

Core to SAL Translation Documentation

Gilad Hemi
Ayelet Bar-Niv Shiloh

TOC

Core to SAL Translation.....	1
TOC.....	2
1. Abstract.....	3
2. Phase 1: Elimination of Complex Combinations.....	4
The CTS_Flat program	4
Changes in the Flat Utility	6
3. Phase 2: Core to SAL translation.....	7
Handle Defines.....	7
Remove Multiple “ModCombin” Modules	7
Handle Partial Synchronization	7
Relax Relations	8
Split Transitions with relations	9
Handle No Hold Previous	11
Analyze Parameters	12
Translation to SAL.....	13
4. PC - Figures	14

1. Abstract

This document describes the translation from a program written in the Core language to a program written in SAL language by SRI.

Although the main structure of the two languages are similar, some features in Core language need to be converted to a way that can be expressed in SAL.

The translation is composed from two phases.

The main phase is the second one, which translates a Core XML program to a SAL-friendly Core XML program, and then to SAL XML.

Some complex programs can't be handled by phase 2 and need to be flattened.

The first phase flattens these programs.

2. Phase 1: Elimination of Complex Combinations

This phase is a helper phase for eliminating partial synchronization as described in [phase 2](#). Some synchronized combinations can't be expressed by the solution we will introduce. These cases are of the form of partial synchronized combination nested in another partial synchronization, or synchronized combination nested in partial synchronized combination.

This problem is solved using the sub-flattener tool (FlatSubTree function of the flattener). We flat the problematic part of the tree, as demonstrated in diagram 2.

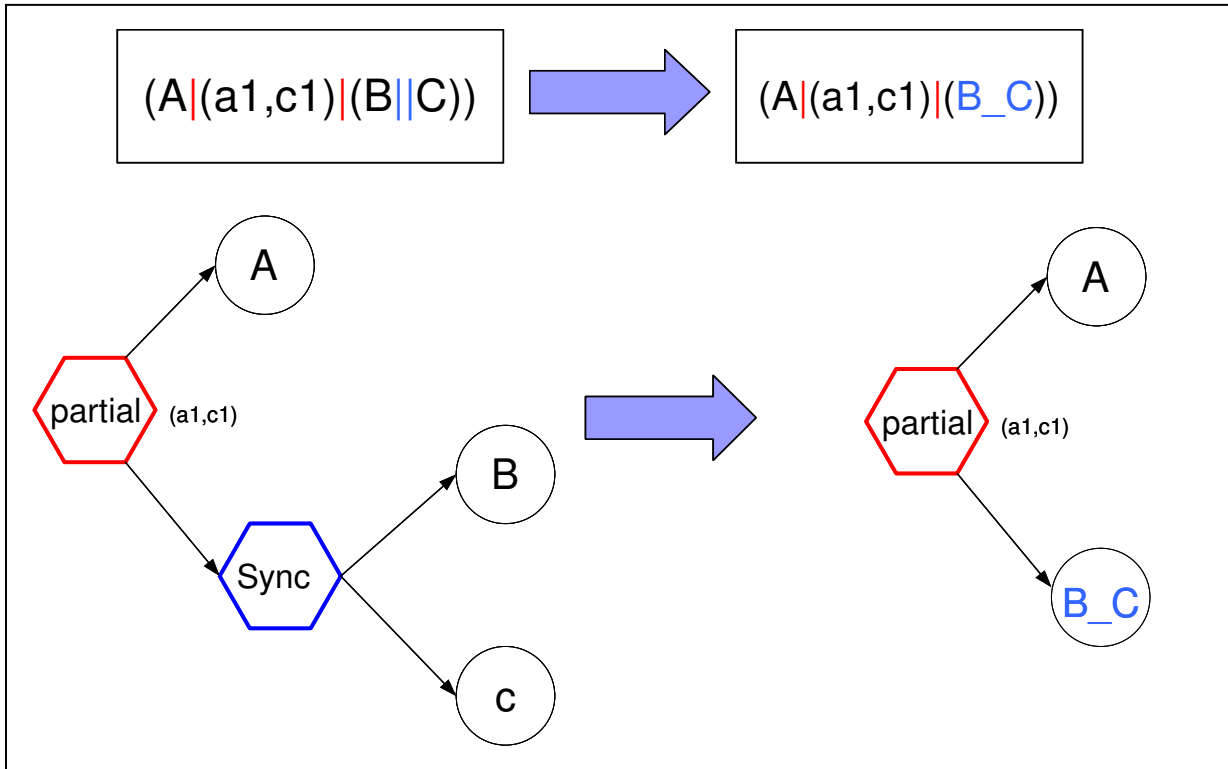


Diagram 2: CTS_Flat program flattening the sub-sync part in a partial sync combination

The flattening is done in the first phase of the CTS translation. This way the second phase can handle only simple partial synchronizations.

