

# **Core to SAL Translation**

## **Programmer guide**

Gilad Hemi  
Ayelet Bar-Niv Shiloh

## TOC

Core to SAL Translation.....	1
TOC.....	2
1. Using the Program .....	3
Program files.....	3
Working with the program.....	3
2. Implementation .....	5
Platforms.....	5
Assumptions.....	5
General Implementation Concepts.....	5
Class description .....	6
Class Diagram (for the main classes).....	7
Inheritance Diagram.....	8
Additional Info Classes.....	9
3. Programmer Tips .....	9
Flattener .....	9
Program Flow (phase 2).....	9
Xml .....	9
Variable Scope .....	10

Please note that the “core2sal user guide.doc” is sub-set of sections 1 \$ 2.

## 1. Using the Program

### Program files

- coreToSal main directory
  - 📄 Core\_To\_SAL [unix]: is a script that runs phases 1 and 2 to translate a cdl file to a sal xml file.
  - 📄 Makefile [unix]: make file with the options all,pretty (remove all objects) and clean (remove all objects and binaries).
  - 📄 run\_examples[unix]: a script that runs all the examples under the examples folder.
  - 📄 env.txt [unix]: list of all the environment variables that need to be changed for the oracle library will work
  - 📄 settings.xml: the cts setup file (xml file that holds the values of MAX\_INT and MIN\_INT).
  - 📄 cdl.dtd: the schema file for the cdl xml language.
  - bin: Phase 2 binaries.
    - 📄 coreToSal: The executable that runs phase 2.
  - doc: Documentation folder.
  - examples: Full examples (phase1 and 2).
  - Flat: Phase 1 files.
    - CTS\_Flat
      - bin: Phase 1 binaries.
        - 📄 CTS\_Flat: The executable that runs phase 1.
      - CTS\_flat: visual studio 6 workspace for phase 1.
      - hdr: Phase 1 header files.
      - obj: Phase 1 object files.
      - src: Phase 1 source files.
    - CTS\_Flat\_examples: examples for phase 1 only.
  - hdr: Header files for phase 2.
  - linux: utilities for sal.
    - 📄 env.txt [linux]: lists the environment variables needed by sal programs.
    - 📄 runEx [linux]: a script that runs sal-pretty-printer to convert \*sal.xml to \*.sal.
  - obj: Phase 2 object files.
  - src: Phase 2 source files.
  - VCProj: visual studio 6 workspace for phase 2.
  - xml: Oracle xml library files (used by phase 2).

### Working with the program

#### 1. Working with CTS\_Flat executable (phase 1)

The program gets one parameter which is the filename of the input program file cdl.

- 📄 Input file: *filename.cdl*

- 📄 Output file: *filename.cdl\_cdl.xml*: The representations of the input file in xml.
- 📄 Output file: *filename.cdl\_flat\_cdl.xml*: The xml input file after necessary flattening for phase 2. In case that flattening was not necessary, this file exists and is similar to *filename.cdl\_cdl.xml*.
- 📄 Output files: *filename.cdl\_cdl.xml\_#\_log.xml*: There is a log file for each flattened module, to hold the additional information of the flattening process.

## 2. Working with coreToSal executable (phase 2)

When compiled in unix the program uses oracle xml library (can be found at [http://otn.oracle.com/software/tech/xml/xdk\\_cpp/index.html](http://otn.oracle.com/software/tech/xml/xdk_cpp/index.html)). This requires two environment variables pointing to the msg and nlsdata directories in the project e.g.

```
setenv ORA_XML_MSG $HOME/proj/xml/xdk/msg
setenv ORA-NLS33 $HOME/proj/xml/nlsdata
```

Note: from an unknown reason, the program does not compile at CSA environment. There is no problem to compile it at CSD.

