其中,$\mathcal{N}$是提出协议编号的集合,也称为 Numbers;Q 是一轮投票中所有 *Acceptors* 的集合,称作议会系统;集合$\mathcal{Q}$则为议会系统集合,在下文有其详细的解释与性质说明.$\exists n\in\mathcal{N}$:*sent*(“2*b*”,*n*,*v*,*a*)表示在投票轮次为 *n* 的投票中,由 Acceptor *a* 发出的一个消息,其中消息类型为“2*b*”,编号为 *b* 以及特定值为 *v*.通过发送这种类型的消息,代表着一个 Acceptor 进行了投票.

Basic Paxos 算法的具体推导过程见文献[12],这里只给出最终协议的大概流程.我们将 Proposer 和 Acceptor 的行为放在一起,我们可以看到该算法在以下两个阶段中运行.

- 第 1 阶段:(a) Proposer 选择一个编号 *N*,并向多数 Acceptor 发送一个编号 *N* 的 Prepare 请求;(b) 如果 Acceptor 收到一个编号 *N* 并且 *N* 大于其已响应的任何 Prepare 请求的编号,则 Acceptor 对该请求做出响应,承诺不接受小于 *N* 的任何提案,并以其接受的最大编号的提案(如有)的内容做出响应.
- 第 2 阶段:(a) 如果 Proposer 收到多数 Acceptor 对其编号 *N* 的 Prepare 请求的响应,则其向这些 Acceptor 发送一份编号为 *N*、特定值为 *v* 的 Acceptor 请求,其中,*v* 是响应中编号最高的提案的特定值,或是如果响应中没有任何信息则任意设置;(b) 如果 Acceptor 收到编号 *N* 的 Acceptor 请求,则其接受该提议并发送 Accepted 响应,除非其已经对编号大于 *N* 的 Prepare 请求做出响应.

算法描述见表 1,在经历若干轮次投票之后,若存在一个议会系统的所有 *Acceptors* 发送 Accepted 请求,整个系统就选出了的特定值,所有进程可以开始获取并学习这个选举结果,这个过程称为学习阶段,并未在流程表中反映.

Table 1 Description of Basic Paxos

表 1 Basic Paxos 描述

	Proposer	*Acceptor*
第 1 阶段	Prepare 请求:按照 *N* 获取算法,获取新的 *N* 满足递增唯一性,发起 *Prepare*(*N*)	*Promise* 响应:若 *N*≤*PromiseMaxBal*,不响应或响应 error.若 *N*>*PromiseMaxBal*,令 *N*=*PromiseMaxBal* 并且响应(*N*,*AcceptedMaxBal*,*AcceptedMaxVal*)或者(*N*,*null*,*null*)
第 2 阶段	*Accepte* 请求:若收到大多数(超过半数)的 *Acceptors* 对 *Prepare*(*N*)的响应,则对这些 *Acceptors* 发起 *Accept*(*N*,*V*),其中,*V* 设为收到的所有响应中最大的 *AcceptedMaxBal* 对应的 *AcceptedMaxVal*;否则,*V* 任意选定一个值.若响应未超过半数,则回到第 1 阶段,重新获取 *N*,发起 *Prepare* 请求	*Aeecpted* 响应:若 *N*=*PromiseMaxBal*,接受提案,设置 *N*=*PromiseMaxBal*,*AcceptedMaxBal*=*N*,*AcceptedMaxVal*=*V*,并响应(*N*,*AcceptedMaxVal*).若 *N*<*PromiseMaxBal*,则不响应或者响应 error

我们从流程中可以总结出 Paxos 的三大前提条件,分别为编号条件、议会系统条件和特定值条件.

- 所有投票轮次的编号都不相同,即编号条件;
- 所有投票轮次的议会系统交集不为空,即议会系统条件;
- 每个投票轮次的特定值是组成其议会系统的所有参与者之前投票过的最大编号(最近)的投票轮次的特定值,即特定值条件.

我们将在下面的章节形式化地建模 Paxos,包括抽象所有系统类型、系统行为以及三大前提条件,并在此基础上验证 Paxos 的一致性.

2 基于 Coq 的形式化验证

2.1 Coq简介

在形式化研究领域,Coq 是最受欢迎的交互式定理证明器之一[21].它允许数学断言的表达式,机械性地检查这些假设的证明,帮助寻找形式化的证明,并从构造性的证明中输出一个关于它的形式化规范的被证明有效的程序.Coq 建立在归纳构建演算理论基础上,可以概括为扩展了归纳和余归纳类型的λ演算,以及一种描述数学定义和证明的语言[1].两个方面实际上是相关的,这要归功于著名的 Curry-Howard De Bruijn 同构,它将命题映射到类型,并将证明映射到函数化对象或强规范化程序.

