Scripting to Automate Workflows

Chris Whalen, Clovis Oncology, Boulder, CO

Amy Garrett, Robarts Clinical Trials Inc., San Diego, CA

ABSTRACT

SAS Programmers are often required to perform repetitive tasks. Such is our lot in life. Typically these are best described as a workflow: a set of defined operations performed in fixed sequence. As programmers responsible for such deliverables, we'd sure like to automate our more mundane, repetitive work. Alas, it seems some snafu always arises in our attempts: we're reminded of that one manual step required of us to produce the deliverable, dashing our hopes for automation.

Many conventionally manual steps can be automated by leveraging functionality commonly available in scripting languages. A coordinated suite of parameterized script modules able to emulate the most frequently relied upon manual operations can vastly expand the breadth of workflows that are entirely programmable, forming the basis for a reliable, highly configurable system of automation. Key features of such a system include:

- Facility to read in and process workflow specifications (XML).
- Task-specific modules dedicated to parameterizing common workflow operations.
- Means to trigger the system as required to faithfully execute target workflows autonomously.

Since we're using SAS 9.3 and 9.4 on Windows 7 and 10, we chose to use PowerShell to implement such a system. This scripting language is relatively easy to learn, highly integrated with the Windows environment, and offers ample support for XML. A designated scheduled tasks is created to regularly launch each automated workflow, calling a master script with a parameter designating the location of a target XML workflow specification. The script loads the workflow containing all information needed execute its constituent tasks. The script sequentially dispatches each task, on the basis of type, to the script module created to provide a parameterized implementation of that type. In this way, we've been able to save time and increase efficiency, affording us a greater opportunity to focus on more satisfying challenges.

INTRODUCTION: THE CASE FOR AUTOMATION

We won't sugar coat it: this automation stuff is not for the faint of heart. Sure, it sounds like a good idea on the surface. It's a tantalizing prospect: one can get carried away dreaming of the possibilities--having access to that "easy button" to press, forever after relieved of our less inspiring, more menial job responsibilities. Of course, were it so easy, wouldn't everyone be doing it? Truthfully, it will take some time and resources to succeed at this endeavor. It's fair to question whether the pursuit is worth the significant investment of time and effort required to develop a system for automation. Without some research and experimentation, we might not even be sure of what utility or efficiency we stand to gain from our implementation of a system when all is said and done. And those scheduled, repetitive obligations we're charged with? They're usually not brain surgery—rather they require following rote process amounting to a nuisance and drain on precious time more than anything else. However, in our experience as programmers, we've observed that many of us accumulate such obligations among our list of responsibilities. For anyone unfortunate enough to collect these duties faster than they shed them, the aggregate time require inevitably grows, often amounting to a significant chunk of time. Despite the legitimate reservations some may have about embarking on such project, we believe that the goal of creating a system to automate some of these tasks is quite attainable—well worth the initial outlay of time and effort required to reap the ongoing benefits of having such a system in place.

We hope you can benefit from some of our lessons learned and insights gained as we bumbled through our implementation. Note that, for a wide range of reasons--diverse IT environments, individual experiences, habits, and aptitudes of programmers, chosen scripting language, developer whims, machine configuration, process idiosyncrasies unique to the organization, etc., just to name a few-- each implementation is likely to be quite unique. Rather than focusing on the technical aspects of our implementation (which may well bore you to tears), we intend to discuss the process in general terms, touching on commonalities we believe will be helpful for anyone considering an implementation. With that, let's get started.

KNOW YOUR ENEMY: RECONAISSANCE ON THOSE PESKY, REPETETIVE TASKS

An obvious first step toward the goal of automating a repetitive task is to carefully inventory all of the manual steps required in its completion, start to finish. Take nothing for granted, accounting for every action--from the earliest sourcing or staging of required inputs all the way through delivery of the final output. The exercise will provide a glimpse of the specific challenges we'll face in its automation. The success of our implementation, of course, depends on our ability to handle each of those constituent tasks or actions that we identify. This might seem a daunting task at first, given that we might be surprised by the number of discrete actions that end up in our inventory.

