The GOAL Grammar

Koen V. Hindriks, Vincent J. Koeman

August 6, 2015

1 Introduction

This document contains the grammar specification for the GOAL agent programming language.

A GOAL agent is defined by three types of *modules*: A module used to initialize the agent (a so-called *init module*), a module for processing events (a so-called *event module*), and a module for making decisions (a so-called *main module*). Each of these modules in turn can reference sub-modules. A module can also make references to *knowledge representation (KR) files*. Like modules, KR files can also have different functions or roles. A KR file can be used, for example, to initialise the agent's beliefs or its goals. A module may use user-specified actions and may depend on one or more action specification files that contains the specifications of these actions. Dependencies between modules, of modules on KR files, or on action specification files must be made explicit in a module file. A module, moreover, only should reference dependencies on other components when there is an explicit dependency on that component (i.e. that component is actually *used* somehow in the module).

Which module should be used for which role (initialization, event processing, or decision making) is specified in a multi-agent system (MAS) file. A MAS file not only specifies the role of an agent component but also specifies when an agent will be created or launched. Finally, a MAS file also specifies which environment, if any, will be launched when the MAS is created.

2 Multi-Agent System Files

A MAS file is a recipe for launching a MAS. It provides information about the environment to connect agents to, it defines the agents of the MAS, and it specifies when to launch agents.

mas	:=	[*environment*] *agent*$^+$ *policy*
environment	:=	`use` (*ref* | *path*) `as environment` [`with` *id* `=` *value* (`,` *id* `=` *value*)*] `.`
agent	:=	`define` *id* `as agent` **{** *useclause*$^+$ **}**
useclause	:=	`use` *ref* `as` *usecase* `.`
usecase	:=	`init` [`module`] | `event` [`module`] | `main` [`module`]
policy	:=	`launchpolicy` **{** *launchrule*$^+$ **}**
launchrule	:=	[`when` *entity*] `launch` *instruction* (`,` *instruction*)* `.`
instruction	:=	*id* [`with` *constraint* (`,` *constraint*)*]
entity	:=	`*` | `type` `=` *id* | `name` `=` *id*
constraint	:=	`name` `=` *id* | `name` `=` `*` | `number` `=` *num* | `max` `=` *num*
id	:=	*alphanumeral with underscores that starts with letter or underscore*
num	:=	*natural number, starting from* 0
value	:=	*double quoted string, numeral, or list of values between square brackets*
ref	:=	*id* (`.`*id*)*
path	:=	*double quoted string containing a file path*

Table 1: Multi-Agent System Grammar

2.1 Environment

A MAS may include a reference to an environment but does not have to. If an environment is referenced, it is also possible to specify parameters for initializing this environment.

An environment reference in a MAS file should satisfy the following **constraints**:

1. ...

2.2 Agent Definitions

A MAS file must define one or more agents and their names. An agent is defined by one or more use clauses. Each use clause specifies a reference to a module file as well as the role associated with the referenced file (its use case). Three different use cases for modules are distinguished: a use case for *initializing* an agent (using the `init` keyword), a use case for *processing events* (using the `event` keyword), and a use case for decision making (using the `main` keyword).

The agent definitions in a MAS file should satisfy the following **constraints**:

1. *A name of an agent definition should only be used once.* If not, it is impossible to uniquely resolve a reference to an agent definition in a launch rule.

2. ...

2.3 Launch Policy

A MAS file must specify a launch policy. A launch policy specifies one or more rules for launching agents. These rules can be either conditional or unconditional. A launch rule is a *conditional* rule if it has a `when` clause; in that case, the rule will only apply if an entity in a referenced environment becomes available. An unconditional rule is only applied once, in order, immediately when a MAS is launched. Unconditional rules that might also have been specified are disregarded at that time. For this reason, it is good practice to specify unconditional rules before any unconditional rules.

