A Case Study in Model Checking Software Systems

Jeannette M. Wing
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA 15213, USA
Mandana Vaziri-Farahani[1]
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139, USA

April 1996
CMU-CS-96-124

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

To appear in Science of Computer Programming. A shorter version appeared in the Proceedings of the ACM SIGSOFT Foundations of Software Engineering Conference 1995.

[1] Work done while at the Electrical and Computer Engineering Department, Carnegie Mellon University.

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 authors and should not be interpreted as necessarily representing official policies or endorsements, either expressed or implied, of Wright Laboratory or the United States Government.

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

Abstract

Model checking is a proven successful technology for verifying hardware. It works, however, on only finite state machines, and most software systems have infinitely many states. Our approach to applying model checking to software hinges on identifying appropriate abstractions that exploit the nature of both the system, S, and the property, ϕ, to be verified. We check ϕ on an abstracted, but finite, model of S.

Following this approach we verified three cache coherence protocols used in distributed file systems. These protocols have to satisfy this property: "If a client believes that a cached file is valid then the authorized server believes that the client's copy is valid." In our finite model of the system, we need only represent the "beliefs" that a client and a server have about a cached file; we can abstract from the caches, the files' contents, and even the files themselves. Moreover, by successive application of the generalization rule from predicate logic, we need only consider a model with at most two clients, one server, and one file. We used McMillan's SMV model checker; on our most complicated protocol, SMV took less than 1 second to check over 43,600 reachable states.

1 Motivation: Theorem Proving and Model Checking

Software systems keep growing in size and complexity. Many large, complex software systems must guarantee certain critical functional, real-time, fault-tolerant, and performance properties. Proving that such a system satisfies these kinds of properties can increase our confidence that it will operate correctly and reliably. Proofs based on formal, rather than informal, techniques make our reasoning precise; moreover, they are amenable to mechanical aids such as syntax and semantics checkers.

Formal reasoning entails comparing two formal objects, e.g., establishing the correctness of a program with respect to a specification or showing that one concurrent process simulates another. The starting point is having two formal objects. There are two general approaches to showing the correspondence between these two objects: theorem proving and model checking. We argue that model checking should and will play a larger role in reasoning about software systems than it does today.

The traditional approach to formal reasoning about software is program verification where one formal object is the *program text* and the other is a *specification* written in some mathematical logic. The formal technique used to show a correspondence between the two objects is based on *theorem proving*. Over time this approach has been shown to work on increasingly larger and larger programs, especially as the tool support like theorem provers and proof checkers has become more and more sophisticated. Yet, it still has drawbacks:

- The size of a program we can prove correct is on the order of only a couple thousand lines of code [14].
- To do such a proof requires highly-skilled people, such as theorem-proving experts, domain experts, or both.
- The human time to do such a proof is on the order of months or even years; the machine time, on the order of hours [14].
- In the course of such a proof, we are often forced to prove "obvious" or "uninteresting" theorems; the amount of effort to prove them is often the same as that for proving the "essential" property of interest.

We believe that there is a time and place for program verification, e.g., for key components of a safety-critical system. In this paper, however, we directly address the concerns of the vast majority of the software engineering community, which questions whether the cost in time, effort, and resources for program verification is worth the eventual benefit gained.

We suggest a radically different tack: *model checking*. The two formal objects compared are a *finite state machine model* of the software system, and as before, a specification written in some mathematical logic.

Model checking is a proven successful verification technology in the hardware community. For example, it has been used to find bugs in published circuits [5, 16], the cache coherence protocol for the Encore Gigamax multiprocessor [32], the IEEE Futurebus+ Standard [10], and telephone switching systems [18]. It has been used to prove safety and liveness properties of the T9000 virtual channel processor [4].

Fundamental to model checking is its reliance on *finite* state machines. Model checking exploits this finiteness property by performing an exhaustive case analysis of the machine's set of states. Recent technological advances have greatly improved the ability to apply this technique to real systems: model checkers can now check systems on the order of 10^{20} states, and for some systems this number can be as large as 10^{1300} [11].

The rationale behind why theorem proving has been the primary approach for reasoning about software is that software systems are, in general, *infinite* state machines. We thus rely on induction to prove in a finite number of steps a property over infinite domains. Model checking, at first glance, seems inappropriate.

There are three methodological reasons for why model checking *is* appropriate. First, the inductive arguments used in proof work well for highly structured components (e.g., generic data types like sets and mappings), but fail at the system level, because of discontinuities in value spaces and irregularities in software structure. We are forced to resort to huge case analyses anyway, perhaps with inductive proofs performed only locally. Thus, though it may seem restrictive to use model checking because we cannot prove something about all possible values drawn from an infinite domain, it is exactly the kind of technology needed to handle the huge case analyses at the system level.

Second, checking a model of the system rather than the system itself raises the level of abstraction at which we do our formal reasoning. Though, we may fall short of doing "exact" reasoning about the original system, we can more quickly, with less effort, and completely automatically do "approximate" reasoning. (An argument must be made, of course, that the model checked is not so abstract that it trivially satisfies the property of interest.)

Third, as the hardware community has discovered, model checking has been tremendously successful at finding bugs in hardware designs. It is more common to find that a system has errors than that it is correct. The same is true, if not more so, for software. Thus, model checking can help software designers find bugs in their designs, where a design is a natural abstraction of the actual working system. Moreover, if the goal of formal reasoning is to find bugs, then it matters less that we do only "approximate," rather than "exact," reasoning.

Thus, though theorem proving has its place, e.g., for doing inductive arguments and low-level program verification, model checking can complement theorem proving efforts. It is worth exploring all avenues as to how.

The target audience of this paper are people who build large, complex software systems for their livelihood. With this case study, we intend to alert them that a technology that they might have shied away from before because of its limitations is worth serious consideration because of its outweighing benefits. Thus, our audience is not the model checking community who already know about the power and usefulness of model checking, but the software engineering community. We see our role in the formal methods community to effect this transfer of technology.

We present the gist of our approach in Section 2. Then in Section 3 we give background information to our case study—our application domain and the cache coherence problem; and a brief description of the model checker that we used, SMV, and its input specification language, CTL. We give details of the case study in Section 4; we used SMV to verify three cache coherence protocols for distributed file systems: two implemented for the Andrew File System (AFS) and one for the Coda File System. In Section 5, we use the case study to illustrate how we followed our approach; we explain different kinds of specific abstractions that software engineers can in general apply to their systems to produce finite state machine models. We discuss related work in Section 6 and close with conclusions and future work in Section 7.

2 Our Approach: Finding Good Abstractions

The successes in reasoning about hardware systems raise the obvious question: *How can model checking be applied to reasoning about software systems?* In this paper, we elaborate on this answer:

> **Approach**: Model check a finite state machine *abstraction* of a software system.

This approach (see Figure 1) relies on finding abstraction mappings, A, to apply to a software system (possibly an infinite state machine), S, and then subjecting the abstract model (a finite machine), M, to a model checker. We use a model checker as a black box to check M against the specification, ϕ. The model checker outputs either true, if M satisfies ϕ, or a counterexample, if it does not.

Key to our approach is to exploit the nature of ϕ to determine what abstractions are reasonable to define and apply. Based on our case study, we broadly classify the ones we identified as follows (elaborated on in Section 5):

- Exploiting the *form* of ϕ. In our example, because of the form of our correctness condition, we use the generalization rule of first-order logic to justify that checking a finite case suffices to show that ϕ holds for the infinite case.

