Games–Based Model Checking of Protocols: counting doesn't count

Tim Kempster, Colin Stirling and Peter Thanisch
Institute for Computing Systems Architecture
Division of Informatics
University of Edinburgh,
Edinburgh EH9 3JZ, Scotland,
email: {tdk,cps,pt}@dcs.ed.ac.uk

Abstract

We introduce a technique that can be used to model the behaviour of protocols. In our model each process within a protocol belongs to a particular class. A set of rules governs the behaviour of a process within a process class. The rules give rise to a transition system that models the behaviour of the protocol. By exploiting the homogeneous behavior of a process within a particular class it is possible to model the behaviour of an unbounded number of processes in a way that results in a finite (and in most cases small) transition system. We demonstrate this technique by modeling the popular two-phase commit protocol. CTL properties can be model checked in the resulting transition system using a games based model checking algorithm. This technique has the advantage that the entire transition system need not always be generated. Furthermore, it provides evidence to verify or refute the property being checked. Using the two-phase commit example, we check some elementary properties.

1. Introduction

Computers and computing systems are becoming connected in ways that will shape the world we live in forever. Advances in computer networks will provide huge bandwidth communication channels between every corner of the globe, connecting billions of autonomous computing devices (processes) into large distributed systems. In the future, for example, embedded processors in electrical appliances might communicate their usage patterns, via the Internet, to electricity providers allowing them to better plan production.

A protocol allows a group of processes to exchange information, and solve a common task. Existing protocols solve problems such as electing a leader from a group or reliably exchanging information when failures can occur.

As we adopt these solutions into our lives we become more and more reliant on them. For this reason it is important to fully understand them and to be confident that their behaviour is as intended.

We provide a general method in which to model protocols. Processes that exhibit similar behaviour in a protocol are grouped into classes, and rules for each class are given. We then show how a transition system can be automatically generated from this model. By exploiting the regular behaviour of processes within a particular class we are able to model an unbounded number of processes using a finite transition system. Temporal properties of protocols can be expressed in CTL[2]. We provide a games–based model checking technique to verify or refute properties in a model. Two major advantages with this model checking technique are discussed. Throughout the paper, we use the two-phase commit protocol[1] as as a case study for our techniques.

2. Modeling Protocols

2.1. Processes, States and Views

We model a protocol as a system of processes. Processes communicate by means of asynchronous message passing. Each process belongs to a particular class of processes, corresponding to the role it plays in the protocol. Processes have a set of local state variables. Each class has a set of rules that determine the behaviour of all processes within that class.

Let p be a process from class P. We say p has a local state variable $p.s$. Each variable may take a finite number of values. For instance if p's variable s is in state $\mathbf{x}$ we say $p.s = \mathbf{x}$ holds.

Together with its local state each process has a *view* of the internal state of remote processes. This view is constructed from information it receives from remote processes in the form of messages. We say,

$$q.s = \lhd \mathbf{x}$$

holds at process p if the most up to date message p received from q informed p that q *was* in state $\mathbf{x}$. We restrict our attention to asynchronous message passing where messages are never lost during transmission but there is no bound on their transmission time. We also assume that messages arrive in the order they were sent. It is important to note that if at some point, q is in state $\mathbf{x}$ then this does *not* imply that at a remote process, p, $q.s = \lhd \mathbf{x}$ will hold. This is because the message reporting q's state change may not have arrived at p yet. Similarly, if at p, $q.s = \lhd \mathbf{x}$ holds, this does not imply that q is still in state $\mathbf{x}$.

2.2. Rules

The behaviour of each process within a particular class is defined by a set of rules. Each rule consists of a pre-condition and a postaction. Let R be a rule for a class of processes P, and p a process from that class. The pre-condition of R makes assertions over p's local state together with p's *view* of remote processes' states. If the pre-condition of rule R holds for p then R's postaction may happen changing the local state at p. When p's local state is updated in the postaction of a rule, a message is sent to any process that maintains a view of p's state, enabling them to update their view of p's new local state.