When compiled in windows the program uses msxml com component.

coreToSal works with the following flags:

- I:*filename* : input flattened xml full filename ( required ).
- O:*filename* : output flattened xml full filename ( optional ).
- X:*value* : set the MAX\_INT ( optional. default is taken from settings.xml ).
- N:*value* : set the MIN\_INT (optional. default is taken from settings.xml ).

- 📄 Input file: *filename.\*.xml*
- 📄 Output file: *filename\_sal.xml*:The translated program in sal xml format (unless another filename was specified by the -O flag).
- 📄 Output file: *filename\_addInfo.xml*: The changes log file.

## 3. Working with Core\_To\_SAL script ( phases 1 and 2)

the script can receive 1 to 4 parameters. The first parameter is required and states the name of the input cdl file (without any prefix). After that parameter, 3 more parameters are optional, and must have the same prefixes as the coreToSal program parameters:

- O:*filename* : output flattened xml full filename ( optional ).
- X:*value* : set the MAX\_INT ( optional. default is taken from settings.xml ).
- N:*value* : set the MIN\_INT (optional. default is taken from settings.xml ).

The script sets the environment variables; runs phase 1 and then phase 2.

## 4. Working with SAL tools by SRI.

The SRI programs work under Linux environment, and handle SAL xml program specifications.

They require an environment variable that points to the xml folder e.g.  
`setenv SALPATH /local/home/students-workers/gilad_sl/xml`

5. NOTE : The `sal-pretty-printer` can probably work on input xml files of size < 42KB. The `card_game.cdl` example worked only when we reduced the card options from (2..joker) to (10..joker) reducing the `card_game_sal.xml` file size to 41KB.

## 2. Implementation

### Platforms

- Phase 1 is implemented in c, using the CoreParser utils ( in order to use the flattener ).
- Phase 2 is implemented in c++. It uses the standard library classes: string, vector, list and map as containers, and uses the oracle xml library to read the xml files.
- The flattener at CoreParser/utils/flat was changed to save the addition info for flat sub tree and not only for flat program.

### Assumptions

1. The input file should be a correct core file, namely it should have a SYSTEM module, and not have recursive module combinations.
  2. If hold previous flag is off, all arrays are of one dimension (to simplify the array loop).
  3. When a nondeterministic assignment for integer is needed (split transition and hold previous stages), the integer type is replaced by the `ctsInt` type, and is bounded to `MIN_INT..MAX_INT`. These boundaries are given in a xml file called `settings.xml` .
  4. We assume there are no underscores in the modules and transitions names (actually, we can reduce this assumption, and assume that there is no name of a transition as a sub-string in another transition name, that also have an underscore, but we prefer to simplify the assumptions, to prevent the odds of mistakes caused by misunderstanding this assumption).
- All bad inputs generate friendly error messages except for errors in Core2XML stage (this part in the translation is done using Core2XML utility, and not by our code).

### General Implementation Concepts

1. Pointers are implemented by smart pointers (using reference count ) for classes that inherit from `iUnknown`. This is done with the help of the template class `IPtr`.
2. The main classes (Modules, Transitions etc...) have circular references, therefore they are regular pointers that are constructed and deleted only by their container.
3. Each class knows how to translate itself to SAL XML in a member called `toSAL`. It gets a parent XML node, and creates Sal XML of its content. The program class inserts all global scope data, and then calles `toSAL` in each module and so on...

