Model Checking Cache Coherence Protocols for Distributed File Systems

Mandana Vaziri-Farahani[1]

May 18, 1995

CMU-CS-95-156

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

DTIC
ELECTE
JUL 17 1995
S D
B

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

19950712 049

DTIC QUALITY INSPECTED 5

[1]Department of Electrical and Computer Engineering, Carnegie Mellon University, mv1k@andrew.cmu.edu.
This research is sponsored by the Wright Laboratory, Aeronautical Systems Center, Air Force Materiel Command, USAF, and the Advanced Research Projects Agency (ARPA) under grant number F33615-93-1-1330. Views and conclusions contained in this document are those of the author and should not be interpreted as necessarily representing official policies or endorsements, either expressed or implied, of Wright Laboratory or the United States Government.
This manuscript is submitted for publication with the understanding that the U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes, notwithstanding any copyright notation thereon.

Accession For
NTIS GRA&I ☑
DTIC TAB ☐
Unannounced ☐
Justification
By per DTIC ltr
Distribution/
Availability Codes
Dist | Avail and/or Special
A-1

Keywords: model checking, verification, finite state machines, abstraction mappings, distributed systems, cache coherence protocols

Abstract

Debugging complex software systems is a major problem. Proving properties of software systems can be thought of as a debugging tool. If a system S must satisfy property P but we can prove that it does not, then S has bugs in it. On the other hand, if S is proved to satisfy P then this is just a confirmation that a certain aspect of S is correct.

We can prove properties of software systems at any stage of development. If we do these proofs early in the design stage, we can prevent errors from propagating to later development stages and therefore save time, money, and human effort.

The traditional approach to proving properties of software systems is theorem proving. This approach has several pragmatic drawbacks. The size of the programs that we can prove correct is not very large. Theorem proving must be done by highly skilled experts in the field.

Our approach to proving properties of software systems is model checking, which consists of proving the property by automatically checking every state in the system. Model checking is a technique successfully used in hardware verification. The model checking tool we use is SMV, which takes as input a finite state machine (FSM) and a property P expressed in Computation Tree Logic (CTL) and outputs `true` if the FSM satisfies P or `false` otherwise. If the outcome is false then SMV also outputs a counterexample.

Because software systems are not, in general, finite state machines, model checking seems to be inadequate at first glance. However, we can overcome this problem by abstracting the system and checking a finite model of it.

We use this method to check cache coherence protocols for distributed systems. The protocols we use are those of the Andrew File System and the Coda File System. We check a cache coherence invariant on the specifications of these protocols, which are natural abstractions of the systems. We perform other abstractions to reduce the size of the systems to manageable finite state machines. SMV checked our cache coherence invariant successfully and indicated that the protocol specifications satisfy this property. For our most complicated protocol, SMV took less than 1 second to check a finite state machine with over 43,600 reachable states.

1. Introduction

Software systems are becoming more and more complex and debugging them is a major problem. Proving properties of software systems can be thought of as a debugging tool. If a system S must satisfy property P but we can prove that it does not, then S has bugs in it. On the other hand, if S is proved to satisfy P then this is just a confirmation that a certain aspect of S is correct.

We can prove properties of software systems at any stage of development. If we do these proofs early in the design stage, we can prevent errors from propagating to later development stages and therefore save time, money and human effort.

The traditional approach to proving properties of software systems is theorem proving. Software systems are, in general, infinite state machines and theorem proving is appropriate because we rely on induction to prove properties in an infinite domain. However, this approach has pragmatic disadvantages. The size of programs about which we can prove properties is not very large. Theorem proving must be done by highly skilled experts in the field and it usually takes a considerable amount of time.

Our approach to proving properties of software systems is model checking, which consists of proving the property by automatically checking every state in the system. Model checking is a technique successfully used in hardware verification. Recent technology advances, like the use of Binary Decision Diagrams (BDD), have allowed considerable improvements in this domain. Model checkers can now verify systems with over 10^{20} states. Examples of recent case studies in the hardware domain are the verification of aspects of the Encore Gigamax multiprocessor [7] and the IEEE Futurebus+ Standard [2]. In both of these case studies the authors discovered significant bugs in the systems.

We use McMillan's model checking tool SMV [6] which takes as input a finite state machine (FSM) and a property P expressed in Computation Tree Logic (CTL) and outputs `true` if the FSM satisfies P or `false` otherwise. If the outcome is false then SMV also outputs a counterexample which allows users to understand why the system does not satisfy the property.

Because software systems are not, in general, finite state machines, model checking seems to be inadequate at first glance. However, we can overcome this problem by abstracting the system and checking a finite model of it.

In this paper, we consider model checking cache coherence protocols for two distributed file systems, the Andrew File System (AFS) and the Coda File System. Mummert, Wing and Satyanarayanan derived abstract models of these protocols [9]. They also specified a cache coherence invariant (CC).

Our goal is to check whether these models satisfy CC, using SMV. For this goal, we perform certain application-specific abstractions to reduce the size of the corresponding SMV input programs.

We consider four models AFS0, AFS1, AFS2 and Coda+. AFS0 is a simple model on which all the others are based. Models for AFS1 and AFS2 were defined by Mummert et al. [9]. Coda+ is the version of the cache coherence protocol for Coda developed by Mummert and Satyanarayanan [8].

Section 2 presents these models and CC. Section 3 describes the SMV input language by giving a simple example. Section 4 presents details of the verification of the four models. Section 5 is a discussion of the abstractions we performed and the general method we used to transform models into SMV programs. Finally, Section 6 is an overview of related work and Section 7 is a summary of our conclusions.

2. Cache Coherence Protocols for Distributed Systems

When connectivity and bandwidth are low in a distributed system, caching of data by clients plays an important role. Caching is also helpful when temporary failures occur. A problem arises when there are several copies of a file in a system. If a client updates its own copy then all other copies of that file become invalid. The goal of cache coherence protocols is to address this problem.

A cache coherence protocol specifies the behavior of clients and servers in a distributed system. Servers are the authority on files that may be cached by clients. Clients and servers communicate by sending messages to each other. Clients can only send messages to servers. These messages contain files or information about files, like their validity. A run of the protocol is an exchange of messages between clients and servers. Validity is determined using recency. The most recent version of a file is valid, all other versions are invalid. Recency is determined by a timestamp.

In the models of cache coherence protocols we consider, there is only one server, one client and one file [9]. There is no global knowledge about the validity of the file. However, the client and the server have beliefs about its validity. They also have beliefs about the presence of the file in the client's cache. Before each run, the client has no belief about the validity of its file (if it has one), and the server has no belief about the validity of the client's file. During a run, the client and server exchange messages and change their beliefs. The details of the four models are further described in Section 4, including a description of how the runs end.

The CC invariant property defined by Mummert, Wing and Satyanarayanan is

> "If a client believes that a cached file is valid,
> then the server also believes that the client's file is valid."

3. Symbolic Model Checking

In this section, we present a simple example to illustrate the SMV input language. Consider the program presented in Figure 1. Assume that $-5 \leq n \leq 5$.

