VITAL

Visualization and Implementation of Temporal Action Logic

Jonas Kvarnström, KPLAB, IDA
jonkv@ida.liu.se

DRAFT; version 0.53 (Wed, 09 May 2001 10:57:40 +0200)

Chapter 1

VITAL: The Input Language

This document describes the input language used by VITAL. The reader is assumed to have some knowledge of TAL and the $\mathcal{L}$(ND) narrative description language; we will often not describe that language in detail, instead concentrating on how certain constructions in plain $\mathcal{L}$(ND) are written in the VITAL input language.

Since a large amount of code is shared between VITAL and TALplanner, including the parser, many of the language constructions are identical or very similar for both tools. Instead of writing two completely separate manuals and duplicating a large amount of text, minor differences between the two languages will be pointed out in this manual, and the TALplanner language manual often refers back to this document.

The input language contains a number of extensions to the plain $\mathcal{L}$(ND) language. Many of these extensions are experimental, and may either disappear in the future or be provided with a formal translation into pure $\mathcal{L}$(ND). Such extensions should also be marked clearly below.

For both tools, an input file starts with a number of declarations of features, actions, and similar items. This section is followed by a set of $\mathcal{L}$(ND) statements (or extended statements). We will describe these declarations and statements in the following chapters, but first, we will describe some general properties of the input language and how it is described in this manual.

1.1 The Input Language

As in $\mathcal{L}$(ND), each declaration or statement in the input language is labeled. Unlike $\mathcal{L}$(ND), each label starts with a #, as in #obs or #dom. The reasons for this are purely technical: If statement labels were pure identifiers (obs or dom), labels of future statement classes could clash with the identifiers users have chosen for actions, features, and so on.

For similar reasons, options and similar keywords are always preceded with a colon (for example, :hide and :splitpri). Such colon keywords will generally be explained in a labeled list, where keywords that *must* be specified are underlined.

1.2 Statement Classes

This section will list all the statement classes available in VITAL and TALplanner.

The following statement classes belong to the declaration part of a narrative description:

#domain	Declare a value domain (Section 2.2)
#feature	Declare a feature (Section 2.3)
#action	Declare an action (Section 2.4)
#timevar	Declare a temporal variable (Section 2.5)
#timeconst	Declare a temporal constant (Section 2.6)
#valuevar	Declare a value variable (Section 2.7)
#valueconst	Declare a value constant (Section 2.8)
#function	Declare a pre-defined function (Section 2.9)
#relation	Declare a pre-defined relation (Section 2.10)
#maxocc	Change timeline lengths (Section 2.11)
#timescale	Change the time scale (Section 2.12)
#attach	Add a semantic attachment (Section 2.13)
#resource	Declare a resource (Section 2.14 and the TALplanner manual)

The following statement classes belong to the statement part of a narrative description.

#obs	Observation Statement (Section 4.1.1)
#dom	Domain Constraint Statement (Section 4.1.2)
#init	Initialization Statement (Section 4.1.3)
#dep	Dependency Constraint Statement (Section 4.2.1)
#occ	Action Occurrence Statement (Section 4.2.2)
#acs	Action Law Schema (Section 4.2.3)
#define	Feature Definition Statement (Section 4.3.1)
#objects	Object Declaration Statement (Section 4.3.2)
#assert	Assertion Statement (Section 4.3.3)
#goal	Intended Goal Statement (Section 4.4 and the TALplanner manual)
#control	Non-modal Control Statement (Section 4.4 and the TALplanner manual)
#modal	Modal Control Statement (Section 4.4 and the TALplanner manual)
#redundancy	Redundancy Control Statement (Section 4.4 and the TALplanner manual)
#operator	Operator Definition Statement (Section 4.4 and the TALplanner manual)
#heuristic	Heuristic Specification (Section 4.4 and the TALplanner manual)

Chapter 2

Declarations

Plain TAL provides no language or syntax for declarations; the set of features, actions, value domains, and so on is assumed to be given. Therefore, a new declaration syntax had to be invented for VITAL and TALplanner. This syntax is similar to the existing $\mathcal{L}$(ND) syntax in that it consists of a set of labeled statements.

This chapter will describe all declarations and all options available for each declaration class. Note that all declarations must be given before the first logical statement (observation, domain constraint, etc).

2.1 Pre-declared domains, values and variables

Even before a narrative description is parsed, VITAL or TALplanner automatically declares the following entities:

- The two boolean values true and false
- The boolean value domain boolean = {true, false}
- The boolean value variables boolean, boolean1, boolean2, boolean3, boolean4, and boolean5.
- The temporal variables t, t1, t2, t3, t4, and t5.

See also "Arithmetic Operators" (Section 5.2) for information on functions/features that are generated automatically the first time they are encountered in a narrative description. Such functions include certain arithmetic operators (addition, subtraction, and so on).

2.2 Declaring value domains

TAL uses an order-sorted type structure with finite domains. The exact restrictions are still somewhat unclear, but in VITAL, the type structure must be a forest, where each child domain is a strict subset of its unique parent domain.

The boolean domain, containing the two values true and false, is declared automatically (see Section 2.1).

A root domain is declared as follows:

#domain *name* [:no-vars] :elements {*val1*, *val2*, ... , *valn*}

A child domain is declared as follows:

#domain *name* [:no-vars] :parent *parent* :elements {*val1*, *val2*, *val7*, *val12*, ... , *valn*}

Unless the keyword :no-vars is specified, declaring a domain called *name* automatically declares six value variables of that sort, called *name*, *name1*, *name2*, *name3*, *name4*, and *name5*. An exception is made for names that are already used by declared variables, values, or other entities. That is, after

#domain ball :elements { ball1, ball2 }

there are four pre-declared value variables of sort ball: ball, ball3, ball4, and ball5. Additional value variables, with arbitrary names, can be declared using #valuevar (Section 2.7).

Note that empty value domains can be created by specifying :elements {}.

:no-vars Do not automatically declare value variables for this domain.

:parent Specify the parent of a child domain.

:elements Specify the elements of a domain.

For a root domain (a domain without a parent), the argument is a comma-separated list of elements (identifiers) within braces. Elements must not be present in any existing value domain. The list can be empty.

For a child domain, the argument is a comma-separated list of elements (identifiers) within braces, where each item must be a member of the parent value domain. Values must be specified in the same order as they appear in the declaration of the parent domain. The list can be empty.

There are other ways of defining domains. See Section 5.1 if you want to declare a fixed point or integer value domain (which is done using a #domain statement with the :integer and :fixedpoint options). Value domains can also be defined by semantic attachments (Section 2.13).

2.2.1 Ranges of values in root domains

When you define a root domain, you normally explicitly enumerate every value in the domain. However, VITAL and TALplanner also allow the use of an extended syntax to allow you to define a set of values with a common prefix followed by an index:

#domain *name* [:no-vars] :elements {*val1*, *pre*<n_1..n_2>, ... , *valn*}

For example, the following defines a domain object with the values ball1, ball2, ... , ball17, cube7, cube8, ... , cube22:

#domain object :elements { ball<1..17>, cube<7..22> }

This is mainly used when random planning problem instances are generated for TALplanner.

2.2.2 Ranges of values in subdomains

Using ranges is also useful when defining subdomains. However, in this case each element that will be added to the subdomain is already present in the parent domain. Therefore, VITAL and TALplanner have enough information to allow a more general form of value range: Each item in the :elements definition can be a range of values [*val1* .. *val2*] where both values are members of the parent domain. Every value between *val1* and *val2* (inclusive) will be added to the subdomain, where values are ordered in the declaration order.

```
#domain object :elements { ball<1..17>, cube<7..22> }
#domain someobjects :parent object :elements { ball2, ball7, [cube10 .. cube18], cube22 }
#domain otherobjects :parent object :elements { ball5, [ball14 .. cube8], cube22 }
```

The final domain, otherobjects, will contain the elements ball5, ball14, ball15, ball16, ball17, cube7, cube8, and cube22.

2.3 Declaring features

A feature without arguments is declared as follows:

#feature *name :options*

A feature with arguments of sort $s_1, \ldots, s_n$ is declared as follows, where each arg_i is either the name of a value domain or the name of a value variable of that domain:

#feature *name*(*arg1*,*arg2*, ... ,*argn*) *:options*

Multiple features can be declared in the same #feature statement. Simply write a comma-separated list of features before the list of options. The options will be applied to all features in the list.

#feature *name*(*arg1*,*arg2*, ... ,*argn*), *name2*, *name3*(*arg*) *:options*

For example:

```
#feature alive :domain boolean

#feature at1(integer, integer, integer) :domain boolean

#valuevar x, y, z :domain boolean
#feature at2(x, y, z) :domain boolean // More obvious what the args mean
```

Features are normally limited to $2^{31} - 1$ instances (exceptions will be noted below). In other words, $\Pi_{i=1}^{n}|s_i|$ must be strictly less than 2^{31}. VITAL will notify you if this limit is exceeded. (Note that you will generally run out of memory even with far less than 2^{31} feature instances.)