The CTS_Flat program

The CTS Program uses the FlatSubTree function of the flattener. First it finds the System Module, and if the type of this module is ModCombin, the flatNodes function is called. This function goes (recursively) over the combinations tree, and finds the combinations we want to "flat". The flatNodes function receives 4 parameters: the ModuleNode (starting from the root of the tree), a Boolean parameter which indicates if synchronized

combination is allowed, and another 2 parameter which is used as a pointer to returned values. The role of this function is to go recursively over the tree and if there is a synchronic or partial-synchronic combination, and the `allow_sync` parameter is false, to call the `FlatSubTree` function. The `allow_sync` parameter becomes false when there is a partial synchronic combination (the recursive call after a partial synchronic node, is done with `allow_sync = false`). The `FlatSubTree` function returns a new module, to replace the sub-tree, but it does not change the original tree. There are still many changes to do in the tree database:

- Replace the sub-tree with a `COMBIN_MODIN` node (a call to a module).
- Update the parameters of the new module.
- Update the pairs in the partial combination.

All these changes are done on the tree data structure. The `analyzeParameters` function is used before the flattening. This function creates a set of all the actual parameters in the sub-tree (that will be flatten), and their types (using `TranslationSpecific` variable of `ActualParameter` structure). After the flattening we take the actual parameters set that was created and insert the parameters (as formal parameters) to the new module that was created by the `FlatSubTree` function. After returning from the recursive call – if the sub-tree was flattened, the tree structure is updated with the change. A new node is created and the old node is added to a list of nodes, to be deleted later*. The last update that is done on the tree structure is the update of the sync pairs (on the partial synchronic combination node). The pair list should be changed according to the changes in the transition list made by the flattener.

After the `flatNodes` function is finished and full the tree was scanned, the `CoreProgram` data structure should be updated as well. Before the tree structure is translated to the `CoreProgram` structure, there are some analyses to do. For each module of `modcombin` type, we saved a small structure of type `ModuleUsage` (we used `TranslationSpecific` field of the `Module` structure). On this structure there are 2 fields - `call_count` and `changes_count`. The second field counts the number of changes done in the module combination (e.g. if there is only 1 module of `modcombin` type, and 2 changes were done, the `changes_count` of this module will be set to 2). The first field is counting the number of calls to that module. When creating the `CoreProgram` data structure from the tree, we need to know if the module should be cloned. Cloning is necessary when a module was changed (once or more), and there is more than one call to the module. In this case it is possible that a different changes are made for each call to the module.

On this stage we use the data we collected and saved in the `ModuleUsage` structure.

Another point to take into consideration while rebuilding the `CoreProgram` structure, is that the tree is build as 1 block, while the `CoreProgram` might be composed of several `moducombin` modules. On the `updateModulesCombinations` function, the division to the modules is done by comparing the module owner of each node, to the module owner of its sons.

After rebuilding the `CoreProgram` structure, the `CdIDs_2_XML` tool can translate the partial flattened core program into a XML file, which is used as an input to the main `Core` to SAL translator.

I would like to emphasize the fact that this program was written as a part of the CTS translation. From this reason some things might look “unfinished”. For an example, we

do not remove unused modules. The reason for that is that it is done in the main CoreToSAL program. Another thing is that this function is specific for relaxing partial synchronization. It can be the basis for other relaxations, but major changes should be made.

Changes in the Flat Utility

The Flat utility creates an xml log file, for the additional info. Due to a bug, the FlatSubTree function did not have the essential parameters in order to create this file, so any call to that function ended with a segmentation fault. We added these parameters, and made the necessary changes in the function. Unlike the FlatCoreProgram function, where the input parameter is the Input CDL File name and the log filename is created by parsing it, in the FlatSubTree function we used the log file name as a parameter. The reason for this difference is that there are cases when we call FlatSubTree more than once (with the same input file), and we want different log files.

3. Phase 2: Core to SAL translation

The translation is done in several steps:

1. Load XML
2. Handle Defines
3. Remove Multiple “ModCombin” Modules.
4. Handle Partial Synchronization
5. Relax Relations
6. Split Transitions with Relations
7. Handle No Hold Previous
8. Analyze Parameters
9. Translate to SAL (in XML format)

All the steps (except for the last one) are done without translating the XML to SAL syntax.

