# Static Module Verifier – Technical Overview

## Code Artifacts

All code is currently checked into SLAMSD under the *t-apoup* source branch. There are two major parts:

1. Changes under the SLAM folder so that certain tools now have an “—smv” option that they support. This enables behaviors for SMV. If the flag is not passed, tool behaviors should be normal and unchanged w.r.t. *main* branch.
2. SMV folder in t-apoup branch contains all the code that is used for building and deploying SMV. There are multiple projects, all under one single VS solution (smv.sln):
   1. **SmvLibrary**: this DLL is the core of the SMV system. This is where all the technology for executing tasks, building, farming tasks to the cloud etc. lives.
   2. **SmvSkeleton**: this executable is smv.exe. it utilizes other parts of the solution such as the library etc. to stitch all scenarios together.
   3. **SmvDb**: this project defines the schema for the database and some corresponding utilities. The utilities are SPROCS (SQL stored procedures). The DB can be published by right clicking on the project and publishing to a SQL server of choice.
   4. **Smv2Sql**: this utility will upload build artifacts such as RAWCFG files, LI files, BPL files etc. from the specified directory (usually build output directory) to a SQL server which has the SMV database. The connection strings are specified in the corresponding configuration file for the Smv2Sql executable.
   5. **SmvAccessor**: this library provides high level abstract access functions for the SMV database. This can be used by any utility to access the DB given the right connection strings.
   6. **SmvInterceptor**: the interceptor is a generic executable that when invoked will look for an intercept.xml configuration file which specifies the list of actions that needs to be performed by the interceptor. Before that it will remove itself from the path and then execute the list of actions. A detailed explanation can be found in SmvInterceptor.doc in %slamsrc%\smv\smvinterceptor\SmvInterceptor.doc
   7. **SmvInterceptorWrapper**: the interceptor wrapper is a wrapper on top of the interceptor to provide one layer of specialized actions and then the generic layer of interception. Ideally, this should not be needed, but as we evolved there was need for some special behavior for some of the tools, and this had to be handled correctly and by not hacking the interceptor itself. So this layer can eventually be removed with some more engineering work.
   8. **SMV Plugins**
      1. **SmvLineCounter**: a simple plugin to count the number of lines in a module.
      2. **SmvSdv**: SDV in the SMV world. Almost fully functional.
      3. **SmvSymDiff**: an interesting plugin that performs very specialized actions using SMV (for example, taking two versions of a module, which SMV has no idea about but still accommodates the plugin as needed).
      4. **SmvTest**: plugin used to test SMV. Has to be configured slightly before it can be used. The configuration is just path configurations in the XML configuration file of the plugin.
   9. **SmvAccessorDemo**: defunct now. Used for testing the SmvAccessor. Can be reinstated if needed.

## Design Overview

### General

SMV is a self-contained standalone system that has the following primary objectives:

* facilitate build interception to create the required build artifacts and Intermediate Representation of the module being analyzed
* facilitate the execution of analyses on the produced build artifacts
* facilitate verification as a service wherein build artifacts can be saved for future use and re-use
* facilitate verification as a service wherein analyses can be run in parallel using cloud infrastructure such as Azure

SMV takes as input the following:

* Module that is to be built/analyzed. This is currently the directory where SMV is executed
* Configuration file that specifies how the module is to be built, how the build is to be intercepted, and what analysis if any is to be executed after that. The XML file has two primary sections – Build and Analysis. The schema can be found in %SLAMSD%\src\t-apoup\smv\smv\SmvLibrary\config.xsd.
* Optionally, a path to analysis plugin DLL to specify the analysis plugin that should be used for this run of SMV.

