Parliament: a module for parliamentary procedure software

Bayle Shanks
Computational Neurobiology
University of California, San Diego
9500 Gilman Dr
La Jolla, CA 92093
bshanks at ucsd.edu

Dana Dahlstrom
University of California, San Diego
9500 Gilman Dr
La Jolla, CA 92093
dana at cs.ucsd.edu

ABSTRACT
Parliamentary procedure is a widely used system of rules for group decision making. We describe a reusable software module, *Parliament*, that implements the logic and bookkeeping of parliamentary procedure, given a precise specification of the rules. Parliament is designed to be embedded in applications, such as to support face-to-face meetings or to facilite computer-mediated online deliberation. As a demonstration and testbed, we have created a partial working specification of Robert's Rules of Order and an application equipped with a graphical user interface for use during face-to-face meetings.

Keywords
Robert's Rules, Robert's Rules of Order, parlimentary procedure, meeting, meeting systems, meeting management, group decision support systems, GDSS

1. INTRODUCTION
Parliament is an open-source software module written in Python[1] that can be used to build programs that follow or moderate the conduct of a deliberative assembly using parliamentary procedure. Parliament encapsulates logic and bookkeeping functions necessary for the function of parliamentary procedure, and can be embedded in applications for face-to-face meetings, or for synchronous or asynchronous computer-mediated communication.

Parliament's central functions track meeting state, such as pending motions, the relationships among them, and business already transacted. The outer application is responsible for informing the Parliament module about events as they occur in the meeting, such as "this person made motion X" or "motion Y failed." Parliament answers queries such as "Which motions are presently valid?" or "Which motions have carried in this meeting?" Parliament is also capable of answering questions about hypothetical situations, such as "Which motion will be pending if this one carries?"

The Parliament module does not incorporate the details of parliamentary procedure, such as the motions and customs described in Robert's Rules of Order [4]. Instead, Parliament requires an external rule specification, allowing the user or developer to modify the rules independently and even to develop whole new rule systems.

[1]http://python.org/

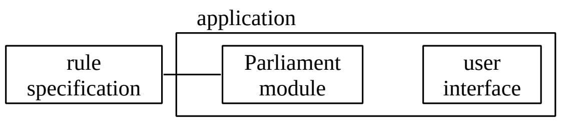


Figure 1: The Parliament module is embedded in an application, and uses an external rule specification.

This paper discusses the design and usage of Parliament and of the rule-specification language. It presents data structures for representing the state of parliamentary meetings. The paper describes a simple "Robert's Rules meeting assistant" built using Parliament. It also explains how Parliament could be used to run online meetings and to support decision-making in online communities such as wiki or IRC.

2. MOTIVATION
A desire for flexibility and reuse motivates modularity and abstraction in our design.

2.1 A reusable module
Many conceivable applications could share a common software implementation of parliamentary procedure, such as programs to support face-to-face meetings, whether to assist a parliamentarian, the chair, the secretary, or an individual participant. Similarly, an application could be used outside of meetings to train people in the use and applications of the rules[2]. Other examples include:

- **Online, synchronous meetings**
  - A networked application which participants use to request the floor and to make and vote on motions
- **Online, asynchronous meetings** (could be WWW, IRC, or other)
  - An application to assist a human chair
  - An application to automatically chair a meeting

[2]Non-modular parliamentary procedure training software called *Robert's Rules in Motion* is available at http://imovethat.com/

  – An application to moderate a large discussion board or wiki according to formal meeting rules
  – An application to automatically update a set of organizational bylaws according to the instructions of an online deliberative assembly

With so many separate applications, it would be inefficient to reimplement the core logic of parliamentary rules and sets of motions for each application. A reusable module could simplify these implementation efforts, and if several applications were built using the module, these applications could share rule specifications in a common format.

2.2 Modular rule specifications

Robert's Rules of Order, Newly Revised is the most common choice of rulesets, but there are others, for example, the public domain version of Robert's Rules of Order, and the Standard Code of Parliamentary Procedure. Each branch of the U.S. congress uses its own ruleset which is similar but not identical to Robert's Rules.

Rather than choosing a specific set of parliamentary rules and "hard-coding" those rules into the module, we opted to create a module flexible enough to accommodate many different parliamentary rulesets. The user or developer specifies the ruleset in a special ruleset specification language, and the specification is loaded into the Parliament module at runtime (see Fig. 1). There are many advantages to this approach:

1. Different deliberative assemblies use different meeting rules. Robert's Rules of Order is the most common ruleset, but there are many variations of Robert's Rules and many other rulesets.

2. Unconventional meeting settings such as the WWW or IRC will probably demand new innovations in parliamentary rules. A flexible module is necessary to adapt to these types of meeting.

3. Allowing the ruleset to be modified gives the assembly complete flexibility to adapt the software to their needs. Assemblies should not be forced to follow a particular set of meeting rules just because their software can't support the rules that they would really prefer.

4. Research on group decision-making support systems (GDSS) is hindered by the difficulty of isolating the effect of individual components of the group decision-making process. A configurable parliamentary ruleset will serve as the ideal platform for testing fine-grained modifications to a group's process.

3. DESIGN OF THE MODULE

Parliament is object-oriented. Each motion made during a meeting is represented by an object, the type of which is a class that encapsulates relevant characters of that kind of motion.

In this section we will begin by introducing the data structures used to represent aspects of parliamentary procedure and meeting state. This will allow us to describe the Parliament application programming interface (API). Finally, we will discuss the language used for specifying parliamentary rulesets.

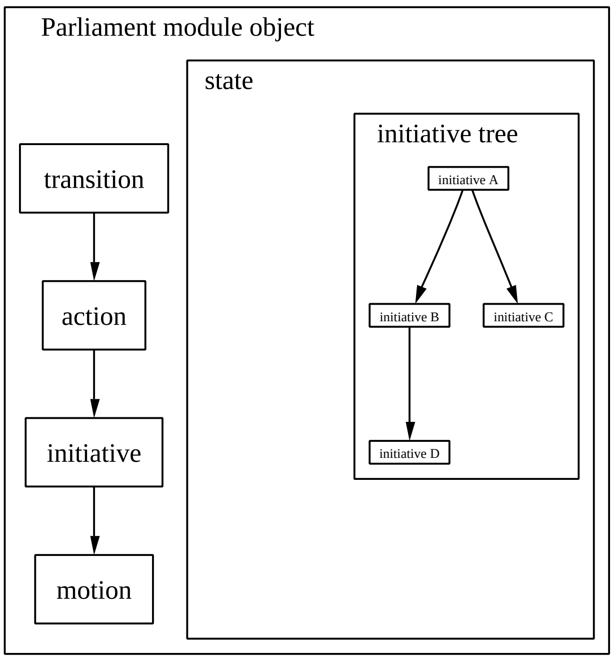


Figure 2: The relationships between the primary types of data structures. All of these data objects are stored within the Parliament object. On the left hand side, the class inheritance diagram shows that "motion" is a subclasses of "initiative", which is a subclasses of "action", which is a subclass of "transition". On the right hand side, we see that objects of class "state" contain initiative trees, which are trees of objects of class "initiative".