Try not to get discouraged. Repeat the process for some additional automation candidates. In doing so, we'll get sense of what's required of, not just a one-off solution targeted at a specific deliverable, but rather of a general system for automation, configurable to handle a wide range of processes. We'll come to appreciate that even the most complex set of steps can be described in terms of a workflow—reduced to a list of discrete operations to be performed in a fixed sequence. More importantly, you'll likely see a very encouraging pattern emerge: while taken individually, those operations defining a workflow are highly specific, they typically fall within a very narrow range of functional categories. In our experience assessing workflows, we've found that vast majority of our operations (the elemental building blocks defining workflows) involve one of the following:

- Uploading/sourcing data via SFTP/FTP
- Performing file system operations: copying, moving, deleting, or renaming files and directories
- Executing SAS programs
- Using compression software such as WinZip, PK-Zip, 7-Zip
- Emailing notifications or attachments.

Regardless of the workflow to which they belong, there's such a high degree of similarity between operations of the same category, that it's only a small leap to imagine them simply as calls to a single parameterized version of the operation. Consider that these categories represent a set of core task types. The vast majority of workflows, then, can be stated in terms of a sequential list of calls to one of these core tasks. We're still in the exploratory phase, but it follows that our effort to implement a flexible, configurable system for automation requires that we develop a callable script module for each of these functional categories, which encapsulates and parameterizes the underlying core task.

You might find this realization encouraging. In principle, we need only automate this short list of operations to gain the capacity to handle most workflows that we encounter. Alas, to the uninitiated, these may look like manually intensive operations sure to stymie our attempts to automate. This is where scripting languages come to the rescue! Dealing with these challenges is precisely the milieu of scripting languages. In general terms, these are programming languages aimed not so much at developing autonomous applications-but rather oriented toward utility: performing operating system tasks and controlling other applications. They're frequently used to facilitate automation. If you're not familiar with scripting languages, it may well be worth dipping your toe in the water to gauge their abilities, as they offer just what we need to leap over these hurdles to automation.

With our target workflows now restated in terms of calls to these parameterized core tasks, and the understanding that a scripting language suitable to our needs and environment might provide the means to implement them, a path to developing a system of automation emerges. Progress now hinges on three tasks: (1) establishing a method to reliably capture the run sequence, functional category, and required parameters of every task that comprises our target workflow in a format that our scripts can understand, (2) creating a master script capable of reading these workflow specifications and sequentially executing the operations described within (3) developing task-specific script modules, accessible to our master script, which implement a flexible, parameterized version of the each core task type (i.e. an abstraction which will group related operations under the umbrella of a single function call with a standard set of parameters). That's a clunky way of putting it. You might just refer to each as a "Task Handler."

CAPTURING THE WORKFLOW

If you wanted to communicate to a coworker all of the information they would need to produce one of your deliverables, you might opt to write them a detailed set of instructions—maybe throw in a diagram or two. As for conveying this information to a scripting language, it turns out that XML (Extensible Markup Language) is particularly well suited to the job of cataloguing these details. This format for storing data is highly flexible (dare I say, extensible), somewhat self-documenting, and human readable (although that last perk might be a stretch sometimes). It lends itself well to capturing and organizing the hierarchy of details that arise from adequately defining a workflow. Comprised of tags, it looks vaguely reminiscent to HTML. In contrast, its focus is to serve as a highly configurable framework for storing data (rather than HTML, which deals mainly in its display). Unlike those used in HTLM, XML tags are not predefined. You create your own tags to define elements and organize them based upon your needs.

As a bonus, most scripting languages have native ability to read and parse XML content built-in. If we'd like to make this implementation as easy on ourselves as possible, XML is just too good a fit to pass up (even if it might be a little ugly).

The following page shows an example of how someone might document a workflow specification using basic XML. With an eye toward portability, the format includes provisions for formally documenting and validating its structure and content. There's certainly no requirement to include these elements in your format. The specification below doesn't conform to any outside standard. Even so, it's very effective in using the basic features of XML as a tool to thoroughly characterize and organize elements of a workflow. There's no right or wrong way to format your workflow specification, so long as it enables you to capture your processes with the requisite accuracy and consistency o allow them to be faithfully and reliably reproduced. In the interest of transparency, that "someone" who might document their workflow like below? That would be us. This an active workflow script which has been stripped down to its basic elements and anonymized.