The Idea behind these steps is to bring the Core input to a form which can be expressed in SAL; so on the last step, only syntactic translation is done.

Handle Defines

There isn't any macro definition (DEFINE) and constants in SAL.

On the “handle defines” step, we search for all the occurrences of the macro name and replace them with the macro expression.

Remove Multiple “ModCombin” Modules

This step is done to enable the analyses of the following steps regarding.

A modCombin module is a module that is a combination of other modules and does not have any transitions. If the SYSTEM module is not the only modCombin, then all the other modCombin modules are integrated into the SYSTEM module. For instance:

```
MODULE BUF_REC(x:elem , y:elem) {           ⇔  MODULE SYSTEM() {
    (BUFFER(x,y) || (RECEIVER(y)))          VAR
}                                           s: elem;
                                           t: elem;
                                           (SENDER(s) || (BUFFER(s,t) || (RECEIVER(t))))
MODULE SYSTEM() {
VAR
s: elem;
t: elem;
(SENDER(s) || BUF_REC(s,t))
}
```

Notice that the parameters change from x and y to s and t.

Handle Partial Synchronization

In the Core language we have 3 kinds of combinations between modules: **Synchronic combination**, **A-Synchronic combination** and **Partial Synchronic combination**. In SAL language there is no **Partial Synchronic combination**.

We simulate partial synchronization by sync and a-sync combinations as demonstrated in the following diagram:

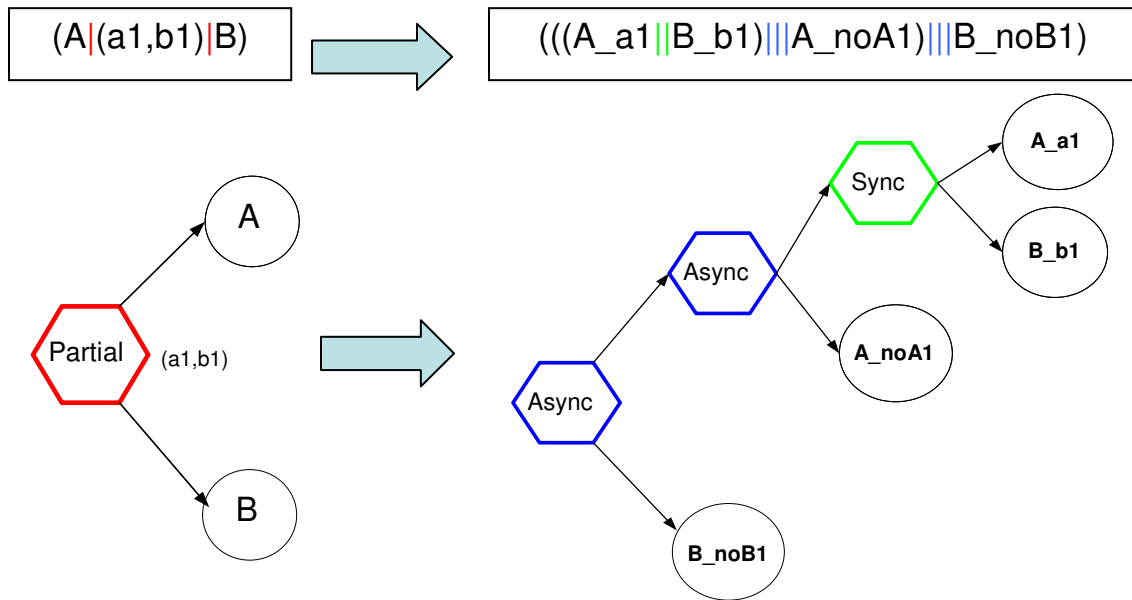


Diagram 1: translating simple partial synchronic combination

This solution works for most of the partial synchronic combinations, but we still needed to solve more complex combinations, meaning partial synchronized combination nested in another partial synchronization, or synchronized combination nested in partial synchronized combination. These complex combinations are handled in [phase 1](#) (described earlier) so by getting to phase 2 there are no complex combinations.

Relax Relations

The transition language in SAL contains only enable (guard in SALs terminology) and assign (assignment) sections, namely it does not contain a relation section. In this step we try to eliminate relations by splitting them into terms, and moving each term to the enable or assign statements if it can be expressed in the terms of the corresponding sections.