- Exploiting *domain-specific knowledge*. In our distributed systems domain, we collapse distinct failures like crashed nodes and links into a single type of failure. Also, we place a bound on transmission delay, and furthermore

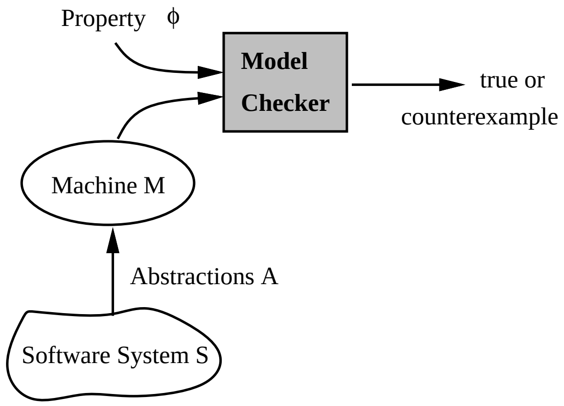

Figure 1: Model Checking Software

model it as a single step taken by the system's environment. Both abstractions are reasonable since implementors of distributed systems make similar simplifications.

- Exploiting *problem-specific knowledge*. ϕ might express an abstract property about an object, not its value. For example, suppose ϕ is a property about the size of a bounded integer set; we do not care about the value of the set itself. If the bound is small, we can model each possible size.

The first kind of abstraction can be applied to any problem or domain. The second can be applied when considering any problem for distributed systems since failures and timing behavior cannot be completely ignored in those systems. The third class of abstractions, though particularly appropriate for our case study, are just examples of abstractions that are generally applicable to other problems. All can be viewed as software *design rules* ("-of-thumb") that when applied raise the level at which we can think about the essence of a software system.

3 Case Study: Cache Coherence Protocols for Distributed File Systems

In a distributed file system, *servers* store files; *clients* store local copies of these files in their caches. Caching increases performance at the client when connectivity and bandwidth are low, and increases availability and reliability when temporary failures occur. Clients communicate with servers by exchanging messages and data (e.g., files). Clients do not communicate with each other. Each file is associated with exactly one designated "home" or *authorized* server.

A problem arises when there are several copies of a file in a system. A file is *valid* if it is the most recent copy in the system. Recency is typically determined by a timestamp associated with the file. If a client updates its copy and it is the most recent update in the system, then all other copies of that file become invalid. The goal of a cache coherence protocol is to make sure a client performs work on only files it believes are valid.

There are two ways to ensure cache coherence. Either the client asks the server whether its copy is valid (*validation-based*) or the server tells the client when the client's copy is no longer valid (*invalidation-based*).

Cache coherence in a distributed system is more difficult to achieve than on a multiprocessor because of the presence of failures and transmission delay. Thus, since global knowledge is impossible to achieve in a distributed system [21], we settle for pairwise knowledge between clients and servers. Our notion of belief captures this pairwise knowledge [34].

An invariant property to prove of all cache coherence protocols is that if a client believes that a cached file is valid, then the server that is the authority on the file believes the client's copy is valid. More formally,

$$\begin{array}{l} \textit{CC:}\ \forall\, C : \textit{client} \;.\; \forall\, S : \textit{server} \;.\; \forall\, f : \textit{file}. \\ \quad C \textit{ believes valid}(f_C) \;\Rightarrow\; S \textit{ believes valid}(f_C) \end{array}$$

where f_C stands for the copy of f at C [34].

A validation-based technique, AFS-1, was used in the Andrew File System from 1984-1985 [38]; for performance reasons, an invalidation-based technique, AFS-2, replaced AFS-1 and has been in use in Andrew since 1985 [24]. In 1993 Mummert, as part of her Ph.D. thesis work, started implementing a more complicated invalidation-based cache coherence protocol [33], similar to AFS-2, as a variation for the Coda Distributed File System [37]. We call Mummert's extension to Coda's protocol Coda+. It was this work that inspired our initial investigation of this case study since Mummert wanted a way to prove formally that her protocol design was correct. Inspired by the logic of authentication [6], Mummert et al. [34] formalized the notions of belief and validity, as used above, and applied the extended logic to reason about cache coherence for AFS-1, AFS-2, and Mummert's variation of Coda's protocol. The actual proofs were done by hand. We observed, however, that the state machine models for all protocols described in [34] are finite and small—trivial for a model checker. So to complete the formal analysis, we subjected all three protocols to model checking.

Before we present the details of the examples, we need to give some background information. We start with an informal description of our system model for reasoning about cache coherence based on the notion of belief. Assumptions made for our model are common to all three protocols we analyzed. The system model and assumptions are taken directly from Mummert et al.'s work in [34]. We then give a brief high-level description of SMV and CTL.

3.1 System Model and Assumptions

We designate hosts as clients or servers of the file system. Clients and servers communicate by sending messages to each other via remote procedure call; each request made by one party requires a response from the other.

Clients speak only to servers, not to other clients. We assume the underlying communication protocol addresses end-to-end concerns such as guaranteeing authenticity and eliminating duplicate messages.

Exactly one repository, which could be one server or a group of servers, is the authority for each file system object. In this paper, we use the generic term "server" for a repository. A file system object is any data contained by a server that may be cached at a client, including files, portions of files, file attributes, or version numbers. For now, it suffices to think of these objects as files; later in Section 4.3 they may also be version numbers.

The local state of a client, C, includes a set of cached data, $C.D$, and a set of beliefs, $C.B$, about objects in its cache. The local state of a server, S, includes a set of data objects, $S.D$, for which it is considered the authority, and for each client C, a set of beliefs, $S_C.B$, that includes which objects are present in C's cache and their validity.

The global state of the system is a tuple of all clients' and servers' local states, plus an *agreement set* A_{CS}, which determines for each data object d whose authority is S and is cached at C, whether the server and client copies are equal. State transitions occur when a component of the global state changes. The agreement set is the state variable used to approximate global knowledge about the validity of all files. It represents pairwise knowledge, which is attained between connected pairs of clients and servers.

We reason about the presence or absence of file system objects cached at clients and their validity. An object is *valid* if it is the most recent copy in the system. Otherwise, it is *invalid*. Recency is determined by a timestamp associated with the file. The timestamp is replaced whenever the file is updated.

Since servers may not hear about updates immediately, validity is global knowledge and cannot always be determined. However, if C and S agree on an object, and S believes its copy is valid, then C should be able to conclude that its cached copy is valid. If S receives an update from a client other than C, then regardless of the global validity of the updated copy, S is justified in telling C that its copy of the object is now invalid.

A cache coherence protocol defines a set of possible *runs*, i.e., message exchanges, between clients and servers about objects. Each run begins with an initial message and ends with a final message; each protocol has a predefined set of initial and final messages. Before a run a client considers all objects in its cache suspect, which means that it does not have a belief on their validity. During a run clients and servers gain beliefs about cached objects. At the end of the run, they all discard all their beliefs.

Failures can terminate runs. If failures occur, beliefs are discarded but clients do not discard their cached files. After a run, a client must again consider all its cached objects suspect because it cannot check if they are valid, nor can the server notify it that they are not. Failures are detected by message timeouts. If a message times out, the principal that sent the message considers it a final message. However, if a client and server both believe a run is in progress, then the run ends once both principals detect the failure.

3.2 SMV and CTL

Users can describe synchronous or asynchronous finite state machines with the model checker SMV [31]. Although for our application domain of distributed systems, it might seem more natural to use an asynchronous model, the protocol itself is more easily described in terms of a synchronous model. The send of a client's message corresponds to the receipt of that message by the server, and in any run of the protocol, the client and server alternate sending messages to each other. Moreover, since the protocol relies on timeouts for detecting failures, in fact the actual system can be viewed as synchronous [39]. Thus, we need only use SMV's synchronous modeling capability.

SMV expects input specifications (ϕ of Figure 1) in the form of Computational Tree Logic (CTL), a subset of branching time temporal logic. A CTL formula is a boolean expression, an existential (`E`) path formula, a universal (`A`) path formula, or the application of standard boolean operators to CTL formulae. A path formula is the application of the temporal operators next (`X`), eventually (`F`), or globally (`G`), to a CTL formula; or the application of until (`U`) to a pair of CTL formulae. Put simply, in a quantified CTL formula the temporal operators always come in pairs of a path quantifier and a state quantifier.

CTL formulae are interpreted with respect to an infinite computation tree derived from finite state transition machines. Each path in the tree is a sequence of states. So, for example, if `P` is a boolean expression then `AG P` is a CTL formula that says "in all paths, in all states `P` holds", i.e., `P` is invariant; `EF P` says "in some path, there is some state in which `P` is true" or more informally, `P` is potentially true.