我们用简单的对自然数的插入排序来说明一些概念.*nat* 是最简单的归纳类型之一,它有构造子 *O* 和 *S*;*nat* 有其自身的类型 Set,具体的数据结构如下.

```
Inductive nat: Type:=
  |O
  |S (n: nat).
```

nat 的定义描述了集合 *nat* 中的表达式是如何构造的:*O* 和 *S* 是构造子;表达式 *O* 属于集合 *nat*;如果 *n* 是属于集合 *nat* 的表达式,那么 *S n* 也是属于集合 *nat* 的表达式;并且只有按照这两种方式构造的表达式才属于集合 *nat*.接下来,我们可以定义在参数类型为 *nat* 集合上的递归函数式程序.

```
Fixpoint insert (i: nat) (l: list nat):=
  match l with
  |nil⇒i::nil
  |h::t⇒if i⩽? h
    then i::h::t else h::insert i t
  end
Fixpoint sort (l: list nat):=
  match l with
  |nil⇒nil
  |h::t⇒insert h (sort t)
  end
```

其中,*i* 是 *nat* 类型的参数;*l* 是 *list* 类型的参数,表示的是自然数列表;*insert* 函数是把 *i* 插入到 *l* 的合适位置中;*sort* 函数是递归地对列表里的每一项进行 *insert*.很显然,*sort* 函数实现了对 *l* 的排序,但是如何去证明经过 *sort* 函数作用之后 *l* 确实是有序的,即我们要证明如下引理:

```
Lemmasort_sorted: forall l, sorted (sort l).
```

其中,*sorted* 是我们要规范说明的有序概念,也可称作谓词.我们要证明 *sort* 算法是正确的,也就是要证明对于任意一个列表 *l*,对它进行 *sort* 后产生的列表是有序的.我们定义一下有序的概念.

```
Inductive sorted: list nat→Prop:=
  |sorted_nil: sorted nil
  |sorted_one: forall x, sorted (x::nil)
  |sorted_cons: forall x y l,
    x⩽y→sorted (y::l)→sorted (x::y::l).
```

这种归纳定义的命题在 Coq 中很常见,并且非常容易理解.上述简单的例子说明了 Coq 类型理论的一些重要特征.Coq 为用户提供了用这些类型、具体对象以及基于这些对象的函数来精确描述数学定义,然后以交互的方式进行证明.总之,Coq 的特性允许我们在一个类型化的、精确的环境中形式化数学理论,而且这种定义基本上是构造性的,这种定义能够为证明提供更多的信息,所以大多数 Coq 用户尽可能地不使用排中律.

2.2 准备工作

通过第 1 节对 Basic Paxos 算法的描述,我们清楚地知道了算法的流程.为了便于形式化的建模以及证明,我们需要将算法进行抽象.明显地,Paxos 算法是一个基于消息传递机制的选举共识算法,系统进程即算法参与者在每一轮次的投票环节分别对特定值进行投票.我们知道,整个算法主要包括 3 个基础类型:参与者、轮次编号、被投票的特定值.依据文献[7],参与者与被投票的特定值只需要可以进行区别即可;轮次编号之间必须不同且可以比较大小,即能够满足唯一性和全序性.为了简化建模过程,并考虑到我们经常使用的自然数完美地满足这两个性质,所以对于这 3 个类型,我们都可以用自然数定义以满足 Paxos 的实际情况需求.因此,我们将参与者、轮次编号、被投票的特定值分别定义为 3 个类型:*priest*、*number*、*decree*,具体的形式化定义如下.

```
Inductive priest: Type:=
  |priestId: nat→priest.
Inductive decree: Type:=
  |decreeId: nat→decree.
Inductive number: Type:=
  |numberId: nat→number.
Definition blt_number (x1 x2: number):=
  match x1, x2 with
  |numberId n1, numberId n2⇒ltb n1 n2
  end
```

通过第 1 节对算法的描述,我们涉及到了许多属于和包含的概念,所以在建模的过程中,不可避免地要处理有关有限集合的问题.我们在 Coq 标准库的基础上,对有限集合的一些定义以及引理做了一些补充.因为算法中有关集合的处理都是对于 *priest* 的操作,所以我们接下来应该都不考虑多态性,直接以类型 *priest* 进行定义.

首先,因为算法的前提条件中涉及到对于空集的描述,所以我们必须在标准库的基础上对空集及其性质进行定义并证明,例如定义了空集不包含任何元素的引理及其两个推论.然后,我们定义了集合之间的包含 *subset* 与真包含 *sincl* 关系,并且对这两种关系之间的性质做了一些证明.具体的形式化定义如下.

```
Lemma empty_spec: forall x, ~(set_In x (empty_set priest)).
Lemma empty_spec_iff: forall x, set_In x (empty_set priest)↔False.
Lemma empty_spec_mem: forall Q, (exists x, set_mem Aeq_dec x Q=true)→
  Q<>empty_set priest.
Lemma double_inclusion: forall u v, subset u v→subset v u→u=v.
```

2.3 投票行为和消息抽象

在每一轮的投票中,参与者都是对这轮投票的唯一特定值进行投票,并且选择投票或者放弃.为了将这一过程的所有信息集中在一起便于分析与证明,我们在 Coq 中运用 Record 方法将一轮投票抽象为一个 *Ballot* 类型,其包括 *dec*,*qrm*,*vot* 以及 *bal* 这 4 个信息,分别表示此轮投票特定值、议会系统、最终投票者以及轮次编号.举个例子,第 8 轮为特定值 *v* 投票,共有参与者 *PA*,*PB*,*PC*,其中 *PC* 不投,*PA* 和 *PB* 投票了,则这一轮投票 *Ballot* 的编号 *bal* 为 8,特定值 *dec* 为 *v*,参与者 *qrm* 为集合[*PA*,*PB*,*PC*],投票者 *vot* 为集合[*PA*,*PB*].

```
Record Ballot: Type:=mkBallot
{
  dec: decree;
  qrm: set priest;
  vot: set priest;
  bal: number;
}.
```