### Plugins

SMV is designed with a plugin based architecture. The following figure shows how an analysis tool can be developed using SMV. The left part of the picture is all the infrastructure that is provided by SMV. This includes the build interception technology, the framework for executing the analysis, and using the cloud services. The right part of the picture shows the analysis plugin, or the analysis tool that needs to be plugged in. Each plugin consists of the following pieces:

1. XML configuration files: these are the tool specific XML files that tell SMV how to build and analyze the module in question
2. Optionally: C# code that implements the ISMVPlugin interface which can be used to perform pre-build, post-build, pre-analysis, post-analysis, actions, parse specialized command line arguments etc. etc.
   1. Documentation about the interface can be found in the code itself and is mostly self-explanatory.
3. The tool binaries and configuration files and other metadata itself. For example, if you have an engine that requires some data files, the bin folder should contain the binaries and the data files.

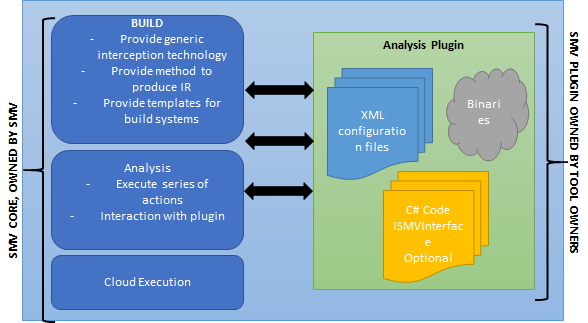
The primary idea behind such an architecture is the clear separation between what the SMV framework and platform provides and what the tool provides. It could be the case that the tool implements some functionality which is generic enough so that SMV should provide that as part of the framework. In this case, SMV can be extended or developed some more. But in a lot of cases the tool will just have some really specific functionality that needs to be owned by the tool developers and not SMV.

Figure : Plugin architecture of SMV

### ISMVPlugin

SMV exposes the following methods through this interface:

·        void Initialize()  
It is called during argument parsing by SMV when the plugin DLL is loaded. Can be used to set plugin specific properties which can be used by the plugin or even in the Config file.

·        void PrintPluginHelp()  
Called when SMV is executed with /help OR /? command line arguments. Can be used to print out additional plugin specific help information to the command line, for eg. information about the extra command line arguments enabled by the plugin.

·        void ProcessPluginArgument(string[] args)  
Called after SMV parses the command line arguments. This method is called with the array of all arguments passed on the command line. Can be used to enable additional plugin specific command line arguments. It can also augment the functionality of the existing SMV arguments.

·        void PreAction(SMVAction action)  
Called before every action is executed with the action object as argument. Can be used to do some preprocessing for all or some specific actions.

·        void PostAction(SMVAction action)  
Called after every action is executed with the action object as argument \*if the action was executed successfully\*. Can be used to do some post-processing for all or some specific actions.

·        void PostBuild(SMVAction[] buildActions)  
\*If build actions are present in the Config\*, called after the build actions have been executed with the array of build actions as argument. Can be used to do post-processing after build / before analysis.

·        bool DoPluginAnalysis(SMVAction[] analysisActions)

\*CANNOT BE A STUB IF ANALYSIS ACTIONS NEED TO BE EXECUTED\*, called during the analysis step with the array of analysis actions as argument. Can be used to do custom analysis or execute them in the default SMV manner.  Refer SmvSdv for custom analysis and SmvTest for default analysis. Returns whether the analysis was successful.

·        void PostAnalysis(SMVAction[] analysisActions)  
\*If analysis actions are present in the Config\*, called after the analysis actions have been executed with the array of analysis actions as argument. Can be used to do post-processing after analysis / before displaying results.

An example of a plugin is the SmvSDV plugin in the SMV solution. This plugin implements the ISMVPlugin interface and contains a series of configuration files for Razzle and MSBUILD. There is also a script – the staticdv.cmd script which essentially is a wrapper to all the calls for SMV. Figure 2 shows the SDV plugin with a potential split of ownership. All SDV specific functionality like the engine, viewer, GUI etc. are to be owned by Driver Quality Team. It could be the case that the viewer becomes very generic in nature and is provided by SMV, but this would require further thought and discussion.