We explain as needed further details of SMV and CTL in the examples.

4 The Three Protocols

In this section, we present how we used McMillan's SMV model checker to verify the AFS-1, AFS-2, and Coda+ protocols. They are small enough to present in their entirety but "big" enough to let us illustrate common abstractions others can apply to their own systems. For each example, we discuss the restrictions of the general model of Section 3.1 and describe the SMV input and output.

4.1 AFS-1

In AFS-1, a client has two initial states: either it has no files or it has one or more files but no beliefs about their 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. If the protocol starts with the client having suspect files, then the client may request a validation for a file from the server. If the file is invalid then the client requests a new copy and the run terminates. If the file is valid, the protocol simply terminates.

Without loss of generality, we can analyze this cache coherence protocol by considering one client, C, one server, S, and one file, f. Implicitly, the system includes at least one other client to represent remote updates. We can capture the system state as a tuple of four variables, $(C.D, C.B, S.B, A)$[1], where

- $C.D$ ranges over $\emptyset$ and $\{f\}$.
- A ranges over $\emptyset, \{f_C = f_S\}, \{f_C \neq f_S\}$.

A run of the protocol maps some initial state $(C.D_i, C.B_i, S.B_i, A_i)$ to some final terminating state $(C.D_t, C.B_t, S.B_t, A_t)$. The state space is restricted in the following ways. For all states, $(C.D, C.B, S.B, A)$, in a run:

- If $f \in C.D$, either $f_C = f_S \in A$ or $f_C \neq f_S \in A$. This simply means if f is cached at C, it either agrees with the copy at S or it does not. If $C.D = \emptyset$ then $A = \emptyset$.
- At the end of a run, f is always cached and valid. More formally, $C.D_t = \{f\}$, and $A_t = \{f_C = f_S\}$.

4.1.1 State Machine Model

The top graph in Figure 2 shows the state transition graphs for the client, and the bottom, for the server. The nodes are labeled by the value for the state variable, `belief`; the arcs, by the name of the message received that causes the state transition. A run of the protocol begins in an initial state (one of the leftmost nodes) and ends in a final state (one of the rightmost nodes).

The client's belief about a file ranges over {`nofile`, `valid`, `invalid`, `suspect`}. The client's belief is `nofile` if the client cache is empty; `valid`, if the client believes its cached file is valid; `invalid` if it believes its cached file is not valid; `suspect`, if it has no belief about the validity of the file (it could be valid or invalid).

The server's belief about the file cached by the client ranges over {`valid`, `invalid`, `none`}. The server's belief is `valid` if the server believes that the file cached at the client is valid; `invalid`, if the server believes it is not valid; `none`,

[1] We do not need $S.D$ because this simplified model includes only one server and one file.

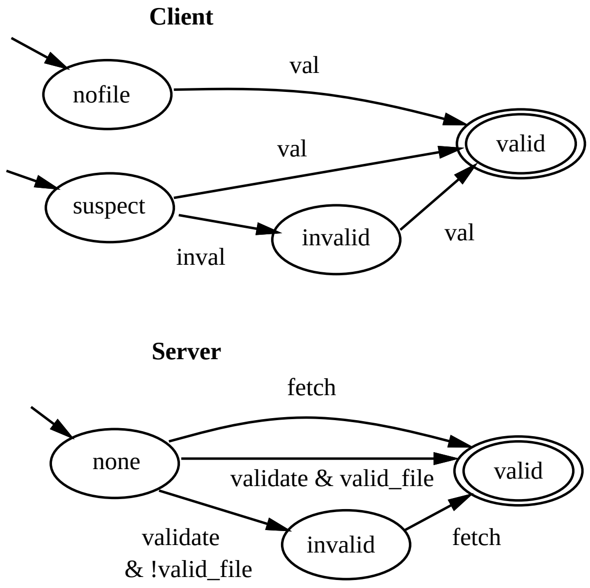

Figure 2: State Transition Graphs for AFS-1

if the server has no belief about the existence of the file in the client's cache or its validity.

The set of messages that the client may send to the server is {`fetch`, `validate`}. The message `fetch` stands for a request for the file. The `validate` message is used by the client to determine the validity of the file in its cache.

The set of messages that the server may send to the client is {`val`, `inval`}. The server sends the `val` (`inval`) message to indicate to the client that its cached file is valid (invalid).

There are three classes of runs of this protocol, corresponding to the three paths in the client's state transition diagram: a cache miss, a cache hit with a successful validation, and a cache hit with a failed validation.

4.1.2 Specification of Cache Coherence in CTL

Given the state variables for a client state machine, Client, and a server state machine, Server, we can express CC as the following CTL formula:

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

Recall that in CTL `AG P` says "for all paths, for all states `P`" or more informally, "invariably, everywhere `P`."

4.1.3 SMV Input for AFS-1

The SMV input program for AFS-1, shown in Figures 3 and 4, is a textual representation of the state transition graphs of Figure 2. We show it for completeness since the transcription of a state transition graph to an SMV program is straightforward. The fifth line in Figure 3 gives the CTL specification against which the state machine is checked.

To show how SMV finds counterexamples, we add in the sixth line another CTL specification for SMV to check, the converse of our correctness condition:

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

The SMV input program is composed of the modules `main`, `server`, and `client`. The third and fourth lines declare instances of the server and client modules.

The module `server` takes a parameter `input` that can be any message coming from the client, indicated by the instantiation of the parameter by `Client.out` in the fourth line.

The initial belief of the server is `none`; the final belief is `valid`. The SMV `init` and `next` functions define the initial value and next-state value for a state variable. In SMV a `case` expression returns the value of the first expression on the right hand side of the colon (`:`), such that the corresponding condition on the left hand side is true. Cases are evaluated in order. Thus, the fifth case (`1:belief`) in the definition of the `next` function for the `belief` variable is the catchall "else" case; here it says that `belief`'s value stays the same if none of the conditions of the previous four cases hold, i.e., the server's belief about the file does not change.

The server module has two other state variables besides `belief`. The `out` variable ranges over the messages that the server may send to the client. The `0` message stands for no message and is needed to model the initial state; the fifth case (`1:0`) in the definition of the `next function` for the `out` variable says that no message is sent if none of the conditions of the previous four cases hold. The `valid-file` boolean variable models the effect of the environment as perceived by the server. It is used when the client has a suspect file in its cache and requests a validation from the server. We need to model both the possibility that the server has received an update by some other client and the possibility that it has not. If an update by some other client has occurred then the server reflects that by nondeterministically setting the value of `valid-file` to 0; otherwise, the server sets the value to 1 (the file cached at the client is still valid). This nondeterminism is captured in the SMV input program in Figure 3 by the assignment of the set of values { `0,1` } to the variable `valid-file`.

In the module `client`, in addition to the state variable `belief`, as for the server, we use the `out` variable to range over the messages that the client may send to the server. The client's initial belief is `nofile` or `suspect`. If the initial belief is `suspect` and a failed validation message is received, then the client

```
MODULE main  -- afs1
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;
```

Figure 3: SMV Input Program for AFS-1: Main and Server Modules

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

Figure 4: SMV Input Program for AFS-1: Client Module

```
-- specification AG (Client.belief = valid -> Server.beli... is true
-- 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: 0.133333 s, system time: 0.116667 s
BDD nodes allocated: 1048
Bytes allocated: 917504
BDD nodes representing transition relation: 112 + 1
reachable states: 26 (2^ 4.70044) out of 216 (2^ 7.75489)
```

Figure 5: SMV Output for AFS-1

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`.

4.1.4 SMV Output for AFS-1

Figure 5 shows the output of SMV for AFS-1.