同时,在 Paxos 算法中,所有的信息都是以消息的形式在各个参与者之间进行传递,所以对消息的抽象是十分重要的.通过分析第 1 节的表 1,很明显地知道整个算法分为两大阶段,每一次请求或者响应都是一次消息传递:第 1 阶段(a)的 *Prepare* 请求中包含了轮次编号;第 1 阶段(b)的 *Promise* 响应中包含了轮次编号、参与者以及其已经参与过的最大轮次编号和特定值;第 2 阶段(a)的 *Accept* 请求包含了轮次编号以及此次投票选出来的特定值;第 2 阶段(b)的 *Accepted* 响应包含了响应者响应的轮次编号以及特定值.为了便于建模之后的分析,我们利用 Coq 系统的强类型特征,将 4 个消息全部定义为类型 *Message*,若某一阶段的消息不包含某一项,设为缺省默认值即可,具体的如下所注释.在之后的定义和证明中,因为 Coq 的类型检查,不会带来混乱.

```
Record Message: Type:=mkMessage
```

{

typeM: *nat*; (*消息的类型.我们以 1~4 分别代表 4 个小阶段的 4 种消息.*)

balM: *number*; (*消息的轮次.*)

maxBalM: *number*; (*消息的轮次.只有 2-类型消息有此项.*)

maxValM: *decree*; (*消息的特定值.2-类型表示其曾经参与的最大特定值,3-类型和 4-类型表示现阶段进行投票的特定值,1-类型没有此项.*)

accM: *priest*; (*消息的发送者.2-类型和 4-类型的消息表示发送者身份,1-类型和 3-类型消息没有此项.*)

}.

2.4 系统状态和三大前提条件抽象

我们将可供所有的投票轮次选择的轮次编号、特定值分别定义为 *Numbers*,*Values*.另外,在整个系统初始状态时还没有任何投票轮次发生,我们定义一个特殊的特定值 *None*,来满足表 1 中 *Promise* 响应内容为 *null* 的要求.同时,我们定义系统中所有的投票轮次为 *aBallots*,因为 Paxos 采用的是非拜占庭的故障模型,所以所有存在的投票轮次的信息都是正确的.显然地,系统中所有的参与者组成了集合 *Acceptors*(文献[12]中还存在 Client 和 Leaner 角色,但是对于算法验证不重要,我们在这里省略其建模),每一轮次投票的议会系统 *Quorum* 都是 *Acceptors* 一个子集,而且必须满足议会系统条件;所有轮次的议会系统组成了集合 *Quorums*.具体的形式化定义如下.

Variables *Numbers*: *list number*.

Variables *Acceptors*: *set priest*.

Variables *Values*: *set decree*.

Variables *None*: *decree*.

Variables *aBallots*: *set Ballot*.

Variables *Quorums*: *set* (*set priest*).

对于编号条件,我们将其形式化定义为 *Unique_Ballot*,显然地,其说明系统中任何两个投票轮次的投票编号相等则其一定为相同的投票轮次,反之亦成立.BasicPaxos 算法另外一个非常重要的定义就是 *Quorum*,它表示在一轮投票中,所有参与者的集合即 *priest* 集合,而且其必须满足议会系统性质.简而言之,就是每个 *Quorum* 集合的元素个数必须是 *Acceptors* 集合的元素个数的半数以上,并且两个不同的 *Quorum* 集合之间必须有交集.这就是 Paxos 算法为了满足容错性而进行的规定,具体的数学定义在第 1 节给出过.我们将所有 *Quorum* 集合组成的集合定义为 *Quorums*,我们规定在每一轮次的投票中,所有参与者组成的集合都是 *Quorums* 的一个元素,故其一定满足 Paxos 对于大多数参与者的要求.我们将其定义为 *QuorumsAssumption*,即系统中的任意两个议会系统的交集不为空.具体的形式化定义如下.

Hypothesis *Unique_Ballot*: **forall** *x y*: *Ballot*,

In *x aBallots*→In *y aBallots*→*x*=*y*↔(*bal x*)=(*bal y*).

Hypothesis *eq_dec_Ballot*: **forall** *x y*: *Ballot*,

{(*bal x*)=(*bal y*)}+{*blt_number* (*bal x*) (*bal y*)=true}.

Hypothesis *QuorumsAssumption*: **forall** *Q*1 *Q*2: *set priest*,

In *Q*1 *Quorums*∧In *Q*2 *Quorums*→*exists q*,

*set*_In *q* (*set_inter priest_eq_dec Q*1 *Q*2).