Example XML Workflow Specification

```
<WORKFLOW name=”SDV_RUN” descr=”Weekly SDV Run” email=”cwhalen@clovisoncology.com”>
   <TASK name=”FTP_IN” descr=”FTP in latest rde” type=”FTP”>
      <connection  descr=”Connection details”>
         <param name=”FTP Host”> ftp02.ftp.mdsol.com</param>
         <param name=”LOGIN”   >cwhalen</param>
         <param name=”Hostkey” >ssh-dss 1024 f2:db...ad:a7:e1:9e:ca:41</param>
      </connection>

      <action type=”DOWNLOAD”>
         <param name=”PATH_SRV”>/HRP_04/Extracts/HRP04_RDE-SASD.zip</param>
         <param name=”PATH_LOC”>S:\PRO4\HR\data\extracts\Nightly.zip</param>
      </action>

   </TASK>
   <TASK name=”UNZIP_RDE” descr=”Exract to srcedata” type=”ZIP_UNZIP”>

      <param=”ACTION”  >UNZIP</param>
      <param=”ZIPPED”  >S:\PRO4\HR\data\extracts\Nightly.zip</param>
      <param=”UNZIPPED”>S:\PRO4\HR\data\srcedata</param>

   </TASK>

   <TASK name="SAS_RUNS" descr="Run SDV Programs" type="SAS">
      <param logdir=”Y” lstdir=”N”>S:\PRO4\HR\SAS\adam\prog\ADSL.sas</param>
      <param logdir=”Y” lstdir=”N”>S:\PRO4\HR\SAS\Report\prog\sdv_report_p1.sas</param>
      <param logdir=”Y” lstdir=”N”>S:\PRO4\HR\SAS\Report\prog\sdv_report_p2.sas</param>
   </TASK>

   <TASK name="TO_PORTAL" descr="Copy up to sharepoint" type="COPYMOVE">
      <filter>ignoretime</filter>
      <param name="from">S:\PRO4\HR\SAS\Report\out\sdv_p1.xlsx</param>
      <param name="to">\\clovis-1.share.com\ClinOps\PRO4\SDV</param>
      <param name="from">S:\PRO4\HR\SAS\Report\out\sdv_p2.xlsx</param>
      <param name="to">\\clovis-1.share.com\ClinOps\PRO4\SDV</param>
   </TASK>

   <TASK name="EMAIL_REPORTS" type="EMAIL" descr="External SDV Dist.">
      <param name="To">name1@domain.biz anonymousCRA@cliniresearch.us</param>
      <param name="Subject">PRO4-HR Weekly SDV Reports</param>
      <param name="Attachments">S:\PRO4\HR\SAS\Report\out\sdv_p1.xlsx</param>
      <param name="Attachments">S:\PRO4\HR\SAS\Report\out\sdv_p2.xlsx</param>
      <param name="Message">
         Hi All,

         Attached, please find the latest SDV Reports for PRO4-HR.

         Please don’t hesitate to contact me if you have any problems or concerns.

         Thanks,

         -Chris</param>
   </TASK>

</WORKFLOW>
```

The example uses a root element "Workflow" defined by a set of opening and closing tags:

<WORKFLOW name= descr= email= > </WORKFLOW>.

As the root element, it represents the workflow as a whole. A few attributes relevant to the workflow (and, by extension, all the elements it contains) are embedded within the opening "Workflow" tag. There's not much to it beyond that. Basically it serves as an organizational element to contain a set of defining tasks.

Nested between the opening and closing "Workflow" tags are one or more task units, each of which is denoted by an opening and closing set of "Task" tags:

<TASK name= type= descr= > </TASK>.