A conditional rule is applied if an entity becomes available in an environment that matches its `when` clause (and the specified maximum number of applications, if any, has not been reached yet). An entity always matches a `*` clause, it matches a `type` clause if the type of the entity matches the specified type, and it matches a `name` clause if the name of the entity matches the specified name. A conditional rule is said to be less specific than another conditional rule if its `when` clause is less specific. There are three types of `when` clauses: `*`, `type`, and `name` clauses. A `name` clause is more specific than a `type` clause, and a `type` clause is more specific than a `*` clause.

Each rule must specify a *launch instruction* that indicates which agent should be launched. If the rule is conditional an entity in an environment must have triggered application of the rule, and the agent that is launched will be connected to that entity. A launch instruction can also specify additional constraints on the number of agents that are launched and the name of these agents. There are three types of constraints:

- A `name` constraint: If of the form `name` = *id*, *id* will be used as a base name to name the agents that are launched (successive agents will be named *id*, *id*`1`, *id*`2`, etc.). If of the form `name` = `*`, the name of the entity that triggered the rule is used as a base name. If no `name` constraint is specified, by default, the name of the agent definition is used as a base name.

- A `number` constraint: if a constraint `number` = n has been specified, each time the rule is applied n agents will be launched; if the rule is conditional, moreover, each of these agents will be connected to one and the same entity.

- A `max` constraint: If a constraint `max` = n has been specified, the rule with be applied at most n times and at most n agents will be launched by applying the rule (repeatedly while the MAS is executed). A `max` constraint has precedence over `number` constraints. That is,

even if a constraint `number` $= n$ has been specified, if a constraint `max` $= m$ also has been specified and $n > m$, at most m agents will be launched.

A launch policy in a MAS file should satisfy the following **constraints**:

1. *It should only contain conditional launch rules if an environment is referenced.* It does not make sense to add conditional launch rules without referencing an environment. A conditional launch rule is only applied if a controllable entity is made available by an environment.

2. *It should be complete.* That is, if an environment is available, at least one launch rule should be applicable for any entity that becomes available. Otherwise it is impossible to connect an agent to the entity to control it.

3. *A conditional launch rule should not follow a less specific conditional launch rule that does not specify a maximum constraint.* It does not make sense to specify a more specific conditional rule after another rule that is less specific and does not specify a maximum number of rule applications. The more specific rule will never be applied in that case.

4. *A constraint `name` = `*` should only be used in conditional rules.* It is impossible to use the name of an entity for naming the agents that are launched if the rule is not triggered by an entity that has become available in an environment.

5. *The specified number of agents (using the `number` option) that should be launched should not exceed the maximum number, if any has been specified (using the `max` option).* It does not make sense to specify a number that exceeds the maximum number of agents that will be launched. A `max` constraint takes precedence over a `number` constraint. By default, no maximum is specified.

6. ...

If violating a constraint makes no sense that is less of a problem than if violating a constraint results in a failure to meet all the constraints specified in the launch policy. In the former case, it is sufficient to issue a warning, but in the latter case an error should be reported and execution of the MAS should be prevented. Whether a launch policy is complete or not can only be established at run time; if not, a runtime error should be generated.

3 Modules

A module must contain one or more rules that enable an agent to select an action. GOAL provides three types of rules (`if-then`, `forall-do`, and `listall-do`). The dependencies on predicates used in the conditions of rules must be made explicit by use clauses that define these predicates either as knowledge, beliefs, or goals. The dependencies on actions and modules, which can also be selected for execution by rules, must also be made explicit by use clauses. Three types of module options can be specified: an option that specifies when a module should be terminated (`exit`), an option that specifies whether the agent should focus on a goal (`focus`), and an option that specifies the order of evaluating rules (`order`). Additionally, macros can be defined for abbreviating mental state conditions. Actions and module calls can be combined by the `+` operator in a rule to select that more than one action is executed in sequence. Various built-in actions are available. The actions for changing the belief base (`insert`, `delete`), the goal base (`adopt`, `drop`) and for sending a message (`send`) may have a selector prefix. A selector selects the mental models of the agents whose state should be changed and the agents that a message should be send to. The `exit-module` action provides additional control over exiting a module. The `log`action facilitates logging output to a file and the `print`action facilitates printing output to a console.