通过仔细分析特定值条件,我们发现其表述的此轮投票轮次之前的所有投票轮次的相关信息.如果通过定义函数得到过往投票轮次的相关信息,整个建模和验证将变得非常麻烦和困难.所以,我们考虑在算法流程中实时地更新记录每个参与者的相关过往投票信息——*PromisePromiseMaxBal*,*AcceptedPromiseMaxBal* 和 *AcceptedMaxVal* 这 3 个字段值的更新,其中,*PromisePromiseMaxBal* 表示的是某个参与者参与过的最大的投票轮次,注意到 *Promise* 子过程也算作参与了,所以其会在 *Promise* 阶段和 *Accepted* 阶段都更新;

AcceptedPromiseMaxBal 和 *AcceptedMaxVal* 可以当作一个整体,表示某个参与者在某个投票轮次中进行了投票,其中,*AcceptedPromiseMaxBal*,*AcceptedMaxVal* 分别表示这个投票轮次的编号和特定值,只会在 *Accepted* 阶段更新.具体的形式化定义如下:

Variables *PromisePromiseMaxBal*: *priest*→*number*.

Variables *AcceptedPromiseMaxBal*: *priest*→*number*.

Variables *AcceptedMaxVal*: *priest*→*decree*.

最后,我们需要强调的是:Basic Paxos 采用的是非拜占庭的故障模型,即进程以任意速度运行,可能因停止而失败,并且可能重新启动;消息传递可能需要任意长的时间,可以复制,也可以丢失,但它们没有被损坏.所以,上述定义过的系统状态相关的值,只要存在,则其所包含的信息一定是正确的.考虑到这个系统性质,我们设置了一个全局消息通道来解决并简化发送响应动作的形式化定义.因为系统会发生故障,所以如何去表征每个参与者在系统中是否处于正常状态也是困难的,也就很难形式化地定义执行发送响应时参与者应该处于何种状态.但是,如果一个参与者执行了发送动作,系统中一定存在着对应这个动作的消息;同时,如果一个参与者对某条消息做出了响应,一定可以说明它收到了对应的请求(如果没有做出响应,并不能说明它没有收到请求,因为它或者消息都可能遇到故障,也可能它收到请求但是选择不参与投票).所以,通过对全局消息通道的内容进行检查,可以确定每个参与者在系统中是否执行了发送动作,以及如果执行了发送动作,我们也能确定其发送的内容.因此,形式化定义(2)时,我们通过检查全局消息通道来定义,从而不考虑发送动作并简化之后的证明.全局消息通道的形式化定义如下.

Variables *msgsChannel*: *set Message*.

2.5 Paxos实例

在准备工作和投票行为建模完成之后,我们以文献[7]中的一个具体的算法例子来说明 Paxos 的实际情况和形式化定义之间的一致性.如表 2 所示,系统中包含 5 个参与者 *A*,*B*,*C*,*D* 和 *E*,并且一共进行了 5 个轮次的投票,其中,*quorum* 即议会系统是每个轮次中实际的参与者,红色标记的是投票了的参与者,*voters* 是 *quorum* 的子集.显然地,5 个投票轮次的编号都不一样,并且议会系统都包含大多数的参与者,保证其交集不会为空,所以编号条件和议会系统条件都是满足的.接下来,我们来具体地分析如何满足特定值条件.

- 编号 2 是最早的投票轮次,所以其特定值条件是平凡的,即可以为任何值.
- 编号 5 的投票轮次,其议会系统中的参与者没有在之前的投票轮次中投过票,故其特定值也是平凡的.
- 编号 14 的投票轮次,只有参与者 *D* 在编号 2 的投票轮次中投过票,所以已经特定值条件,编号 14 的投票轮次的特定值与编号 2 的投票轮次的特定值相等.
- 编号 27 的投票轮次,其议会系统中的参与者 *A* 没有在之前投过票,参与者 *C* 只在编号 5 的投票轮次中投过票,参与者 *D* 只在编号 2 的投票轮次中投过票,所以最大的是编号 5 的投票轮次,依据特定值条件,编号 27 的投票轮次与编号 5 的投票轮次的特定值相等.
- 编号 29 的投票轮次,其议会系统中的参与者 *B* 只在编号 14 的投票轮次中投过票,参与者 *C* 在编号 5 和编号 27 的投票轮次中投过票,参与者 *D* 在编号 2 和编号 27 的投票轮次投过票,所以最大的是编号 27 的投票轮次,依据特定值条件,编号 29 的投票轮次与编号 27 的投票轮次特定值相等.

Table 2 Message flow of Basic Paxos

表 2 Basic Paxos 的消息流

Number	Decree	Quorum and voters				
2	α	*A*	*B*	*C*	*D*	
5	β	*A*	*B*	*C*		*E*
14	α		*B*		*D*	*E*
27	β	*A*		*C*	*D*	
29	β		*B*	*C*	*D*	

所以经过投票行为和消息抽象之后,我们确保了 Paxos 的实际情况与形式化定义的一致性.例如,编号 29 的

投票轮次可以抽象为一个 *Ballot*,其中,*bal* 为 29,*qrm* 为 *B*,*C*,*D* 组成的参与者集合,*vot* 为 *B* 组成的参与者集合;在这一轮投票中,每个参与者会在系统中产生对应的消息,包含消息的类型,此次消息对应的投票轮次,最大投票轮次的编号和特定值信息以及参与者信息,这些消息内容与形式化定义的 *Message* 是一一对应的;最后,我们通过系统中的消息内容,可以确定编号 29 的投票轮次的特定值 *decree* 为 β.