For this program we check the property

"The program terminates"

This property can be expressed in CTL as

```
n := 3;
finish := 0;
while (n != 0) n := n - 1;
finish := 1;
```

Figure 1: Simple Program

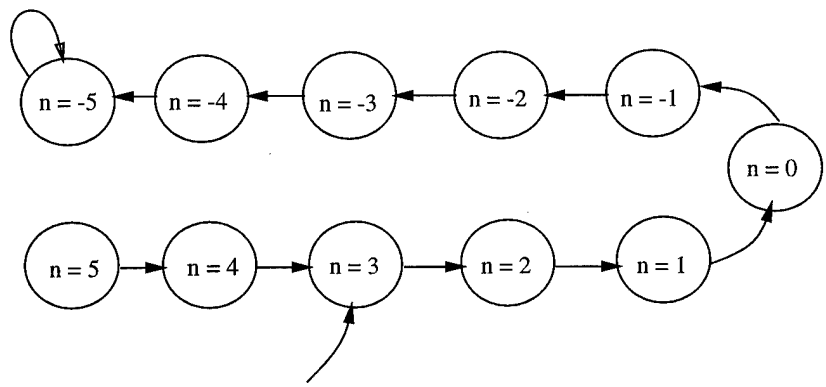


Figure 2: State Diagram Example

```
AF (finish)
```

In CTL, `A` means for all paths and `F` means in some future state along a path. So the formula expresses that along all paths, in some future state, the variable `finish` will be set to `1`.

Figure 2 represents the state diagram for this program. It consists of 11 states, one for each of the possible values of `n`. If we do not restrict the range of `n` then the state diagram is infinite. Figure 2 also shows the transitions between each of the states. There is a transition between each pair of states with values `n` and n', if $n' = n - 1$.

The corresponding SMV input program is shown in Figure 3. One line 1, we declare the module `main`. Each SMV input program must have a module named `main`. The symbol `--` is used for comments. The variable `n` is declared to be an integer ranging from `-5` to `5`. `finish` is a boolean state variable used to detect termination of the program.

The `ASSIGN` construct is used to define the initial values of the state variables as well as their next values. Thus the `ASSIGN` statement declares the state transitions of the system. On line 6, the initial value of `n` is set to `3`. The next value of `n` is defined using a `case` statement. In each line of a `case` statement if the expression to the left of the colon is true then the whole statement gets the value of the expression to the right of the colon. If none of the lines contains a true condition then the statement gets the value to the right of `1` as shown on line 10. In this example, the next value of `n` is `n - 1`, if `n` is currently greater than `-5`, otherwise `n` retains its value. The next value of `finish` is also expressed using a `case` statement. If `(n = 0)` then we know that the loop must terminate and `finish` is set to `1`, corresponding to `finish = true`. Finally the `SPEC` declaration (line 18) is used to declare, in CTL, the property to be checked by SMV.

Figure 4 shows the output of SMV for this example. It indicates that our finite state machine

```
1:  MODULE main -- while loop
2:  VAR
3:      n :  {-5,-4,-3,-2,-1,0,1,2,3,4,5};
4:      finish :  boolean;
5:  ASSIGN
6:      init(n) := 3;
7:      next(n) :=
8:              case
9:              (n > -5) :  n - 1;
10:             1 :  n;
11:             esac;
12:     init(finish) := 0;
13:     next(finish) :=
14:             case
15:             (n = 0) :  1;
16:             1 :  finish;
17:             esac;
18:  SPEC AF(finish)
```

Figure 3: SMV input example

satisfies the termination property, confirming the expected result that the program terminates with an initial value of 3. SMV also outputs some other information about the run. In particular, the user time is 0.1 seconds for this example and the number of reachable states is 9.

If we change the initial value of `n` to `-3`, SMV indicates that the termination property does not hold, as expected. The output shown in Figure 5 also demonstrates a counterexample. SMV prints counterexamples by showing the values of the state variables in successive states, starting with an initial state. For this example, in the initial state (indicated by `state 1.1`), the value of `n` is `-3` and the value of `finish` is `0`. In the next states, the value of `n` decreases to `-4` and then to `-5`. At `state 1.3`, SMV indicates that the system has reached a state it has encountered before, by printing the sentence `loop starts here`. Therefore the value of `finish` stays at 0 forever and there is no path in which it becomes `1`. This proves that the termination property does not hold when the initial value of `n` is `-3`.

```
-- specification AF finish is true

resources used:
user time:  0.1 s, system time:  0.0833333 s
BDD nodes allocated:  170
Bytes allocated:  917504
BDD nodes representing transition relation:  32 + 1
reachable states:  9 (2^3.16993) out of 22 (2^4.45943)
```

Figure 4: SMV output. The specification is true.

```
-- specification AF finish is false
-- as demonstrated by the following execution sequence
state 1.1:
n = -3
finish = 0

state 1.2:
n = -4

-- loop starts here --
state 1.3:
n = -5

state 1.4:


resources used:
user time:  0.1 s, system time:  0.0833333 s
BDD nodes allocated:  533
Bytes allocated:  917504
BDD nodes representing transition relation:  32 + 1
reachable states:  3 (2^1.58496) out of 22 (2^4.45943)
```

Figure 5: SMV output. The specification is false

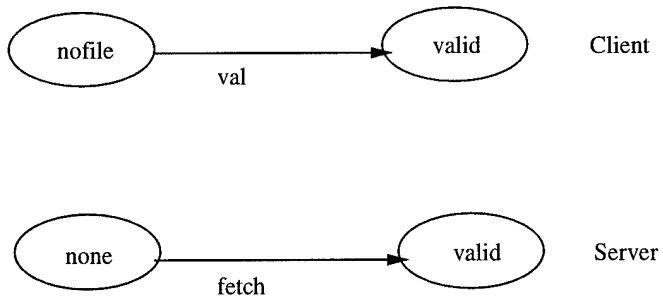


Figure 6: Finite State Diagrams for AFS0

4. Distributed File System Examples

The following sections describe the four examples that we have checked using SMV. In each case, we give the specification of the protocol, construct its finite state diagram, and describe the corresponding SMV input and output.

4.1. AFS0

4.1.1. Specification and State Diagrams for AFS0

In this model, the client simply requests a copy of the file from the server. The server then sends a copy to the client. The server's belief about the file cached by the client ranges over {`valid`, `none`}. If the server's belief is `valid` then the server thinks that there is a file in the client's cache and it is valid; `none`, then the server has no belief about the existence of a file in the client's cache or its validity.

The client's belief ranges over {`valid`, `nofile`}. The client's belief is `valid` if the client thinks that there is a file in its cache and it is valid; `nofile` if it believes that there is no file in its cache.

The client and server communicate by sending messages to each other. These messages range over {`fetch`,`val`}. The message `fetch` is sent by the client to the server to request a new copy of the file. The message `val` is from the server to the client indicating that a copy of the file has been sent.

In AFS0, the client's initial belief is `nofile` and the server's initial belief is `none`. The client sends a `fetch` message to the server. The server then sends a `val` message to the client. At the end of the run, both the client and the server believe that the file cached by the client is valid.

Note that the actual file is not sent by the server in our model of the protocol. We assume that the file is sent along with the message `val`.

Figure 6 represents the finite state diagrams for AFS0. The labels of the transitions are received messages. Upon receipt of the message `val` the client's belief changes from `nofile` to `valid`. Similarly, upon receipt of the message `fetch`, the server's belief changes from `none` to `valid`.

4.1.2. SMV Input Program and Output for AFS0

Figure 7 represents the input program for AFS0. The property to be verified is the following:

```
AG ((Client.belief = valid) -> (Server.belief = valid))
```

The above CTL formula expresses our cache coherence invariant (CC): If the client believes that its file is valid, then the server also believes that the file cached by the client is valid.