SMV indicates that (1) the first property, the cache coherence invariant, is true and (2) the second property, the converse, is false. SMV produces a counterexample indicated by the sequence of three states named 1.1, 1.2, and 1.3 in the output, where in the first state, the initial values of the state variables are given, and in subsequent states, the new values of only the changed state variables are given. Thus in the last state of the sequence, the second property is false since the value of the server's `belief` variable is `valid` but that of the client's is `nofile`. This counterexample corresponds to the following scenario. Initially, the client has no file and the server has no beliefs. The client then requests a copy from the server. Then, the server receives this fetch request and sends a copy to the client, believing, of course, that the file sent and thus cached by the client is valid. Thus, the server believes the file is valid, but in this last state, the client has not received the server's message and still believes that it has no file.

The line after `resources used:` says that SMV takes fractions of a second to check both properties and the last line says that the number of reachable states for AFS-1 is 26.

4.2 AFS-2

The two main differences between AFS-1 and AFS-2 are that (1) AFS-2 maintains cache coherence using *callbacks* [24], and (2) it needs to handle the case of failures. When a client caches a file, the server promises that it will notify that client if the file changes. This is called a *callback promise*, or simply *callback*. If the file changes, the server's invalidation message is called a *callback break*.

AFS-2 works as follows. Initially, a client may have one of two beliefs about a file. It either believes it has no copy of the file or it has a suspect copy. If the client's initial belief is that it has no file, the client 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 client's initial belief is that there is a suspect file in its cache, the client 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 then consider their copies of the file suspect and the server discards its beliefs about the validity of the files cached by the clients. The client does not discard the file, because it is cheaper to validate the file when the failure is repaired than to refetch it. We assume for simplicity that clients do not discard files for any reason other than invalidation. Of course, in practice clients may discard files for other reasons, such as lack of space.

The values for $C.D$ and A are as for AFS-1 described in Section 4.1. Restrictions on the state space are as follows:

1. If $f \in C.D$, either $f_C = f_S \in A$ or $f_C \neq f_S \in A$. This simply means if f is cached at C, it either agrees with the copy at S or it does not. If $C.D = \emptyset$ then $A = \emptyset$.

2. When an object is invalidated, C must discard it. Then

$$\{f_C \neq f_S\} \in A \Rightarrow C.D_t = \emptyset$$

3. At the end of a run, either f is not cached ($C.D_t = \emptyset$) and $A = \emptyset$, or f is cached ($C.D_t = \{f\}$) and agrees with the server ($A_t = \{f_C = f_S\}$).

4.2.1 State Machine Models

For AFS-2, we also consider a simplified model with just one server, two identical clients (Client1 and Client2), and one file. Figure 6 gives the state transition graphs for each client and the server.

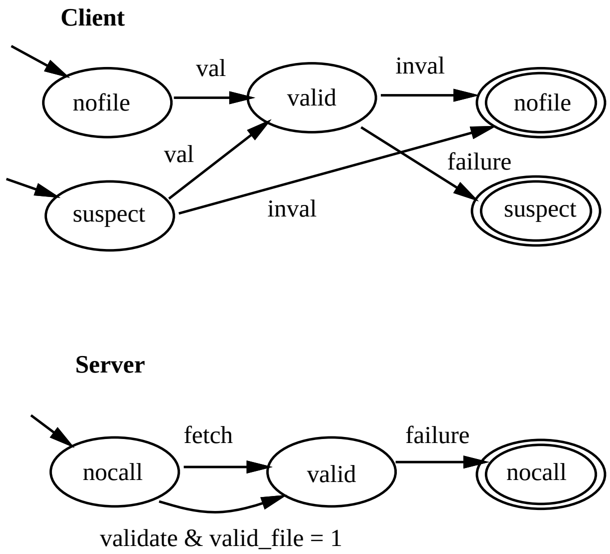

Figure 6: State Transition Graphs for AFS-2

The client's belief about a file ranges over {`valid`, `suspect`, `nofile`}. The belief `valid` indicates that the file is in cache and it is valid; `suspect`, that the file is in cache but the client has no belief about its validity; `nofile`, that there is no file in cache. Since a file that is believed to be invalid is immediately discarded by the clients, we have chosen not to represent the belief `invalid` to simplify the system.

For each client, the server has a belief about the validity of the file cached by that client. Each belief ranges over {`valid`, `nocall`}, where `valid` indicates that there is a file in the client's cache and it is valid; `nocall`, that the server has no callback on the file cached by a client.

The messages that the clients may send to the server are {`fetch`, `validate`, `update`}. An `update` message to the server indicates that the file cached by the client has been updated. The server's messages are the same as for AFS-1.

There are five classes of runs, corresponding to the five paths shown in the client's state transition diagram:

- cache miss, no failures
- successful validation, no failures
- failed validation, no other failures
- cache miss followed by failure

- successful validation followed by failure

Recall that at the beginning of each run, the client neither believes its cached state is valid, nor believes it invalid.

4.2.2 SMV Input and Output for AFS-2

If we use the same correctness criterion for AFS-2 as we did for AFS-1, SMV gives a counterexample shown in Figure 7. Our "invariant" is not an invariant! The reason is that the cache coherence invariant holds for AFS-2 only within certain timing constraints due to transmission delay. Consider the following scenario, indicated by the counterexample. 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 notify it. In this state, Client1 has not yet received the server's message and it still believes that its file is valid. So there is a period of time due to transmission delay during which the invariant does not hold. Let τ represent the upper bound on this time interval, then the following property is true:

> CC': If a client believes its file is valid at the present, then before some past interval of time of length τ, the server believed the file cached by that client is valid.

In CTL, there are operators about the future, not the past, so we reformulate this property using its contrapositive. We also model the transmission delay by the amount of time it takes to go from one state to another, i.e., a single step. (We justify this aspect of our model in Section 5.2.) This leads us to the following CTL formula, which is the cache coherence condition (CC' above) we use for AFS-2:

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

where `AX` means "invariably, in the next state."

The SMV input for AFS-2 is similar to that for AFS-1. The system consists of the instances Client1, Client2, Server, and Env. We introduce the `env` module (Figure 8), representing the environment, so that we can explicitly model failures that occur between Client1 and the Server and between Client2 and the Server. The `env` module 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 `server` module has two beliefs, `belief1` and `belief2`. The `server` module in AFS-2 is otherwise similar to the `server` module in AFS-1. The `client` module is also similar as that for AFS-1. The only difference is that in

```
-- specification AG (Client1.belief = valid -> Server.bel... is false
-- as demonstrated by the following execution sequence
state 1.1:
    Client1.out = 0
    Client1.belief = nofile
    Client2.out = 0
    Client2.belief = nofile
    Server.out1 = 0
    Server.out2 = 0
    Server.belief1 = nocall
    Server.belief2 = nocall
    Server.validFile1 = 0
    Server.validFile2 = 0
    Env.failure1 = 0
    Env.failure2 = 0
state 1.2:
    Client1.out = fetch
    Client2.out = fetch
state 1.3:
    Server.out1 = val
    Server.out2 = val
    Server.belief1 = valid
    Server.belief2 = valid
state 1.4:
    Client1.belief = valid
    Client2.belief = valid
    Server.out1 = 0
    Server.out2 = 0
state 1.5:
    Client1.out = update
    Client2.out = update
state 1.6:
    Server.out1 = inval
    Server.out2 = inval
    Server.belief1 = nocall
    Server.belief2 = nocall
```