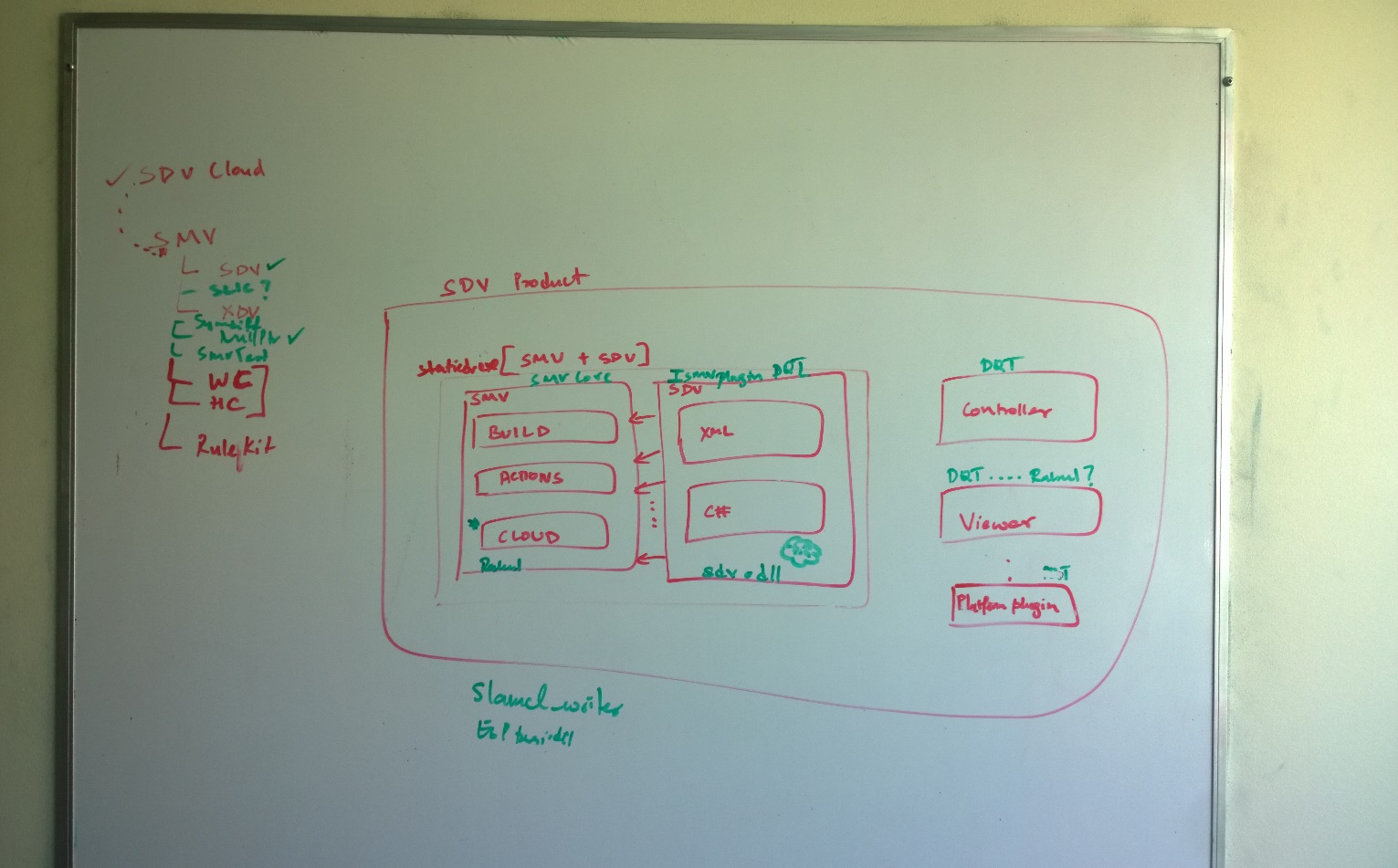
Other examples of plugins are the SmvTest plugin which is used to test SMV itself. It contains a configuration file which compromises of 98% of the plugin, and then some very minimal implementation of the ISMVPlugin interface. SmvSymdiff is similar, with one exception: it implements custom arguments in the ISMVPlugin interface for accepting arguments about two versions of the same module. It also is a good example of retrieving objects from the server for doing analysis.