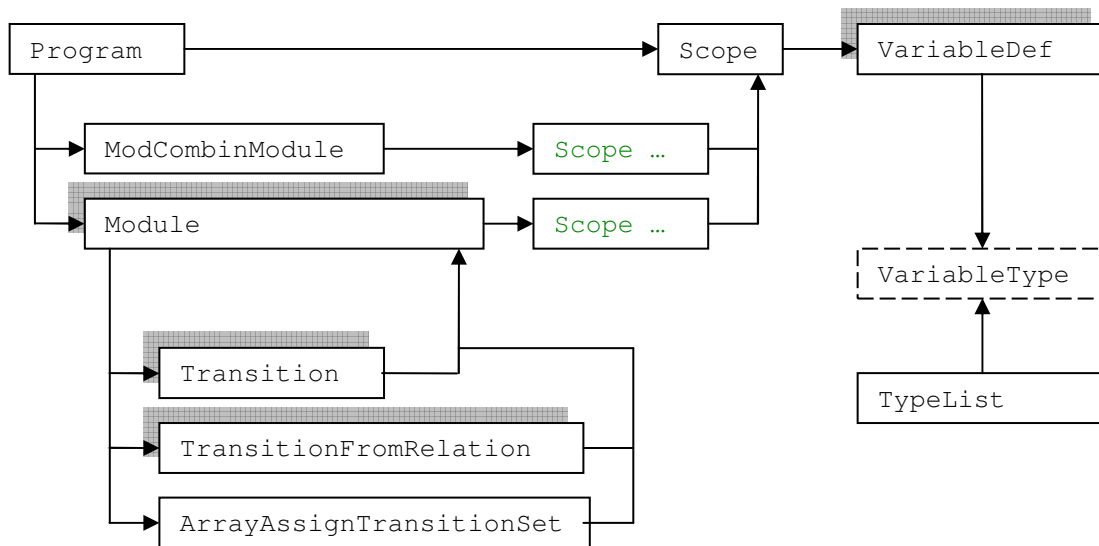
4. Each class keeps record of its origin. Classes that need to save data in add info hold the ID of their sources and hold the creation reason until the end of the program when the additional info is written. Expressions do not have a class and are represented in xml nodes only. If an expression is created by the program,(in the process of relaxing relations for example), the original ID is saved in the additional info change list. When converting the expression to SAL, the program looks in the changes list to see if addition-info binding is needed.

