ViewSys

David S. Warren

A View Maintenance System Library

Table of Contents

Summary

The ViewSys package supports workflows that can be structured as a DAG of view definitions. ViewSys allows a user to define such workflows and manage their execution and maintenance. At this point it does not provide support for incremental view maintenance, only full regeneration of views.

view_sys

Overview

A View System supports a DAG of views. Most simply a view is a file that is generated by a process applied to a set of input files. A view that has no inputs is called a 'base view'.

More precisely we can think of a view as a data source. Base views are data sources from outside the system. A non-base view is a data source that is determined (and computed) by its process applied to its input data sources. The process must be idempotent, so normally it creates a new file (or table).

A view system workflow (**ViewSys** for short) describes the names of the views, their input views, the command to be run to generate a view from its inputs, etc. A particular **instance** of a ViewSys is determined by the specific external data sources associated with the base views of the ViewSys. It is useful to give names to such an instance, usually indicating the external source of the base data sources. Also since external data sources may change over time, producing new versions of those sources, ViewSys is structured to support versions of all views (but this is not currently implemented; every version is version 0.) Many view systems will have only one instance, and each instance only one version.[1]

Another useful component of a view system is what is called a consistency view. The purpose of a consistency view is to check to see whether a regular view is 'consistent'. The command for a consistency view should return non-zero if the view instance is not deemed to be consistent.

The view system will run consistency views where applicable and will not use a view as input to another view that it supports if it is deemed not consistemt. A single view may have zero or more consistency views associated with it.

The ViewSys Data Model

A ViewSys workflow is described by a set of facts of the following predicates. Users should put the appropriate facts for these predicates that define their view system into a file named `viewsys_view_info.P`. (But see the split(N) option described below for an exception.)

View Framework Model

For each view (base or derived), there is a view/7 fact that describes it:

view(View,Type,ViewNameTemplate,StdoutFileTemplate,[InputViews],[Opts],ShCmd)
where:

- `View` is the name of the view;
- `Type` is file, dir(<FileNames>) (or maybe in the future db(...)?). It is 'file' if the view is stored in a file (that is generated by the `ShCmd`). It is 'dir(<FileNames>)' if the view is stored in multiple files in a directory. <FileNames> are the (relative) names of the files that store the view in that directory (instance).

[1] The contents of a version can change, and the system will manage such cases. This is the reason versioning has not (yet?) been implemented; adding explicit version numbers may be over-engineering and it would better for the user to manage backing up of earlier versions outside the ViewSys system.

- `ViewNameTemplate` is the path template for where instance versions are stored. This template string normally contains the pattern variable $INSTANCE$ which will be replaced by the instance name to obtain the filename (or directory name) of an instance of this view. A file template may also contain user-defined pattern variables of the form '$USERVARNAME$' where USERVARNAME is any upper-case letter sequence. User-defined pattern variable values are defined in facts of the form viewsys_uservar('$USERVARNAME$',VarValueString). When instantiated by an instance name and user-variable values, this will be the filename that contains the view contents, or a directory name that contains the files containing the view contents.
- `StdoutFileTemplate` is the file name template for where to put the standard output (stdout and stderr) of the execution of the shell command that generates instance versions of this view. It normally contains the pattern variable $INSTANCE$ which will be replaced by the instance name to obtain the stdout filename for the generation of an instance of this view. This template is the empty string for base views.
- `[InputViews]` is a list of the names of views that this view directly depends on, i.e., the inputs needed to generate this view. This is an empty list for base views. Normally these input view indicators are atoms for which there is another `view/7` fact that describes it. However, if that view generates a directory and the input to this view is a file in that directory, then that filename should be put as an argument to the view atom. E.g., if the view, m_view, generates a directory and several files in it and this view needs to use the file 'first_file.P' from that directory, then the input view indicator in this list should be the term m_view('first_file.P').
- `[Opts]` is a list of options. The possible options are:
  - `split(N)` where `N` is a positive integer. This tells `viewsys` to split the first input view file into `N` subfiles; to run this command on each of those subfiles; and to concatenate all the resulting subfiles back together to get the output file for this view. Of course, this is only appropriate for view commands for which this process gives the same answer as running it on the large unsplit file. When the command satisfies this property, this option can allow the records in a large file to be processed in parallel.

