Application of Epistemic Logic Methods to Fault-Tolerance and Program Recovery

Alexander Pokluda

CS745 Computer Aided Verification Research Project

University of Waterloo

apokluda@uwaterloo.ca

December 17, 2012

Abstract

The most important application of formal methods is automated program verification. However, if an error is identified when a program is verified against its specification, a cycle of manual revision and verification must be performed. In some situations, the correct revision of a program (that is, all existing properties of the input program are preserved) is critical. The automated synthesis of fault-tolerant program from a fault-intolerant one is particularly well suited for this case. This report first introduces the reader to epistemic logic and uses the Muddy Children Puzzle*, also known as the* Cheating Husbands Problem*,* Unfaithful Wives Problem *or* Josephine's Problem*, to reinforce epistemic logic concepts and then discusses how temporal epistemic modal logic can be applied to the verification of fault-tolerance and recovery properties. Throughout the report, the* Byzantine Generals Problem*, a classic fault-tolerance example from the literature is discussed. The report concludes with a discussion of related and future work.*

Table of Contents

List of Figures

1 Introduction and Motivation

AUTOMATED program verification–the act of proving or disproving the correctness of a program relative to a specification–is arguably the most important contributions of the field of formal method to date. However, verification is an after-the-fact task. A program that has been manually constructed can only be verified against its specification once the implementation is thought to be complete. (If the implementation is known to be incomplete, it will naturally fail the verification step). If the program fails to satisfy its specification during verification, then it must be manually revised and re-verified. This situation naturally leads to a cycle of manual construction/revision and verification. This cycle can be quite costly both in terms of human and computational power.

In addition to human error during program construction, other factors may also require that a program go through a cycle of revision and verification. For example, it is often the case that the specification for a program will change before or after construction of the program is complete. Or, a change in the environment in which the program operates could require that a program be revised and re-verified. Industrial control software could be deployed in a different environment that gives different sensor values in different ranges for instance.

Clearly a better solution than manual program revision and verification would be one in which the revision step is automated together with the verification step. In this scenario, the revision process would output a new program based on the input program that preserves all the existing properties of the input program and a set of new properties that ensure the specification is never violated. In fact, the output of the revision step would not need to be verified because it would be *correct by construction*.

Taking this idea to the extreme leads to program synthesis from specification, where a program is constructed from scratch based on a specification. Program revision, on the other hand, transforms an input program into an output program that is guaranteed to satisfy its specification. The automated synthesis of distributed fault-tolerant programs from intolerant ones is discussed in depth in [2]. In particular, the authors present a symbolic algorithm that adds masking fault-tolerance to distributed programs and introduce the tool SYCRAFT that is able to generate synthesized programs that provide masking fault-tolerance. Intuitively, a program provides masking fault-tolerance if it is able to recover from a finite number of faults and continue to run correctly, meeting its specification.

The original goal of this project was to reduce the complexity of creating a masking fault-tolerant program from a fault intolerant one by applying epistemic modal logic, also known as the logic of knowledge, to the synthesis process. While this goal has not been fully met yet (the full realization of this goal is left as future

work), this report presents important backrgound information and demonstrates how properties of a program can be verified agaist specifications defined using epistemic modal logic formulae.

Section 2 introduces the reader to epistemic logic, which is a form of logic that allows agents to reason about the world around them. Sections 3, 4 and discuss how epistemic logic may be applied to the verification of program specifications. Section 5 provides a brief overview of distributed programs and specifications as presented in [2]. The symbolic synthesis algorithm presented is also discussed briefly. The Byzantine Generals Problem, as it was originally presented in [6], is used throughout this report and discussed in Section 6. An implementation of the Byzantine Generals Problem using masking fault-tolerance is then discussed in Section 7. Section 8 discusses the results of using epistemic modal logic to verify the fault-tolerance and recovery properties of the Byzantine Generals Problem. Section 9 discusses related work and finally section 10 concludes with a brief summary and a discussion of future work.

2 An Introduction to Epistemic Logic

Epistemology, the study of knowledge sometimes referred to as the *theory of knowledge* has a long history dating back to Ancient Greece. It is a branch of philosophy that is concerned with the nature and limitations of knowledge. *Epistemic logic* is a relatively modern field of study that emerged in the 1950's and flourished through the 1960's in the philosophy community [3].

Researchers in scientific fields have recently become interested in the subject as well. The scientists are interested in practical and concrete applications of reasoning about knowledge, but many of the issues that concerted the philosophers concern the scientists also. Epistemic logic has been applied successfully in many areas of theoretical computer science. A distributed system consisting of processes communicating over a computer network can be modelled in epistemic logic using the general model of a multi-agent system. This is one of its major areas of application; however, to this author's knowledge, little work has been done in applying epistemic logic the verification of fault-tolerance and recovery properties, which will be discussed in following sections. First, however, we will discuss a model for epistemic logic and epistemic modal logic formulae and apply these concepts to an example problem known as the Muddy Children Puzzle.

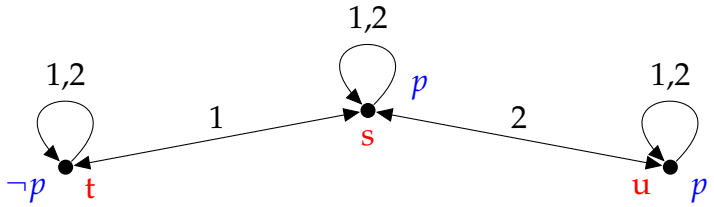


Figure 1: *A simple* Kripke *structure*

2.1 A Model for Epistemic Logic: Possible Worlds Framework

Epistemic logic is often modelled in a *possible worlds* framework. In this framework, an agent may not be able to tell which of a number of possible worlds describes the actual state of affairs given her current information. For example, suppose Alice is standing on a street in Waterloo, Ontario, and she observes that it is a beautiful and sunny day. From Alice's perspective at that moment, it is a sunny day in Waterloo in all the worlds that she considers possible. An agent is said to *know* a fact φ if φ is true in all world that she considers possible. Thus we can say that Alice *knows* that it is sunny in Waterloo.

On the other hand, however, when Alice is standing on the street in Waterloo she has no information about the current weather in Vancouver, British Columbia. From her perspective at that moment, she considers a world possible in which it is sunny in Waterloo and raining in Vancouver and another world possible in which it is sunny in both Waterloo and Vancouver. Since it is not sunny in Vancouver in all the worlds that Alice considers possible, we say that Alice *does not know* that it is sunny in Vancouver.

We can represent the notion of possible worlds graphically using *Kripke* structures. A simple Kripke structure that could correspond to the example just discussed is shown in Figure 1. The statement "It is sunny in Vancouver" is represented by the proposition letter p, Alice is represented by agent 1 and three possible worlds are represented by the states s, t and u. Let agent 2 represent Alice's cousin Bob, who conveniently happens to be standing on a street in Vancouver at the same instant. From the figure we can see that if Alice is in state s, she considers two worlds possible: one in which p is true (that is, it is sunny in Vancouver) and another one in which $\neg p$ is true (that is, it is not sunny in Vancouver) because she cannot distinguist the state s from u. However, Alice knows that Bob knows the true situation because in both the worlds that Alice considers possible, Bob knows whether p or $\neg p$ is true.

By contrast, in state s Bob knows that it is sunny in Vancouver but he does not know that Alice does not know this. In state s, Bob also considers the world described by state u possible, but in both cases p is true, therefore he *knows* that p

is true. He does not know that Alice does not know this fact, because in state s, Alice also considers the word described by state t possible (even though Bob does not), in which $\neg p$ is true.

2.2 Epistemic Modal Logic

Suppose that agents $1, ..., n$ wish to reason about a non-empty set Φ of propositions $p, q, ...$ To express *knowledge* we use the modal operators $K_1, ..., K_n$. The expression $K_1 p$ is read as "Agent 1 knows p."