Figure 2: SDV plugin using SMV

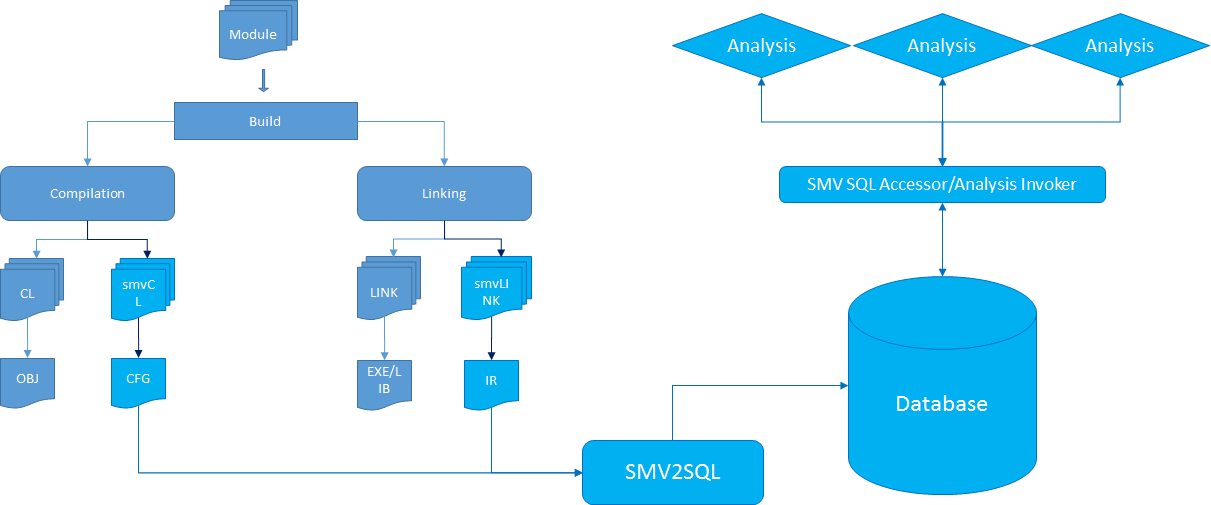
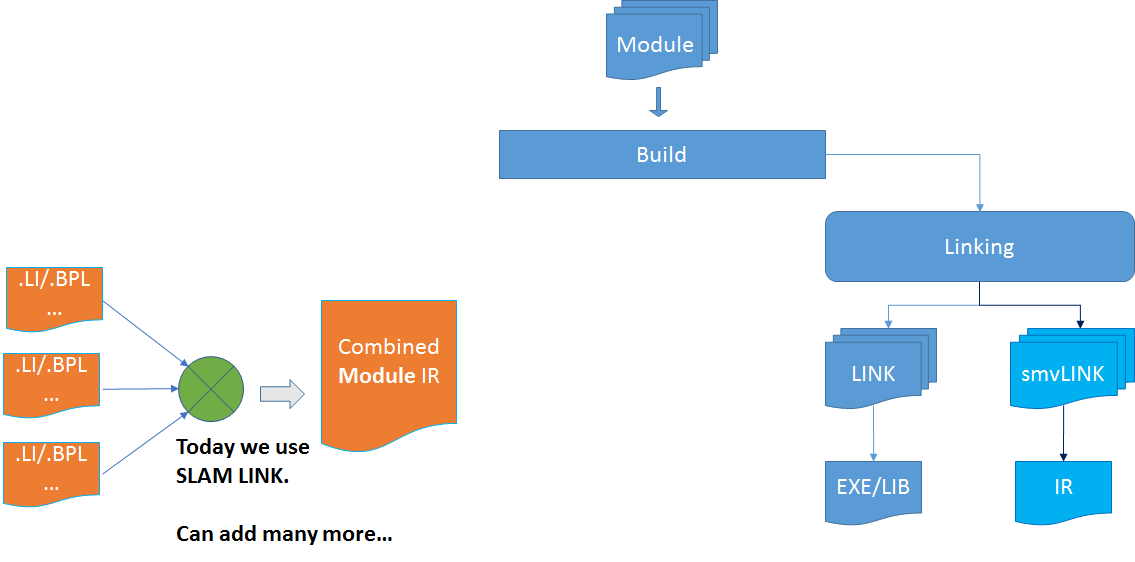