Figure 7: SMV Counterexample for Incorrect Invariant for AFS-2

```
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 8: SMV Environment Module for AFS2

AFS-2 a client may also send an update message to the server when it believes its file is valid. This difference is captured in the third to last line in the definition of `out` in the `client` module.

When SMV checks that the cache coherence invariant (CC') is true, it uses a total time of less than .5 seconds; the number of reachable states is 7776.

4.3 Coda+

Coda supports server replication, allowing *volumes* to be stored on a group of servers. A *volume* is a collection of files forming a partial subtree in the file name space [41]. A file is contained in exactly one volume. The servers on which a volume is stored is called the *volume storage group* (VSG). At any time, the subset of those servers available is called the *accessible volume storage group* (AVSG).

The Coda cache coherence protocol is based on the AFS-2 protocol, but with a group of servers as a repository. Clients contact all servers in the AVSG (though data is shipped from only one), and all servers maintain callbacks for objects cached from the VSG. To reason about this protocol, all that is required is a small change to the definition of a run, so that it ends when the AVSG changes. This is natural because if the AVSG shrinks, there exists the potential for a lost callback from the server that disappeared. If the AVSG grows, the additional server may hold updated versions of cached data.

Mummert's *large granularity* cache coherence protocol extends the Coda scheme. To reduce client-server communication in failure-prone environments, callbacks may be maintained on *volumes* in addition to or instead of files. A callback on a volume constitutes proof that all cached files in the volume are valid. To establish a volume callback, the client caches the version number for the volume. The server increments the volume version number whenever a file in that volume is updated.

A run of this protocol concerns a file f, and optionally the version number v from volume V containing f. Before requesting v, the client must have at least one file in V in its cache, and all cached files in V must be valid. This requirement ensures the files at C correspond to the version number it receives.

A client may validate v just as it would a file. If it has both file and volume state at the beginning of a run, it may validate them in either order. If a client validates v successfully, it receives a callback for the volume. No further communication is necessary to read any file in the volume until the callback is broken or a failure occurs.

Thus, Coda+ is complicated by the addition of a new type of cached data, called a *volume*. This addition leads to a richer state space, more state transitions, and more cases in which failures can arise. For example, whereas AFS-2 has only five classes of runs, the Coda+ protocol has fifteen. Though one could argue that analyzing AFS-1 and AFS-2 using mechanical tools like model checkers is overkill, Coda+ is large enough to warrant validation beyond pencil-and-paper analysis.

We model this cache coherence protocol by considering one client, C, one server, S, one file, f, and one volume V with version number v. The system state is again a tuple of four variables, $(C.D, C.B, S.B, A)$, where

- $C.D$ ranges over $\emptyset, \{f\}$, and $\{f, v\}$. This means if the volume version number is cached then so is a file from that volume.
- A is the agreement set on the cached objects. It ranges over the following values:

$$\emptyset, \{f_C = f_S\}, \{f_C \neq f_S\}, \{f_C = f_S, v_C = v_S\}$$
$$\{f_C = f_S, v_C \neq v_S\}, \{f_C \neq f_S, v_C \neq v_S\}$$

Note that because the volume version number is updated whenever an object in the volume is updated, it is not possible for f to be invalid and v to be valid at the same time.

A run of the protocol maps some initial state $(C.D_i, C.B_i, S.B_i, A_i)$ to some terminating state $(C.D_t,\ C.B_t, S.B_t, A_t)$. The state space is restricted in the following ways. For all states, $(C.D, C.B, S.B, A)$, in a run:

1. For each object d (f or v) in $C.D$, $d_C = d_S$ or $d_C \neq d_S$ must be in A. If $C.D = \emptyset$ then $A = \emptyset$.

2. When an object is invalidated, the client must discard it. An invalidation for the file is an implicit invalidation for the volume. More precisely,

$$(f_C \neq f_S) \in A \Rightarrow C.D_t = \emptyset$$

$$(v_C \neq v_S) \in A \Rightarrow v \notin C.D_t$$

3. At the end of a run, either d is not cached, or it is cached and agrees with the server. This follows from 2 above, because once the client discovers d is invalid it discards it. Thus $A_t = \emptyset$ or $A_t = \{d_C = d_S\}$.

4.3.1 State Machine Model

For the Coda+ model, we consider one server, two identical clients, one file, and one volume version number. To model failures, we also use an environment module that is identical to that of AFS-2.

The server now has four beliefs, two for the files cached by the clients and two for their cached version numbers. These beliefs can take the values `valid` and `nocall`. The server does not have an explicit `invalid` belief.

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

The messages sent to the server by the clients range over {`Ffetch, Fvalidate, Fupdate, Vfetch, Vvalidate, Vupdate`}. The messages that the server sends to the clients range over {`Fval, Finval, Vval, Vinval`}. Messages starting with a `V` (`F`) relate to the version numbers (files).

Figures 9 and 10 give the state transition diagrams for a Coda+ client and server. Each node is labeled by the value of the belief about the file and the value of the belief about the volume version number. Each arc is labeled by the name of the message received by the client (server).

4.3.2 SMV Input and Output for Coda+

We omit the actual SMV input and output for Coda+ since they are similar to that for AFS-1 and AFS-2 (see [42] for details). The property we check is the same cache coherence property, CC', as that checked for AFS-2 (since we need to take into account transmission delay for Coda+ too).

SMV's output for Coda+ indicates that the cache coherence invariant (CC') is true. SMV takes less than one second to check 43,684 reachable states.

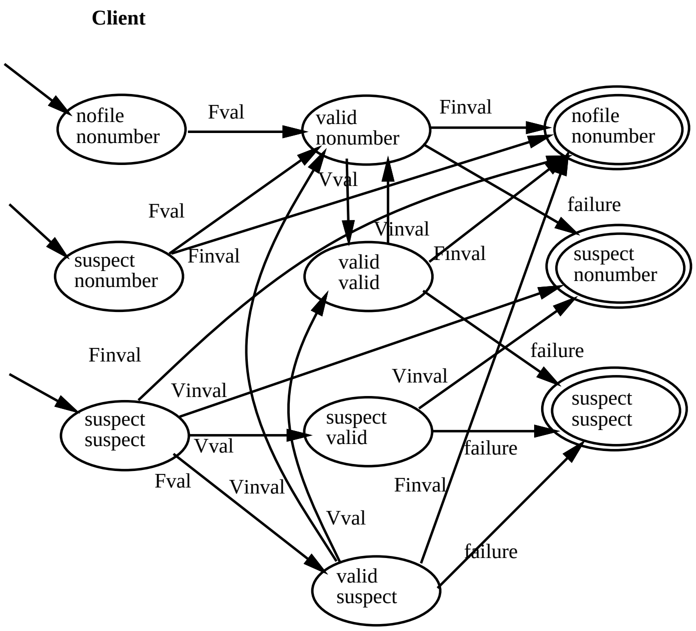

Figure 9: State Transition Diagram for Coda+ Client

5 Discussion: Different Kinds of Abstractions

What made it possible for us to use model checking in our case study is that we chose different abstraction mappings to apply to the real system. We certainly did not check the actual C code that implements AFS-1, AFS-2, or Coda+, but then the level at which we would want to verify a protocol like cache coherence is much higher than the code level; if the design is incorrect or incomplete (misses some cases) then the code is apt to reflect those mistakes and omissions. Designs, especially for distributed systems protocols, are thus good subjects for model checking.

In this section we explain in more detail some of the abstraction mappings that we applied, which can be used more generally in other settings. We exploit (1) the form of the property, ϕ to be satisfied, (2) domain-specific knowledge, and (3) problem-specific knowledge.

5.1 Exploiting the Form of ϕ

In a real instance of a distributed file system like Andrew or Coda, there are an unbounded number of files, clients, and servers. To analyze the state space of a real instance would be beyond the capability of any model checker today.

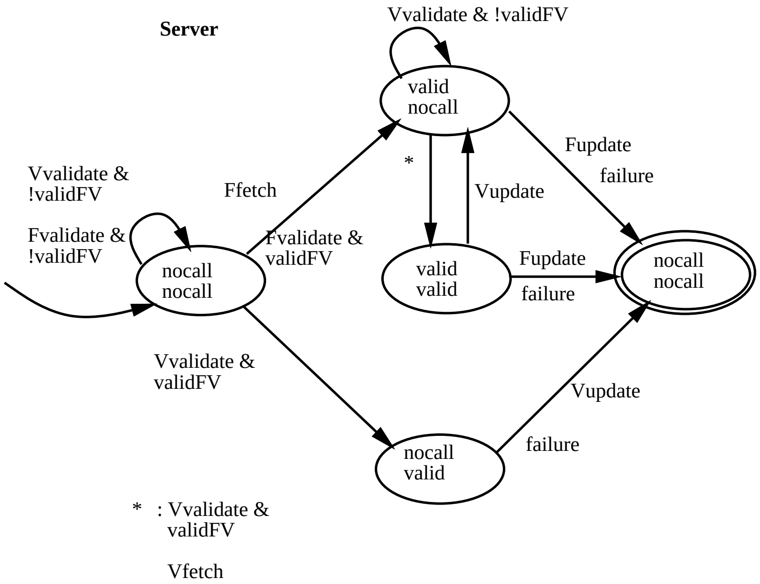

Figure 10: State Transition Diagram for Coda+ Server

If ϕ is in the form of $\forall x.P(x)$ then if for any instance, a, proving $P(a)$ will let us deduce ϕ by the generalization rule from logic:

$$\frac{P(a)}{\forall x.P(x)}$$

Our cache coherence invariant (CC) is stated as "for all clients, for all servers, for all files ...." Thus, for example, for AFS-1, we simply successively apply the generalization rule three times upon proving the property for a model with just one client, one server, and one file.

Roscoe and MacCarthy make a similar point in their work using FDR to model check *data-independent* properties of concurrent processes [36]; Wolper provides a formal justification for data-independence [43]. Dams, Grumberg, and Gerth use the idea of exploiting the form of ϕ in extending abstract interpretation to nondeterministic systems; they consider two different subsets of CTL*, universal CTL* and existential CTL* [15], where path quantifiers are restricted to be only of one kind (universal or existential).

5.2 Exploiting Domain-Specific Knowledge