3.1 Data structures

The major data structures in Parliament can be separated into two groups: states and state transitions.

States

At any given time, Parliament assigns the meeting a certain *state*, which is realized as an object of class "state". The history of a meeting is the history of its progression through various states. At the beginning of the meeting, the meeting is in a preset "initial state". States are "Markovian" in that the current state of the meeting contains all of the information about the meeting history which the parliamentary rules might need to know about. Keeping a list of past states allows Parliament to support multiple undo/redo.

State transitions: actions, initiatives, motions

Any event that causes a change of state is recorded as an object of class *transition*. The class of the transition object indicates which type of event occurred. For example, passing a motion is recorded as a transition of type "motionPassedTransition". After passing a motion, a new state object is created, the "current state" is set to point to the new state, and the new state is appended to the list of past states.

The most common type of transition is an *action*. An action is something which is usually done by a specific participant in the meeting. The class "action" is a subclass of "transition".

The most common type of action is an *initiative*. An initiative is loosely defined as a topic or proposal which affects the legal actions available to meeting participants until the initiative is disposed of. Examples of initiatives are making a motion, beginning a discussion about some topic, or entering a discussion phase such as "brainstorm phase". The class "initiative" is a subclass of the class "action".

One class of initiative is a *motion*, such as the motions in Robert's Rules. Motions are initiatives which can be passed or failed by the assembly. The class "motion" is a subclass of the class "initiative".

The initiative tree

When an initiative is initiated, a new state object is created and appended to the meeting history list. However, the parliamentary logic will typically need access to some information about which initiatives have come before. Therefore, each state contains a partial history of the meeting, in the form of an *initiative tree*. Eventually, as initiatives are disposed of, they are removed from the initiative tree.

Figure 2 summarizes the relationships between the classes of data objects.

Initiatives, motions and the initiative tree: example

For example, at the start of a Robert's Rules meeting, the current state is "Initial state". The initiative tree inside the state "initial state" consist only of a single initiative, "root initiative". Fig. 3 shows what the meeting history looks like at this point:

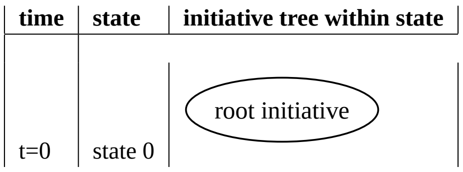


Figure 3: The history of the meeting after time 0

Now, let's say Adam makes a motion to walk the dog. A new state is added to the meeting history. Within the new state, Adam's motion appears as a child of the "root initiative" (fig 4).

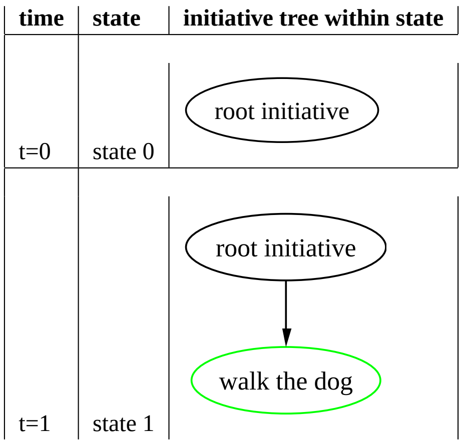


Figure 4: The history of the meeting after time 1; green denotes the immediately pending motion

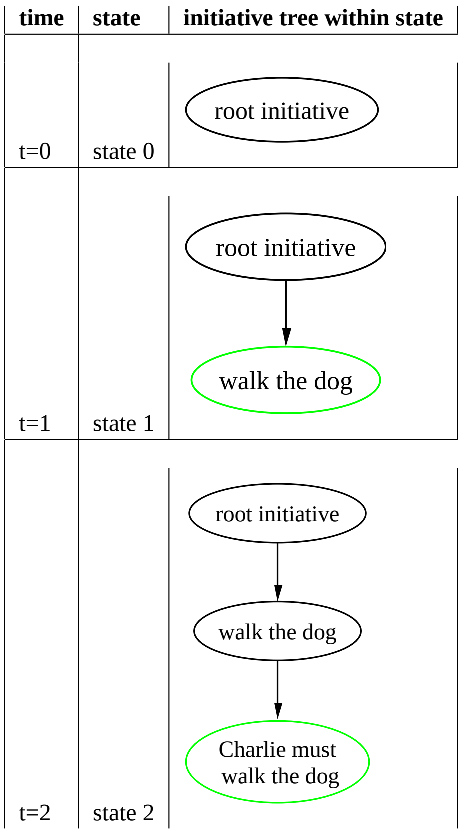


Figure 5: The history of the meeting after time 2

Now Beth proposes an amendment to Adam's motion; that Charlie be the one who walks the dog. Figure 5 shows the meeting history at this time.

Next, debate proceeds, the motions if put to a vote, and the amendment is voted down. A new state is created. At this point, the amendment is no longer pending, and so it is immediately removed from the initiative tree. Figure 6 shows the meeting history after $t = 3$.

Note that after t=3, the initiative tree no longer holds any record of the amendment ever taking place; this is because, when using the Parliament system to manage a Robert's Rules meeting, the initiative tree only models those motions currently in progress[3]. However, the initiative tree is only part of the meeting state, and other components of the current state object maintain a list of all motion which have passed or failed.

To summarize, when a person makes a motion, this corresponds to a node in the initiative tree. Each state holds an entire initiative tree. And the meeting history holds a list of states.

Parallelism

Although we have not yet created a "parallelized" parliamentary ruleset, Parliament is designed with this possibility in mind. Robert's

[3]However, there are points in which the application of Robert's Rules depend on whether a motion has been previously debated or not; since Parliament does not model the semantics of principal motions, however, humans must interpret whether a motion is identical to a previous one anyway.