There is a large set of options that can be specified in arbitrary order. The only required option is the :domain specification.

:domain d — Specify the value domain of the feature. Here, *d* can be the predefined domain boolean or the name of a value domain previously declared using a #domain declaration or a semantic attachment.

The following options control whether a certain feature is persistent, durational, or semidynamic (see the explanation of :semidynamic below). These options should only be used in VITAL, not in TALplanner – the planner will most likely never support semidynamic features, and although it will support durational features, this is not completely implemented yet.

Although some versions of TAL allows this characterization of a feature to vary over time, this is not yet permitted in VITAL: Each feature must be either persistent at all timepoints, durational with the same default value at all timepoints, or semidynamic.

:durational v Make this feature durational, rather than persistent (the default) or semidynamic. Make v the default value, for all timepoints (meaning that at any timepoint where the feature is unoccluded, it will take on its default value).

A :durational feature cannot be declared :semidynamic, :defined, :valuetype, :injective-always, or :double-injective. It can be declared :injective, if it is a boolean feature and the default value is false. It can also be declared ::function, although this would not make sense since ::function features cannot be occluded (and therefore the durational feature would have to take on its default value at all timepoints).

:semidynamic Make this fluent semidynamic, rather than persistent (the default) or durational.

True dynamic fluents are not affected by nochange or default value assumptions, but can vary freely (except for restrictions placed by ordinary logic formulas involving the dynamic fluents, of course). This is equivalent to occluding the fluent in the interval $[0, \infty)$.

However, VITAL works by performing constraint propagation over a finite part of the timeline for each fluent, and allowing true dynamic fluents would require considering the entire (infinite) timeline. Therefore, true dynamic fluents are not supported.

Instead, VITAL supports *semidynamic* fluents – fluents that are dynamic over an initial prefix $[0, t_*]$ of the timeline and persistent in (t_*, ∞), where t_* is the final timepoint of occlusion as defined in Chapter 9. Thus, marking a fluent as semidynamic is equivalent to occluding it in [0, $maxocc], except that such syntactic constructions are not allowed by VITAL (see Section 3.1 for an explanation of the $maxocc term).

TALplanner does not support dynamic or semidynamic fluents, and most likely never will.

A :semidynamic feature cannot be declared :durational, :defined, :valuetype, :injective, :injective-always, :double-injective, or :function.

The following option controls certain aspects of how VITAL splits partial interpretations in order to reason by cases. It does not affect the operation of TALplanner.

:splitpri n Set the split priority for this feature to *n*. The default split priority is 1024; lowering the priority makes instances of this feature more likely to be chosen

when selecting a splitpoint, while raising it makes it less likely to be chosen. See Chapter 8 for more detailed information.

The following options control the graphical appearance of a feature in VITAL. They do not affect the operation of TALplanner. Note that these options only specify a default which can be modified by right-clicking on the feature names in a narrative timeline window.

:showname Rather than showing a colour-coded timeline for each value the feature can take on, show the names of the values the feature can take on for each timepoint. For features with large value domains, this often saves a large amount of space in the graphical representation.

The main disadvantage of using this option is that for partial interpretations where a feature may take on a value in a given (possibly large) set, VITAL might not have enough space to show you the entire set, instead having to fall back to showing you the *size* of the set (the number of possible values the feature can take on, rather than the actual values).

:hide Do not show this feature in the graphical representation.

The following options allows the narrative constructor (you) to specify that certain conditions will be guaranteed to hold at all timepoints, thereby allowing VITAL or TALplanner to enable certain optimizations. Some of the names are extremely counter-intuitive (due to being inspired by similar declarations in another tool) and will be changed in the future.

:function This feature is not a fluent, that is, it cannot vary over time.

Setting the :function flag for a feature f serves two different purposes:

First, it acts as an actual constraint on the values of f – semantically, it is equivalent to adding a logical constraint that $\forall t, t', \overline{x}.value(t, f(x)) = value(t', f(x))$. Since this makes it pointless to occlude the feature at any timepoint, VITAL will not allow the feature to occur within the scope of an R, I or X operator, and TALplanner will not allow the feature to be assigned a new value in an operator definition.

Second, it allows VITAL to use a more efficient data structure for the given feature or features, since there is no need to store different values at different timepoints, and the nochange axiom does not need to be checked or applied. (TALplanner's performance is not affected by this since it analyzes this property automatically regardless of whether the :function option is set or not. VITAL will also do this in the future.)

A :function feature cannot be declared :semidynamic. It can be declared :durational, although this would not make sense since ::function features cannot be occluded (and therefore the durational feature would have to take on its default value at all timepoints).

:injective This is only applicable to boolean features f with at least one argument, and specifies that $\forall t, \overline{x}, y, z.([t]\ f(\overline{x}, y) \wedge f(\overline{x}, z) \rightarrow y = z)$. In other words, the

feature is in all models and at all timepoints a boolean representation of a partial function from the first $n-1$ arguments to the last argument.

An example of this would be the holding(block) feature in the blocks world planning domain, given that there is only one gripper: The gripper is holding a unique block or no block at all.

This option also has an impact on the performance of VITAL and TALplanner, since instead of storing the boolean representation of a partial function (for example, the boolean holding(block) feature) it permits the system to store the partial function itself (for example, a holding' feature whose value is a block or null). This is more efficient in terms of memory as well as time. (The details of this automatic conversion is available in +++.)

An :injective feature cannot be declared :semidynamic, :defined, :valuetype, :injective-always, or :double-injective.

:injective-always This is only applicable to boolean features f with at least one argument, and specifies both that $\forall t, \overline{x}, y, z.([t]\ f(\overline{x}, y) \wedge f(\overline{x}, z) \rightarrow y = z)$ and that $\forall t, \overline{x} \exists y.[t]\ f(\overline{x}, y)$. In other words, the feature is in all models and at all timepoints a boolean representation of a total function from the first $n-1$ arguments to the last argument.

An example of this would be the in-city(location,city) feature in the logistics planning domain: A location is always in exactly one city.

This option also has an impact on the performance of VITAL and TALplanner, since instead of storing the boolean representation of a total function (for example, the boolean in-city(location,city) feature) it permits the system to store the partial function itself (for example, a in-city'(location) feature whose value is a city). This is more efficient in terms of memory as well as time. (The details of this automatic conversion is available in +++.)

An :injective-always feature cannot be declared :durational, :semidynamic, :defined, :valuetype, :injective, or :double-injective. It can be declared :durational with default value false.

:double-injective This is only applicable to boolean features f with two arguments, and specifies both that $\forall x, y, z.(f(x, y) \wedge f(x, z) \rightarrow y = z)$ and that $\forall x, y, z.(f(x, z) \wedge f(y, z) \rightarrow x = y)$. In other words, the feature is a boolean representation of a partial function from the first argument to the second argument *and* a boolean representation of a partial function from the second argument to the first argument.

An example of this would be the on(block1,block2) feature in the blocks world planning domain: A block is always on top of at most one other block, and given a block, there is at most one block on top of it.

This option also has an impact on the performance of VITAL and TALplanner, since instead of storing the boolean representation of a partial function (for example, the boolean on(block1,block2) feature) it permits the system to store the two partial functions themselves (for example, a below(block1) feature whose value is a block block2 or null, together with an above(block2)

feature whose value is a block block1 or null). This is more efficient in terms of memory as well as time. (The details of this automatic conversion is available in +++.)

A :double-injective feature cannot be declared :durational, :semidynamic, :defined, :valuetype, :injective, or :injective-always.

The following options are used for declaring features that are defined in terms of logical formulas.

:defined This feature is defined in terms of a logical formula. This can currently only be used for features with a boolean value domain. The feature takes on the value true for exactly those arguments that make a certain formula is true. This formula must be specified in the statement section using a #define statement (Section 4.3.1).

:uncached Do not cache the calculated values for this feature (which must be :defined). If :uncached is not specified, then whenever the value of the feature is calculated for a specific set of arguments, the calculated value is saved for future reference. Caching usually (but not always) saves time, but may use large amounts of memory.

Although features are normally limited to less than 2^31 instances, this limit does not apply to uncached defined features. For example, suppose you want to declare a defined feature less(a, b) where the sorts of a and b have size 2^{20}. The resulting feature will have 2^{40} instances, but this is not a problem as long as the feature is uncached.

Finally, there are a few more options that do not fit into either of the categories above.

:cut In VITAL (but not TALplanner), the user can specify that certain features, temporal constants and value constants should be 'cut'. This was inspired by 'cut' in FINDER, and is also similar to 'cut' in Prolog.