We can construct formulae using the primitive propositions from Φ, the logical operators $\neg$ and $\wedge$, and the modal operators $K_1, ..., K_n$. In addition, we use the standard abbreviations[1] from propositional logic: conjunction $\varphi \vee \phi$, implication $\varphi \Rightarrow \phi$, and bi-conditional $\varphi \Leftrightarrow \phi$. We also use *true* as an abbreviation of some fixed propositional tautology such as $p \vee \neg p$, and $false$ to be an abbreviation of $\neg true$. We use the abbreviation Ep to denote $K_1 p \wedge ... \wedge K_n p$. That is, E_p denotes "Everyone knows p."

Using this language, we can express complicated statements in a straight forward way. For example, the formula

$$K_1 K_2 p \wedge \neg K_2 K_1 K_2 p$$

says that "Agent 1 knows that agent 2 knows p, and agent 2 does not know that agent 1 knows that agent 2 knows p."

Two more important concepts from epistemic modal logic are *common* and *distributed* knowledge. Common knowledge of a proposition p holds when everyone knows p, everyone knows that everyone knows p, everyone knows that everyone knows that everyone knows p and so on. Intuitively, p is common knowledge among the agents $1, ..., n$ if each knows that p holds and is confident that all that other agents in the system know that p holds. For example, in our society, it is common knowledge that a red traffic signal means stop and a green traffic signal means go.

Distributed knowledge means that together a number of agents could pool their individual information to deduce that a fact q holds. For example, suppose that Alice knows that Charlie had either oatmeal or toast for breakfast and Bob knows that Charlie did not have oatmeal for breakfast. Together, Alice and Bob have distributed knowledge that Charlie had toast for breakfast.

[1] We refer to these logical operators as abbreviations because the set $\{\neg, \wedge\}$ forms a functionally complete set from which all other logical operators can be defined.

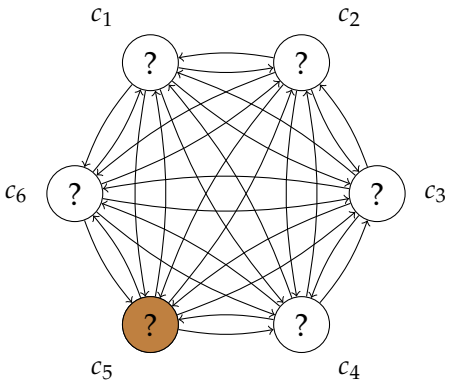


Figure 2: *Base Case: 1 Muddy Child*

2.3 The Muddy Children Puzzle

We now reinforce the concepts of epistemic logic discussed so far with a puzzle known as the Muddy Children Puzzle. A number of children (two or more) have been outside playing and while they were playing one or more of them got mud on their foreheads. When the children come inside, the father makes each child figure out whether or not they have mud on their forehead. (Presumably if the children can do this correctly, they will all get candy or some other reward). When they come inside, each child can see each other's forehead, but cannot tell whether he or she has mud on his or her own forehead. The father states "At least one of you have mud on your forehead" and then asks "Do any of you know if you have mud on your forehead?" The father asks this question repeatedly until all children with muddy foreheads answer "Yes, I have mud on my forehead." Assuming that all the children are intelligent, perceptive and truthful, how do they manage to figure this out?

Let us consider first the case that one child got mud on his forehead when playing outside. This case is depicted graphically in Figure 2–there are six children in total and child c_5 has a muddy forehead. Figure 3 shows what happens after the father asks his question. In part 3a we see that child c_1 can see that one other child has mud on his forehead, therefore he does not know whether or not he has mud on his own forehead. In part 3b we see that the situation is the same for child c_3. The situation is also the same for children c_2, c_4 and c_6; however this is not depicted in the figure. In part 3c, we see that child c_5 cannot see another child with a muddy forehead and therefore concludes that his own forehead must be muddy. At the end of the first round, child c_5 answers "Yes, I have mud on my forehead." This is depicted in part 3d. In this case it is easy to see that child c_5 saw no other child with a muddy forehead, and therefore concludes that is own

forehead must be muddy.

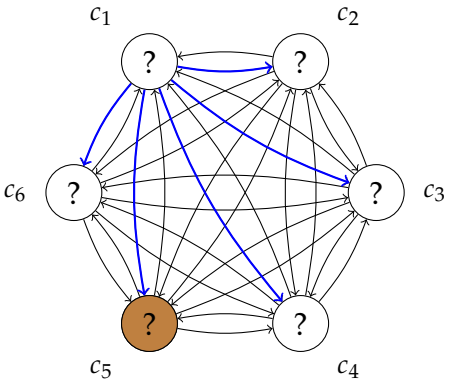
Now consider the case in which there are two children with muddy foreheads after playing outside. When they come inside, the father states "At least one of you have mud on your foreheads" and then asks "Do any of you know if you have mud on your forehead?" The situation after the father asks this question for the first time is shown in Figure 4. In parts 4a-4c we see that all the children can see at least one other child with a muddy forehead. Therefore, they cannot conclude anything about their own. From this point forward, we will call it a "round" each time that the father asks the question, the children each reason about their own forehead using the information that they have gathered and answer the father's question. At the end of the first round, none of the children know whether or not their own forehead is muddy. This is shown in Figure 4d. Although the none of the children know whether or not their own forehead is muddy yet, they do gain some important information that will become clear shortly.

During the first round, child c_3 considered two worlds possible: one in which c_5 was the only child with a muddy forehead and one in which c_5 and herself were the only ones with muddy foreheads. However, at the end of the first round, when none of the children knew whether or not they had mud on their foreheads, and in particular when c_5 did not answer "Yes," c_3 learned that c_5 must have seen another child with a muddy forehead. According to c_3's knowledge, the only possible world in which c_5 could have seen another child with a muddy forehead is the world in which she has mud on her own forehead. Child c_5 reasons in the same manner.

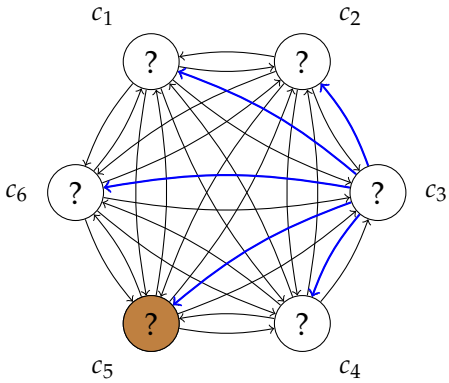
The second round is depicted in Figure 5. Although the children do not see anything different in parts 5a-5c, they gained knowledge in the first round that can help them to reason about whether or not they have mud on their own foreheads. At the end of the second round, children c_3 and c_5 conclude that they have mud on their own foreheads and answer "Yes" to their father's question.

There is an important subtlety at play that we cannot overlook: The fact that the father openly stated that at least one of the children have mud on their foreheads makes this fact common knowledge. Had the father not made this statement openly and publicly, the reasoning presented above does not work. For example, if the father took each one of the children aside and told him or her that at least one child has mud on his or her forehead, each child would not know that all the others know this, would not know that all the others know that all the others know, and so on.