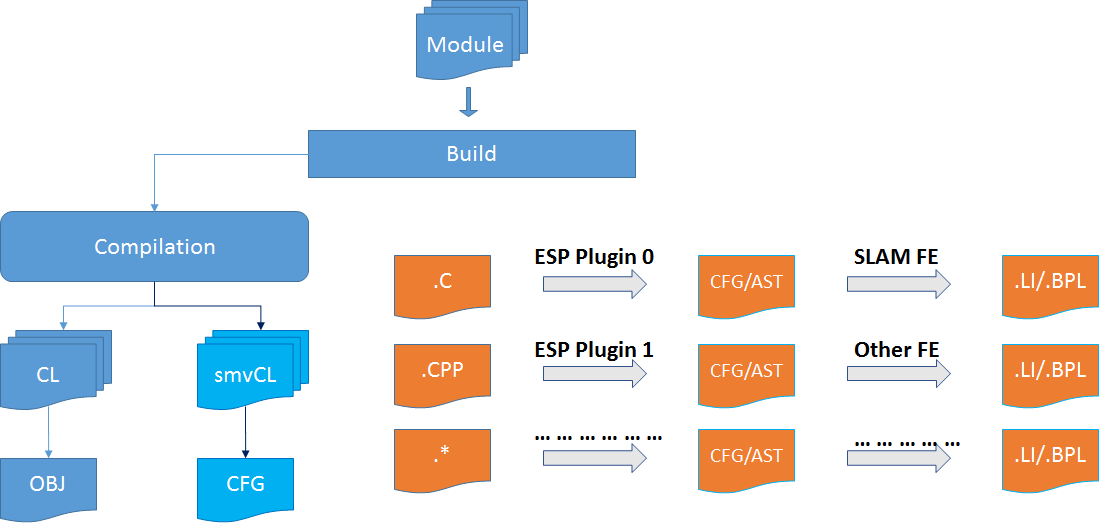
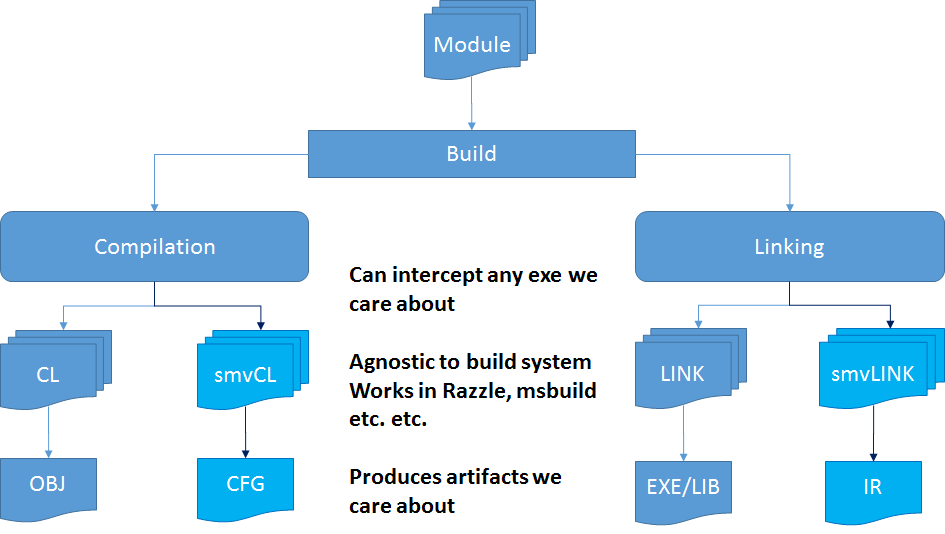
### Interception

