

# Core to SMV Translation

## Version 2.0

Student Project at SSDL Laboratory, CS Faculty

Alon Zvirin

Supervised by  
Prof. Shmuel Katz  
Mr. Shahar Dag

Previous Versions:

Core2Smv 1.0, Efrat Shabtai, 2002

Core2Smv 1.8, Yoav Gur & Alon Zvirin, 2005

Last Updated: 15/06/2006

# Table of Contents<sup>1</sup>

1. INTRODUCTION .....	6
2. TRANSLATION STAGES .....	7
2.1. CORE2SMV STAGES .....	7
3. TRANSLATION FEATURES .....	9
3.1. CORE DATA STRUCTURE .....	9
3.2. SMV DATA STRUCTURE .....	13
3.3. HANDLING DEFINES AND CONSTANTS .....	15
3.4. HANDLING TYPEDEFS AND VARIABLE TYPES .....	16
3.5. EXPRESSIONS .....	18
4. FROM XML TO CORE DATA STRUCTURE .....	19
4.1. MAIN PARSING STAGES .....	19
4.2. ADAPTING TO MODIFICATIONS IN THE DTD .....	20
5. PRE-TRANSLATION .....	21
5.1. ELIMINATING PARTIAL SYNCHRONIZATIONS IN A CORE PROGRAM .....	22
5.1.1. Abbreviations .....	22
5.1.2. Grammar .....	23
5.1.3. General Description .....	23
5.1.4. Basic Algorithm .....	23
5.1.5. Pseudo-Code for the Basic Algorithm .....	26
5.1.6. First Improvement .....	29
5.1.7. Second Improvement .....	32
5.1.8. Refinements .....	34
5.1.8.1. Empty Modules .....	34
5.1.8.2. Variable Uniqueness and Variable Relocation .....	34
5.1.8.3. Transition Renaming .....	35
5.1.8.4. Book-Keeping .....	35
5.1.9. S Type MC Under P Type .....	36

---

<sup>1</sup> It is recommended to read this document electronically, enabling usage of cross references and hyperlinks.

5.2. FLATTENING .....	37
5.2.1. Pseudo-code for Flattening (First Version) .....	38
5.2.2. Merging of Transitions .....	39
5.2.3. Resolving Name Ambiguity .....	39
5.2.4. Passing of Arguments .....	39
5.2.5. The Cojoin Expression .....	40
5.2.6. Flattening the Whole Core Program .....	41
5.2.7. Naming the Flattened Module .....	41
5.2.8. Pseudo-code for Flattening (Final Version) .....	42
6. TRANSLATION OF A CORE PROGRAM .....	44
6.1. MAIN TRANSLATION STEPS .....	44
6.2. TRANSLATION OF A CORE NON-LEAF MODULE .....	45
6.2.1. Translation Procedure .....	46
6.3. TRANSLATION OF A CORE LEAF MODULE .....	47
6.3.1. State to State Transitions in Core .....	47
6.3.2. State to State Transitions in Smv .....	48
6.3.2.1. Transitions and Variable Assignments .....	48
6.3.2.2. Post-Conditions .....	50
6.3.2.3. Termination States .....	51
6.3.2.4. Avoiding Invalid States .....	52
6.3.3. Format of Leaf Modules (Intermediate Version) .....	53
6.3.4. Out of Bounds Check .....	54
6.3.5. Array Element Assignment .....	55
6.3.6. The Hold Previous Flag .....	56
6.3.7. Avoiding Incorrect Assignments .....	57
6.3.8. Translation Procedure .....	59
6.3.9. Format of Leaf Modules (Final Version) .....	60
7. FROM SMV DATA STRUCTURE TO XML .....	62
7.1. MAIN PARSING STAGES .....	62
7.2. ADAPTING TO MODIFICATIONS IN THE DTD .....	63
8. GATHERING ADDITIONAL INFORMATION .....	64
8.1. THE PURPOSE OF INFORMATION GATHERING .....	64
8.2. THE PROCESS OF INFORMATION GATHERING .....	64
8.3. INFORMATION DOWNLOADING .....	65
8.4. TYPES OF INFORMATION ITEMS .....	67
BIBLIOGRAPHY .....	69

APPENDIX 1 - TRANSLATION OF IDENTIFIERS .....	70
1. NAMING CONFLICTS .....	70
2. RESERVED NAMES .....	70
3. SYNTAX RESTRICTIONS.....	71
4. RESOLVING SYNTAX RESTRICTIONS AND RESERVED NAMES .....	71
APPENDIX 2 - SIMPLIFYING SMV EXPRESSIONS .....	72
APPENDIX 3 - DOES TREATMENT OF GLOBAL VARIABLES AFFECT THE SYNCHRONIZATION STRUCTURE? .....	73
1. OUTLINE .....	73
2. CURRENT TREATMENT OF GLOBAL VARIABLES.....	74
3. EXAMPLE – CASE STUDY .....	74
4. POSSIBLE SOLUTIONS.....	75
APPENDIX 4 – PROGRAMMING NOTES .....	77
1. RELEVANT FILES .....	77
2. SCRIPT FILES .....	78
3. MAKEFILE .....	78
4. MANIFEST FILES .....	78
5. CONFIGURATION FILE .....	79
6. PACKAGES & SOURCE FILES .....	79
6.1. translation package.....	80
6.2. coreDataStructure package.....	80
6.2.1. coreDataStructure.coreExpression package.....	81
6.2.2. coreDataStructure.coreType package .....	82
6.3. smvDataStructure package .....	82
6.3.1. smvDataStructure.smvExpression package .....	82
6.3.2. smvDataStructure.smvType package.....	83
6.4. addInfo package .....	83
6.5. utilities package.....	84
7. ERRORS, EXCEPTIONS & WARNINGS .....	85
8. PRODUCING JAVADOCS .....	85
9. DELICATE PARTS IN THE IMPLEMENTATION .....	86
9.1. Pre-translation .....	86
9.2. Cloning Core Objects .....	86
9.3. Additional Information.....	87
10. UNDERSTANDING THE IMPLEMENTATION.....	87
APPENDIX 5 - EXTERNAL TOOLS .....	88
1. CORETOXML.....	88
2. XML2TXT .....	88
3. XMLVIEWER .....	89
4. SMV MODEL VERIFIER .....	89

APPENDIX 6 - FUTURE PROSPECTS.....	90
1. PARTIAL SYNCHRONIZATIONS.....	90
2. ADDITIONAL INFORMATION .....	90
3. PROBLEMS IN THE CORETOXML APPLICATION .....	91
4. PROBLEMS IN THE XML2TXT APPLICATION .....	91
5. GLOBAL VARIABLES AND SYNCHRONIZATION .....	92
6. SIMPLIFYING SMV EXPRESSIONS.....	92
7. RESOLVING NAMING CONFLICTS.....	93
8. RELEASE & DEBUG VERSIONS .....	94
APPENDIX 7 - EXAMPLES.....	95
1. CURRENTLY EXISTING EXAMPLES.....	95
2. FILE NAMING CONVENTIONS .....	96
3. EXAMPLES OF CORE / SMV PROGRAMS.....	97
3.1. Simple Example no. 2 .....	97
3.2. Dining Philosophers Example .....	100
3.3. Buffer Example .....	105
ACKNOWLEDGMENTS .....	114

# 1. Introduction

This project is done in the VeriTech framework and is part of a system of translations between several specification languages. The translation receives as input a cdl program (in either textual or XML form) and produces an equivalent smv program (in both textual and XML form).

Cdl (core design language, core definition language) is a language developed at VeriTech, serving as junction when translating between other languages.

Smv (symbolic model verifier) is a commonly used language, especially for model-verification and verifying specifications.

During the translation, an "additional information" file is also produced, containing links between elements comprising the source program and corresponding elements in the target program.

A "XML Viewer" device, also developed at VeriTech, enables simultaneous viewing of linked objects from the input file, output file and additional information file.

Two previous translations from core to smv exist:

- Core2Smv 1.0, Efrat Shabtai, 2002.
- Core2Smv 1.8, Yoav Gur & Alon Zvirin, 2005.

The present version (Core2Smv 2.0) is based on the 1.8 version. The main modifications and improvements are:

- Elimination of most cases in which partial synchronization was previously treated by flattening.
- Extensive testing, including usage of a smv model verifier, to check both syntax and logical specifications.
- Extension of the additional information, to include "regular" information as well.
- Correction of faulty logic in the previous versions, especially issues concerning variable assignments and state to state transitions
- Correction of various mistakes and bugs in the implementation.
- Upgrading the implementation to Java 1.5.
- Creating stylesheets (.xsl files) for smv and the additional information.

## 2. Translation Stages

The translation is implemented as a sequence of the following programs ([See also Figure 1 - Program Flow](#)):

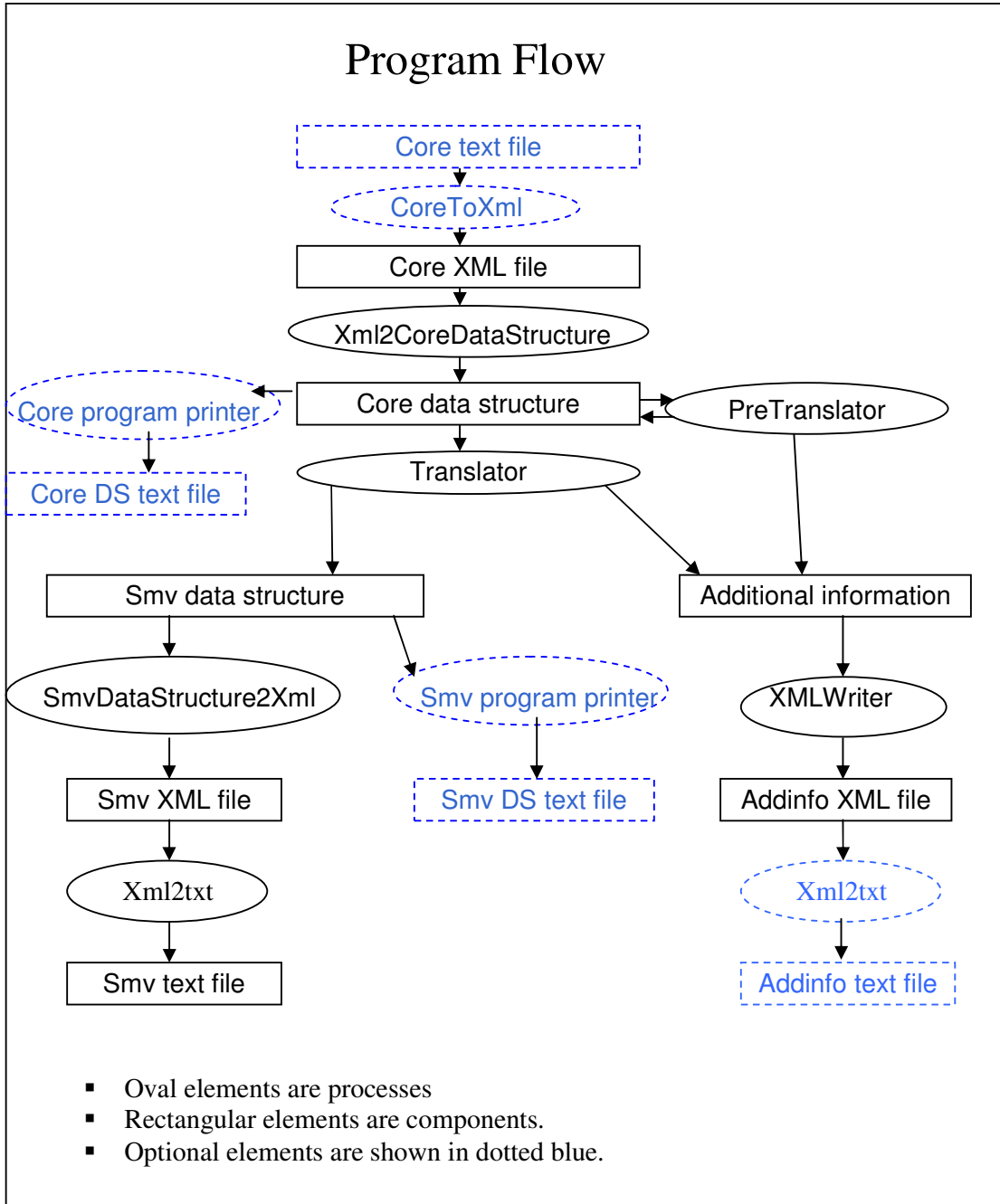
1. CoreToXml - a program getting as input a core program and producing a XML version of the same program. An implementation already exists.
2. Core2Smv – a program getting as input a core program in XML format and producing an equivalent smv program in XML format. Additional information gathered during the translation is also written in XML format.
3. xml2txt – a program getting as input a XML file and a stylesheet, and downloads the contents of the XML file to text. An implementation already exists.

These parts are independent and are linked by a script file.

### 2.1. Core2Smv Stages

1. Building of a core data structure representing the XML version of the core program.
2. Processing of the core data structure. Elimination of partial synchronizations. Flattening if needed (or if desired). Gathering of additional information.
3. Translating the core data structure to a smv data structure. Gathering of additional information.
4. Creating a XML version of the smv program from the smv data structure.
5. Creating a XML version of the additional information collected during the translation.

Options for downloading the data structures to text files are also available. The "CoreProgramPrinter" and "SmvProgramPrinter" are independent of other printing components used in the translation, and were helpful during the development and testing stages.



**Figure 1 - Program Flow**

## 3. Translation Features

### 3.1. Core Data Structure

The core data structure is the data base containing and managing the translation's representation of a core program.

Components of the core data structure represent elements of a core program. Each component contains an identifying string, according to the ID allocated to it in the XML version of the core program. This identification is necessary for the gathering of additional information.

The core data structure is generic, covering all aspects of a core program. Therefore it can be considered as an independent component, ready for use in other applications (translations from / to core language, and processing of a core program).

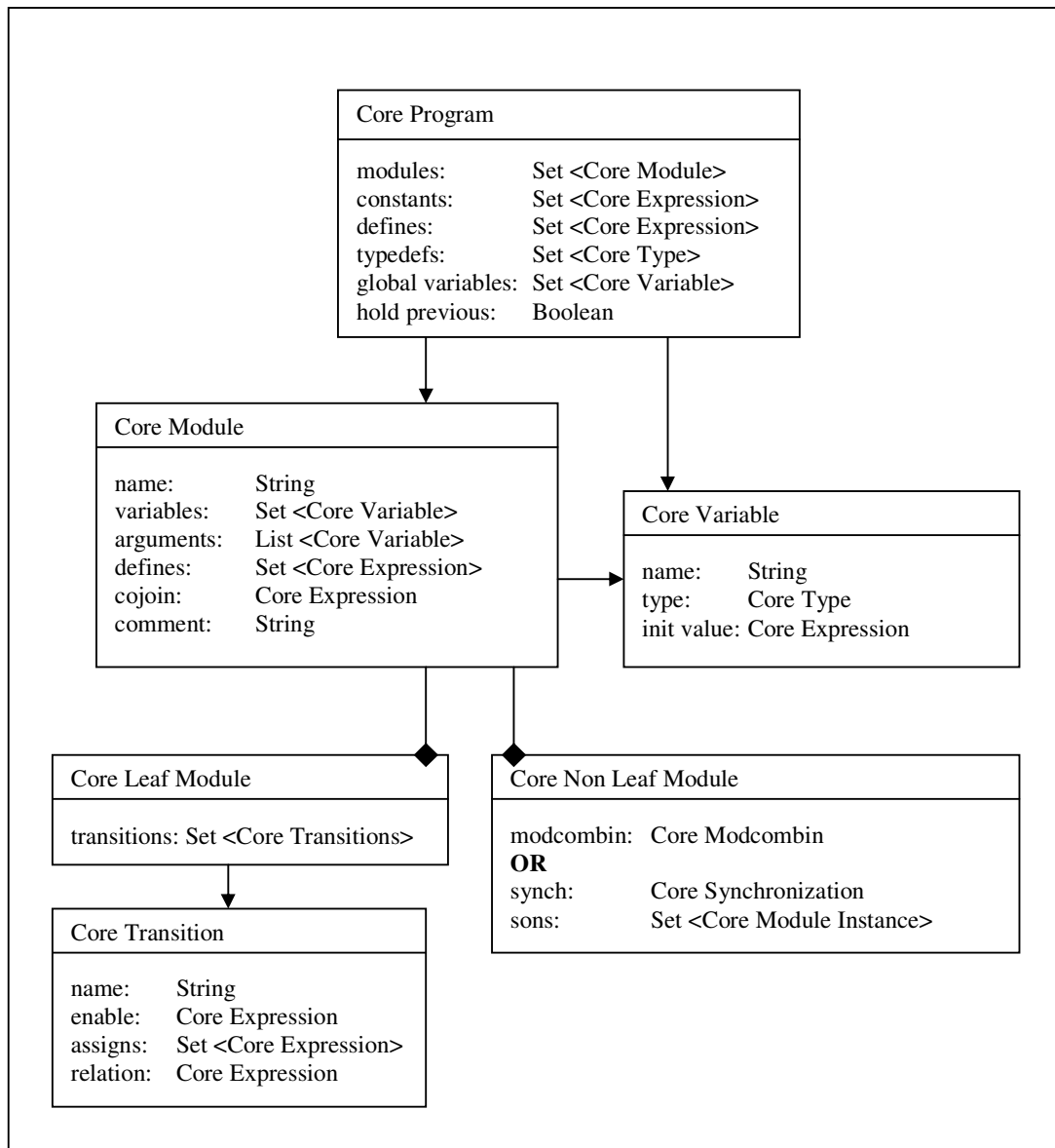
The components of the core data structure are (see also Figure 2 - Core Data Structure and Figure 3 - Core Modcombins and Synchronizations):

- A core program, containing:
  - A set of core modules.
  - Global constants and define declarations.
  - Global type definitions.
  - Global variables.
  - A 'hold previous' flag.
- A core module contains:
  - A name
  - A set of variables.
  - A list of arguments.
  - A set of DEFINE declarations.
  - A cojoin expression (optional).
  - A comment (optional).
- A leaf module (extending module) contains also:
  - A set of transitions.
- A non-leaf module (extending module) contains a sub-tree of child modules. This sub-tree can be represented in two different ways:
  1. A core modcombin structure, similar to core language grammar. (Modcombin is short for module-combination.)
  2. A synchronization type and a set of son module instances. Each module instance references a child module. Child modules may have their own son instances.

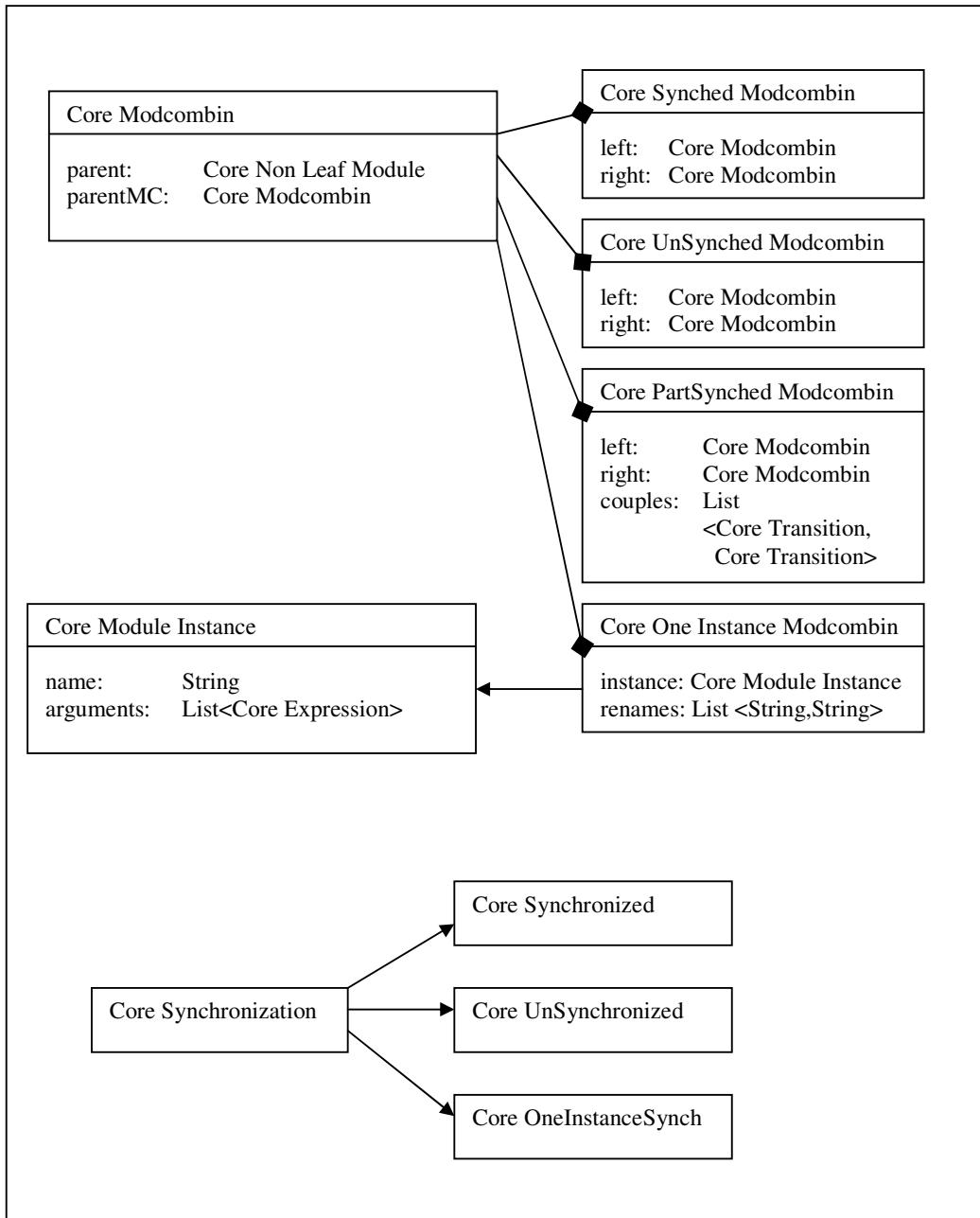
The reason for the two different representations is that the first one is comfortable when parsing the XML file and filling the contents of the core data structure; the second representation is comfortable when translating the components to their smv counterparts.

- A core modcombin is one of the following types:
  - Synchronized modcombin, containing two child modcombins.  
(Representing A || B)

- Un-synchronized (interleaved) modcombin, containing two child modcombins.  
(Representing  $A \parallel B$ )
- Partially-synchronized, containing two child modcombins and a list of synchronized transition couples.  
(Representing  $A \mid (a1,b1) (a2,b2) \mid B$ )
- Instance modcombin, containing:
  - A core module-instance.
  - A list of transition renames (representing  $A (p1,p2) [t1 \rightarrow t2]$ ).
- A core synchronization is one of the following types:
  - Synched – signifying full synchronization over child modules.
  - Un-synched (interleaved) – signifying complete a-synchronization over child modules.
  - One-instance synch – signifying a single child module.
- A core module-instance contains:
  - A name of the referenced module.
  - A list of parameters passed to the referenced module.
- A core variable contains:
  - A name.
  - A type.
  - An initial value (optional).
- A core transition contains:
  - A name.
  - An enable expression.
  - A relation expression.
  - A set of assignments (an assignment is an expression assigned to next-state value of a variable).



**Figure 2 - Core Data Structure**



**Figure 3 - Core Modcombins and Synchronizations**

## 3.2. Smv Data Structure

The smv data structure is the data base containing and managing the translation's representation of a smv program.