    If this option is used, then the user should create a file named `viewsys_view_orig_info.P` containing all these predicate definitions, and use `expand_views/1` to generate the appropriate `viewsys_view_info.P` file, which will drive the `viewsys` processing.
- `ShCmd` is the shell command to execute to generate the view instance from its input view instances. (Ignored for base views.) The shell command can be in one of three forms: 1) a string containing metavariables of the form $INP1$, $INP2$, ..., and OUT, which will be replaced by the filenames of the input view instance files/directories and the output view instance file/directory, respectively; or 2) a string containing the metavariables $INPUTFILES$ and $OUTPUTFILE$, which will be replaces with the sequence of input filenames and the output filename, respectively, where each filename is enclosed in double-quotes. This is often appropreate for shell commands. If the shell string doesn't contain any of the metavariables, then it is treated as if it were: '<ShCmd> $INPUTFILES$ $OUTPUTFILE$'.

User-defined syntactic variables can be used in filename templates and in shell command templates to make it easier to define filenames and commands. The predicate `viewsys_uservar/2` is used to define user variables, and facts for this predicate should be places in the `viewsys_view_info.P` file. For example, assume the user adds the following facts to that file:

```
viewsys_uservar('$DATA_DIR$','C:/userfiles/project1/data').
viewsys_uservar('$SCRIPT_LIB$','c:/userfiles/project1/scripts').
```

With these declarations in a `viewsys_view_info.P` file, a file templiate string could be of the form '$DATA_DIR$/data_file_13', which after replacement of the syntactic variable by its value would refer to the file named 'C:/userfiles/project1/data/data_file_13'. A shell command string could be 'sh $SCRIPT_LIB$/script_cc.sh', which after replacements would casue the command 'sh c:/userfiles/project1/scripts/script_cc.sh' to be run. User variables are normally defind at the beginning of the view file and can be used to allow locations to be easily changed. The value of a user variable may contain another user variable, but, of course, cycles are not permitted.

For each consistency view, there is a consView/6 fact:

consView(ConsViewName, CheckedViewName, FileTemplate, StdoutFileTemplate, [Inputs], ShCmd) where

- `ConsViewName` is the name of the consistency view.
- `ViewName` is the name of the view this view checks.
- `FileTemplate` is the template for the output file for this consistency check. This file may be used to provide information as to why the consistency check failed (or passed.)
- `StdoutFileTemplate` is the template for the filename of stdout for an execution of this script.
- `[InputViews]` is a list of parameter input views (maybe empty)
- `ShCmd` is the shell command the executes the consistency check. The inputs are the the filename containing the view instance to be checked followd by the input view file instances. The output is the output file instance. These parameters are processed similarly to the processing for shell-commands for regular views.

View Instance Model

A ViewSys Instance is a particular instantiation of a ViewSys workflow that is identified by a name, usually indicating the source of the base views. The base views have associated version numbers, and non-base views will then have versions determined by the versions of their input views. Of course, the files (directories) that contain instances of views must all be distinct.

View instances are described by another set of facts, which are stored in a file named `viewsys_instance_info.P`. Whereas the user is responsible for creating the `viewsys_view_info.P` file, viewsys creates and maintains the `viewsys_instance_info.P` file in response to viewsys commands entered by the user.

For each view instance (base or derived), there is a viewInst/7 fact:

viewInst(View,InstName,Version,Status,Date,[InputVersions],Began) where:

- `View` is the name of a view;
- `InstName` is the name of the instance;

- `Version` is a version number of the workflow instance of this view;
- `Status` is the status of this view instance `not_generated`, `being_generated(ProcName)`, `generated`, `generation_failed`. (For base view instances this is always generated.)
- `Date` is the date-time the view instance was generated.
- `[InputVersions]` is a list of the version numbers of instances of the workflow instances of views that are inputs to this version of this workflow instance of this view. This is the empty list for an instance of a base view.
- `Began` is the date-time at which the generation of this view began. (This is the same as Date above for base view instances.) It is used to estimate how long it will take to generated this view output given its inputs.