Internally, VITAL works with partial interpretations, each of which corresponds to a number of total interpretations. After propagating constraints as far as possible in a partial interpretation, the system looks for a splitpoint, in order to split the partial interpretation into a number of children.

Before the system looks for a splitpoint, it first checks whether the partial interpretation is already equivalent to or included in some partial *model* that has already been found, *modulo* the features and constants that are declared :cut. If so, the partial interpretation is immediately discarded: Even if it would turn out to be a model, it would only differ in :cut features and constants, which are not considered significant.

Whether using :cut makes VITAL unsound or incomplete depends on whether you consider the set of models generated or the set of conclusions that can be drawn from those models. If we consider the set of models generated by the system, using :cut will not yield non-models but might remove some valid

models and therefore makes VITAL sound but incomplete in this sense. If we consider the set of conclusions that can be drawn from the generated models, using :cut will not remove any valid conclusions but (due to the additional filtering) may yield new conclusions, and therefore makes VITAL unsound but complete in this sense.

:valuetype dom Instead of creating a normal feature, create a pre-defined feature which is true if and only if its argument belongs to the given domain.

The feature must take a single argument whose domain is a parent of *dom*. The value domain of the feature (specified using :domain) must be boolean. Also, the feature must not be declared :durational, :injective, :injective-always, :double-injective, or :defined.

For example, in order to determine whether a given vehicle is an airplane:

```
#domain vehicle :elements { ... }
#domain airplane :parent vehicle :elements { ... }
#feature is_plane(vehicle) :domain boolean :valuetype airplane
```

:noinit A keyword temporarily used for supporting certain features of PDDL in TALplanner. Do not use this option.

:fault Create a fault feature. This was used in a diagnosis procedure and is currently not supported.

There are several other ways of defining features. See Section 2.9 if you want to declare a pre-defined function, Section 2.10 if you want to declare a pre-defined relation, and Section 5.2 and Chapter 6 if you want to use arithmetic functions and ordering functions on value domains. Features can also be defined by semantic attachments (Section 2.13).

2.4 Declaring actions

For VITAL, actions (or action types) are declared using #action statements; later, the actions that have been declared are *defined* using #acs statements (action law schemas, Section 4.2.3). Exactly which actions occur is specified using action occurrence statements (#occ, Section 4.2.2). For TALplanner, you should use #operator statements instead (see the TALplanner manual).

An action type without arguments is declared as follows:

#action *name*

An action type with arguments of sort $s_1, \ldots, s_n$ is declared as follows, where each arg_i is either the name of a value domain or the name of a value variable of that domain:

#action *name*(*arg1*,*arg2*, ... ,*argn*)

Multiple action types can be declared in a single #action statement using a comma-separated list:

#action *name*(*arg1*,*arg2*, ... ,*argn*), *name2*, *name3*(*arg4*)

For example:

```
#action Load, Spin, Fire
#action travel(person, location, location)
#action board(person, airplane)

#valuevar from, to :domain location
#action travel(person, from, to)
```

2.5 Declaring temporal variables

VITAL and TALplanner automatically creates the six temporal variables t, t1, t2, t3, t4, and t5. If you need more variables, or you want your variables to have more meaningful names, additional temporal variables can be declared as follows:

#timevar *name1*, *name2*, ... , *namen*

2.6 Declaring temporal constants

In VITAL, temporal constants can be declared as follows:

#timeconst *name1*, *name2*, ... , *namen :options*

Any options given at the end of the declaration are applied to the entire list of constants. The following options are available:

:min n	A lower limit on the value of the temporal constant (inclusive). This value must be a non-negative integer.
:max n	An upper limit on the value of the temporal constant (inclusive). This value must be a non-negative integer.
:cut	See :cut in Section 2.3.
:parameter	Allows the value of this constant to be specified from the command line. This can be useful when you want to create parameterized narratives (see Chapter 10).
:default n	Specifies a non-negative default value for the constant in case its value is not specified as a parameter on the command line. A default value can only be specified if the :parameter keyword is used.

TALplanner currently does not support temporal constants.

2.7 Declaring value variables

Whenever a domain is declared, a number of value variables are automatically created (Sections 2.2 and 2.1). If you need more variables, or you want your variables to have more meaningful names, additional value variables can be declared as follows:

#valuevar *name1*, *name2*, ... , *namen* :domain *dom*

2.8 Declaring value constants

In VITAL, value constants can be declared as follows:

#valueconst *name1*, *name2*, ... , *namen :options*

Any options given at the end of the declaration are applied to the entire list of constants. The following options are available:

:domain dom The domain of the value variable(s).

:cut See :cut in Section 2.3.

:parameter Allows the value of this constant to be specified from the command line. This can be useful when you want to create parameterized narratives (see Chapter 10).

:default v Specifies a default value for the value constant in case its value is not specified as a parameter on the command line. The default value can be a value constant (that is, a value defined in a #domain definition); certain other forms of value terms are also accepted, as long as VITAL can easily compute the value the term corresponds to at parsing time.

A default value can only be specified if the :parameter keyword is used.

TALplanner currently does not support value constants.

2.9 Declaring predefined functions

Functions that are independent of time can be represented as features (see Section 2.3, especially the :function keyword). This also allows the user to specify the value of that function using ordinary observation statements or domain constraints.

In some cases, however, it is easier to specify such functions as a simple enumeration of function values. This can be done using #function statements, with or without a default value:

#function *name*(*dom1*, ... ,*domn*) :domain *dom* = { *name*(*arg1*, ... ,*argn*) == *val*, ... }

#function *name*(*dom1*, ... ,*domn*) :domain *dom* :default *def*
:exceptions { *name*(*arg1*, ... ,*argn*) == *val*, ... }

For example:

#function inv(colour) :domain colour = { inv(red) == cyan, inv(blue) == yellow, ... }

If a default value is specified, any feature instance not mentioned in the initialization list is given the default value. Otherwise, all feature instances must be given specific values in the initialization list. Feature instances can be enumerated in any order.

2.10 Declaring predefined relations

Relations, in this context, are simply features with a boolean value domain. Relations that are independent of time can be represented as features (see Section 2.3, especially the :function keyword). This also allows the user to specify the value of that relation using ordinary observation statements or domain constraints.

In some cases, however, it is easier to specify such relations as a simple enumeration of those feature instances that map to a true value, or possibly to a false value.. This can be done using #relation statements.

A relation declaration can specify a default value. If no default value is specified, or if false is specified as default value, any feature instance mentioned in the initialization list maps to true and all other feature instances map to false. If true is specified as a default value, any feature instance mentioned in the initialization list maps to false and all other feature instances map to true.

#relation *name*(*dom1*, ... ,*domn*) = { *name*(*arg1*, ... ,*argn*) == *val*, ... }

#relation *name*(*dom1*, ... ,*domn*) :default *def* :exceptions { *name*(*arg1*, ... ,*argn*), ... }

For example:

#relation square(integer) :default false :exceptions { square(1), square(4), square(9), ... }

Feature instances can be enumerated in any order.

2.11 Altering Timeline Lengths: #maxocc

Although each TAL fluent has an infinite timeline, VITAL only considers a finite part of that timeline. We assume the existence of a timepoint t_* such that in all preferred models, all fluents are unoccluded after t_*. This timepoint can under certain circumstances be calculated by VITAL, but can also be altered using a #maxocc declaration specifying the value t that t_* should take on:

#maxocc t

See Chapter 9 for further information.

2.12 Time Scale Declarations

In order to facilitate the use of VITAL for simulating physical systems with varying temporal granularity, VITAL provides an extension for specifying scaling factors for timepoints. A scaling factor r must be a floating point literal or a fixed point (Section 5.1) value term that can be evaluated to a constant, and is specified using a #timescale declaration:

#timescale r

This does not automatically alter the way temporal terms are interpreted. Instead, the VITAL parser allows the use of a special scaling macro $scale(n), where n must be a numeric temporal constant. The parser immediately converts this into the numeric temporal constant $\lfloor(n/r)+0.5\rfloor$; in other words, it divides the argument of $scale by r and rounds the result to the nearest integer timepoint.

Given that the time scale is 4, the following two statements are equivalent:

```
#occ [5,6] SetVoltage(pump, 0.57)
#occ [$scale(20),$scale(20)+1] SetVoltage(pump, 0.57)
```

2.13 Semantic Attachment

VITAL allows additional code to be linked into itself. Such code can provide external definitions for features and value domains using semantic attachment, and can also hook into the system in order to provide additional functionality such as specialized visualization modules for specific narrative classes. There is a well-defined interface for such attachments, which will be documented in a future version of this manual. In this version, however, we only describe the narrative syntax used to link a piece of code into VITAL.