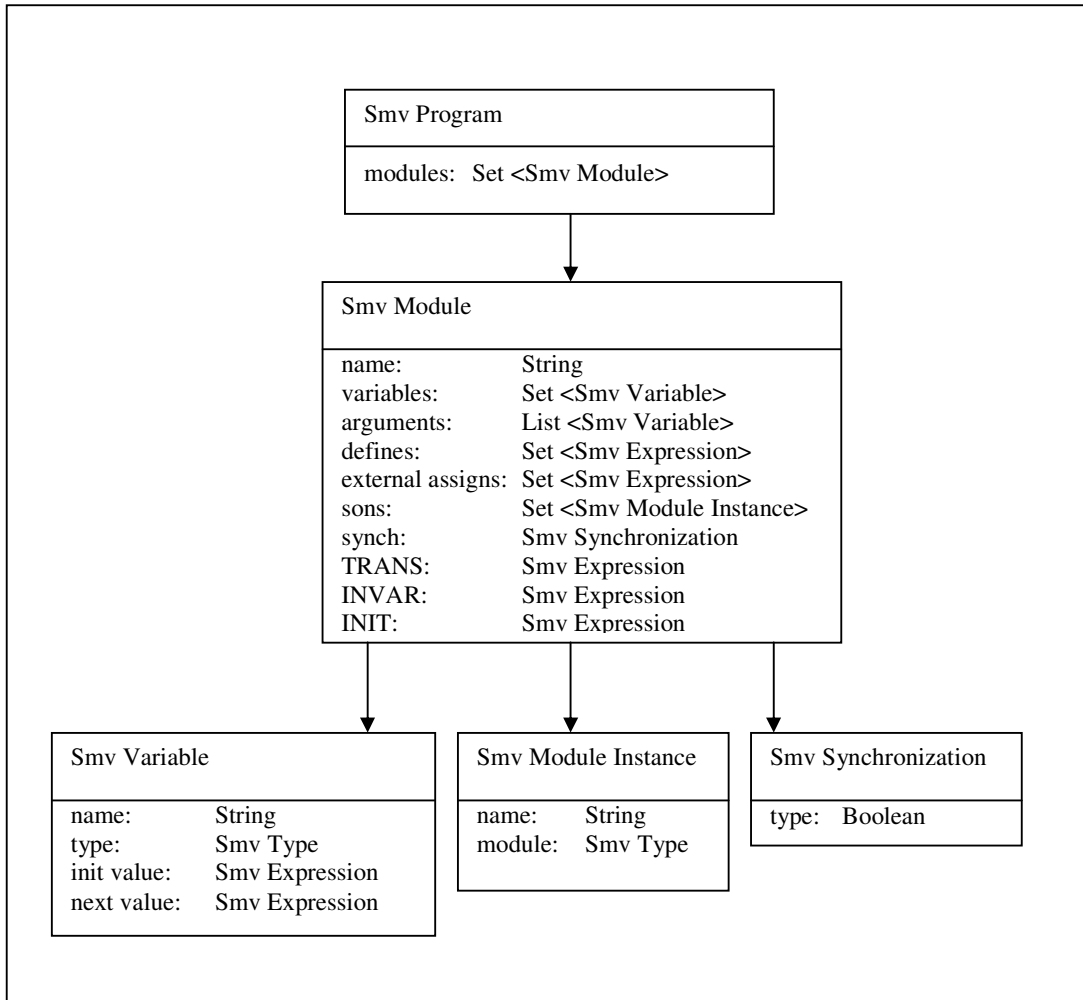
Components of the smv data structure represent elements of a smv program. Each component contains an identifying string, serving as ID in the xml version of the smv program. This identification is necessary for the gathering of additional information.

The smv data structure does not completely cover all aspects of a smv program, only those needed for a translation from core language. The smv declarations 'SPEC', 'FAIRNESS' and 'PRINT' are not represented in the smv data structure. ([See also \*The SMV System.\*](#)) Representations of these declarations should be added in case a generic data structure is desired.

The components of the smv data structure are (see also Figure 4 - Smv Data Structure):

- A smv program, containing:
  - A set of modules.
- A smv module contains:
  - A name.
  - A set of variables.
  - A list of arguments.
  - A set of DEFINE declarations.
  - A set of external assigns. These represent assignments to variables of other modules, such as modules passed as argument or son modules.
  - A set of son module instances. Each module instance references a child module. Child modules may have their own son instances.
  - Synchronization type. This can be either synchronized or un-synchronized.
  - A TRANS expression.
  - An INVAR expression.
  - An INIT expression.
- A smv module-instance contains:
  - A name (represents the name of the variable having the type of the module)
  - A module user type (containing argument values and a reference to a module in the program).
- A smv variable contains:
  - A name.
  - A type.
  - An initial value.
  - An expression representing either next or current value.
- A smv dotted-name is a reference to a variable external to a module. For example, when module A references variable x, local to module B, the textual representation will be 'B.x'. The smv dotted name contains:
  - A name – the variable's original name (without the prefix).
  - A prefix – a list of values, allowing multiple "dottings".

- A separator – the string visualizing the name separations; this is simply the dot (".") string.



**Figure 4 - Smv Data Structure**

### 3.3. Handling Defines and constants

Core language allows global define and constant declarations, and local define declarations.

Smv language allows only local define declarations. Inside a smv module it is not possible to reference defines declared in other modules. (This has been checked with the Smv model verifier.)

#### Defines

Each core define declaration is translated to an equivalent smv define declaration. Local defines in a core module are translated to local defines in the translated smv module. Global core defines are translated and added to each of the smv modules created during the translation. Local smv defines derived from global core defines are listed in the additional information.

#### Constants

Core constant declarations are treated as 'define' declarations, since smv's grammar does not allow constants, and this modification is listed in the additional information. Core constants can only be global.

In order to distinguish original global declarations, global defines are prefixed with 'GD\_', and constants with 'GC\_'.

If a local define hides a global define (both having the same name), the global define is suffixed with a unique integer.

It is also possible to handle core global defines and constants by translating them to variables, assigning them with an initial value. They can then be handled as core global variables are handled, by locating them in the smv 'GLOBALS' module ([see also 6.1 Main Translation Steps, section 5 – Translation of global variables](#)). This possibility has been ruled out since it introduces a drawback: The next-state value of a smv variable, if not implicitly assigned, is chosen in-deterministically from its range. Variables originating from define (or constant) declarations would have to be reassigned with the same constant value.

### 3.4. Handling Typedefs and Variable Types

Core language allows the following types:

- Boolean
- Integer
- Range (subrange of integer type)
- Enumerated
- Array
- Scalarset ( scalarset(x) is equivalent to the subrange 0..x-1 )
- User defined (using a TYPEDEF declaration)

Smv language does not have the TYPEDEF feature and allows the following types:

- Boolean
- Range
- Enumerated
- Array
- User defined (module type)

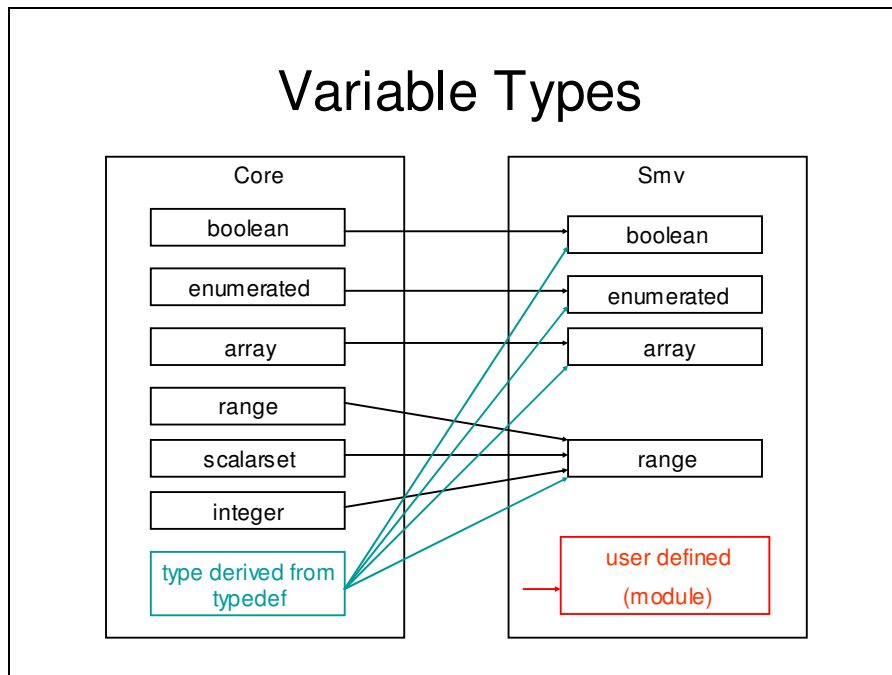


Figure 5 – Translation of Variable Types

Translation of types ([See also Figure 5 – Translation of Variable Types](#)):

- Boolean and enumerated types need no special treatment.
- Core integer type is translated to the range [MININT .. MAXINT]. These edge values are user programmable with default values.
- The scalarset type is translated to smv range type.
- Edge values of the range type are expressions in core and numerical values in smv. Not all expressions can be translated to a numerical value prior to runtime. If the expressions can be represented by a numerical value they are translated to their appropriate string representation; otherwise they are translated to MININT or MAXINT.
- The array type is composed of an index range and an element type. The index range is an expression in core and a subrange in smv. If the expression can be represented by a numerical value (prior to runtime) it is translated to the range [0 .. (numerical value of the expression)-1]; otherwise the high index range value is translated to MAXINT. The array element type can be any type in both languages and therefore translated accordingly.
- Core variable types derived from a typedef declaration are translated to their appropriate basic type. The CoreToXml program does not allow a series of recursive typedef declarations.

All type modifications are listed in the additional information.

Regarding variables having derived types, the option of prefixing (or suffixing) the variable name by the corresponding type symbol has been raised. For example, in the following case:

```
TYPEDEF
    color : {red, green, blue}
VAR
    myMorningColor : color;
```

The variable name 'myMorningColor' would be replaced by 'color\_myMorningColor' (or by 'myMorningColor\_color'). This option was discarded, since in many cases it results in burdensome identifiers.

## 3.5. Expressions

The translation uses the following expression types:

- Simple (string) expressions.
- Unary expressions:
  - Unary minus
  - Parenthesis
  - Tagged (in core) and 'next()' (in smv).
  - Logical not.
- Binary expressions:
  - Logical operators - and, or, equal, not-equal, <, <=, >, >=.
  - Mathematical operators – plus, minus, mul, div, mod.
- N-nary expressions, such as ( exp\_1 & exp\_2 & ... & exp\_n ).
- Array-element expressions
- Set expressions.
- Constant logical expressions – true / false.

Not compromising correctness, attempts are made to create and use expressions having clear, simple, readable and short textual forms.

Composite core expressions have the ability to 'simplify' themselves – remove redundant sub-expressions, parenthesis, etc. This simplification is done upon creating the core expressions, while parsing the core xml file and building of the core data structure. For example, if the core xml file contains the sub-expression '((exp))', the corresponding sub-expression in the core data structure will be '(exp)'.

Smv expressions also have the ability to 'simplify' themselves. This simplification is done after the whole smv program already exists. For example, suppose the smv program contains an expression 'exp', built as 'exp1 & exp2'. If 'exp2' happens to be a constant 'false' expression, the whole expression may be reduced to 'false', but this fact may not be known while creating 'exp'. Furthermore, 'exp2' might contain objects which need to be listed in the additional information; 'exp2' itself may be discarded, but references to these objects are still needed. ([See also Appendix 2 – Simplifying Smv Expressions.](#))

## 4. From XML to Core Data Structure

This section describes the building of a core data structure representing the XML version of a core program.

The XML format is determined according to an existing dtd (compiled at VeriTech) describing the structure of the XML version of a core program.

The core data structure (described above) is not compatible with the XML structure, and hence requires some transformations, but has the same expressive power. The main reason for the incompatibility of the (predefined) XML structure and the core data structure (defined by us), is the need for a data structure that would simplify the translation as much as possible while being indifferent to changes in the XML structure definition.

The translation between the two structures is dependant on the dtd that defines the XML structure, and would need to change upon a change to the dtd. This is, however, the only part of the program that would have to be changed in such a case.

### 4.1. Main Parsing Stages

- Creating an empty core program in the core data structure, to be filled with XML-parsed data.
- Building a DOM (Document Object Model) tree that represents the XML structure using java's existing tools:
  - javax.xml.parsers
  - javax.xml.transform
  - org.w3c.dom
- Traveling the tree DFS-wise (Depth First Search), creating core data structure objects corresponding to the DOM's elements, and inserting them in their appropriate place in the core data structure.
- All created core data structure objects are set with a XML id. This id is a string concatenation of the XML file name (local, not full path), the '#' character, and the 'ID' attribute of the corresponding DOM element.

## 4.2. Adapting to Modifications in the DTD

Parsing is done DFS-wise, according to the dtd. When encountering a specific XML element, it is assumed that its child elements are of a certain type, or that it has certain attributes. Modifications to the existing dtd would require corresponding modifications to the parsing process.

For example, in the existing dtd, modules' local variables are defined by:

```
ELEMENT Module (... , VAR?, ...)  
ELEMENT VAR (VarDef+)  
ELEMENT VarDef (ATOM, Typename, INITVAL?)
```

A reasonable modification would be:

```
ELEMENT Module (... , VarDef*, ...)  
ELEMENT VarDef (ATOM, Typename, INITVAL?)
```

Currently, when parsing a 'Module' element, if one of its child elements is a 'VAR' element, a call is made to parse a 'VAR' element, and then a call is made to parse one or more 'VarDef' elements.

After modification, when parsing a 'Module' element, for each child element of 'VarDef' type, a call should be made to parse the 'VarDef' element.

## 5. Pre-Translation

Pre-translation means processing the core program and transforming it to an equivalent core program which would be easier to translate to smv. This processing is independent of the translation, so it could also serve translations to other languages.

Programs in both languages are structured as a tree of modules. A non-leaf module has child modules with some type of synchronization between them. A leaf module has no child modules. Root modules are named 'SYSTEM' in core and 'main' in smv.

In smv, child modules are either fully-synchronized or interleaved (unsynchronized). In core, there is also a possibility of partial synchronization among child modules. The main purpose of the pre-translation stage is to eliminate partial synchronizations in the core program.

One possibility is "flattening" the core program – a process of transforming the core program to an equivalent program having only one module.

The flattening option was used in the 1.0 version, in all cases, resulting of course in a single-module smv program.

The 1.8 version introduced the first step aiming at preserving the tree-of-modules structure of the core program. A sub-tree with partial synchronization between two leaf modules was transformed to an equivalent sub-tree containing full or interleaved combinations. This transformation involves "splitting" existing modules and generation of new "intermediate" modules. Other cases of partial synchronization were treated by flattening a sub-tree, instead of flattening the whole core program.

This version (2.0) introduces a more general (although not complete) algorithm for elimination of partial synchronizations. Some cases are still treated by flattening.

A CoreFlattener application already exists at Veritech. It can be used to flatten a whole core program, or a sub-tree represented in a specific C++ data structure. The existing CoreFlattener could not be used as part of this implementation, since this version is Java implemented, with a different database representing components of the core program; furthermore, the decision whether to flatten a sub-tree is made while processing a core program represented in the Java data structure.

It is still possible, if desired, to flatten the whole core program, by using a command line flag or an entry in the configuration file.

This version includes a new Java flattener application, which can be used independently of the translation to smv. It produces a textual version of the flattened core program.

In order to produce an XML version of the flattened program, a "CoreDataStructure2Xml" component has to be created. This component has to scan the core program's elements in the core data structure, create corresponding XML elements and insert them into a DOM tree, and drop the DOM tree to a file (Similar to the function of the "SmvDataStructure2Xml" component).

In order to produce the additional information of the core to flat-core transformation, it is possible to use the existing "XmlAddInfoWriter" component, with minor adjustments. Collection of links between objects composing the original program and

those composing the flattened program already exists. The current implementation creates XML source elements from core objects and XML target elements from smv objects. What remains to be done is to create XML target elements from (flattened) core objects.

## 5.1. Eliminating Partial Synchronizations in a Core Program

Presented here is an algorithm transforming a core program to an equivalent core program having no partial synchronizations.

The motivations are:

- Preservation, as much as possible, of the tree-of-modules structure of the core program.
- Avoidance, as much as possible, of flattening.
- Separating between processing of the core program and the translation to smv.

The algorithm is executed after the core program has been parsed, and its representation exists in the core data structure, and before building a representation of an equivalent smv program in the smv data structure.

First, the basic algorithm will be presented, along with several examples. It covers all "non-problematic" cases. Later, some of the "problematic" cases are handled; this requires modifications to the basic algorithm. Finally, all other cases will be handled by flattening. Hopefully, no flattening will be used (in the future if not at the present time).

Elimination of partial synchronizations involves modifications to the structure of the core program:

- Synchronization types may be changed.
- New modules may be added to the core program.
- Existing modules may be "split" into two or more modules.
- Variables may be relocated.

These modifications compose part of the additional information gathered during the translation ([see also 8 - Gathering Additional Information](#)).

### 5.1.1. Abbreviations

MC – modcombin (a combination of modules, having some synchronization type)

S – synchronized modcombin

(Also denoted by  $A \parallel B$ )

U – unsynchronized modcombin

(Also denoted by  $A \parallel\parallel B$ )

P – partially synchronized modcombin

(Also denoted by  $A \mid (a_1, b_1), \dots, (a_n, b_n) \mid B$

where  $a_1, \dots, a_n$  are transitions of A and  $b_1, \dots, b_n$  are transitions of B)

I – modcombin containing a single module instance

(The module instance references a module)

### 5.1.2. Grammar

According to CDL grammar, a module can be either a leaf module (containing one or more transitions, and no MC) or a non-leaf module (containing a MC, and no transitions). A module can not be both a 'leaf' and a 'non-leaf'.

Following are the grammatical rules relating to MCs:

MC  $\rightarrow$  I | S | U | P

I  $\rightarrow$  name [, params] [,renames]

(where name – name of the module referenced by this instance.

params - parameters passed to the module.

renames – renames of transition names.)

S  $\rightarrow$  MC, MC

U  $\rightarrow$  MC, MC

P  $\rightarrow$  MC, pairs+, MC

(where pairs is a list of synched transition names.)

### 5.1.3. General Description

The MC parsing works in a DFS fashion: Deeper MCs existing under the current MC are parsed. When encountering an instance of a module which is a non-leaf, its own MC is parsed.

The first call is to parse the 'SYSTEM' module's MC.

In addition, the algorithm changes the terms defining a MC inside a non-leaf module. Prior to execution, a non-leaf module contains a MC, defining a tree structure of child modules, as dictated by the above mentioned grammar. After execution, the module will have a set of child modules and a synchronization type ("synched" or "unsynched").

The reason for this is to allow a more convenient translation of modules to smv: Smv modules may have child modules, and one of these two synchronization types.

### 5.1.4. Basic Algorithm

Assumptions of the basic algorithm:

1. There are no occurrences of a S type under a P type.
2. There are no occurrences of the same transition appearing more than once in a synched pair list of P type MCs along the same path in the MC sub-tree.

For example, the following MCs can be handled by the basic algorithm:

(A || B) ||| (C || (D || E))

A | (a,b1) | (B | (b2,c) | C)

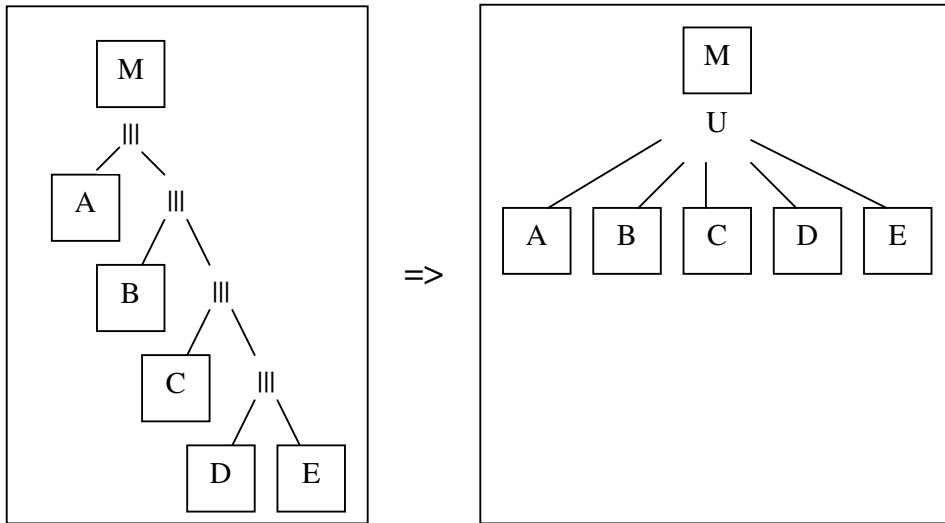
While the following MCs can not:

A | (a,b) | (B || C)

A | (a,b) | (B | (b,c) | C)

Several examples are presented to clarify the transformation produced by the algorithm.

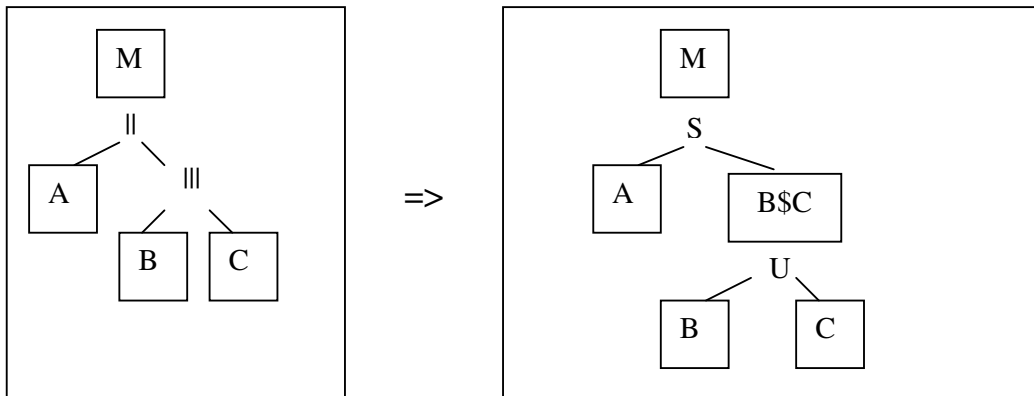
Example 1 (Dining Philosophers)



**Figure 6 – Modcombin Parsing, Dining Philosophers Example**

Module M  
 A ||| (B ||| (C ||| (D ||| E)))  
 =>  
 Module M  
 synch type: 'unsynched'  
 sons: A,B,C,D,E

Example 2 (Mixed Synched / Unsynched)



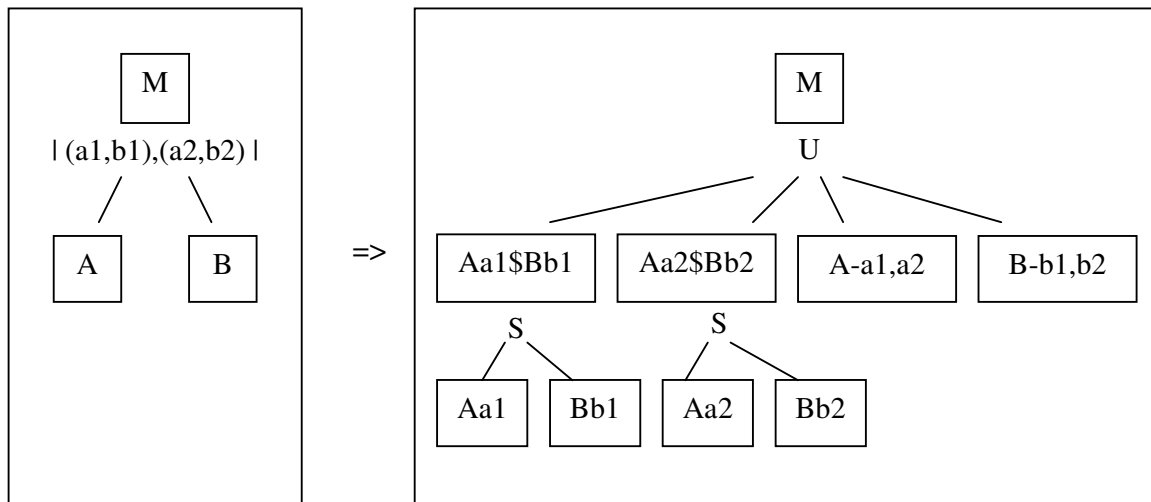
**Figure 7 - Modcombin Parsing, Mixed Synched / UnSynched Example**

```

Module M
  A || (B ||| C)
=>
Module M
  synch type: 'synched'
  sons: A,B$C
Module B$C
  synch type: 'unsynched'
  sons: B,C

```

Example 3 (Simple Partial Synch)



**Figure 8 - Modcombin Parsing, Simple Partial Synch Example**

```

Module M
  A | (a1,b1),(a2,b2) | B
=>
Module M
  synch type: 'unsynched'
  sons: Aa1$Bb1, Aa2$Bb2, A-{a1,a2}, B-{b1,b2}
Module Aa1$Bb1
  synch type: 'synched'
  sons: Aa1,Bb1
Module Aa2$Bb2
  synch type: 'synched'
  sons: Aa2,Bb2
Module Aa1 // leaf module containing a1 only
Module Aa2 // leaf module containing a2 only
Module Bb1 // leaf module containing b1 only
Module Bb2 // leaf module containing b2 only
Module A-{a1,a2} // leaf module containing all A's transitions except a1,a2
Module B-{b1,b2} // leaf module containing all B's transitions except b1,b2

```

### 5.1.5. Pseudo-Code for the Basic Algorithm

The following pseudo-code is an outline of the basic algorithm. Details and refinements will be presented later.

Differently typed MCs are separated, due to different treatment.

Notice the parameters:

- thisMC – The currently parsed MC.
- parentMC – Parent MC (in the original program) of the currently processed MC.
- parentModule – Module parenting the currently processed MC. This module may be either the original module parenting the MC or a new "intermediate" module.
- unNeededTransitions – A set of transitions already "taken" higher up the sub-tree, if participating in partial synchronization.