For each consistency view instance, there is a consViewInst/7 fact:

consViewInst(ConsViewName, InstName, Version, Status, Date, [InputVersions], Began) where;

- `ConsViewName` is the name of the consistency view.
- `Version` is the version of the View that is checked. (It is also used to version the files generated by the consistency check.)
- `Status` is this consistency view, same as for viewInst status.
- `Date` is the date-time the check was generated.
- `[InputVersions]` is a list of the version numbers of the instances used in the consistency check.
- `Began` is the date-time at which the generation of this view began.

The ViewSys relations, `view/7`, `consView/6`, and `viewOrig/7`, are stored in the file named `viewsys_view_info.P`. It is read for most commands, but not updated. (Only `expand_views/1` generates this file from the file namsd `viewsys_view_orig_info.P`.) `viewInst/7`, and `consViewInst/7` are stored in the file named `viewsys_instance_info.P`, and the directory containing these files is explicitly provided to predicates that need to operate on it. The contents of the files are Prolog terms in canonical form.

A lockfile (named `lock_view` in the viewsys directory) is obtained whenever these files are read, and it is kept until reading and rewriting (if necessary) is completed.

Using ViewSys

The viewsys system is normally used as follows. The user creates a directory to hold the viewsys information. She creates a file `viewsys_view_info.P` in this directory containing the desired `view/7`, and `consView/6` facts that describe the desired view system. Then the user consults the viewsys.P package, and runs `check_viewsys/1` to report any obvious inconsistencies in the viewsys specification in `viewsys_view_info.P`. After the check passes, if any views have the `split(N)` option, the user should copy the `viewsys_view_info.P` file to a file named `viewsys_orig_view_info.P` and then run `expand_views/1` to generate the appropriate file `viewsys_view_info.P` to contain the views necessary to split, execute and combine the results. This will overwrite the `viewsys_view_info.P` file. (From then on, should the viewsys need to be modified, the user should edit the `viewsys_orig_view_info.P` file, and rerun `expand_views/1` to regenerate the `viewsys_view_info.P` file.) The

user will then run `generate_view_instance/2` to generate an instance (or instances) of the view system into the file `viewsys_instance_info.P`. After that the user will run `update_views/4` to generate all the view contents. Then the user checks the generated logging to determine if there were any errors. If so, the user corrects the programs (the viewsys specification, whatever), executes `reset_failed/2` and reruns `update_views/4`. The user can also use `viewsys_status/1` to determine what the state of the view system is, and to determine what needs to be fixed and what needs to be rerun. If the execution of `update_views/4` is aborted or somehow does not complete, the user can run `reset_unfinished/2` to reset the views that were in process, so that a subsequent `update_views/4` will try to recompute those unfinished computations.

Ideas for Possible Future Extensions

The data structures support version numbers. For now, versions are not supported; only version 0 is used. It is intended, if it becomes useful, to extend the system to support multiple versions of views.

It may be useful to somehow associate or connect multiple view systems. This might support a base view in one ViewSys that is defined in another ViewSys framework.

Perhaps we should support annotations/options to indicate how/when to delete versions of intermediate views.

We might explore the integration of incrementally maintained views, by adding difference files, and generating difference sets to be applied to the old view. This will probably initially have to be constrained to views whose increments can be computed from the inserts/deletes to a single input file.

Usage and interface (`view_sys`)

- **Exports:**
  - *Predicates:*

    `check_viewsys/1`,
    `copy_required_files/2`, `delete_instance/2`, `expand_views/1`, `generate_new_instance/2`, `generate_required_dirs/2`, `invalidate_all_instances/1`, `invalidate_view_instances/2`,
    `logfile_directory/2`, `logfile_file/2`, `monitor_running_procs/7`, `print_viewsys/1`, `reset_failed/2`, `reset_unfinished/2`, `show_failed/2`, `start_available_procs/7`, `update_instance/2`, `update_views/4`, `viewsys_status/1`, `viewsys_status/2`.
- **Other modules used:**
  - *Application modules:*

    `basics`, `file_io`, `machine`, `shell`, `standard`, `string`, `xsb_configuration`.