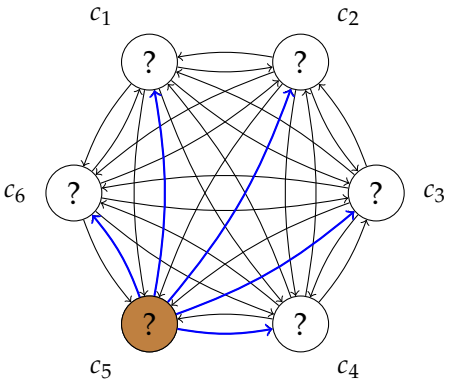
The Muddy Children Puzzle is also known as the *Cheating Husbands Problem*, *Unfaithful Wives Problem* or *Josephine's Problem*. In each of these other variations, the description of the problem is different, but the solution is the same. For example, in Josephine's Problem, every woman in Josephine's kingdom knows about the



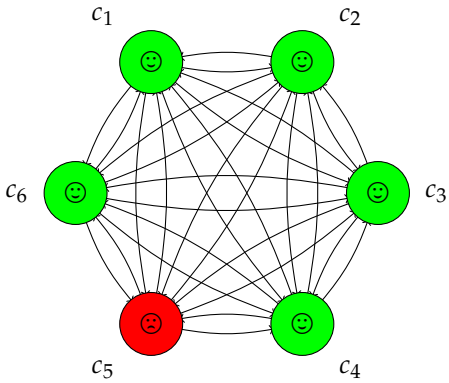
(a) *Child c_1 can see a child with a muddy forehead, therefore he does not know if his own forehead is muddy.*



(b) *Child c_3 can also see a child with a muddy forehead, therefore she does not know if her own forehead is muddy.*

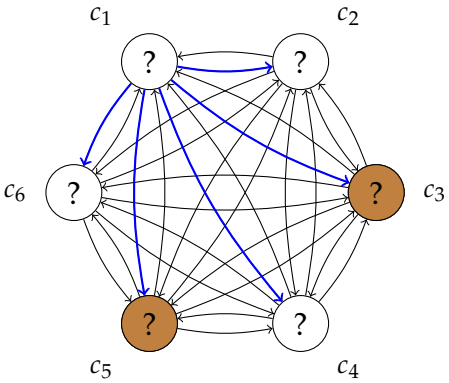


(c) *Child c_5 cannot see another child with a muddy forehead, therefore he knows that his own forehead is muddy.*

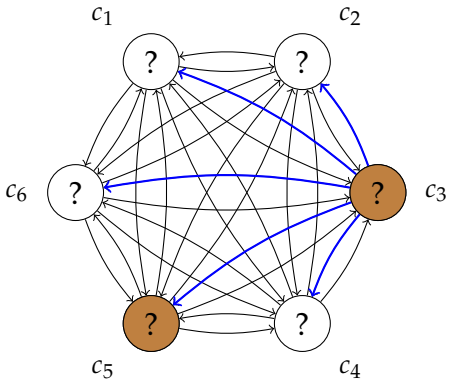


(d) *At the end of the first round, child c_5 answers "yes, I have a muddy forehead" and then the other children know that their foreheads are clean.*

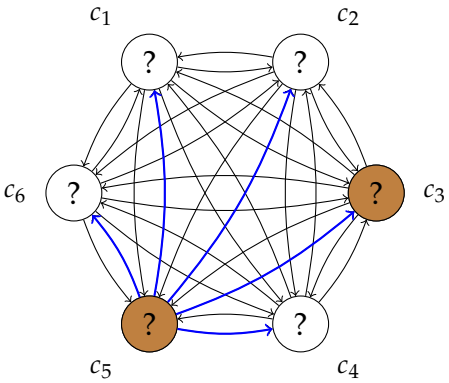
Figure 3: *The first round of the base case with one muddy child.*



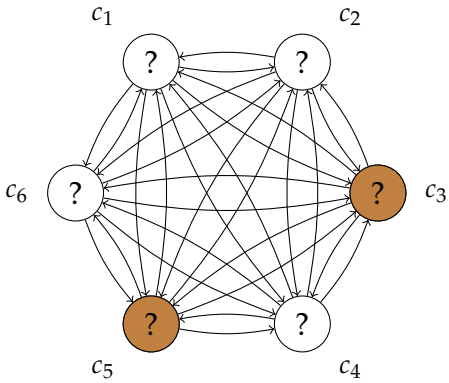
(a) *Child c_1 can see two children with muddy foreheads, therefore he does not know if his own forehead is muddy.*



(b) *Child c_3 can also see a child with a muddy forehead, therefore she does not know if her own forehead is muddy.*

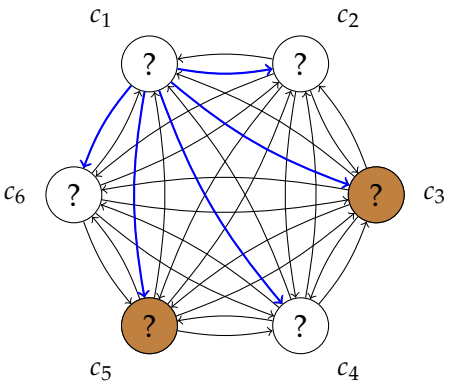


(c) *Child c_5 can also see a child with a muddy forehead, therefore he does not know if his own forehead is muddy.*

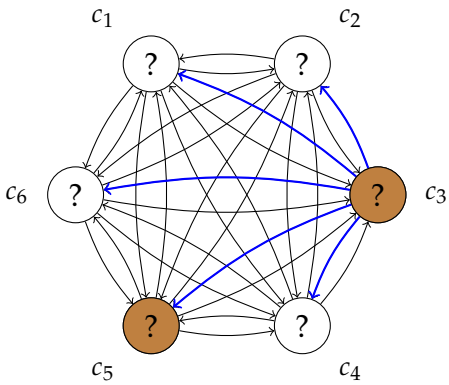


(d) *All children can see at least one other child with a muddy forehead, therefore none of them know if their own forehead is muddy at the end of the first round.*

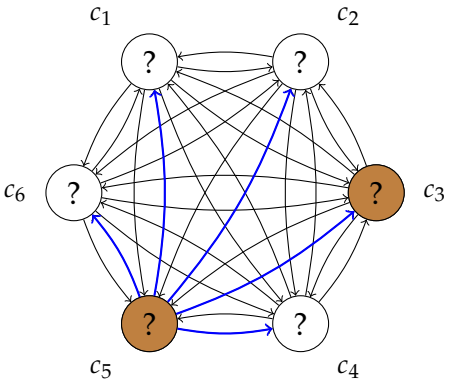
Figure 4: *The first round of the inductive case with two muddy children.*



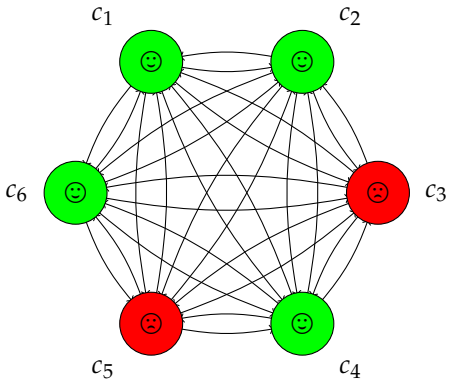
(a) *Child c_1 can see two children with muddy foreheads, therefore he does not know if his own forehead is muddy.*



(b) *In the first round, child c_3 considered it possible that child c_5 saw no other child with a muddy forehead. When c_5 did not answer "yes" at the end of the first round, c_3 discovered that this was not the case. Therefore, she reasons that there must be at least one other child with a muddy forehead, and since she cannot see another child with a muddy forehead, she reasons that her own forehead must be muddy.*



(c) *Child c_5 follows the same reasoning as c_3, and concludes that his forehead is muddy also.*



(d) *At the end of the second round, children c_3 and c_5 simultaneously answer, "yes, I have a muddy forehead" and then the other children know that their foreheads are clean.*

Figure 5: *The second round of the inductive case with two muddy children.*

fidelity of every woman's husband *except* her own. Queen Josephine states that there is at least one unfaithful husband in the kingdom and every woman is required to shoot her husband at midnight following the day she discovers that he is unfaithful.

This and the previous sections demonstrated the syntax and semantics of epistemic modal logic. In the sections that follow we discuss a model for distributed computer programs and how temporal epistemic modal logic can be applied to the verification of fault tolerance and recovery properties.

3 Problem Description

In imperative programming languages a process or thread take actions based on the results of tests applied to the program's local state. We call these types of programs *standard programs*. Standard programs, however, cannot easily express the relationship between knowledge and action that we would often like to capture.