There are two variations of the #attach statement. If a quoted string is provided as an argument, this string is assumed to be the name of a Java class in the class path, complying with the logic.semantics.SemanticAttachment interface. This may optionally be followed by a quoted argument string, which is passed to and parsed by the linked code:

```
#attach "se.liu.ida.jonkv.MyAttachment"
#attach "se.liu.ida.jonkv.MyAttachment" "some arguments"
```

On the other hand, if the first argument is a plain identifier id, this is assumed to be an abbreviation for the name of an attachment included with VITAL. The identifier is expanded to "logic.semantics" + id.toLowerCase() + "." + id + "Attachment". For example, "FixedPoint" is expanded to "logic.semantics.fixedpoint.FixedPointAttachment". This usage will probably disappear in the near future, as new syntax is introduced for the built-in attachments.

```
#attach FixedPoint "fixedpoint - 0 100 2
```

2.14 Planning-related Declarations

For completeness, we will now briefly explain a number of planning-related declarations that are supported by the parser. These declarations have no effect on VITAL, and should only be used in plan narratives. They are explained in more detail in the TALplanner documentation.

The planning-related declarations are the following:

- Resource declarations, labeled #resource, are used to specify resource types.

Chapter 3

Terms and Formulas

The previous chapter described how entities such as values, variables, constants and features could be declared. In this chapter, we will describe how such entities can be combined into terms and logical formulas in VITAL and TALplanner, while the next chapter will describe how such formulas can be used in various statement classes.

Please note that this chapter is not intended to *explain* the meaning of each term and formula. It is assumed that the reader knows $\mathcal{L}$(ND) and is interested in knowing how a certain formula can be translated into the text-only representation used by VITAL.

3.1 Temporal Terms

Temporal terms are often denoted by τ. VITAL and TALplanner provides the following forms of temporal terms:

Numeric temporal constants. Non-negative numbers 0, 1, 2, ...

Symbolic temporal constants. Constants declared using #timeconst (Section 2.6).

Temporal variables. The pre-declared temporal variables t, t1, t2, t3, t4, and t5, and other temporal variables declared using #timevar (Section 2.5).

Temporal expressions on the form $\tau + \tau'$ and $\tau - \tau'$. Addition and subtraction are the only operators available for temporal terms.

Temporal cast expressions on the form $maketime($\omega$), where ω is a value term whose type is an integer type with minimum value 0 (Section 5.1). This is an experimental VITAL extension that allows value terms to be used as timepoints. The extension may disappear in the future, if the type structure is modified to allow the specifications of value sorts that are finite subsorts of the temporal sort.

Temporal scale expressions on the form $scale($\tau$), where τ is a temporal numeral. This is an experimental VITAL extension intended to allow the user to easily modify the granularity of a physical simulation; see Section 2.12 for more information.

Maxocc expressions of the form $maxocc. This is an experimental VITAL extension that allows the user to refer to the timepoint t_* after which no occlusion is allowed, as defined in Chapter 9. Note that maxocc expressions cannot occur within the temporal context of an R, I or X formula (Section 3.5).

Infinity is not a true temporal term, but can be used in the temporal context of a fixed fluent formula, where it is denoted by $inf (Section 3.5).

3.2 Value Terms

Value terms are often denoted by ω. VITAL and TALplanner provides the following forms of value terms:

Value name constants. Value names declared in a #domain declaration (Section 2.2), or one of the predefined value names true and false in the boolean domain.

Symbolic value constants. Value constants declared using #valueconst (Section 2.8).

Value variables. Value variables automatically declared when a domain is declared (Sections 2.1 and 2.2), and other value variables declared using #valuevar (Section 2.7).

Value expressions of the form value(τ, fe), where τ is a temporal term and fe is a feature expression (Section 3.3). Such expressions denote the value of the given feature expression at the given timepoint.

Conditional value expressions of the form $ite ($\phi$, ω, ω'), where ϕ is a logic formula and ω and ω' are value terms. The value of this value term is ω if ϕ is true, ω' otherwise. This is an experimental VITAL extension.

Note that VITAL does not support "true" functions that are independent of time. Such functions are always "emulated" using fluents, and therefore always need a temporal context. Therefore, even if a function such as plus(int,int) is defined using #function (Section 2.9), the expression plus(1,1) is only a feature expression and not a value term. It can be converted to a value term using the value() function, as in value(0,plus(1,1)).

3.3 Feature Expressions

Feature expressions are usually of either of the following forms:

- f, where f is a feature symbol of arity 0
- f(v_1, ... ,v_n), where f is a feature symbol of arity n and v_1, ... ,v_n are value terms of suitable sorts.

VITAL and TALplanner also allow the arguments of a feature to be pure feature expressions (as opposed to value terms). This is seen as a shorthand for a value() expression where the temporal term is borrowed from the temporal context of the outer feature expression. For example, assuming f, g and h are features of suitable sorts, value(τ, f(g(h))) is a shorthand for value(τ, f(value(τ, g(value(τ, h))))), and the formula [τ] f(g(h)) == v is a shorthand for [τ] f(value(τ, g(value(τ, h)))) == v.

Feature symbols can be declared using #feature (Section 2.3), #function (Section 2.9), or #relation (Section 2.10). However, VITAL can also generate certain arithmetic functions and ordering functions automatically (Section 5.2), along with definitions of those functions (semantic attachment).

See also the discussion of infix notation in Chapter 6.

3.4 Fluent Formulas

In the current version of the TAL tutorial and specification, fluent formulas no longer exist. However, I have currently ignored that, since they make it easier to explain the way formulas are built. This section may be rewritten in the future.

Fluent formulas are built from elementary fluent formulas using the common operators and quantifiers. Elementary fluent formulas are of either of the following forms:

- *fe* == ω, where *fe* is a feature expression and ω is a value term. Here, the double equals sign corresponds to the $\hat{=}$ "equals-with-a-hat" symbol in $\mathcal{L}$(ND).
- *fe*, where *fe* is a boolean feature expression. This is a shorthand notation which is also available in plain $\mathcal{L}$(ND), and is equivalent to *fe* == true.
- *fe* == { $\omega_1, \ldots, \omega_n$ }, where *fe* is a feature expression and $\omega_1 \ldots \omega_n$ are value terms, is equivalent to (*fe* == ω_1 | ... | *fe* == ω_n), where | is the text representation of the disjunction operator $\vee$.

[[+++ Is the last form a VITAL extension or not?]]

VITAL and TALplanner add a negated form of this operator:

- *fe* !== ω is equivalent to !(*fe* == ω), where ! is the text representation of the unary negation operator $\neg$.
- *fe* !== { $\omega_1, \ldots, \omega_n$ }, where *fe* is a feature expression and $\omega_1 \ldots \omega_n$ are value terms, is equivalent to !(*fe* == ω_1 | ... | *fe* == ω_n), where ! is the text representation of the unary negation operator $\neg$ and | is the text representation of the disjunction operator $\vee$.

Elementary fluent formulas can be combined in the ordinary manner using the boolean connectives and quantification over values. The following connectives are available (shown in order of precedence):

- `!` or ¬ (Unicode 0x00AC = 172) corresponds to logical negation (¬).
- `&` and `!&` correspond to logical conjunction (∧) and its negation (⊼), respectively.
- `|` and `!|` correspond to logical disjunction (∨) and its negation (⊽), respectively.
- `->`, `!->`, `<-` and `!<-` correspond to logical implication (→) and its negation (↛) and reverse implication (←) and its negation (↚), respectively.
- `<->` and `!<->` correspond to logical equivalence (≡) and its negation (≢), respectively.

Quantifiers can be used as follows. Note that VITAL and TALplanner allow a list of variables after the quantifier. The square brackets around the formula ϕ are required.

- Universal quantification over values: forall $v_1, \ldots, v_n$ [ϕ], where $v_1, \ldots, v_n$ are value variables and ϕ is a fluent formula.
- Existential quantification over values: exists $v_1, \ldots, v_n$ [ϕ], where $v_1, \ldots, v_n$ are value variables and ϕ is a fluent formula.

3.5 Logic Formulas

Logic formulas are built from atomic formulas using the common operators and quantifiers. The following are the atomic formulas available in VITAL:

- true and false. Note that these atomic formulas are *not* the same as the elements true and false in the boolean value domain.
- τ `=` τ', τ `!=` τ', τ `<` τ', and τ `<=` τ', where τ and τ' are temporal terms as defined in Section 3.1.
- ω `=` ω' and ω `!=` ω', where ω and ω' are value terms as defined in Section 3.2.
- Fixed fluent formulas of a number of different forms. For consistency with the Ct, R, I and X operators, VITAL allows these fixed fluent formulas to be enclosed in an H() operator, as in H([0] alive).

  The current parser requires that intervals where the lower bound is open be enclosed in the H() operator, due to technical difficulties in parsing intervals beginning with a left parenthesis.