Documentation on exports (view_sys)

check_viewsys/1: [PREDICATE]

check_viewsys(+ViewDir) checks the contents of the viewsys_view_info.P file of the ViewDir viewsys directory for consistency and completeness.

copy_required_files/2: [PREDICATE]

This predicate can be used (perhaps with configuration help from generate_required_dirs/2) to copy and deploy view systems and the files they need to run. This predicate is not needed for normal execution of view systems.

copy_required_files(+VSDir,+FromToSubs) uses the viewsys_required_file/1 facts in the viewsys_view_info.P file in the VSDir viewsys directory to copy all directories (and files) in those facts. FromToSubs are terms of the form s(USERVAR,FROMVAL,TOVAL), where USERVAR is a variable in the file templates in the viewsys_required_file/1 facts. A recusrive cp shell command will be generated and executed for each template in viewsys_required_file/1, the source file being the template with USERVAR replaced by FROMVAL and the target file being the template with USERVAR replaced by TOVAL.

All necessary intermediate directories will be automatically created.

E.g.,

```
copy_required_files('.',[s('$DIR$','C:/XSBSYS/XSBLIB','C:/XSBSYS/XSBTEST/XSBLIB')]).
```

would copy all files/directories indicated in the viewsys_required_file/1 facts in the local viewsys_view_info.P file from under C:/XSB/XSBLIB to a (possibly) new directory C:/XSBSYS/XSBTEST/XSBLIB (assuming all file templates were rooted with DIR.)

delete_instance/2: [PREDICATE]

delete_instance(+ViewSys,+VInst) removes an entire instance (including all versions) from the view system. Any files of view contents that have been generated remain; only information concerning this instance in the viewsys_instance_info.P file is removed, so these view instances are no longer maintained.

expand_views/1: [PREDICATE]

expand_views(+ViewSys) processes view/7 definitions that have a split(N) option, generates the necessary new view/7 facts to do the split, component processing, and rejoin. It overwrites the viewsys_view_info.P file, putting the original view/7 facts into viewOrig/7 facts. This must be called (if necessary) when creating a new viewsys system and before calling generate_view_instance/2.

generate_new_instance/2: [PREDICATE]

generate_new_instance(+ViewSys,+VInst) creates a brand new instance of the view system ViewSys named VInst. It generates new viewInst/7 facts for every view (base and derived) according to the file templates defined in the baseView/4, and

view/7 facts of the ViewSys. `VInst` may be a list of instance names, in which case initial instances are created for each one.

`generate_required_dirs/2:` [PREDICATE]

This predicate can be used to help the user generate `viewsys_required_file/1` facts that may help in configuration and deployment of view systems. It is not needed to create and run normal view systems, only help configure the viewsys_view_info.P file to support using `copy_required_files/2` to move them for deployment, when that is necessary.

`generate_required_dirs(+SubstList,+LogFiles)` takes an XSB_LOGFILE (or list of XSB_LOGFILEs), normally generated by running a step in the view system, and generates (to userout) viewsys_required_file/1 facts. These can be edited and the copied into the viewsys_view_info.P file to document what directories (XSB code and general data files) are required for running this view system. The viewsys_required_file/1 facts are used by copy_required_files/2 to generate a new set of files that can run the view system.

`SubstList` is a list of substitutions of the form `s(VarString,RootDir)` that are applied to *generalize* each directory name. For example if we have a large library file structure, in subdirectores of `C:/XSBSYS/XSBLIB`, the many loaded files (in an `XSB_LOGFILE`) will start with this prefix, for example, `C:/XSBSYS/XSBLIB/apps/app_1/proc_code.xwam`. By using the substitution, `s('$DIR$','C:/XSBCVS/XSBLIB')`, that file name will be abstracted to: `'$DIR$/apps/app_1'` in the `viewsys_required_file/1` fact. Then `copy_required_files/2` can replace this variable `$DIR$` with different roots to determine the source and target of the copying.

`LogFiles` is an `XSB_LOGFILE`, that is generated by running xsb and initially calling machine:stat_set_flag(99,1). This will generate a file named `XSB_LOGFILE.txt` (in the current directory) that contains the names of all files loaded during that execution of xsb. (If the flag is set to {tt}K > 1, then the name of the generated file will be `XSB_LOGFILE_<K>.txt` where `<K>` is the number `K`.)

So, for example, after running three steps in a workflow, setting flag 99 to 2, 3, and 4 for each step respectively, one could execute:

```
| ?- generate_required_dirs([s('$DIR$','C:/XSBCVS/XSBLIB')],
                            ['XSB_LOGFILE_2.txt',
                             'XSB_LOGFILE_3.txt',
                             'XSB_LOGFILE_4.txt']).
```

which would print out facts for all directories for files in those LOGFILEs, each with the root directory abstracted.