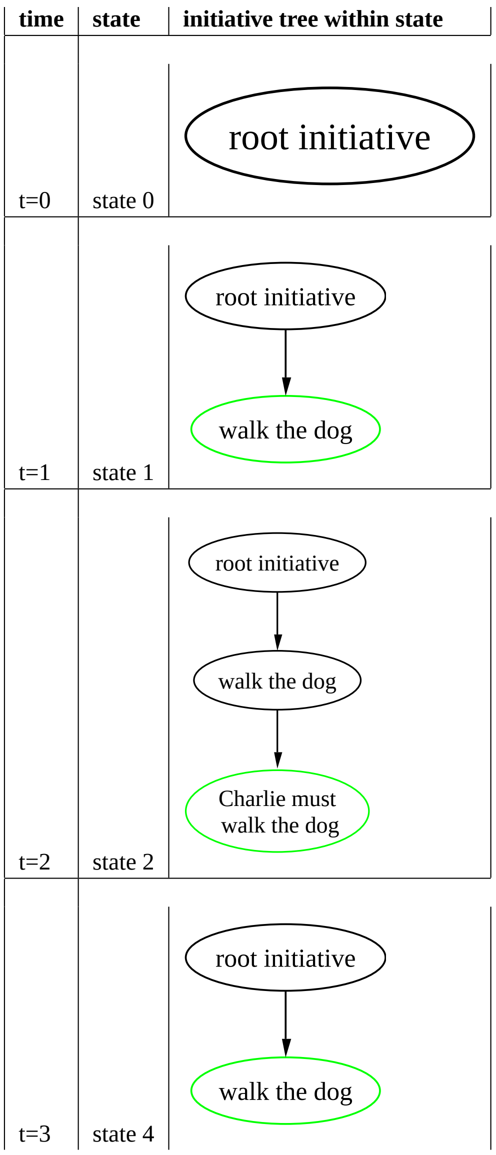
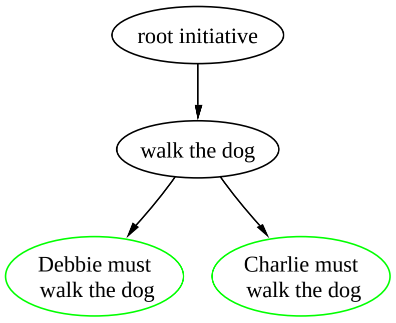
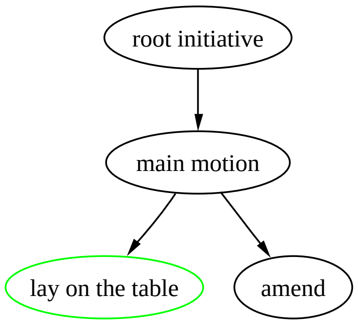


Figure 6: The history of the meeting after time 3

Rules is *serial*; at any given time, there is only one immediately pending motion. Enforcing and dealing with seriality is a major design goal of Robert's Rules, as is necessary for the sanity of a large offline meeting; however, one could imagine that in an online meeting, it would be possible for a group to discuss multiple motions at once. In our example, perhaps the amendment "Charlie walks the dog" is debated simultaneously with another amendment, "Debbie walks the dog". In that case, the initiative tree would look like this (with green denoting the immediately pending motions):



However, while Robert's Rules only allow one immediately pending motion at a time, it is not the case that Robert's Rules do not permit branching trees of initiatives. If Harvey moved a main motion, and then Lisa moved to amend, and then Jacob moved to lay the main motion on the table, then the initiative tree would look like this (with green denoting the immediately pending motion):



Note that the structure of the initiative tree reflects the targets of the various motions (in the above example, both "amend" and "lay on the table" target "main motion"). It does not necessarily reflect their chronological order. In Robert's Rules, when motion A passes or fails, then the motion which was immediately pending when motion A was moved becomes the new immediately pending motion. To implement this, we use a linked list; each initiative object remembers which initiative(s) were immediately pending when it was created. In a serial meeting system such as Robert's Rules, this induces a total ordering on the initiative tree (which is the chronological ordering of the initiatives).

So, in Robert's Rules, there are two different structures that apply to the set of pending initiatives. There is the initiative tree structure, which represents which motions apply to which. There is also the chronological structure, which usually[4] operates as a stack; when

[4]some motions have additional semantics which alter other items in the stack. For instance, a motion to postpone indefinitely is de-

a new motion is made, it is pushed onto the top of the stack, and when a motion is disposed of, it is popped from the stack, and the next motion becomes the immediately pending motion.

In the above example, if Jacob's motion to lay on the table fails[5], then Lisa's amendment becomes the immediately pending motion, because it was the motion that was immediately pending when Jacob moved his motion.

BOTH of these structures can affect whether or not a particular motion is legal ("in order") at a particular time. In Robert's Rules, the effects of these two structures can be different depending on the particular motions at issue. This makes answering the question of which motions are in order rather complex.

The status of actions and initiatives: valid, live, open

Figure 7 summarizes the following terminology.

In the context of a particular meeting state, a parlimentary ruleset may permit some actions and prohibit others. When the parliamentary ruleset permits an action, we say that that action is *valid*. If the action is a motion, then saying that the motion is *in order* or *admissable* is equivalent to calling it valid.

An initiative is said to be *live* whenever it still resides in the initiative tree. Note that initiatives which are no longer in the initiative tree may still be recorded in the meeting history, and hence may still influence current state in some way. If an initiative is a motion, then saying it is *pending* is equivalent to calling it live.

Some subset of the live initiatives are *open*. The semantics of *open* may depend on the ruleset, but in general it implies that an initiative may be directly acted upon in the context of the current state. If an initiative is a motion, then saying it is *immediately pending* is equivalent to calling it open.

Relations between different actions and initiatives: target, applicable, ancestor

Often an action is performed with respect to a *target*. We say that an action *targets* its target. Some actions may not have any target.

When an initiative is made a child of another initiative in the initiative tree, we say that the child initiative has been *applied* to the parent initiative. In the context of a meeting state, the ruleset may specify that some initiatives may or may not be applied to some other initiatives. If it is permitted for initiative C to be made a child of initiative P, we say that C is *applicable to* P.

If, in the initiative tree, the vertex corresponding to initiative A is an ancestor of the vertex corresponding to initiative B, then we say that A is an ancestor of B. Note that since all of the initiatives in the tree are live, therefore an initiative's ancestors must all be live.

3.2 The Parliament module API

signed to end consideration of a target motion. If the motion to postpone indefinitely passes, then not only the motion to postpone indefinitely, but also the target motion (call it M), is removed from the stack. In addition, any other motions which targeted M are now no longer relevant and are also removed.

[5]If Jacob's motion succeeds, then this is like the case in footnote 4; the main motion is laid on the table, and Lisa's motion to amend is laid on the table along with it

The functions of the Parliament module are to keep track of meeting state and to interpret the parliamentary ruleset in the context of the current state.

Meeting state includes data such as which motions are currently pending, the relationships between these motions, which motions are valid at the present time, and which motions have passed or failed earlier in the meeting. The Parliament API provides functions to manipulate and query meeting state.

The program using the module is responsible for determining what is happening in the meeting and calling the API to record meeting events as they occur. For example, if someone makes a motion, the caller calls Parliament's `apply` function to apply the new motion to the state. If a motion passes, the caller calls Parliament's `motionPassed` to record the change in state.

A history of previous meeting states is kept in memory so that Parliament can provide multiple undo/redo functionality.

Here is a list of the methods available in the Parliament API.