Some of the parameters may be null.

```
parseMC (MC thisMC, MC parentMC,  
        Module parentModule, Set unNeededTransitions)
```

```
  if ( thisMC instance of I )  
    parseITypeMC ()  
  else if ( thisMC instance of S )  
    parseSTypeMC ()  
  else if ( thisMC instance of U )  
    parseUTypeMC ()  
  else if ( thisMC instance of P )  
    parsePTypeMC ()  
  else // none of the valid MC types  
    error()
```

```
// end of parseMC ()
```

```
parseITypeMC (MC thisMC, MC parentMC,  
             Module parentModule, Set unNeededTransitions )
```

```
  module = getModuleByName(thisMC->getName)  
  if ( module instance of leafModule & unNeededTransitions != null )  
    remove unNeededTransitions from module  
  if ( module instance of nonLeafModule )  
    parseMC (module->MC, null, module, unNeededTransitions)  
  add module as son to parentModule
```

```
// end of parseITypeMC ()
```

```

parseSTypeMC (MC thisMC, MC parentMC,
              Module parentModule, Set unNeededTransitions )

MC leftMC = thisMC -> leftMC
MC rightMC = thisMC -> rightMC
if ( parentModule's synch type isn't set )
    parentModule.setSynch ('Synchronized')
if ( parentMC == null || parentMC instance of S )
    parseMC ( leftMC, thisMC, parentModule, unNeededTransitions )
    parseMC ( rightMC, thisMC, parentModule, unNeededTransitions )
if ( parentMC instance of U || parentMC instance of P )
    Module newModule = createNewIntermediateModule()
    newModule.setSynch ('Synchronized')
    add newModule as son to parentModule
    parseMC ( leftMC, thisMC, newModule, unNeededTransitions )
    parseMC ( rightMC, thisMC, newModule, unNeededTransitions )

// end of parseSTypeMC ()

parseUTypeMC (MC thisMC, MC parentMC,
              Module parentModule, Set unNeededTransitions )

MC leftMC = thisMC -> leftMC
MC rightMC = thisMC -> rightMC
if ( parentModule's synch type isn't set )
    parentModule.setSynch ('UnSynchronized')
if ( parentMC == null || parentMC instance of U || parentMC instance of P )
    parseMC ( leftMC, thisMC, parentModule, unNeededTransitions )
    parseMC ( rightMC, thisMC, parentModule, unNeededTransitions )
if ( parentMC instance of S )
    Module newModule = createNewIntermediateModule()
    newModule.setSynch ('UnSynchronized')
    add newModule as son to parentModule
    parseMC ( leftMC, thisMC, newModule, unNeededTransitions )
    parseMC ( rightMC, thisMC, newModule, unNeededTransitions )

// end of parseUTypeMC ()

```

```

parsePTypeMC (MC thisMC, MC parentMC,
              Module parentModule, Set unNeededTransitions )

if ( shouldFlat(thisMC) )
  Module flatModule = flatten(thisMC)
  if ( parentMC != null )
    add flatModule as son to parentModule
  return

MC leftMC = thisMC -> leftMC
MC rightMC = thisMC -> rightMC
Set leftUnNeededTransitions = unNeededTransitions + left side from synched
                             transitions pairs
Set rightUnNeededTransitions = unNeededTransitions + right side from
                              synched transitions pairs
if ( parentModule's synch type isn't set )
  parentModule.setSynch ('UnSynchronized')
buildModulesFromSynchedTranstitionPairs (thisMC, parentModule)
if ( parentMC == null || parentMC instance of U || parentMC instance of P )
  parseMC ( leftMC,thisMC,parentModule,leftUnNeededTransitions )
  parseMC ( rightMC,thisMC,parentModule,rightUnNeededTransitions )
if ( parentMC instance of S )
  Module newModule = createNewIntermediateModule()
  newModule.setSynch ('UnSynchronized')
  add newModule as son to parentModule
  parseMC ( leftMC,thisMC,newModule,leftUnNeededTransitions )
  parseMC ( rightMC,thisMC,newModule,rightUnNeededTransitions )

// end of parsePTypeMC ()

buildModulesFromSynchedTranstitionPairs ( MC partialMC, Module parentModule )

for each pair of transitions in partialMC -> pairs
  Transition t1 = pair -> left transition
  Transition t2 = pair -> right transition
  Module m1 = new leaf module containing a single transition (t1)
  Module m2 = new leaf module containing a single transition (t2)
  Module m1$m2 = new non-leaf module
  m1$m2.setSynch ('Synchronized')
  add m1 and m2 as sons to m1$m2
  add m1$m2 as son to parentModule

// end of buildModulesFromSynchedTranstitionPairs()

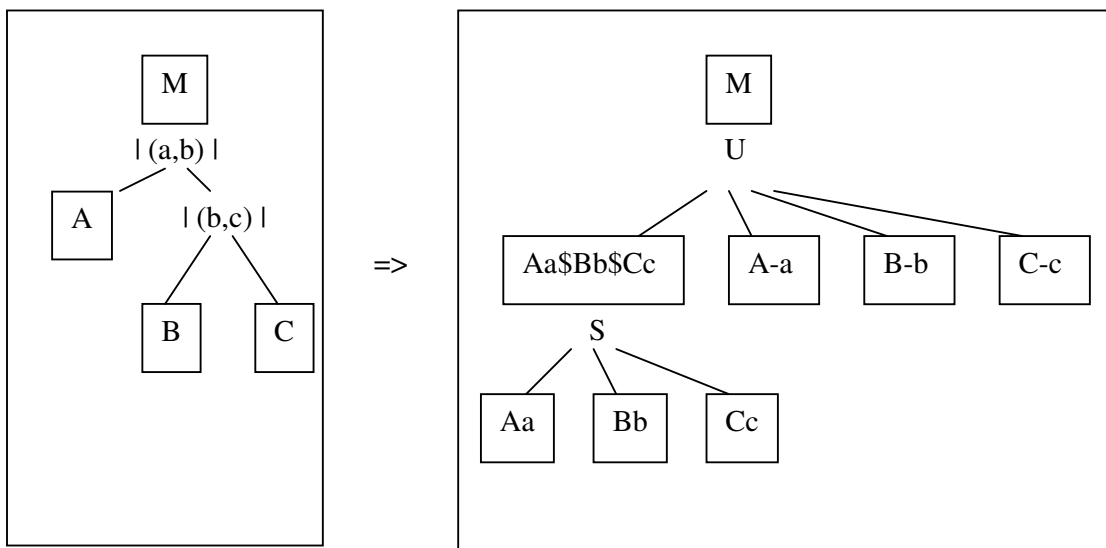
```

### 5.1.6. First Improvement

This improvement covers cases of a transition appearing more than once in the same path along a MC sub-tree. The basic algorithm does not apply to such cases. As illustrated in example 3, when a pair of synched transitions is encountered, a sub-tree representing them is constructed. If transition t1 is paired with t2 in a P type MC, and t1 is also paired with t3 in a deeper MC along the same path, a sub-tree representing the synchronization of {t1,t2,t3} needs to be constructed.

As with the basic algorithm, this improvement still assumes that no S type MC exists under P type MCs.

#### Example 4 (Same Transition in Two P Type MCs)



**Figure 9 - Same Transition in Two P Type MCs, Example 4**

Module M

A |(a,b)| (B |(b,c)| C)

=>

Module M

synch type: 'unsynched'

sons: Aa\$Bb\$Cc, A-{a},B-{b},C-{c}

Module Aa\$Bb\$Cc

synch type: 'synched'

sons: Aa,Bb,Cc

Module Aa // leaf module containing a only

Module Bb // leaf module containing b only

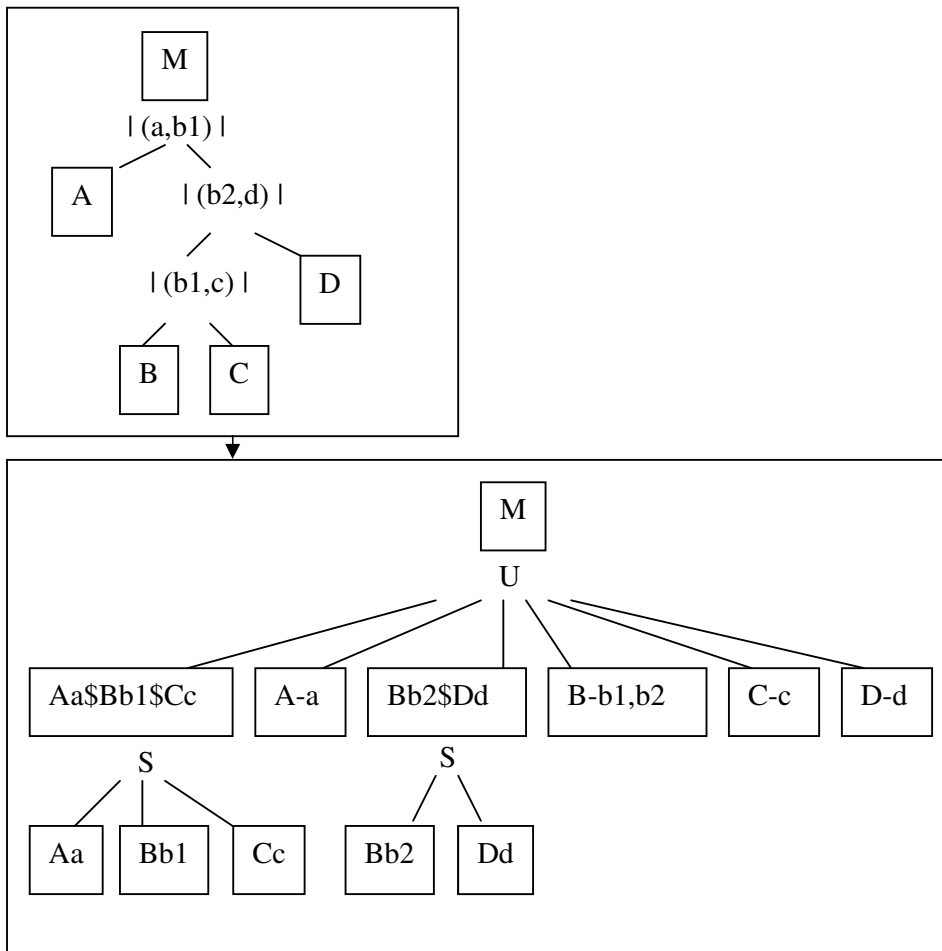
Module Cc // leaf module containing c only

Module A-{a} // leaf module containing all A's transitions except a

Module B-{b} // leaf module containing all B's transitions except b

Module C-{c} // leaf module containing all C's transitions except c

Example 5 (Same Transition in Two P Type MCs)



**Figure 10 - Same Transition in Two P Type MCs, Example 5**

Module M

A |(a,b1)| ((B |(b1,c)| C) |(b2,d)| D)

=>

Module M

synch type: 'unsynched'

sons: Aa\$Bb1\$Cc, A-{a},Bb2\$Dd,B-{b1,b2},C-{c},D-{d}

Module Aa\$Bb1\$Cc

synch type: 'synched'

sons: Aa,Bb1,Cc

Module Bb2\$Dd

synch type: 'synched'

sons: Bb2,Dd

Module Aa // leaf module containing a only

Module Bb1 // leaf module containing b1 only

Module Bb2 // leaf module containing b2 only  
 Module Cc // leaf module containing c only  
 Module Dd // leaf module containing d only  
 Module A-{a} // leaf module containing all A's transitions except a  
 Module B-{b1,b2} // leaf module containing all B's transitions except b1,b2  
 Module C-{c} // leaf module containing all C's transitions except c  
 Module D-{d} // leaf module containing all D's transitions except d

This improvement does not work in the following case: Suppose transition a1, contained in module A, appears in the synched-pair list of a P type MC<sub>1</sub>, and also in the synched-pair list of P type MC<sub>2</sub> lower along the same path. Suppose further, without loss of generality, that a1 is on the left-side of the synched-pair list of MC<sub>1</sub>. If A also appears under a non-P type MC<sub>3</sub>, which is not contained in MC<sub>2</sub>'s sub-tree, the suggested solution is erroneous. The following example illustrates this case, which will be treated by flattening.

Example 6 (Same Transition in Two P Type MCs, with same module under different typed MCs)

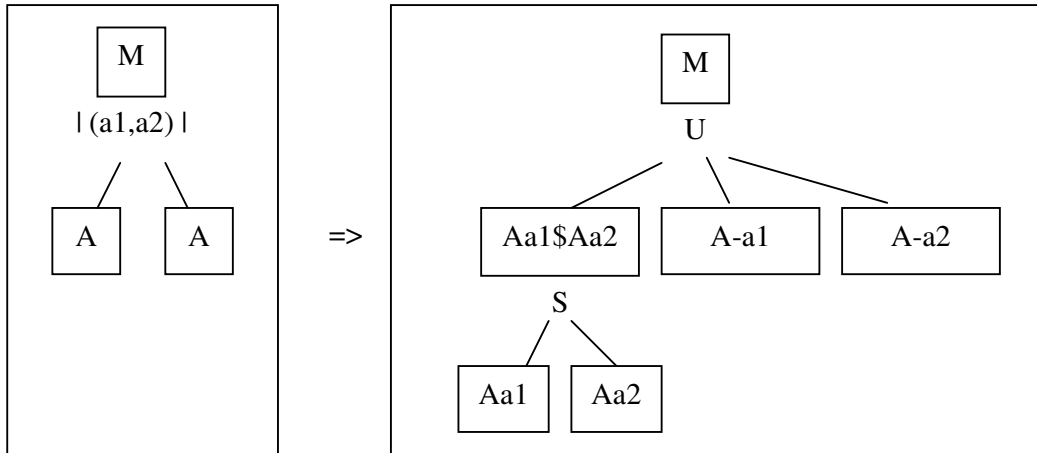


**Figure 11 - Same Transition in Two P Type MCs, Example 6**

### 5.1.7. Second Improvement

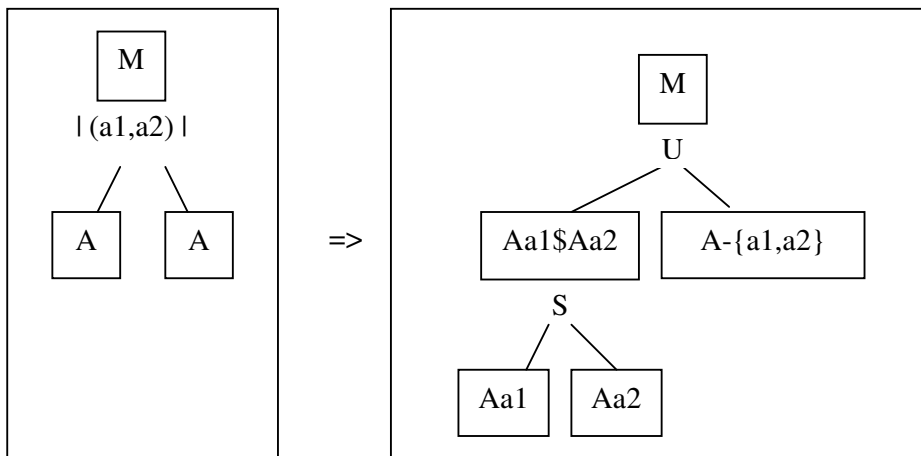
This section deals with cases in which two brother modules, under a P type, are actually the same module. The basic algorithm is still correct, but an improvement can be made, enhancing readability and minimizing the number of newly created modules.

#### Example 7 (Identical brothers under P type MC)



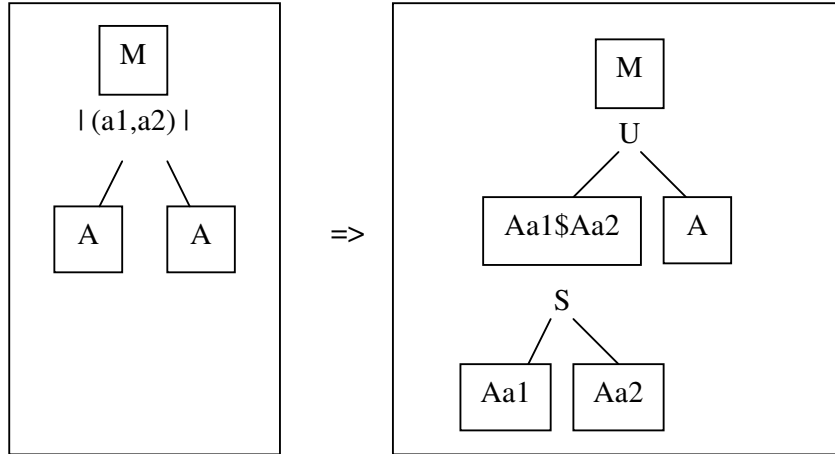
**Figure 12 - Identical brothers under P type MC, Example 7 (Correct version)**

Notice that the following transformation is incorrect: Originally, it is possible to choose transition a2 from the left instance of A, without a2 being synchronized with a1. This is possible according to the transformation shown in Figure 12 - Identical brothers under P type MC, Example 7 (Correct version), but not according to Figure 13 - Identical brothers under P type MC, Example 7 (Incorrect version).



**Figure 13 - Identical brothers under P type MC, Example 7 (Incorrect version)**

If the set of arguments passed to  $A-\{a1\}$  is different from the set of arguments passed to  $A-\{a2\}$ , the structure illustrated above should be kept. However, if the two sets are exactly identical, a better transformation would be the following (see Figure 14 - Identical brothers under P type MC, Example 7 (Special case)):



**Figure 14 - Identical brothers under P type MC, Example 7 (Special case)**

This improvement is not yet implemented.

## 5.1.8. Refinements

### 5.1.8.1. Empty Modules

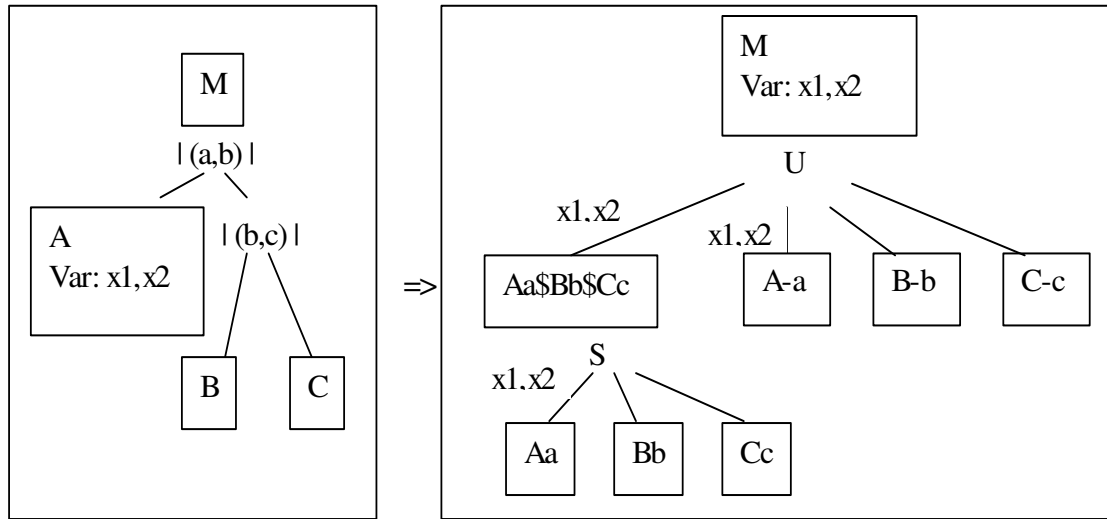
When building a new module by removing some transitions from an existing module, such as  $B-\{b1,b2\}$ , it may happen that the new module has no transitions. In this case the new module is not needed and should be eliminated.

### 5.1.8.2. Variable Uniqueness and Variable Relocation

When "splitting" an existing module, such as  $A$  to  $Aa1, A-\{a1\}$ , it is important to keep exactly one instance of each of  $A$ 's local variables. Therefore  $A$ 's variables need to be moved higher up in the sub-tree, and passed as arguments to the newly created modules. In case of name ambiguity, variable names are suffixed with a unique integer.

Relocation of variables is done after parsing of modcombins is completed.

#### Example 4, Revisited

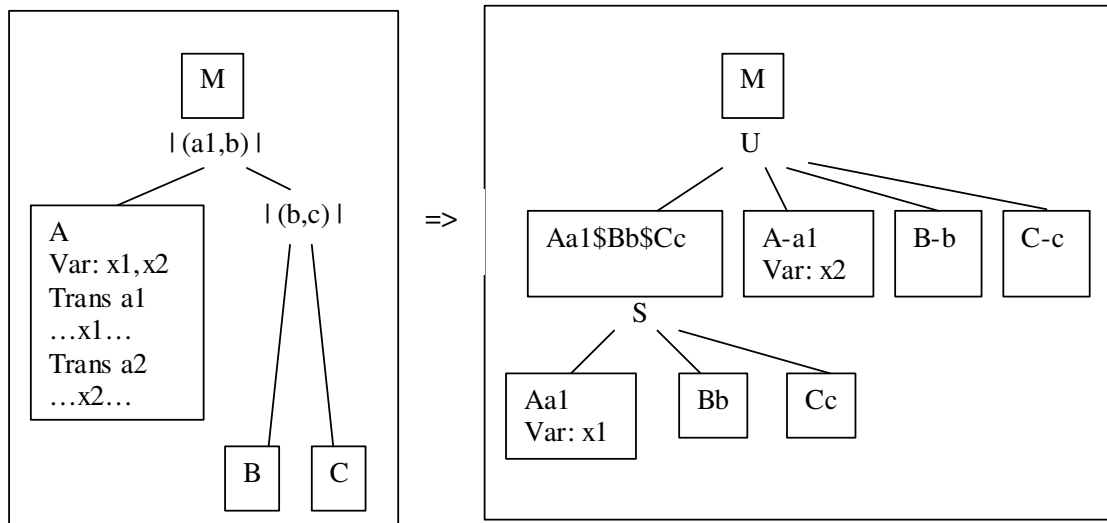


**Figure 15 – Variable Relocation, Example 4 Revisited**

$A$ 's local variables are relocated at  $M$ .  
 $M$  passes them as arguments to  $Aa$  and  $A-\{a\}$ .

In some cases, variables need not be relocated higher up in the sub-tree:  
 If  $A$  contains local variables  $x1$  and  $x2$ ,  $x1$  is referenced only in transition  $a1$ , and  $x2$  is referenced only in transition  $a2$ , then  $x1$  should be placed in  $Aa1$  and  $x2$  should be placed in  $A-\{a1\}$ . This improvement is not yet implemented (currently, all variables are relocated).

### Example 4, Revisited Again



**Figure 16 – Variable Relocation, Example 4, Revisited Again**

#### 5.1.8.3. Transition Renaming

When a simple rename (renaming of a single transition) is encountered, the transition is retrieved (by its original name) and handled.

When encountering a 'pair renaming' (renaming two transitions under a single name), both original transitions are retrieved. This is actually a special case of the first improvement.

#### 5.1.8.4. Book-Keeping

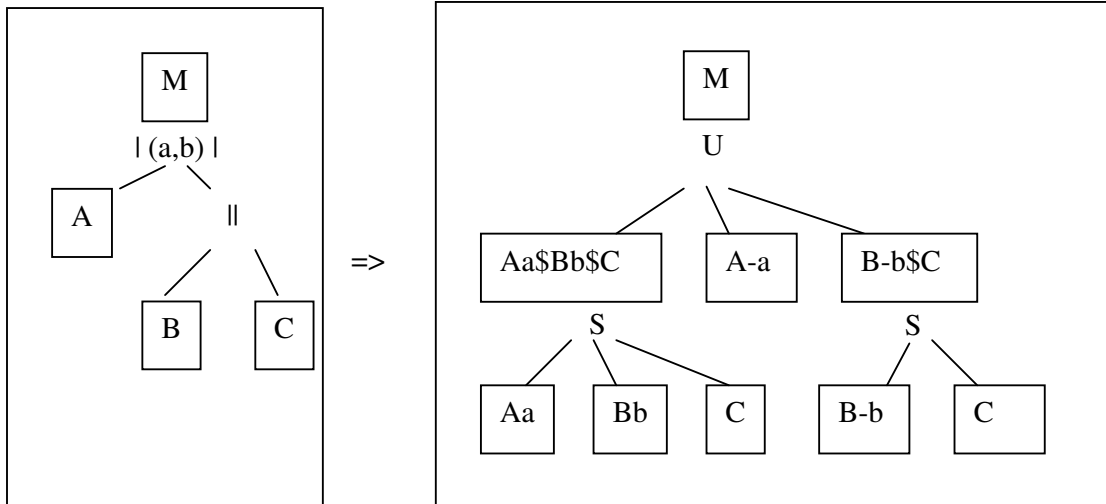
During the pre-translation stage a "BookKeeper" is used. Its function is to keep information of original modules encountered during the process, and newly created modules.

For example, if module A was "split" to Aa1 and A-{a1}, the newly created modules should be added to the program. If A was encountered elsewhere in the core program, not under a P type MC, it should be kept. However, if A does not appear elsewhere in the core program, it can be discarded.

### 5.1.9. S Type MC Under P Type

Flattening will be used to handle cases in which a S type MC appears under a P type. Some simple cases (such as the following example) can be treated without flattening. At present, however, an algorithm fully capable of handling more complex cases is not yet phrased.