`invalidate_all_instances/1:` [PREDICATE]

`invalidate_all_instances(+ViewSys)` invalidate all views, so a subsequent invocation of `update_views/4` would recompute them all.

`invalidate_view_instances/2:` [PREDICATE]

`invalidate_view_instances(+ViewSys,+ViewInstList)` invalidates a set of view instances indicated by `ViewInstList`. If `ViewInstList` is the atom 'all', this

invalidates all instances (exactly as `invalidate_all_instances/1)` does.) If `ViewInstList` is a list of terms of the form `View:VInst:0` then these indicated view instances (and all views that depend on them) will be invalidated. If `ViewInstList` is the atom 'filetime', then the times of the instance files will be used to invalidate view instances where the filetime of some view instance input file is later than the filetime of the view instance output file. Note this does not account for the time it takes to run the shell command that generates the view output, so for it to work, no view instance input file should be changed while a view instance is in the process of being generated.

This predicate can be used if a base instance file is replaced with a new instance. It can be used if the contents of a view instance are found not to be correct, and the generating process has been modified to fix it.

`print_viewsys/1:` [PREDICATE]

`print_viewsys(+ViewDir)` prints an indented hierarchy of the view definitions.

`reset_failed/2:` [PREDICATE]

`reset_failed(+ViewSys,+VInst)` resets view instances with name `VInst` that had failed, i.e., that are marked as `generation_failed`. Their status will be reset to `not_generated`, so after this, the next applicable call to `update_views/4` will try to regenerate the view. If `VInst` is 'all', then views of all instances will be reset.

`reset_unfinished/2:` [PREDICATE]

`reset_unfinished(+ViewSys,+ProcName)` resets view instances that are unfinished due to some abort, i.e., that are marked as `being_generated(ProcName)` after the `view_update` process named `ProcName` is no longer running scripts to generate view instances. This should only be called when the `ProcName view_update` process is not running. The statuses of these view instances will be reset to `not_generated`. After this, the next applicable `update_views/4` will try to recreate these view instances.

`show_failed/2:` [PREDICATE]

`show_failed(+VSDir,+VInst)` displays each failed view instance and consistency view instance, with file information to help a user track down why the generation, or check, of the view failed.

`update_instance/2:` [PREDICATE]

`update_instance(+ViewSys,+VInst)` updates an instance of the view system `ViewSys` named VInst. It is similar to `generate_new_instance/2` but doesn't change existing instance records. It generates a new `viewInst/7` (or `consViewInst/7`) fact for every view (base and derived) that doesn't already exist in the `viewsys_instance_info.P` file. It doesn't change instances that already exisit, thus preserving their statuses and process times.

`update_views/4:` [PREDICATE]

`update_views(+ViewSys, +ViewInstList, +ProcName, +NProcs)` is the predicate that runs the shell commands of view instances to create view instance contents. It ensures that most recent versions of the view instances in `ViewInstList` (and all instances required for those views, recursively) are up to date by executing the commands as necessary. A view instance is represented in this list by a term `View:InstName:0`. If `ViewInstList` is the atom 'all', all view instances will be processed. This predicate will determine what computations can be done concurrently and will use up to `NProcs` concurrent processes (using spawn_process on the current machine) to compute them. `ProcName` is a user-provided process namde that used to identify this (perhaps very long-running) process; it is used to indicate, in `Ststus=being_updated(ProcName)` that a view instance is in the process of being computing by this update_views invocation. `reset_unfinished/2` uses the name to identify the view instances that a particular invocation of this process is responsible for.

`viewsys_status/1:` [PREDICATE]

`viewsys_status(+ViewDir)` prints out the status of the view system indicated in `ViewDir` for all the options in `viewsys_status/2`.

`viewsys_status/2:` [PREDICATE]

`viewsys_status(+ViewDir,+Option)` prints out a particular list of view instance statuses as indicated by the value of `option` as follows:

active: View instances currently in the process of being generated.

roots: Root View instances and their current statuses. A root view instance is one that no other view depends on.

failed: View instances whose generation has failed

waiting: View instances whose computations are waiting until views they depend on are successfully update.

checks_waiting:
View instances that are waiting for consistency checks to be executed.

checks_failed:
View instances whose checks have executed and failed.