SMV relies on the SmvInterceptor executable for build interception. The documentation for the interception is detailed and listed in the project list earlier. The essential idea is that if an executable by the name of “ABC.exe” needs to be intercepted, we copy the interceptor.exe as “ABC.exe” and ensure that our “ABC.exe” is placed in the path before any other “ABC.exe”. Then when ABC.exe is called, our interceptor gets called automatically. The interceptor then removes itself from the path and performs the actions as described in the intercept.xml file. The advantage here is that multiple new cl.exe calls can be issues with different ESP plugins etc. to create all kinds of build artifacts. This technology is very very generalized and scalable and can have many applications.

Currently we use this to intercept

* Cl.exe:
  + produce rawcfg files for each cl.exe call issued by the build system.
  + Multiple ESP plugins and cl calls can be made here
* Link.exe:
  + convert the rawcfg files to LI files
  + stitch them together to create module level LI files
  + convert rawcfg files to BPL files
  + upload all the build artifcats using smv2sql.exe
  + link in any library LI files that can be found for the current module being built
* Lib.exe: same behavior as Link.exe

Figures 3, 4, and 5 visually depict the interception etc. It should be noted that the configuration files ensure that the path is updated in a manner which guarantees that our interception executables are in the path first. Additionally, the build environment has to support some hook for changing the path etc. For example, in Razzle, we use the STATIC\_DRIVER\_VERIFIER to set the path that is picked up by build.exe so that the cl and link and lib executables are the interceptor executables as specified by us. This can be seen in the intercepted build section of the SDV plugin’s configuration files in SmvSdv project.



All the stages of SMV interception.

## Miscellaneous

* Build paths, output folder paths, working directory paths etc. are all extracted by first doing a normal build and analyzing the build output. This method could be improved a bit, but has proven to be good so far. We just handle different types of regular expressions based on cl or cl.exe.
* There is an implicit assumption that link output will also be placed wherever cl.exe output is placed. This is find for MSBUILD, Razzle, and CoreXT, but seems to breakdown sometimes in a complicated makefile based modules such as OpenSSL.
* The build output folder is where all SMV artifacts and output is placed

## Deployment

Under the SMV source folder, there is a deployment folder. The deployment folder contains a smvDeploy.cmd script that can be used to deploy SMV. Usage is as follows:

smvDeploy.cmd configuration directory\_name

The first argument is used to specify the build configuration to use for deployment: release or debug. The second argument specifies the directory where SMV is to be deployed to. The second argument, where SMV is deployed to will be used for %SMV% (has to be set explicitly). Examples of using the script include:

smvDeploy.cmd release %slam%\smv

smvDeploy.cmd release %sdxroot%\tools\analysis\x86\smv

Note that currently the deployment script does not build the SMV product, but just deploys it along with the analysis plugins that exist.

After deployment, an intercept.xml file has to be copied into %smv%\bin. The Intercept.xml sample files can be found in %smv%\interceptorSamples. The name of the file specifies the environment it is written for. Alternatively, you can modify them and then copy them, or create a new one based on the samples.

## Threat Model

Since SMV builds on top of Azure, and there are no services that are being deployed as part of a SMV release.

* The client side application is a simple executable with libraries. The core SMV executable and libraries present no threat or danger.
* The configurable interception technology can potentially be used maliciously by someone if they want to intercept an executable on a system. But, similar technologies have been seen in the past. Something like this can possible be used as malware, but this is not the first time such kind of technology has been released or seen.
* The Azure infrastructure that is in SMV can be used to create services that essentially facilitate a map-reduce style computation. The threat model here is the same as any other such infrastructure – Hadoop, Cosmos, Azure etc.