You'll notice that these tags are indented, indicating that they are subordinate (child) elements belonging to <WORKFLOW>.   The task unit is the fundamental element defining each individual task belonging to the workflow. It serves as an informational header to providing each task a common set of useful attributes: name, type, and description. Name allows the task to have an identity (not so important now, but we'll see how this can be helpful later).   Type denotes the category to which the task belongs (specifically, its core task type). This information is critical because it allows the master script to associate each task unit to the script module was written to implement that task unit's core task type. During runtime, the master processes each task in sequence and, based on "Type", dispatches a copy of the task unit (and all of its contained parameter values) to its associated script module/core task implementation to be handled for execution. Thankfully for us, there's one more attribute, "descr", which briefly summarizes the purpose of the task in plain English. Although superfluous to the executing script, it helps us to make sense of this jumble of tags.

Finally, contained within each task unit are the actual parameter values which the relevant task-specific script module requires to successfully complete the operation. This content is highly variable (and may even have its own hierarchy), reflecting the unique input parameters required by the each of the core task types that will be implemented.

With some careful consideration we can use the flexibility offered by XML to land on a standard format.   In doing so, we have the means to adequately document any workflow in a consistent manner that's both readable by us humans, and easily parsed by our implementation scripts.

WE HAVE AUTOMATION

It's about time to do some scripting. We already have one critical component out of the way: a stable set of conventions we can use to effectively encode our workflows with XML. You'll recall that we're still on the hook for two others: the master script which will govern the overall execution of our workflow specification, and the suite of scripts implementing the core tasks (each of which provides an abstraction layer essentially transforming all of the those distinct but functionally similar operations within a task category, into the same thing: a call to the scripted implementation of that core task, using a uniform set of parameters).

While it might have required time and thoughtful planning to define our specification format, those efforts will immediately start paying off during script development. Firstly, our choice of defining our format using XML is going to save us a lot of time (and more than a little aggravation). With native ability to parse XML likely built in to your scripting language, the master script and core functions might not be built yet, but they already know how to read our workflow specification! Furthermore, beyond its intended use, this specification format we labored over offers another perk: each workflow which has been translated into our standard format can then serve as a rough specification to guide development.

As we work to code our task-specific script modules, a completed workflow specification (preferably one containing operations representative of every core task type) can help document our system through example. Disparate as our core tasks may be, we can rely on getting helpful details from all of them: a "Type", indicating which script module will support the task, a "Descr", reminding us of the functionality expected to be provided by that script module, and a set of task-specific internal parameters, showing what information we can draw upon at runtime to meet functional requirements. This will be helpful information to the person coding the script module, as they attempt to implement the dynamic, parameterized version of the core task

In designing and coding the master script, we can turn our attention to the higher level operations required to manage the execution process as whole.   By now, we have a solid concept of the how tasks will be processed: the master script will see only an abstraction of each task—a container with three attributes. Looping through each instance of <TASK> units, it will read the value of "type", and call the script module associated with that value. That's it. Then it'll just wait until control returns and do it again, until the end of the list.

At the risk of belaboring the point, we'd like to remind you of the ability to process XML that your script will likely possess (without you doing a thing). When the master script loads your XML specification, the entire contents will probably be parsed and organized into a very easily navigated object. To provide perspective for how this can help simplify the coding, consider this example. In our implementation, the call to the applicable script module for each <TASK>.type? It actually **IS** the <TASK>.type! Given these facts, code for the whole process resembles something like this:

```
foreach($task in $workflow.TASK){

    $task.type;

}
```

Aren't you glad we opted to use XML?

In short, the master script might not be very had to implement as you might have imagined. In the basic implementation, it need only: load the specified XML, create a log (very nice to have), and loop through the <TASK> units. We've seen just how easy that might be when dealing with XML.

Once the coding required for the task handlers and the master script is completed, we'll be well on our way. We'll have a consistent workflow specification format, a suite of script modules to handle those parameterized tasks that define each workflow, and a callable master script which need only be directed to a workflow specification, to autonomously load the XML and execute all the operations specified within. With these in place, we have a basic framework to handle any permutations in core tasks that a workflow might require. This gives our system the potential to automate a surprisingly wide range of workflows. We're ready to ready for liftoff!