To exemplify this idea let P and Q be classes of processes both with a single state variable s, which may take one of two values $\mathbf{x}$ or $\mathbf{y}$. Processes, p and q, from classes P and Q respectively both have their s variable initialised to $\mathbf{x}$. Let class Q have the single rule

$$\mathbf{Q1}\ \frac{q.s = \mathbf{x}}{q.s := \mathbf{y}}$$

and class P have the single rule

$$\mathbf{P1}\ \frac{\exists q \in Q,\ q.s = \lhd \mathbf{y}}{p.s := \mathbf{y}}$$

If we consider a system with two processes, p_1 and p_2, from class P and one process, q from class Q we will see all the possible behaviours in Figure 1.

3. Modeling Two-Phase Commit

We now introduce a model of the popular centralised two-phase commit protocol widely used in distributed database technology[3]. This protocol consists of a transaction coordinator and some number of data managers or participants. When the operations of a transaction have completed[1] the commit protocol is invoked. The coordinator asks each participant to vote on the feasibility of the transaction. If *all* participants vote 'yes' then the coordinator can decide to commit the transaction and thus sends a commit message to each participant. On receipt of this commit message a participant enters its commit state. If *any* participant votes 'no' it enters its abort state and sends an abort message to its coordinator. When the coordinator receives this message it sends an abort message to each remaining participants. On receiving an abort message each participant enters its abort state. Perhaps the most important property of commit protocols is that of *atomicity*, either all participants commit or else they all abort.

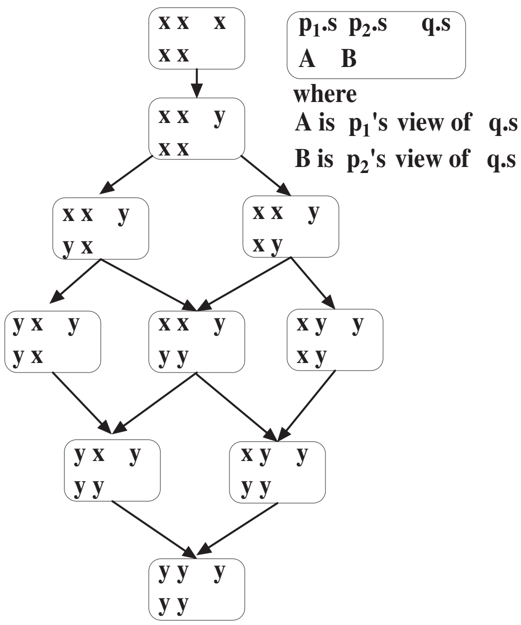

Figure 1. We can see the possible different behaviour of the simple protocol above. A single Q class process moves from state $\mathbf{x}$ to state $\mathbf{y}$. Once the two processes p_1 and p_2 view this state change they also move from state $\mathbf{x}$ to state $\mathbf{y}$.

In our notation we have two classes of processes. The coordinator class C and the participant class P. Processes from each class have a single state variable s. In the case of the coordinator this variable may take values **i** (initial), **c** (commit) or abort **a** (abort). In addition to these states participants have a further state **w** (wait). In the centralised two-phase commit we model, all communication is between a participant and a single coordinator. For this reason each participant maintains a view of it's coordinators state and the coordinator maintains a view of the state of each of the participants it is coordinating. Table 1 summarises this.

[1] Many different varieties of two-phase commit exist. In some the commit protocol is invoked before all the operations have completed.