For instance let examine transition R1:

```
TRANS R1:
  enable: !b;
  assign: b: = true;
  relation: (i = 0 /\ i' = 2*i);
```

This transition has 2 terms in its relation section: $i = 0$ and $i' = 2*i$.

The first term does not have any next-state variables in it; therefore it does not depend on the assignments in the transition (and assignments in transitions that run in the same time). Thus it can be moved to the enable section without changing the meaning of the program.

The second term is an equality that can be interpreted as an assignment $i' := 2*i$.

The transition would now be:

```
TRANS R1:
  enable: !b /\ i = 0;
  assign: b' := true;
         i' := 2*i
```

The rules for moving a term to the enable section is:

- There are no next-state variable references in the term.

The rules for moving a term to the assign section are:

- A term of the form (b') when b is a Boolean variable will create the assignment: $b' := true$.
- A term of the form $(!b')$ when b is a Boolean variable will create the assignment: $b' := false$.
- A term of the form $(x' = expr)$ will create the assignment $x' := expr$ under the following restrictions:
 - $expr$ is an expression that does not contain any next-state variables.
 - x' does not have any other assignment in the transition.

Other rules may be applied, but where not implemented. One option for example:

A term of the form $i' = expr(X_1' \dots X_n')$ when variable i does not have an assignment and each next-state variable X_j in the expression $expr$ is assigned with a value. This would result an assignment $i' := expr(V_1' \dots V_n')$ where V_j is the value assigned to X_j .

When a term does not fall in each of the categories that are handled, we split the transition.

Split Transitions with relations

When a transition has a relation with a term that can't be relaxed we split the transition into two parts: We first guess a state and then check it to see if it satisfies the relation condition. We use guessing because relations semantics hold an assignment in them, for example:

Core Original

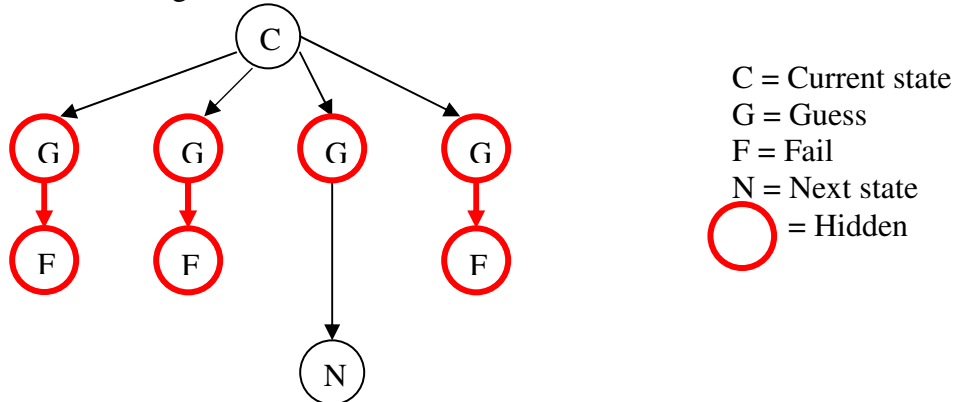
```
TRANS One:
enable:  i > j;
relation: i < j';
```



Core Equivalent

```
TRANS One:
enable:  i > j;
assign: j' := {any value bigger than i can come here};
```

Since we do not know the value of variable i in compile time, we need to guess a value for j from the whole range, and only in run time check if it is consistent with the relation. The state diagram is:



The original transition is split into two new transitions. One part holds the enable and assignment sections of the original transition, and the second part holds the relation section of the original transition. The second transition checks the relation, and if it is ok it returns to the program, otherwise the program goes to a fail state. For example:

Core Original

```

HOLD_PREVIOUS
TRANS One:
enable:  i > j;
relation: i' < j';
  
```

SAL Output

```

↔ 1  One :
    2    i > j  AND PC = pc_normal -->
    3      j' = {A NON DETERMINISTIC ASSIGNMENT};
    4      i' = {A NON DETERMINISTIC ASSIGNMENT};
    5      PC' = pc_rel;
    6      PCrel' = 0

    7  relation_One:
    8    PC = pc_rel  AND PCrel = 0 -->
    9    PCrel' = -1;
   10    PC' = IF i < j THEN pc_normal  ELSE
   11    pc_fail  ENDIF
  
```

A PC is used to control the program flow. Some of the available states are `pc_normal`, `pc_rel` and `pc_fail`.