  - [τ] ϕ, where τ is a temporal term and ϕ is a fluent formula as defined in Section 3.4.
  - [τ,τ'] ϕ, [τ,τ') ϕ, and [τ,$inf) ϕ, where τ and τ' are temporal terms and ϕ is a fluent formula.
  - (τ,τ'] ϕ, (τ,τ') ϕ, or (τ,$inf) ϕ, where τ and τ' are temporal terms and ϕ is a fluent formula. Note again that these formulas must be enclosed in the H() operator, as in H((0,5] f).

- R(ϕ), I(ϕ) and X(ϕ), where ϕ is a fixed fluent formula as defined above. The temporal interval specified in the formula must not contain the $maxocc term, since this would lead to a circular definition of $maxocc. Also, the temporal interval must not contain infinity ($inf), since this would lead to occlusion during an infinite interval, which VITAL currently cannot handle. Finally, note that certain feature expressions cannot occluded; this includes all features whose values are independent of time, such as pre-defined arithmetic functions and features declared in #function or #relation declarations.
- Ct([τ] ϕ, where τ is a temporal term and ϕ is a fluent formula.

Atomic formulas can be combined in the ordinary manner using the boolean connectives and quantification over values. The following connectives are available (shown in order of precedence):

- `!` or ¬ (Unicode 0x00AC = 172) corresponds to logical negation (¬).
- `&` and `!&` correspond to logical conjunction (∧) and its negation (⊼), respectively.
- `|` and `!|` correspond to logical disjunction (∨) and its negation (⊽), respectively.
- `->`, `!->`, `<-` and `!<-` correspond to logical implication (→) and its negation (↛) and reverse implication (←) and its negation (↚), respectively.
- `<->` and `!<->` correspond to logical equivalence (≡) and its negation (≢), respectively.

We also allow the use of a ternary conditional similar to the `?:` operator in C. Note that this conditional only operates on logic formulas, not on fluent formulas.

- $ifthenelse($\phi$, ϕ', ϕ'') is equivalent to $(\phi \rightarrow \phi') \wedge (\neg\phi \rightarrow \phi'')$.

Quantifiers can be used as follows. Note that VITAL and TALplanner allow a list of variables after the quantifier. The square brackets around the formula are required.

- Universal quantification over values and time: forall $v_1, \ldots, v_n$ [ϕ], where $v_1, \ldots, v_n$ are value variables or temporal variables and ϕ is a logic formula.
- Existential quantification over values and time: exists $v_1, \ldots, v_n$ [ϕ], where $v_1, \ldots, v_n$ are value variables or temporal variables and ϕ is a logic formula.

We also allow the use of an extension where a variable is immediately bound to a value term:

- $bind($v$, ω, ϕ) is equivalent to forall v [$v = \omega$ -> ϕ], with the exception that in a $bind expression, ω must be guaranteed to take on a value in the domain of v.

  This macro is especially useful in TALplanner, since the only way to relate values of ordinary fluents to values in a modal goal context is to "quantify into" the modal context.

Chapter 4

Logical Statements

After the declarations, VITAL and TALplanner expect a sequence of labeled logical statements in $\mathcal{L}$(ND) or extended variations of $\mathcal{L}$(ND). This chapter will describe all of those statement classes.

[[+++ References to where these are defined]]

4.1 Observations and Other Constraints

There are three different statement classes that relate to plain logical constraints on the world, either as observations at single timepoints (#obs, #init) or as domain constraints that must hold in all states (#dom).

Unfortunately, some statements intuitively fall outside these classes, and TAL currently lacks a generic statement class to be used for those statements. For such statements, you should currently use domain constraints.

These statement classes must not contain R, I, or X operators.

4.1.1 Observation Statements

Observation statements, labeled #obs, are intended to correspond to actual observations of values in the world. Therefore, in addition to the restriction that they not contain occlusion expressions (occurrences of R, I, or X operators), they must also satisfy certain additional constraints. To be exact, they must be of one of the following forms:

- $[\tau]$ ϕ – a fixed fluent formula [[current terminology?]] with a singleton interval, where τ is a numeric temporal constant. The fluent formula ϕ must not use features whose meaning is defined using semantic attachment (arithmetic operators, for example). Also, feature arguments in ϕ must be values, value constants, or value variables (not value() expressions), and the == operator must always be followed by a single value, value constant, or value variable.

- ω=ω', where ω and ω' are value terms.
- ω!=ω', where ω and ω' are value terms.
- τ=τ', where τ and τ' are temporal terms.
- τ<τ', where τ and τ' are temporal terms.
- τ<=τ', where τ and τ' are temporal terms.

4.1.2 Domain Constraint Statements

Domain constraint statements, labeled #dom, are intended to correspond to logical constraints that must hold at every point in time, or across point in time. Currently, a domain constraint can be any logical formula as defined in Section +++. However, it cannot contain occlusion expressions (occurrences of R, I, or X operators).

The syntax of a domain constraint statement is as follows:

#dom *:options* ϕ

:last Process this statement after all observation statements, initialization statements, and domain constraint statements not marked with :last. This is a hack implemented in order to support certain features of PDDL in TALplanner, and will most likely be removed once proper support for these features has been implemented. Do not use :last.

VITAL and TALplanner also accept the old label #acc (acausal constraint) for dependency constraints. This label is deprecated, though.

4.1.3 Initialization Statements

Initialization statements, labeled #init, are an experimental extension. Their main use is to provide a more succinct syntax for initializing features at time 0. Initialization statements were introduced for use by TALplanner, but can also be used in VITAL.

The syntax of an initializion statement is as follows:

#init *feature :options*

Here, the *feature* is the name of an ordinary feature declared using #feature. The feature must not be declared :defined or :valuetype. The following options are available:

:at τ The feature is initialized (constrained) at time τ. The default timepoint is 0.

:default ω The feature is initialized to ω (a value term) for all instances that are not listed in :constrain.

:constrain

:predicate-value :for

[[+++ Update this for the new syntax]]

4.2 Action- and Change-related Statements

There are also a number of statement classes related to actions and change. Dependency constraints specify causal relationships in a dynamic world, where certain conditions trigger changes. Action occurrence statements specify which actions occur when, while action law schemas provide the definitions of the actions that have been declared (Section 2.4). These three classes are currently only used in VITAL, while TALplanner declares and defines operators in a single step using the extended operator statements from $\mathcal{L}(\mathrm{ND})^*$.

4.2.1 Dependency Constraint Statements

Dependency constraint statements (labeled #dep) specify causal relationships in a dynamic world, where certain conditions trigger changes. See the TAL tutorial [[+++ref]] for a definition and an explanation of the restrictions placed on dependency constraints.

The syntax of a dependency constraint statement is as follows:

#dep ϕ

Above, ϕ is an arbitrary logic formula, which can contain occurrences of R, I, and X operators (subject to the restrictions defined in the TAL tutorial).

4.2.2 Action Occurrence Statements

Action occurrence statements (labeled #occ) specify exactly which actions occur, and when they occur.

The syntax of an action occurrence statement is as follows, depending on whether the action type Act has arguments:

#occ [τ, τ'] Act
#occ [τ, τ'] Act(ω_1, ... ,ω_n)

Here, τ and τ' must be temporal terms with no free variables, and ω_1 through ω_n must be value terms with no free variables.

[[+++ Rewrite this explanation]]

4.2.3 Action Law Schemas

Action law schemas (labeled #acs) provide the definitions of the actions that have been declared using #action declarations (Section 2.4).

The syntax of an action law schema is as follows, depending on whether the action type Act has arguments:

#acs [τ, τ'] Act ~> ϕ
#acs [τ, τ'] Act(ω_1, ... ,ω_n) ~> ϕ

Here, τ and τ' must be temporal variables, and ω_1 through ω_n must be value variables. The formula ϕ must have no other free variables than $\tau, \tau', \omega_1, \ldots, \omega_n$.

The special arrow ~> (tilde / greater than) can also be replaced with a plain implication arrow ->.

4.3 Miscellaneous Extensions

The following statements are experimental extensions implemented by VITAL. They are not part of plain $\mathcal{L}$(ND) or of any published extension.

4.3.1 Feature Definition Statements

VITAL allows the use of *defined features*, boolean features defined in terms of logical formulas, where the feature takes on the boolean value true iff the logic formula holds. Defined features are evaluated by evaluating their definitions, which may of course lead to recursively evaluating the feature again, hopefully with different arguments. Note that VITAL currently has no protection against infinite recursion.

Defined features are declared using a #feature statement where the :defined option is specified. Two more options, :uncached and :expand, control certain aspects of the defined feature. See Section 2.3 for further information about these three options.