	Participant P	Coordinator C
State Values	**i, w, a, c**	**i, a, c**
Number	Many processes	Single Process
Views	View of Coordinator	View of participants

Table 1. Table of state in centralised two-phase commit

We now give the rules for each of the classes. The participant processes have four rules, the first two rules, **P1** and **P2**, allows a participant to vote 'yes' and enter its **w** state or vote 'no' and enter its **a** state. It should be noted that participants have a non-deterministic choice of whether to commit or abort.

$$\mathbf{P1} : \frac{p.s = \mathbf{i}}{p.s := \mathbf{w}} \ \mathbf{P2} : \frac{p.s = \mathbf{i}}{p.s := \mathbf{a}}$$

The next two rules, **P3** and **P4**, allow a participant to enter either a **c** state if it views its coordinator in the **c** state or to abort entering **a** if the coordinator aborts.

$$\mathbf{P3} : \frac{p.s = \mathbf{w} \ \wedge \ \exists c \in C,\ c.s = \lhd\mathbf{c}}{p.s := \mathbf{c}}$$

$$\mathbf{P4} : \frac{p.s \in \{\mathbf{w}, \mathbf{i}\} \ \wedge \exists c \in C,\ c.s = \lhd\mathbf{a}}{p.s := \mathbf{a}}$$

Only one process exists in the coordinator class C and this has only two rules. The first allows a coordinator to enter the **c** state if its view of *all* of its participants shows that they have all entered their **w** state (i.e. they have all voted 'yes'). The second rule allows a coordinator to enter the **a** state if its view of *any* of its participants shows that one has entered the **a** state (i.e. one of the participants has voted 'no').

$$\mathbf{C1} : \frac{c.s = \mathbf{i} \wedge \forall p \in P,\ p.s = \lhd\mathbf{w}}{c.s := \mathbf{c}}$$

$$\mathbf{C2} : \frac{c.s = \mathbf{i} \wedge \ \exists p \in P,\ p.s = \lhd\mathbf{a}}{p.s := \mathbf{a}}$$

4. Representing the System

4.1. Naive Approach

If we restrict our model to a fixed number of participants $p_1, \ldots, p_n$ and a single coordinator c, a system configuration $\mathcal{C}$ can be modeled as the composition of these processes, $(p_1, \ldots, p_n, c)$. Our system evolves by taking steps. The system may take a step if any process within this composition takes a step by executing one of the rules. Thus if rule $\mathbf{P2}$ happens at process p_i which we can write as $p_i \xrightarrow{\mathbf{P2}} p_i'$, then $\mathcal{C} \xrightarrow{\mathbf{P2}} \mathcal{C}'$ where $\mathcal{C}' = (p_1, \ldots, p_i', \ldots, p_n, c)$.

The system can also take a step by updating the view of a process. For example, if process p_i's view of the coordinator is **i** but the coordinators state is actually **a** then p_i's view of $c.s$ can be updated to the value **a**. This models a message arriving.

A run of the protocol is therefore an evolution of the system

$$\mathcal{C}_1 \rightarrow \mathcal{C}_2 \rightarrow \ldots$$

.

4.2. Counting Doesn't Count

In many protocols of interest processes within each class are anonymous. The above representation therefore is wasteful because if two processes exist in the same state within a class they are represented twice. Furthermore, the pre-conditions of rules often only assert that either *none*, *some*, or *all* processes from a class are in a particular state. For example in the two-phase commit protocol a coordinator moves to **c** if its view of the participant set is that they are *all* in the **w** state.

We can therefore represent an unbounded number of processes of a particular class using a set of states. This set represents the *possible* states a process might be in. For example, initially participants have an internal state $p.s = \mathbf{i}$ together with a view of the coordinators state, also equal to **i**, which we can write this as $\mathbf{i}^{\mathbf{i}}$. Where the super-script represents the participants view of the coordinators state. We can now represent a set of participants all in this state as $\{\mathbf{i}^{\mathbf{i}}\}$. Suppose one of these participants now votes 'yes' and moves to state $\mathbf{w}$. We represent this as the set $\{\mathbf{i}^{\mathbf{i}}, \mathbf{w}^{\mathbf{i}}\}$. This means some participants are in state $\mathbf{i}^{\mathbf{i}}$ and some are in $\mathbf{w}^{\mathbf{i}}$. Suppose now another participant votes 'yes'. We can derive two different transitions depending on whether or not this is the last participant in the set to vote. If we do we arrive at the set $\{\mathbf{w}^{\mathbf{i}}\}$ or if not we cycle to the set $\{\mathbf{w}^{\mathbf{i}}, \mathbf{i}^{\mathbf{i}}\}$. Figure 2 demonstrates the idea of collapsing a representation by exploiting processes anonymity within a class, by showing the first few transitions for participants.