The SMV input program is composed of the modules `main`, `client`, and `server`. The module `server` takes a parameter `input` that can be any message coming from the client. The module starts with a `VAR` declaration that defines the belief of the server and its output (denoted by `out`). `out` ranges over {`0`, `fetch`} and denotes a message that the server sends to the client. The message `0` stands for no message. The module `server` then declares the state transitions for each state variable using the `ASSIGN` declaration. The initial value of `belief` is `none`. Its next value is `valid`, if the current value of belief is `none` and the message `fetch` is received. The initial value of `out` is `0`, meaning that no message is sent. The next value of `out` is `val`, if the current value of belief is `none` and the message received is `fetch`. Recall that the server sends the actual copy of the file to the client while sending the message `val`.

The module `client` also takes a parameter `input` that can be any message. The state variables `belief` and `out` are declared using again the `VAR` declaration. The `out` variable ranges over {`0`, `fetch`}. The `ASSIGN` declaration defines the state transitions for the state variables. The initial value of `belief` is `nofile`. Its next value is `valid` if the current belief is `nofile` and the message `val` is received. The initial value of `out` is `0`. If the current value of belief is `nofile`, its next value may be either `0` or `fetch`.

The output of SMV (Figure 8) indicates that the cache coherence invariant holds for AFS0. The user time is 0.083 s and there are six reachable states in this example.

4.2. AFS1

4.2.1. Specification and State Diagrams for AFS1

In AFS1, the client has two initial states: either it has no file or it has a file but no belief about its validity. If the protocol starts with the client having no file in its cache, then the client may request a copy from the server and the protocol terminates when the file is received by the client.

If the protocol starts with the client having a suspect file (one for which it has no belief), then the client can request a validation from the server. If the file is invalid then the client requests a new copy and the run terminates. Otherwise, the protocol simply terminates.

The client's belief about a file ranges over {`nofile`,`valid`, `suspect`}. Its belief is `nofile` if the client thinks that there is no file in its cache; `valid` if it thinks that there is a file in its cache and it is valid; `suspect` if it thinks that there is a file in its cache but it has no belief about the validity of the file. The server's belief about the file cached by the client ranges over {`none`,`valid`} as before.

```
MODULE main --AFS0
VAR
    Client :  client(Server.out);
    Server :  server(Client.out);
SPEC AG ((Client.belief = valid) -> (Server.belief = valid))
.
MODULE server(input)
VAR
    out :  { 0, val };
    belief :  { none, valid } ;
ASSIGN
    init(belief) := none;
    next(belief) :=
                 case
                 (belief = none) & (input = fetch) :  valid;
                 1 :  belief;
                 esac;
    init(out) := 0;
    next(out) :=
                 case
                 (belief = none) & (input = fetch) :  val;
                 1 :  0;
                 esac;
MODULE client(input)
VAR
    out :  { 0, fetch };
    belief :  { valid, nofile };
ASSIGN
    init(belief) := nofile;
    next(belief) :=
                 case
                 (belief = nofile) & (input = val) :  valid;
                 1 :  belief;
                 esac;
    init(out) := 0;
    next(out) :=
                 case
                 (belief = nofile) :  { 0,fetch };
                 1 :  0;
                 esac;
```

Figure 7: SMV input program for AFS0

```
-- specification AG (Client.belief = valid -> Server.beli...  is true

resources used:
user time:  0.0833333 s, system time:  0.116667 s
BDD nodes allocated:  94
Bytes allocated:  917504
BDD nodes representing transition relation:  24 + 1
reachable states:  6 (2^2.58496) out of 16 (2^4)
```

Figure 8: SMV output for AFS0

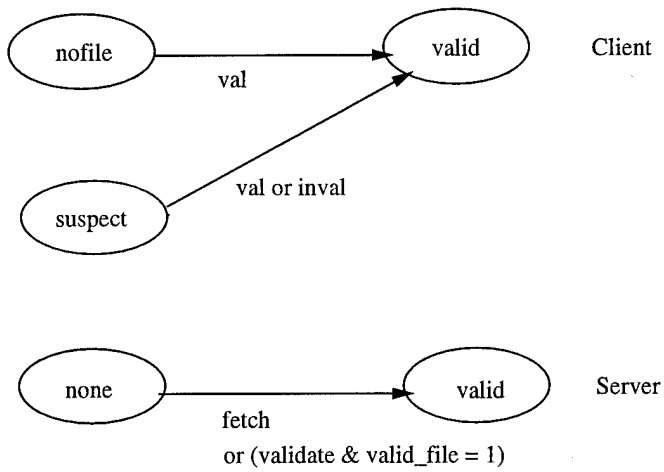


Figure 9: Finite State Diagrams for AFS1

The messages that the client sends to the server range over { `0`, `fetch`, `validate`}. As with AFS0, the message `0` stands for no message and `fetch` stands for a fetch request. The message `validate` is used by the client to validate an existing file in its cache.

The messages that the server sends to the client range over {`0`, `val`,`inval`}. The messages `0` and `val` have the same meanings as in AFS0; `inval` is used by the server to indicate to the client that its cached file is not valid.

Figure 9 shows the finite state diagrams for AFS1.

4.2.2. SMV Input Program and Output for AFS1.

Figure 10 shows the SMV input program for AFS1. The SMV input program for AFS1 is organized in the same way as the one for AFS0. The modules are `main`, `server`, and `client`. The `main` module in AFS1 is identical to the one for AFS0. The property we check for AFS1 is our cache coherence invariant CC.

In the module `server`, the state variables are `out`, `belief` and `valid-file`. The variables `out` and `belief` play the same role as before. `valid-file` is a boolean variable used by the server to decide about the validity of a file cached by the client. The server uses this variable when the

```
MODULE main
VAR
    Client :  client (Server.out);
    Server :  server (Client.out);
SPEC AG ((Client.belief = valid) -> (Server.belief = valid))
SPEC AG ((Server.belief = valid) -> (Client.belief = valid))
MODULE server(input)
VAR
    out :  {0, val, inval };
    belief :  { none, valid,invalid };
    valid-file :  boolean;
ASSIGN
    valid-file := { 0,1 };
    init(belief) :=none;
    next(belief) :=
                 case
                 (belief = none) & (input = fetch) :  valid;
                 (belief = none) & (input = validate) & valid-file :  valid;
                 (belief = none) & (input = validate) & !valid-file :  invalid;
                 (belief = invalid) & (input = fetch) :  valid;
                 1 :  belief;
                 esac;
    init(out) := 0;
    next(out) :=
                 case
                 (belief = none) & (input = fetch) :  val;
                 (belief = none) & (input = validate) & valid-file :  val;
                 (belief = none) & (input = validate) & !valid-file :  inval;
                 (belief = invalid) & (input = fetch) :  val;
                 1 :  0;
                 esac;
MODULE client(input)
VAR
    out :{ 0, fetch, validate};
    belief :  { valid, invalid, suspect, nofile};
ASSIGN
    init(belief) := { nofile, suspect };
    next(belief) :=
                 case
                 (belief = nofile) & (input = val) :  valid;
                 (belief = suspect) & (input = val) :  valid;
                 (belief = suspect) & (input = inval) :  invalid;
                 (belief = invalid) & (input = val) :  valid;
                 1:  belief;
                 esac;
```

SMV input program for AFS1. Figure continues on next page.

```
init(out) := 0;
next(out) :=
          case
          (belief = nofile) :  fetch;
          (belief = invalid) :  fetch;
          (belief = suspect) :  validate;
          1 :  0;
          esac;
```

Figure 10: SMV Input Program for AFS1.

```
-- specification AG (Client.belief = valid -> Server.beli...  is true

resources used:
user time:  0.05 s, system time:  0.133333 s
BDD nodes allocated:  419
Bytes allocated:  917504
BDD nodes representing transition relation:  112 + 1
reachable states:  26 (2^4.70044) out of 216 (2^7.75489)
```

Figure 11: SMV Output for AFS1

client has a suspect file in its cache and requests a validation from the server. In the module server, `valid-file` is non-deterministically set to 0 or 1. If it is set to 1 then the server thinks that the file is valid. Otherwise the server believes that the file is invalid. The initial belief of the server is `none`; its final belief is `valid`.