Initialization methods

`__init__(compiledRuleFilepath)`

The constructor of the Parliament object. Takes a filepath to a file containing the compiled ruleset. This instance of Parliament will be bound to that ruleset.

`initState(args = None)`

Reset meeting state to the initial state. Takes a variable `args` which is passed to the ruleset's initial state.

Methods to get information about the ruleset which do not depend on the current state

`getInitialState(args = None)`

Returns a new initial state object, without actually changing the current meeting state. Useful if the caller wishes to inspect or operate on an inital meeting state object without resetting the real meeting state.

`nonAbstractActInternalNames()`

Returns a list of the internal names of all actions which exist in the current ruleset except for abstract actions (an "abstract" action is one which is supposed to be hidden from the user; often, these are actions which are used as superclasses but never themselves instantiated).

`getTransitionClass(internalNameOfTransition)`

Get a class object for the specified transition type.

`isAction(object)`

Taken an object. Returns a boolean indicating whether that object is an instance of type Action (or an instance of a subclass of Action).

Methods to get information about the current state

action	initiative	motion	(comments)
valid	valid	in order/admissible	
	live	pending	pending. . . but not necessarily immediately pending
	open	immediately pending	
target	target	target	some actions don't have any target
	applicable to	applicable to	A is eligible to be B's parent in the initiative tree
	ancestor	ancestor	an ancestor is always live

Figure 7: Some terminology on initiatives. Each row represents one term. The first three columns indicate how the term may be expressed when the initiative in question is of the subclass given by the column label. The last column gives comments on the term.

`getCurrentState()`

Returns the state object representing the current meeting state.

`validActs(state)`

Returns a list of all actions which are valid from meeting state `state`.

`currentValidActs()`

Returns a list of all actions which are valid in the current meeting state.

`getOpenInitiatives()`

Returns the current list of open initiatives. In Robert's Rules, this is a list of length one containing the immediately pending motion.

`getTransition(internalNameOfTransition)`

Returns an instantiation of the specified transition type. The transition is instantiated as if it began in the current state (remember that state transition constructors take the current state as an argument; so, what is returned by this method may depend on what the current state is).

`getTransitionName(internalNameOfTransition)`

Returns the name attribute of the specified transition. The transition is temporarily instantiated as if it began in the current state.

`getInitiativeTreeRoot()`

Returns the root vertex of the initiative tree of the current state.

`getInitiativeTree()`

Returns the initiative tree of the current state.

`getValidTargets(initiative)`

Takes either an initiative object or the internal name of an initiative type.

Returns a list of other initiatives which would be valid targets of the specified initiative if the specified initiative were initiated in the current state. The first element of this list is the default target. If there are no valid targets, the empty list is returned.

`getDefaultTarget(initiative)`

Takes either an initiative object or the internal name of an initiative type.

Returns the default target of the specified initiative if the specified initiative were initiated in the current state.

`getMeetingEventHistory()`

Returns the history of initiatives which have been initiated in this meeting. Returns a list of initiatives, ordered by the times at which the initiatives were made.

Methods to modify the current state

`appendNewState(newState, newInitiativeAdded)`

Append `newState` to the state history, and make it the current state. `newInitiativeAdded` is boolean and should be `True` if there was a new initiative added in this state transition.

```
apply(initiativeInternalName,
    position, args = None)
```

Instantiate an initiative of the specified type, and apply it to the current state. `position` is the vertex of the initiative tree which will be the parent of the new initiative. `args` will be passed to the initiative's constructor.

`reparent(initiative, newParentInitiative)`

Relocates the position of `initiative` within the initiative tree so that its parent initiative becomes `newParentInitiative`.

`motionFailed()`

Fail the immediately pending motion, and add the appropriate new state and state transition to the state history.

`motionPassed()`

Pass the immediately pending motion, and add the appropriate new state and state transition to the state history.

Methods involving undo/redo

`canGoBack()`

Returns True if it's possible to go farther back in the undo history.

`goBack()`

Undo the previous state transition.

`canGoFwd()`

Returns True if it's possible to go farther forwards in the undo history.

`goFwd()`

Go forwards in the undo history (redo).

3.3 The ruleset specification language

The rule specification language has a quasi-English syntax in the same manner as SQL. A ruleset is typically written in a separate textfile and then loaded into the Parliament module upon initialization. The design goals for the rule specification language are:

1. The language should be as readable as possible by parliamentarians.
2. Someone with only a little bit of programming experience should be able to understand the language well enough to write or to edit a rule specification.
3. Robert's Rules should be able to be specified concisely in the language.
4. Recognizing that there will always be things that people want to do with the ruleset that the language does not support, the language should allow complex programming code to be embedded in a specification.

The rule specification language is object-oriented. Different types of actions, initiatives, and motions can be defined, and each type of object has a collection of attribute-value pairs. Each type of object derives from a "superclass", from which it inherits default attribute values. This allows a compact description of motions; for example, the Robert's Rules `Motion to make a general order` is a special case of the superclass `Principal Motion`, allowing it to inherit the attribute value `True` for the attribute `debatable`.

The determination of attribute values can also be computed dynamically at runtime by evaluating expressions. For instance, the value of the `debatable` attribute of the `reconsider` motion is given by the expression `ONLY WHEN PARENT IS debatable`.

The Parliament module uses the ruleset by compiling the object types specified in the ruleset file into Python code. A special compiler was developed for this purpose. Then, the resulting Python code is used as a library by the Parliament module. The object definitions in the ruleset files are compiled into actual Python objects of types `transition`, `action`, `initiative`, `motion`, and `state`.

The ruleset specification can be arbitrarily expressive; if there is no other way to express some desired behavior, arbitrary Python code can be embedded into any object.

Examples of the rule specification language

Here's how the Robert's Rules motion "Lay on the table" is defined (the actual definition has a longer summary and is not word-wrapped):

```
-----------------
NAME: Lay on the table
MOTION TO FORM OF NAME: "Motion to lay
on the table"

TYPE: Subsidiary motion

SUMMARY: "The objective of this motion is to
temporarily lay a question aside"

motion precedence: 1
debatable: NO
amendable: NO
subsidiaries allowed: NO
reconsiderable: ONLY WHEN (WAS_ACCEPTED)

TARGET: ancestor motion
ON PASS: table target

category: "scheduling"
purpose: "delay"

# comments can be embedded like this

RRO section ref: "19"
RROR section ref: "28"

{
def example_method(self):
    print 'This is embedded Python code'

}

-----------------
```

The compiled Python class definition may be found in Appendix A. Remember that this entire block will be turned into a single Python class.

Let's step through the specification of this motion and explain it.

```
-----------------
```

The dashes are a signal to the compiler that this is the beginning of a new object type (which corresponds to a Python class).

```
NAME: Lay on the table
```

The NAME line is the first example of an "attribute" of this object type. An "attribute" has an attribute name (in this case, NAME), and a value. The value may or may not change depending on the context of the current meeting state. Attributes will be usually be compiled into instance methods of the Python object, but will sometimes be represented as an actual Python attribute of the class (when the attribute is known to be a constant).