A process of another class can now keep a collapsed view of a class using the same method. In our example the single coordinator process now maintains a view *set*. This represents the coordinators view of the possible different states of the participant processes. Thus if the participants are represented by the set $\{\mathbf{i}^{\mathbf{i}}, \mathbf{w}^{\mathbf{i}}, \mathbf{a}^{\mathbf{i}}\}$, then the coordinator's state can be represented as $\mathbf{i}^{\{\mathbf{w},\mathbf{i}\}}$. The coordinators view is updated when a message arrives informing it that one of the participants has moved to the **a**. This results in the state $\mathbf{i}^{\{\mathbf{w},\mathbf{i},\mathbf{a}\}}$.

This technique provides us with a method that enables us to model an unbounded number of participants with only very few states. It does however introduce two problems.

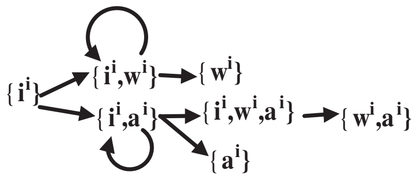

Figure 2. Collapsing multiple processes within the participant class. In order to keep our example small we omit the possible transitions which occur due to communication between the coordinator and participant classes.

Firstly, it is no longer possible to make certain assertions about our group of processes. An example would be, “half the group are in state **w**”. If our protocol requires this type of assertion then we must make this differentiation in our representation. For example we would need to record when none, some, half, or all processes entered state **w**. Fortunately, in our simple example we only use universal and existential quantification so our current collapsing scheme is sufficient.

Secondly, the collapsing scheme inadvertently introduces new behaviour. In Figure 2 we can see that cycles are present around certain states. Our transition system is therefore inaccurate in the sense that no run of the protocol with a finite number of participants could cycle in these states forever. When checking safety properties we need not worry about these cycles but for liveness properties they are a problem. Fortunately, it is possible to detect these false cycles using a counting scheme and remove them. We do not provide the details of this detection algorithm here.

Using the “counting doesn’t count” technique above software was written to automatically generate a transition system from a pre-condition postaction specification. Figure 3 shows the transition system generated from the specification of the two-phase commit protocol of Section 3. In the diagram only, in order to keep the state space small, we omit rule **P4**, i.e. all participants must vote ’yes’. We can see this only results in 18 states.

5. Expressing Properties of Protocols

In the previous sections we have shown how our pre-condition postaction model can be translated into a labeled transition system. Rules of our protocol move one system configuration to another producing labelled transitions. Our protocol rules take one system configuration to the next. In this section we show how to verify or refute temporal properties of a systems. We say

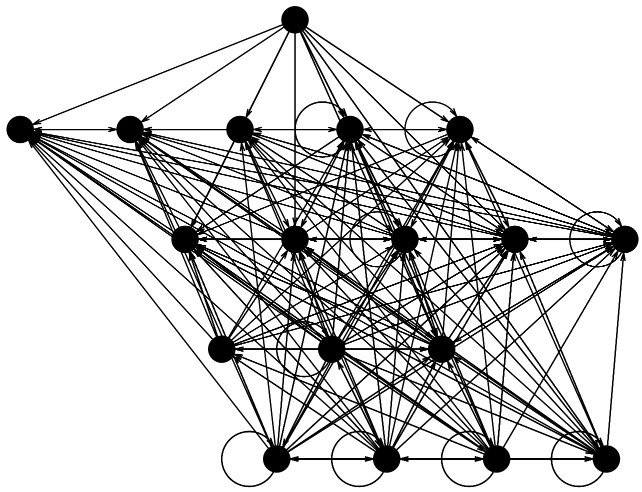

Figure 3. Transition system generated from the pre-condition postaction specification of two-phase commit in Section 3.

$$\mathcal{C} \models \Phi$$

If property Φ is valid for our system configuration $\mathcal{C}$.