#### Example 8 (S Type Under P Type)



**Figure 17 - S Type Under P Type, Example 8**

Module M

A | (a,b) | (B || C)

=>

Module M

synch type: 'unsynched'

sons: Aa\$Bb\$C, A-{a},B-{b}\$C

Module Aa\$Bb\$C

synch type: 'synched'

sons: Aa,Bb,C

Module B-{b}\$C

synch type: 'synched'

sons: B-{b},C

Module Aa // leaf module containing a only

Module Bb // leaf module containing b only

Module C // leaf module containing all C's transitions

Module A-{a} // leaf module containing all A's transitions except a

Module B-{b} // leaf module containing all B's transitions except b

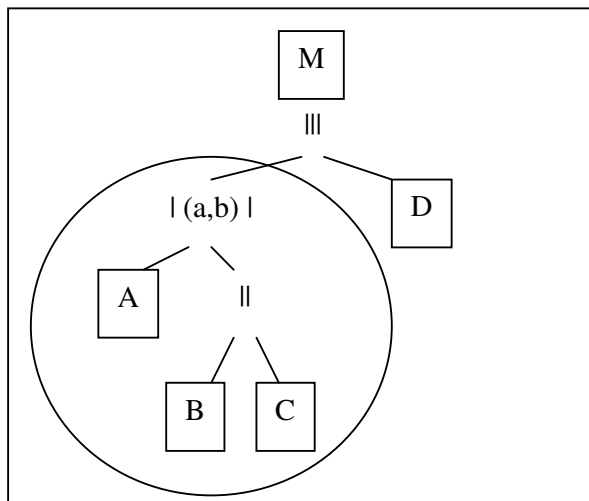
## 5.2. Flattening

Flattening is applied to a modcombin sub-tree if no other option exists. The decision whether to flatten is made while parsing the core program at the pre-translation stage. It is also possible to flatten the whole core program, if desired, by using a command line flag or an entry in the configuration file. This possibility has two main advantages:

- Compatibility with Core2Smv version 1.0, in which complete flattening was applied in all cases.
- In some cases, the smv program produced by applying complete flattening is simpler and more compact, compared to the smv program produced by applying the algorithm for elimination of partial synchronizations ([see also Appendix 7 – Examples, 3.3 Buffer Example](#)).

The flattening process receives a modcombin as input and returns a core leaf module. It is a recursive process, working DFS-wise: Deeper modcombins are flattened prior to the currently processed modcombin.

Flattening is applied to a modcombin, not to a module. The reason is that sometimes a structure defined by a modcombin, and not by a specific module, needs to be flattened. In the following example, only the circled modcombin needs to be flattened.



**Figure 18 - Modcombin Flattening**

Flattening involves modifications to the structure of the core program. These modifications compose part of the additional information gathered during the translation ([see also 8 - Gathering Additional Information](#)).

First, the general outline is presented with pseudo-code; several details are explained later.

### 5.2.1. Pseudo-code for Flattening (First Version)

```
1 Core_LeafModule flatten (Modcombin MC)
2   if ( MC's type is I )
3     module = get module by name(MC->name)
4     if ( module is a leaf )
5       newModule = module.clone()
6     else // module is a non-leaf
7       childMC = module->MC
8       flatChild = flatten(childMC)
9       newModule = module.cloneAsLeaf
10      add flatChild's components to newModule
11    return newModule
12  else // MC has two child MCs
13    leftMC = MC->left
14    rightMC = MC->right
15    leftModule = flatten(leftMC)
16    rightModule = flatten(rightMC)
17    newModule = new Core_LeafModule
18    add leftModule's components to newModule // except transitions
19    add rightModule's components to newModule // except transitions
20    if ( MC's type is U )
21      add leftModule's transitions to newModule
22      add rightModule's transitions to newModule
23    else if ( MC's type is S )
24      T1 x T2 = cartesian product of transitions from leftModule, rightModule
25      for each pair of transitions (t1,t2) in T1 x T2
26        mergedTransition = merge(t1,t2)
27        add mergedTransition to newModule
28    else // MC's type is P
29      for each pair of transitions (t1,t2) from P's list of synched transition pairs
30        mergedTransition = merge(t1,t2)
31        add mergedTransition to newModule
32        add transitions of leftModule, not in P's list of synched transitions, to
33          newModule
34        add transitions of rightModule, not in P's list of synched transitions, to
35          newModule
34  return newModule
35 // end of flatten()
```

### 5.2.2. Merging of Transitions

Two transitions t1, t2 are merged in the following way:

1. merged.name = t1.name + "\$" + t2.name
2. merged.enable = t1.enable & t2.enable
3. merged.relation = t1.relation & t2.relation
4. The set of assignments in the merged transition is the union of the assignments in t1 and t2.

If t1 and t2 assign different values to the same variable, the transitions are not merged, and a warning message is printed. Note that transitions should be merged only when they are synchronized in the original core program, and if they contradict each other – this is already an error in the original program.

### 5.2.3. Resolving Name Ambiguity

Name ambiguity is resolved by adding the module name to all appearances of its identifiers (defines, variables, transitions), before inserting the (flattened) child's components into its parent. If an ambiguity still occurs, identifiers are suffixed by a unique integer.

The module name is added as prefix to its identifiers.

In (original) leaf modules, the module name is not added to its identifiers.

Therefore, the following modifications are made to the above pseudo-code:

- After line 8, the following line is inserted:  
(8a) flatChild .add module name to components ( )
- After line 15, the following line is inserted:  
(15a) leftModule .add module name to components ( )
- After line 16, the following line is inserted:  
(16a) rightModule .add module name to components ( )

If leftModule and rightModule have the same name, one of them is suffixed by a unique integer. This case occurs in the following example:

```
Module M
  A || A
```

### 5.2.4. Passing of Arguments

An I type modcombin contains a name of a referenced module, with a list of actual parameters. The referenced module, in turn, has a list of formal parameters. In the core data structure, actual parameters of a module instance are expressions, while formal parameters of a module are variables. (In the CDL dtd, an actual parameter is an expression and a formal parameter is a string name + a variable type).

Furthermore, an I type modcombin contains a module-instance, not a module.

When flattening a sub-structure containing a module instance referencing a child module, all appearances of the child's formal parameters, inside the child, are replaced by respective values found in the actual parameter list of the corresponding module instance.

For example, while flattening the following structure:

```

Module M
  VAR: x: Integer, y: Boolean
  A(x,y)

Module A (k: Integer, b: Boolean)
  TRANS t
    enable: b' = true;
    assign: k' := k+1;

```

The flattened version of M will contain all A's components. Therefore, inside A, all appearances of 'k' should be replaced by 'x', and all appearances of 'b' should be replaced by 'y'.

Therefore, the following modifications are made to the above pseudo-code:

- Line 3 is changed to:
  - (3a) instance = MC->moduleInstance
  - (3b) module = get module by name (instance->name)
- After line 5, the following line is inserted:
  - (5a) replace argument values (instance,newModule)
- After lines 8 and 8a, the following line is inserted:
  - (8b) replace argument values (instance,flatChild)

### 5.2.5. The Cojoin Expression

A cojoin expression, if appearing in a core module, is a pre-condition for instantiating the module. When a child module is flattened, before its components are inserted to its parent module, the cojoin expression is added to all the child transition's enable expressions, in the following way:

```

if ( cojoin != null & cojoin != true )
  for each transition t of the child
    t.enable = cojoin & t.enable

```

Therefore, the following modifications are made to the above pseudo-code:

- After lines 15 and 15a, the following line is inserted:
  - (15b) leftModule .add cojoin to transitions ( )
- After lines 16 and 16a, the following line is inserted:
  - (16b) rightModule .add cojoin to transitions ( )

The line

```
(10) add flatChild's components to newModule
```

Already assumes a conjunction of the cojoin expressions of flatChild and newModule.

## 5.2.6. Flattening the Whole Core Program

Since flattening is applied to a modcombin and not to a module, when flattening the 'SYSTEM' module, the following adjustment should be made:

- Line 17 is change to:
  - (17a) if ( MC is 'SYSTEM' module's MC )
  - (17b)     newModule = 'SYSTEM' module.clone
  - (17c) else
  - (17d)     newModule = new Core\_LeafModule

## 5.2.7. Naming the Flattened Module

The (recursive) flattening process returns a module. When processing a modcombin, after its child modcombins have been flattened to leaf modules, the returned module will have its name set to the concatenation of its (flattened) child names.

Therefore, the following adjustment is made to the above pseudo-code:

- After lines 17 and 17d, the following line is added (inside the 'else' of 17c) :
  - (17e) newModule.name = leftModule.name + "\$" + rightModule.name

### 5.2.8. Pseudo-code for Flattening (Final Version)

After the above mentioned modifications, the flattening pseudo-code would be:

```
1 Core_LeafModule flatten (Modcombin MC)
2   if ( MC's type is I )
3a     instance = MC->moduleInstance
3b     module = get module by name (instance->name)
4     if ( module is a leaf )
5       newModule = module.clone()
5a      replace argument values (instance,newModule)
6     else // module is a non-leaf
7       childMC = module->MC
8       flatChild = flatten(childMC)
8a      flatChild .add module name to components ( )
8b      replace argument values (instance,flatChild)
9       newModule = module.cloneAsLeaf
10      add flatChild's components to newModule
11     return newModule
12   else // MC has two child MCs
13     leftMC = MC->left
14     rightMC = MC->right
15     leftModule = flatten(leftMC)
15a    leftModule .add module name to components ( )
15b    leftModule .add cojoin to transitions ( )
16     rightModule = flatten(rightMC)
16a    rightModule .add module name to components ( )
16b    rightModule .add cojoin to transitions ( )
17a    if ( MC is 'SYSTEM' module's MC )
17b      newModule = 'SYSTEM' module.clone
17c    else
17d      newModule = new Core_LeafModule
17e      newModule.name = leftModule.name + "$" + rightModule.name
18    add leftModule's components to newModule // except transitions
19    add rightModule's components to newModule // except transitions
20    if ( MC's type is U )
21      add leftModule's transitions to newModule
22      add rightModule's transitions to newModule
23    else if ( MC's type is S )
24      T1 x T2 = cartesian product of transitions from leftModule, rightModule
25      for each pair of transitions (t1,t2) in T1 x T2
26        mergedTransition = merge(t1,t2)
27      add mergedTransition to newModule
```

```
28     else // MC's type is P
29         for each pair of transitions (t1,t2) from P's list of synched transition pairs
30             mergedTransition = merge(t1,t2)
31             add mergedTransition to newModule
32             add transitions of leftModule, not in P's list of synched transitions, to
                newModule
33             add transitions of rightModule, not in P's list of synched transitions, to
                newModule
34     return newModule
35 // end of flatten()
```

## 6. Translation of a Core Program

Translation from core to smv is done after the pre-translation stage. The core program already exists in the core data structure, and has been pre-processed. At this point, the core program is equivalent to the original program, but differently structured, thus enabling a more comfortable translation to smv. Two main distinctions should be noted:

- There are no partial synchronizations in the core program – these have been eliminated at the pre-translation stage.
- Non-leaf core modules have a set of son module-instances, and a synchronization type (either synched or un-synched). Core modcombins may still exist, but are no longer used when translating to smv.

### 6.1. Main Translation Steps

Translation of a core program to a smv program is composed of the following steps:

1. Core global defines and global constants are inserted to all the core modules (smv does not allow global identifiers).
  - Global define symbols are prefixed with 'GD\_'.
  - Global constant symbols are prefixed with 'GC\_'.
  - If a name conflict occurs with a local identifier, the global symbol is suffixed with a unique integer.
  - All appearances in the core program, of the original identifiers, are replaced by the new symbols.
  - The reason for including this step in the translation to smv, instead of in the pre-translation stage, is that a core program, whichever way processed, may still contain global declarations. Special treatment of global declarations is needed only upon translation to a language not allowing them.  
([See also 3.3 – Handling Defines and Constants.](#))
2. Resolving name conflicts involving transition names, in all leaf modules of the core program.
  - Core language allows an identifier to denote both a transition name and another object in the same module (a variable name, and argument, or a define symbol).
  - In such cases, the transition name is changed, by suffixing it with a unique integer.
  - The reason for including this step in the translation to smv, instead of in the pre-translation stage, is that transition names will be used as identifiers in the smv program. The core program, whichever way processed, may still contain these naming conflicts.
3. Translation of typedefs.
  - A typedef declaration consists of a symbol (denoting a derived type) and a (basic) type.
  - A mapping of symbols to basic types is formed, to be used later, upon translation of variables.

- When encountering a variable whose type is a symbol in the mapping, its corresponding basic type is extracted.  
([See also 3.4 – Handling Typedefs and Variable Types.](#))
4. Creation of a new empty smv program.
  5. Translation of global variables.
    - If the core program has no global variables – nothing to do.
    - Core global variables are collected and translated to corresponding smv variables.
    - A smv module named 'GLOBALS' is created. The translated smv variables are inserted to this module as local variables.
    - The 'GLOBALS' module is inserted to the smv program.
    - This 'GLOBALS' module will later be passed as argument to all other smv modules present in the smv program, except to the 'main' module.
  6. Translation of modules.
    - Each core module is translated to its smv counterpart.
    - Smv modules are inserted to the smv program.
    - A detailed explanation follows in sections [6.2 - Translation of a Core Non-Leaf Module](#) and [6.3 - Translation of a Core Leaf Modules](#).
  7. Passing of the 'GLOBALS' module to all other modules.
    - The 'GLOBALS' module is passed as argument to all other smv modules present in the smv program, except to the 'main' module.
    - A 'globals' module instance is created, and added as son-module to the 'main' module.
    - This step is separated from step 3, since at the time of creating the 'GLOBALS' module and the variables it contains, other smv module are not yet created. On the other hand, translation of core global variables to smv can not be done after translation of modules, since the modules may reference these global variables.
  8. Simplifying all expressions of the smv program ([See also Appendix 2 – Simplifying Smv Expressions](#)).

The option of locating global defines and constants in the 'GLOBALS' module has not been overlooked. However, the smv model verifier proved that accessing define declarations of other modules is not possible, so this option has been ruled out.

## 6.2. Translation of a Core Non-Leaf Module

This section describes the translation of a core non-leaf module (coreM) to a smv non-leaf module (smvM), along with several examples.

A core non-leaf module has one or more child modules (represented by module-instances) and a synchronization type. Possible synchronization types are synched, un-synched, and single-instance synch (partial synchs have previously been eliminated).

In smv, child modules are considered local variables. In order to distinguish them from "regular" variables, and also due to their different functionality, they are represented in the smv data structure by smv module-instances. When printing the

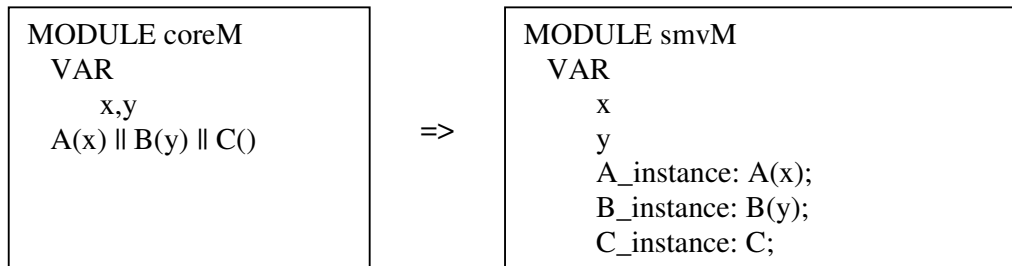
smv program to a file, smv module-instances are simply gathered under the 'VAR' section of their containing module.

The smv reserved word 'process' denotes interleaving of son modules. If present, in the 'VAR' section of a module, all son modules are interleaved; otherwise all son modules are (fully) synchronized.

### 6.2.1. Translation Procedure

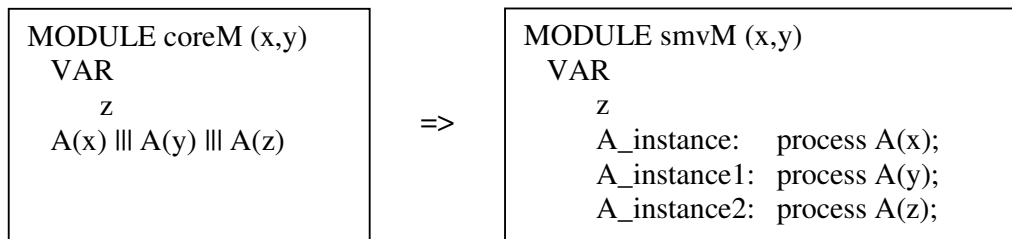
1. smvM = new empty smv module.
2. Check the synchronization type of coreM.
  - a. If it is 'synched':  
Set smvM's synch type to 'NO\_PROCESS'.
  - b. If it is 'un-synched':  
Set smvM's synch type to 'PROCESS'.
  - c. If it is 'single-instance synch':  
Set smvM's synch type to 'NO\_PROCESS'.  
(This decision is arbitrary, since the module has a single child.)
3. For each of coreM's child module instances:
  - a. Translate the core module instance to a smv module instance:
    - A core module instance has a name and a list of argument values. The name references the child module.
    - A smv module instance contains a name and a user-type. The user-type contains a name (referencing the child module) and a list of argument values.
    - The smv user-type name is set by translating the core module instance name.
    - The smv module instance name is the same name, suffixed by '\_instance'. In case of name ambiguity (different module instances referencing the same module) it is furtherly suffixed by a unique integer.
  - b. Add the smv module instance as child to smvM.
4. Translate local defines.
5. Translate local variables.
6. Translate arguments.
7. Create a 'terminate' variable and add it to smvM's set of local variables.
  - 'terminate' is a boolean variable determining whether calculation paths are finite.
  - All leaf modules also have a 'terminate' variable. The value assigned to it in leaf modules is explained in the 'translation of leaf modules' section ([See also 6.3.2.3 – Termination States](#)).
  - If the smvM has 'PROCESS' synchronization – its 'terminate' variable is assigned the logical 'and' of all the 'terminate's of its sons. Otherwise (full synchronization over all sons) – it is assigned the logical 'or' of all the 'terminate's of its sons.
8. If coreM includes a comment – write to the additional information.
9. If coreM includes a cojoin expression – translate it to a smv expression and place it in the INIT section of smvM.
10. Add smvM to the smv program.

### Example 1 (Translation of a Non-Leaf Module, Synchronized)



**Figure 19 - Translation of a Synchronized Non-Leaf Module**

### Example 2 (Translation of a Non-Leaf Module, Interleaved)



**Figure 20 - Translation of an UnSynchronized Non-Leaf Module**

## 6.3. Translation of a Core Leaf Module

This section describes the translation of a core leaf module (coreM) to a smv leaf module (smvM), and contains a discussion of variable assignments and state to state transitions. Various aspects will gradually be introduced.

### 6.3.1. State to State Transitions in Core

Apart from variables, arguments and defines, a core leaf module contains a set of transitions. A core transition is composed of:

- An 'assign' section, containing assignments to variables.
- An 'enable' expression - a pre-condition for executing the transition.
- A 'relation' expression - a post-condition for executing the transition.

A core transition must contain:

- An 'enable' expression (this can be a constant 'true' expression if no 'enable' is desired).
- An 'assign' section (containing at least one assignment expression), or a 'relation' expression, or both.

Among the core module's transitions, exactly one is chosen, in-deterministically, to be executed. Of course, the chosen transition's 'enable', 'relation', and assignment statements must be valid. When a transition is chosen, all its assignments are executed.

In core, only next-state values can be assigned to variables. (In smv, either current-state or next-state values can be assigned, as long as there are no contradictions.)

A tagged variable denotes the next value of the variable.

A typical core leaf module is presented; the notations will be used later.

```
MODULE coreM ()
{
  VAR
    v: type_v;
  TRANS t1:
    enable: t1E;
    assign: v' = t1Av;
    relation: t1R;
    ...

  TRANS tn:
    enable: tnE;
    assign: v' = tnAv;
    relation: tnR;
}
```

A core module may also receive arguments. Arguments are variables, assigned in the same way as variables, and referenced in the module's expressions in the same way as variables. In the following discussion, if not mentioned otherwise, arguments are treated as variables.

## 6.3.2. State to State Transitions in Smv

### 6.3.2.1. Transitions and Variable Assignments

In the translation to smv, a variable named 'trans' is introduced; its purpose is to simulate which among the core module's transitions was chosen. Its type is 'enumerated', and its values are the names of the core module's transitions – {t1,...,tn}. All smv leaf modules will contain a local 'trans' variable.

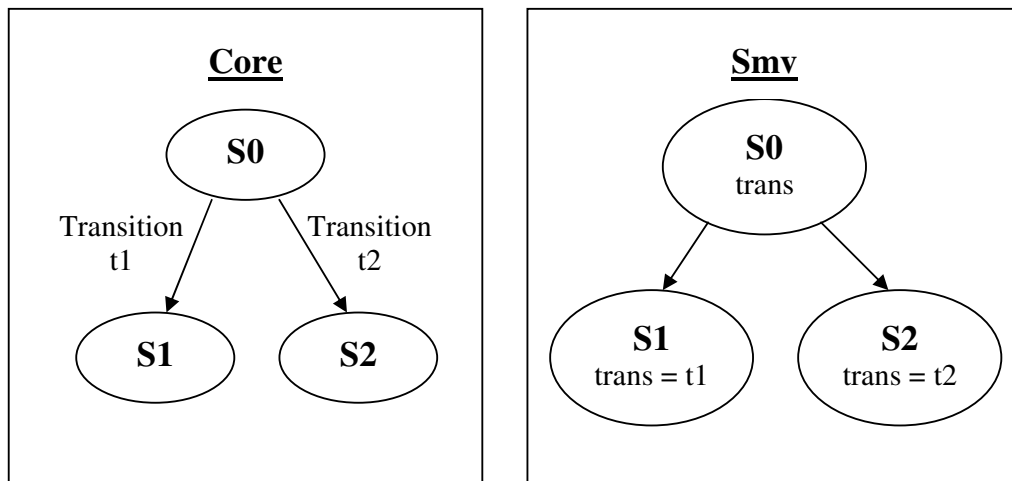
The semantic meaning of this variable is extremely important. All other decisions regarding the state to state transitions in smv depend on it.

The value of the 'trans' variable represents the **last** transition which was executed.

The value of next(trans) represents the **next** transition which will be executed.

In other words, if the model (**note**: model, not module) passed from state S0 to state S1 by executing the core transition t1, the value of 'trans' in S1 is t1 (see also Figure 21 - State to State Transitions).

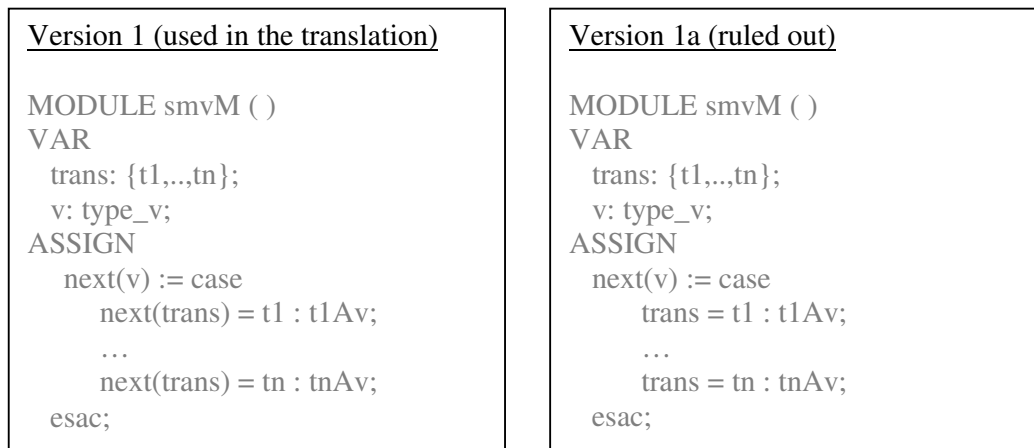
The initial value of the 'trans' variable is undefined, and appropriately so, since no transition caused the model a transformation to its initial state.



**Figure 21 - State to State Transitions**

The option of assigning 'trans' with the transition "about to be executed" has been discussed, and ruled out (in the present translation as well as in the previous versions). Deciding upon which option to use is not arbitrary but logical: It is more appropriate to base the current-state value of a variable on past events, rather than on future events.

Figure 22 - Basic Forms for a Smv Leaf Module serves as a basis for illustrating the form of the translated smv leaf module, as well as demonstrating the two above-mentioned semantic options for the 'trans' variable.



**Figure 22 - Basic Forms for a Smv Leaf Module**

Notice the assignment of v's next-state value: The smv 'case' expression is used. A 'case' expression contains one or more 'case lines'. Each 'case line' is composed of a

condition (left side) and a value (right side). The variable is assigned with the value corresponding to the first condition which is true. If none of the conditions are true, the variable is assigned a value chosen in-deterministically from its range.

The smv assignment to  $v$  means this: If the next-state of the model is determined by executing a core transition  $t_i$ , and  $t_i$  assigns the value  $t_iAv$  to  $v$ , then in the smv model  $v$  is also assigned the value  $t_iAv$ .

### 6.3.2.2. Post-Conditions

A core transition  $t_i$  may also modify variable values by its relation expression (if transition  $t_i$  is chosen and its relation expression is true, and the relation expression contains tagged variables).

Handling variable modifications due to relation expressions is done by exploiting the 'TRANS' section of smv modules. The 'TRANS' section contains an expression connecting current state / next state of the model, which must be true for the next state to exist.