## **Class description**

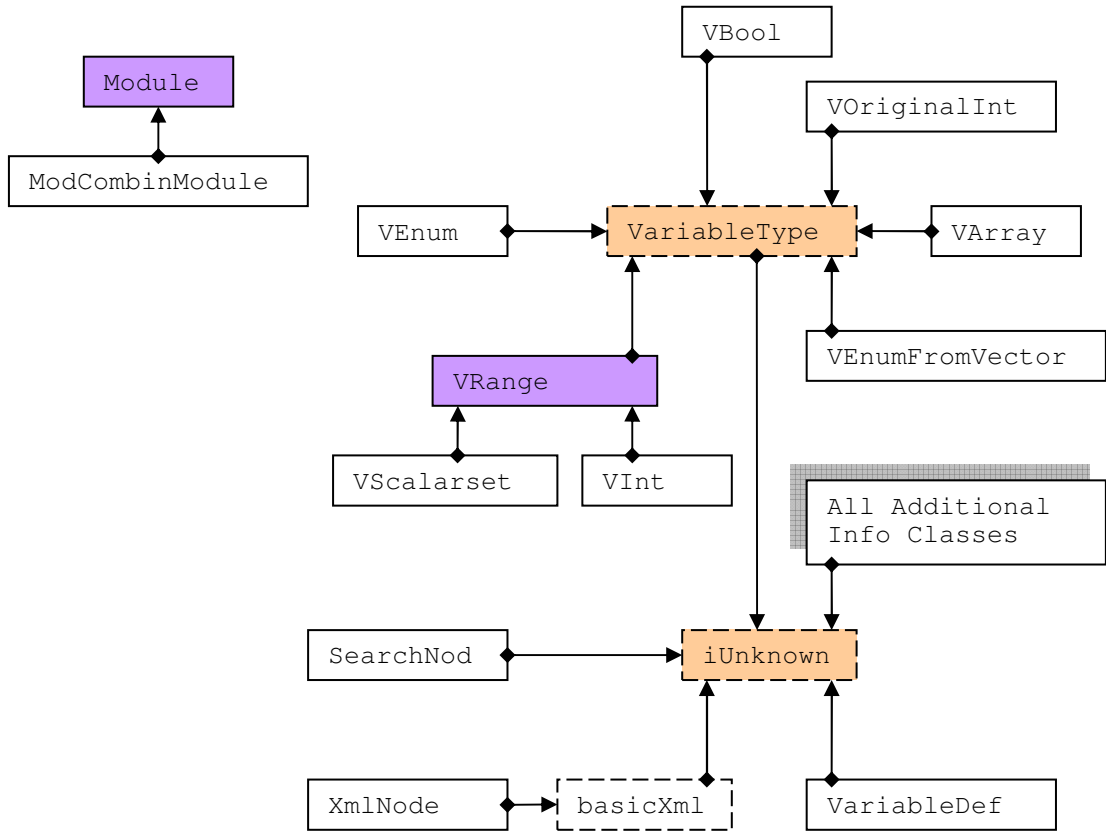
1. Library Classes
  - a. iUnknown is a base class for all classes that use smart pointers.
  - b. IPtr is a smart pointer template class that wraps an iUnknown class.
2. XML Classes
  - a. basicXml A platform depended class for xml.
  - b. XmlNode A wrapper class for an xml element node (not dependent on the platform).
  - c. SearchNode Base class for all xml search classes.
  - d. SearchByTag Used to search an xml node that matches a specific tag name.
  - e. SearchByText Used to search an xml node that matches a specific text.
  - f. SearchCombine Combines 2 search objects (ANDs the criteria of the two searches).
3. Program Structure
  - a. Program Holds information about the global scope such as types, global variables, list of modules etc ...
  - b. Module Describes a module built from transitions.
  - c. ParentModule an interface implemented by Module class so that each Transition can point to it's containing module.
  - d. ModCombinModule Describes a module that is a combination of other modules.
  - e. Transition Describes a transition.
  - f. TransitionFromRelation Describes a new transition that was added because of a relation that could not be relaxed.
  - g. ArrayAssignTransitionSet Describes the new transitions that are added because of hold-previous = false array assignment.
4. Variable scope and Definition
  - a. Scope Describes a variable scope (Global variables, Module variables, or parameters).
  - b. VariableDef A definition of a variable in a scope (name, type and initial value).
  - c. VariableType An abstract class for variable types.
  - d. VArray, VBool, VEnum, VInt, VOriginalInt, VRange, VScalarSet all classes derived from VariableType.
  - e. VEnumFromVector derives from VariableType and is used to create new enumerations that weren't in the original program.

- f. VOriginalInt derives from VariableType and describes an Int type, not like VInt that can describe an Int or a ctsInt type (this is determined in runtime).
  - g. TypeList The list of all types.
5. Additional info classes
- a. Changes The manager that tracks all the changes
  - b. SingleChange a connection between source nodes, destination nodes and the reason for the change.
  - c. ChangeSource the source core node for the change.
  - d. ChangeOrigin the sources of the change and the reason for the change.
  - e. ChangeDestination the target sal of the change.
  - f. ChangeSrcList a list of sources.
  - g. ScopeChange a class that handle changes of variable.

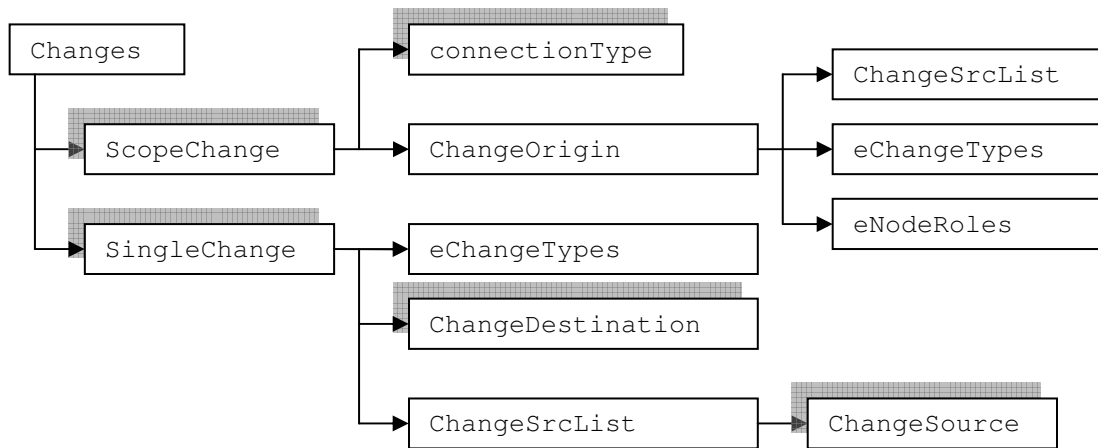
**Class Diagram (for the main classes)**