Generally the case of the attributes does not matter; in compilation, all attribute names are converted to lowercase. As a convention, we like to use uppercase attribute names to mark attributes which are *standard*, and lowercase attribute names for attributes which are arbitrary and specific to this ruleset.

Some attribute names are *standard*. Standard attributes have a defined semantics that is the same for any ruleset. Some standard attributes are also treated in a special way by the compiler. But any alphanumeric name with spaces can be used as an attribute. Non-standard attributes can be used any way that the ruleset author wishes. However, the user interface should not use any non-standard attributes if it wishes to remain compatible with all rulesets. See Section B.1 in Appendix B for a list of standard attributes.

NAME is a standard attribute, and gets special treatment by the compiler. First, although its value is a string, no quotes are needed around its value (in general, quotes are required around string-typed attribute values). Second, NAME has a special meaning. NAME is required for every object, and Parliament UIs expect it to be a string suitable for display to the user which identifies this object type.

NAME also has a third function. If no value for attribute "INTERNAL NAME" is specified, then the value of NAME is copied to INTERNAL NAME. INTERNAL NAME is used to generate the name of the Python class, and other parts of the ruleset specification can refer to this function by its INTERNAL NAME. INTERNAL NAME is not expected to be suitable for display to the user.

Usually the ruleset will only explicitly specify NAME, and will let INTERNAL NAME be automatic. But if the ruleset author wants NAME and INTERNAL NAME to be different, they are free to specify an INTERNAL NAME, too. This is desirable when you want to implement a single action type using two different internal classes, only one of which is valid at any one time. In this way the user will be shown only the NAME string, and will see the two different classes as one type of action. In our Robert's Rules implementation, we used this trick to deal with motions which can be different types depending on meeting context. For example, the Motion to Recess can be either Principal or Privileged depending on context. Internally, we created two motions, a Principal one and a Privileged one, with two distinct INTERNAL NAMEs; but the NAME of both motions was set to "Recess".

```
MOTION TO FORM OF NAME: "Motion to lay
on the table"

TYPE: Subsidiary motion
```

In contrast to NAME and TYPE, MOTION TO FORM OF NAME is just a run-of-the-mill attribute, and does not have any standard predefined semantics. Therefore, since in this case its value is a string, the string must be enclosed in quotation marks[6].

"TYPE" indicates which class is the superclass of this class. In this case, it is class "Subsidiary motion". Since "Subsidiary motion" is a subclass of "motion", "Lay on the table" is also a subclass of "motion".

```
SUMMARY: "The objective of this motion is to
temporarily lay a question aside"

motion precedence: 1
debatable: NO
amendable: NO
subsidiaries allowed: NO
```

"SUMMARY" is a standard attribute that every ruleset should support. It contains a medium-length textual summary that may be displayed to the user to explain the action.

"motion precedence" shows an example of an attribute with a numerical value.

"debatable", "amendable", and "subsidiaries allowed" show examples of attributes with a boolean value (NO is compiled into Python's `False`, YES becomes Python's `True`).

```
reconsiderable: ONLY WHEN (WAS_ACCEPTED)
```

Here is our first example of a non-static attribute value. The value of the reconsiderable attribute of motion type "Lay on the table" will be the result of evaluating this expression at runtime. There are a number of keywords that may be used to build expressions, such as AND, OR, NOT, IF, and EXCEPT. ONLY WHEN is a synonym for IF, and either of these simply return the value of their argument (their only use is for easier reading). So, in this case, the value returned for the attribute `reconsiderable` will be the return value of the method `was_accepted` of the current `state` object. See Table B.5 in Appendix B for details about the syntax of expressions.

```
TARGET: ancestor motion
```

This line sets a condition on the type of objects which this motion is allowed to target[7]. In this case, the motion "Lay on the table" can target another motion M if M is an ancestor motion (that is, M is an ancestor of "Lay on the table" in the initiative tree). See Table B.2 in Appendix B for other available values for a TARGET attribute.

[6] We have wordwrapped this line for presentation purposes; parsing is mostly line-by-line and hence it wouldn't really work this way; the whole string is on one line in the actual file

[7] Actually, in our current implementation, the "TARGET: ancestor" condition has no effect; in the case of "Lay on the table", an equivalent condition is already enforced in the superclass "Subsidiary motion".

```
ON PASS: table target
```

This line tells Parliament what to do when a motion of type “Lay on the table” passes. In this case, `table target` is a command to lay the target of this motion on the table. See Table B.3 in Appendix B for other available actions.

```
category: "scheduling"
purpose: "delay"
```

“category” and “purpose” are not standard attributes. In this case, the ruleset is using them to categorize the motions in two different ways. The UI could choose to use these attributes be used to inform the user or to help sort available motions.

```
# comments can be embedded like this
```

A # sign at the beginning of a line makes that line a comment.

```
RRO section ref: "19"
RROR section ref: "28"
```

These are two more non-standard attributes. In this case, the ruleset is using them to note where to find more information about this motion in two different official Robert’s Rules editions.

```
{
def example_method(self):
    print ’This is embedded Python code’

}
```

Whatever is between curly braces is inserted directly into the compiled Python class.

```
-----------------
```

The dashes at the end delimit the end of the block for the “Lay on the table” object type.

4. THE ROBERT’S RULES PARLIAMENTARY RULESET SPECIFICATION

We have written a partial ruleset specification for the public domain version of Robert’s Rules. The ruleset includes over 25 of the most common motions, the important attributes of those motions (such as when they are debatable, what vote is required for them to carry), most of the precedence relations between them, and some of their semantics.

This specification was initially based on Henry Prakken’s formalization of the Rules [3], which he kindly provided to us in machine-readable form. We made many changes to Prakken’s formalization, including the addition of the complicated logic of precedence.

We discovered that Robert’s Rules are more complex than we had expected, even though one of us [DD] has had extensive experience in actual meetings using Robert’s Rules. It is not just that the Rules are described in a verbose fashion in the Robert’s Rules books; rather, the logic of the rules is fundamentally complex. Much of this complexity is from context-dependent special cases. This complexity is an additional argument in favor of the need for a formal specification of the Rules such as we are providing.

One unexpected complexity which surprised the author who has not had much experience in Robert’s Rules meetings [BS] is the relations between the concepts of *the tree of pending motions*, *precedence* and the *target* of a motion. Initially, this author thought that

a) A new motion can only be applied to the immediately pending motion (that is, the initiative tree never branches).

b) Whether a motion is in order can be computed by only looking at the new motion and the immediately pending motion.