If a variable  $v$  is affected by relation expressions of core transitions  $t_j$  and  $t_k$ , the translation should incorporate a combination of:

- Adding the following case-line to the next() value of the variable  $v$ :  
 $\text{next(trans)} = t_j \mid \text{next(trans)} = t_k : \text{"the range of v"}$ ;
- Using the TRANS expression:  
 $(\text{next(trans)} = t_1 \ \& \ t_1R) \mid \dots \mid (\text{next(trans)} = t_n \ \& \ t_nR)$

The form of the translated smv module would now be:

```

MODULE smvM ( )
VAR
  trans: {t1,..,tn};
  v: type_v;
ASSIGN
  next(v) := case
    next(trans) = t1 : t1Av;
    ...
    next(trans) = tn : tnAv;
    next(trans) = tj | next(trans) = tk : "the range of v"; // (*)
  esac;
TRANS
  (next(trans) = t1 & t1R) | ... | (next(trans) = tn & tnR)
(*) This line appears if the tagged variable (assigned in the next() expression) is
referenced by the relation expressions of transitions tj and tk.

```

The purpose of the 'TRANS' expression is to ensure that a transition to the next state can be chosen only if its relation expression is true. Notice that although  $v$  may seemingly be assigned with any value from its range (according to the newly added

case-line), it will only be assigned in a way satisfying the 'relation' expression (due to the 'TRANS' expression).

The previous Core2Smv versions used the following incorrect 'TRANS' expression:

```
TRANS
  (trans = t1 & t1R) | ... | trans = tn & tnR)
```

Due to incoherency regarding the semantic of the 'trans' variable; again – its value is the **last** transition which was executed, while the value of next(trans) is the **next** transition to be executed.

### 6.3.2.3. Termination States

Generally, calculation paths in smv are defined to be infinite. If a calculation path is finite, it can be viewed as reaching a 'termination' state, and staying forever in this state. To this end, the following adjustments are made:

1. A boolean variable named 'terminate' is inserted to all smv leaf modules. It determines whether a calculation path is finite; its value is true only if there are no enabled transitions from the current state.

'terminate's assignment is (note that its current, **not next**, value is assigned):

```
terminate := case
  !( t1E | t2E | ... | tnE ) : 1;
  1 : 0;
esac;
```

Since 'terminate's current value is assigned, initializing it would be redundant. (Current state assignments include initial state assignments.)

2. All variable assignments will be modified to:

```
next(v) := case
  terminate : v; // This case-line is added
  next(trans) = t1 : t1Av;
  ...
esac;
```

This means that in a termination state, all variables hold their previous values.

3. The 'TRANS' expression is extended to:

```
TRANS
  (next(trans) = t1 & t1R) | ... | (next(trans) = tn & tnR) | terminate
```

This means that if a termination state has been reached, it is always possible to pass to a next state (being the same termination state, since all variables remain unchanged).

In both previous Core2Smv translations, 'terminate' was assigned in exactly the same way.

Adjustments 2 and 3 were not present in the previous Core2Smv translations.

The different status of pre-conditions ('enable' expressions) and post-conditions ('relation' expressions) should be mentioned. A transition to the model's next state can be executed only if the (corresponding) post-condition is met, while transitions to the next state can still occur if none of the pre-conditions are true (but only to the termination state).

Due to the decision to incorporate a termination state, the following possibility for the 'TRANS' expression was discarded:

```
TRANS
  (next(trans) = t1 & t1R & t1E) | ... | (next(trans) = tn & tnR & tnE)
```

#### 6.3.2.4. Avoiding Invalid States

A boolean variable named 'fail' is inserted to all smv leaf modules. This variable will ensure that no transition with a false enable expression is chosen; its value is true only if a transition with a false enable condition is chosen.

The 'fail' variable is assigned

```
next(fail) := case
  terminate : 0;
  (next(trans) = t1 & !t1E) | ... | (next(trans) = tn & !tnE) : 1;
  1 : 0;
esac;
```

This assignment means that

- If "there are no valid transitions to the next state" – then 'fail' is false.
- Otherwise - 'fail' is true only if a transition with a false enable was chosen.

In addition, in order to fulfill the meaning of 'fail', the 'INVAR' section of smv modules is exploited. The 'INVAR' section contains an expression which is an invariant – all states of the model must satisfy this expression.

The 'INVAR' section will simply be

```
INVAR
!fail
```

This means that 'fail' will never be false, and together with 'fail's assignment this means that never will a transition with a false enable be chosen.

'fail' is not initialized. The 'INVAR' expression forces its value to be true in the model's initial state, as in all other states.

In both previous Core2Smv versions, 'fail' was assigned incorrectly.

Version 1.0's assignment to 'fail' was:

```
fail := case
    terminate : 0;
    (trans = t1 & !t1E) | ... | (trans = tn & !tnE) : 1;
    1 : 0;
esac;
```

The faulty reasoning here is that the enable condition (allowing transition to the **next** state) is coupled with the **current** value of 'trans' (which corresponds to the **last** transition executed), instead of coupling it with the **next** value of 'trans' (which corresponds to the **next** transition which will be executed).

The smv model verifier was applied to simple examples using this (incorrect) assignment. It showed that invalid states are reached.

Version 1.8's assignment to 'fail' was:

```
fail := case
    terminate : 0;
    (next(trans) = t1 & !t1E) | ... | (next(trans) = tn & !tnE) : 1;
    1 : 0;
esac;
```

This assignment is syntactically incorrect (since a present state value of a variable can not be determined according to future values of any variables).

### 6.3.3. Format of Leaf Modules (Intermediate Version)

Before discussing other considerations, the general forms of the core and smv leaf modules are presented:

```
MODULE coreM ()
{
  VAR
    v: type_v;
  TRANS t1:
    enable: t1E;
    assign: v' = t1Av;
    relation: t1R;
  ...
  TRANS tn:
    enable: tnE;
    assign: v' = tnAv;
    relation: tnR;
}
```

```

MODULE SMV_M ()
VAR
  trans : {t1,..,tn};
  terminate : boolean;
  fail : boolean;
  v : type_v;
ASSIGN
  terminate := case
    !( t1E | t2E | ... | tnE) : 1;
    1 : 0;
  esac;
  next(fail) := case
    terminate : 0;
    (next(trans) = t1 & ! t1E) | ... | (next(trans) = tn & ! tnE) : 1;
    1 : 0;
  esac;
  next(v) := case
    terminate : v;
    next(trans) = t1 : t1Av;
    ...
    next(trans) = tn : tnAv;
    next(trans) = tj | next(trans) = tk : "the range of v"; // (*)
  esac;
TRANS
  (next(trans) = t1 & t1R) | ... | (next(trans) = tn & tnR) | terminate
INVAR
  !fail
(*) This line appears if the tagged variable (assigned in the next() expression) is
referenced by the relation expressions of transitions tj and tk.

```

### 6.3.4. Out of Bounds Check

If an assigned variable is range-typed and the assigned value is non-numerical (or numerical but out of bounds), then the case-line's condition is supplemented with sub-expressions preventing an out of bounds assignment.

For example, if  $v$  is a range-typed variable having edge values 'low' and 'high', and  $t_i$  assigns the value  $t_iAv$  to  $v$ , then the following case-line

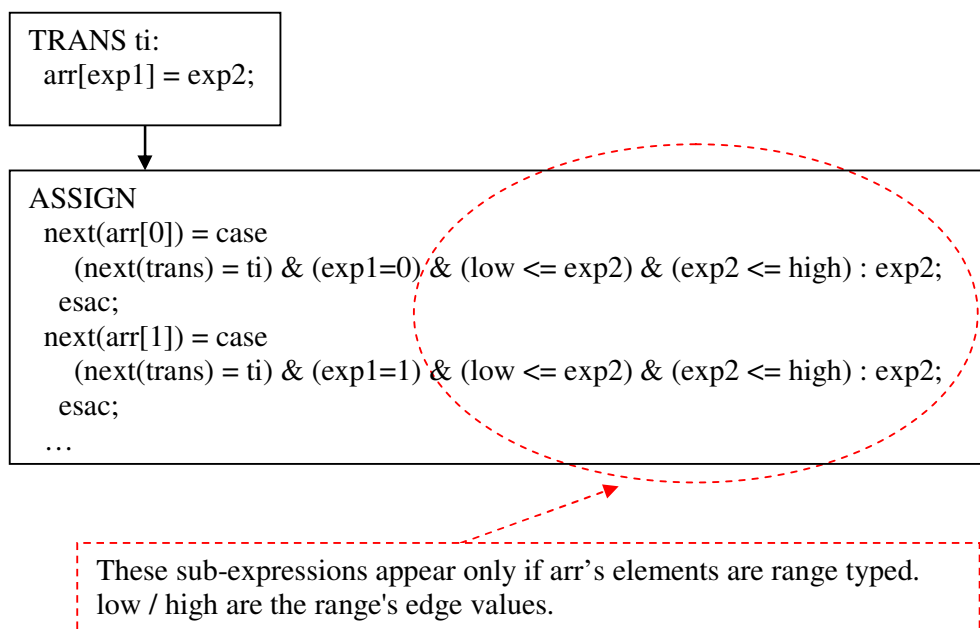
```
next(trans) = ti : tiAv;
```

is replaced by

```
(next(trans) = ti) & (low <= tiAv) & (tiAv <= high) : tiAv;
```

### 6.3.5. Array Element Assignment

Assignment of values to array elements, of the form 'arr[exp1] = exp2', where exp1 is a non-numerical expression, are possible in core language but not allowed in smv. When encountering such an assignment, the expression 'exp1' is checked to see whether it evaluates to a numerical constant. If it does, it can easily be translated, by replacing 'exp1' with its numerical value. Otherwise it is translated in the following manner:



**Figure 23 - Expansion of Array Elements**

To avoid vast expansions of array elements, limits have been set:  
A maximum number of array elements per array (default value of 100).  
A maximum number of array elements per program (default value of 300).  
These default values can be overridden using command line arguments or entries in the configuration file. ([See also Appendix 4, section 5 - Configuration File.](#))

If the limit per variable is exceeded, the translation is aborted and a message is printed.

If the limit per program is exceeded, a warning message is printed; the translation program will continue (without further array expansion), thus resulting in an incorrect smv program. If an array element, created prior to reaching the limit, is encountered again after reaching the limit, it is treated as every other existing variable.

### 6.3.6. The Hold Previous Flag

In case the core program has the 'HOLD PREVIOUS' flag set, all non-assigned variables should keep their current values when passing to the next state. Therefore, all variable assignments are supplemented with an additional case-line:

```
next(v) := case
    terminate : v;
    next(trans) = t1 : t1Av;
    ...
    1 : v;    // this line is added
esac;
```

This means that if the variable was not assigned in any other way, its next-state value is equal to its current-state value.

The special variables 'trans', 'terminate', and 'fail' do not need this assignment; they are not variables of the original core program, and their value does not depend on the existence of the 'HOLD PREVIOUS' flag.

In the previous Core2Smv version, a special (global) boolean variable, named 'hold\_previous', was created; its value set according to the whether the 'HOLD PREVIOUS' flag exists in the core program, and the added case-line to assignments was:

```
globals.hold_previous : v;
```

This variable is not needed, since its (constant) value is known at compilation time.

An issue overlooked in the previous Core2Smv translation is that some variables (especially global variables) are not encountered in a leaf module. Although construction of a 'case' expression for their next-state value may not be necessary, they must be assigned if the 'HOLD PREVIOUS' flag is set.

Therefore, for each variable x (be it a one of the module's local variables, arguments, or a global variable), if a 'case' expression assigning its next value does not yet exist, the following assignment is created:

```
next(x) := x;
```

### 6.3.7. Avoiding Incorrect Assignments

So far, execution of transitions with false enable or relation expressions has been avoided.

A variable assignment is a special case of a relation expression. Even though assignments seem to be correct (properly dependant upon the chosen transition and possibly other conditions), faulty assignments may creep in because assign statements were not checked the same way relation expression were checked. Let us consider the following core program as an example:

```
TYPE
  trinity : 1..3;

MODULE SYSTEM ()
{
  VAR
    x : trinity INITVAL 1;
  TRANS t1:
    enable:true;
    assign:  x' := x+1;
    relation:true;
}
```

It is obvious that the program terminates when  $x=3$ ;  $x$  can not be further incremented because of its range.

The translation to the smv would be:

```
MODULE main
  VAR
    fail : boolean;
    terminate : boolean;
    trans : {t1};
    x : 1..3;
  ASSIGN
    next(fail) := case
      terminate : 0;
      1 : 0;
    esac;
    terminate := 0;
    init(x) := 1;
    next(x) := case
      terminate : x;
      (next(trans) = t1) & (1 <= x + 1)
        & (x + 1 <= 3) : x + 1;
    esac;
  TRANS
    (next(trans) = t1) | terminate
  INVAR
    !fail
```

In the SMV version, we find execution paths containing states after the "x=3" state. Why?

In the next(x) statement, none of the conditions are true. So these conditions will not determine next(x), and the value of next(x) will be chosen in-deterministically from its range.

The solution to this problem would be to extend the 'TRANS' expression to:

```
TRANS
  ( (next(trans) = t1 & t1R & t1A) | ... | (next(trans) = tn & tnR & tnA) ) |
  terminate
```

where t1A,...,tnA are conjunctions of all assignments in the transitions t1,... tn.

The assignment statements are inserted as sub-expressions under the 'TRANS' declaration only when there is a possibility of out-of-range assignments.

Although correct (states with faulty assignments will not be reached), this solution is awkward – the assignment statements already exist inside the next() statements. Of course, if appearing under the 'TRANS' declaration, they are no longer needed as part of the variable assignments. However, without the "direct" assignments, the smv program will lose its clarity. It is definitely more appropriate to assign variables in a section intended for this purpose, rather than "back-door" assignments in other sections.

A different solution for avoiding invalid states and invalid assignments has also been thought of: Extending the use of the 'fail' variable, and discarding the 'TRANS' expression. It was decided not to use the following option:

```
next(fail) = case
  terminate : 0;
  (next(trans) = t1 & !(t1E & t1R & t1A) ) : 1;
  ...
  (next(trans) = tn & !(tnE & tnR & tnA) ) : 1;
  1 : 0;
esac;

INVAR
  !fail
```

This option was rejected since it obscures the distinction between the meaning of 'fail' (avoiding invalid pre-conditions) and 'TRANS' (avoiding invalid post-conditions).

### 6.3.8. Translation Procedure<sup>2</sup>

1. smvM = new empty smv module.
2. Translate local defines.
3. Create new smv variables 'trans', 'terminate', and 'fail'.  
Set their types, and add them to smvM's set of variables.  
([See also under the VAR section of the smv module.](#))
4. Create the current value expression for the 'terminate' variable. ([See also under the ASSIGN section of the smv module.](#))
5. Create the next value expression for the 'fail' variable. ([See also under the ASSIGN section of the smv module.](#))
6. For each local variable v in the VAR section of coreM:
  - a. Create a new smv variable v, with an appropriate type.
  - b. If it has an INITVAL place it in the initvalue field of the smv variable.
  - c. Add the new smv variable to smvM's set of variables.
7. For each variable u passed as argument to coreM:
  - a. Create a new smv variable u, with an appropriate type.
  - b. Add the new smv variable to smvM's list of arguments.
8. For each transition ti in coreM:
  - a. For each assigned variable in ti's assign list, add a case-line to the 'next' expression of the corresponding smv variable.
    - The condition of the case-line is 'next(trans) = ti', ti being the translated name of the transition containing the specified assignment.
    - The value of the case-line is the translated assignment expression.  
([See also case-lines in a variable's next\(\) expression, under the ASSIGN section of the smv module.](#))
    - The assigned variable can be a local variable, an argument, or a global variable.
    - The 'next' expressions of global variables are kept in smvM's external assigns.
  - b. If the assigned core variable is a local array element, a new smv variable is created (if not already existing) representing the array element, and the case-line is added to this variable's 'next' value.
  - c. If the assigned core variable is a global array element, a new smv dotted name is created (if not already existing) representing the array element, and the case-line is added to the smv module's external assigns.
  - d. If the assigned variable is range-typed (or a range-typed array element) and the assigned value is non-numerical (or numerical but out of bounds), then the case-line's condition is supplemented with sub-expressions preventing an out of bounds assignment. ([See also 6.3.4 – Out of Bounds Check.](#))

---

<sup>2</sup> Note: All hyperlinks in this sections, unless otherwise stated, reference sections in the smv module, presented in 6.3.9 – Format of Leaf Modules (Final version).

9. For each core tagged variable appearing in relation expressions of coreM's transitions, add a case-line to the 'next' expression of the corresponding smv variable.
  - a. The case-line condition is 'next(trans) = tj | next(trans) = tk', tj and tk being translated names of transitions referencing the tagged variable in their relation expressions.
  - b. The case-line value is an expression representing the range of values which can be assigned to the variable.
 ([See also the relation case-line in the smv module.](#))
10. Add the 'hold\_previous' case-line to all variables. ([See also the hold previous case-line in the smv module.](#))
11. Create the TRANS expression of smvM. ([See also in the smv module.](#))
12. Create the INVAR expression of smvM. ([See also in the smv module.](#))
13. If coreM contains a cojoin expression – translate it to a smv expression and place it in the INIT section of smvM.
  - The core cojoin is an expression which must be true when a module is instantiated. This is exactly the same as smv's INIT expression.
14. If coreM contains a comment – write to the additional information.
15. Add smvM to the smv program.

### 6.3.9. Format of Leaf Modules (Final Version)

After all the above-mentioned considerations, a typical core leaf module and a final version of the translated smv leaf module can be presented.

```

MODULE coreM ( )
{
  VAR
    v: type_v;
  TRANS t1:
    enable: t1E;
    assign: v' = t1Av;
    relation: t1R;
  ...

  TRANS tn:
    enable: tnE;
    assign: v' = tnAv;
    relation: tnR;
}

```

```

MODULE SMV_M ()
VAR
  trans : {t1,...,tn};
  terminate : boolean;
  fail : boolean;
  v : type_v;
ASSIGN
  terminate := case
    !( t1E | t2E | ... | tnE) : 1;
    1 : 0;
  esac;
  next(fail) := case
    terminate : 0;
    (next(trans) = t1 & ! t1E) | ... | (next(trans) = tn & ! tnE) : 1;
    1 : 0;
  esac;
  next(v) := case
    terminate : v;
    next(trans) = t1 : t1Av; // (#)
    ...
    next(trans) = tn : tnAv; // (#)
    next(trans) = tj | next(trans) = tk : "the range of v"; // (*)
    1 : v; // (**)
  esac;
TRANS
  (next(trans) = t1 & t1R) | ... | (next(trans) = tn & tnR) | terminate // (***)
INVAR
  !fail

```

- (\*) This line appears if the tagged variable (assigned in the next() expression) is referenced by the relation expressions of transitions tj and tk.
- (\*\*) This line appears only if the core program has the 'HOLD PREVIOUS' flag.
- (\*\*\*) If transition tj contains assignments to range type variables, which may possibly be out-of-bounds, the sub-expression  
 $(next(trans) = tj \ \& \ tjR)$   
will be replaced by  
 $(next(trans) = tj \ \& \ tjR \ \& \ tjA)$   
where tjA is the conjunction of all tj's such assignment expressions.
- (#) If the assigned variable is range-typed and the assigned value is non-numerical or numerical but out of bounds, then the case-line is replaced by  
 $(next(trans) = ti) \ \& \ (low \leq tiAv) \ \& \ (tiAv \leq high) : tiAv;$   
where low / high are the range's edge values.

## 7. From Smv Data Structure to XML

This section describes the building of a XML version of a smv program from a smv data structure.

The XML format is determined according to an existing dtd (compiled at VeriTech) describing the structure of the XML version of a smv program.

The smv data structure (described above) is not compatible with the XML structure, and hence requires some transformations, but has the same expressive power. The main reason for the incompatibility of the (predefined) XML structure and the smv data structure (defined by us), is the need for a data structure that would simplify the translation as much as possible while being indifferent to changes in the XML structure definition.

The translation between the two structures is dependant on the dtd that defines the XML structure, and would need to change upon a change to the dtd. This is, however, the only part of the program that would have to be changed in such a case.

### 7.1. Main Parsing Stages

- Creating an empty DOM tree to represent the XML structure.
- Scanning the smv program in the smv data structure module by module, creating corresponding XML elements, and adding them to the DOM tree.
- XML ids are assigned to smv data structure objects upon creation of the corresponding xml elements (using a counter for each element type).
- Dropping the DOM tree to file.
- Java APIs used are
  - javax.xml.parsers
  - javax.xml.transform
  - org.w3c.dom

## 7.2. Adapting to Modifications in the DTD

Creation of XML elements is done according to the dtd. When encountering a smv object in the smv data structure, it is assumed that it corresponds to a specific XML element in the DOM tree. Modifications to the existing dtd would require corresponding modifications to the process of creating XML elements and locating them in the DOM tree.

For example, in the existing dtd, "DEFINE" declarations are defined by:

```
ELEMENT define      (dlist)
ELEMENT dlist       (dstatement*)
ELEMENT dstatement (term, exp)
```

A reasonable modification would be:

```
ELEMENT define      (dstatement*)
ELEMENT dstatement (term, exp)
```

Currently, when encountering the "DEFINE" section of a smv module in the smv data structure, a XML 'define' element is created, a XML 'dlist' element is created and added as child to the 'define' element, and for each (term,exp) pair – a XML 'dstatement' element is created and added as child to the 'dlist' element.

After modification, there is no need for creating the 'dlist' element.

## 8. Gathering Additional Information

### 8.1. The Purpose of Information Gathering

The aim of information gathering is to link each of the source program's components to its corresponding components in the target program, and vice versa. Some such links are trivial, others are not. In this context, "trivial" means that by looking at a target object one can deduce the source object without having the source program at hand. The set of non-trivial links is defined as "additional information". The additional information can be useful for a backwards translation (Smv2Core) producing the original core program in its same textual form.

"Extended information" is defined as the set of all links between source and target objects. The "extended information" is the union of "additional" and "regular" (trivial) information.

Although "regular information" is trivial, its collecting completes the picture of linking all source objects to their corresponding target objects, and vice versa. If only "additional information" is collected, and some source (or target) object is not part of any connection, it is possible that the object has a trivial link, or an un-trivial ("additional") link which for some reason was not listed. In both cases, it is not linked to any corresponding object in the target (or source) program. Supplementing the "additional information" with "regular information" is a means of ensuring that all objects (source and target) are linked to their corresponding counterparts.

This feature is especially useful when applying the XmlViewer, an application showing source and target programs in both textual and XML formats, while highlighting the connections between linked objects.

If not otherwise specified, only additional information will be present in the "addInfo" XML file. Using a command line flag or an entry in the configuration file, this file will contain "extended information".

### 8.2. The Process of Information Gathering

The information is collected during the translation, and printed to a XML file according to an existing dtd. Basically, the information is a list of connections; each connection consists of a source XML id, a target XML id, and a string describing the connection.

Information is collected during the following translation stages:

1. Pre-processing of the core program.
2. Translation from core to smv.
3. Simplifying the smv program's expressions.

(1) Pre-processing is applied to a core program already present in the core data structure, after parsing the input XML file containing it. Therefore, all core objects existing in the original program already have XML ids. During the pre-processing, some original core objects are modified, and some new core objects are created. New objects do not have a XML id. Modified and newly created objects are temporarily linked to original objects.

(2) During translation from core to smv, smv objects are created according to core objects. Each smv object is temporarily linked to a core object (or objects). If the core object has been linked to other core objects (during pre-processing), the smv object is linked to them as well. At this stage, smv objects do not have XML ids.

(3) When simplifying the smv program, some expressions are replaced by new (simpler) expressions. A replacing expression is linked to the core objects linked (during the previous stage) to the replaced expression. Replaced expressions are temporarily listed.

### 8.3. Information Downloading

During composition of the smv XML file (after the three above-mentioned stages) smv objects obtain their XML ids. 'Connection' objects are now created from the source / target objects, and written to the XML "addInfo" file. Replaced smv expressions (not present in the final smv program) are discarded.

The 'addInfo' package is responsible for collecting the temporary information during the various translation stages, organizing it after the translation is completed, and producing a XML file.

When encountering "information items", four parameters are passed to be temporarily stored:

1. A source object.
2. A target object.
3. A string describing the connection.
4. A flag determining whether the item is "additional" or "regular".

The source and target objects may be either core or smv objects, depending on the translation stage.

After all information is collected, 'Connection' objects are composed of:

- Original core objects (having XML ids).
- "Final" smv objects (having XML ids).
- A string describing the connection.

The 'Connection' object contains the first and last objects of a chain of links (original core -> intermediate core -> intermediate smv -> final smv).

If "additional information" is desired, only connections with at least one "additional" flag are listed in the XML file.

If "extended information" is desired, all connections are listed.

The describing string is a concatenation of the description string in the chain. The string descriptions written to the XML file are very similar to the descriptions listed in Table 1 - Additional Information.

If the information item is "regular", the string description simply states the type of the connected objects. String descriptions of "regular" items are prefixed with 'Regular: ', in order to clearly distinguish them from "additional" items. For example, a "regular" connection linking a core variable to its corresponding smv variable consists of the core variable (as source object), the smv variable (as target object), and the string "Regular: Variable" as the describing string.

In rare cases, connections with no source XML id, or no target XML id, are listed ([see also null objects in](#) Table 1 - Additional Information).