For example, let us consider the Muddy Children Problem again. The children are asked by their father if they know whether or not they have mud on their forehead and if a child knows that he has mud on his forehead then he should answer "yes;" otherwise he should answer "no." If we let the proposition p_i represent "child i has mud on his or her forehead" then we can think of each child as the following program from [3]:

case of
  if $childheard_i \wedge (K_i p_i \vee K_i \neg p_i)$ **do** say "Yes"
  if $childheard_i \wedge \neg K_i p_i \wedge \neg K_i \neg p_i$ **do** say "No"
end case

In addition to expressing the relationship between knowledge and action, epistemic modal logic can express certain problems in computer science more succinctly than standard approaches. This is a new an exciting area of computer science and is getting a lot of attention in the research community. The goal of this project is to apply concepts from epistemic logic to the verification of fault-tolerance and recovery properties for a fault-tolerant program that has been synthesized from a fault-intolerant one. The application of epistemic modal logic to the process of synthesizing a fault-tolerant program is discussed as future work.

Operator	Description
F f	eventually f
G f	always f
f **U** g	f until g
f **R** g	f release g
X f	f in the next state

Table 1: *Temporal operators available for use in MCK epistemic logic specifications*

4 Method

The goal of verifying fault-tolerance and recovery properties of a synthesized fault-tolerant program has been achieved by modelling a synthesized fault-tolerant program based on the Byzantine Generals Problem in the model checker MCK. MCK is a prototype model checker for *temporal epistemic specifications* [14]. That is, the specification language in MCK combines epistemic modal logic with linear temporal logic. Table 1 shows the temporal operators that are available for use in MCK specifications. Programs are defined as a set of protocols operating in a common environment. Each running instance of a protocol represents an agent in the system.

In MCK, agents make observations about the state of the environment. They may use their observations in a variety of ways to determine what they *know*. One way that corresponds to the observational interpretation of knowledge, is to make inferences about the state based on their last observation. Another way that corresponds to the clock interpretation of knowledge is to make inferences about the state based on their last observation and the current clock value. Yet a third way that corresponds to the perfect recall interpretation of knowledge is to make inferences based on their last observation and clock value and all previous observations with clock value. When we use temporal epistemic logic specifications in section 8 to verify fault-tolerance and recovery properties, we use the observational interpretation of knowledge.

Section 5 describes a model for distributed programs and the synthesis problem in detail. Section 6 discusses the abstract Byzantine Generals Problem and section 7 presents a solution to the Byzantine Generals Problem using the framework from section 5 and a model of the solution in MCK. Section 8 presents the results of verifying three properties that prove the correctness of the synthesized Byzantine Generals Problem solution.

5 A Representation for Distributed Programs and Specifications

Formally we model a distributed program $\mathcal{P}$ as a set of processes $\Pi_{\mathcal{P}}$. Each process is defined by a transitions system and constrained by read/write restrictions over its set of variables $V = \{v_0, v_1, ..., v_n\}$. A specification $SPEC$ is a set of infinite computations that specify the correct behaviour of the system. The specification $SPEC$ can also be represented as an invariant I that is a set of valid states.

The problem of synthesizing a fault-tolerant program from a fault-intolerant one is as follows: Given a fault-intolerant program $\mathcal{P} = \langle \Pi_{\mathcal{P}}, I_{\mathcal{P}} \rangle$, a set F of faults and a specification $SPEC$, synthesize a program $\mathcal{P}'$ such that $\mathcal{P}'$ is F-tolerant to $SPEC$ from $I_{\mathcal{P}'}$. The synthesis method used must obtain $\mathcal{P}'$ from $\mathcal{P}$ by adding fault-tolerance to $\mathcal{P}$ without introducing new behaviours in the absence of faults.

We now briefly summarize the algorithm for synthesizing a masking fault-tolerant program $\mathcal{P}'$ from a fault-intolerant one $\mathcal{P}$ presented in [2]. The input to the program is a fault-intolerant program, a safety specification (a set of bad transitions/computations that should not happen), and a set of fault transitions. The output of the algorithm is a fault-tolerant program.

The first step is initialization. State and transition predicates from where the executions of fault transitions may cause the execution to violate the safety specification are identified. A *fault-span* is computed in the second step that is the set of all reachable states of the program in the presence of faults. The third step eliminates transitions that lead to the violation of the safety specification. The fourth step resolves deadlock states to ensure that the liveness properties of the program are satisfied and the fifth step recomputes the program invariant so that it is closed in the final output program. Steps 2-3, 2-4 and 2-5 are repeated until a fixpoint is reached–that is, the algorithm repeats until no more progress is possible and then terminates.

In the next two sections, we discuss the Byzantine Generals Problem and how this algorithm can be applied to the problem in order to solve the Byzantine Generals Problem.

6 The Byzantine Generals Problem

The Byzantine Generals Problem was first published by Lamport, Shostak and Pease in 1982 [6]. Since then, the abstract problem and variations of it have become a standard example used throughout the literature. Reliable computer systems must be able to handle a variety of errors from malfunctioning

parts of the system, including the case in which a failing component sends conflicting information to different parts of the system. This is described abstractly by the Byzantine Generals Problem. The following is a summary of the problem as it was originally proposed.

Imagine several divisions of the Byzantine army camped around an enemy city. Each division of the army is commanded by a general. The generals may each have some information gathered by observing the enemy that helps them predict whether an attack would be successful or not, and together they must reach an agreement whether to attack or retreat. The generals may communicate only by messenger–there is no broadcast facility. Furthermore, one or more of the generals might be traitors and try to confuse the others. The problem is to develop an algorithm that ensures that the loyal generals all agree to either attack or retreat. Specifically, the algorithm must meet the following two conditions:

A *All loyal generals decide on the same plan of action*

The algorithm must guarantee that the loyal generals decide on the same plan of action, while the traitors may do whatever they wish. The loyal generals must agree on a reasonable action despite what the traitors do. This requirement motivates the second condition.

B *A small number of traitors cannot cause the loyal generals to adopt a bad plan*

Each general sends his decision to each of the other generals. The first condition can be satisfied by having each of the generals combine the decision that he received from the others in a uniform way. The second condition is hard to formalize, but we can think of it like this: if the majority of the loyal generals decide to attack, a small number of traitors cannot cause them to decide to retreat. In order to satisfy this condition, the method that the generals use to come to a decision must be robust. For example, if the generals use a majority vote to decide then a small number of traitors cannot affect the decision unless the vote among the loyal generals is nearly evenly split. The problem with this solution, however, is that a traitorous general may not send the same $d(i)$ value to each of the $n-1$ generals.

Various formulations of the problem and solutions exist. In the next section we look at the variation presented in [2].

7 Byzantine General Problem Implementation

In the version of the Byzantine Generals Problem presented in [2] there is a commanding general g and three lieutenant generals j, k and l represented as

processes where at most one of the processes can be byzantine (faulty). If the commanding general is not byzantine, then the non-byzantine lieutenant generals must follow the commander's decision otherwise the lieutenant general processes must reach an agreement on their own.

The commanding general only provides a decision while the lieutenant generals receive a decision from the commanding general and the other lieutenant generals. The commanding general is represented by a decision variable d that can have a value of either 0 or 1. The lieutenant generals are represented by a decision variable that can have a value of either 0, 1 or $\perp$, where $\perp$ means that the lieutenant has not yet received a decision value from the commander. Each lieutenant general process also maintains a Boolean variable f that indicates whether or not that process has finalized its decision. A process is free to change its decision until it is finalized. Additionally, a Boolean variable b for each process indicates whether or not that process is byzantine.