The temporal logic we use to express properties is a slight variant of computation tree temporal logic, CTL due to Clarke, Emerson and Sistla. In the following K is a set of rule names (transition labels or actions), we use a to range over this set. We use the symbol ’$-$’ to represent any rule from the set of all rules. We introduce atomic formula, X, which are propositions over a system state. An example from the two-phase commit protocol might be, “the coordinator is in state **c**”.

$$\begin{array}{rcl} \Phi & ::= & \mathtt{tt} \mid \neg\Phi \mid \Phi_1 \wedge \Phi_2 \mid [K]\Phi \mid \langle K \rangle \Phi \\ & & \mathrm{A}(\Phi_1 \,\mathrm{U}\, \Phi_2) \mid \mathrm{E}(\Phi_1 \,\mathrm{U}\, \Phi_2) \mid X \end{array}$$

The definition of satisfaction between a configuration $\mathcal{C}_0$ and a formula proceeds by induction on the formula.

$\mathcal{C} \models \mathtt{tt}$

$\mathcal{C} \models \Phi \wedge \Psi$ iff $\mathcal{C} \models \Phi$ and $\mathcal{C} \models \Psi$

$\mathcal{C} \models [K]\Phi$ iff $\forall \mathcal{C}'' \in \{\mathcal{C}' : \mathcal{C} \xrightarrow{a} \mathcal{C}',\, a \in K\}$, $\mathcal{C}'' \models \Phi$

$\mathcal{C} \models \langle K \rangle \Phi$ iff $\exists \mathcal{C}'' \in \{\mathcal{C}' : \mathcal{C} \xrightarrow{a} \mathcal{C}',\, a \in K\}$, $\mathcal{C}'' \models \Phi$

$\mathcal{C}_0 \models \mathrm{A}(\Phi \,\mathrm{U}\, \Psi)$ iff for all runs $\mathcal{C}_0 \xrightarrow{a_1} \mathcal{C}_1 \xrightarrow{a_2} \ldots$ there is $i \geq 0$ with $\mathcal{C}_i \models \Psi$ and for all $j : 0 \leq j < i$, $\mathcal{C}_j \models \Phi$

$\mathcal{C}_0 \models \mathrm{E}(\Phi \,\mathrm{U}\, \Psi)$ iff for some run $\mathcal{C}_0 \xrightarrow{a_1} \mathcal{C}_1 \xrightarrow{a_2} \ldots$ there is $i \geq 0$ with $\mathcal{C}_i \models \Psi$ and for all $j : 0 \leq j < i$, $\mathcal{C}_j \models \Phi$

$\mathcal{C}_0 \models \neg\mathrm{A}(\Phi \,\mathrm{U}\, \Psi)$ iff for some run $\mathcal{C}_0 \xrightarrow{a_1} \mathcal{C}_1 \xrightarrow{a_2} \ldots$ for all $i \geq 0$, ($\mathcal{C}_i \models \neg\Psi$ or there is $j : 0 \leq j < i$, $\mathcal{C}_j \models \neg\Phi$)

$\mathcal{C}_0 \models \neg\mathrm{E}(\Phi \,\mathrm{U}\, \Psi)$ iff for all runs $\mathcal{C}_0 \xrightarrow{a_1} \mathcal{C}_1 \xrightarrow{a_2} \ldots$ for all $i \geq 0$, ($\mathcal{C}_i \models \neg\Psi$ or there is $j : 0 \leq j < i$, $\mathcal{C}_j \models \neg\Phi$)

$\mathcal{C} \models X$ iff Atomic formula X holds

For the sake of brevity we have omitted some of the negated formulas for example $\mathcal{C} \models \neg\langle K \rangle \Phi$ iff $\mathcal{C} \models [K]\neg\Phi$.

Strong Liveness properties are expressed using $A(\Phi\, U\, \Psi)$ For example if X is the atomic proposition "The coordinator is in state **c**", then formula $A(\mathtt{tt}\, U\, X)$ means in all runs the coordinator eventually commits.