Neither of these are true in Robert’s Rules. Section 3.1 gives an counterexample to (a). A counterexample of (b) is seen in Figure 8.

Another confusing point is the distinction between the *target* of a motion, and the *motion to which that motion is applied* (the parent in the initiative tree). Usually, these two notions cohere in Robert’s Rules. However, there are a few special cases (such as the motion to reconsider) in which it is possible to have a target which is not even a pending motion.

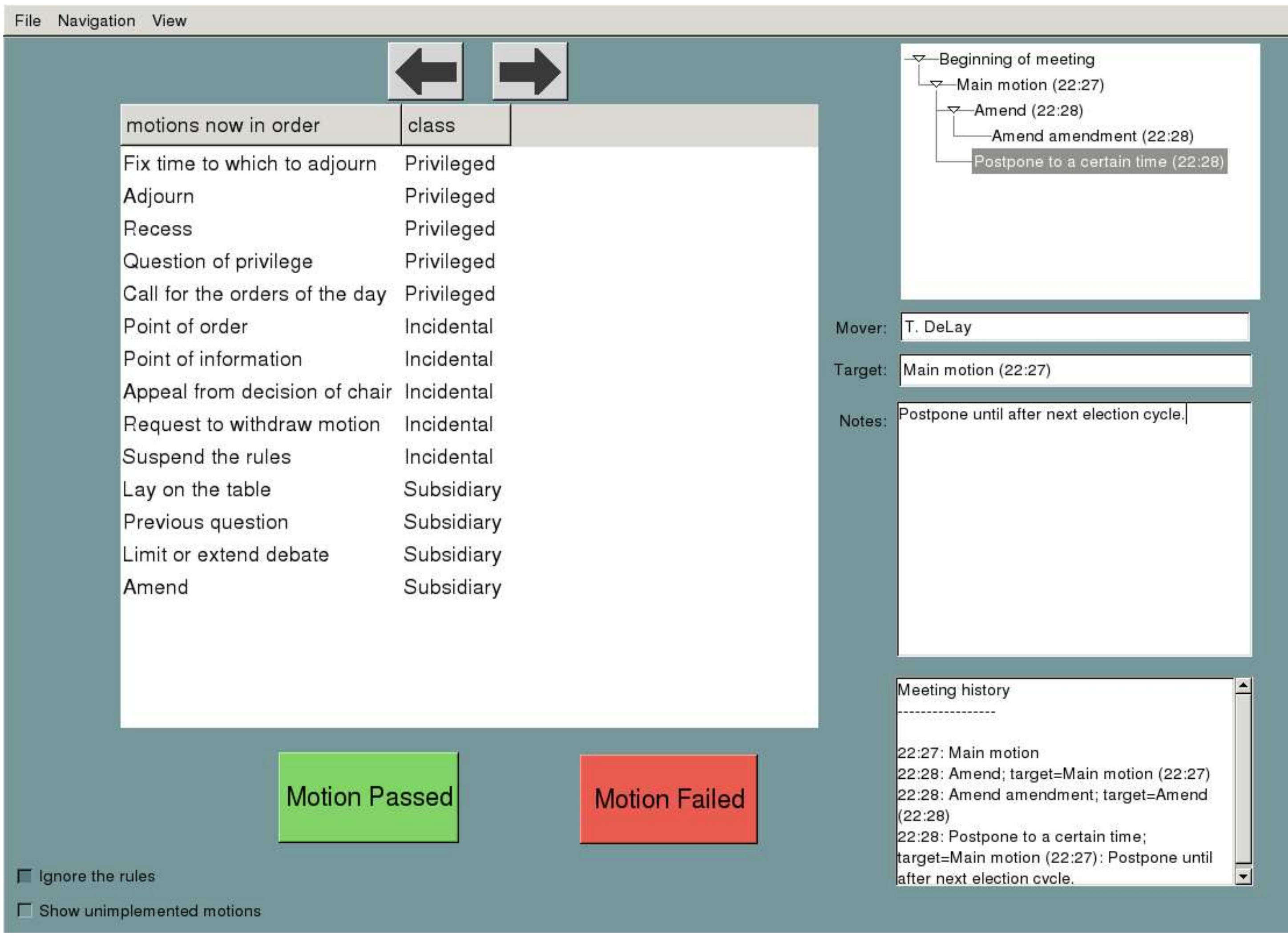
Future researchers would do well to avoid these pitfalls.

5. AN APPLICATION: MEETING ASSISTANT

The prototype *meeting assistant* application uses the Parliament module, and is designed to operate in a face-to-face meeting conducted according to Robert’s Rules of Order. The user provides input about events in the meeting, such as motions and votes. The meeting assistant tracks the meeting state and displays useful information such as currently pending motions, motions currently in order, and transacted business. More discussion on the motivations and usage of such an application can be found in [**?**].

The Primary User Interface

Figure 9 depicts the meeting assistant’s graphical user interface. Its features are designed especially for use by an organization’s secretary, who is responsible for producing the minutes. The informa-



motions now in order	class
Fix time to which to adjourn	Privileged
Adjourn	Privileged
Recess	Privileged
Question of privilege	Privileged
Call for the orders of the day	Privileged
Point of order	Incidental
Point of information	Incidental
Appeal from decision of chair	Incidental
Request to withdraw motion	Incidental
Suspend the rules	Incidental
Lay on the table	Subsidiary
Previous question	Subsidiary
Limit or extend debate	Subsidiary
Amend	Subsidiary

```
Beginning of meeting
  Main motion (22:27)
    Amend (22:28)
      Amend amendment (22:28)
    Postpone to a certain time (22:28)
```

Mover:	T. DeLay
Target:	Main motion (22:27)
Notes:	Postpone until after next election cycle.

```
Meeting history
-----------------

22:27: Main motion
22:28: Amend; target=Main motion (22:27)
22:28: Amend amendment; target=Amend
(22:28)
22:28: Postpone to a certain time;
target=Main motion (22:27): Postpone until
after next election cycle.
```

Figure 9: The Robert's Rules meeting assistant. The list of *motions currently in order* is on the left. The *back* and *forward* buttons are above, and the *pass* and *fail* buttons below along with the *ignore rules* checkbox. The tree of *currently pending motions* is in the upper-right corner; below it are the motion-detail fields: *mover*, *target*, and *text*. The event log is in the bottom-right corner.

tion captured by the meeting assistant is very close to exactly that required for this task.

The interface provides a list of *motions currently in order*. Which motions appear in the list depends on the meeting state—primarily, which motions are currently pending. The user may activate any of motions in the list to indicate it has been moved in the meeting. *Pass* and *fail* buttons allow the user to indicate the fate of the immediately pending motion. *Back* and *forward* buttons navigate through meeting history, providing a multiple undo/redo mechanism critical for usability.