The fault-intolerant version of byzantine agreement from [2] works as follows. Each lieutenant process copies the commanding general's decision and then finalizes (outputs) that decision. The transitions that a lieutenant general process, say j, can take are specified by the following actions:

$$\mathcal{BA}1_j :: (d_j = \perp) \wedge (f.j = false) \rightarrow d.j := d.g;$$
$$\mathcal{BA}2_j :: (d_j \neq \perp) \wedge (f.j = false) \rightarrow f.j := true;$$

7.1 Addition of Masking Fault Tolerance

In [2] the authors present a tool SYCRAFT that is able to synthesize fault-tolerant programs from fault-intolerant ones using the algorithm discussed in section 5. The following shows the output of SYCRAFT when run on the Byzantine Generals Problem program $\mathcal{BA}$ from the previous section:

$$
\begin{aligned}
&\mathcal{BA}'1_j :: d.j = \bot \wedge f.j := false \\
&\qquad \rightarrow d.j := d.g; \\
&\mathcal{BA}'2_j :: d.j \neq \bot \wedge f.j = false \wedge (d.k = \bot \vee d.k = d.j) \wedge \\
&\qquad (d.l = \bot \vee d.l = d.j) \wedge (d.k \neq \bot \vee d.l \neq \bot) \\
&\qquad \rightarrow f.j := true; \\
&\mathcal{BA}'3_j :: d.j = 1 \wedge d.k = 0 \wedge d.l = 0 \wedge f.j = false \\
&\qquad \rightarrow d.j, f.j := 0, false|true; \\
&\mathcal{BA}'4_j :: d.j = 0 \wedge d.k = 1 \wedge d.l = 1 \wedge f.j = false \\
&\qquad \rightarrow d.j, f.j := 1, false|true; \\
&\mathcal{BA}'5_j :: d.j \neq \bot \wedge f.j = false \wedge ((d.j = d.k \wedge d.j \neq d.l) \vee \\
&\qquad (d.j = d.l \wedge d.j \neq d.k)) \\
&\qquad \rightarrow f.j := true;
\end{aligned}
$$

In the output program, the action $\mathcal{BA}'1$ is unchanged, while the actions $\mathcal{BA}'3$ and $\mathcal{BA}'4$ are recovery actions and the actions $\mathcal{BA}'2$ and $\mathcal{BA}'5$ are strengthened actions. This version of the program is able to recover from or *mask* byzantine faults in which a faulty process may change its decision arbitrarily. (This is like a traitorous general sending different values for its decision to the the other generals).

The next sections present a model of the program $\mathcal{BA}'$ and then verify its fault tolerance and recovery properties using temporal epistemic modal logic.

7.2 A Model of the Byzantine Generals Problem

Listing 1 shows a model of the Byzantine Generals Problem. This model is based on the fault-intolerant and synthesized fault-tolerant versions of the program in [2]. In the model, any single process may start byzantine or may become byzantine during a future round. A "fault counter" is used to restrict the number of faults that may occur to a finite number. (If an infinite number of faults occur, program recovery is not possible). The current decision for each process is maintained by a variable d_p for $p \in g, j, k, l$. The decision variable can have any of the values Undec, Zero, or One corresponding to the values $\bot$, 0, 1 respectively.

The actions that a lieutenant general process can take in the fault-intolerant program presented in section 7 are defined on lines 119-124. (These actions have been commented out in listing 1). Lines 126-143 define the actions taken by the lieutenant general processes in the fault-tolerant version.

Lines 62-73 contain the invariant for the synthesized fault-tolerant program from [2]. Lines 78-81, 85-91 and 95-98 are specifications that verify that the final decisions of the commanding general and lieutenant generals meet the requirements from section 7. Specifically, if the commanding general is not byzantine then the non-byzantine lieutenant generals must follow the commander's decision, otherwise the lieutenant general processes must reach an agreement on their own. The temporal epistemic specification starting on line 78 verifies that if a lieutenant general process knows that it is not byzantine and has made its final decision and the commander is not byzantine, then its decision agrees with the commanding general's decision, if the commander is not byzantine. The temporal epistemic specification starting on line 85 verifies that a lieutenant general process knows that its decision agrees with a second process if both processes are not byzantine. Finally, the specification starting on line 95 verifies that the commanding general process knows that the non-byzantine lieutenants will follow its decision if it is not byzantine.

```
1 -- Byzantine Generals Problem
2 --
3 -- This is a model to a solution of the Byzantine Generals Problem.
4 -- The model is based on the program from [1] that used
5 -- a symbolic synthesis approach to add fault-tolerance to
6 -- a fault-intolerant program.
7 --
8 -- [1] Borzoo Bonakdarpour, Sandeep Kulkarni, and Fuad Abujarad.
9 --     Symbolic synthesis of masking fault-tolerant distributed
10 --    programs. Distributed Computing, 25:83âĂŞ108, 2012.
11 --    10.1007/s00446-011-0139-3.
12
13 type Decision   = {Undec, Zero, One}
14 type FaultCounter = {0..3}
15
16 d_g : Decision -- The commanding general's decision
17 d_j : Decision -- The lieutenant general j's decision
18 d_k : Decision -- The lieutenant general k's decision
19 d_l : Decision -- The lieutenant general l's decision
20
21 b_g : Bool     -- Whether or not commander is byzantine
22 b_j : Bool     -- Whether or not lieutenant j is byzantine
23 b_k : Bool     -- Whether or not lieutenant k is byzantine
24 b_l : Bool     -- Whether or not lieutenant l is byzantine
25
26 initialization
27 from all_init
28 begin
29     if  True -> d_g := Zero -- The general decides either Zero or One
30      [] True -> d_g := One  --    non-deterministically
```

```
31      fi
32  -- FAULTS
33      ;if True -> b_g := True -- At most one process may be byzantine
34       [] True -> b_j := True
35       [] True -> b_k := True
36       [] True -> b_l := True
37       [] True -> skip
38      fi
39  end
40
41  agent G "general" (d_g, b_g)
42  agent J "non_general" (d_g, d_j, d_k, d_l, b_j)
43  agent K "non_general" (d_g, d_k, d_l, d_j, b_k)
44  agent L "non_general" (d_g, d_l, d_j, d_k, b_l)
45
46  -- FAULTS
47  transitions
48  begin
49      -- F0: If no process is byzantine,
50      --   at most one agent may become byzantine
51      if (neg b_g) /\ (neg b_j) /\ (neg b_k) /\ (neg b_l) ->
52          if  True -> b_g := True
53           [] True -> b_j := True
54           [] True -> b_k := True
55           [] True -> b_l := True
56           [] True -> skip
57          fi
58      fi
59  end
60
61  -- Invariant Predicate --
62  spec_obs_ltl = G ((
63       neg b_g /\ (neg b_j \/ neg b_k) /\ (neg b_k \/ neg b_l)
64          /\ (neg b_l \/ neg b_j) /\
65      (neg b_j => (d_j == Undec \/ d_j == d_g)) /\
66      (neg b_k => (d_k == Undec \/ d_k == d_g)) /\
67      (neg b_l => (d_l == Undec \/ d_l == d_g)) /\
68      ((neg b_j /\ J.f) => (d_j /= Undec)) /\
69      ((neg b_k /\ K.f) => (d_k /= Undec)) /\
70      ((neg b_l /\ L.f) => (d_l /= Undec)))
71      \/
72      (b_g /\ neg b_j /\ neg b_k /\ neg b_l /\
73      d_j == d_k /\ d_k == d_l /\ d_j /= Undec))
74
75  -- P1) A lieutenant process knows that if it is not byzantine and
76  --     it has made its final decision and the general is not byzantine,
77  --     then its final decision agrees with the commander's decision
78  spec_obs_ltl = G
```

```
79     ((Knows J (neg b_j /\ J.f /\ neg b_g) => (d_j == d_g)) /\
80      (Knows K (neg b_k /\ K.f /\ neg b_g) => (d_k == d_g)) /\
81      (Knows L (neg b_l /\ L.f /\ neg b_g) => (d_l == d_g)))
82
83 -- P2) A lieutenant general process knows that its final decision
84 --     agrees with a second process if both processes are not byzantine
85 spec_obs_ltl = G
86     ((neg b_j /\ J.f) => Knows J ((neg b_k /\ K.f)
87      => (d_j == d_k) /\ (neg b_l /\ L.f) => (d_j == d_l)) /\
88      (neg b_k /\ K.f) => Knows K ((neg b_j /\ J.f)
89      => (d_k == d_j) /\ (neg b_l /\ L.f) => (d_k == d_l)) /\
90      (neg b_l /\ L.f) => Knows L ((neg b_j /\ J.f)
91      => (d_l == d_j) /\ (neg b_k /\ K.f) => (d_l == d_k)))
92
93 -- P3) The commander knows that the final decision of the
94 --     non-byzantine lieutenants will agree with its decision
95 spec_obs_ltl = G neg b_g => Knows G
96     (((neg b_j /\ J.f) => (d_g == d_j)) /\
97      ((neg b_k /\ K.f) => (d_g == d_k))  /\
98      ((neg b_l /\ L.f) => (d_g == d_l)))
99
100 protocol "general" (d : Decision, b : Bool)
101 c : FaultCounter
102     where c == 3
103 begin
104     -- F1: A byzantine process may change its decision arbitrarily
105     do  b /\ c > 0 -> << d.write(Zero) | c := c - 1 >>
106      [] b /\ c > 0 -> << d.write(One)  | c := c - 1 >>
107      [] neg b -> skip
108     od
109 end
110
111 protocol "non_general" (d_g : Decision, d_j : Decision, d_k : Decision,
112                                 d_l : Decision, b_j : Bool)
113 c : FaultCounter
114 f : Bool
115     where c == 3 /\ f == False
116 begin
117     do
118      -- fault intolerant version
119 --         (d_j == Undec) /\ (neg f) /\ (d_g == Zero)
120 --         -> << d_j.write(Zero) >> -- BA1a
121 --      [] (d_j == Undec) /\ (neg f) /\ (d_g == One)
122 --         -> << d_j.write(One)  >> -- BA1b
123 --      [] (d_j /= Undec) /\ (neg f)
124 --         -> f := True             -- BA2
125      -- fault tolerant version
126         (d_j == Undec) /\ (neg f) /\ (d_g == Zero)
```

```
127         -> << d_j.write(Zero) >> -- BA1a (unchanged)
128       [] (d_j == Undec) /\ (neg f) /\ (d_g == One)
129         -> << d_j.write(One)  >> -- BA1b (unchanged)
130       [] (d_j /= Undec) /\ (neg f) /\ (d_k == Undec \/ d_k == d_j) /\
131         (d_l == Undec \/ d_l == d_j)/\(d_k /= Undec \/ d_l /= Undec)
132         -> f := True
133       [] (d_j == One) /\ (d_k == Zero) /\ (d_l == Zero) /\ (neg f)
134         -> << d_j.write(Zero) >>
135       [] (d_j == One) /\ (d_k == Zero) /\ (d_l == Zero) /\ (neg f)
136         -> << d_j.write(Zero) | f := True >>
137       [] (d_j == Zero) /\ (d_k == One) /\ (d_l == One)  /\ (neg f)
138         -> << d_j.write(One)  >>
139       [] (d_j == Zero) /\ (d_k == One) /\ (d_l == One)  /\ (neg f)
140         -> << d_j.write(One)  | f := True >>
141       [] (d_j /= Undec) /\ (neg f) /\ ((d_j == d_k /\ d_j /= d_l) \/
142         (d_j == d_l /\ d_j /= d_k))
143         -> f := True
144
145       -- F1: A byzantine process may change its decision arbitrarily
146       [] b_j /\ c > 0
147         -> << d_j.write(Zero) | c := c - 1 >>
148       [] b_j /\ c > 0
149         -> << d_j.write(One)  | c := c - 1 >>
150     od
151 end
```