In our domain of distributed systems, we need to worry about failures and transmission delays. First, we abstract from different types of failures like crashed nodes and downed communication links since our protocols act the same regardless of the type of failure. We do, however, need to model that a failure may occur; we use an environment machine (`env`) to model this possibility. We also need to model that clients and servers can detect a failure; we use the `failure1` and `failure2` parameters for this purpose.

Second, transmission delay in conjunction with failures complicate reasoning about correctness in distributed systems. During the interval in which a message is traveling, the sender may have made some state change which will render the correctness condition false until the receiver processes the message and changes its state. Transmission delay is loosely bounded by the timeout period used by the underlying communication protocol, denoted by β. It also takes time for clients and servers to detect failures. The interval between the occurrence of a failure and its detection, denoted by τ, defines a window of vulnerability. To bound the failure detection interval, clients and servers probe each other periodically, and declare failures if messages time out. Let ρ be the probe interval and assume that clients and servers use the same probe interval, but do not necessarily probe each other at the same time. Then the failure detection interval is at most $\tau = \rho + \beta$. In Coda τ is composed of a probe interval of 10 minutes and a message timeout of 15 seconds.

We abstract from the exact times that contribute to τ and model this interval as a single step in a state machine. This interpretation lets us characterize the correctness condition, CC', in terms of the next-state temporal logic operator (`AX`) rather than the original, simpler CC. If transmission were instantaneous and there were no failures, there would be no need for modeling τ since client and server beliefs could be updated simultaneously.

In effect, τ lets us represent transmission delay inherent in any distributed system. We represent τ by a single step transition in SMV since we can view this time delay as another kind of state transition effected by the environment. We did not model this environment's effect explicitly, as we did failures, but rather implicitly by exploiting SMV's model of synchronous state machines (and the implicit presence of a global clock). Thus the passage of time in SMV corresponds to the passage of time needed in the protocol. Only a single step is needed since we assume clients and servers will detect failures within τ in which case the run of the protocol ends.

Pong and Dubois exploit knowledge of their domain in choosing abstractions for verifying cache coherence protocols for shared memory multiprocessor systems [35] (a different domain from ours since we have a different model of communication and we need to deal with faults). They observe that correctness of a cache coherence protocol for those systems is not dependent on the exact number of cached copies, which in general could be unbounded. Rather, states need to keep track of whether the caches have 0, 1, or multiple copies, thereby reducing the possibilities to just three from an unbounded number. Pong and Dubois devise a *symbolic state expansion procedure* that intimately relies on this insight.

5.3 Exploiting Problem-Specific Knowledge

We use ϕ to drive the choice of "appropriate" abstractions. For example, if ϕ is a property about an integer, x, we may care only that x is negative, positive, or zero. We can define an abstraction function that maps an infinite set of values to three values. Or, if ϕ is a property about an integer set, s, we may care only about whether the set is empty or not; we may not care at all about what its elements are. We can define an abstraction function that maps an infinite set of set values to two values.

In our problem of cache coherence between clients and servers, the property ϕ that we want to verify is that the client's beliefs about the validity of cached files are consistent with a server's beliefs. We let ϕ guide us in determining what details of the real system we can safely ignore.

First, we abstract from the clients' caches, i.e., sets of files and volume version numbers. Since we have simplified our model to handle just one file, we need only model whether the cache, C, is empty or not. In the case of AFS-1 and AFS-2, $C = \emptyset$ or $C = \{f\}$; for Coda+, C can additionally be $\{f, v\}$. This abstraction gives us two (or three) cases to consider instead of an exponential number of cases (2^n) for an unbounded number, n, of files and volumes. Note also that we never have to consider the case of the cache value, $\{v\}$, since the protocol prohibits a volume version number from being cached if a file is not also cached.

Second, we need only represent the belief a client has about the cached file. We do not need to model sets of beliefs (given we have only one file or one volume version number about which to have a belief). For example, for AFS-2, in reality, we might have a cache, C, and a belief set B (for file f), that ranges over $\emptyset$ and $\{valid(f)\}$. Combined with the abstraction above, where we consider a simplification of the cache, rather than two state variables, we need only one (`belief`), and rather than four possibilities (2×2), we need represent only three. We define a (partial) abstraction function, A, that maps a pair $\langle C, B \rangle$ to a belief in $\{\texttt{valid}, \texttt{nofile}, \texttt{suspect}\}$:

$$
\begin{array}{ll}
\langle\ \{f\}, \{valid(f)\}\ \rangle & \mapsto \texttt{valid} \\
\langle\ \emptyset, \{valid(f)\}\ \rangle & \mapsto \bot \text{ (unreachable case)} \\
\langle\ \{f\}, \emptyset\ \rangle & \mapsto \texttt{suspect} \\
\langle\ \emptyset, \emptyset\ \rangle & \mapsto \texttt{nofile}
\end{array}
$$

For Coda+, we apply a similar abstraction for beliefs about the volume version number since it also can take only three possible values, `valid`, `suspect`, `nonumber`. Also, for Coda+ we do not need to consider all nine possible combinations of beliefs for the file and volume version number since two of them are impossible, again knowing that a volume version number is cached if and only if a file (in that volume) is cached. (To be pedantic, in Figure 9, only seven possible combinations of the values for the file and volume version numbers appear in the nodes.)

Third, we do not even need to model the file itself since we do not care at all about the contents (value) of the file. We would if we needed to compare the values of two different files or extract information about the file based on its value. Thus, we identify the transmission of the file with the transmission of the message about the file; for example, in AFS-2, the `val` message can be thought of as abstractly containing the file itself as well as the status about its validity.

Fourth, we abstract from the type of data cached. In the AFS protocols, the types include files and directories. For our analysis, however, that there are different types of data is completely irrelevant to the correctness of the protocol. We use the generic term "file" to stand for any kind of data. However, in Coda+, a third type, volumes, is treated differently, and thus must be modeled explicitly; as mentioned earlier, it is this new data type that complicates the Coda protocol.

Fifth, we abstract from the notion of validity, which in practice is determined by the recency of a file. Suppose as in the implementation, recency is determined by comparing the totally ordered timestamps associated with files. For any pair of timestamped files, f_{t1} and f_{t2}, we can determine whether one is more recent by comparing their timestamps, $t1 < t2$; but since this always returns true or false, we can model recency, and hence validity, as a boolean variable representing whether a file is valid or not. This abstraction appears explicitly in the way we use the `valid-file` boolean variable in the server module, letting the server nondeterministically choose between the two possibilities.

Finally, we abstract from individual runs of protocol. We consider *classes* of runs, categorized by a protocol's sets of initial and final states. For Coda+, this reduces the number of cases to consider from forty-four to fifteen (corresponding to the fifteen paths in the client state transition diagram of Figure 9) [34].

6 Related Work