## 8.4. Types of Information Items

The following table presents the core-smv connections:

**Table 1 - Additional Information**

<b>Description</b>	<b>Core object</b>	<b>Smv object</b>
Module representing a single transition in partial synchronization (See also <a href="#">5.1 – Elimination of Partial Synchronizations</a> )	Core transition	Smv module
Module representing a set of transitions participating in partial synchronization (See also <a href="#">5.1 – Elimination of Partial Synchronizations</a> )	Set of core transitions	Smv module
Module representing the transitions not participating in partial synchronization (See also <a href="#">5.1 – Elimination of Partial Synchronizations</a> )	Partial synchronization	Smv module
‘process’ synchronization derived from partial synchronization (See also <a href="#">5.1 – Elimination of Partial Synchronizations</a> )	Partial synchronization	Smv synchronization
Module not present in the translation; no longer needed after translation of partial synchronization, or flattening. (See also <a href="#">5.2 – Flattening</a> )	Core module	Null
Module translated from flattened module (See also <a href="#">5.2 - Flattening</a> )	pre-flattened module	Smv module
DEFINE derived from CONST (See also <a href="#">3.3 – Handling Defines and Constants</a> )	CONST declaration	DEFINE declaration
Local define derived from global define (or global constant) (See also <a href="#">3.3 – Handling Defines and Constants</a> )	Global define (or global constant)	Local define
Type 0..x-1 derived from SCALARSET(x) (See also <a href="#">3.4 – Handling Typedefs and Variable Types</a> )	Scalarset type	Range type
Range type derived from integer type (See also <a href="#">3.4 – Handling Typedefs and Variable Types</a> )	Integer type	Range type
Variable type derived from TypeDef (See also <a href="#">3.4 – Handling Typedefs and Variable Types</a> )	Typedef	Variable type
Identifier changed due to syntactic rules (See also <a href="#">Appendix 1 – Translation of Identifiers</a> )	Core identifier	Smv identifier

'GLOBALS' module ( <a href="#">See also 6.1.5 - Main Translation Steps</a> )	Global variables' node	'GLOBALS' module
'GLOBALS' module passed as argument ( <a href="#">See also 6.1.5 - Main Translation Steps</a> )	Global variables' node	The argument
Each variable in 'GLOBALS' module ( <a href="#">See also 6.1.5 - Main Translation Steps</a> )	Global variable	The variable
'trans' variable ( <a href="#">See also 6.3.2.1 – Transitions and Variable Assignments</a> )	All the transitions it can represent	'trans' variable
'terminate' variable in a leaf module ( <a href="#">See also 6.3.2.3 – Termination States</a> )	All the 'enable' expressions it refers to	'terminate' variable
'terminate' variable in a non-leaf module ( <a href="#">See also non-leaf 'terminate' variable, bullet 7 in 6.2.1 – Translation Procedure</a> )	Null	'terminate' variable
'fail' variable ( <a href="#">See also 6.3.2.4 – Avoiding Invalid States</a> )	All the 'enable' expressions it refers to	'fail' variable
case-line in a next() assignment ( <a href="#">See also creation of case-lines, bullet 8 in 6.3.8 – Translation Procedure</a> )	'assign' expression	case-line
case-line sub-expression resulting from tagged variable appearing in relation expression ( <a href="#">See also relation case-line, bullet 9 in 6.3.8 – Translation Procedure</a> )	'relation' expression	sub-expression
case-line added due to Hold Previous flag ( <a href="#">See also hold previous case-line, bullet 10 in 6.3.8 – Translation Procedure</a> )	HOLDPREVIOUS node	case-line
Low and high bound condition expressions added to a case-line in order to prevent an out-of-range assignment ( <a href="#">See also 6.3.4 – Out of Bound Check</a> )	'assign' expression	The added condition sub-expressions
case-line in a next() assignment of an array element derived from an 'assign' expression having the form a[exp1]:=exp2 ( <a href="#">See also 6.3.5 - Array Element Assignment</a> )	'assign' expression having the form a[exp1]:=exp2	case-line
'trans = ti & tiR' in a TRANS expression ( <a href="#">See also the TRANS expression of the smv module in 6.3.9 – Format of Leaf Modules</a> )	relation expression	sub-expression
Sub-expression in INIT derived from cojoin ( <a href="#">See also INIT expression, bullet 13 in 6.3.8 – Translation Procedure</a> )	Cojoin	sub-expression
Core comment ( <a href="#">See also comment encounter, bullet 14 in 6.3.8 – Translation Procedure</a> )	Comment	Null

# Bibliography

## Core Language

Shahar Dag, *The Core Language*, last updated 18/11/2002.

[www.cs.technion.ac.il/~ssdl/veritech/core/CDL\\_language.pdf](http://www.cs.technion.ac.il/~ssdl/veritech/core/CDL_language.pdf)

## Smv Language

K.L.McMillan, *The SMV System*, last updated 6/11/2000.

[www.cs.cmu.edu/~modelcheck/smv/smvmanual.ps](http://www.cs.cmu.edu/~modelcheck/smv/smvmanual.ps)

## Previous Translations

1. Efrat Shabtai, *Core to smv translation*, 9/2002. (Currently located on the csd server at veritech\_home/VeriTech.all/core\_2\_smv 1.0/doc/)
2. Yoav Gur & Alon Zvirin, *Core To Smv project*, 7/2005. (Currently located on the csd server at veritech\_home/VeriTech.all/core\_2\_smv 1.8/doc/project.pdf)

## Generic Translations

1. Shmuel Katz and Orna Grumberg, *A Framework for Translating Models and Specifications*, 2002.  
[www.cs.technion.ac.il/~ssdl/veritech/Veritech.Doc/overview.ps](http://www.cs.technion.ac.il/~ssdl/veritech/Veritech.Doc/overview.ps)
2. Orna Grumberg and Shmuel Katz, *Faithful Translations Among Models and Specifications in VeriTech*, 5/2000.  
[www.cs.technion.ac.il/~katz/faithfme2001.ps](http://www.cs.technion.ac.il/~katz/faithfme2001.ps)

## Additional Information

*Representation of Additional Information*

[www.cs.technion.ac.il/~ssdl/veritech/xml/add\\_info.ps](http://www.cs.technion.ac.il/~ssdl/veritech/xml/add_info.ps)

## dtd Files

[www.cs.technion.ac.il/~ssdl/veritech/xml/dtd/add\\_info.dtd](http://www.cs.technion.ac.il/~ssdl/veritech/xml/dtd/add_info.dtd)

[www.cs.technion.ac.il/~ssdl/veritech/xml/dtd/cdl.dtd](http://www.cs.technion.ac.il/~ssdl/veritech/xml/dtd/cdl.dtd)

[www.cs.technion.ac.il/~ssdl/veritech/xml/dtd/smv.dtd](http://www.cs.technion.ac.il/~ssdl/veritech/xml/dtd/smv.dtd)

## xsl Files

[www.cs.technion.ac.il/~ssdl/veritech/xml/xsl/xml2addinfo.xsl](http://www.cs.technion.ac.il/~ssdl/veritech/xml/xsl/xml2addinfo.xsl)

[www.cs.technion.ac.il/~ssdl/veritech/xml/xsl/xml2cdl.xsl](http://www.cs.technion.ac.il/~ssdl/veritech/xml/xsl/xml2cdl.xsl)

[www.cs.technion.ac.il/~ssdl/veritech/xml/xsl/xml2smv.xsl](http://www.cs.technion.ac.il/~ssdl/veritech/xml/xsl/xml2smv.xsl)

# Appendix 1 - Translation of Identifiers

As much as possible, smv identifiers are exactly the same as their originating core identifiers. Some identifiers must be changed, due to:

- Naming conflicts arising during the translation process.
- Reserved names used by the translation program.
- Reserved smv names.
- Syntax-imposed restrictions.

## 1. Naming Conflicts

Naming conflicts are resolved by suffixing the identifier by a unique integer. For example, if a core module M has three son modules, all of which are instances of the same module A, the core representation might be

```
MODULE M
  A () || ( A () || A () )
```

While the smv representation would be

```
MODULE M
  A_instance : A;
  A_instance1 : A;
  A_instance2 : A;
```

An improvement to this representation is suggested in [Appendix 6 – Future Prospects, section 7 – Resolving Naming Conflicts](#).

## 2. Reserved Names

Special names used by the translation program are:

trans, terminate, fail, SYSTEM, main, GLOBALS, globals.

Reserved smv names are:

A, AF, AG, array, ASSIGN, AX, boolean, case, DEFINE, E, EF, EG, EX, esac, FAIR, FAIRNESS, in, INIT, init, INVAR, mod, MODULE, module, next, process, SPEC, TRANS, U, union, VAR.

### 3. Syntax Restrictions

Core identifiers are regular expression having the form

[A-Z, a-z, &, \_] [A-Z, a-z, 0-9, \_, \$, #, -, \]\*

Smv identifiers are regular expression having the form

[A-Z, a-z] [A-Z, a-z, 0-9, \_,-]\*

The '&' character causes problems in XML, and is not allowed in smv. This character has been omitted or modified wherever it appeared in test-examples. My suggestion is to cancel it altogether.

The characters '\$', '#', '\ ' have not been encountered in test-examples.

The '\$' character is used in the translation (in transition merging and when concatenating module names, but never two successive appearances of '\$'). My suggestion is that wherever it appears, double it.

The '\ ' character also causes problems in XML. My suggestion is to cancel it.  
The '#' character has not been taken care of.

### 4. Resolving Syntax Restrictions and Reserved Names

The following routine translates a core name to a smv name:

1. Check the core name.
2. If it is 'SYSTEM', change it to 'main'.
3. If the first character is '\_ ', or if it is a reserved name, appearing in the above lists, add the prefix '\_ ' to the name.

This way,

All smv names not prefixed with '\_ '  
have either the same name as in core  
or are reserved names.

All smv names starting with exactly one '\_ ' character are:  
names having this prefix as conflict resolution with reserved names.

All smv names starting with two or more '\_ ' characters are:  
names having the core prefix '\_ '

## Appendix 2 - Simplifying Smv Expressions

Smv expressions are simplified after the whole smv program has been built. They are not simplified upon construction since some expressions are constructed gradually during the translation process.

Expressions are recursively simplified; sub-expressions are simplified prior to the expression containing them. The following simplifications are used:

- String "true" / "false" expression => constant logical 'true' / 'false' expression.
- Elimination of redundant parenthesis.
- !true => false ; !false => true
- !(!exp) => exp
- -(-exp) => exp
- !(exp1 = exp2) => exp1 != exp2
- !(exp1 != exp2) => exp1 = exp2
- exp & true => exp ; true & exp => exp
- exp & false => false ; false & exp => false
- exp | false => exp ; false | exp => exp
- exp | true => true ; true | exp => true
- In a NWayAnd expression (exp1 & exp2 & ... & exp\_n) :
  - If one of the sub-expressions is false => the whole expression becomes false.
  - Remove all 'true' sub-expressions.
  - If after removal of 'true' sub-expressions the remaining number of sub-expressions is:
    - 0 => return a 'true' expression
    - 1 => return the single sub-expression
    - 2 => return an 'And' expression containing them.
- In a NWayOr expression (exp1 | exp2 | ... | exp\_n) :
  - If one of the sub-expressions is true => the whole expression becomes true.
  - Remove all 'false' sub-expressions.
  - If after removal of 'false' sub-expressions the remaining number of sub-expressions is:
    - 0 => return a 'false' expression
    - 1 => return the single sub-expression
    - 2 => return an 'Or' expression containing them.
- In a case-line expression – simplify its condition and value expressions.
- In a case expression:
  - Simplify its case line expressions.
  - If a case-line's condition is false – remove the case-line.
  - If a case-line's condition is true – remove all following case-lines.
  - If after removal of case-lines the remaining number of case-lines is:
    - 0 => return null (the caller will recognize this and treat it accordingly).
    - 1 => If the (single) case-line's condition is true – return the case-line's value.  
Otherwise, return the simplified (single case-lined) case expression

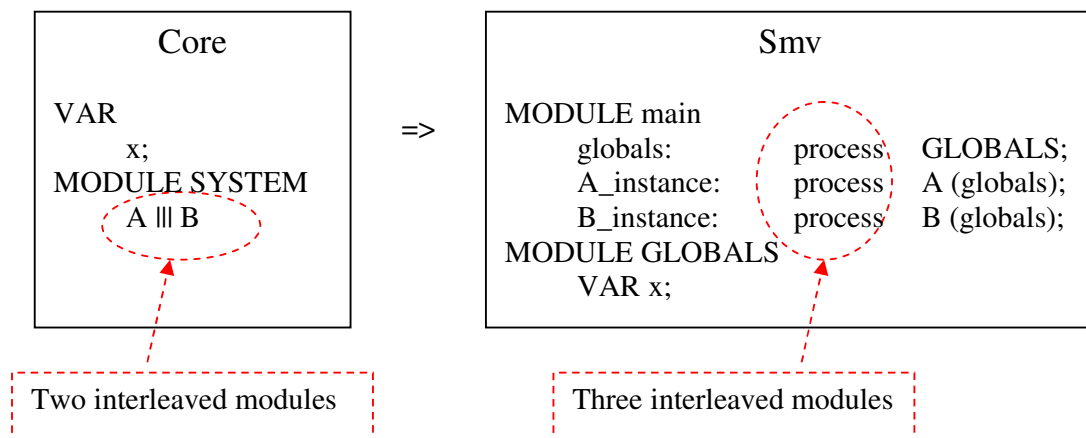
# Appendix 3 - Does Treatment of Global Variables Affect the Synchronization Structure?

## 1. Outline

Core global variables are handled by locating them in a special-purpose smv 'GLOBALS' module, and this module (as any other module in the smv program) participates in the program's synchronization structure. Therefore, the issue of affecting the program's synchronization structure must be addressed.

In some cases, as will be shown, this treatment of global variables causes an inconsistency between the core / smv synchronization structures. The reason for this inconsistency is that the smv 'GLOBALS' module has a "synchronization status" equal to its sibling-modules, therefore having an active role in determining the program's execution paths. In core, global variables have no such status.

Figure 24 - Global Variables and Synchronization serves as a basis for the discussion:



**Figure 24 - Global Variables and Synchronization**

The discussion is relevant when the core 'SYSTEM' module has interleaved synchronization, and the core program contains global variables.

A simple example is presented, demonstrating the inconsistency.

The SMV model checker was used. It showed that the problem remains "theoretical". Two solutions are presented. Currently, they are not implemented, but remain as options for future use.

It may be that I have not completely understood smv semantics. Also, the inconsistency may remain hidden in some examples, and surface in others.

## 2. Current Treatment of Global Variables

Core global variables are handled by:

1. Creating a special smv 'GLOBALS' module.
2. Locating the (translated) global variables in the 'GLOBALS' module.
3. If a core variable has an init-value, the initial value is assigned inside the 'GLOBALS' module. (Initial values are optional in both languages.)
4. The 'GLOBALS' module is added as child module to the 'main' module.
5. The 'GLOBALS' module is passed as argument to all other smv modules, except to the 'main' module and the 'GLOBALS' module itself.

## 3. Example – Case Study

Let us consider the following core program (written shorthand):

```
VAR
  x: {0,1,2} INITVAL 0; // global variable
MODULE SYSTEM
  A ||| B // notice the interleaving of A and B
MODULE A
  x' := 1;
MODULE B
  x' := 2;
```

The translation to smv is (again, written shorthand):

```
MODULE main
  globals: process GLOBALS;
  A_instance: process A (globals);
  B_instance: process B (globals);
MODULE GLOBALS
  VAR x : {0,1,2};
  init(x) := 0;
MODULE A (globals)
  next(globals.x) := 1;
MODULE B (globals)
  next(globals.x) := 2;
```

It should be noted that 'GLOBALS' contains only init() assignments; it never contains next-state assignments.

In core, it is obvious that x is initialized with 0; afterwards, forever, it is assigned indeterministically with a value from the sub-set {1,2}.

In smv, theoretically, other scenarios are possible:

1. A\_instance (or B\_instance) is chosen to be executed prior to 'globals', resulting in x not being initialized with 0.
2. During the (infinite) execution sequence, 'globals' may be chosen to be executed. Since it does not assign a next-state value to x, this value is chosen indeterministically from x's range, possibly assigning 0 to x.
3. In all steps of the execution sequence, 'globals' is chosen. (Very much opposed to the core execution sequence.)

The SMV model checker proved that these scenarios never occur. So I inferred that:

1. If a variable is initialized somewhere in the program, it is never assigned prior to the initialization. In the above example, A\_instance and B\_instance are never executed prior to the execution of the 'globals' instance.
2. A module-instance containing only init() assignments is never executed after initializations. In the above example, after A\_instance (or B\_instance) have been executed, the 'global's instance is never executed.

These observations are definitely not trivial. The smv manual does not refer to these cases.

## 4. Possible Solutions

### Solution 1

Abandoning the idea of a 'GLOBALS' module. All global variables will be passed as arguments to all smv modules (or at least to those referencing them). No other modifications are needed. This solution is very awkward when global variables are numerous, and has been rejected.

### Solution 2

Smv allows modules a "fairness" property, thus ensuring that a module under interleaved synchronization is executed infinitely often. The smv module A (and B, similarly) would be modified to:

```
MODULE A (globals)
  next(globals.x) := 1;
  FAIRNESS
    running
```

This modification ensures that A and B will be executed infinitely often. Assuming that 'globals' is never executed after initialization (due to its lack of next() assignments), this modification is enough.

However, if this assumption is rejected, a further modification is introduced, using an intermediate module:

```
MODULE main
  globals:  GLOBALS;
  actual_program: actual_program (globals);
MODULE GLOBALS
  VAR x : {0,1,2};
  init(x) := 0;
MODULE actual_program (globals)
  A_instance:process A (globals);
  B_instance:process B (globals);
MODULE A (globals)
  next(globals.x) := 1;
MODULE B (globals)
  next(globals.x) := 2;
```

'main' has full synchronization over its sons; 'actual\_program' has interleaved synchronization over its sons.

In each execution step of this suggested solution, a synchronized combination of two modules will be executed – either GLOBALS and A, or GLOBALS and B. Since A (and B) assign x, and GLOBALS doesn't, the assignment is determined according to A (or B). This will rule out the possibility of a non-deterministic assignment to x by GLOBALS.

Should this solution be implemented, the dtd and xsl files must be modified accordingly (incorporating a XML 'FAIRNESS' element).

# Appendix 4 – Programming Notes

## 1. Relevant Files

The Core2Smv project includes the following files and directories, currently located on the csd server at:

~veritech/Veritech/Core2Smv 2.0/

- readme.txt:
  - Command line usage for running the Core2Smv translation program.
  - Command line usage for running the Flat program.
- Executable jar files:
  - Core2Smv.jar – executable for running the translation program.
  - Flat.jar – executable for flattening the entire core program.
- Script files:
  - Core2Smv
  - Flat
  - DoAll – script for running all the examples.
  - UnDoAll - script for deleting all the products of DoAll.
- Makefile - makefile for compiling sources and creating executables.
- Manifest files:
  - Manifest.txt - marks the 'main' class to be run by Core2Smv.jar
  - Manifest\_flat - marks the 'main' class to be run by Flat.jar
- config.txt – configuration file.
- db\_alon.xml – database file for web-generation of the Core2Smv project.
- src - a subdirectory containing the java source files.
- docs - a subdirectory containing documentation
  - docs/Core2Smv.doc – this file.
  - docs/User Guide.doc – containing information useful for running the program.
  - docs/variable\_assignments.doc – intermediate article discussing variable assignments and state-to-state transitions.
  - docs/partial\_synchronization.doc – intermediate article discussing elimination of partial synchronizations in a core program.
  - docs/comments.doc – a few comments on the input files and external tools.
  - docs/javadocs – a subdirectory with 'JavaDoc' documentation
- examples - a subdirectory containing tested examples ([See also](#) Appendix 7 - Examples.)
- eclipse\_prj – a subdirectory containing eclipse project settings.

Other files used by the translation are dtd and xsl files (compiled at Veritech); their url's are:

[www.cs.technion.ac.il/~ssdl/veritech/xml/dtd/cdl.dtd](http://www.cs.technion.ac.il/~ssdl/veritech/xml/dtd/cdl.dtd)  
[www.cs.technion.ac.il/~ssdl/veritech/xml/dtd/smv.dtd](http://www.cs.technion.ac.il/~ssdl/veritech/xml/dtd/smv.dtd)  
[www.cs.technion.ac.il/~ssdl/veritech/xml/dtd/add\\_info.dtd](http://www.cs.technion.ac.il/~ssdl/veritech/xml/dtd/add_info.dtd)  
[www.cs.technion.ac.il/~ssdl/veritech/xml/xsl/xml2cdl.xsl](http://www.cs.technion.ac.il/~ssdl/veritech/xml/xsl/xml2cdl.xsl)  
[www.cs.technion.ac.il/~ssdl/veritech/xml/xsl/xml2smv.xsl](http://www.cs.technion.ac.il/~ssdl/veritech/xml/xsl/xml2smv.xsl)  
[www.cs.technion.ac.il/~ssdl/veritech/xml/xsl/xml2addinfo.xsl](http://www.cs.technion.ac.il/~ssdl/veritech/xml/xsl/xml2addinfo.xsl)

It is recommended to read the user guide. It includes instructions for running the translation program, limitations of the program, and contact information.

## 2. Script Files

The script accepts the command line arguments and invokes the executable jar file. As described in the user guide, when the command line does not include the '-xml' flag, the CoreToXml application is invoked prior to the jar, thus producing a xml file representing the core program. Otherwise, the xml file is assumed to exist, and the CoreToXml application is not invoked.

Command line arguments are checked; if invalid – a correct 'usage' message is printed. The correct usage is the one found in readme.txt

## 3. Makefile

The Makefile contains the makefile information for creating the executable jar files from the java source files under the 'src' directory.

It is possible to invoke the Makefile using the following commands:

- make: the "normal" usage. This command creates java '.class' files from the '.java' source files under the 'src' subdirectory, and places them under a 'bin' subdirectory. Then the executable files Core2Smv.jar and Flat.jar are created from these classes. This command also creates "javadoc" documentation, placing the files in the subdirectory 'docs/javadocs' ([see also section 8 – Producing Javadocs, in this appendix](#)).
- make Core2Smv.jar : This command creates java '.class' files from the '.java' source files under the 'src' subdirectory, and places them under a 'bin' subdirectory. Then, the executable file Core2Smv.jar is created from these classes
- make Flat.jar : This command creates java '.class' files from the '.java' source files under the 'src' subdirectory, and places them under a 'bin' subdirectory. Then, the executable file Flat.jar is created from these classes.
- make clean : This command removes all products (if existing) created by the 'make Core2Smv.jar' and 'make Flat.jar' commands (in this case – the jar files, the bin subdirectory and its contents, and the 'javadoc' documentation).
- make all : Similar to the 'make' command. However, this command creates the desired outputs regardless if they are actually needed (as opposed to the "normal" usage, creating the outputs only if they do not already exist or if their sources were modified since their previous make.
- make release : Similar to the 'make' command. However, this command also removes all unnecessary intermediate products (the bin subdirectory and its contents, in this case).

## 4. Manifest Files

A Manifest file is needed in order to mark the class containing the 'main' method of the jar files.

'Core2Smv' contains the 'main' method of the Core2Smv translation program.

'Flat' contains the 'main' method of the flattening program.

## 5. Configuration File

The configuration file (config.txt) contains several parameters used by the translation program.

When running the translation program, the configuration file is looked for, and the contained parameters are extracted. If it is not found, a default configuration file is created. If still not found, or if it is syntactically incorrect, the translation is aborted and an error message is printed.

The form of the default configuration file is:

```
MIN_INT = -100
MAX_INT = 100
MAX_ARRAY_ELEMENTS_PER_VARIABLE = 100
MAX_ARRAY_ELEMENTS_PER_PROGRAM = 300
SMV_DTD_FILE = "http://www.cs.technion.ac.il/~ssdl/
                veritech/xml/dtd/smv.dtd"
ADD_INFO_DTD_FILE = "http://www.cs.technion.ac.il/~ssdl/
                    veritech/xml/dtd/add_info.dtd"
INCLUDE_EXTENDED_INFO = false
SHOULD_FLAT = false
PRINT_DATA_STRUCTURES = false
```

[For more information regarding these parameters and command line usage, refer to 'User Guide.doc', section 3 – Running the Core2Smv Translation Program and section 6 – Configuration File & Default Values.](#)

## 6. Packages & Source Files

The core2smv project is composed of the following packages:

- translation – a package containing the main stages of the translation process: Xml2CoreDataStructure, Pre-Translator, Translator, and Smv2XmlDataStructure.
- coreDataStructure – the data base containing and managing the translator's representation of the inputted core program. This package includes two sub-packages:
  - coreExpression – handling core language expressions.
  - coreType – handling core language types.
- smvDataStructure - the data base serving for building the translator's representation of the smv program expected as the program output. This package includes two sub-packages:
  - smvExpression - handling smv language expressions.
  - smvType - handling smv language types.
- addInfo – collection and retrieval of the additional information gathered during the translation.
- utilities – various special-purpose utilities.

## 6.1. translation package

This package contains the components acting as main stages of the translation.