A tree diagram displays the *currently pending motions* and how they are related. The immediately pending motion is always at the bottom of the diagram. Displayed and editable in several fields are the details of a motion, including who its *mover* is, what its *target* is, and its *text* or other related notes.

The interface also provides an event log with a record of each motion in the order it was moved, and whether it passed or failed.

Real-world assemblies sometimes deviate from the rules, either by means of a motion to suspend the rules or by sheer mistake. In either case, to be useful, the software must continue to track the state of the meeting. Hence the interface provides an *ignore rules* checkbox that allows the user to record actions and motions despite these being out of order according to the module's interpretation of the rules.

The "view only" window

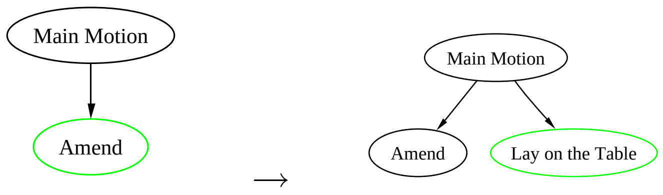
Figure **??** shows a window which displays the most relevant information about the current meeting state in an uncluttered easy-to-read format.

6. FUTURE WORK

There are many possibilities for using and for improving the Parliament module.

6.1 Improve Parliament

If a Main Motion is made, followed by a motion to Amend, then Lay on the Table is in order, because Amend yields to Lay on the Table, and Main Motion yields to Lay on the Table, and Lay on the Table may be applied to Main Motion:



But if a Main Motion is made, followed by a motion to Recess, followed by a motion to Amend, then Lay on the Table is **not** in order, because although Amend yields to Lay on the Table, Recess does not, and so there are no eligible targets for Lay on the Table:

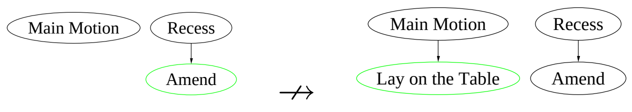


Figure 8: Knowledge of the immediately pending motion is not sufficient information to decide which motions are in order

- Ideally, a ruleset specification should not have to contain much embedded Python code. By making the ruleset language more expressive, we can eliminate most of the embedded code which is currently in the Robert's Rules specification.
- Currently, there are only a few motions whose semantics are modeled by Parliament (that is, motions which *do something special* when they pass (see Table B.3 in Appendix B). More motion semantics should be implemented until most of Robert's Rules' motions are covered.
- Parliament should track scheduling and agenda management.
- Parliament should track floor control, speaker's lists, and limits on debate.
- Parliament should track voting, membership and attendance.
- Parliament works, but it is not at a mature stage of development. The source code must be cleaned up and commented, and a battery of unit tests must be written.

6.2 Improve the Robert's Rules specification

While we have covered enough of the basics of Robert's Rules to be useful—indeed, Parliament already obeys the Rules to an extent greater than most assemblies that might use it—the Rules contain many special exceptions which we have not yet modeled.

6.3 Improve the meeting assistant

The meeting assistant should:

- Be documented, and made easy to install.
- Generate various reports after the meeting, including standard meeting minutes, an event log, and a list of main motions which carried during the meeting.
- Allow the user to see various addition attributes of the motions, such as whether a motion is debatable or reconsiderable.
- Have an interface to an English copy of the official Rules, and should allow users to jump to the location where a given motion is discussed.

6.4 Make more ruleset specifications

We plan to utilize the modular nature of Parliament to experiment with many different types of rulesets, including:

- The *Standard Code of Parliamentary Procedure*, a simplified competitor to Robert's Rules.
- A parallelized version of the rules, suitable for asynchronous online assemblies.
- Simplified rulesets for beginners.

6.5 Make more applications

The meeting assistant is only one of many possible parliamentary applications which Parliament could be used in. Other intriguing applications include:

- A web-based application which automatically chairs an online, asynchronous meeting.
- An IRC-based automatic meeting chair for synchronous parliamentary meetings on IRC.
- A wiki-based automatic meeting chair to help online communities asynchronously make important decisions.

7. RELATED WORK

Other researchers have investigated implementing Robert's Rules of Order in software, and adapting the rules for use in the context of computer-mediated communication. Zhang *et al.* designed a client-server architecture for collaboration mitigated by what they call "extended RRO" [5]. Horan and Benington describe a protocol for conducting electronic deliberations by e-mail in academic committees that use Robert's Rules [2]. They assume that users will implement the protocol, but software could automate some of what they recommend. Davies *et al.* have built an online deliberation environment, *Deme* [1], primarily to supplement the activities of groups that already meet face-to-face.

8. CONCLUSION

The Parliament module solves a major portion of the task of developing parliamentary procedure software. A reusable module was created which implements and interprets parliamentary rules, which tracks meeting state, and which infers important information such as which motions are in order at a given time. The module is flexible and could be used with almost any set of parliamentary rules.

A concise specification language was created to allow others to efficiently create and modify rulesets. A partial, yet usable specification of Robert's Rules was created with over 25 motions. The module was combined with the Robert's Rules specification and used to build a complete Robert's Rules meeting assistant. The meeting assistant was used in a real face-to-face meeting[**?**]. The Parliament module shows great potential for use in many contexts, including both face-to-face and online meetings. The module will hopefully lead to the creation of a variety of useful parliamentary software.

9. REFERENCES

[1] T. Davies, B. O'Connor, A. A. Cochran, and J. J. Effrat. An online environment for democratic deliberation: Motivations, principles, and design. Working paper, March 2004.

[2] S. M. Horan and J. H. Benington. A protocol for using electronic messaging to facilitate academic committee deliberations. *Journal of Higher Education Policy and Management*, 22(2):187–197, November 2000.

[3] H. Prakken. Formalizing robert's rules of order: An experiment in automating mediation of group decision making. Technical Report REP-FIT-1998-12, GMD, April 1998.

[4] H. M. Robert. *Robert's Rules of Order Revised*. Public Domain, 1915.

[5] J. Zhang, C. K. Chang, K. H. Chang, and F. K. H. Quek. Rule-mitigated collaboration framework. In *Proceedings of the Eighth IEEE International Symposium on Computers and Communication (ISCC 2003)*, volume 1, pages 614–619, Washington, DC, USA, June 2003. IEEE Computer Society.