Model checking originated with Clarke and Emerson's work in 1981 [9]. As mentioned in the introduction, it has already proven to be extremely successful in debugging hardware [5, 16, 32, 11, 10, 4]. Tool support for model checking includes SMV [31], FDR [19], COSPAN [22], the Concurrency Workbench [12], Murϕ [17], and Mec [2]. There are more and more documented case studies; for example, the proceedings of the 1995 Workshop of Industrial-Strength Formal Techniques contains four model checking case study papers [20].

We are not the first to explore the use of model checking in the software domain. Three other approaches complement ours and each other. Since they are all recent (dated 1993-94), we expect that over time results from one approach will carry over to the others.

- Atlee and Gannon follow a specification-language based approach [3].

They verify safety properties for event-driven systems described by the SCR tabular requirements language. Their case studies include an automobile cruise control system and a water-level monitoring system. They show how to represent any specification written in a subset of SCR as a finite state machine. They use an extended version of SMV for their model checker.

- Jackson explores the richness of types in software systems. State variables for hardware (and SCR) are of simple types like boolean, but in software they range over more complex type like sets, graphs, and relations. He exploits symmetry in mathematical relations to reduce the state space; he shows how to model check Z specifications [26], which is essentially based on his relational calculus. His Nitpick tool implements his model enumeration method [27].

- Allen and Garlan's use of model checking focuses at the level of software architecture, a level of abstraction far above the real system, but again where many design flaws can be detected. They use FDR to detect deadlocks in software architectures described in the Wright architectural description language [1]. Wright is based on a subset of CSP, and thus it leaves states completely uninterpreted.

Our approach complements all three of the above since in each case, the researchers first build some finite model of the real system and express it in terms of SCR, Z, or Wright. In doing so, they implicitly apply the kinds of abstractions we used in our examples. By restricting themselves to a specific language, they have the advantage of avoiding having to define different abstractions per problem, since they do this mapping once and for all. In our approach, we let ϕ drive the choice of abstractions and simply express our models directly in terms of SMV input. We have the advantages of bypassing the "intermediate" specification language translation step, and of not being restricted to the domain of systems that a given specification language is most suited for describing. Thus, our focus is on finding "appropriate" abstractions that work across different domains and different problems, not on checking models expressed in a particular specification language.

Cheung and Kramer's two-step analysis approach applied to reasoning about large-scale distributed systems is similar in spirit to ours [8]. They use dataflow analysis as a way to approximate a system's behavior and then contextual analysis to do an exhaustive search of the resulting state space.

Approaches to combine model checking with theorem proving include work by Hungar[25] and Kurshan and Lamport[28], and in tools such as the Stanford Temporal Prover (STeP) [30] and SRI's PVS [40][2]. Relevant to our approach of using abstraction mappings, Havelund and Shankar used PVS to justify formally

[2] April 1995 version.

an abstraction mapping they use in a case study on a bounded retransmission protocol [23]; the point behind their case study was to compare the performance of Dill's model checker Murϕ [17], SMV, and the PVS model checker on a finite abstraction of the protocol.

Finally, we could have used other, more general, proof-based approaches such as Unity [7] or I/O automata [29], than the Burrows-Abadi-Needham Logic of Authentication, to reason about our protocol examples; however, the particular proof system in which we do our reasoning is secondary to the main point of this paper. Rather, we show that we can take a finite state model such as that which was already developed (using the BAN logic) in Mummert et al. [34] and "feed it" into an existing model checker. More importantly, the point of this paper is to encourage the software engineering community to consider seriously model checking technology and tools for reasoning about software systems.

7 Conclusions and Future Directions

Model checking has the significant advantage over more traditional forms of software verification in that much of the hard work is done automatically by the machine. Moreover, both the inputs and the results of model checking tools are straightforward to understand by non-experts: since a model checker's interface is well-defined and it is straightforward to provide its expected inputs, we can readily use it as a "black box." This suggests that for gaining assurance about software, model checking can become a technology that is much more broadly accessible to practitioners than other techniques.

The choice of what abstractions to apply takes some good judgment. After all, we could define a model of a system that is so abstract that any property would be trivially satisfied or that would allow any possible concrete realization. Further research is needed to characterize more formally what makes an abstraction "good."

Another direction of further research is to devise ways to justify abstractions formally. Ideally, a property shown to be true of a finite abstraction of a system should be true of the original system. One way to justify an abstraction formally is to restrict the specification language, e.g., the approach taken by Dams, Grumberg, and Gerth. Another way is to prove that an abstraction preserves the correctness properties of interest, which is what Havelund and Shankar do using PVS, a theorem prover, in their bounded retransmission protocol example. Ironically, theorem proving is still required; the expectation is that the theorem to be proved is "smaller" than that corresponding to proving the property of the original system.

To make our reasoning more precise, we need to justify the abstractions we chose in this case study; however, as stated in the introduction, the goal of our work is to demonstrate to builders of large software systems that "approximate" reasoning by use of model checking is a low-cost, yet highly effective way of debugging system designs. Our choice of abstractions were guided entirely by either our own intuition or that of the original file system designers and implementers. Even based on this imprecise method of choosing abstractions, software engineers can use our approach to gain greater confidence in and a better understanding of their systems. More research needs to be done in developing methods and tools to enable more precise reasoning, while maintaining a low-cost to using them.

We lack a formal justification of our abstractions. To give one requires a formal model of the original protocols; this does not exist. The protocols as presented in the literature are either given a one-paragraph textual description or a lengthy textual description that reveals implementation details (like the fields in the C structs) that completely obscure how the protocol works. Indeed the first formalization of these protocols is given in the Mummert et al. paper [34], which we have reproduced in part in Sections 3.1, 4.1.1, 4.2.1, and 4.3.1. This formalization is also an abstraction of the real system.

In our own work, toward making progress both in demonstrating feasibility and in understanding characteristics of good abstractions, we plan to push on more examples. We have recently applied the approach described in this paper to validate recovery protocols [13] for redundant disk arrays, in particular for the RAID Level-5 architecture. We are just beginning to work on protocols proposed and in use for electronic commerce. Our primary goal is to provide more convincing evidence to systems designers and builders that formal reasoning tools are ready for day-to-day use. As a useful by-product, we expect to identify other kinds of abstractions appropriate to apply to real software systems.

We are optimistic about the future of the use of automated tools like model checkers to reason about software systems. One way to measure the practicality of such tools is by how easy they are to teach and learn. The second author did the model checking case study in this paper as part of her senior honor's thesis and took a graduate-level course regularly taught by Professor Clarke on model checking. In CMU's Master's of Software Engineering core course on Analysis of Software Artifacts, we have students do a series of three two-week projects using SMV, Nitpick, and FDR; students in the MSE course on Architectures of Software Systems also use FDR to analyze Wright specifications.