- Core2Smv - 'main' method running the translation: parsing command line arguments, creating the components needed for the translation, and executing the translation stages.
- Flat - 'main' method for flattening the core program: parsing command line arguments, creating the components needed for flattening, and executing the flattening stages.
- Xml2CoreDataStructure – responsible for reading and parsing the core xml file, written according to the supplied dtd, and building a core data structure representing the core program. ([See also 4 – From XML to Core Data Structure.](#))
- PreTranslator – pre-processing of the core program residing in the core data structure. ([See also 5 – Pre-Translation.](#))
- Translator – translates a core program (defined by the core data structure) to a smv program (defined by the smv data structure). ([See also 6 – Translation of a Core Program.](#))
- SmvDataStructure2Xml – responsible for creating the xml file of the smv program from the smv data structure. ([See also 7 – From Smv Data Structure to XML.](#))
- Core2SmvException (extending Exception) – base class for all special exceptions raised during the translation. ([See also section 7 in this appendix - Errors, Exceptions & Warnings.](#))
- XmlParserError – a general error handler needed for parsing a XML file.

## 6.2. coreDataStructure package

This package serves as a data base representing a core program and its components. The data base is built by the Xml2CoreDataStructure component and referenced by the translator component. ([See also 3.1 – Core Data Structure.](#))

- Core\_Object – base class for all core data structure objects, holding a unique xml id. This class has a 'clone()' method; therefore all derived classes in need of deep cloning must override 'clone()'.
- Core\_String – holds string values (identifiers and constants). Core\_String overrides Java Object's equal() method. Core strings are considered equal if their string values are equal. Core\_String also overrides Java Object's hashCode() method, returning the hash-code of the contained string value.
- Core\_StringCouple – a pair of core strings. Used for specifying two synched transitions in partial synchronization, or renaming two transitions under a single name.
- Core\_Program – representation of a core program.
- Core\_Module – base class for core modules:
  - Core\_LeafModule
  - Core\_NonLeafModuleA Core module can be cloned in several different ways: as leaf, as non-leaf, with or without a set of transitions.
- Core\_ModuleInstance – instance of a child module inside another module; has a name of the referenced child and a list of arguments passed to the child.

- Core\_Modcombin – base class for core modcombins:
  - Core\_InstanceModcombin
  - Core\_SynchedModcombin
  - Core\_UnSynchedModcombin
  - Core\_PartSynchedModcombin
- Core\_Synchronization – base class for synchronization types:
  - Core\_Synchronized
  - Core\_UnSynchronized
  - Core\_OneInstanceSynch
- Core\_Variable – representing variables (local and global), and arguments.
- Core\_Transition – representing a core transition, with its expressions.
- Core\_String2ExpHashMap – mapping of core strings to core expressions.
- Core\_String2TypeHashMap – mapping of core strings to core types.
- Core\_NullObject – dummy object.
- Core\_ProgramException (extending Core2SmvException) – is raised in case a problem occurs when processing the core program.

### 6.2.1. coreDataStructure.coreExpression package

- Core\_Expression (extends Core\_Object) – base class for all core expressions.
- Core\_SimpleExp – simple string expression.
- Core\_UnaryExp – base class for unary expressions.
  - Core\_NotExp – logical not.
  - Core\_ParExp – parenthesis.
  - Core\_UnaryMinus – unary minus.
  - Core\_TagExp – tagged expression.
- Core\_BinaryExp – base class for binary expressions:
  - Logical binary operators - Core\_AndExp, Core\_OrExp, Core\_EqualExp, Core\_NotEqualExp, Core\_LTEExp, Core\_LEExp, Core\_GTEExp, Core\_GEExp.
  - Arithmetical binary operators – Core\_PlusExp, Core\_MinusExp, Core\_MulExp, Core\_DivExp, Core\_ModExp.
- Core\_NnaryExp – base class for n-nary expressions:
  - Core\_NWayAndExp, Core\_NWayOrExp – logical and/or operators over n sub-expressions.
- Core\_SetExp – enumerated set of values.
- Core\_ArrayElementExp – indexed array element.

Notes on core expressions:

1. All core expressions must implement the 'asString()' method, returning a Core\_String containing a string representation of the expression.
2. Core\_Expression overrides Java Object's equal() method. Core expressions are considered equal if their string representations are equal. Core\_Expression also overrides Java Object's hashCode() method, returning the hash-code of the string representation.
3. All core expressions must implement the 'replace(String,Core\_Expression)' method, replacing all appearances of a string with an expression.
4. All core expressions must implement the 'contains(String)' method, checking if a string is contained in the expression.

## 6.2.2. coreDataStructure.coreType package

- Core\_Type (extends Core\_Object) – base class for all core types.
- Various types - Core\_BooleanType, Core\_IntegerType, Core\_RangeType, Core\_ScalarSetType, Core\_ArrayType, Core\_EnumType, Core\_UserType.

Notes on core types:

1. All core types must implement the 'asString()' method, returning a Core\_String containing a string representation of the type.
2. All core types containing inner expressions must implement the 'replace(String,Core\_Expression)' method, replacing all appearances of a string with an expression.

## 6.3. smvDataStructure package

The data base for the smv program and its components, built by the translator and referenced by the SmvDataStructure2Xml component. ([See also 3.2 – Smv Data Structure.](#))

- Smv\_Object – base class of all smv data structure objects, holding a unique xml id.
- Smv\_String – holds string values (identifiers and constants). Smv\_String overrides Java Object's equal() method. Smv strings are considered equal if their string values are equal. Smv\_String also overrides Java Object's hashCode() method, returning the hash-code of the contained string value
- Smv\_Program – representation of a smv program.
- Smv\_Module – a smv module with its contents.
- Smv\_ModuleInstance – instance of a child module inside another module. Contains a formal name and a Smv\_UserType; the user-type contains a name of the referenced child and a list of arguments passed to the child.
- Smv\_Synchronization – synchronization type; can be either 'process' or 'no process'
- Smv\_Variable – representing variables and arguments
- Smv\_DottedName – inner references (a.b denoting component b of module a).
- Smv\_String2ExpHashMap – mapping of smv strings to smv expressions.
- Smv\_String2TypeHashMap – mapping of smv strings to smv types.
- Smv\_NullObject – dummy object.

### 6.3.1. smvDataStructure.smvExpression package

- Smv\_Expression (extending Smv\_Object) – base class for all smv expressions.
- Smv\_SimpleExp – simple string expression.
- Smv\_UnaryExp – base class for unary expressions:
  - Smv\_NotExp – logical not.
  - Smv\_ParExp – parenthesis.
  - Smv\_UnaryMinus – unary minus.
  - Smv\_NextExp – 'next' expression.

- Smv\_BinaryExp – base class for binary expressions:
  - logical binary operators - Smv\_AndExp, Smv\_OrExp, Smv\_EqualExp, Smv\_NotEqualExp, Smv\_LTExp, Smv\_LEExp, Smv\_GTExp, Smv\_GEExp.
  - arithmetical binary operators – Smv\_PlusExp, Smv\_MinusExp, Smv\_MulExp, Smv\_DivExp, Smv\_ModExp.
- Smv\_NnaryExp – base class for n-ary expressions:
  - Smv\_NWayAndExp, Smv\_NWayOrExp – logical and/or operators over n sub-expressions.
- Smv\_TrueExp, Smv\_FalseExp – constant logical expressions.
- Smv\_CaseExp – case expression.
- Smv\_CaseLine – line in a case expression (condition + value).
- Smv\_EnumExp – enumerated set of values.
- Smv\_ArrayElementExp – indexed array element.

Notes on smv expressions:

1. All smv expressions must implement the 'asString()' method, returning a Smv\_String containing a string representation of the expression.
2. All smv expressions must implement the 'isNumeric()' method, checking if the expression evaluates to a constant numerical expression.
3. All smv expressions must implement the 'simplify()' method, recursively simplifying the expression.

### 6.3.2. smvDataStructure.smvType package

- Smv\_Type (extending Smv\_Object) – base class for all smv types.
- Various types - Smv\_BooleanType, Smv\_IntegerType, Smv\_RangeType, Smv\_ArrayType, Smv\_ArrayElementType, Smv\_EnumType, Smv\_UserType.
- ArrayOverflowException (extends Core2SmvException) – raised in case an overflow will occur in the maximum number of array elements allowed.

Notes on smv types:

1. All smv types must implement the 'asString()' method, returning a Smv\_String containing a string representation of the type.

### 6.4. addInfo package

- XmlWriter – responsible for collecting the additional information and writing it to a XML file according to the dtd. This information is represented by a set of connections. ([See also 8 – Gathering Additional Information.](#))
- Change - – represents the 'change' element in add\_info.dtd.
- Connection – represents the 'connection' element in add\_info.dtd.
- Ref – representing the 'source' and 'target' elements in add\_info.dtd.
- ConnectionActuate – tokens of the 'actuate' xlink field in a connection.
- ConnectionShow – tokens of the 'show' xlink field in a connection.
- CoreItem – represents a link to a core object. This might be a temporary core-to-core link, a temporary core-to-smv link, or a final core-to-smv link.

## 6.5. utilities package

- BookKeeper – keeps information relevant at the pre-translation stage. ([See also 5.1.8.4 – Book-Keeping.](#))
- Configuration – handles configuration issues ([See also section 5 in this appendix - Configuration File](#)):
  - Creating a default configuration file.
  - Reading parameters from the configuration file.
  - Storage and retrieval of user parameters.
- ConfigurationException (extends Core2SmvException) – raised in case any problem relating to the configuration file occurs. ([See also section 7 in this appendix - Errors, Exceptions & Warnings.](#))
- CoreFlattener – handles flattening issues:
  - Flattening core modcombins, and the decision whether to flatten a modcombin. ([See also 5.2 - Flattening](#))
  - Flattening of the whole core program. ([See also 5.2.6 – Flattening the Whole Core Program.](#))
- ExpressionHandler – processing of expressions:
  - Translation of core expressions to equivalent smv expressions.
  - Checking whether expressions are numerical; if so, replacing them with an expression holding a numerical value.
- FileNames – handles file names used by the translation program. ([See also Appendix 7, section 2 - File Naming Conventions.](#))
- Identifierhandler – translation of names and identifiers. ([See also Appendix 1 – Translation of Identifiers.](#))
- ProgramParameters – holds:
  - Reserved names used by the translation.
  - Reserved smv names.([See also Appendix 1, section2 - Reserved Names.](#))
- TransitionHandler – handles passing of core transitions from child modules to parent modules, as part of the flattening process. ([See also 5.2.1 – Pseudo-code for Flattening.](#))
- TransitionInfo – an information packet, containing a core transition and a module-instance; used by the flattening process.
- VariableRelocator – responsible for relocation of core variables, due to elimination of partial synchronizations. ([See also 5.1.8.2 – Variable Uniqueness and Variable Relocation.](#))
- Textual downloading of the data structures to text:
  - CoreProgramPrinter – downloads the core data structure representation of a core program to a text file.
  - SmvProgramPrinter - downloads the smv data structure representation of a smv program to a text file.

These components are independent from the xml2txt application.

Using command line flags the data structures can be textually downloaded.

The CoreProgramPrinter was helpful during the development and testing stages: It was used to check if the core data structure built by the Xml2CoreDataStructure, and the core program after pre-translation, are equivalent to the core input file.

The SmvProgramPrinter can be used if problems occur in the xml2txt application.

- VariableNamesSorter – used for lexicographical sorting of identifiers contained in a set. The translation often uses hashmaps, improving efficiency; alphabetical ordering of hashmap key sets enhances readability of the output products (module names, variable names, etc.).

## 7. Errors, Exceptions & Warnings

In case of errors in IO, XML parsing, or XML configuration – the translation is aborted. These errors raise Java's built in exceptions, carrying their own error messages. The error message is printed, along with a message stating at which stage of the translation the error occurred (for example – "Can't parse input XML file").

The 'Core2SmvException' class, extending Java's 'Exception' is used for handling errors occurring during the translation and translation-specific errors.

Any unexpected error will raise a 'Core2SmvException' instance, which is handled by aborting the program, and printing the error message along with a message stating at which stage of the translation the error occurred (for example – "Problem in pre-translation"). Unexpected errors include "common" errors, such as "Null Pointer Exception".

Due to several "anticipated" errors, the following exceptions, extending 'Core2SmvException', are used (in all cases, an appropriate error / warning message is printed):

- Errors handled by aborting the translation:
  - ConfigurationException - raised if the configuration file is syntactically invalid.
  - CoreProgramException – raised if the core program does not have a 'SYSTEM' module.
  - ArrayOverflowException – when encountering an array typed variable having more elements than the allowed number per variable.
- Errors handled without aborting the translation:
  - ArrayOverflowException – raised when trying to create a smv array element after the allowed maximum per program has been reached.
  - Transition merging – if trying to merge transitions having contradictory assignments.

## 8. Producing Javadocs

In order to produce the "javadoc" documentation, use the following command line, from the Core2Smv directory:

```
javadoc -d docs/javadocs -sourcepath src src/**/*.java src/**/*.java -private
```

This command will create the documents in the subdirectory "docs/javadocs/". Documentation of private members and methods will be included. Omit the "-private" argument if private members and methods are not desired.

The 'javadoc' command may not be recognized by the shell; if so, add it to the 'path' environment variable, using the following command:

```
set path = ($path /usr/local/java/jsdk-latest/bin)
```

## 9. Delicate Parts in the Implementation

### 9.1. Pre-translation

The PreTranslator component works jointly with two other components – the BookKeeper and the VariableRelocator.

While parsing modcombins of the core program, information is stored for later use. Parsing the tree structure of the core program is done DFS-wise. When encountering a partially-synched modcombin, only the names of the synched transitions are known. The transitions themselves are encountered later, deeper in the sub-tree. Since new modules are created, and variables are relocated, all the information collected during the parsing stage is needed before a full modification of the tree structure can be completed.

The BookKeeper stores and handles relationships between modules and transitions.

Encountered modules are mapped to sets of un-needed transitions.

The VariableRelocator stores information on modules and module instances (under partial synchronization), and later uses this information to relocate variables and pass them as arguments. The VariableRelocator contains an inner class, ModuleInfo, responsible for holding relationships between original and new module-instances, and paths along which relocated variables need to be passed as arguments.

Members and methods of the PreTranslator, BookKeeper, and VariableRelocator are well documented in the source files and the javadoc files.

### 9.2. Cloning Core Objects

The Core\_Object class overrides Java's 'clone( )' method (cloning the XML id and role of the Core\_Object).

All core data structure components extend Core\_Object. Some, but not all, of them further override the 'clone( )' method, implementing deep-cloning. (Objects cloned during the translation, such as Core\_Variable, implement deep-cloning; Others, such as Core\_Modcombin, are not cloned and do not implement deep-cloning.)

In case the translation process is modified, deep-cloning methods should be added where necessary.

A core module has several different methods of cloning, depending on the desired clone:

- cloneAsLeaf( )
- cloneAsNonLeaf( )
- cloneWithoutTransitions( )
- cloneWithoutUnNeededTransitions( )

When cloning a core module use the proper cloning method.

### 9.3. Additional Information

Additional Information is collected during various parts of the translation, by calling the `addConnection( )` methods of the `XmlAddInfoWriter` class. After the translation, the information is organized and downloaded to a XML file.

If parts of the translation are modified, or new ones added, they should be supplemented with corresponding calls to `addConnection( )`.

## 10. Understanding the Implementation

1. Read this document.
2. The data structures (core and smv) are rather simple, and serve as good starting points. Notice the packaging and the inheritance.
3. Afterwards, I suggest top-down reviewing, starting from the 'Core2Smv' class (the main routine running the translation).
4. Reading and writing XML files is done top-down. After instantiation, the process is rather technical.

The XML components import the following Java APIs (which are well documented):

```
javax.xml.parsers
javax.xml.transform
org.w3c.dom
org.xml.sax
```

Except `java.util` and `java.io`, these are the only external APIs used by the translation.

5. The complex parts are the Pre-Translation, `CoreFlattener`, and creating the next-state values of smv variables in the `Translator`

## Appendix 5 - External Tools

### 1. CoreToXml

A program receiving as input a core program in textual form and producing a XML version of this core program. This is a previously existing VeriTech application. It is invoked only when the inputted core program is in textual form.

It is currently located at

```
~veritech/VeriTech/CoreParser/bin/
```

Running the application is done with the command line

```
CoreToXml <input text file>
```

The XML file's name is the input file name, suffixed by '\_xml.xml'.

For example,

```
CoreToXml example.cdl
```

Produces the output file `example.cdl_xml.xml`

### 2. xml2txt

A program receiving as input a XML file and producing a text file with its contents. This is a previously existing VeriTech application. It is invoked to produce a smv text file after the smv XML file has been created.

It is currently located at

```
~veritech/VeriTech/xml/xml2txt/bin/
```

Running the application is done with the command line

```
java -jar xml2txt.jar <xsl file> <input xml file>  
                                <output text file>
```

For example,

```
java -jar xml2txt.jar xml2smv.xsl example.smv.xml example.smv.txt
```

The .xsl file is a style-sheet defining the transformation of the XML's contents.

### 3. XMLViewer

This application, also developed at VeriTech, enables simultaneous viewing of:

- A source program in textual form.
- A source program in xml form.
- A target program in textual form.
- A target program in xml form.
- The additional information file in xml form.

Source and target programs may be in any language, provided they are supported by appropriate dtd and xsl definitions.

Highlighting links between connected objects is a major feature.

The XMLViewer is not used directly by the Core2Smv translation program, but very helpful for comparing source and target programs.

### 4. SMV Model Verifier

The SMV model verifier used is Cadence SMV, version 10\_11.

This application is not used directly by the Core2Smv translation program, but very helpful for examining textual versions of the produced smv programs.

First, syntax validity was checked. Upon loading of a smv file, syntax errors are highlighted.

Afterwards, several specifications were checked, relating to the expected behavior of state to state transitions described by the smv program.

In order to verify or refute an assertion, a SPEC declaration is inserted to the smv program ([See also "The SMV System", section 2.8](#))

The model verifier states whether the assertion is true or false.

If it is false and a computation path refuting the assertion exists – it is shown (in form of a table with variables' values in successive states).

The model verifier does not present a valid path for a true assertion.

Note that some assertions are false, but can not be supported by a path refuting them.

For example if the assertion "EF (x=5)" is false (meaning no path exists in which eventually the value of x is 5), then of course a refuting path can not be supplied.

A few notes on using Cadence SMV:

- To open a file, click File->New. In the Source/Trace/Log toolbar, select "Source".
- To verify (or refute) a specification, click Prop->Verify (or Prop->Verify all).  
In the Browser/Properties/Results/... toolbar, select "Results". The specification with its result ("true " or "false") is shown.  
In the Source/Trace/Log toolbar, select "Trace". A table with variable values in successive states is shown.

## Appendix 6 - Future Prospects

### 1. Partial Synchronizations

1. Phrasing a complete algorithm for elimination of partial synchronizations in the core program, without use of flattening. Maybe a different approach would be more comprehensive, and also easier to implement.
2. A few improvements to the current algorithm were suggested, but not implemented:
  - When two identical brother modules receive the same set of arguments, it is possible to reduce the number of newly created modules ([see also 5.1.7 - Second Improvement](#)).
  - Relocation of variables – currently, all variables appearing in modules under partial synchronization are relocated. Not all of them need to be relocated ([see also 5.1.8.2 - Variable Uniqueness and Variable Relocation](#)).
  - Some cases of S type MC under P type MC can be identified and handled without flattening ([see also 5.1.9 - S Type MC Under P Type](#)).
3. In some of the examples, it is obvious that a choice of flattening (as opposed to the elimination algorithm) results in a shorter and more compact smv program. It is recommended to add a routine for checking, during runtime, which of the two options is better ([see also Appendix 7 – Examples, 3.3 Buffer Example](#)).

### 2. Additional Information

1. Additional information items can be organized better. According to the additional information dtd, a "Change" element is defined by

```
<!ELEMENT Change (Source+, Target+, Connection+)>
```

Currently all "Change" elements written to the additional information file consist of one "Source" element, one "Target" element, and one "Connection" element. However, some information items represent a one-to-many (or many-to-many) relationship.

For example, each core global define is translated and inserted to all smv modules ([see also 3.3 - Handling Defines and Constants](#)). This is a one-to-many relationship, and one "Change" element (containing one source and several targets) is more appropriate than several "Change" elements (each of them containing one source and one target).

2. An improvement to the XMLViewer application is suggested ([see also Appendix 5 – External Tools, section 3 – XMLViewer](#)): Currently it does not distinguish between "additional" and "regular" information. It is recommended to add a feature enabling the user to choose which type of information should be displayed – "additional" only, or "extended" ("additional" + "regular"). This improvement requires:

- Incorporating the feature in the XMLViewer implementation.
- Modifying the additional information dtd: Currently, the "Change" element has the following attribute:

```
xlink:type (extended) #FIXED "extended"
```

This attribute can be modified to:

```
xlink:type (extended | regular) #IMPLIED
```

- When composing a "Change" element, the proper attribute field should be set. "Change" elements are composed in the 'addChangeToDocument()' method of the XmlAddInfoWriter component. Currently, the following code lines set this attribute:

```
Attr typeAttr = doc.createAttribute("xlink:type");
typeAttr.setNodeValue("extended");
```

The 'addChangeToDocument()' method should be supplemented with a boolean parameter determining whether the attribute's value would be "extended" or "regular".

### 3. Problems in the CoreToXml Application

1. Successive typedef declarations are not handled. For example, the following set of declarations will result in an error:
 

```
TYPE
    color: ENUM {red, green, blue};
    myFavoriteColor: color;
```
2. Undeclared identifiers, in some cases, are un-warned against. For example, in one of the specially built test-cases, the identifier 'k' was used without declaration, and passed un-noticed ([see simple example no.3](#)).
3. Multi-dimensional arrays are not handled properly. For example, in one of the specially built test-cases, a three dimensional array named '\_Fives' was created. Expressions such as '\_Fives[1][0][exp]' were transformed to '(null)[exp]' ([see simple example no. 3](#)).

### 4. Problems in the xml2txt Application

This program doesn't properly handle special notations used in xml / html:

- '&amp;' actually means '&'
- '&lt;.' actually means '<'
- '&gt;.' actually means '>'

For example, the textual string expression

```
a < b
```

is written in XML as

```
a &lt; b
```

and when transforming XML to text, these three symbols should be taken into account.

## 5. Global Variables and Synchronization

The mutual relationship between global variables and the synchronization structure deserves further thought ([see Appendix 3 - Does Treatment of Global Variables Affect the Synchronization Structure?](#)). This issue was tested by only one case-example.

## 6. Simplifying smv Expressions

Several smv 'case' expressions should be further simplified:

- The following expression ([see 'fail's assignment in Module A, in Appendix 7 – Examples, Section 3.1 - Simple Example 2](#)).

```
next(fail) := case
  terminate : 0;
  1 : 0;
esac;
```

can be simplified to

```
next(fail) := 0;
```

- The following expression ([see 'terminate's assignment in Module BUFFER\\_get, in Appendix 7 – Examples, Section 3.3 - Buffer Example](#)).

```
terminate := case
  !cok : 1;
  1 : 0;
esac;
```

can be simplified to

```
terminate := !cok;
```

## 7. Resolving Naming Conflicts

Name ambiguity among identifiers is resolved by suffixing identifiers with a unique integer.

For example, if a core module M has three son modules, all of which are instances of the same module A, the core representation might be:

```
MODULE M
  A () || ( A () || A () )
```

While the smv representation would be:

```
MODULE M
  A_instance: A;
  A_instance1: A;
  A_instance2: A;
```

A better representation would be:

```
MODULE M
  A_instance0: A; // this line is modified
  A_instance1: A;
  A_instance2: A;
```

Currently, the implementation uses the suffixing method only when an encountered identifier conflicts with an existing identifier (within the same scope). In the above example, the first encounter with the son module A produces the smv identifier 'A\_instance', while the following encounters produce the identifiers 'A\_instance1' and 'A\_instance2'.

In order to implement the improvement, the following steps are suggested:

1. When applying the suffixing method, keep a temporary list of previously encountered identifiers. (In the above example, upon the second encounter of A, 'A\_instance1' is created and 'A\_instance' is temporarily stored.)
2. After all identifiers within the same scope are handled, iterate through the list, and suffix each element with '0'. ('A\_instance' is modified to 'A\_instance0'.)
3. Replace all appearances of the replaced identifier ('A\_instance') by the replacing identifier ('A\_instance0').

Replacement of identifier appearances can be executed by using the 'replaceInAllExpressions()' method. This method is a member of some core data structure components (core variable, core module, core program). If the replacement occurs in smv objects – similar methods need to be implemented.

## 8. Release & Debug Versions

It is customary practice in software development to produce two executable versions – 'release' and 'debug'. Currently, the Core2Smv application has only one version, serving both purposes. More specifically, two components (the CoreProgramPrinter and SmvProgramPrinter, [see also 'Textual downloading' bullet, in Appendix 4 – Programming Notes, section 6.5 – utilities package](#)) are debug-oriented, and should be excluded from a 'release' version.

Currently, printing of the data structures is enabled by a combination of command line flags and an entry in the configuration file ([see also 'User Guide.doc', section 3 – Running the Core2Smv Translation Program and section 6 – Configuration File & Default Values](#)).