The KR files that are used as knowledge and beliefs should be files that can be parsed by the KR technology associated with the KR. That is, a Prolog system, for example, should be able to parse the files used as knowledge or as beliefs in the agent; in other words, files used as knowledge

module	:=	*useclause*$^+$ *option** *macro** `module` *id*`(`*parameters*`) { `*rule*$^+$ `}`
useclause	:=	`use` *id* [`as` *usecase*] `.`
usecase	:=	`knowledge` | `beliefs` | `goals`
option	:=	`exit =` *exitoption* `.` | `focus =` *focusoption* `.` | `order =` *orderoption* `.`
exitoption	:=	`always` | `never` | `nogoals` | `noaction`
focusoption	:=	`none` | `new` | `select` | `filter`
orderoption	:=	`linear` | `linearall` | `linearrandom` | `random` | `randomall` | `adaptive`
macro	:=	`define` *id*[`(`*parameters*`)`] `as` *msc* `.`
rule	:=	`if` *msc* `then` (*actioncombo* | `{` *rule*$^+$ `}`) `.` |
		`forall` *msc* `do` (*actioncombo* | `{` *rule*$^+$ `}`) `.` |
		`listall` *var* `<-` *msc* `do` (*actioncombo* | `{` *rule*$^+$ `}`) `.`
msc	:=	*mentalliteral* (`,` mentalliteral)*
mentalliteral	:=	*mentalatom* | `not(`mentalatom`)` | `true` | *id*`(`*parameters*`)`
mentalatom	:=	[*selector*`.`]*mentalop*`(`qry`)`
mentalop	:=	`bel` | `goal` | `a-goal` | `goal-a` | `percept` | *sent*
sent	:=	`sent` | `sent:` | `sent?` | `sent!`
actioncombo	:=	*action* (`+` action)*
action	:=	*id*`(`*parameters*`)` | *selectoraction* | *generalaction*
selectoraction	:=	[*selector*`.`](`insert(`*upd*`)` | `delete(`*upd*`)` | `adopt(`*qry*`)` | `drop(`*qry*`)` | *send*)
send	:=	`send(`*qry*`)` | `send:(`*qry*`)` | `send?(`*qry*`)` | `send!(`*qry*`)`
generalaction	:=	`exit-module` | `log(`*parameters*`)` | `print(`*term*`)`
selector	:=	`(`*parameters*`)` | `all` | `allother` | `some` | `someother` | `self` | `this`
qry	:=	*a valid KR query*
upd	:=	*a valid KR update*
parameters	:=	*term* (`,` *term*)*
term	:=	*a valid KR term*

Table 2: Module Grammar

or as beliefs in an agent, should be Prolog files. The same may but does not need to be true for files that are used as goals. The point is that in a single file one would like, for example, to be able to specify multiple conjunctive goals that the agent might have. But, e.g. a Prolog file, that contains a conjunctive statement does not parse successfully.

3.1 Selectors

A selector can be used to specify the agent(s) that a query or action applies to. A query `bot.bel(`φ`)`, for example, evaluates whether φ is believed by agent `bot`, an action `all.send(`φ`)` sends a message with content φ to all other agents in a MAS, and an action `bot.insert(`φ`)` inserts φ into a mental model of agent `bot`.

A selector is either a name selector, a variable selector, or a quantifier selector. A name selector names a specific agent. A variable selector can only be used in combination with queries and retrieves the name of an agent for which the query is satisfied. The `self` and `this` selectors refer to the agent itself, i.e. to the agent that is evaluating the query or executing the instruction. The other quantifier selectors `all`, `allother`, `some`, and `someother` refer to all or some agents that are known to and modelled by the agent (when applied to queries or mental actions) or to all agents available in the MAS (when applied to a send action).