2.6 算法建模

在前面的工作中,我们对算法涉及的基础类型、投票轮次、消息机制以及三大前提条件进行了形式化的抽象和定义,我们将在此基础上,对系统如何选定一个特定值即公式(2)以及消息流程进行形式化定义.

首先,我们要精确地定义一个特定值在满足特定条件时才表示其被选择了.为了说明这一个过程,我们将其拆解为 3 个定义.在对全局消息通道介绍的时候,我们已经说明了可以通过形式化定义对全局消息通道的检查来代替形式化定义 *sent* 动作.通过对消息流程的分析,一个参与者 *priest* 对一个特定值投票是发生在 *Accepted* 响应中,所以在全局消息通道中,必然存在一个消息类型为 4 的消息,并且表明它对这一轮次投票的特定值投票了,*VotedForIn* 就是对这一行为结果的形式化定义.

```
Inductive VotedForIn: priest→decree→number→Prop:=
  |cons_VotedForIn: forall a v b, In a Acceptors→In v Values→In b Numbers→
    (exists m, In m msgsChannel∧(typeM m)=4
      ∧(balM m)=b
      ∧(maxValM m)=v
      ∧(accM m)=a)→VotedForIn a v b.
```

VotedForIn 只是说明了一个参与者投票了,如果某一轮次的所有参与者都投票了,则代表这是一次成功的投票,是一个成功投票轮次的性质说明.如果在所有轮次的投票中,存在一轮成功的投票,则表示这一轮成功的投票的特定值被选择了,这是系统真实存在这样一个成功投票轮次的说明.

```
Inductive SuccessfulIn: decree→number→Prop:=
  |cons_SuccessfulIn: forall v b, In v Values→In b Numbers→
    (exists Q, In Q Quorums→
      forall a, set_In a Q→VotedForIn a v b)→SuccessfulIn v b.
Inductive Chosen: decree→Prop:=
  |cons_Chosen: forall v, In v Values→
    (exists b, In b Numbers→SuccessfulIn v b)→Chosen v.
```

SuccessfulIn 表示的是“成功的投票”这一性质,即一次投票的全部参与者都对这一轮的特定值投票了.*Chosen* 表示的是“成功的投票”这一性质的存在性,即存在一轮投票,其投票轮次为 *b*,满足这个性质.很显然地,这 3 个重要的归纳定义是对公式(2)的拆分定义.而且根据系统消息的不可篡改性质,全局消息通道中的消息内容一定是所有参与者发送的,所以其保证了系统中最后选择的特定值一定是系统参与者发送的,即满足共识性安全要求中的“只能选择已提出的值”.

在这里,我们需要进一步分析从一次投票行为 *Ballot* 可以得到的结论.显然地,任意一次投票行为 *Ballot* 的 *qrm* 集合和 *vot* 集合都是 *Acceptors* 集合的子集,而且 *vot* 集合也是 *qrm* 集合的子集.同时,根据 *vot* 集合的定义,它包含的所有参与者都在一轮投票行为中投票,所以在系统消息中,一定存在一条对应其投票行为的 4-类型消息.系统中任意两个不同的投票行为 *Ballot* 的 *bal* 是不同的,并且 *bal*,*dec* 分别是 *Number*,*Value* 的元素.这些平凡性质不属于附加前提,都可以将其形式化定义,并且在证明中是可以直接引用的,用来说明存在性和合理性,我们将其形式化定义为 *trivial*,因为比较简单,我们不在论文中列出.

另外,我们在前面提到过,对于所有参与者其只能选择投票或者不投票,我们定义参与者没有在某一轮次投票,即 *VotedForIn* 的否定形式,并且以后也不会在这一轮次投票,即其 *PromiseMaxBal* 一定是大于这个投票轮次的轮次编号的,我们将此性质形式化定义为 *WontVoteIn*.同时,如果某一投票轮次进行到 *Accept* 阶段,对于这个投

票轮次有一个安全状态,即在这一轮次编号为 *N* 的投票过程中,对于特定值 *V* 是安全的表示在任何轮次编号小于 *N* 的投票轮次中,除了 *V* 以外,没有别的特定值被选择,或者永远不会被选择.这个定义其实就是对系统中所有 *priest* 行为的一个规范,我们将此性质形式化定义为 *SafeAt*.*WontVoteIn* 和 *SafeAt* 具体定义如下.

```
Inductive WontVoteIn: priest→number→Prop:=
  | cons_WontVoteIn: forall a b, In a Acceptors→In b Numbers→
      (forall v, In v Values→~VotedForIn a v b)
    ∧blt_number b (PromiseMaxBal a)=true→WontVoteIn a b.
Inductive SafeAt: decree→number→Prop:=
  | cons_SafeAt: forall v b, In v Values→In b Numbers→
    (forall c, In c Numbers→blt_number c b=true→
    ((exists Q,
    (forall ballot, trivial ballot→In ballot aBallots→(c=(bal ballot))→
    Q=(qrm ballot)∧(forall a, In a Q→VotedForIn a v c∨WontVoteIn a c)
    ))))→SafeAt v b.
```