Listing 1: *The Byzantine Generals Problem modelled in MCK*

8 Results

The MCK model of the Byzantine Generals Problem in Listing 1 contains the actions performed by both the fault intolerant version of the Byzantine Generals Problem $\mathcal{BA}$ and fault tolerant version $\mathcal{BA}'$ on lines 119-124 and 126-143 respectively. Faults are introduced on lines 33-38 in the initialization block and lines 47-59 in the transition block. Lines 33-38 in the initialization block enable up to one process to start in a byzantine state. Lines 47-59 allow a process to become byzantine if no process is byzantine already at the end of each round. Once a process becomes byzantine, it will remain byzantine; however, a fault counter is used on lines 105 and 106 to ensure that if the commanding general becomes byzantine then only a finite number of faults can occur and similarly the fault counter on lines 147 and 149 ensures that if a lieutenant general becomes byzantine then only a finite number of faults can occur.

In this section, we use MCK to test whether or not the program invariant from [2] holds, as well as the three properties verifying the correctness of the solution to

the Byzantine Generals Problem from section 7.2 defined using temporal epistemic modal logic.

8.1 Fault-Intolerant version of $\mathcal{BA}$ Without Faults

The following shows the output from MCK when the model checker with added line breaks when it is run against the fault-intolerant version of $\mathcal{BA}$ without any faults occurring. To verify the model with using the fault-intolerant version of $\mathcal{BA}$ in the absence of faults, lines 33-38, 47-59, 126-143 and 146-149 are commented out and lines 119-124 are uncommented.

```
apokluda@scslt077:~/Documents/CS745/Project/mck$ mck byzantine.mck
MCK, version 1.0.0, January, 2012.
(C)opyright [2002..2012] University of New South Wales.

spec_obs_ltl =
G ((((((((((neg b_g) /\ ((neg b_j) \/ (neg b_k))) /\ ((neg b_k) \/
(neg b_l))) /\ ((neg b_l) \/ (neg b_j))) /\ ((neg b_j) =>
((d_j == Undec) \/ (d_j == d_g)))) /\ ((neg b_k) => ((d_k == Undec) \/
(d_k == d_g)))) /\ ((neg b_l) => ((d_l == Undec) \/ (d_l == d_g)))) /\
(((neg b_j) /\ J.f) => (neg (d_j == Undec)))) /\ (((neg b_k) /\ K.f) =>
(neg (d_k == Undec)))) /\ (((neg b_l) /\ L.f) => (neg (d_l == Undec))))
\/ ((((((b_g /\ (neg b_j)) /\ (neg b_k)) /\ (neg b_l)) /\ (d_j == d_k))
/\ (d_k == d_l)) /\ (neg (d_j == Undec))))
>> Spec holds.


spec_obs_ltl =
G ((((Knows J (((neg b_j) /\ J.f) /\ (neg b_g))) => (d_j == d_g)) /\
((Knows K (((neg b_k) /\ K.f) /\ (neg b_g))) => (d_k == d_g))) /\
((Knows L (((neg b_l) /\ L.f) /\ (neg b_g))) => (d_l == d_g)))
>> Spec holds.


spec_obs_ltl =
G (((((neg b_j) /\ J.f) => ((Knows J ((((neg b_k) /\ K.f) =>
((d_j == d_k) /\ ((neg b_l) /\ L.f))) => (d_j == d_l))) /\ ((neg b_k)
/\ K.f))) => ((Knows K ((((neg b_j) /\ J.f) => ((d_k == d_j) /\
((neg b_l) /\ L.f))) => (d_k == d_l))) /\ ((neg b_l) /\ L.f))) =>
(Knows L ((((neg b_j) /\ J.f) => ((d_l == d_j) /\ ((neg b_k) /\ K.f)))
```

```
=> (d_l == d_k))))
>> Spec holds.


spec_obs_ltl =
(G (neg b_g)) => (Knows G (((((neg b_j) /\ J.f) => (d_g == d_j)) /\
(((neg b_k) /\ K.f) => (d_g == d_k))) /\ (((neg b_l) /\ L.f) =>
(d_g == d_l))))
>> Spec holds.
```

As can be seen from the output, the program invariant and the temporal epistemic modal logic properties P1 - P3 hold for the fault-intolerant version of $\mathcal{BA}$ in the absense of faults.