OK, WELL SORT OF

In principle, we now have a basic system of automation at our disposal: standard conventions allowing us to comprehensively encode workflows into a standard, machine readable format and the programming in place to autonomously load a valid specification, parse the instructions within, and execute all operations of a workflow in their proper sequence, thus flawlessly automating the process. In this alone, we may already find some utility to reap. However, there's still work to be done and much to consider before we arrive at a stable, dependable, and flexible automation tool. That's ok. You might opt for an iterative approach to your project, reworking and refining existing features or adding new ones along the way. Once the fundamentals are in place, it may make sense to turn efforts toward enhancing the reliability and usability of the system. Some enhancements might be very minor. For instance, in our implementation, all workflow specifications reside in a single dedicated folder and are named using a specific convention. To call the tool, we need only specify what amounts to an alias for the target workflow. The script expands that alias into fully its qualified filename equivalent, and proceeds to load the target workflow specification from our designated repository. Some things amount to being a convenience (like this previous example). Others might have a larger impact on reliability, ease of use, and reusability. Here are some things to consider right away. While we can share our experience in implementing a system, describing what worked well for us, we're certainly not experts. You might want to handle these concerns differently—whichever solution works for you.

SOME CONSIDERATIONS

Triggers

Our system is ready to take the wheel and perform some workflows autonomously. So how can we control when it runs? For our implementation we opted to create each workflow as its own dedicated scheduled job, setting each to run daily (using the "Task Scheduler" utility for those who speak Windows). This works fine if you wish to run every day. There may be instances where that's preferable (after all, our script never gets tired). However, in order to both grant users more control over when workflows get executed, while retaining that simple scheme of having a lone daily scheduled job for each workflow, we decided to add an enhancement. In our implementation, the root element in the workflow specification XML may contain another type of child element beyond its task units: a "Dates" element denoted by an opening and closing set of "Dates" tags: <DATES> </DATES>. The "JobRunner" runner still gets kicked off every day for each workflow by Task Scheduler. Only, now it first performs a check against the contents of this element. If the current date matches any condition among one or more string tokens enclosed by those "Dates" tags, the script will proceed with executing the workflow. If no match is found, it just calls it a day and exits out. In this scheme, an empty or altogether missing "Date" element frees the master script to execute the workflow any time it's called. In an effort to increase the tool's flexibility, we got a little fancy and modified our master to allow a few different options for specifying valid run dates. The directives contained in this example "Dates" tag show what's available:

<DATES> 12-Mar-2017 23Jul2017 MON THURS 1ST*TUES </DATES>.

At runtime, the executing master script attempts to match the current date (as stated in any of a few formats: fully qualified dates dd-mon-yyyy or ddmonyyy, one or more days of the week, or Nth occurrence of a certain weekday so far in the current month) against whatever directives may be enclosed by the “Dates” tags. If there’s a match, the master script will proceed as usual to execute all of the task units in the specification.

Sometimes we need to provide a deliverable frequently, yet not necessarily on a fixed schedule. Ideas for alternative triggers to address this scenario include launching the script or scheduled job directly if conditions warrant:

```
%if ("&new_rde" ne "" or "&coding_file" ne "") %then %do;
    /** call the job runner task to refresh the CO338014 data **/
    data _null_;
      call system ('schtasks /RUN  /TN "SAS_JOBS_CO338014_MANUAL"');
    run;
%end;
```

Another strategy could be to implement an event based solution such as a file trigger. Admittedly, we haven’t implemented this model yet (but look forward to working out the details when the requirement arises). One strategy might include modifying the master script to maintain a list of last run dates of every workflow that it executes and scheduling the master to run frequently (maybe hourly would suffice for your needs). Then, when activated each hour, the script could poll the created or last modified date for each file in the predefined list of triggers, executing any workflow linked to a trigger file that’s been updated since the last run of that workflow. If the need exists, surely you’ll come up with plenty innovative ideas—and hopefully have fun doing it.