通过第 1 节对 Paxos 的介绍,我们知道算法一共分为两大阶段,每个阶段细分为两个子过程.同时,Paxos 是一个基于消息传递机制的选举共识算法,涉及到消息的发送与接收.但是我们可以通过检查整个系统中存在的消息,去判断是否发生的消息的传递.例如,如果消息通道中存在一个 1-类型的消息,我们一定可以推断出系统中某个参与者发出了 *Prepare* 请求;同时,因为系统中可能存在各种延时、宕机等,有些参与者是无法收到消息或者发送消息的,所以我们无法去良好地定义接收行为.但是我们可以依据消息通道中是否存在 2-类型的消息去判断一个参与者对一轮投票进行参与,因为只有在参与者接收到 *Prepare* 请求时才会发送 *Promise* 请求;类似地,我们也可以根据全局消息通道中是否存在 4-类型的消息去判断一个参与者是否对某一轮次投票的特定值进行了投票,因为参与者只有收到 *Accept* 请求之后才会发送 *Accepted* 响应.通过这个全局消息机制,我们解决了形式化定义消息发送与接收的困难点,不用去复杂地定义系统客观存在的故障和异常,还成功地体现了 Paxos 实际运用中会遇到的各种消息问题,并且保证了形式化定义的合理性和准确性,所以这个形式化定义更加形式化和构造性.我们根据算法描述定义系统中的所有消息性质,我们先给出其归纳定义然后进行说明.

```
Inductive MsgInv: Prop:=
  |cons_MsgInv: (forall m, In m msgsChannel→
    ((typeM m)=1→True)
  ∧((typeM m)=2→less_or_equal_number (balM m) (PromiseMaxBal (accM m))
              ∧(In (maxValM m) Values∧In (maxVBalM m) Numbers
                  ∧VotedForIn m (accM m) (maxValM m) (maxVBalM m)
                ∨((maxValM m)=None∧(maxVBalM m)=numberId 0)))
  ∧((typeM m)=3→(SafeAt m (maxValM m) (balM m)
              ∧forall ma, set_In ma msgsChannel→(typeM m)=3→
                  (balM m)=(balM ma)→(maxValM ma)=(maxValM m)))
  ∧((typeM m)=4→(less_or_equal_number (balM m) (maxVBal (accM m))
              ∧forall ma, set_In ma msgsChannel→(typeM ma)=3
                  ∧(balM ma)=(balM m)
                  ∧(maxValM ma)=(maxValM m))))→MsgInv.
```

通过观察上面对 *MsgInv* 的归纳定义,很明显地看出我们是根据消息类型对其进行归纳定义,所有在全局通道中的消息行为都必须满足 *MsgInv* 规范.因为 1-类型消息是每个轮次的发起阶段,系统中的每个参与者都可以随时发起 *Prepare* 请求,请求的轮次编号可以是任意的 *number* 类型值,故系统中的所有 1-类型的消息都是合理

的.某个参与者发出了 2-类型的消息,则其必须满足此轮次投票的编号大于其曾经参与过的投票的最大轮次编号,并且发送最大轮次编号及其特定值,若没有则发送默认值,即特定值为 *None*,编号为 0 的默认消息;如果不满足 2-类型的消息发送要求,则全局消息通道中一定不会出现对应的 2-类型消息.若发出了 3-类型的消息,则其处于一种安全状态,并且对于整个系统中所有 3-类型的消息,如果其编号相等,则消息里的特定值一定是相同的,因为根据编号条件,相同编号的投票轮次一定是同一个投票轮次,故其特定值一定是相同的.若某个参与者发出了 4-类型的消息,则这个消息发送者的最大特定值不小于此消息的特定值,并且在系统所有的消息中,一定存在 3-类型的消息,并且与此消息的轮次编号和特定值相同,因为 4-类型消息是对 3-类型消息的响应,参与者不可能主动发送 4-类型消息.

通过上面的分析,我们确保并坚信对 Paxos 的形式化建模与 Paxos 的实际情况之间的一致性.我们要证明 Paxos 的一致性,就需要在上述形式化建模的基础上验证公式(1)的正确性.

2.7 算法验证

在这一节,我们将在前面建模的结果中,描述一些对一致性定理比较重要的引理及一致性定理的证明.

首先,我们从消息传递的角度来看待算法,在系统中的任意两个参与者,分别发送了两条消息,其中一条消息表示其中一个参与者为轮次编号 *b* 的特定值 *v*1 投票,另一条消息表示另一个参与者同样为相同轮次编号的特定值 *v*2 投票了,根据 *MsgInv* 和 *VotedForIn* 的定义,我们可以推断出这两个特定值 *v*1 和 *v*2 是相等的,我们将此性质形式化定义为引理 *VotedOnce*.我们描述这个推论过程:根据 *VotedForIn* 的定义,我们知道在系统的消息中存在着两条 4-类型消息,又根据 *MsgInv* 的定义,存在 4-类型的消息则系统中一定存在着对应的 3-类型消息,并且对于所有的 3-类型消息,如果轮次编号相同,则消息相同即消息包含的特定值是相同的.其实通过对 *Ballot* 的定义,我们能够更简单地意识到这个问题:根据编号条件,任意两个投票行为 *Ballot* 的投票轮次 *bal* 是唯一的,所以如果两个参与者的轮次编号一致,则代表他们肯定是在同一轮进行的投票,同一轮的特定值肯定是相同的.通过上面的推论分析,我们也能够很快地以同样的方式可以推断得到引理 *VotedInv*.