Mental models play an important role in how selectors of mental queries are evaluated and how mental actions are applied. An agent cannot access or modify the mental state of another agent. In order to evaluate, for example, whether another agent `oaid` believes something, an agent has to model the beliefs of that other agent in a mental model. If an agent has a mental model of agent `oaid`, the query `oaid.bel(`φ`)` can be evaluated on that mental model. On the other hand, without a mental model of an agent, a query with a selector that refers to that agent cannot be evaluated. In other words, a mental model of an agent is essential for evaluating a query that refers

to that agent; if it is not available, an error occurs and the agent's execution will be interrupted. This is even true for the agent itself. The mental state of an agent therefore always contains a mental model for the agent itself with its own knowledge, beliefs, etc. In addition, a mental state can contain additional mental models of other agents that model the knowledge, beliefs, etc. of those agents. It should be noted that there are a few important differences between the agent's own mental model and those of other agents: a model of another agent usually will be *incomplete* and, even worse, the information it contains may be *incorrect*.

Queries with quantifier selectors such as `all.bel(`φ`)` are evaluated by inspecting all relevant mental models that are available in an agent's mental state. `all.bel(`φ`)` is evaluated on all mental models, and is satisfied if φ is implied by the beliefs in each of these models. For the `some` selector, the query is satisfied if it is satisfied in at least one mental model. Note that the agent's own mental state is sufficient to evaluate `all` and `some` selectors even though it is not very useful to apply these selectors if there are no other mental models. Similarly, `allother` and `someother` require that the query is satisfied in all or at least one mental model excluding the model of the agent itself. If there are no other models, this results in an error that will interrupt the execution of the agent. Finally, a selector `self` or `this` requires that the query holds in the mental model of the agent itself. The selectors `self` and `this` are interchangeable for belief queries but not for goal queries. If an agent has more than one attention set that each contain different goals, the `self` selector will evaluate a goal query on the initial attention set (the agent's top goals) and the `this` selector will evaluate the query on the most recent attention set associated with the current module that the agent is executing. There is no need to use these selectors in combination with belief queries or if an agent does not create new attention sets, because a mental query or action without a selector is assumed to refer to the agent itself.

The use of a variable selector in a query plays a role similar to the `some` selector. Note that, if successful, after evaluating a mental query with a `some` selector we only know that there is at least one mental model of an agent that satisfies the query but we do not know for which agent. The difference with using a variable selector is that if the query succeeds, the variable will be bound to the name of an agent whose model satisfied the query. For example, if a mental model of agent `oaid` satisfies a mental query `bel(`φ`)`, the query `(X).bel(`φ`)` succeeds and the variable `X` will be bound to `oaid` (if there is no other model that also satisfies the query).

Selectors applied to mental actions are similar to those applied to mental queries but there are two important differences. First, in contrast with queries, the mental model of an agent that a name selector refers to *does not already need to exist*. If it does not yet exist, it will be created before the mental action is performed. Only the combination of a name selector with a mental action can give rise to the creation of new mental models. Other selectors such as `some` or `all` cannot result in new mental models. For `some` it would not be clear which mental model would need to be created and a mental action that uses the `all` selector is applied only to those mental models that already exist. A problem may arise therefore if a mental action is used with an `allother` selector but no other mental models exist (yet); this will produce an error at runtime and interrupt the agent. It is important, therefore, to make sure that all relevant mental models have been created first before using the `all` and `allother` selectors. A second difference is that selectors of mental actions must be *determinative*, i.e. uniquely determine mental models of the agents to which the action should be applied. A name selector and the quantifier selectors `self`, `this`, `all` and `allother` therefore can be combined with mental actions, but a variable selector and the quantifiers selectors `some` and `someother` cannot. It does not make sense to combine a mental action with a `some` or `someother` selector because it cannot be determined which mental model should be changed.