## Inheritance Diagram



## Additional Info Classes



## 3. Programmer Tips

### Flattener

The CTS Flattener criterion for flattening is a complex combination (see chapter 4). In order to change that criterion, the `flatNodes` function should be drastically changed (or even to be rewritten).

### Program Flow (phase 2)

1. Each step in the translation starts from the Program class, which calls all Modules, which call their Transitions that handle the step (if the step is handled in a Transition level). Expressions do not have a class and are hold as xml nodes.
2. Each class has the responsibility of transforming itself into SAL representation. It also has the responsibility of recording non trivial changes to the additional info log.

### Xml

1. The basicXml library may seem a bit odd in the sense that it implements functions that are supported to be implemented by the XML DOM. The reason for that are some bugs in the oracle library (such as cloneNode function that does not clone children correctly), and a slight difference between the msxml library and the oracle library in the semantics of the methods.
2. The SAL programs require that XML files start with a comment tag, and do not parse the XML otherwise. Therefore we added a comment "`<!-- produced with Core 2 sal-->`" to the SAL XML files we create.

## Variable Scope

A Scope object is a list of variables and definitions regarding a specific scope as parameters, global variables, local variables and more. These scopes change in the translation either due to structural changes or as a result from the difference of the global variables syntax in Core and in SAL. Since it can be tricky to follow these changes here is a list of the scope changes throughout the translation.

1. Xml loading and class initialization:

Program::m\_globalScope ← all global parameters.

Module::m\_parameters ← module parameters.

Module::m\_locals ← module local parameters.

Module::m\_context ← all parameters defined in the context meaning  
m\_parameters+ m\_locals.

Module::m\_transformToGlobals ← Nil. This is a list of all local parameters that will be made global later on in the translation.

2. Partial synchronization splitting of a module. For each generated module we will get:

Module::m\_parameters ← Unchanged.

Module::m\_transformToGlobals ← Module::m\_locals.

Module::m\_locals ← Nil.

Module::m\_context ← Unchanged.

(Since the parameters need to be shared with the other modules generated from the same parent module, they need to be global).

3. Partial synchronization splitting of a module, In the SYSTEM module:

- a. If a module named M, called in an xml node with id ="ID" has a local variable VAR, add to the local scope of the SYSTEM module a variable named M\_ID\_VAR.

- b. We keep remembering that the variable M\_ID\_VAR should be mapped to the variable VAR in the child modules, so when converting to SAL we use variable renaming e.g.

*RENAME M\_ID\_VAR TO VAR IN M\_childModule*

The change is done because all the child modules of M for this specific instance need to share the same variables, but do not need to share them with other instances of the module M.

4. Relation handling: sometimes a temp variable needs to be added with the assignment "temp\_VAR' = VAR". Then temp\_VAR is added to the local module scope.
5. Translation to SAL: variables are split into four classes.  
Local variables stay local.  
The rest of the variables are the m\_parameters, m\_transformToGlobals and the global variable scope. They are grouped to INUT/OUTPUT or GLOBAL definitions depending on the reading and writing to each variable.
6. SYSTEM local variables are made global since there are no local variables to a modCombin module in SAL.