APPENDIX

A. COMPILED CLASS DEFINITION OF MOTION “LAY ON THE TABLE”

```
class lay_on_the_table(subsidiary_motion):
    name = ’Lay on the table’

    internal_name = ’lay_on_the_table’

    def motion_to_form_of_name(self, state):
        try:
            return r’Motion to lay on the table’

        except AttributeError:
            return None

    def type(self, state):
        try:
            return r’subsidiary_motion’

        except AttributeError:
            return None

    def summary(self, state):
        try:
            return r’The objective of this motion is to temporarily lay a question aside.’


        except AttributeError:
            return None

    def motion_precedence(self, state):
        try:
            return 1

        except AttributeError:
            return None
    def debatable(self, state):
        try:
            return False

        except AttributeError:
            return None
    def amendable(self, state):
        try:
            return False

        except AttributeError:
            return None
    def subsidiaries_allowed(self, state, transition):
        try:
            return False

        except AttributeError:
            return None
    def reconsiderable(self, state, transition):
        try:
            return (state.was_accepted(self))
        except AttributeError:
            return None
    def getTargetType(self):
        return ’ancestor motion’

    def getPotentialTargets(self):
        result = []
        cur = self.prevMotion()
        while cur:
            result = result.append(cur)
            cur = cur.prevMotion()

        return result


    def motionPassed(self, state):
        newState = subsidiary_motion.motionPassed(self,state)

        if self.target:
             (newState, subtree) = self.target.assignResultAndDetach(
                   newState, ’table’, transition = None,
                   descendentLabel = ’table’)
             newState.subtreeTables[’table’].append(subtree)



        return newState

    def category(self, state):
        try:
            return r’scheduling’

        except AttributeError:
            return None
    def purpose(self, state):
        try:
            return r’delay’

        except AttributeError:
            return None

    # comments can be embedded like this

    def rro_section_ref(self, state):
        try:
            return r’19’

        except AttributeError:
            return None
    def rror_section_ref(self, state):
        try:
            return r’28’

        except AttributeError:
            return None

    def example_method(self):
        print ’This is embedded Python code’
```

B. RULE SPECIFICATION LANGUAGE REFERENCE

B.1 Table of standard attributes

The NAME and TYPE attributes are required; all others are optional. All attribute names must be composed only of alphanumeric ASCII characters and spaces. Although some of the examples below run over multiple lines in order to fit them into the table, in reality, each attribute definition must be on a single line. The phrases “object type” and “object class” mean the same thing, although TYPE is the attribute name used to specify the superclass of an object.

Attribute name	**Meaning**	**Special syntax**	**Example**
NAME	printable name of object type	no quotes needed	`NAME: Lay on the table`
INTERNAL NAME	the name used to refer to object type elsewhere in the rule specification	no quotes needed; if not specified, NAME is used	`INTERNAL NAME: Lay on the table`
TYPE	superclass of this object type	no quotes needed; if followed by `(ABSTRACT)` then this object type is marked as ABSTRACT and will be invisible to the user	`TYPE: motion (ABSTRACT)`
SUMMARY	a mid-length printable description of the object type		`SUMMARY: "The objective of this motion is to temporarily lay a question aside"`
TARGET	currently unused (will be used to specify which actions are allowable targets of the object type)	see Table B.2	`TARGET: ancestor motion`
ON PASS	action to be taken if this motion passes	see Table B.3	`ON PASS: table target`
ON FAIL	action to be taken if this motion fails	see Table B.3	`ON FAIL: table target`
MOTION PRECEDENCE	a number used to compute motion precedence; lower numbers take precedence		`MOTION PRECEDENCE: 3`
DECISION MODE	What determines whether this motion passes or fails?	see Table B.4	`DECISION MODE: "vote"`
REQUIRES TARGET	Does this motion require potential targets to be available in order to be valid? Value should be “yes” or “no”.		`REQUIRES TARGET: YES`
APPLIES ONLY TO TYPE	If this attribute is given, then this initiative is only applicable to initiatives of the specified type		`APPLIES ONLY TO TYPE: "Principal motion"`
REQUIRED MAJORITY	What proportion of the assembly must agree in order pass this motion		`REQUIRED MAJORITY: $\frac{2}{3}$`

B.2 Table of standard values of TARGET attribute

The semantics of the TARGET attribute have not been completely implemented.

Value	**Meaning**
`ancestor motion`	Only ancestor motions are eligible targets
`pending motion`	Only pending motions are eligible targets

B.3 Table of standard values of ON PASS and ON FAIL attributes

Value	**Meaning**
`withdraw target`	Withdraw the target motion
`table target`	Table the target motion

B.4 Table of standard values of DECISION MODE attribute

Value	**Meaning**
`YES`	The motion automatically passes as soon as it is made
`"vote"`	The motion goes to a vote

B.5 Table of syntax used in attribute value expressions

Syntactical element or **Keyword**	**Meaning**	**Example**

“(. . .)”	Used for grouping	`subsidiaries allowed:`
		`  ONLY WHEN (this motion`
		`    is debatable)`
		`  AND (motion is "Lay on`
		`    the table")`

“<string>”	Depending on context, either evaluates to <string> or evaluates to whatever the method `state.askUser(<string>)` returns	`inventor: "robert"`
YES	The truth value “True”	`debatable: YES`
NO	The truth value “False”	`debatable: NO`

EXCEPT class name	Only for use in attributes which generate an expression which taken the extra argument “transition”. (the way to make attributes do this has not been finalized yet)
	Compares the name of “transition” to “class name”, and returns TRUE unless they are the same. The complement of ONLY.
		`subsidiaries allowed:`
		` EXCEPT "postpone`
		`  indefinitely"`

ONLY WHEN expression	returns the value of “expression”	`reconsiderable: ONLY WHEN`
		`  "rejected the first time"`

IF expression	synonym for ONLY WHEN	

ONLY class name	Only for use in attributes which generate an expression which taken the extra argument “transition”. (the way to make attributes do this has not been finalized yet)
	Compares the name of “transition” to “class name”, and returns TRUE if and only if they are the same. The complement of EXCEPT.
		`subsidiaries allowed:`
		` ONLY ’previous question’`

Syntactical element or **Keyword**	**Meaning**	**Example**
THIS INITIATIVE IS attribute name	returns the value of attribute “attribute name” of this initiative	`subsidiaries allowed: ONLY WHEN (this initiative is debatable)`
THIS MOTION IS attribute name	synonym for “THIS INITIATIVE IS”	
THIS IS attribute name	synonym for “THIS INITIATIVE IS”	
THIS attribute name	synonym for “THIS INITIATIVE IS”	
PARENT INITIATIVE IS attribute name	returns the value of attribute	
PARENT MOTION IS attribute name	synonym for “PARENT INITIATIVE IS”	
PARENT IS attribute name	synonym for “PARENT INITIATIVE IS” “attribute name” of this initiative	`debatable: ONLY WHEN PARENT IS debatable`
PARENT attribute name	synonym for “PARENT INITIATIVE IS”	
condition1 AND condition2	returns boolean AND of condition1 and condition2	`subsidiaries allowed: ONLY WHEN (this motion is debatable) AND (motion is "Lay on the table") OR (motion is ’Previous question’)`
condition1 OR condition2	analogous to AND	
NOT condition	analogous to AND	
anymethod	Evaluated to the return value of `state.anymethod`.	`reconsiderable: ONLY WHEN (WAS_ACCEPTED)`