In the module `client`, the state variables are `out` and `belief` and play the same role as in AFS0. The client's initial belief is `nofile` or `suspect`. If its initial belief is `suspect` and the client receives a failed validation message, then the client believes its file is `invalid`. It then sends a `fetch` message to the server, as indicated in the definition of the transitions for `out`. The client's final belief is `valid`.

Figure 11 shows the output of SMV for AFS1. SMV indicates that the cache coherence invariant is satisfied. The user time is 0.05 s and the number of reachable states is 26.

If we checked the converse of the cache coherence invariant:

```
AG ((Server.belief = valid) -> (Client.belief = valid)),
```

we would get the output shown in Figure 12. Intuitively, this new property should be false as illustrated in the following scenario. Initially, the client has no file in its cache and requests a file from the server. The server receives this request, sends a copy to the client and changes its belief to valid. However the client has not received the file yet, so its belief is not valid. This situation is a counterexample for the property above.

The SMV output represented in Figure 12 gives this counterexample. In `state 1.1` the client's

```
-- specification AG (Server.belief = valid -> Client.beli...  is false
-- as demonstrated by the following execution sequence
state 1.1:
Client.out = 0
Client.belief = nofile
Server.out = 0
Server.belief = none
Server.valid-file = 0

state 1.2:
Client.out = fetch

state 1.3:
Server.out = val
Server.belief = valid


resources used:
user time:  0.116667 s, system time:  0.116667 s
BDD nodes allocated:  1019
Bytes allocated:  917504
BDD nodes representing transition relation:  112 + 1
reachable states:  26 (2^4.70044) out of 216 (2^7.75489)
```

Figure 12: SMV Output for AFS1

belief is `nofile` and the server's belief is `none`. In the next state, the client does a fetch request. This is indicated by `Client.out = fetch`. In `state 1.3`, the server receives this message and sends the `val` message to the client. At the same time, the server's belief becomes `valid`. However in this state the belief of the client is still `nofile`. This proves that the second property is false.

4.3. AFS2

4.3.1. Specification and Finite State Diagrams for AFS2

For AFS2, we consider one server, two identical clients (Client1 and Client2) and one file. In this model, clients have the capability of updating their files and failures may occur in the environment. The model is based on callbacks. When a client caches a valid file, the server promises to notify that client if the file is updated. This promise is called a *callback* [9].

The protocol works as follows. Initially, the clients may have one of two beliefs. A client either believes it has no copy of the file or it has a suspect copy. If the initial belief of a client is that it has no file, it may request a copy from the server. The server then has a callback on that file. If the file is ever updated, the server notifies the client and the client discards its copy.

If the initial belief of a client is that there is a suspect file in its cache, it may request a validation from the server. If the file is valid, then the server has a callback on that file. If the file is invalid the client discards its copy.

If, at any time during a run, a failure occurs in the system, the clients hold their copies of the file suspect and the server discards its beliefs about the validity and the existence of the files cached by the clients.

The server's belief about the file cached by Client1 (Client2) is `belief1` (`belief2`). Each of the server's beliefs ranges over {`valid`, `nocall`}. The belief `valid` has the same meaning as before. The belief `nocall` indicates that the server has no callback on the file cached by the client.

Each client's belief about the file in its cache ranges over {`valid`, `suspect`,`nofile`}. These beliefs have the same meaning as in the AFS1 model. Note that a client discards any file that it believes to be invalid. For this reason we have chosen not to represent the belief `invalid`.

The clients may send the following messages to the server {`fetch`, `validate`,`update`}. An `update` message indicates to the server that the file cached by the client has been updated. The server's messages to the clients are the same as before {`val`,`inval`}.

Figure 13 gives the finite state diagrams for AFS2.

4.3.2. SMV Input Program and Output for AFS2

The cache coherence invariant holds for AFS2 only within certain timing constraints because of transmission delay. Consider the following scenario. Client1 has a valid file in its cache and the server has a callback on that file. Client2 suddenly updates its copy of the file. Then the server immediately believes that the file cached by Client1 is not valid and sends a message to Client1 to

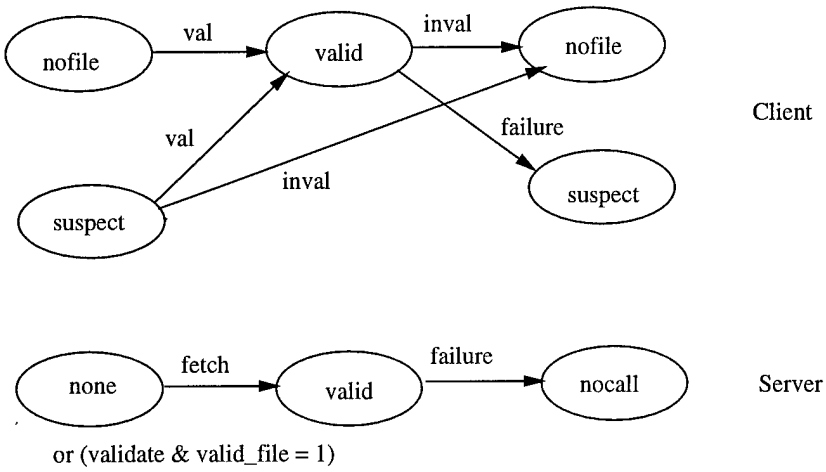


Figure 13: Finite State Diagrams for AFS2

notify it. In this state, Client1 has not yet received the server's message and it still believes that its file is valid. So the invariant does not hold. However if T represents the upper bound on the transmission delay, then the following property is true:

> If a client believes its file is valid at the present time, then at the instant of time right before an interval of time T in the past, the server must have believed that the copy of the client is valid.

In CTL, there are no operators about the past. So this property must be formulated using its contrapositive. The transmission delay is modeled by the amount of time its takes to go from one state to another. This leads us to the following CTL formula:

```
AG ((Server.belief1 = nocall) - >
AX ((Client1.belief = nofile) | (Client1.belief = suspect)))
```

In CTL, `AX` means invariably in the next state.

Figure 14 gives the input program for AFS2. The program consists of instances of modules `Client1`, `Client2`, `Server` and `Env`. The `env` module represents the environment and causes failures to occur between `Client1` and the `Server` and between `Client2` and the `Server`. It has two state variables, `failure1` and `failure2`. Each one of them can be independently set to 1. Once a variable is set to 1, it remains at that value for the rest of the run.

The module `client` works exactly in the same way as in AFS1. The only difference is that in AFS2 a client may also send an update message to the server when it believes its file is valid. The last line in the definition of `out` in the `client` module captures this difference.