In summary, a selector of a mental query or action refers to those agents that are *known to and modelled by the agent*, including the agent itself. A mental query that refers to an agent is evaluated on a mental model of that agent, if available, and produces an error otherwise. A mental action that refers to an agent is applied to the mental model of that agent; in case that mental model does not yet exist, it is first created.

Different from mental queries, sent queries are evaluated on the mailbox of an agent. A mailbox contains messages which have three components: a name of the sender, the mood, and the content of the message. There is no requirement here that a mental model of the agent referred to by a selector already exists as that is not needed for evaluating sent queries. For both name and variable selectors, a sent query is evaluated by a basic pattern matching mechanism. If a message has been received for which the name of the sender matches with the name selector and the remaining mood and content components of the message match as well, the query succeeds. The variable selector works similarly as for mental queries and retrieves the name of a sender of a message that matches. In both cases, it is not important whether the sender of the message is known or modelled. This is different again, however, for quantifier selectors that are combined with sent queries. Quantifier selectors of sent queries need to be resolved and we need a mechanism for doing so. For `self` and `this` this is easy enough; both selectors refer to the agent's own name. The natural interpretation of a sent query with a `some` or `someother` quantifier is that it succeeds if the mood and content components match with at least one message in the agent's mailbox (as long as the agent is not itself the sender in case of the `someother` quantifier). The `all` or `allother` quantifiers, however, require some frame of reference. This frame of reference is provided by the available mental models of the agent. A query `allother.sent(`φ`)` succeeds for all queries `Name.sent(`φ`)` succeed with `Name` a name of an(other) agent for which a mental model exists. In other words, a sent query `all.sent(...)` only checks whether all the agents known and modelled have sent a message and *not* whether all agents that exist in a MAS have sent a message.

Finally, we discuss selectors and send actions. Similar to mental actions, the selectors of send actions also must be determinative, which means that variable selectors and the quantifiers `some` and `someother` cannot be used in combination with send actions. For send actions, however, there is no connection with mental models at all. Selectors of send actions can refer to *any agent that exists in a MAS* (when the action is performed). Of course, a name selector should refer to an existing agent in the MAS but no mental model needs to be available for that agent. In principle, an agent can also send a message to itself (as a mental note) and for that case the `self` or `this` selectors can be used. The `all` and `allother` selectors will result in a message being sent to all (other) agents that exist in the MAS. Note that this is quite different from the mental actions which require that the agents should be known. Where a mental action is a kind of private action changing only the mental state of the agent, a send action with an `all(other)` selector is a broadcast action and more like shouting in a public area where everyone including agents that are unknown to the agent can hear what the agent says. [**TODO:** this naturally raises the question whether we should also introduce a concept of a *group*, adding e.g. `join(`*group*`)` and `leave(`*group*`)` as built-in actions that allow agents to structure their communication.]

A launch policy in a MAS file should satisfy the following **constraints**:

1. *A selector should not be used when it makes no sense.* Here, we distinguish between redundant and meaningless use of selectors. Things can go wrong when a mental model of another agent has not yet been created. In the following cases, the use of a selector is incorrect:
   - a term that is used as selector and refers to an agent that is not registered in the MAS (an exception should be thrown at runtime in this case).
   - a term that refers to another agent (that exists in the MAS but) for which no mental model has been created yet *and* that is used as a selector of a mental query (an exception should be thrown at runtime in this case).[1]
   - An unbound variable used as selector combined with a mental or send action.[2] (An error should be generated at compile time if it can be established that a variable used as selector will never be bound, and an exception should be thrown at runtime if the variable is not bound when the action is performed.)

[1] If a selector applied to a mental action (e.g. `insert`) refers to an agent for which no mental model has been created, the mental model should be created to enable the execution of the action.