Strong Safety properties are expressed using $\neg E(\Phi\, U\, \Phi)$. For example if X is the atomic proposition "A participant is in **c** and another is in **a**", then the formula $\neg E(\langle - \rangle \mathtt{tt}\, U\, \neg X)$ expresses the inability to reach a state where *atomicity*[2] is violated.

6. Games-Based Model Checking

The "property checking game" $G(\mathcal{C}, \Phi)$, when C is a transition system and Φ is a normal formula, is played by two players, players R (the refuter) and V (the verifier). Player R attempts to show that $\mathcal{C}$ fails to have the property Φ whereas player V wishes to establish that the property holds of $\mathcal{C}$.

A play of the game $G(\mathcal{C}_0, \Phi_0)$ is a finite or infinite length sequence of the form

$$(\mathcal{C}_0, \Phi_0) \ldots (\mathcal{C}_n, \Phi_n) \ldots$$

where each formula Φ_i is a subformula of Φ_0, and each $\mathcal{C}_i$ is a system configuration. If part of a play is $(\mathcal{C}_0, \Phi_0) \ldots (\mathcal{C}_j, \Phi_j)$. then the next move and which player makes it depends on the main connective of the formula Φ_j. All the possibilities are presented below.

[2]Atomicity ensures all participants either abort or commit.

- if $\Phi_j = \Psi_1 \wedge \Psi_2$ then player R chooses one of the conjuncts Ψ_i, $i \in \{1, 2\}$: the process $\mathcal{C}_{j+1}$ is $\mathcal{C}_j$ and Φ_{j+1} is Ψ_i.
- if $\Phi_j = \neg(\Psi_1 \wedge \Psi_2)$ then player V chooses one of the conjuncts Ψ_i, $i \in \{1, 2\}$: the process $\mathcal{C}_{j+1}$ is $\mathcal{C}_j$ and Φ_{j+1} is $\neg\Psi_i$.
- if $\Phi_j = [K]\Psi$ then player R chooses a transition $\mathcal{C}_j \xrightarrow{a} \mathcal{C}_{j+1}$ with $a \in K$ and Φ_{j+1} is Ψ.
- if $\Phi_j = \langle K \rangle \Psi$ then player V chooses a transition $\mathcal{C}_j \xrightarrow{a} \mathcal{C}_{j+1}$ with $a \in K$ and Φ_{j+1} is Ψ.
- if $\Phi_j = \neg\neg\Psi$ then $\mathcal{C}_{j+1}$ is $\mathcal{C}_j$ and Φ_{j+1} is Ψ.
- if $\Phi_j = \mathrm{A}(\Psi_1 \mathrm{U} \Psi_2)$ then $\mathcal{C}_{j+1}$ is $\mathcal{C}_j$ and Φ_{j+1} is $\Psi_2 \vee (\langle - \rangle \mathtt{tt} \wedge [-](\Psi_1 \wedge \mathrm{A}(\Psi_1 \mathrm{U} \Psi_2)))$
- if $\Phi_j = \neg\mathrm{A}(\Psi_1 \mathrm{U} \Psi_2)$ then $\mathcal{C}_{j+1}$ is $\mathcal{C}_j$ and Φ_{j+1} is $\neg(\Psi_2 \vee (\langle - \rangle \mathtt{tt} \wedge [-](\Psi_1 \wedge \mathrm{A}(\Psi_1 \mathrm{U} \Psi_2))))$
- if $\Phi_j = \mathrm{E}(\Psi_1 \mathrm{U} \Psi_2)$ then $\mathcal{C}_{j+1}$ is $\mathcal{C}_j$ and Φ_{j+1} is $\Psi_2 \vee \langle - \rangle(\Psi_1 \wedge \mathrm{E}(\Psi_1 \mathrm{U} \Psi_2))$
- if $\Phi_j = \neg\mathrm{E}(\Psi_1 \mathrm{U} \Psi_2)$ then $\mathcal{C}_{j+1}$ is $\mathcal{C}_j$ and Φ_{j+1} is $\neg(\Psi_2 \vee \langle - \rangle(\Psi_1 \wedge \mathrm{E}(\Psi_1 \mathrm{U} \Psi_2)))$