8.2 Fault-Intolerant version of $\mathcal{BA}$ With Faults

The following shows the output from MCK when the model checker is run against the fault-intolerant version of $\mathcal{BA}$ in the presence of faults. To enable faults, lines 33-38 and 47-58 are uncommented.

```
apokluda@scslt077:~/Documents/CS745/Project/mck$ mck byzantine.mck
MCK, version 1.0.0, January, 2012.
(C)opyright [2002..2012] University of New South Wales.

spec_obs_ltl =
G ((((((((((neg b_g) /\ ((neg b_j) \/ (neg b_k))) /\ ((neg b_k) \/
(neg b_l))) /\ ((neg b_l) \/ (neg b_j))) /\ ((neg b_j) =>
((d_j == Undec) \/ (d_j == d_g)))) /\ ((neg b_k) => ((d_k == Undec) \/
(d_k == d_g)))) /\ ((neg b_l) => ((d_l == Undec) \/ (d_l == d_g)))) /\
(((neg b_j) /\ J.f) => (neg (d_j == Undec)))) /\ (((neg b_k) /\ K.f) =>
(neg (d_k == Undec)))) /\ (((neg b_l) /\ L.f) => (neg (d_l == Undec))))
\/ ((((((b_g /\ (neg b_j)) /\ (neg b_k)) /\ (neg b_l)) /\
(d_j == d_k)) /\ (d_k == d_l)) /\ (neg (d_j == Undec))))
>> Spec fails.


spec_obs_ltl =
G ((((Knows J (((neg b_j) /\ J.f) /\ (neg b_g))) => (d_j == d_g)) /\
((Knows K (((neg b_k) /\ K.f) /\ (neg b_g))) => (d_k == d_g))) /\
((Knows L (((neg b_l) /\ L.f) /\ (neg b_g))) => (d_l == d_g)))
>> Spec holds.
```

```
spec_obs_ltl =
G (((((neg b_j) /\ J.f) => ((Knows J ((((neg b_k) /\ K.f) =>
((d_j == d_k) /\ ((neg b_l) /\ L.f))) => (d_j == d_l))) /\ ((neg b_k)
/\ K.f))) => ((Knows K ((((neg b_j) /\ J.f) => ((d_k == d_j) /\
((neg b_l) /\ L.f))) => (d_k == d_l))) /\ ((neg b_l) /\ L.f))) =>
(Knows L ((((neg b_j) /\ J.f) => ((d_l == d_j) /\ ((neg b_k) /\ K.f)))
=> (d_l == d_k))))
>> Spec fails.


spec_obs_ltl =
(G (neg b_g)) => (Knows G (((((neg b_j) /\ J.f) => (d_g == d_j)) /\
(((neg b_k) /\ K.f) => (d_g == d_k))) /\ (((neg b_l) /\ L.f) =>
(d_g == d_l))))
>> Spec fails.
```

As we would expect, the program invariant is not satisfied. Interestingly, property P1 still holds; however, on careful consideration, this makes sense. The property states that if a lieutenant and commander are not byzantine, then that lieutenant knows that its final decision agrees with the general. No recovery actions are needed to make this statement true, therefore it is still satisfied by the fault-intolerant version of $\mathcal{BA}$ in the presence of faults. Properties P2 and P3, which require agreement between lieutenant processes, are not satisfied.

8.3 Fault-Tolerant version of $\mathcal{BA}$ Without Faults

The following shows the output from MCK when the model checker is run against the fault-tolerant version of $\mathcal{BA}$ without any faults occurring. To verify the model using the fault-tolerant version of $\mathcal{BA}$ without faults occurring, lines 32-38, 47-59, 105-108, 119-124 and 146-149 are commented out while lines 126-143 are uncommented.

```
apokluda@scslt077:~/Documents/CS745/Project/mck$ mck byzantine.mck
MCK, version 1.0.0, January, 2012.
(C)opyright [2002..2012] University of New South Wales.

spec_obs_ltl =
G ((((((((((neg b_g) /\ ((neg b_j) \/ (neg b_k))) /\ ((neg b_k) \/
(neg b_l))) /\ ((neg b_l) \/ (neg b_j))) /\ ((neg b_j) =>
```

```
((d_j == Undec) \/ (d_j == d_g)))) /\ ((neg b_k) => ((d_k == Undec) \/
(d_k == d_g)))) /\ ((neg b_l) => ((d_l == Undec) \/ (d_l == d_g)))) /\
(((neg b_j) /\ J.f) => (neg (d_j == Undec)))) /\ (((neg b_k) /\ K.f) =>
(neg (d_k == Undec)))) /\ (((neg b_l) /\ L.f) => (neg (d_l == Undec))))
\/ ((((((b_g /\ (neg b_j)) /\ (neg b_k)) /\ (neg b_l)) /\ (d_j == d_k))
/\ (d_k == d_l)) /\ (neg (d_j == Undec))))
>> Spec holds.


spec_obs_ltl =
G ((((Knows J (((neg b_j) /\ J.f) /\ (neg b_g))) => (d_j == d_g)) /\
((Knows K (((neg b_k) /\ K.f) /\ (neg b_g))) => (d_k == d_g))) /\
((Knows L (((neg b_l) /\ L.f) /\ (neg b_g))) => (d_l == d_g)))
>> Spec holds.


spec_obs_ltl =
G (((((neg b_j) /\ J.f) => ((Knows J ((((neg b_k) /\ K.f) =>
((d_j == d_k) /\ ((neg b_l) /\ L.f))) => (d_j == d_l))) /\ ((neg b_k)
/\ K.f))) => ((Knows K ((((neg b_j) /\ J.f) => ((d_k == d_j) /\
((neg b_l) /\ L.f))) => (d_k == d_l))) /\ ((neg b_l) /\ L.f))) =>
(Knows L ((((neg b_j) /\ J.f) => ((d_l == d_j) /\ ((neg b_k) /\ K.f)))
=> (d_l == d_k))))
>> Spec holds.


spec_obs_ltl =
(G (neg b_g)) => (Knows G (((((neg b_j) /\ J.f) => (d_g == d_j)) /\
(((neg b_k) /\ K.f) => (d_g == d_k))) /\ (((neg b_l) /\ L.f) =>
(d_g == d_l))))
>> Spec holds.
```

As we would expect, the program invariant and properties P1 - P3 are satisfied. This demonstrates that the addition of the recovery actions in the fault-tolerant version of $\mathcal{BA}$ does not alter the behaviour of the program in the absence of faults.

8.4 Fault-Tolerant version of $\mathcal{BA}$ With Faults

The following shows the output from MCK model checker with added line breaks when it is run against the fault-tolerant version of $\mathcal{BA}$ with any one process