`pc_normal` is the only non-hidden state. It means that we are in a transition that has a similar equivalent in the original program. All original transitions check for `PC = pc_normal` (line 2)

`pc_rel` (lines 5,8) is used for transitions that were added to handle relations. Each relation gets a number, and variable `PCrel` holds the number of the relation to execute (lines 6,8). This is not part of the PC to simplify a case were hold-previous is false which will be described later on.

`pc_fail` state (line 11) is used when the guess was wrong, and this program-state is not legal for the program.

[Figure 1 in section 7](#) illustrates the pc states for relations.

Each next-state variable in the relation that does not have an assignment in the original transition gets a non-deterministic value (lines 3,4). When we insert a non-deterministic assignment for integers, we need to specify the whole range. “ctsInt” is a new type that we add, so the integer set can be determined by the user (see “Using the Program”).

Sometimes a new temp variable is needed for the relation. For example

```
TRANS One:
enable:   i > j;
assign:   i' := 65;
relation: i < j';
```

The relation depends on ‘i’ which is the current value of variable i. The variable is changed in the transition, so the current value of i in the relation-transition is not the current value of i in the first transition. Therefore the example will be translated to:

```
Two :   i > j  AND PC = pc_normal -->
        temp_i' := i;
        j' = {A NON DETERMINISTIC ASSIGNMENT};
        i' = 65;
        PC' = pc_rel;
        PCrel' = 1

relation_ Two: PC = pc_rel  AND PCrel = 1 -->
               PCrel' = -1;
               PC' = IF temp_i < j THEN pc_normal ELSE pc_fail ENDIF
```

Handle No Hold Previous

In SAL, for a specific transition, all variables not mentioned in the assignment section retain their previous values (in core this is the situation only when HOLD_PREVIOUS is present). If HOLD_PREVIOUS flag is false, we need to assign a nondeterministic value to each variable that does not have an assignment in all non-hidden transitions. For that we need to specify the whole range of values for each variable. All variable types besides array, Booleans and Integers should be defined in the program, so we know the range of values they get. Booleans are of the known range {TRUE,FALSE}. If an Integer is about to get a non-deterministic assignment we need to bound its range. Instead of the integer type we insert a new type “ctsInt”. This way the integer set can be determined by the user (see “Using the Program”).

For example the module containing transition One has x and y of type Integer in the scope

```
TRANS One:
enable:   x > 0;
relation: x' = 2;
```

We get a module with x and y of type ctsInt in the scope

```

One : x > 0 AND PC = pc_normal -->
      x' = 2;
      y' = {MIN_INT, ..., -2, -1, 0, 1, 2, ... ,MAX_INT}

```

When a module has arrays in its scope each transition is first executed in two phases. First the original transition should be executed. A list of all array indices that are assigned a value is made. Each index of the array that does not have an assignment will get an assignment in the next transition. The module has an array assignment transition for each array in the scope since arrays may have different sizes. The indices are checked in a loop to see if this index of the array was changed and if not it is assigned a value. To do this the following variables are added:

- For each array $a[0..n]$ of type T , a new array $a_hparr[0..n]$ of type Boolean is added. This will keep track of array indices that were changed. The array is initiated to FALSE. For each assignment to $a[i]$ we add an assignment $a_hparr[i] := TRUE$. In the loop we restore all the entries to FALSE.
- For each array in the scope a state is added to the PC. This way we control the program flow: first the transition, loop first array, loop second array etc... Figure 2 illustrates the PC states. Figure 3 illustrates the PC states when there is a transition that has a relation in the module.
- The variable $pc_arrIndex$ is added to control the loop for each array transition.

The example file HP.sal shows the different solutions when hold previous is false.

NOTE:

- Another possible way (which we did not implement) to handle the absence of no HP for simple variables has a similar structure to the way we handled the arrays. It can be done in the following way:
 1. Before executing a transition t execute t_before , where t_before is a new transition that saves all current-state variables in transition t into temporary variables.
 2. Execute HP_M transition which is a transition for each module that assigns a nondeterministic value to all the variables in the scope.
 3. Execute transition t and use the temporary variables instead of the current state ones.

This method will produce a smaller program than the method we implemented if the number of variables changed in each transition \ll the number of variables in the scope.