Handling Missteps

You can count on something going awry with this Rube Goldberg at some point—especially in the early days initiating your system, or when introducing new workflows to its repertoire. Even later, as your automation framework matures, it’s bound to accumulate new features (and commensurate complexity). The list of interrelated “moving parts” orchestrated to work in close harmony to implement enhancements will grow, and with it, the modes of potential failure. It’s prudent to be proactive from the start and make the graceful handling of failures a priority.

In the example workflow specification, you may have noticed workflow-level email contact. It’s imperative that you build provisions into the system, allowing it to reliably track the success (or failure) of tasks, and alert an appropriate contact should something go wrong. The more detail you can provide with failure notifications, the better (you’re a SAS programmer after all and know the importance of warning and error messages in the SAS log). It’s also helpful to have all the modules contribute to maintaining a detailed log of processes to take away some of the mystery in debugging inevitable problems.

All failures aren’t created equal. There may be even be circumstances in which the failure of given task can be tolerated in the immediate term. Of course, there’ll certainly be operations that are entirely critical. With our implementation, we decided to take error tracking a step further and create a rudimentary system for defining task dependencies. It’s not so fancy that it can roll back tasks, but it’s helpful to have control over the execution of workflow operations defined further along in the workflow spec, subsequent to a failed task. You can imagine a scenario whereby a task which creates input files used by subsequent operation fails, causing that dependent operation to overwrite the last known good output with missing or corrupted content. It might serve our customers better to leave the output from the prior run intact until the point of failure is addressed and the workflow be rerun without issue.

To provide this control, we updated the various task-specific modules with an appropriate means of evaluating the success or failure of any task they’re dispatched to handle. When execution is completed, the generic name attribute of that dispatched task is logged in a globally accessible array (a hash table) using task name as the key, and its run status as the value). For its part, the master script checks each task unit it encounters during runtime for any specified dependencies, designated by enclosing the name attribute of the required task within a set of <DEPENDENCY></DEPENDENCY> tags. For any that exist, the master script confirms that the global log contains an entry indicating successful completion of the all dependencies (key matching the name attribute of the dependency, and value indicating successful completion). If all specified dependencies requirements are met, the master will dispatch the task for execution. Otherwise, the current task is skipped and logged as failure for lacking its required dependencies.

Typically, dependencies are just chained, with each task unit dependent upon the success of the task immediately preceding task. The below example shows a slightly different scenario. Here EMAIL_REPORTS is free to run so long as the task identified as SAS_RUNS ran with success (incidentally, the SAS task implementation determines run status based on the return code it receives from a batch submit of program call it contains). If the TO_PORTAL task encounters a hiccup copying content up to Sharepoint (not terribly uncommon) this arrangement allows at least part of the deliverable to be completed. It’s especially nice to have control like this with regard to an email task (saves the embarrassment of spamming folks with junk email containing stale output or nothing attached).

```
<TASK name="TO_PORTAL" descr="Copy up to sharepoint" type="COPYMOVE">
    <DEPENDENCY>SAS_RUNS</DEPENDENCY>
    <filter>ignoretime</filter>
    <param name="from">S:\PRO4\HR\SAS\Report\out\sdv_p1.xlsx</param>
    . . .
</TASK>

<TASK name="EMAIL_REPORTS" type="EMAIL" descr="External SDV Dist.">
    <DEPENDENCY>SAS_RUNS</DEPENDENCY>
    <param name="To">name1@domain.biz anonymousCRA@cliniresearch.us</param>
    . . .
</TASK>
```

Of course, it might be safer (and easier to implement) to have your system always stop everything to a grinding halt if anything unexpected rears its head—or maybe using the previous task as a default dependency that can be overridden by including a <DEPENDENCY></DEPENDENCY> element (even if empty to declare a lack of any dependency). Again, whatever works for you!

Hacks for Usability and Reusability

I'm sure you'll come up with plenty of ideas to get the most out of your system. We thought we would share just a couple ideas that have worked out well for us.