When a defined feature has been declared, the user must provide a definition for the feature using a #define statement. The syntax of this statement is as follows:

#define [τ] f(v_1, ... ,v_n) : ϕ

Here, τ and v_1, ... ,v_n are the formal parameters to the defined feature: τ must be a temporal variable, and v_1, ... ,v_n must be value variables of sorts suitable as arguments to the feature f. Whenever the defined feature is evaluated, τ will be bound to the evaluation timepoint and v_1, ... ,v_n to the actual arguments. Then, the logic formula ϕ will be evaluated. If it is true, the boolean value true is returned; otherwise, the boolean value false is returned.

For example:

```
#feature alive(person) :domain boolean
#feature dead(person) :domain boolean :defined

#define [t] dead(person): [t] !alive(person)
#obs [0] dead(alice) & alive(bob)
#dep forall person [ Ct([t] dead(person)) -¿ I([t] doSomething) ]
```

4.3.2 Object Declaration Statements

Object declaration statements (labeled #objects) can be used for adding values to a value domain after its declaration. This can be used in order to modularize definitions of value domains.

An object declaration statement has the following syntax:

#objects :domain *dom* :elements {*val1*, *val2*, ... , *valn*}

If the domain dom is a top-level domain, this statement adds the new values {*val1*, *val2*, ... , *valn*} to that domain. This is subject to all ordinary consistency checks; for example, the values must not already exist in any other domain.

If the domain dom is a child domain, this statement adds the values {*val1*, *val2*, ... , *valn*} to that domain. However, the values are not created; they must already be present in the parent domain. If they are not, you need to use additional #objects statements to add the values to all parent domains before adding them to the child domain.

For example:

```
#domain colors :elements { red, green, blue, cyan, magenta, yellow }
...
#objects colors :elements { black, white }
```

4.3.3 Assertion Statements

Assertion statements, labeled #assert, provide VITAL or TALplanner with information that the user asserts to hold in any preferred model of the narrative. Although VITAL currently ignores this information, TALplanner can sometimes use it to infer certain information in order to simplify action preconditions or control rules.

It is possible to assert the truth of any logic formula ϕ using the following syntax:

#assert ϕ

Note that this does *not* act as a constraint on the narrative; the formula will *not* affect the set of models generated by VITAL or the plan generated by TALplanner (unless, of course, the assertion is false and causes VITAL or TALplanner to perform optimizations that should not have been allowed).

The following are some simple examples from the blocks world:

```
#assert forall t, block [ [t] ontable(block) -> !holding(block) ]
#assert forall t, block1, block2 [ [t] on(block1, block2) -> !clear(block2) ]
#assert forall t, block1, block2 [ [t] on(block1, block2) -> !ontable(block1) ]
#assert forall t, block1, block2 [ [t] on(block1, block2) -> !holding(block1) ]
#assert forall t, block1, block2 [ [t] on(block1, block2) -> !holding(block2) ]
#assert forall t, block1, block2 [ [t] holding(block2) -> !clear(block2) ]
#assert forall t, block1, block2 [ [t] holding(block2) -> !handempty ]
```

TALplanner currently prefers assertion statements on this simple form (a universal quantifier prefix followed by an implication or disjunction).

4.4 Planning-related Statements

For completeness, we will now briefly explain a number of planning-related statement classes that are supported by the parser. These statement classes have no effect on VITAL, and should only be used in plan narratives. They are explained in more detail in the TALplanner documentation.

The planning-related statement classes are the following:

- Intended goal statements, labeled #goal, are used to specify the goal that the planner should attempt to reach.
- Non-modal control statements, labeled #control, and modal control statements, labeled #modal, help the planner to prune the search tree.
- Redundancy control statements, labeled #redundancy, provide stronger pruning cababilities by allowing pruning rules to consider multiple plan prefixes simultaneously.
- Operator definition statements, labeled #operator, replace action declarations (#action) and action law schemas (#acs) and provide a more succinct way of defining plan operators. They also facilitate the specification of resource usage.
- Heuristic specifications, labeled #heuristic, specify functions to be used in heuristic search.

Chapter 5

Numbers and Arithmetics

Many narratives and planning domains require the use of numbers and arithmetics. Although any finite value domain can be specified specified by enumerating its elements in a #domain declaration, this can quickly become tedious for larger value domains. Defining operations on such domains by enumerating their definitions in #relation or #function declarations is even more difficult, due to the need to list n^2 function values for every binary operator.

Consequently, VITAL/TALplanner provides a way to automatically generate numeric value domains and certain operations on these domains. This will be explained in the following sections.

5.1 Fixed Point and Integer Value Domains

VITAL allows you to declare integer and fixed point value domains (that is, value domains with a fixed number of decimals, as opposed to floating point value domains whose exact semantics would be more difficult to define). These domains are declared using the ordinary #domain statement, but instead of explicitly specifying a set of elements, you specify upper and lower bounds for the value domain together with the desired number of decimals.

A fixed-point value domain is declared as follows:

#domain *name* [:no-vars] :integer :prefix *pre* :lb *lower* :ub *upper* :decimals *dec*

An integer value domain is declared as follows (or by using :decimals 0 above):

#domain *name* [:no-vars] :integer :prefix *pre* :lb *lower* :ub *upper*

The *name* is the name of the value domain that will be generated. Most options are mandatory, and options must be given in the order defined above.

:fixedpoint Define a fixed point domain.

:integer Define an integer domain. This is the same as defining a fixed point domain with :decimals 0.

:lb n The lower bound of the value domain. This integer will be included in the domain.

:ub n The upper bound of the value domain. This integer will NOT be included in the domain.

:decimals n The number of decimals for the numbers in this domain.

:prefix str All values in this domain will have names beginning with *str*. If no prefix is given, the values will be pure numbers such as 7 or 412.88.

A few examples:

- #domain integer :fixedpoint :prefix int :lb 0 :ub 10 :decimals 0

  Generate a value domain integer with the values int0, int1, ... , int9.

- #domain integer :integer :prefix int :lb 0 :ub 10

  Generate a value domain integer with the values int0, int1, ... , int9.

- #domain length :fixedpoint :prefix fp :lb 0 :ub 100 :decimals 2

  Generate a value domain length with the values fp0.00, fp0.01, ... , fp99.98, fp99.99.

- #domain length :fixedpoint :lb 0 :ub 100 :decimals 2

  Generate a value domain length with the values 0.00, 0.01, ... , 99.98, 99.99.

As in ordinary domain declarations (Section 2.2), declaring a fixed point or integer domain called *name* automatically declares six value variables of that sort, called *name*, *name1*, *name2*, *name3*, *name4*, and *name5*, except for those names that are already used by declared variables, values, or other entities.

It is possible to generate more than one fixed point or integer domain by using multiple #attach statements. Note, however, that prefixes must be unique, and that at most one fixed point value domain can have an empty (unspecified) prefix – otherwise, VITAL/TALplanner cannot determine which value domain a given literal belongs to.

5.2 Arithmetic Operators

VITAL/TALplanner can automatically generate a large number of arithmetic operators on any numeric value domain (that is, any value domain generated by #domain :integer or #domain :fixedpoint).

Such operators are represented as ordinary features whose values for any given arguments are defined using semantic attachment. Since they are features, they can be used more or less anywhere ordinary features can be used – which means, among other things, that they must be placed in a temporal context, even if they happen to be independent of time. That is, the following observation statement is not allowed:

#obs value(0,level) = $times(5, 5)

The $times feature needs a temporal context (given by an enclosing $[\tau]$ or value() construction), even though its value is independent of that context:

#obs value(0,level) = value(0, $times(5, 5))

Or, equivalently:

#obs [0] level == $times(5, 5)

Since these operators work on finite domains, we must define what happens when there is overflow or underflow, and define rounding characteristics for operations such as division. One possibility would be to "wrap around", using arithmethics modulo n (where $times(5, 5) = 3 if the value domain is limited to $[0, 22)$). However, in most applications we have explored, another semantics has appeared more useful, where $times(5, 5) = 22 if the value domain is limited to $[0, 22)$.

[[+++ What should this be called, formally?]]

Below, we will define the exact semantics of all automatically generated operators. First, given a value v, let $num(v)$ be its numerical value (a real), and given a real number r, let $val(r)$ be the corresponding value. Let *minval* be the minimum value in the value domain, and let *maxval* be the maximum value in the value domain. Given a real number r, let $adjust(r)$ be $\min(num(maxval), \max(num(minval), r))$. Given a real number r, let $wrap(r)$ be $num(minval)$ if $r < num(minval)$, $num(maxval)$ if $r > num(maxval)$, or r otherwise.

The following arithmetic operators are available:

$plus $\$plus(v_1, v_2) = val(adjust(num(v_1) + num(v_2)))$

$minus $\$minus(v_1, v_2) = val(adjust(num(v_1) - num(v_2)))$

$times $\$times(v_1, v_2) = val(adjust(num(v_1) * num(v_2)))$

$div $\$div(v_1, v_2) = val(adjust(num(v_1)/num(v_2)))$

$sqrt $\$sqrt(v) = val(\lfloor\sqrt{num(v)} + 0.5\rfloor)$, where the $\sqrt{}$ operator follows IEEE double precision floating point rounding rules (+++ CHECK/FORMALIZE).