The module `server` now has two beliefs as noted before, `belief1` and `belief2`. The server in AFS2 works in a similar way to the server module in AFS1.

```
MODULE main
VAR
    Client1 :  client (Server.out1, Env.failure1);
    Client2 :  client (Server.out2, Env.failure2);
    Server :  server (Client1.out, Client2.out, Env.failure1, Env.failure2);
    Env :  env;
SPEC AG ((Server.belief1 = nocall) → AX ((Client1.belief = nofile) | (Client1.belief = susp
SPEC AG ((Client1.belief1 = valid) → (Server.belief1 = valid))
.
MODULE server(input1, input2, failure1, failure2)
VAR
    out1 :  { 0, val, inval };
    out2 :  { 0, val, inval };
    belief1 :  { valid, nocall };
    belief2 :  { valid, nocall };
    validFile1 :  boolean;
    validFile2 :  boolean;
ASSIGN
    validFile1 := { 0,1 };
    validFile2 := { 0,1 };
    init(belief1) := nocall;
    next(belief1) :=
                  case
                  failure1 :  nocall;
                  (belief1 = nocall) & (input1 = fetch) :  valid;
                  (belief1 = nocall) & (input1 = validate) & validFile1 :  valid;
                  (belief1 = nocall) & (input1 = validate) & !validFile1 :  nocall;
                  (belief1 = valid) & (input2 = update) :  nocall;
                  1 :  belief1;
                  esac;
    init(out1) := 0;
    next(out1) :=
                  case
                  failure1 :  0;
                  (belief1 = nocall) & (input1 = fetch) :  val;
                  (belief1 = nocall) & (input1 = validate) & validFile1 :  val;
                  (belief1 = nocall) & (input1 = validate) & !validFile1 :  inval;
                  (belief1 = valid) & (input2 = update) :  inval;
                  1 :  0;
                  esac;
    init(belief2) := nocall;
    next(belief2) :=
                  case
                  failure2 :  nocall;
                  (belief2 = nocall) & (input2 = fetch) :  valid;
                  (belief2 = nocall) & (input2 = validate) & validFile2 :  valid;
                  (belief2 = nocall) & (input2 = validate) & !validFile2 :  nocall;
                  (belief2 = valid) & (input1 = update) :  nocall;
                  1 :  belief2;
```

```
    init(out2) := 0;
    next(out2) :=
                case
                failure2 :  0;
                (belief2 = nocall) & (input2 = fetch) :  val;
                (belief2 = nocall) & (input2 = validate) & validFile2 :  val;
                (belief2 = nocall) & (input2 = validate) & !validFile2 :  inval;
                (belief2 = valid) & (input1 = update) :  inval;
                1 :  0;
                esac;
MODULE client(input, failure)
VAR
   out :{ 0, fetch, validate, update };
   belief :  {valid, suspect, nofile };
ASSIGN
   init(belief) := {nofile, suspect };
   next(belief) :=
                case
                (belief = nofile) & (input = val) :  valid;
                (belief = suspect) & (input = val) :  valid;
                (belief = suspect) & (input = inval) :  nofile;
                (belief = valid) & failure :  suspect;
                (belief = valid) & (input = inval) :  nofile;
                1:  belief;
                esac;
   init(out) := 0;
   next(out) :=
                case
                (belief = nofile) :  { fetch, 0 };
                (belief = suspect) :  { validate, 0 };
                (belief = valid) :  update;
                1 :  0;
                esac;
MODULE env
VAR
   failure1 :  boolean;
   failure2 :  boolean;
ASSIGN
   init(failure1) := 0;
   next(failure1) :=
                case
                !failure1 :  { 0,1 };
                1 :  1;
                esac;
   init(failure2) := 0;
   next(failure2) :=
                case
                !failure2 :  { 0,1 };
                1 :  1:
```

```
-- specification AG (Server.belief1 = nocall → AX (Clien...  is true
resources used:
user time:  2 s, system time:  0.2 s
BDD nodes allocated:  9742
Bytes allocated:  1048576
BDD nodes representing transition relation:  3710 + 1
reachable states:  7776 (2^12.9248) out of 82944 (2^16.3399)
```

Figure 15: SMV Output for AFS2

Figure 15 gives the output for AFS2. SMV indicates that the finite state machine for AFS2 satisfies our cache coherence invariant. The number of reachable states for AFS2 is 7776 and SMV takes 2 seconds to check it.

4.4. Coda+

4.4.1. Specification and Finite State Diagrams for Coda+

The Coda cache coherence protocol that we consider is the one defined by Mummert and Satyanarayanan [8]. We call this version of the protocol Coda+. For the Coda+ model, we consider one server, two identical clients, one file and one volume version number. A volume is a collection of files. Volume version numbers are cached by the clients in addition to files to reduce client-server communication [8]. A callback on a volume constitutes proof that all cached files in that volume are valid. Coda+ works in the same way as AFS2 does, with the exception that it also deals with version numbers. The server now has four beliefs, two for the files cached by the clients and two for their cached version numbers. A client also has two beliefs, one for its file and one for its volume version number. The messages sent to the server by the clients range over {`Ffetch`, `Fvalidate`,`Fupdate`,`Vfetch`,`Vvalidate`,`Vupdate`}. Messages starting with a `V` (`F`) relate to the version numbers (to the files). The messages that the server sends to the clients range over {`Fval`,`Finval`, `Vval`,`Vinval`}.

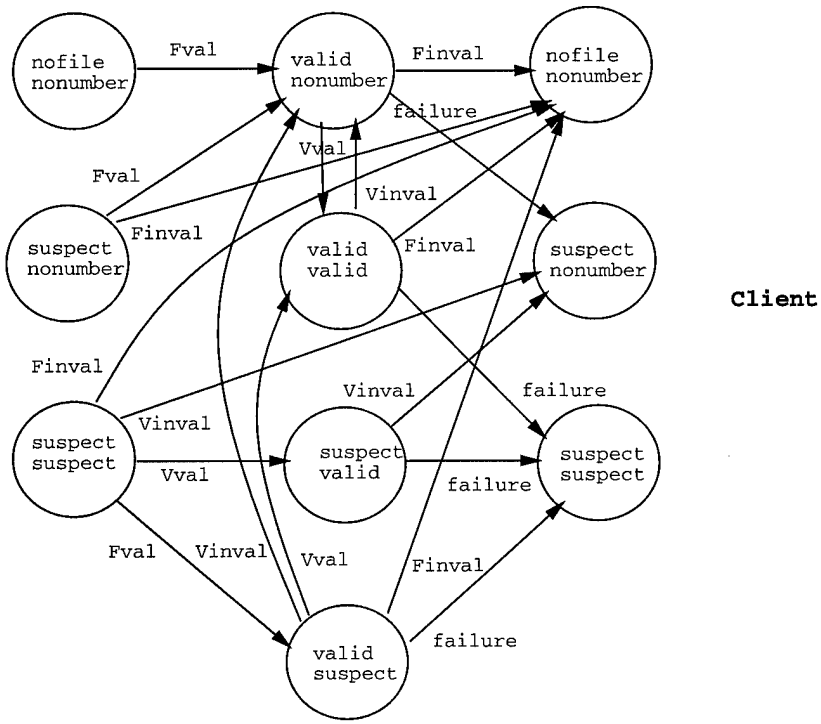
Figure 16 gives the finite state diagrams for Coda+.

4.4.2. SMV Input Program and Output for Coda+

Figure 17 gives the input program for Coda+. The property we check is the same as the one for AFS2. It expresses the contrapositive of our cache coherence invariant and takes into account transmission delay.