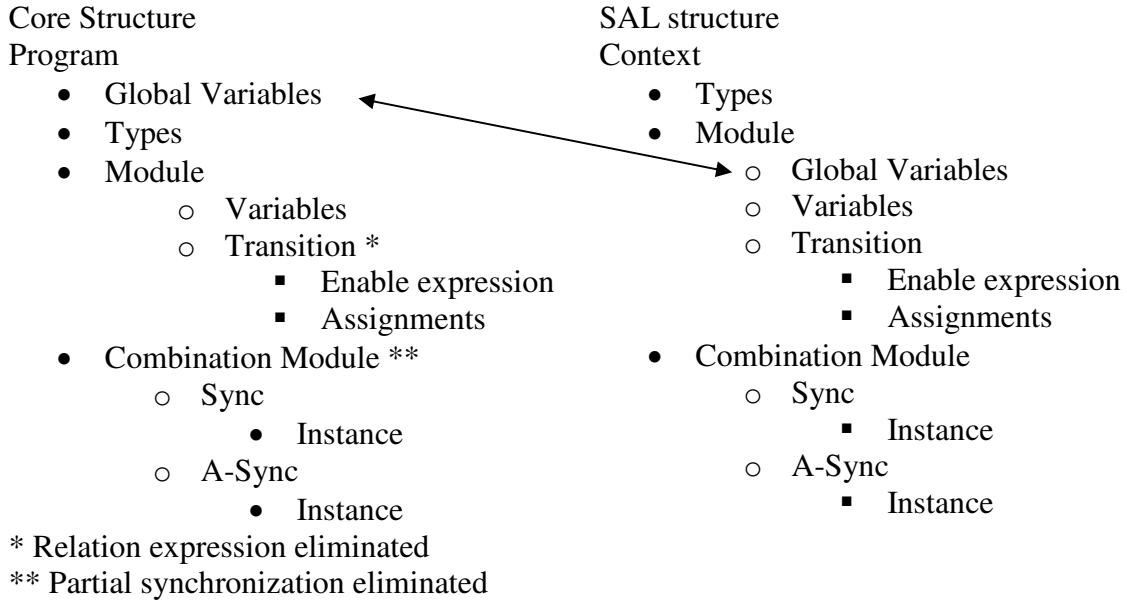
Analyze Parameters

SAL treats parameters in a different way. The global variables are made local for the SYSTEM module. An init module is added to initialize all global variables. The Parameters of a module are converted to In and Out variables since in SAL the parameters are read-only.

Translation to SAL

In this point we have a Core program that can be represented in SAL.

The structure of a program is similar in SAL and in Core, so the translation should be now strait forward:



4. PC - Figures

Fig 1. PC States When relations are present

Notes:

1. Each relation has a relationId.
PCrel is another pc variable that holds the relationId to execute.

2. PCrel is set to -1.

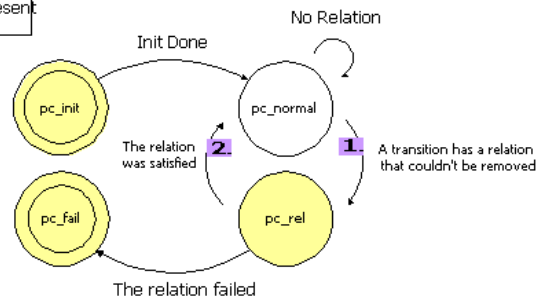


Fig 2. PC States When HP=false and there arrays in the scope

Notes:

If a module M has N+1 arrays in it's scope (the module's scope + the global one)
The states **hp_arr_M_0** ... **hp_arr_M_1** are added to the pc.

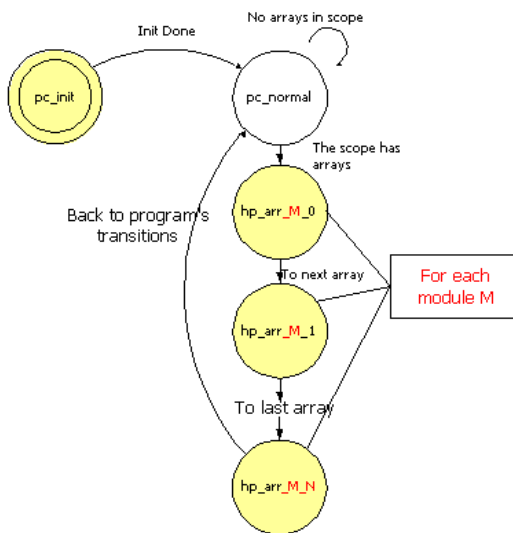


Fig 3. PC States When HP=false, there arrays in the scope and there is a relation in at least one transition in the module

Notes:

First we set the arrays (as in Fig. 2) and only then check the transition (as in Fig. 1).

1. Each relation has a relationId.
pc_relationId is another pc variable that holds the relationId to execute.

2. PCrel is set to -1.