[2] Note that even if the agent has no mental models of other agents, a query `(Var).bel(`φ`)` can still be evaluated on the agent's own belief base and, if successful, `Var` should be bound to the name of the agent in that case.

- The use of the selectors `allother` and `someother` if there are no other agents in the MAS (an exception should be thrown at runtime in this case).
- `allother` and `someother` applied to a mental query or action when the agent does not have a mental model of another agent (an exception should be thrown at runtime in these cases).
- `some` and `someother` applied to a mental or send action. It should be possible to uniquely identify the agent whose mental model should be updated or who should receive a message. (At compile time an error should be generated.)

2. *A selector should not be used redundantly.* Although not wrong, the following combinations or selectors and queries or actions are redundant:[3]
   - `self` and `this` combined with `bel` or `percept`. (At compile time an error should be generated.)
   - `self` and `this` combined with `insert`, `delete`, or `drop`. (At compile time an error should be generated.)
   - The use of the `some` or `all` quantifiers combined with mental queries if there will never be any mental models of other agents created. (A compile time error should be generated; runtime checks are not considered worthwhile in this case.)

4 Action Specifications

An action specification file consists of use clauses and one or more action specifications. The use clauses, if any, should reference KR files that define the predicates that are used in the pre- and post-conditions of actions. An action specification defines the actions name and parameters, whether it is an internal or external action, and the action's pre- and post-condition.

specification	:=	*useclause*$^+$ *actionspec*$^+$
useclause	:=	`use` *id* [`as knowledge`] `.`
actionspec	:=	`define` *id*[`(`*parameters*`)`] [*asclause*] `with pre{` *qry* `} post{` *upd* `}`
asclause	:=	`as` (`internal` | `external`)
qry	:=	*a valid KR query* | `true`
upd	:=	*a valid KR update* | `true`
parameters	:=	*term* (`,` *term*)*
term	:=	*a valid KR term*

Table 3: Action Specification Grammar

5 Test Files

A test file consists of use clauses and sets of temporal conditions that should hold whilst an agent executes an action in its environment. The use clause should reference the MAS file that defines the agents and the environment they operate in, any module or action specification that is referenced by an action, and any KR file that defines the predicates that are referenced in the temporal conditions.

When executing a test, all agents in the MAS are started and placed in the environment, although no code is automatically executed for these agents. A test file allows specifying actions that one or more agents should execute, and the conditions that should hold before, after, or during the execution of these actions. Different agents can have different tests, but tests can be shared

[3]Strictly speaking, the use of the name of the agent itself as selector is also redundant in the cases listed below, but there is no way to detect this at compile time and we do not consider it worthwhile to check this at runtime.

amongst agents as well. All agent tests are evaluated at the same time though. The test grammar re-uses several clauses from the module grammar, including *id*, *actioncombo*, and *msc*, and re-defines (extends) the *mentalatom* clause to allow also conditions of the form `done(`*actioncombo*`)` in a mental state condition.

test	:=	*useclause*$^+$ [*timeout*] *moduletest** *agenttest*$^+$
useclause	:=	`use` *id* [`as knowledge`] `.`
timeout	:=	`timeout =` *integer* `.`
moduletest	:=	`test` *id*[`(`*parameters*`)`] `with` [`pre{` *msc* `}`] [`in {` *testcondition*$^+$ `}`] [`post{` *msc* `}`]
testcondition	:=	(`always` | `never` | `eventually`) *msc* `.` | `reactTo(`*msc* `,` *msc*`)` `.`
agenttest	:=	*id* (`,` *id*)* `{` *testaction*$^+$ `}`
testaction	:=	`do` *actioncombo* [*runcondition*] `.`
runcondition	:=	`until` *msc*
mentalatom	:=	[*selector*`.`]*mentalop*`(`*qry*`)` | `done(`*actioncombo*`)`