Instances of modules in Coda+ are `Server`, `Client1`, `Client2` and `Env`. The module `env` is identical to the one in AFS2. The server now has four beliefs, about the validity of the files and volume version numbers cached by Client1 and Client2. These beliefs can take the values `valid` and `nocall`. Note that again the server does not have an `invalid` belief.

Each of the clients has two beliefs. One belief is about the validity of the file and ranges over



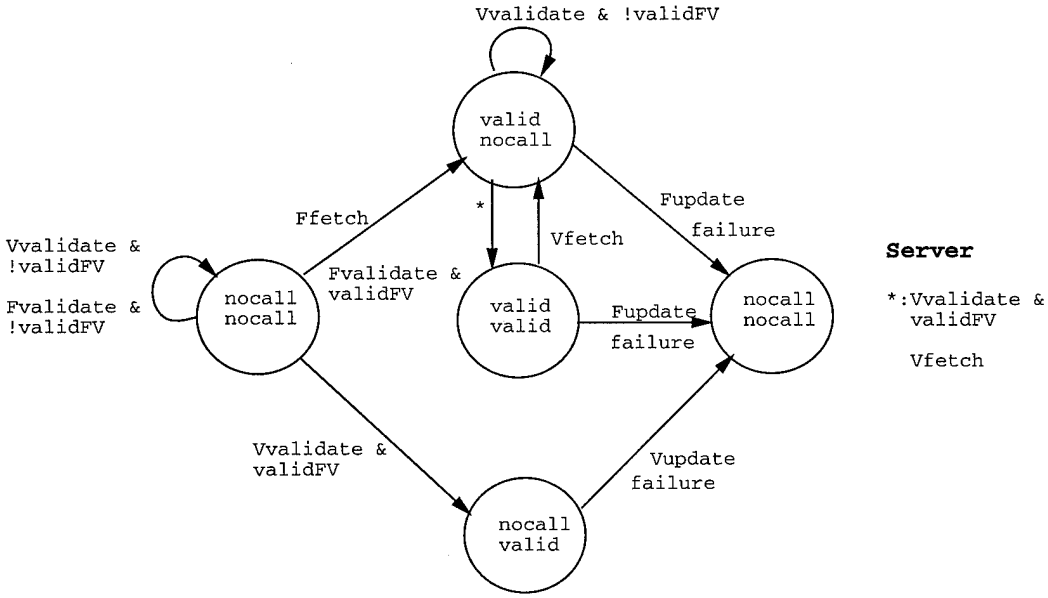


Figure 16: Finite State Diagrams for Coda+. In each circle, the first word represents a belief about the file and the second word represents a belief about the version number.

```
MODULE main
VAR
    Client1 :  client (Server.out1, Env.failure1);
    Client2 :  client (Server.out2, Env.failure2);
    Server :  server (Client1.out, Client2.out, Env.failure1, Env.failure2);
    Env :  env;
SPEC AG ((Server.Fbelief1 = nocall) -> AX ((Client1.Fbelief = nofile) | (Client1.Fbelief =
MODULE server(input1, input2, failure1, failure2)
VAR
    out1 :  {0, Fval, Finval, Vval, Vinval};
    out2 :  {0, Fval, Finval, Vval, Vinval};
    Fbelief1 :  {valid, nocall};
    Fbelief2 :  {valid, nocall};
    Vbelief1 :  {valid, nocall};
    Vbelief2 :  {valid, nocall};
    validFV1 :  boolean;
    validFV2 :  boolean;
ASSIGN
    validFV1 := {0,1};
    validFV2 := {0,1};
    init(Fbelief1) := nocall;
    next(Fbelief1) :=
                    case
                    failure1 :  nocall;
                    (Fbelief1 = nocall) & (input1 = Ffetch) :  valid;
                    (Fbelief1 = nocall) & (input1 = Fvalidate) & validFV1 :  valid;
                    (Fbelief1 = nocall) & (input1 = Fvalidate) & !validFV1 :  nocall;
                    (Fbelief1 = valid) & (input2 = Fupdate) :  nocall;
                    1 :  Fbelief1;
                    esac;
    init(Fbelief2) := nocall;
    next(Fbelief2) :=
                    case
                    failure2 :  nocall;
                    (Fbelief2 = nocall) & (input2 = Ffetch) :  valid;
                    (Fbelief2 = nocall) & (input2 = Fvalidate) & validFV2 :  valid;
                    (Fbelief2 = nocall) & (input2 = Fvalidate) & !validFV2 :  nocall;
                    (Fbelief2 = valid) & (input1 = Fupdate) :  nocall;
                    1 :  Fbelief2;
                    esac;
    init(Vbelief1) := nocall;
    next(Vbelief1) :=
                    case
                    failure1 :  nocall;
                    (Fbelief1 = valid) & (Vbelief1 = nocall) & (input1 = Vfetch) :  valid;
                    (Vbelief1 = nocall) & (input1 = Vvalidate) & validFV1 :  valid;
                    (Vbelief1 = nocall) & (input1 = Vvalidate) & !validFV1 :  nocall;
                    (Vbelief1 = valid) & (input2 = Fupdate) :  nocall;
                    (Vbelief1 = valid) & (input2 = Vupdate) :  nocall;
                    1 :  Vbelief1;
```

```
init(Vbelief2) := nocall;
next(Vbelief2) :=
                case
                failure2 :  nocall;
                (Fbelief2 = valid) & (Vbelief2 = nocall) & (input2 = Vfetch) :  valid;
                (Vbelief2 = nocall) & (input2 = Vvalidate) & validFV2 :  valid;
                (Vbelief2 = nocall) & (input2 = Vvalidate) & !validFV2 :  nocall;
                (Vbelief2 = valid) & (input1 = Fupdate) :  nocall;
                (Vbelief2 = valid) & (input1 = Vupdate) :  nocall;
                1 :  Vbelief2;
                esac;
init(out1) := 0;
next(out1) :=
                case
                failure1 :  0;
                (Fbelief1 = nocall) & (input1 = Ffetch) :  Fval;
                (Fbelief1 = nocall) & (input1 = Fvalidate) & validFV1 :  Fval;
                (Fbelief1 = nocall) & (input1 = Fvalidate) & !validFV1 :  Finval;
                (Fbelief1 = valid) & (input2 = Fupdate) :  Finval;
                (Fbelief1 = valid) & (Vbelief1 = nocall) & (input1 = Vfetch) :  Vval;
                (Vbelief1 = nocall) & (input1 = Vvalidate) & validFV1 :  Vval;
                (Vbelief1 = nocall) & (input1 = Vvalidate) & !validFV1 :  Vinval;
                (Vbelief1 = valid) & (input2 = Vupdate) :  Vinval;
                (Vbelief1 = valid) & (input2 = Fupdate) :  Vinval;
                1 :  0;
                esac;
init(out2) := 0;
next(out2) :=
                case
                failure2 :  0;
                (Fbelief2 = nocall) & (input2 = Ffetch) :  Fval;
                (Fbelief2 = nocall) & (input2 = Fvalidate) & validFV2 :  Fval;
                (Fbelief2 = nocall) & (input2 = Fvalidate) & !validFV2 :  Finval;
                (Fbelief2 = valid) & (input1 = Fupdate) :  Finval;
                (Fbelief2 = valid) & (Vbelief2 = nocall) & (input2 = Vfetch) :  Vval;
                (Vbelief2 = nocall) & (input2 = Vvalidate) & validFV2 :  Vval;
                (Vbelief2 = nocall) & (input2 = Vvalidate) & !validFV2 :  Vinval;
                (Vbelief2 = valid) & (input1 = Vupdate) :  Vinval;
                (Vbelief2 = valid) & (input1 = Fupdate) :  Vinval;
                1 :  0;
                esac;
```