This predicate can be called in one shell when `update_views/4` is running in another shell. This allows the user to monitor the status a long-running invocation of `update_views/4`.

`logfile_directory/2:` [PREDICATE]

No further documentation available for this predicate.

`logfile_file/2:` [PREDICATE]

No further documentation available for this predicate.

`start_available_procs/7:` [PREDICATE]

`start_available_procs(+ViewSys, +ViewInstList, +ExecutingPids, +ProcName, +NProcs, +Slp, +OStr)` is an internal predicate that supports the `view_update/4` processing. It finds all views that can be generated (or checked), starts processes to compute `NProcs` of them, and then calls `monitor_running_procs/7` to monitor their progress and start more processes as these terminate. This is an internal predicate, not available for call from outside the module. The parameters to `start_available_procs/7` are:

1. `ViewSys` is the directory containing the `viewsys_info.P` file describing the view system.
2. `ViewInstList` is a) an explicit list of records of the form `View:Inst:0` identifying the (derived) views, normally 'root' views, that are intended to be generated by the currently running `update_view/4` invocation; or b) the constant 'all' indicating that all view instances of the view system are intended to be generated.
3. `ExecutingPids` are pid records of the currently running processes that have been spawned. A pid record is of the form: `pid(Pid,ShCmd,SStr,FileOut,Datime,View,Inst,Ver)`, where
   - `Pid` is the process ID of the process (as returned by `spawn_process/5`.)
   - `ShCmd` is the shell command that was used to start the process.
   - `SStr` is the output stream of the process's stdout and stderr file.
   - `FileOut` is the name of the file connected to the stdout/stderr stream.
   - `Datime` is the datime that the process was started.
   - `View` is the view the process is generating.
   - `Inst` is the instance of the view the process is generating.
   - `Ver` is the version of the instance of the view the process is generating. (Currently always 0.)
4. `ProcName` is the user-provided name of this entire update process, and is used to mark views (in the `viewsys_instance_info.P` file) during processing so they can be identified as associated to this view-update process if some error occurs.
5. `NProcs` is the number of 'processors' available for a process to be scheduled on. The 'processors' are virtual, and this is used to control the maximum number of concurrently running processes.
6. `Slp` is the number of seconds to sleep if no subprocess is available for starting before checking again to see if some subprocess has completed in the interim.
7. `OStr` is the output stream used to write progress messages when processes start and complete.

`monitor_running_procs/7:` [PREDICATE]

`monitor_running_procs(+Pids, +NProcs, +ViewSys, +VInstList, +ProcName, +Slp, +OStr)` is an internal predicate that monitors previously spawned running processes, calling `start_available_procs/7` to spawn new ones when running processes finish.

1. `Pids` is the list of process IDs of running processes. Each entry is a record of the form `pid(Pid,Cmd,StdStr,FileOut,Datime,View,Inst,Ver)` where:
   - `Pid` is the process ID of the process (as returned by `spawn_process/5`.)
   - `ShCmd` is the shell command that was used to start the process.
   - `SStr` is the output stream of the process's stdout and stderr file.
   - `FileOut` is the name of the file connected to the stdout/stderr stream.
   - `Datime` is the datime that the process was started.
   - `View` is the view the process is generating.
   - `Inst` is the instance of the view the process is generating.
   - `Ver` is the version of the instance of the view the process is generating. (Currently always 0.)
2. `NProcs` is the number of 'processors' that are currently available for use. starrt_available_procs can start up to this number of new processes.
3. `ViewSys` is the viewsys directory;
4. `VInstList` is the list of view instances (or 'all') that are being updated by this execution of update_views/4.;
5. `ProcName` is the caller-provided name of this update processor used to mark views that are being updated by this update process; and
6. `Slp` is the number of seconds to sleep if no process is available for starting.
7. `OStr` is the output stream for writing status messages;

Predicate Definition Index

Operator Definition Index

(Index is empty)

Concept Definition Index

(Index is empty)

Global Index

This is a global index containing pointers to places where concepts, predicates, modes, properties, types, applications, etc., are referred to in the text of the document. Note that due to limitations of the `info` format unfortunately only the first reference will appear in online versions of the document.

V

X