->
  ClientAddDelHandler ?

clientAddHandlers : ClientGroup ? -> ClientAddHandlers_Entry
clientAddHandlers_key : static ClientAddHandlers_Entry ? -> ClientFilter ?
clientAddHandlers_value : static ClientAddHandlers_Entry ? ->
  ClientAddDelHandler ?

defaultRecvHandler : HandlerManager -> MessageHandler ?
sendHandlers : HandlerManager -> HandlerTable ?
Notice that this model only has Domain and State schemas. The extracted model from Womble only gives the structure of the object model and inv and assert schemas are added later for analysis using Fox.

One major difference between the object model extracted from Womble and the translated Alloy model is the addition of entry objects. These objects represent entries to hashtables which Womble can not model perfectly. This point is discussed in the evaluation of tools chapter as a deficiency of Womble.

4.3 Using Fox

After the backbone of the object model is built in Alloy, a repeated process of adding invariants and checking claims is performed.

The first step is discovering invariants that should be added to the object model. For example, let’s take a look at figure 4-3. This is a little portion of the object
model captures RA manager in engine cluster. The functionality of RA Manager is to keep track of RAs connected to CM and aircraft assignments to RAs. AcidRAs is a hashtable which keeps aircraft id and RA assignment pairs. RALoads is a hashtable which keeps how many aircrafts are assigned to each RA. A vector unassigned keeps unassigned aircrafts to be assigned in later time. This vector is only used when there is no RA connected to CM. Then, we have an invariant that no aircraft should be assigned and unassigned at the same time. In other words, unassigned and AcidRAs.acidRAs_key should be disjointed. We will denote this invariant in Alloy notation:

no i | i in RAManager.(unassigned + acidRAs.acidRAs_key)

After adding this invariant to the model, we check the correctness of this invariant by checking the source code. This invariant is actually correct from code analysis and is shown in next chapter for details.

We repeat this local analysis and keep building up the object model. Below is the some of invariants achieved by this process.

- all e | e.aircrafts_key = e.aircrafts_value.id: for every aircraftId in aircraft table is the same as the id of associated aircraft.

- all i : RAManager.(unassigned + acidRAs.acidRAs_key)
  | i in AircraftTable.aircrafts.aircrafts_key: for every aircraftId in RAManager(both unassigned or assigned) in aircraft table.

- no i | i in RAManager.(unassigned + acidRAs.acidRAs_key): unassigned and assigned aircrafts are disjoint.

- all r | all e:r.acidRAs | e.acidRAs_value in r.RAs: every RA in acidRAs table is in RAs table.

- all r | all e:r.RALoads | e.RALoads_key in r.RAs: every RA in RALoads table is in RAs table.
• all c | c in c.clientGroup.clients : every client is in its clientGroup.

• all c | c = c.curMsg.client : for every client, its message's client is itself.

Fox can check for consistency of these invariants by finding a configuration satisfies all the invariants. As expected, Fox successfully found a configuration to show consistency of the above invariants.
Figure 4-3: RA Manager
Chapter 5

Analysis

In this chapter, results from the analysis is discussed. A few subtle flaws which we didn’t encounter during the implementation and the test phase were found. For each flaw, its problem, explanation with code fragments, and possible correction scheme are described in detail.

5.1 Receiving track data from ISM

5.1.1 Problem

A problem was detected while considering the following invariant:

\[
\forall i : \text{RAManager.}(\text{unassigned} + \text{acidRAs.acidRAs_key}) \mid i \in \text{AircraftTable.aircrafts.aircrafts_key}
\]

It turned out that the mistake was from the message script. The following is the excerpt from the message script which handles track data coming in from ISM module.

```
mesg IC_TRACK_DATA IC_sending_track_st {
  recv ISM {
    Cm_ac_st_obj cm = new Cm_ac_st_obj();
    System.err.println("Id: " + track.id.id);
    cm.id = track.id.id;
```
After initializing fields of `Cm_ac_st_obj cm`, new `AircraftId` is created from the message and used to find the aircraft associated in the aircraft table. If the aircraft is found from the table, new track is added to the aircraft, otherwise error message is set to “Ac track for UNKNOWN aircraft”. Then, new `SENDING_AIRCRAFT` message is sent to DP, RA and PGUI. Also, new acid is added to the RAManager indicating that aircraft with the acid is active now and ready to be assigned to a RA.

This implementation breaks very obvious invariants that all the AircraftIds kept in the RA manager, either unassigned or assigned, should exist in the aircraft table. When new track is received and the aircraft associated is not found in the aircraft table, the aircraftId generated is still added to the RA manager to be assigned. This inconsistency can cause problems since RA manager only keeps aircraftIds and the actual aircrafts are stored in the aircraft table.
It is true that this implementation will work under the condition that ISM only sends track data for the aircrafts already in the aircraft table and currently ISM sends ADD_FLIGHT_PLAN message before any track data. However, the track data should be ignored when the associated aircraft is missing from the aircraft table as in the original implementation of CM.