SMV input for Coda+. Figure continues on next page.

```
MODULE client(input, failure)
VAR
    out :{0, Ffetch, Fvalidate, Fupdate, Vfetch, Vvalidate, Vupdate};
    Fbelief :  {valid, suspect, nofile};
    Vbelief :  {valid, suspect, nonumber};
ASSIGN
    init(Fbelief) := {nofile, suspect};
    next(Fbelief) :=
                  case
                  (Fbelief = nofile) & (Vbelief = nonumber) & (input = Fval) :  valid;
                  (Fbelief = suspect) & (Vbelief = nonumber) & (input = Fval) :  valid;
                  (Fbelief = suspect) & (Vbelief = nonumber) & (input = Finval) :  nofile;
                  (Fbelief = suspect) & (Vbelief = suspect) & (input = Fval) :  valid;
                  (Fbelief = suspect) & (Vbelief = suspect) & (input = Finval) :  nofile;
                  (Fbelief = valid) & (Vbelief = nonumber) & failure :  suspect;
                  (Fbelief = valid) & (Vbelief = nonumber) & (input = Finval) :  nofile;
                  (Fbelief = valid) & (Vbelief = valid) & failure :  suspect;
                  (Fbelief = valid) & (Vbelief = valid) & (input = Finval) :  nofile;
                  (Fbelief = valid) & (Vbelief = suspect) & failure :  suspect;
                  (Fbelief = valid) & (Vbelief = suspect) & (input = Finval) :  nofile;
                  1:  Fbelief;
                  esac;
    init(Vbelief) := {nonumber, suspect};
    next(Vbelief) :=
                  case
                  (Fbelief = suspect) & (Vbelief = suspect) & (input = Vval) :  valid;
                  (Fbelief = suspect) & (Vbelief = suspect) & (input = Finval) :  nonumber;
                  (Fbelief = suspect) & (Vbelief = suspect) & (input = Vinval) :  nonumber;
                  (Fbelief = valid) & (Vbelief = nonumber) & (input = Vval) :  valid;
                  (Fbelief = valid) & (Vbelief = valid) & failure :  suspect;
                  (Fbelief = valid) & (Vbelief = valid) & (input = Finval) :  nonumber;
                  (Fbelief = valid) & (Vbelief = valid) & (input = Vinval) :  nonumber;
                  (Fbelief = suspect) & (Vbelief = valid) & (input = Vinval) :  nonumber;
                  (Fbelief = suspect) & (Vbelief = valid) & failure :  suspect;
                  (Fbelief = valid) & (Vbelief = suspect) & failure :  suspect;
                  (Fbelief = valid) & (Vbelief = suspect) & (input = Finval) :  nonumber;
                  (Fbelief = valid) & (Vbelief = suspect) & (input = Vval) :  valid;
                  (Fbelief = valid) & (Vbelief = suspect) & (input = Vinval) :  nonumber;
                  1 :  Vbelief;
                  esac;
```

SMV input for Coda+. Figure continues on next page.

```
    init(out) := 0;
    next(out) :=
              case
              failure :  0;
              (Fbelief = nofile) :  {Ffetch, 0};
              (Fbelief = suspect) & !(Vbelief = valid) :  {Fvalidate, 0};
              (Fbelief = valid) & (Vbelief = nonumber) :  {Vfetch, 0};
              (Vbelief = suspect) :  {Vvalidate, 0};
              (Fbelief = valid) :  Fupdate;
              (Vbelief = valid) :  Vupdate;
              1 :  0;
              esac;
MODULE env
VAR
    failure1 :  boolean;
    failure2 :  boolean;
ASSIGN
    init(failure1) := 0;
    next(failure1) :=
              case
              !failure1 :  {0,1};
              1 :  1;
              esac;
    init(failure2) := 0;
    next(failure2) :=
              case
              !failure2 :  {0,1};
              1 :  1;
              esac;
```

Figure 17: SMV input program for Coda+

```
-- specification AG (Server.Fbelief1 = nocall)-> AX (Clie...  is true
resources used:
user time:  0.7 s, system time:  0.0833333 s
BDD nodes allocated:  6491
Bytes allocated:  917504
BDD nodes representing transition relation:  1572 + 1
reachable states:  43684 (2^15.4148) out of 2.54016e+07 (2^24.5984)
```

Figure 18: SMV output for Coda+

{valid, suspect, nofile}. The other belief is about the validity of the volume version number and ranges over {valid, suspect, nonumber}. The belief nonumber indicates that there is no volume version number in cache.

Figure 18 gives the output of Coda+. SMV indicates that the cache coherence invariant is satisfied. The SMV program for Coda+ has 43,684 reachable states and runs in 0.7 seconds.

5. Discussion

The difficulty in model checking software systems is that software systems are not, in general, finite state machines. To overcome this problem, we abstract systems to make them finite state. In this paper we checked abstract finite models of the cache coherence protocols for the Andrew File System and the Coda file system. However, we also performed other abstractions on these models to reduce the size of the resulting SMV programs even further. Section 5.1 is an overview of these abstractions.

In Section 5.2, we give the general method we used to transform models represented as finite state machines into SMV programs.

5.1. Abstractions

In this section, we consider different aspects of our models and present the abstractions performed for each aspect. These aspects are the number of clients and servers, beliefs, file validity, transmission of messages and failures. The abstractions described are all application-specific.

5.1.1. One server, one client, one file

The models we give have one server, at most two clients, one file and one volume version number. However, the actual protocols were designed for an arbitrary number of servers and clients. We appeal to the generalization rule from logic to justify this abstraction since our property is of the form $\forall x.P(x)$:

$$P(\mathrm{a})$$

$$\overline{\forall x.P(x)}$$

5.1.2. Beliefs

The state variable `belief` really represents two beliefs, one is about the presence of the file in the client's cache, the other is about its validity. The belief `valid` means that the file is present in the cache and it is valid; `suspect` means the file is present but there is no belief about its validity; `nofile` means there no file in cache and thus no belief about its validity.

We could have represented these two beliefs using two state variables, but using only one simplifies the SMV programs.

5.1.3. File Validity

We abstract from the file by not representing it. We also do not represent the client's cache that holds the file. These are reasonable abstractions because the value of the file is not relevant to our cache coherence invariant. We perform this abstraction having in mind the property to be checked.

We also do not represent the timestamp associated with each file. However the server needs to determine the validity of a file when the client requests a validation for an existing file. The server accomplishes this by using a boolean variable `valid-file` which is set to 0 or 1 nondeterministically. The computation tree for the system then contains both possibilities (valid or invalid file) and both possibilities are checked by SMV.

5.1.4. Transmission of messages

Modules in SMV can be thought of as hardware modules connected together with wires. We use these connections between the client and the server module to transmit symbolic messages.

We represent transmission delays using the one-step delay, between cycles, inherent in SMV. This abstracts away from an explicit representation of transmission delay. It also abstracts the exact amount of time due to transmission delays. One step stands for any period of time. This abstraction is possible because our cache coherence invariant is not related to the exact amount of time a transmission delay could take.