$abs $\$abs(v) = val(|num(v)|)$. Note that this feature can only be generated for value domains where $-num(minval) \leq num(maxval)$.

$next $\$next(v) = val(wrap(num(v) + 1))$.

$prev $\$prev(v) = val(wrap(num(v) - 1))$.

$min $\$min(v_1, v_2) = val(\min(num(v_1), num(v_2))$.

$max $\$max(v_1, v_2) = val(\max(num(v_1), num(v_2))$.

There are also a number of relational operators. These operators take two arguments and return true or false (that is, a value in the boolean domain). The following relational operators are available:

$less $\$\mathsf{less}(v_1, v_2) = \begin{cases} \mathsf{true} & \text{if } num(v_1) < num(v_2) \\ \mathsf{false} & \text{otherwise} \end{cases}$

$lessequal $\$\mathsf{lessequal}(v_1, v_2) = \begin{cases} \mathsf{true} & \text{if } num(v_1) \leq num(v_2) \\ \mathsf{false} & \text{otherwise} \end{cases}$

$equals $\$\mathsf{equals}(v_1, v_2) = \begin{cases} \mathsf{true} & \text{if } num(v_1) = num(v_2) \\ \mathsf{false} & \text{otherwise} \end{cases}$

$greaterequal $\$\mathsf{greaterequal}(v_1, v_2) = \begin{cases} \mathsf{true} & \text{if } num(v_1) \geq num(v_2) \\ \mathsf{false} & \text{otherwise} \end{cases}$

$greater $\$\mathsf{greater}(v_1, v_2) = \begin{cases} \mathsf{true} & \text{if } num(v_1) > num(v_2) \\ \mathsf{false} & \text{otherwise} \end{cases}$

VITAL/TALplanner also provides a number of automatically generated pseudo-features that perform operations over all instantiations of a set of variables. For example, the $sum pseudo-feature corresponds to the $\sum$ operator.

We call these operators *pseudo-features* since they act like features except that their arguments are of a different form and type. The generic form of such a feature expression is as follows:

name(<$v_1, \ldots, v_n$>, ψ, ω)

Above, $v_1, \ldots, v_n$ are value variables ($n \geq 1$), ψ is a logic formula acting as a condition on which instantiations of the value variables should be considered, and ω is a value term.

The pseudo-features can be informally described as follows:

$$\$\mathsf{sum}(<v_1, \ldots, v_n>, \psi, \omega) = val(adjust(\textstyle\sum_{v_1 \in dom(v_1)} \cdots \sum_{v_n \in dom(v_n)} \begin{cases} 0 & \text{if } \psi \text{ is false} \\ num(\omega) & \text{if } \psi \text{ is true} \end{cases}))$$

$$\$\mathsf{mmax}(<v_1, \ldots, v_n>, \psi, \omega) = val(adjust(\max_{v_1 \in dom(v_1)} \cdots \max_{v_n \in dom(v_n)} \begin{cases} -\infty & \text{if } \psi \text{ is false} \\ num(\omega) & \text{if } \psi \text{ is true} \end{cases}))$$

$$\$\mathsf{mmin}(<v_1, \ldots, v_n>, \psi, \omega) = val(adjust(\min_{v_1 \in dom(v_1)} \cdots \min_{v_n \in dom(v_n)} \begin{cases} \infty & \text{if } \psi \text{ is false} \\ num(\omega) & \text{if } \psi \text{ is true} \end{cases}))$$

5.2.1 Arithmetic Operators for Arbitrary Domains

The arithmetic operators supported by VITAL and TALplanner can also be used for ordinary value domains, regardless of their contents. In this case, the values in the domain are mapped to a finite subset of the non-negative integers, starting with 0. For example, suppose you define a domain of colors as follows:

#domain colors :elements { red, green, blue, cyan, magenta, yellow }

Then, red is mapped to 0, green is mapped to 1, and so on.

All arithmetic operations are performed on these integers. This implies that the following statements will hold:

#obs [0] $plus(red, red) == red // True; 0 + 0 == 0
#obs [0] $times(blue, blue) == magenta // True; 2 * 2 == 4
#obs [0] $less(green, blue) // True; 1 < 2
#obs [0] $sqrt(yellow) == blue // True; sqrt(5) rounded to an integer is 2

WARNING: If you use subdomains, you should be aware that the argument type of the generated arithmetic feature will be *the topmost common parent* of the two actual arguments.
Suppose you have the following domains:
#domain int :elements { i0, i1, i2, i3, i4, i5, i6, i7, i8, i9 }
#domain small :parent int :elements { i0, i1, i2, i3, i4 }
#domain large :parent int :elements { i5, i6, i7, i8, i9 }
Then, in the following statements, all generated arithmetic features will take ints as argument, and will return int.
#obs i3 + i3 == i6
#dom exists small [[0] small + small == i6] #obs exists small [$next(small) == i5]
In the first two statements, the + operator is an addition operator over ints, rather that an addition operator over smalls. Thus, i3 + i3 is equal to i6, rather than i4.
In the third statement, the $next operator is an increment operator over ints, rather than an increment operator over smalls. Thus, $next(i4) is equal to i5, rather than i0.
If you for some reason need an operator where the argument sort is not a top-level sort, you must declare this operator manually (Section 5.2.2). This may be the case if you need a $next operator that only operates on the members of a subsort, for example.

5.2.2 Declaring Operators Manually

Instead of using automatically generated features, you can also specify exactly which operators you want to generate. This may be useful if you need an operator that operates on a subsort rather than a top-level sort, as explained in the warning box above, or if you want to define your own names for an operator rather than using a built-in name.

Operators can be declared using a #attach statement, similar to the one used for declaring a fixed point or integer value domain. The syntax is as follows:

#attach DomainOrder "*dom min minus prev less lesseq equal greatereq greater next plus max sqrt mult div sum abs*"

Here, *dom* is the name of a domain, while *min* through *sum* are the names you wish to give the corresponding operators on the domain (or a hyphen to indicate that the corresponding feature should not be generated). For example:

#attach DomainOrder "location minloc minusloc previousloc lessloc lessequalloc equalloc greaterequalloc greaterloc nextloc plusloc maxloc - - - sumloc -"

This syntax may be extended in the future, since it is easy to make mistakes in the argument order for the current syntax.

Chapter 6

Infix Notation

As an extension to the plain $\mathcal{L}$(ND) syntax, VITAL and TALplanner also allow infix notation to be used for any feature taking two arguments.

Currently, it is not possible to define precedence rules for features. Instead, certain built-in features (Section 5.2) have predefined precedence levels, while all other features share the highest possible predecence (that is, they bind harder). Also, all operators are left associative. Use parentheses where necessary.

In order to improve clarity and make narratives more succinct, many of the arithmetic and relational operators can also be written using common symbols (+, −, <=, and so on) when using infix notation.

6.1 Precedence Rules

The precedence levels are as follows, from highest to lowest precedence:

- Any binary feature not specified below
- * ($times), / ($div), % ($mod)
- + ($plus), - ($minus)
- < ($less), <= ($lessequal), >= ($greaterequal), > ($greater), $equals

6.2 A Note on Equality

Equality exists in plain $\mathcal{L}$(ND), and is a logical operator rather than a feature with a semantic attachment. Therefore, $equals does not have a corresponding infix operator =. This is also the reason why $equals *requires* a temporal context provided by $[\tau]$ or value(), while = *cannot* be used within a temporal context.

6.3 Examples

The following examples should help you understand the precedence and associativity rules used by VITAL and TALplanner.

```
#domain colors :elements { red, green, blue, cyan, magenta, yellow }
#attach FixedPoint "integer - 0 1000 0"
#feature foo(integer, integer) :domain integer

#obs [0] $less(12, 42) // true
#obs [0] 12 < 42 // equivalent
#obs [7] 12 < 42 // equivalent, temporal context is ignored but required
#obs 12 < 42 // Wrong, no temporal context

#obs 12 = 12 // true
#obs [0] 12 = 12 // Wrong, no equality within temporal context

#obs [0] 12 + 42 * 3 == 138 // True; note the use of '=='
#obs [0] 12 $plus 42 $times 3 == 138 // True
#obs [0] $plus(12, $times(42, 3)) == 138 // True

#obs [0] foo(12+7, 12*3/2) == 5 // True or false...
#obs [0] (12+7) foo (12*3/2) == 5 // Equivalent
#obs [0] 12+7 foo 12*3/2 == 5 // Not equivalent, foo binds harder

#obs [0] red + red == red // True; 0 + 0 == 0
#obs [0] blue * blue == magenta // True; 2 * 2 == 4
#obs [0] green < blue // True; 1 < 2
```

Chapter 7

Controlling Constraint Propagation

The #propagation statement can be used for controlling the order in which constraints are applied to a partial interpretationThis only affects VITAL; TALplanner is unaffected. The syntax is as follows:

#propagation *:options*

The following options are available:

:constraints-by-time In a narrative with multiple domain constraints $d_1, \ldots, d_n$ universally quantified over time, VITAL normally applies d_1 at all timepoints, then d_2 at all timepoints and so on.