Lemma *VotedOnce*: **forall** *a*1 *a*2 *b v*1 *v*2,

MsgInv→In *a*1 *Acceptors*→In *a*2 *Acceptors*→

In *b Numbers*→In *v*1 *Values*→In *v*2 *Values*→

*VotedForIn a*1 *v*1 *b*∧*VotedForIn a*2 *v*2 *b*→(*v*1=*v*2).

Lemma *VotedInv*: **forall** *a v b*,

MsgInv→In *a Acceptors*∧In *v Values*∧In *b Numbers*

→*VotedForIn a v b*→*less_or_equal_number b* (*AcceptedMaxBal a*)∧*SafeAt v b*.

介绍完两个最重要的引理之后,我们来定义一致性定理 *Consistent*:对于系统中的任意两个轮次的投票行为 *Ballot*,我们最直接地能够看到每个 *Ballot* 的各个组成部分的内容,如果 *Ballot* 所属的议会系统 *qrm* 是 *vot* 的子集(更直观地说,*qrm* 集合和 *vot* 集合是相等的),可以推论出 *qrm* 的所有参与者都参与了投票,即满足第 1 节公式(2)的定义,这轮的投票行为都成功地选择了一个特定值,所以这个条件与公式(2)是等价的,我们将这种性质定义为引理 *SuccToChosen*,具体的形式化定义如下.为了与投票行为的抽象保持一致,我们将公式(1)中的抽象定义具体定义为一致性定理 *Consistent*.如果两个 *Ballot* 所属的 *qrm* 都是 *vot* 的子集,即这两轮的投票行为都成功地选择了一个特定值,则这两个轮次的特定值是相等的.对于一致性定理,我们可以将其以两个 *Ballot* 是否是同一个投票行为为依据分为两个引理进行证明,两个引理分别为 *ConsistentOfEqual* 和 *ConsistentOfNotEqual*.对于引理 *ConsistentOfEqual* 的证明很容易,因为两轮次投票行为的编号是相同的,根据前面的分析,能够确定就是同一轮投票,故其特定值一定是相等的.接下来给出投票轮次不相等证明的大概思路.

Lemma *SuccToChosen*: **forall** *b a*,

trivial b→*set_In b aBallots*→*set_In a* (*qrm b*)→*subset* (*qrm b*) (*vot b*)→

Chosen (*dec b*).

Lemma *ConsistentOfEqual*: **forall** *b*1 *b*2, *trivial b*1→*trivial b*2→

subset (*qrm b*1) (*vot b*1)→*subset* (*qrm b*2) (*vot b*2)→
(*bal b*1)=(*bal b*2)→(*dec b*1)=(*dec b*2).

Lemma *ConsistentOfNotEqual*: **forall** *b*1 *b*2, *MsgInv*→*trivial b*1→*trivial b*2→
subset (*qrm b*1) (*vot b*1)→*subset* (*qrm b*2) (*vot b*2)→
blt_number (*bal b*1) (*bal b*2)=true→(*dec b*1)=(*dec b*2).

Theorem *Consistent*: **forall** *b*1 *b*2, *MsgInv*→*trivial b*1→*trivial b*2→
subset (*qrm b*1) (*vot b*1)→*subset* (*qrm b*2) (*vot b*2)→
(*dec b*1)=(*dec b*2).

对于定理前提谓语条件即系统中所有成功的投票轮次,其议会系统都是投票者的子集,又依据投票轮次的平凡性质比如其投票的人一定是议会系统的子集,根据集合性质,很容易知道成功投票轮次的议会系统和投票者其实是相等的,即所有参与者在这一轮次投票行为中全部投票了,这也很符合 *SuccessfulIn* 的定义.根据这个议会系统全部投票了的推论进而能得出很多结论.比如,我们可以确定很多存在性,即在系统中,成功的投票轮次都满足 *Chosen* 的规范,以及在所有的消息中,肯定也存在成功轮次相关 4-类型的 *VotedForIn* 消息等.在这些初步的推论中,结合上述的 *VotedForIn* 引理,就能够推论出 *SafeAt* 规范.仔细分析 *SafeAt* 规范,我们发现其涉及到两个投票轮次之间编号的大小比较,所以由此我们很容易确认将一致性定理分为两部分进行证明的思路是正确的.同时,根据 *SafeAt* 和 *VotedForIn* 得到的相关结果进行分析,结合我们已经证明过的引理 *VotedOnce*,就能够得到两轮成功投票行为的特定值是一致的.最后,根据投票行为的平凡性质,我们就能通过这两个引理最终得到一致性定理证明的完整过程.我们需要强调的是:所有的平凡性质虽然简单但都是必须的,它反映了 Paxos 非拜占庭故障模型一系列的基础规范,比如轮次编号和特定值都是类型准确的、投票消息一定会在全局消息通道中真实反映等.

通过上述的规范的数学定义之后,我们对 BasicPaxos 算法所涉及的类型、消息通道以及性质有了一个完整并且严格的形式化定义,并且最终成功地证明了算法的一致性定理,即在整个系统中,两轮成功投票的值一定是相等的.