1. One nice feature to have is the ability to specify dynamic date substitutions. This wasn't planned for during our initial implementation. Alas, experience has shown that it's fairly common for an established stable workflow to contain steps that reference dynamic folders and file names (typically these reflect the run-time date). This threw a curve ball to our script module implementations which were developed with the assumption that the details of each workflow specification (and thus any task units contained within) were known up front—while parameter values were anticipated to be highly variable between calls to a task implementation, no provision was made to make them dynamic in any way.

   In our implementation, we had to create a workaround to handle these scenarios. Again, modifications were made such that now, almost every script module which requires parameters defining a path or filename first routes those parameter values to a function dedicated to preprocessing them. The function scans each value to check for a token indicating that a dynamic date substitution is required to be performed on the parameter value. It's simpler than it might sound. In this below example, the local file name for the FTP download has a date substitution (denoted by embedding a set of brackets [ ] within the parameter value for a path or filename which specify a basic format model to use for date the substitution). Were the workflow containing this example task to be triggered on 12/Feb/2018, the local file name will get written as "*S:\PRO4\HR\data\extracts\HRP04_2018FEB12.zip ".* While the coding isn't too terribly difficult to pull something like this off, small enhancements such like these can go a long way toward expanding the system's flexibility and reach.

   ```
   <TASK name="FTP_IN" descr="FTP in latest rde" type="FTP">
   . . .
   <action type="DOWNLOAD">
       <param name="PATH_SRV">/HRP_04/Extracts/HRP04_RDE-SASD.zip</param>
       <param name="PATH_LOC">S:\PRO4\HR\data\extracts\HRP04_[YYYYMONDD].zip</param>
       </action>
   </TASK>
   ```

2. A very minor feature that proved to be very helpful was to provide alternate modes for running a workflow. Other than being scheduled, we have the ability to run workflows interactively. You might not expect it, but we get a lot of utility out of being able to easily run our workflows on an ad hoc basis. In our implementation, we use a .bat file to invoke the master script with parameters instructing it to run interactively. The .bat prompts the user to designate a workflow spec to be immediately executed. We also have the option to specify a space delimited lists of task names to execute during the interactive run, allowing us to target a subset of member tasks within the workflow. In this mode, our master script just ignores any of those "Dates" tags that may be coded into workflow definition. "Interactive mode" is invaluable in testing new workflows. You can immediately unit test any specific task or the workflow as a whole (no need to alter the scheduled task for waiting around for it to trigger).

3. Some astute readers might have looked at the example XML workflow specification presented on page three

and smelled something fishy. Something about the FTP task doesn't add up. Perhaps the reader is thinking, "Wait. Where's the password?" The answer is a little complicated.

In our environment, the master script runs under the rights of its executing user (or the user who created the scheduled task which executed it) Windows provides some encryption capabilities which allowing encryption /decryption with a key that's specific to the current windows user and the host providing the encryption services. Or is that the machine on which the encrypted files are stored? Clearly, we're no experts and are learning as we go along. The current configuration of our implementation allows for a server install. We developed a companion script which allows each user to maintain their own list of cached passwords, accessible only to them, and the automation tool when executing with their privileges. When entering the password, in the companion script, we're also prompted to enter the exact URL for which it will be used. The password entry is assigned key value unique for that user (simply the user's name and the URL of the resource the password will be used to access. The following page demonstrates a password cache (the encrypted string is truncated-it's actually quite long). The "key" relevant to the FTP example highlighted. Note, at this point it should come as no surprise that we've chose to store this data in an XML format.

User-Specific Cache Example

```
<CONFIG name="JobRunner">

    <ACCOUNTS type="websites">

        <ACCOUNT  Site="ftp://ftp.christopherwhalen.com"
                  User="wwhalenc"
                  Key="wwhalencftp://ftp.christopherwhalen.com"
                  Index="4"
                  Note="cw.com ftp testing"
                  Password="01000000d08c000000e3cf3eb. . . 236c1d b0a259ce333" />

        <ACCOUNT  Site=" ftp02.ftp.mdsol.com"
                  User="cwhalen"
                  Key="chwhalencftp02.ftp.mdsol.com "
                  Index="1"
                  Note=""
                  Password="01000000d08c9ddf01. . .1c5286f40fc6e3d1637a" />
    </ACCOUNTS>
</CONFIG>
```