The rules appear to be complicated because of the presence of negation in the logic. The refuter chooses the next position when the formula has the form $\Psi_1 \wedge \Psi_2$ or $[K]\Psi$ and the verifier chooses when it has the form $\neg(\Psi_1 \wedge \Psi_2)$ or $\neg[K]\Psi$. As there are no choices in the remaining rules neither player is responsible for them. The first of these reduces a double negation. The remaining rules are for until formulas and their negations.

Figure 4 captures when a player is said to win a play of a game. Player R wins if a blatantly false position is reached and player V wins if a blatantly true position is reached. Condition 2 identifies which player wins an infinite length play. For any infinite length play of a CTL game there is only one until formula or negation of an until formula which occurs infinitely often. It is this formula which decides who wins. If it is an until formula then it is the refuter that wins and if it is the negation of an until formula then it is the verifier that wins. To detect when a formula and configuration can appear infinitely often we need only detect repeats.

A strategy for a player is a family of rules which tell the player how to move. It suffices to consider history-free strategies, whose rules do not depend upon previous positions in the play. For player R rules have the following form.

- At position $(\mathcal{C}, \Phi_1 \wedge \Phi_2)$ choose $(\mathcal{C}, \Phi_i)$ where $i \in \{1, 2\}$.
- At position $(\mathcal{C}, [K]\Phi)$ choose $(\mathcal{C}', \Phi)$ where $\mathcal{C} \xrightarrow{a} \mathcal{C}'$ and $a \in K$.

Player R wins

1. The play is $(\mathcal{C}_0, \Phi_0) \ldots (\mathcal{C}_n, \Phi_n)$ and
   - $\Phi_n = \mathtt{ff}^3$ or
   - $\Phi_n = \neg([K]\Psi)$ and $\{\mathcal{C}' : \mathcal{C} \xrightarrow{a} \mathcal{C}' \text{ and } a \in K\} = \emptyset$.
2. The play $(\mathcal{C}_0, \Phi_0) \ldots (\mathcal{C}_n, \Phi_n) \ldots$ has infinite length and there is an until formula $\mathrm{A}(\Psi_1 \mathrm{U} \Psi_2)$ or $\mathrm{E}(\Psi_1 \mathrm{U} \Psi_2)$ which occurs infinitely often.

Player V wins

1. The play is $(\mathcal{C}_0, \Phi_0) \ldots (\mathcal{C}_n, \Phi_n)$ and
   - $\Phi_n = \mathtt{tt}$ or
   - $\Phi_n = [K]\Psi$ and $\{\mathcal{C}' : \mathcal{C} \xrightarrow{a} \mathcal{C}' \text{ and } a \in K\} = \emptyset$.
2. The play $(\mathcal{C}_0, \Phi_0) \ldots (\mathcal{C}_n, \Phi_n) \ldots$ has infinite length and there is a negated until formula $\neg(\mathrm{A}(\Psi_1 \mathrm{U} \Psi_2))$ or $\neg(\mathrm{E}(\Psi_1 \mathrm{U} \Psi_2))$ which occurs infinitely often.

Figure 4. Winning conditions

For the verifier rules have a similar form.

- At position $(\mathcal{C}, \neg(\Phi_1 \wedge \Phi_2))$ choose $(\mathcal{C}, \Phi_i)$ where $i \in \{1, 2\}$.
- At position $(\mathcal{C}, \langle K \rangle \Phi)$ choose $(\mathcal{C}', \Phi)$ where $\mathcal{C} \xrightarrow{a} \mathcal{C}'$ and $a \in K$.

A player uses the strategy π in a play if all her moves in the play obey the rules in π. The strategy π is winning if the player wins every play in which she uses π. The following result provides an alternative account of the satisfaction relation between system configurations and formulas.