With :constraints-by-time, VITAL first applies d_1 at time 0, d_2 at time 0, ... , d_n at time 0. This is repeated as long as changes are made to the partial interpretation. When a fixpoint is found, VITAL starts applying constraints at time 1, and so on.

Using :constraints-by-time can speed up constraint propagation considerably in some cases, such as when several domain constraints together propagate information forward in time. An example of this would be the following:

```
#obs [0] f & g
#dom forall t [ [t] f & g -> [t+1] f ]
#dom forall t [ [t] f & g -> [t+1] g ]
```

Chapter 8

Controlling Splitting

Internally, VITAL uses constraint propagation to find all possible models for a given narrative. The algorithms start with a partial interpretation where all features and constants are completely unknown (can take on arbitrary values), and then performs constraint propagation on this structure using the logical formulas as constraints.

In many cases, however, the complete set of models for the narrative cannot be represented as a single partial interpretation. One example of this would be a narrative with a constraint such as $[0]f \oplus g$. For such narratives, VITAL must *split* the original partial interpretation into a number of children in order to reason by cases. In this example, VITAL might split on f, generating one child where f is definitely true (eventually resuming constraint propagation and concluding that g must be false), and one child where f is definitely false (eventually concluding that g must be true).

Splitting may also be necessary when the constraint propagation methods implemented in VITAL are not strong enough: Instead of immediately constraining a partial interpretation, VITAL splits it into a number of children in order to reason by cases. Eventually, it may conclude that certain cases were in fact not possible, in which case those partial interpretations are discarded.

The performance of VITAL can be affected significantly by exactly which splitpoints are chosen. Above, for example, VITAL might choose to split either on f or on g. Therefore, before splitting, VITAL considers all possible splitpoints in the partial interpretation, assigns each splitpoint a goodness value, and chooses a splitpoint with maximum goodness. The goodness of a splitpoint is a non-positive value; the best possible splitpoint has a goodness of 0.

Since it is difficult to find a heuristic for assigning goodness values that works well for all narratives, the user can affect certain aspects of this process. The next section lists the different kinds of splitpoint currently considered by VITAL and shows how their goodness values are calculated by default, while Section 8 shows how to tell VITAL whether splitpoints at earlier timepoints should always be preferred more than those at later timepoints.

8.1 Splitpoints and Priorities

Given a partial interpretation and a set of formulas that should hold in that interpretation, there are a number of different kinds of splitpoints.

Each splitpoint is assigned a goodness value, a non-positive integer. For splitpoints associated with the value of a fluent at a given timepoint, this value can be affected by the *split priority* of that feature (denoted by splitpri(feature) below). This priority is 1024 by default, but can be changed using the :splitpri option of the corresponding #feature declaration (Section 2.3).

- Suppose a formula depends on a temporal constant, whose value is unknown (that is, which is not assigned a single possible value by the partial interpretation). Then, the partial interpretation can be split into one child for each value the constant can currently take on. The goodness of this splitpoint is -2048.
- Suppose a formula depends on a value constant, whose value is unknown. Then, the partial interpretation can be split into one child for each value the constant can currently take on. The goodness of this splitpoint is $-$(the number of children) $\times$ 1024.
- Suppose a formula depends on an expression $[\tau]\ f(\overline{\omega}) \hat{=} \{v_1, \ldots, v_n\}$ whose value is unknown, and that VITAL has already determined the values of the temporal term τ, the feature arguments $\overline{\omega}$ and all the value terms in the set $\{v_1, \ldots, v_n\}$. Then, then the partial interpretation can be split into one child where the comparison holds and one where it does not hold. The goodness of this splitpoint is $-2 \times$ splitpri(feature).
- Suppose a formula depends on an expression $value(\tau, f(\overline{\omega})$ whose value is unknown, and that VITAL has already determined the values of the temporal term τ and the feature arguments $\overline{\omega}$. Then, the partial interpretation can be split into one child for each possible value that f can currently take on at τ. The goodness of this splitpoint is $-$(the number of children) $\times$ splitpri(feature).

  Feature expressions occurring as arguments in other feature expressions, or occurring after $\hat{=}$, are implicitly considered to be embedded in *value*().
- Suppose a formula depends on a temporal comparison $\tau = \tau'$, $\tau < \tau'$ or $\tau \leq \tau'$ whose value is unknown, but that one of the temporal terms is known and the other is a temporal constant. Then, the partial interpretation can be split into two children, where the comparison holds or does not hold, respectively. The goodness of this splitpoint is -2048.

8.2 Controlling Splitpoint Selection

As discussed above, splitpoint selection can be affected by altering the split priorities of the features occurring in a narrative. Certain other aspects of this procedure can be controlled by the #splitting statement. This only affects VITAL; TALplanner is unaffected. The syntax is as follows:

#splitting *:options*

The following options are available:

:split-by-time Usually, splitpoint priorities are calculated as explained in Section 2.3 (see :splitpri). For some narratives, however, it makes more sense to always choose a splitpoint as early in time as possible, even if there is a splitpoint later in time that would result in fewer children. Using :split-by-time forces VITAL to prioritize earlier splitpoints above later ones.

:breadth When VITAL splits a partial interpretation into a number of children, some of those child interpretions may contradict the remaining set of formulas. Although VITAL will always discover this and discard those interpretations, this may take a long time, requiring a large number of additional splits.

VITAL normally uses depth-first search during splitting and constraint propagation. In other words, when a partial interpretation is split, all its children are created and placed on a stack, and constraint propagation is restarted for the topmost child. When no more constraints can be applied but formulas still remain, the child is split into grandchildren, and the constraint propagation begins for the topmost grandchild, and so on.

In most cases, this works well. For some narratives, however, almost all children are contradictory and discovering this is not very difficult or time consuming. For such narratives, breadth-first search may be better, in the sense that it discards contradictory children more quickly and keeps fewer interpretations in memory. Use the :breadth option to force VITAL to use breadth-first search.

Chapter 9

The t_* (maxocc) Constant

Although each TAL fluent has an infinite timeline, VITAL only considers a finite part of that timeline. We assume the existence of a timepoint t_* such that in all preferred models, all fluents are unoccluded after t_*. This, of course, limits the set of TAL narratives for which VITAL is applicable.

The condition that no fluent is occluded in $[t_* + 1, \infty)$ implies that each fluent has a constant value in the interval $[t_* + 1, \infty)$: Persistent fluents and semidynamic fluents (Section 2.3, option :semidynamic) retain their values from t_*, and durational fluents take on their default values.

9.1 Calculating a Value for t_*

Determining whether a t_* exists, and determining a suitable value, is not trivial in the general case.

VITAL currently makes the following assumptions:

- No action has direct effects after the end of its invocation interval. That is, an action law schema $[t, t']$ $\mathbf{A} \rightsquigarrow \Psi$ cannot imply occlusion after t'.
- No dependency constraint can be triggered at a timepoint t without there being a change in the value in some fluent at t. This was always true for old-style causal constraints, but is not necessarily true for the new dependency constraints.
- All dependency constraints (not only the non-delayed dependency constraints) are stratified, so that no dependency constraint can cause a chain of changes that eventually triggers the same dependency constraint.

[[+++ Describe the assumptions more formally]]

Given that these assumptions hold, VITAL can calculate an overestimate of t_*. [[+++ Describe how]] If this value turns out to be greater than necessary, VITAL may take longer than necessary to calculate the set of preferred models for your narrative. If the value turns out to be too small, VITAL will eventually complain about conditions implying occlusion after t_*.

9.2 Using #maxocc and $maxocc

If you are not satisfied with the value VITAL calculates for t_*, you can specify a value using a #maxocc declaration (Section 2.11).

It is also possible to refer to the value VITAL uses for t_* using the temporal constant $maxocc. This makes it possible to conditionalize formulas such as dependency constraints on the value of t_*.

This may be especially useful when you want to simulate some process where dependency constraints can trigger each other in infinite chains. Although VITAL cannot handle infinite chains, you can refer to the value of t_* in domain constraints and dependency constraints using the $maxocc constant and use this constant to ensure that no effects are triggered after t_*. You can then "simulate" the process over arbitrary finite prefixes of the timeline by specifying different values of t_* using a #maxocc declaration.

For example, [[+++ more explanations and an example]]

Chapter 10

Parameterized Narratives

Not yet written

Bibliography

Index