5.1.5. Failures

We abstract from all types of failures that can occur in a distributed system by having only two variables `failure1` and `failure2` (AFS2 and Coda+). If any of these variables is set to 1, that indicates a failure of any type between the server and the corresponding client.

5.2. General Method

The abstractions mentioned above allow us to reduce the size of our models. In this section, we present the general method we used to transform these models into SMV programs.

5.2.1. Decomposing the System

The first step in transforming a model into an SMV program is to decompose the system into convenient subsystems. If we did no decomposition, then we would have to specify the state diagram for the whole system, which could be extremely large. Decomposition allows us to describe the state diagrams for the subsystems only. In our examples, we used a natural structuring based on separating clients and servers into different modules and isolating the environment which causes failures into a separate module. This structuring is very similar to decomposition in hardware applications. We can think of clients and servers as hardware modules connected with wires. Although in this application decomposition seems trivial, in more complex systems a more complex decomposition may be needed.

5.2.2. Defining State Diagrams

The second step consists of defining the state diagrams for each subsystem. This task is accomplished by finding first the relevant state variables. These may be found using CTL formulae that describe the properties of the system. In fact, the process of defining formally these properties helps to identify the essential state variables. In our examples, the essential state variable is `belief`. The task of defining formally the cache coherence invariant and finding this state variable had been done by Mummert, Wing and Satyanarayanan [9].

5.2.3. SMV modules

Finally, the SMV module system allows us to describe finite state machines. It is helpful to think about SMV modules as hardware structures. In fact, each module can be represented as a Moore machine. In a Moore machine, the state is usually implemented with latches and represents the current values of the state variables. A logic block is used to compute the next values of the state variables. A module in the SMV input language is roughly a specification of the logic block for each state variable. The description of the Moore machine for a module represents its finite state diagram.

6. Related Work

We used application-specific abstraction mappings to reduce the size of our models. Other approaches to reduce hardware or software systems state spaces consist of exploiting symmetries and using compositional reasoning. The first subsection describes these approaches.

We use SMV directly to describe our abstract models. Another approach consists of using a specification language and mapping that language to the input language of a model checker. The second subsection describes these methods.

6.1. Reducing the State Space

6.1.1. Exploiting Symmetries in the Systems

In the hardware domain, Ip and Dill [4] exploit symmetries to reduce the size of systems. They make symmetries easy to detect by introducing a new data type *scalarset*, a finite and unordered set, to their description language. They have extended their $Mur\varphi$ verifier to generate reduced state spaces. This method can also reduce infinite domains to finite domains. They call this property *data saturation.*

In the software domain, Jackson [5] exploits symmetry of mathematical relations. He analyzes Z specifications, based on his relational calculus, using model checking. In his approach, a state that can be shown to be symmetrical to another state, which has been already checked, is guaranteed not to have an error.

For the examples that we considered, we eliminated any form of symmetry by considering only one server, one client and one file. For AFS2 and Coda+ we have two identical clients and this adds a certain amount of symmetry in the models. Indeed, we could have eliminated one of the clients and replaced it with a module that only updates the file. The reason why we kept the two clients is that they came "for free" in the SMV system. The two clients are instances of the same module.

6.1.2. Compositional Reasoning

Compositional reasoning exploits the natural decomposition of a system into simpler components. The components are model checked separately and therefore the verification of the system as whole is greatly simplified.

In the hardware domain, Grumberg and Long use compositional reasoning and Pnueli's *Assume-Guarantee* paradigm to implement a verifier system, which they use for the verification of a CPU controller [3]. In the *Assume-Guarantee* style of reasoning, when a component is checked, we *assume* that the environment behaves in a certain manner. If the other components in the system *guarantee* this behavior, then the property is satisfied by the system.

In the software domain, Ostroff uses compositional reasoning to model check real-time reactive systems. He uses the StateTime toolset and the temporal logic RTTL, to verify compositionally a real-time resource allocation problem [10].

In our approach, we model check highly abstracted models of systems. These models are small enough that they can be handled by SMV directly. Therefore, we check properties about the entire system, without decomposing it. SMV does not support either kind of compositional reasoning.

6.2. Using a Specification Language

In the software domain, two approaches in verification consists of using a specification language to describe software systems and map the specifications to a model checker's input language.

Jackson [5], as mentioned above, uses a subset of the specification language Z, that is based on his relational calculus. Atlee and Gannon verify properties of event-driven systems using the SCR tabular requirements language [1]. They show how to represent any specification written in a subset of SCR as a finite state machine. They check an automobile cruise control system and a water-level monitoring system with this approach.

The advantage of using a specification language is that the mapping between these specifications and the model checker's input language is done only once. However, this approach has the disadvantage that the domain of systems that can be verified is restricted to the specification language's domain. We use model checking directly, therefore our approach is not restricted to a particular software domain.

7. Conclusion

Model checking is a powerful tool used in hardware verification. By using judiciously defined abstraction mappings, this technique can also be applied to software systems. The critical part of this approach is the process of transforming software systems into finite state machines small enough for model checking. Once these finite abstractions are defined, transforming them into input in a form required by a tool like SMV, is not a difficult task since a knowledge of how the tool works is not required.

More case studies will allow us to discover other kinds of abstractions and to demonstrate the utility of model checking for verifying software systems in general.

Acknowledgments

I would like to thank my advisor Prof. Jeannette Wing for all her help and support and Prof. Edmund Clarke for helping me learn about model checking. I would also like to thank Xudong Zhao and Sergio Campos, who helped me learn how to use SMV.

References

[1] ATLEE, J. M., AND GANNON, J. D. State-based model checking of event driven systems requirements. *IEEE Trans. Soft. Eng.* (Jan. 1993).

[2] CLARKE, E. M., GRUMBERG, O., HIRAISHI, H., JHA, S., LONG, D., MCMILLAN, K. L., AND NESS, L. A. Verification of the futurebus+ cache coherence protocol. In *Proc. of CHDL '93* (1993).

[3] GRUMBERG, O., AND LONG, D. E. Model checking and modular verification. *ACM Transactions on Programming Languages and Systems* (May 1994).

[4] IP, C. N., AND DILL, D. Better verification through symmetry. In *Computer Hardware Description Languages* (1993).

[5] JACKSON, D. Exploiting symmetry in the model checking of relational specifications. Tech. Rep. CMU-CS-94-219, Carnegie Mellon University, School of Computer Science, 1994.

[6] MCMILLAN, K. L. *Symbolic Model Checking: An Approach to the State Explosion Problem.* PhD thesis, Carnegie Mellon University, 1992. CMU-CS-92-131.

[7] MCMILLAN, K. L., AND SCHWALBE, J. Formal verification of the gigamax cache consistency protocol. In *Shared Memory Multiprocessing*, N. Suzuki, Ed. MIT Press, 1992.

[8] MUMMERT, L., AND SATYANARAYANAN, M. Large Granularity Cache Coherence for Intermittent Connectivity. In *USENIX Summer Conference Proceedings* (June 1994), USENIX Association, pp. 279 – 289.

[9] MUMMERT, L., WING, J., AND SATYANARAYANAN, M. Using belief to reason about cache coherence. In *Proceedings of the Symposium on Principles of Distributed Computing* (Aug. 1994). Also CMU-CS-94-151, May 1994.

[10] OSTROFF, J. S. Automated modular specification and verification of real-time reactive systems. In *Proc. of WIFT '95* (1995).