Proposition 1

1. *If* $\Phi \in \mathrm{CTL}$ *then* $\mathcal{C} \models \Phi$ *iff player V has a history-free winning strategy for* $G(\mathcal{C}, \Phi)$.
2. *If* $\Phi \in \mathrm{CTL}$ *then* $\mathcal{C} \not\models \Phi$ *iff player R has a history-free winning strategy for* $G(\mathcal{C}, \Phi)$.

Proof: See [5].

7. Checking Atomicity in Two Phase Commit

As a simple example let X be the atomic formula "there exists a participant p with $p.s = \mathbf{c}$" and Y the atomic formula "there exists a participant p with $p.s = \mathbf{a}$. The safety property $\neg E(\langle - \rangle \mathtt{tt}\, U \, \neg(X \text{ and } Y))$ expresses the property that our system never reaches a state where one participant has committed and another has aborted.

Again we rely of the anonymity of processes in atomic formulae X, Y. If P is the set representing the possible states of participants in our system then for some process to be in state **c** means $\exists p \in P$, $p.s = \mathbf{c}$.

We model checked this property, using an implementation of the games based algorithm above, against our simple two-phase commit protocol. As expected the verifier 'V' produced a history free winning strategy. If we add the following rule to our system

$$\mathbf{P5} : \frac{p.s = \mathbf{i}}{p.s := \mathbf{a}}$$

and check the same property we find that the refuter 'R' has a winning strategy. This is to be expected because participants are now able to unilaterally commit as well as abort. Many other interesting properties can be checked in this way.

Games-based model checking has the advantage of producing a strategy that proves or refutes the property being checked. This provides a protocol designer with a design cycle. For example, if a safety property were to fail a designer can 'play' the game and see exactly why the property fails. This then leads to modifications of the original protocol and re-verification.

8. Conclusions and Future Work

We have shown how protocols can be modeled using a simple rule based notation. In this notation a process has some internal state together with a view of remote processes state. A rule changes this state based on its current state and its view of other processes state. When a process changes state a message is sent to other processes informing them of this fact.

Given such a model we have shown how a transition system can be generated. A naive strategy leads to very large state spaces but we can exploit the homogeneous behaviour of processes within a particular class to reduce this dramatically. This technique allows us to reason about protocols with an unbounded number or processes. It does however restrict the types of assertions we can make about our systems. From a practical perspective this restriction does not prevent us from checking many interesting properties.

A games-based model checking procedure was described where a verifier 'plays' a game against a refuter. The verifier tries to prove a temporal property, expressed in CTL, whereas the refuter tries to disprove it. This strategy has two advantages over some other model checking techniques. Firstly, when a property holds or fails evidence is given to support the fact. Secondly, our algorithm does not always require the whole transition system to be generated leading often making more efficient than algorithms that expand the entire state space.

In this paper we assume messages are never lost, only delayed, and that processes never crash. By relaxing these assumption many interesting properties of protocols emerge. Future work will incorporate these ideas as well as modeling more complex protocols. The model described in this paper has already been sucessful in describing more complex protocols [4]. It is expected that using the ideas we have outlined in this paper we will be successful in providing automated verification techniques for a wide variety of protocols and protocol environments.

References

[1] P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, Reading, MA, 1987.

[2] E. E. E. Clarke and A. Sistla. Automatic verification of finite state concurrent systems using temporal logic specifications: a practical approach. In *Proc. 10th ACM Symposium on Principlesof Programming Languages*, pages 117–126, 1983.

[3] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, San Mateo, CA, 1993.

[4] T. Kempster, C. Stirling, and P. Thanisch. More committed quorum-based three phase commit protocol. In *Lecture Notes in Computer Science: The Twelth International Symposium on Distributed Computing*, page 246, 1998. On-line, http://www.dcs.ed.ac.uk/home/tdk/.

[5] P. Stevens and C. Stirling. Practical model-checking using games. In *Proc. 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems, TACAS'98*, volume 1384 of *LNCS*, pages 85–101. Springer-Verlag, 1998.