Here's rundown of how a secure resource is accessed in our implementation.

a. During its initialization, the script attempts load a password cache for the current user. Caches are stored in a designated folder and use a standard naming convention reflecting the user with whom they're associated (e.g. if Chris Whalen is logged on (or his scheduled task is executing), the script checks the cache repository for a file named "cfg_cwhalen.config".

b. If a matching cache is found, the file is loaded, its passwords decrypted stored in memory as a hash table. Note: this process will fail miserably if the cache file loaded was not both created by cwhalen and done so on this server.

c. At runtime, when the FTP task is executed, it uses other task parameters cues to determine which of the cached passwords to use for the task.

```
<TASK name="FTP_IN" descr="FTP in latest rde" type="FTP">
    <connection  descr="Connection details">
        <param name="FTP Host"> ftp02.ftp.mdsol.com</param>
        <param name="LOGIN"   >cwhalen</param>
        <param name="Hostkey" >ssh-dss 1024 f2:db...ad:a7:e1:9e:ca:41</param>
</connection>
. . .
```

Recall that I entered the relevant URL into the companion tool (which, in this task, equates to the parameter value for "FTP Host". The companion script creates an identifier for each set of credentials consisting of concatenation of the user and URL (see highlighted text in "User-Specified Cache Example" above). During the handling of this task, the FTP implementation module is coded to use these parameter names whenever deducing the key needed to identify a stored password to this task. This entails just piecing together two required, unencrypted, task-specific parameters: "FTP Host" and "LOGIN" at runtime to determine the identifier for the relevant password: "ftp02.ftp.mdsol.comcwhalen"

FINALLY, it uses this key to pull the decrypted value of the applicable password from its array of runtime passwords it loaded at initialization and passes that value to the FTP server to create its connection.

```
$pw=$credentials['ftp02.ftp.mdsol.comcwhalen']
```

Wow, is that clunky! It does work though, and provides the ability for multiple people to share a server based installation, with some key limitations:

1) As discussed above, the task must run with the privileges of the user who created the launching task.
2) While this model provides for users to share the tool without sharing secure credentials, it detracts from the tool's ease of use, having implications which might not be immediately intuitive. For instance, under normal circumstances, all users can use the same workflow specification, provided they have the rights to all the resources tasks within need to access. Where workflows involving a secure connection are involved, everyone needs to use a personalized copy of what's essentially the same workflow. Even though nothing is encrypted in the workflow specification itself, the system still uses those parameter values as a key to apply the correct password for the executing user.

Perhaps these shouldn't be considered limitations, but, rather, adequate security measures. Yes, the solution adds complexity, but this scenario of multiple people using a shared application to access private resources makes is bound to give rise to a convoluted security picture.

CONCLUSION

We hope that we've made a reasonable case to support our assertion that automation is achievable.
Alas, the path toward automation is anything but automatic, requiring some research, ingenuity, and more than a little hard work to for success. Of course, we think you can do it--after all, you're SAS Programmers! There's a message we wanted to be sure didn't get lost among all of that discussion about process and standards and design considerations and whatever else we touched on: we found intrinsic merit in the implementation process. It provided both a plethora of opportunities to push us out of our comfort zone and embrace new skills, as well as unanticipated problems, challenging us to exercise creative ingenuity to overcome. Sure, a lot of work was involved, but all in all it was pretty fun! We wish you success if you decide to take on the task and only hope you find it as interesting and satisfying a process as we did. Have fun - you can do it!

CONTACT INFORMATION

Thanks for allowing us to share our interest in this topic. Please feel free to contact us with any comments or questions about what we've discussed. It's always interesting to hear what creative solutions people may come up with the meet the challenges of implementing an automation.

Chris Whalen	Amy Garrett
Clovis Oncology	Robarts Clinical Trails Inc.
5500 Flatiron Parkway	4350 Executive Drive, Suite 210
Boulder, CO 80301	San Diego, CA 92121
(w) 303.625.5060	
cwhalen@clovisoncology.com	amy.garrett@robartsinc.com