Acknowledgments

Jeannette Wing would like to credit Daniel Jackson for his insight as to why model checking is especially appropriate for proofs about software at the system level. Discussions with Daniel and David Garlan have clarified our optimistic views toward the use of model checking for software.

We thank Ed Clarke and indirectly Randy Bryant for providing the model checking technology that made it possible to do our case study. Mandana Vaziri-

Farahani especially thanks Xudong Zhao and Sergio Campos for their help in learning how to use SMV.

We also would like to thank the fellow "systems" faculty at CMU (like M. Satyanarayanan) who have continually challenged their formal specification and verification colleagues to demonstrate the utility of their research results on systems that "matter." That we were able to use model checking for showing Coda's cache coherence protocol correct is a step towards meeting that challenge.

Finally, we thank the anonymous referees for their perceptive comments on an earlier draft of this paper.

References

[1] R. Allen and D. Garlan. Formalizing architectural connection. In *Proceedings of the Sixteenth International Conference on Software Engineering*, May 1994.

[2] A. Arnold. Mec: a system for constructing and analysing transition systems. In J. Sifakis, editor, *Automatic verification methods of finite state systems*, volume LNCS 407. Springer-Verlag, 1989.

[3] J.M. Atlee and J.D. Gannon. State-based model checking of event driven systems requirements. *IEEE Trans. Soft. Eng.*, January 1993.

[4] G. Barrett. Model checking in practice: The t9000 virtual channel processor. *IEEE Trans. on Soft. Eng.*, 21(2):69–78, February 1995.

[5] M. C. Browne, E. M. Clarke, D. L. Dill, and B. Mishra. Automatic verification of sequential circuits using temporal logic. *IEEE Transactions on Computers*, C-35(12):1035–1044, 1986.

[6] M. Burrows, M. Abadi, and R. Needham. A logic of authentication. *ACM Transactions on Computer Systems*, 8(1):18–36, February 1990.

[7] K.M. Chandy and J. Misra. *Parallel Program Design*. Addison-Wesley, 1988.

[8] S.C. Cheung and J. Kramer. An integrated method for effective behavior analysis of distributed systems. In *Proc. 16th Int'l Conf. on Soft. Eng.*, Sorrento, Italy, May 1994.

[9] E. M. Clarke and E. A. Emerson. Synthesis of synchronization skeletons for branching time temporal logic. In *Logic of Programs: Workshop, Yorktown Heights, NY, May 1981*, volume 131 of *Lecture Notes in Computer Science*. Springer-Verlag, 1981.

[10] E. M. Clarke, O. Grumberg, H. Hiraishi, S. Jha, D.E. Long, K. L. McMillan, and L. A. Ness. Verification of the futurebus+ cache coherence protocol. In *Proc. of CHDL '93*, 1993.

[11] E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. In *Proc. of POPL '92*, 1992.

[12] R. Cleaveland, J. Parrow, and B. Steffen. The concurrency workbench: A semantics-based tool for the verification of concurrent systems. Technical Report ECS-LFCS-89-83, Edinburgh Univeristy, 1983.

[13] W. Courtright and G. Gibson. Backward error recovery in redundant disk arrays. In *Proceedings of the 1994 Computer Measurement Group Conference (CMG)*, pages 63–74, Orlando, FL, December 1994.

[14] D. Craigen, S. Gerhart, and T. Ralston. Formal methods reality check: Industrial usage. *IEEE Trans. on Soft. Eng.*, 21(2):90–98, February 1995.

[15] D.R. Dams, O. Grumberg, and R. Gerth. Abstract interpretation of reactive systems: Abstractions preserving ACTL*, ECTL* and CTL*. In *IFIP working conference on Programming Concepts, Methods and Calculi (PROCOMET'94)*, San Miniato, Italy, June 1994.

[16] D. L. Dill and E. M. Clarke. Automatic verification of asynchronous circuits using temporal logic. *IEE Proceedings*, Part E 133(5), 1986.

[17] David L. Dill, Andreas J. Drexler, Alan J. Hu, and C. Han Yang. Protocol verification as a hardware design aid. In *IEEE International Conference on Computer Design VLSI in Computers and Processors*, pages 522–525, 1992.

[18] A.R. Flora-Holmquist and M.G. Staskauskas. Formal validation of virtual finite state machines. In *Proceedings of the Workshop on Industrial-Strength Formal Specification Techniques*, April 1995. to appear.

[19] Limited Formal Systems Europe. *FDR: User Manual and Tutorial.* Oxford, England, October 1992.

[20] R. France and S. Gerhart. *Proceedings of the Workshop on Industrial-Strength Formal Specification Techniques.* IEEE, 1995.

[21] J.Y. Halpern and Y. Moses. Knowledge and common knowledge in a distributed environment. In *Proceedings of the Third ACM Symp. on Principles of Distributed Computing*, pages 50–61, August 1984.

[22] Z. Har'El and R. P. Kurshan. Software for analytical development of communications protocols. *AT&T Technical Journal*, 69(1):45–59, Jan.–Feb. 1990.

[23] Klaus Havelund and Natarajan Shankar. Experiments in theorem proving and model checking for protocol verification. Submitted to FME '96, 1996.

[24] John Howard, Michael Kazar, Sherri Menees, David Nichols, Mahadev Satyanarayanan, Robert Sidebotham, and Michael West. Scale and Performance in a Distributed File System. *ACM Transactions of Computer Systems*, 6(1):51–81, February 1988.

[25] H. Hungar. Combining model checking and theorem proving to verify parallel processes. In *Proc. 5th Int'l. Conf. on Computer Aided Verification*, volume LNCS 697, pages 154–165. Springer-Verlag, 1993.

[26] D. Jackson. Abstract model checking of infinite specifications. In *Proc. of FME '94*, October 1994.

[27] Daniel Jackson. Nitpick: A checkable specification language. In *Proc. Workshop on Formal Methods in Software Practice*, San Diego, CA, January 1996.

[28] R. Kurshan and L. Lamport. Verification of a multiplier: 64 bits and beyond. In *Proc. 5th Int'l. Conf. on Computer Aided Verification*, volume LNCS 697, pages 166–179. Springer-Verlag, 1993.

[29] N. Lynch and M. Tuttle. Hierarchical correctness proofs for distributed algorithms. Technical report, MIT Laboratory for Computer Science, Cambridge, MA, April 1987.

[30] Z. Manna. Beyond model checking. In *Proc. 6th Int'l. Conf. on Computer Aided Verification*, volume LNCS 818, pages 220–221. Springer-Verlag, 1994.

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

[32] K. L. McMillan and J. Schwalbe. Formal verification of the gigamax cache consistency protocol. In N. Suzuki, editor, *Shared Memory Multiprocessing.* MIT Press, 1992.

[33] L. Mummert and M. Satyanarayanan. Large Granularity Cache Coherence for Intermittent Connectivity. In *USENIX Summer Conference Proceedings*, pages 279 – 289. USENIX Association, June 1994.

[34] L. Mummert, J.M. Wing, and M. Satyanarayanan. Using belief to reason about cache coherence. In *Proceedings of the Symposium on Principles of Distributed Computing*, August 1994. Also CMU-CS-94-151, May 1994.

[35] Fong Pong and Michel Dubois. A new approach for the verification of cache coherence protocols. *IEEE Trans. Parallel and Distributed Systems*, 6(8), August 1995.

[36] A.W. Roscoe and H. MacCarthy. A case study in model-checking CSP. submitted for publication, October 1994.

[37] M. Satyanarayanan, J.J. Kistler, P. Kumar, M.E. Okasaki, E.H. Siegel, and D.C. Steere. Coda: A highly available file system for a distributed workstation environment. *IEEE Transactions on Computers*, 39(4), April 1990.

[38] Mahadev Satyanarayanan, John Howard, David Nicols, Robert Sidebotham, Alfred Spector, and Michael West. The ITC Distributed File System: Principles and Design. In *The Tenth ACM Symposium on Operating Systems Principles*, pages 35–50. ACM, December 1985.

[39] Fred B. Schneider. What good are models and what models are good? In Sape Mullender, editor, *Distributed Systems*, New York, 1993. Addison-Wesley Publishing Company. pp. 20-21.

[40] N. Shankar, S. Owre, and J. Rushby. The pvs proof checker: A reference manual (draft). Technical report, Comp. Sci. Lab. SRI International, Menlo Park, CA, 1993.

[41] R.N. Sidebotham. Volumes: The Andrew File System data structuring primitive. In *European Unix User Group Conference Proceedings*, August 1986. Also available as technical report CMU-ITC-053, Carnegie Mellon University, Information Technology Center.

[42] Mandana Vaziri-Farahani. Using symbolic model checking to verify cache coherence in a distributed file system. Technical Report CMU-CS-95-156, Carnegie Mellon Computer Science Department, 1995. CMU Electrical and Computer Engineering Bachelor's Thesis.

[43] P. Wolper. Expressing interesting properties of programs in propositional temporal logic. In *Proc. of Principles of Programming Languages*, pages 184–193, 1986.