Table 4: Test Grammar

We note that, in classical temporal logic, `never` *msc* means the same as `always not` *msc*. In case we could always move the `not` operator inwards to the mental atom level, we could do without the `never` operator and it would be syntactic sugar; we cannot always do this however, as we cannot e.g. distribute the `not` operator over a conjunction operator. In classical temporal logic, the `eventually` operator can be defined in terms of the `always` operator as well, but we cannot use this definition because in our grammar we cannot prefix a temporal operator with `not`. Finally, in classical temporal logic, we could have defined `reactTo(`*msc*`,`*msc*`)` as `always (`*msc*$\rightarrow$`eventually` *msc*`)`. Informally, `reactTo(`msc_1`,`msc_2`)` means that whenever msc_1 holds, eventually msc_2 holds, or, more casually, msc_1 is always followed by msc_2.

A test condition can have either one of three values: *undetermined*, *passed*, or *failed*. Initially, the test conditions in a certain section all have the value *undetermined*. The value of a test condition is (re-)evaluated each time that a mental state is changed. The temporal operator of the test condition determines how the value is updated:

- `always` *msc*: the value is changed to *failed* if *msc* does not hold in the mental state in which the condition is evaluated; the value is changed to *passed* if the test is terminated and the value still is *undetermined*; otherwise, its value remains *undetermined*.
- `never` *msc*: the value is changed to *failed* if *msc* holds in the mental state in which the condition is evaluated; the value is changed to *passed* if the test is terminated and the value still is *undetermined*; otherwise, its value remains *undetermined*.
- `eventually` *msc*: the value is changed to *passed* if *msc* holds in the mental state in which the condition is evaluated; the value is changed to *failed* if the test is terminated and the value still is *undetermined*; otherwise, its value remains *undetermined*.
- `reactTo(`msc_1`,`msc_2`)`: if the test is terminated, the value is changed to *passed* if every mental state where msc_1 holds has been followed by a mental state where msc_2 holds; otherwise, the value is changed to *failed*. While the test has not been terminated yet, the value is *undetermined*. We note that the value of a `reactTo(`msc_1`,`msc_2`)` condition is vacuously set to *passed* if msc_1 never holds.

A test consists of two different kinds of definitions: module tests and agent tests. In general, a module test contains a set of conditions that a module is expected to adhere to, whilst an agent test determines which modules are executed. An agent test can also set-up a mental state to test a module with.

A module test can have a pre- and a post-condition. These conditions are mental state conditions that should apply when respectively entering and exiting the module that is being tested. In addition, an `in` section can be given. Such a section consists of a set of temporal conditions that

are evaluated during the module's execution, as explained above. A test is aborted as soon as a temporal condition is assigned the value *failed*, or when any pre- or post-condition did not hold. In such a case, the test is regarded as a failure as well. Note that when a test is aborted some of the test conditions may still have the value undetermined, but that this is never the case for a test that terminates normally. We note that we might have chosen for these conditions to be specified at a module definition itself, but this would not allow multiple combinations of conditions to be specified for different test cases (i.e., different post-conditions for certain pre-conditions) .

An agent test consists of a set of `do` statements, referred to as test actions. Each test action executes the action(s) and/or module(s) that are part of the *actioncombo*. A `do` *actioncombo* statement may not terminate because it executes a module that does not terminate. There are two options available to terminate such sections nevertheless: specifying a *run condition* or a *timeout*. The first option adds a run condition to the statement of the form `until` *msc*. A run condition specifies a termination condition. The `until` *msc* condition terminates a `do` *actioncombo* statement when *msc* holds. Note that such a run condition *does not guarantee termination* of a `do` *actioncombo* section. The second option is to add a *timeout* option. A timeout is global and specifies how many time (in seconds) is allowed to pass before the entire test should have been completed. If a timeout happens, the test is terminated. We note that if a test is terminated because of a timeout, this does not always imply that the test is a failure; if all test conditions are passed, the test is considered to have been passed as well. Finally, we remark that these options are not exclusive but can be used both.