becoming byzantine and a finite amount of faults occurring. To verify the model using the fault-tolerant version of $\mathcal{BA}$ in the presence of faults, the model is used as it is presented in listing 1 except that the definition of the program invariant and properties P1 - P3 are changes changed slightly: previously these specifications were defined using the global temporal operator **G**, specifying that the program invariant and properties P1 - P3 must always hold but we do not expect the program invariant and these properties to hold between the time that a fault occurs and the fault-tolerant version of the program has recovered from the faults, so we replace the global temporal operator with the finally temporal operator **F**, specifying that the program invariant must *eventually* hold and that properties P1 - P3 must eventually be satisfied. That is, the program invariant and properties P1 - P3 must be satisfied after the program has recovered from any faults.

```
apokluda@scslt077:~/Documents/CS745/Project/mck$ mck byzantine.mck
MCK, version 1.0.0, January, 2012.
(C)opyright [2002..2012] University of New South Wales.

spec_obs_ltl =
F ((((((((((neg b_g) /\ ((neg b_j) \/ (neg b_k))) /\ ((neg b_k) \/
(neg b_l))) /\ ((neg b_l) \/ (neg b_j))) /\ ((neg b_j) =>
((d_j == Undec) \/ (d_j == d_g)))) /\ ((neg b_k) => ((d_k == Undec) \/
(d_k == d_g)))) /\ ((neg b_l) => ((d_l == Undec) \/ (d_l == d_g)))) /\
(((neg b_j) /\ J.f) => (neg (d_j == Undec)))) /\ (((neg b_k) /\ K.f) =>
(neg (d_k == Undec)))) /\ (((neg b_l) /\ L.f) => (neg (d_l == Undec))))
\/ ((((((b_g /\ (neg b_j)) /\ (neg b_k)) /\ (neg b_l)) /\ (d_j == d_k))
/\ (d_k == d_l)) /\ (neg (d_j == Undec))))
>> Spec holds.


spec_obs_ltl =
F ((((Knows J (((neg b_j) /\ J.f) /\ (neg b_g))) => (d_j == d_g)) /\
((Knows K (((neg b_k) /\ K.f) /\ (neg b_g))) => (d_k == d_g))) /\
((Knows L (((neg b_l) /\ L.f) /\ (neg b_g))) => (d_l == d_g)))
>> Spec holds.


spec_obs_ltl =
F (((((neg b_j) /\ J.f) => ((Knows J ((((neg b_k) /\ K.f) =>
((d_j == d_k) /\ ((neg b_l) /\ L.f))) => (d_j == d_l))) /\ ((neg b_k)
/\ K.f))) => ((Knows K ((((neg b_j) /\ J.f) => ((d_k == d_j) /\
```

```
((neg b_l) /\ L.f))) => (d_k == d_l))) /\ ((neg b_l) /\ L.f))) =>
(Knows L ((((neg b_j) /\ J.f) => ((d_l == d_j) /\ ((neg b_k) /\ K.f)))
=> (d_l == d_k))))
>> Spec holds.


spec_obs_ltl =
(neg b_g) => (F (Knows G (((((neg b_j) /\ J.f) => (d_g == d_j)) /\
(((neg b_k) /\ K.f) => (d_g == d_k))) /\ (((neg b_l) /\ L.f) =>
(d_g == d_l)))))
>> Spec holds.
```

As we expect, the fault-tolerant version of $\mathcal{BA}$ satisfies the properties P1 - P3, which verifies that the program can recover from a finite number of faults.

9 Related Work

The modelling and verification of multi-agent systems has received quite a bit of attention over the last few decades. For example, [3] is a full-length textbook dedicated to the topic of epistemic logic and knowledge-based programming. The book [4] is another full-length textbook that devotes a chapter to epistemic logic but also covers a broad range of other topics relevant to formal verification. Both of these books abstractly discuss using epistemic modal logic to model multi-agent systems.

This report used the fault-intolerant and synthesized fault-tolerant implementation of the Byzantine Generals Problem from [2], with the aim of eventually improving the authors' synthesis algorithm with epistemic logic. The authors of that paper have other published works on program synthesis, including the related paper [1].

The authors of the MCK model checker have published a number of works on modelling and verifying multi-agent systems using epistemic modal logic including [8, 12, 9, 13]. Other researchers have also focused on model checking knowledge and time, for instance [11].

Work has also been done on the verification of epistemic properties of multi-agent systems, as was done in this report [10, 15]. A number of publications were made in the 1970s and 1980s on the use of knowledge in program synthesis from specification, for example [7, 5]; however, little if any work has been done recently on using epistemic modal logic in the process of synthesizing a fault-tolerant program from a fault-intolerant one.

10 Conclusion and Future Work

This report first discussed the importance of program synthesis, and in particular the synthesis of a fault-tolerant program from a fault-intolerant one. Subsequently, the reader was introduced to epistemic modal logic and the possible worlds model for knowledge. These concepts were reinforced with the example of the Muddy Children Puzzle.

The refined goal of this project was to verify the fault-tolerance and recovery properties of a synthesized fault-tolerant program. This was achieved by modelling a version of the Byzantine General's Problem in the MCK model checker and verifying that the synthesized fault-tolerant program met its specification in the presence of faults, where the specification was formulated using properties expressed as temporal epistemic modal logic formulae.

As was mentioned in the introduction, the original goal of this research project was to incorporate epistemic modal logic into the program synthesis algorithm presented in [2], which is now left to future work. However, this report defined the problem and educated the reader on related topics which lays the groundwork for this important future work.

References

[1] Borzoo Bonakdarpour and Sandeep Kulkarni. Revising distributed unity programs is np-complete. In Theodore Baker, Alain Bui, and Sébastien Tixeuil, editors, *Principles of Distributed Systems*, volume 5401 of *Lecture Notes in Computer Science*, pages 408–427. Springer Berlin / Heidelberg, 2008. 10.1007/978-3-540-92221-6_26.

[2] Borzoo Bonakdarpour, Sandeep Kulkarni, and Fuad Abujarad. Symbolic synthesis of masking fault-tolerant distributed programs. *Distributed Computing*, 25:83–108, 2012. 10.1007/s00446-011-0139-3.

[3] Ronald Fagin, Joseph Y Halpern, Yoram Moses, and Moshe Y Vardi. *Reasoning About Knowledge*. MIT Press, 2003.

[4] M Huth and M Ryan. *Logic in Computer Science: modelling and reasoning about systems*. Cambridge University Press, 2000.

[5] E. Kant and D.R. Barstow. The refinement paradigm: The interaction of coding and efficiency knowledge in program synthesis. *Software Engineering, IEEE Transactions on*, SE-7(5):458 – 471, sept. 1981.

[6] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, July 1982.

[7] Zohar Manna and Richard Waldinger. Knowledge and reasoning in program synthesis. *Artificial Intelligence*, 6(2):175 – 208, 1975.

[8] Ron Van Der Meyden. Knowledge based programs: On the complexity of perfect recall in finite environments (extended abstract). In *Proceedings of the Sixth Conference on Theoretical Aspects of Rationality and Knowledge*, pages 31–50. Morgan Kaufmann Publishers, 1996.

[9] Ron van der Meyden and Kaile Su. Symbolic model checking the knowledge of the dining cryptographers. In *Proceedings of the 17th IEEE workshop on Computer Security Foundations*, CSFW '04, pages 280–, Washington, DC, USA, 2004. IEEE Computer Society.

[10] W. Penczek and A. Lomuscio. Verifying epistemic properties of multi-agent systems via bounded model checking. In *Proceedings of the second international joint conference on Autonomous agents and multiagent systems*, AAMAS '03, pages 209–216, New York, NY, USA, 2003. ACM.

[11] Wiebe van der Hoek and Michael Wooldridge. Model checking knowledge and time. In Dragan BoÅąnacki and Stefan Leue, editors, *Model Checking Software*, volume 2318 of *Lecture Notes in Computer Science*, pages 25–26. Springer Berlin / Heidelberg, 2002. 10.1007/3-540-46017-9_9.

[12] Ron van der Meyden. Common knowledge and update in finite environments. i: extended abstract. In *Proceedings of the 5th conference on Theoretical aspects of reasoning about knowledge*, TARK '94, pages 225–242, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[13] Ron van der Meyden. Finite state implementations of knowledge-based programs. In V. Chandru and V. Vinay, editors, *Foundations of Software Technology and Theoretical Computer Science*, volume 1180 of *Lecture Notes in Computer Science*, pages 262–273. Springer Berlin / Heidelberg, 1996. 10.1007/3-540-62034-6_55.

[14] Ron van der Meyden, Peter Gammie, Kai Baukus, Jeremy Lee, Cheng Luo, and Xiaowei Huang. *MCK 1.0.0 User Manual*. University of New South Wales, February 2012.

[15] Michael Wooldridge. Verifying that agents implement a communication language. In *Proceedings of the sixteenth national conference on Artificial intelligence and the eleventh Innovative applications of artificial intelligence conference*

innovative applications of artificial intelligence, AAAI '99/IAAI '99, pages 52–57, Menlo Park, CA, USA, 1999. American Association for Artificial Intelligence.