5.1.2 Solution

Solution to this problem is quite easy. We should only execute the last two lines of code when the aircraft is found. So, we should just place the last two lines inside the else clause. The following shows the fixed if statement.

```java
if (ac != null) {
    ac.newTrack(cm);
    Debug.status("Ac track for KNOWN aircraft");
}
else {
    Debug.status("Ac track for UNKNOWN aircraft" + acid);
    send(SENDING_AIRCRAFT, PGUI_RA_DP, cm);
    raManager.assignAcid(acid);
}
```

5.2 Closing connection

5.2.1 Problem

The connection manager listens to the port to see if there is a new client waiting to be connected to CM. A very serious performance flaw was encountered while examining the code.

```java
public void runOneSlice() {
    Debug.status("Processing new connection");
    Socket newSocket = (Socket) socketQueue.firstElement();
    socketQueue.removeElementAt(0);
    try {
```
scheduler.addEvent(new Client(clientGroup, newSocket));
} catch(IOException io) {
    System.err.println("Can't construct new client proxy.");
}

The problem is that after new clients are added to the scheduler, they are never removed from the scheduler after the connection to CM is lost. The above method is from ConnectionManager.java and called by main scheduler of CM. This is only the method invoked to handle the connection from clients. As we can see from the code, it only tries to make an connection to new clients and never deals with lost connections.

### 5.2.2 Solution

The solution to this problem is quite simple. When a connection is lost and detected by the connection manager, it has to be removed from the scheduler by calling a method, removeEvent. Detecting lost connection could be tricky to implement, but it is left to readers since the idea is simple.

### 5.3 Deleting RA

#### 5.3.1 Problem

When a RA is disconnected from CM, RA manager redistributes aircrafts assigned to the disconnected RA to other RAs. The following code shows the code fragment for this particular functionality. However, our implementation missed out a point that the function findLeastLoadedRA() requires to have at least one RA connected to CM. If the last RA is deleted, all the aircrafts assigned to it should be unassigned, but our implementation only writes out error message for this case.

```java
public void delRA(Client ra) throws ClientClosedException {
    RAs.removeElement(ra);
}
```
RALoads.remove(ra);

// reassign ACIDs from this RA to the remaining RAs
Enumeration e = acidRAs.keys();
while(e.hasMoreElements()) {
    AircraftId acid = (AircraftId)e.nextElement();
    Client acidRA = (Client)acidRAs.get(acid);
    if (acidRA.equals(ra)) {
        assignAcidToRA(acid, findLeastLoadedRA());
    }
}

private void assignAcidToRA(AircraftId acid, Client ra) throws ClientClosedException {
    Debug.status("assignAcidToRA: acid="+acid);
    messageProcessor.send_CM_SET_RA_OWNERSHIP(new OneClientFilter(ra), acid);
    acidRAs.put(acid, ra);
    incrementRALoad(ra);
}

5.3.2 Solution

This problem can be easily fixed. All we have to do is when the last RA is deleted, we put all the aircraftIds to unassigned vector, so they can be reassigned later.
Chapter 6

Evaluation and Conclusion

6.1 Womble

Womble was a very powerful tool to extract an object model from CM’s Java bytecode. It was relatively easy to use, but further improvement in GUI is expected. For instance, recursive addition of class files will be useful for programs structured in many sub-directories.

In terms of functionality, Womble was able to create a useful enough object model, so that users can get information right out of it. However, a deficiency of Womble was found during the analysis.

When Womble deals with Hashtables, it just adds an object for keys and an object for values. Then, it adds the same relation connects to both key and value objects. Figure 6-1 shows an example of this.

I recommend to add an intermediate object called “hashtable name”+_entry object first and have two distinct relations from the entry object to key and value object. The example is shown in Figure 6-2.
Figure 6-1: original object model from Womble
Figure 6-2: modified object model
6.2 Alloy and Fox

In terms of functionality, Alloy and Fox were very powerful. Alloy was intuitive and easy to use, and Fox performed various checks in reasonable amount of time. However, in terms of usability, they need some improvement. Alloy model for CM was over 100 lines. Given that our source code was about 6K lines of code, we suspect about 50-100 to 1 ratio here. That means for larger programs – probably over 100K lines of code – will result in around 1000 lines of Alloy model to capture the whole system. In this case, we have to break up the system into subsystems, build object model for each, and perform local analysis. So, some effective ways to deal with large system need to be added Alloy notation.

Fox is still under the development and some user interface bugs – but not severe to function – were found. Also, some improvements in GUI will help users as well.

6.3 Redesign of CM

Redesign of the CM made some radical changes to the existing CM and demonstrated itself to be functional with other existing components of CTAS. However, a few problems were encountered during the analysis. Three of these problems were listed and examined in detail previously. Most of problems occur around the implementation of aircraft and client. Solutions to problems are proposed, but more thorough testing and debugging by testing suite is recommended.

6.4 Future Work

During this project, a lot of time were dedicated to dealing with series of process to create, modify and analyze an object model. As described in previous chapters, creating an object model from Womble, translating to Alloy is a very first step to do an object model analysis, and it was a very time consuming process. So, if we can combine Womble to create an object model in Alloy, we could save a lot of time dealing with translating Womble model to Alloy model.
Another more ambitious improvement will be having graphical interface to Fox. Instead of typing in declaration of objects and relations, users just draw boxes and arrows with label. Graphical front-end of Fox will be more intuitive for user to create an object model and easier to fix. Also, graphical model and textual model should be easier changed to each other within Fox.
Bibliography