3 总 结

在整个建模和验证过程中,我们在 Coq 中用 1 092 行代码一共证明了 13 个重要引理以及最终的共识性定理,完整的 Coq 脚本可以在 https://github.com/Liyanan0410/BPOfCoq 中查看并使用.通过对 Basic Paxos 算法的建模与验证,我们进一步理解了分布式共识问题,以及确信 Paxos 算法能够解决问题并且是正确的.通过这次研究,我们确定并坚信在分布式领域,运用形式化方法是能够取得进步的.将来我们还可以对算法模型进一步优化,以及增加一些特定的符号说明来改善证明的可阅读性.同时,因为 Paxos 算法具有很多的变种[22,23],我们将进一步地对这些算法变种进行研究,对整个 Paxos 算法家族做一个整体地建模与验证,以增强算法在实际应用中的信心.

References:

[1] Bertot Y, Castéran P. Interactive Theorem Proving and Program Development: Coq' Art: The Calculus of Inductive Constructions. Springer Science & Business Media, 2013.

[2] Gonthier G. A computer-checked proof of the four colour theorem. 2005. https://www.cl.cam.ac.uk/~lp15/Pages/4colproof.pdf

[3] Paulin-Mohring C. Circuits as streams in Coq: Verification of a sequential multiplier. In: Proc. of the Int'l Workshop on Types for Proofs and Programs. Springer-Verlag, 1995. 216−230.

[4] Leroy X. Formal certification of a compiler back-end or: Programming a compiler with a proof assistant. ACM SIGPLAN Notices, 2006,41(1):42−54.

[5] Affeldt R, Kobayashi N. Formalization and verification of a mail server in Coq. In: Proc. of the Int'l Symp. on Software Security. Springer-Verlag, 2002. 217−233.

[6] Deng YX, Monin JF. Verifying self-stabilizing population protocols with Coq. In: Proc. of the 3rd IEEE Int'l Symp. on Theoretical Aspects of Software Engineering. IEEE, 2009. 201−208.

[7] Lamport L. The part-time parliament. ACM Trans. on Computer Systems, 1998,16(2):133−169.

[8] Burrows M. The Chubby lock service for loosely-coupled distributed systems. In: Proc. of the 7th Symp. on Operating Systems Design and Implementation. USENIX Association, 2006. 335−350.

[9] Isard M. Autopilot: Automatic data center management. ACM SIGOPS Operating Systems Review, 2007,41(2):60−67.

[10] Zheng JJ, Lin Q, Xu JT, Wei C, Zeng CW, Yang PA, Zhang YF. Paxos store: High-availability storage made practical in wechat. Proc. of the VLDB Endowment, 2017,10(12):1730−1741.

[11] Lamport L. My writings. 2019. http://lamport.azurewebsites.net/pubs/pubs.html#lamport-paxos

[12] Lamport L, *et al*. Paxos made simple. ACM SIGACT News, 2001,32(4):18−25.

[13] Lamport L. Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers. Addison-Wesley, 2002.

[14] Microsoft Research-Inria Joint Center. TLA+Proof System (TLAPS). 2019. http://tla.msr-inria.inria.fr/tlaps

[15] Chand S, Liu YA, Dstoller S. Formal verification of multi-Paxos for distributed consensus. In: Proc. of the Int'l Symp. on Formal Methods. Springer-Verlag, 2016. 119−136.

[16] Kellomäki P. An annotated specification of the consensus protocol of Paxos using superposition in PVS. Technical Report, No.36, Institute of Software Systems, Tampere University of Technology, 2004.

[17] Jaskelioff M, Merz S. Proving the correctness of disk Paxos. In: The Archive of Formal Proofs, 2005. http://afp.sf.net/entries/DiskPaxos.shtml

[18] Padon O, Losa G, Sagiv M, Shoham S. Paxos made EPR: Decidable reasoning about distributed protocols. In: Proc. of the ACM on Programming Languages 1 (OOPSLA), 2017. 108:1−108:31.

[19] García-Pérez Á, Gotsman A, Meshman Y, Sergey I. Paxos consensus, deconstructed and abstracted. In: Proc. of the European Symp. on Programming. Cham: Springer-Verlag, 2018. 912−939.

[20] Lamport L. Byzantizing Paxos by refinement. In: Proc. of the Int'l Symp. on Distributed Computing. Springer-Verlag, 2011. 211−224.

[21] The Coq Development Team. The Coq Proof Assistant Reference Manual. 2019. https://coq.inria.fr/distrib/current/refman

[22] Lamport L. Fast Paxos. Distributed Computing, 2006,19(2):79−103.

[23] Chandra TD, Griesemer R, Redstone J. Paxos made live: An engineering perspective. In: Proc. of the 26th Annual ACM Symp. on Principles of Distributed Computing. ACM, 2007. 398−407.

李亚男(1992－),男,硕士,主要研究领域为形式化方法.

邓玉欣(1978－),男,博士,教授,博士生导师,CCF 高级会员,主要研究领域为形式化方法.

刘静(1964－),女,博士,教授,博士生导师,CCF 专业会员,主要研究领域为可信软件,模型驱动式软件开发方法,面向服务的软件架构.