It is advised to create a separate 'release' and 'debug' versions:

- The 'Makefile' should be modified – either producing two separate executables, or only one of them (using #ifdef).
- The CoreProgramPrinter and SmvProgramPrinter classes should be excluded from the 'release' version.
- Command line flags enabling printing of data structures should be excluded from the 'release' version.
- The 'PRINT\_DATA\_STRUCTURES' entry in the configuration file can be removed.
- Minor modifications are needed in the 'Core2Smv' and 'Flat' classes (these are the 'main' classes – checking command line arguments and configuration entries):
  - In the 'debug' version, the option of printing the data structures should depend only on command line arguments.
  - In the 'release' version, this option should be removed.

## Appendix 7 - Examples

The subdirectory 'Core2Smv/examples' contains core / smv programs produced by the core to smv translation.

### 1. Currently Existing Examples

The examples are of two kinds:

1. Several core / smv programs addressing common issues in computer science (buffer, card game, dining philosophers, synch, pc example). Most of these examples have translations from / to other languages also. Each example is located at its own subdirectory:

- Core2Smv/examples/abp
- Core2Smv/examples/bin\_counter
- Core2Smv/examples/bounded\_buffer
- Core2Smv/examples/buffer
- Core2Smv/examples/card\_game
- Core2Smv/examples/combination1
- Core2Smv/examples/combination2
- Core2Smv/examples/combination21
- Core2Smv/examples/combination3
- Core2Smv/examples/combination4
- Core2Smv/examples/combination5
- Core2Smv/examples/dining\_philosophers
- Core2Smv/examples/election
- Core2Smv/examples/game1
- Core2Smv/examples/game2
- Core2Smv/examples/go\_back\_n
- Core2Smv/examples/muex1
- Core2Smv/examples/nondeter\_parallel\_buffers
- Core2Smv/examples/nondeterministic\_assignment
- Core2Smv/examples/nondeterministic\_initialization
- Core2Smv/examples/open\_safe
- Core2Smv/examples/parameters
- Core2Smv/examples/pc\_example
- Core2Smv/examples/program\_counters
- Core2Smv/examples/semaphore
- Core2Smv/examples/simple\_buffer
- Core2Smv/examples/sync\_core
- Core2Smv/examples/tst
- Core2Smv/examples/tst2

2. Simple examples used for testing the translation program .They also serve as an easy preliminary view of core / smv programs. These examples are located at the subdirectories:

```
Core2Smv/examples/checks/terminate_det
Core2Smv/examples/checks/three_state_non_det
Core2Smv/examples/checks/two_state
Core2Smv/examples/checks/three_valued_variable
Core2Smv/examples/checks/refutation1
Core2Smv/examples/checks/refutation2
Core2Smv/examples/simple/1
Core2Smv/examples/simple/2
Core2Smv/examples/simple/3
Core2Smv/examples/simple/4
Core2Smv/examples/simple/5
Core2Smv/examples/simple/6
```

Each of these subdirectories includes a WORD document describing which aspects of the translation are demonstrated (in the 'simple' examples) or which specifications were checked using the SMV model verifier (in the 'checks' examples).

## 2. File Naming Conventions

The following naming conventions are used:

- example.cdl - textual representation of the input (the original core program in its textual format).
- example.cdl\_xml.xml – xml file of the core xml version (This name is supplied by the CoreToXml program).
- example.cdl.ds.txt – textual download of the core data structure.
- example.smv.ds.txt – textual download of the smv data structure.
- example.smv.xml – xml file of the smv program.
- example.smv.txt – text file of the smv program.
- example.flat.txt – text file of the flattened core program. (This name is supplied only when executing Flat.jar. If Core2Smv.jar is executed, and the core program is flattened, other above-mentioned names are used.)

### 3. Examples of Core / Smv Programs

#### 3.1. Simple Example no. 2

This example demonstrates:

- Integer typed variables translated to range type.
- Added edge conditions to avoid out-of-bounds assignments.
- Added "hold previous" condition.
- The transition name 'a' is changed to 'a1', since it is also a variable name.
- Resolving module name ambiguity (there are two instances of module A).
- The module name 'A' is changed to '\_A', since 'A' is a reserved name.

#### **Core Version**

```
HOLD_PREVIOUS

MODULE A (a: integer)
{
  VAR
    num1: INTEGER INITVAL 0;
    num2: INTEGER INITVAL 0;
  TRANS a:
    enable: true;
    assign:
      num1' := num1 + 1;
      a' := a + 1;
    relation: true;
  TRANS b:
    enable: true;
    assign:
      num2' := num2 + 1;
      a' := a - 1;
    relation: true;
}

MODULE SYSTEM ()
{
  VAR
    num1 : INTEGER INITVAL 0;
    num2 : INTEGER INITVAL 0;

  (A(num1) || A(num2))
}
```

## Smv Version

```
MODULE _A(a)
VAR
  fail : boolean;
  num1 : -100..100;
  num2 : -100..100;
  terminate : boolean;
  trans : {a1,b};
ASSIGN
  next(fail) := case
    terminate : 0;
    1 : 0;
  esac;
  init(num1) := 0;
  next(num1) := case
    terminate : num1;
    (next(trans) = a1) & (-100 <= num1 + 1) & (num1 + 1 <= 100) :
      num1 + 1;
    1 : num1;
  esac;
  init(num2) := 0;
  next(num2) := case
    terminate : num2;
    (next(trans) = b) & (-100 <= num2 + 1) & (num2 + 1 <= 100) :
      num2 + 1;
    1 : num2;
  esac;
  terminate := 0;
  next(a) := case
    terminate : a;
    (next(trans) = a1) & (-100 <= a + 1) & (a + 1 <= 100) : a + 1;
    (next(trans) = b) & (-100 <= a - 1) & (a - 1 <= 100) : a - 1;
    1 : a;
  esac;
TRANS
  (next(trans) = a1) | (next(trans) = b) | terminate
INVAR
  !fail

MODULE main
VAR
  num1 : -100..100;
  num2 : -100..100;
  terminate : boolean;
  _A_instance : _A(num1);
  _A_instancel : _A(num2);
ASSIGN
  init(num1) := 0;
  init(num2) := 0;
  terminate := _A_instancel.terminate | _A_instance.terminate;
```

## Flattened Core Version

The flattened version demonstrates:

- Resolving variable name ambiguity (in the core program there are three different variables named "num1" and three different variables named "num2").
- Prefixing the modules' components by the module name. (Notice that in the core program there are two instances of module A.)
- Merging of transitions.

```
HOLD_PREVIOUS
```

```
MODULE SYSTEM ()
{
  A1_num1 : INTEGER INITVAL 0;
  A1_num2 : INTEGER INITVAL 0;
  A_num1  : INTEGER INITVAL 0;
  A_num2  : INTEGER INITVAL 0;
  num1    : INTEGER INITVAL 0;
  num2    : INTEGER INITVAL 0;
  TRANS A_a$A1_a:
    enable: TRUE /\ TRUE;
    assign:
      A1_num2' := A1_num2 + 1;
      A_num1'  := A_num1 + 1;
      A1_num1' := A1_num1 + 1;
    relation: TRUE /\ TRUE;
  TRANS A_a$A1_b:
    enable: TRUE /\ TRUE;
    assign:
      A1_num2' := A1_num2 - 1;
      A_num1'  := A_num1 + 1;
    relation: TRUE /\ TRUE;
  TRANS A_b$A1_a:
    enable: TRUE /\ TRUE;
    assign:
      A1_num2' := A1_num2 + 1;
      A_num1'  := A_num1 - 1;
      A_num2'  := A_num2 + 1;
      A1_num1' := A1_num1 + 1;
    relation: TRUE /\ TRUE;
  TRANS A_b$A1_b:
    enable: TRUE /\ TRUE;
    assign:
      A1_num2' := A1_num2 - 1;
      A_num1'  := A_num1 - 1;
      A_num2'  := A_num2 + 1;
    relation: TRUE /\ TRUE;
}
```

## 3.2. Dining Philosophers Example

This example demonstrates:

- Translation of a modcombin, having several child module-instances, all interleaved.
- Usage of typedefs.
- Global variables.
- Translation of global constants.
- Expansion of array elements.

### Core Version

```
HOLD_PREVIOUS

CONST
  NumOfPhilosophers: 5;
  NumOfForks: 5;
TYPE
  PhilosopherState: ENUM {thinking , eating};
  ForkState: ENUM {available , nonavailable};
VAR
  philosophers: ARRAY[NumOfPhilosophers] OF PhilosopherState
    INITVAL thinking;
  forks: ARRAY[NumOfForks] OF ForkState INITVAL available;

MODULE Philosopher(i: INTEGER){

  TRANS PickupForks:
    enable: ((philosophers[i] = thinking) /\ (forks[i] = available) /\
      (forks[(i+1) % NumOfForks]= available));
    assign: philosophers[i]' := eating;
      forks[i]' := nonavailable;
      forks[(i+1) % NumOfForks]' := nonavailable;
  TRANS ReturnForks:
    enable: philosophers[i] = eating;
    assign: philosophers[i]' := thinking;
      forks[i]' := available;
      forks[(i+1) % NumOfForks]' := available;
}

MODULE SYSTEM(){
  VAR

  p0: integer INITVAL 0;
  p1: integer INITVAL 1;
  p2: integer INITVAL 2;
  p3: integer INITVAL 3;
  p4: integer INITVAL 4;

  (((Philosopher(p0) ||| Philosopher(p1)) ||| Philosopher(p2)) |||
    Philosopher(p3)) ||| Philosopher(p4))
}

```

## Smv Version

```
MODULE GLOBALS
VAR
  forks : array 0..4 of {available,nonavailable};
  philosophers : array 0..4 of {thinking,eating};
DEFINE
ASSIGN
  init(forks[0]) := available;
  init(forks[1]) := available;
  init(forks[2]) := available;
  init(forks[3]) := available;
  init(forks[4]) := available;
  init(philosophers[0]) := thinking;
  init(philosophers[1]) := thinking;
  init(philosophers[2]) := thinking;
  init(philosophers[3]) := thinking;
  init(philosophers[4]) := thinking;

MODULE main
VAR
  p0 : -100..100;
  p1 : -100..100;
  p2 : -100..100;
  p3 : -100..100;
  p4 : -100..100;
  terminate : boolean;
  Philosopher_instance : process Philosopher(p0,globals);
  Philosopher_instance1 : process Philosopher(p1,globals);
  Philosopher_instance2 : process Philosopher(p2,globals);
  Philosopher_instance3 : process Philosopher(p3,globals);
  Philosopher_instance4 : process Philosopher(p4,globals);
  globals : process GLOBALS;
DEFINE
  GC_NumOfForks := 5;
  GC_NumOfPhilosophers := 5;
ASSIGN
  init(p0) := 0;
  init(p1) := 1;
  init(p2) := 2;
  init(p3) := 3;
  init(p4) := 4;
  terminate := Philosopher_instance4.terminate &
    Philosopher_instance1.terminate & Philosopher_instance.terminate &
    Philosopher_instance3.terminate & Philosopher_instance2.terminate;
```

```

MODULE Philosopher(i, globals)
VAR
  fail : boolean;
  terminate : boolean;
  trans : {PickUpForks, ReturnForks};
DEFINE
  GC_NumOfForks := 5;
  GC_NumOfPhilosophers := 5;
ASSIGN
  next(fail) := case
    terminate : 0;
    ((next(trans) = PickUpForks) & !((globals.philosophers[i] =
      thinking) & (globals.forks[i] = available) & (globals.forks[(i+1)
      mod GC_NumOfForks] = available))) | ((next(trans) = ReturnForks) &
      !(globals.philosophers[i] = eating)) : 1;
    1 : 0;
  esac;
  terminate := case
    !((globals.philosophers[i] = thinking) & (globals.forks[i] =
      available) & (globals.forks[(i + 1) mod GC_NumOfForks] =
      available)) | (globals.philosophers[i] = eating)) : 1;
    1 : 0;
  esac;
  next(i) := i;

```

```

next(globals.forks[0]) := case
  terminate : globals.forks[0];
  (next(trans) = PickUpForks) & (i = 0) : nonavailable;
  (next(trans) = PickUpForks) & ((i + 1) mod GC_NumOfForks = 0) :
    nonavailable;
  (next(trans) = ReturnForks) & (i = 0) : available;
  (next(trans) = ReturnForks) & ((i + 1) mod GC_NumOfForks = 0) :
    available;
  1 : globals.forks[0];
esac;
next(globals.forks[1]) := case
  terminate : globals.forks[1];
  (next(trans) = PickUpForks) & (i = 1) : nonavailable;
  (next(trans) = PickUpForks) & ((i + 1) mod GC_NumOfForks = 1) :
    nonavailable;
  (next(trans) = ReturnForks) & (i = 1) : available;
  (next(trans) = ReturnForks) & ((i + 1) mod GC_NumOfForks = 1) :
    available;
  1 : globals.forks[1];
esac;
next(globals.forks[2]) := case
  terminate : globals.forks[2];
  (next(trans) = PickUpForks) & (i = 2) : nonavailable;
  (next(trans) = PickUpForks) & ((i + 1) mod GC_NumOfForks = 2) :
    nonavailable;
  (next(trans) = ReturnForks) & (i = 2) : available;
  (next(trans) = ReturnForks) & ((i + 1) mod GC_NumOfForks = 2) :
    available;
  1 : globals.forks[2];
esac;
next(globals.forks[3]) := case
  terminate : globals.forks[3];
  (next(trans) = PickUpForks) & (i = 3) : nonavailable;
  (next(trans) = PickUpForks) & ((i + 1) mod GC_NumOfForks = 3) :
    nonavailable;
  (next(trans) = ReturnForks) & (i = 3) : available;
  (next(trans) = ReturnForks) & ((i + 1) mod GC_NumOfForks = 3) :
    available;
  1 : globals.forks[3];
esac;
next(globals.forks[4]) := case
  terminate : globals.forks[4];
  (next(trans) = PickUpForks) & (i = 4) : nonavailable;
  (next(trans) = PickUpForks) & ((i + 1) mod GC_NumOfForks = 4) :
    nonavailable;
  (next(trans) = ReturnForks) & (i = 4) : available;
  (next(trans) = ReturnForks) & ((i + 1) mod GC_NumOfForks = 4) :
    available;
  1 : globals.forks[4];
esac;

```

```

next(globals.philosophers[0]) := case
    terminate : globals.philosophers[0];
    (next(trans) = PickUpForks) & (i = 0) : eating;
    (next(trans) = ReturnForks) & (i = 0) : thinking;
    l : globals.philosophers[0];
esac;
next(globals.philosophers[1]) := case
    terminate : globals.philosophers[1];
    (next(trans) = PickUpForks) & (i = 1) : eating;
    (next(trans) = ReturnForks) & (i = 1) : thinking;
    l : globals.philosophers[1];
esac;
next(globals.philosophers[2]) := case
    terminate : globals.philosophers[2];
    (next(trans) = PickUpForks) & (i = 2) : eating;
    (next(trans) = ReturnForks) & (i = 2) : thinking;
    l : globals.philosophers[2];
esac;
next(globals.philosophers[3]) := case
    terminate : globals.philosophers[3];
    (next(trans) = PickUpForks) & (i = 3) : eating;
    (next(trans) = ReturnForks) & (i = 3) : thinking;
    l : globals.philosophers[3];
esac;
next(globals.philosophers[4]) := case
    terminate : globals.philosophers[4];
    (next(trans) = PickUpForks) & (i = 4) : eating;
    (next(trans) = ReturnForks) & (i = 4) : thinking;
    l : globals.philosophers[4];
esac;
TRANS
    (next(trans) = PickUpForks) | (next(trans) = ReturnForks) | terminate
INVAR
    !fail

```

### 3.3. Buffer Example

This example demonstrates elimination of two levels of partial synchronization. Flattened and non-flattened smv versions are presented. The effect of the elimination algorithm, in this case, is that each transition is represented by a distinct smv module. A disadvantage in flattening (in all cases) is that it results in long identifiers.

## Core Version

```
HOLD_PREVIOUS

TYPE
elem: 0..4;

MODULE SENDER(a: elem){
  VAR
  readys: boolean INITVAL false;

  TRANS produce:
    enable: !readys;
    assign: a' := {0,1,2,3,4};
    relation: readys';

  TRANS send:
    enable: readys;
    relation: !readys';
}

MODULE BUFFER(c: elem; d:elem){
  VAR
  cok: boolean INITVAL true;
  dok: boolean INITVAL false;

  TRANS get:
    enable: cok;
    relation: !cok';

  TRANS move:
    enable: !cok /\ !dok;
    assign: d' := c;
           cok' := true;
           dok' := true;

  TRANS put:
    enable: dok;
    relation: !dok';
}

MODULE RECEIVER(b: elem){
  VAR
  vr: elem;
  readyr: boolean INITVAL true;

  TRANS consume:
    enable: !readyr;
    assign: vr' := b;
           readyr' := true;

  TRANS receive:
    enable: readyr;
    relation: !readyr';
}

MODULE SYSTEM(){
  VAR
  s : elem;
  t : elem;

  (SENDER(s) |(send, get)| (BUFFER(s,t) |(put, receive)| RECEIVER(t)))
}
```

## Smv Version

```
MODULE BUFFER_get(c,d,cok,dok)
VAR
  fail : boolean;
  terminate : boolean;
  trans : {get};
DEFINE
ASSIGN
  next(fail) := case
    terminate : 0;
    ((next(trans) = get) & !cok) : 1;
    1 : 0;
  esac;
  terminate := case
    !cok : 1;
    1 : 0;
  esac;
  next(c) := c;
  next(d) := d;
  next(cok) := case
    next(trans) = get : {0,1};
    1 : cok;
  esac;
  next(dok) := dok;
TRANS
  ((next(trans) = get) & !next(cok)) | terminate
INVAR
  !fail

MODULE BUFFER_move(c,d,cok,dok)
VAR
  fail : boolean;
  terminate : boolean;
  trans : {move};
DEFINE
ASSIGN
  next(fail) := case
    terminate : 0;
    ((next(trans) = move) & !(!cok & !dok)) : 1;
    1 : 0;
  esac;
  terminate := case
    !(!cok & !dok) : 1;
    1 : 0;
  esac;
  next(c) := c;
  next(d) := case
    terminate : d;
    (next(trans) = move) & (0 <= c) & (c <= 4) : c;
    1 : d;
  esac;
  next(cok) := case
    terminate : cok;
    next(trans) = move : 1;
    1 : cok;
  esac;
  next(dok) := case
    terminate : dok;
    next(trans) = move : 1;
    1 : dok;
  esac;
TRANS
  (next(trans) = move) | terminate
INVAR
  !fail
```

```

MODULE BUFFER_put (c, d, cok, dok)
VAR
    fail : boolean;
    terminate : boolean;
    trans : {put};
DEFINE
ASSIGN
    next(fail) := case
        terminate : 0;
        ((next(trans) = put) & !dok) : 1;
        1 : 0;
    esac;
    terminate := case
        !dok : 1;
        1 : 0;
    esac;
    next(c) := c;
    next(d) := d;
    next(cok) := cok;
    next(dok) := case
        next(trans) = put : {0,1};
        1 : dok;
    esac;
TRANS
    ((next(trans) = put) & !next(dok)) | terminate
INVAR
    !fail

MODULE BUFFER_put$RECEIVER_receive (s, t, cok, dok, readyr, vr)
VAR
    terminate : boolean;
    BUFFER_put_instance : BUFFER_put (s, t, cok, dok);
    RECEIVER_receive_instance : RECEIVER_receive (t, readyr, vr);
DEFINE
ASSIGN
    terminate := BUFFER_put_instance.terminate |
    RECEIVER_receive_instance.terminate;

```

```

MODULE RECEIVER_consume(b, readyr, vr)
VAR
    fail : boolean;
    terminate : boolean;
    trans : {consume};
DEFINE
ASSIGN
    next(fail) := case
        terminate : 0;
        ((next(trans) = consume) & readyr) : 1;
        1 : 0;
    esac;
    terminate := case
        readyr : 1;
        1 : 0;
    esac;
    next(b) := b;
    next(readyr) := case
        terminate : readyr;
        next(trans) = consume : 1;
        1 : readyr;
    esac;
    next(vr) := case
        terminate : vr;
        (next(trans) = consume) & (0 <= b) & (b <= 4) : b;
        1 : vr;
    esac;
TRANS
    (next(trans) = consume) | terminate
INVAR
    !fail

MODULE RECEIVER_receive(b, readyr, vr)
VAR
    fail : boolean;
    terminate : boolean;
    trans : {receive};
DEFINE
ASSIGN
    next(fail) := case
        terminate : 0;
        ((next(trans) = receive) & !readyr) : 1;
        1 : 0;
    esac;
    terminate := case
        !readyr : 1;
        1 : 0;
    esac;
    next(b) := b;
    next(readyr) := case
        next(trans) = receive : {0,1};
        1 : readyr;
    esac;
    next(vr) := vr;
TRANS
    ((next(trans) = receive) & !next(readyr)) | terminate
INVAR
    !fail

```

```

MODULE SENDER_produce(a,readys)
VAR
    fail : boolean;
    terminate : boolean;
    trans : {produce};
DEFINE
ASSIGN
    next(fail) := case
        terminate : 0;
        ((next(trans) = produce) & readys) : 1;
        1 : 0;
    esac;
    terminate := case
        readys : 1;
        1 : 0;
    esac;
    next(a) := case
        terminate : a;
        next(trans) = produce : {0,1,2,3,4};
        1 : a;
    esac;
    next(readys) := case
        next(trans) = produce : {0,1};
        1 : readys;
    esac;
TRANS
    ((next(trans) = produce) & (next(readys))) | terminate
INVAR
    !fail

```

```

MODULE SENDER_send(a,readys)
VAR
    fail : boolean;
    terminate : boolean;
    trans : {send};
DEFINE
ASSIGN
    next(fail) := case
        terminate : 0;
        ((next(trans) = send) & !readys) : 1;
        1 : 0;
    esac;
    terminate := case
        !readys : 1;
        1 : 0;
    esac;
    next(a) := a;
    next(readys) := case
        next(trans) = send : {0,1};
        1 : readys;
    esac;
TRANS
    ((next(trans) = send) & !next(readys)) | terminate
INVAR
    !fail

```

```

MODULE SENDER_send$BUFFER_get (s,t,cok,dok,readys)
VAR
    terminate : boolean;
    BUFFER_get_instance : BUFFER_get (s,t,cok,dok);
    SENDER_send_instance : SENDER_send (s,readys);
DEFINE
ASSIGN
    terminate := BUFFER_get_instance.terminate |
SENDER_send_instance.terminate;

MODULE main
VAR
    cok : boolean;
    dok : boolean;
    readyr : boolean;
    readys : boolean;
    s : 0..4;
    t : 0..4;
    terminate : boolean;
    vr : 0..4;
    BUFFER_move_instance : process BUFFER_move (s,t,cok,dok);
    BUFFER_put$RECEIVER_receive_instance : process
BUFFER_put$RECEIVER_receive (s,t,cok,dok,readyr,vr);
    RECEIVER_consume_instance : process RECEIVER_consume (t,readyr,vr);
    SENDER_produce_instance : process SENDER_produce (s,readys);
    SENDER_send$BUFFER_get_instance : process
SENDER_send$BUFFER_get (s,t,cok,dok,readys);
DEFINE
ASSIGN
    init(cok) := 1;
    init(dok) := 0;
    init(readyr) := 1;
    init(readys) := 0;
    terminate := RECEIVER_consume_instance.terminate &
BUFFER_put$RECEIVER_receive_instance.terminate &
SENDER_produce_instance.terminate &
SENDER_send$BUFFER_get_instance.terminate & BUFFER_move_instance.terminate;

```

## Flattened Smv Version

```
MODULE main
VAR
  BUFFER$RECEIVER_BUFFER_cok : boolean;
  BUFFER$RECEIVER_BUFFER_dok : boolean;
  BUFFER$RECEIVER_RECEIVER_readyr : boolean;
  BUFFER$RECEIVER_RECEIVER_vr : 0..4;
  SENDER_readys : boolean;
  fail : boolean;
  s : 0..4;
  t : 0..4;
  terminate : boolean;
  trans :
    {BUFFER$RECEIVER_RECEIVER_consume, SENDER_produce, SENDER_send$BUFFER$RECEIVER_BUFFER_get, BUFFER$RECEIVER_BUFFER_put$RECEIVER_receive, BUFFER$RECEIVER_BUFFER_move};
DEFINE
ASSIGN
  init(BUFFER$RECEIVER_BUFFER_cok) := 1;
  next(BUFFER$RECEIVER_BUFFER_cok) := case
    terminate : BUFFER$RECEIVER_BUFFER_cok;
    next(trans) = BUFFER$RECEIVER_BUFFER_move : 1;
    next(trans) = SENDER_send$BUFFER$RECEIVER_BUFFER_get : {0,1};
    1 : BUFFER$RECEIVER_BUFFER_cok;
  esac;
  init(BUFFER$RECEIVER_BUFFER_dok) := 0;
  next(BUFFER$RECEIVER_BUFFER_dok) := case
    terminate : BUFFER$RECEIVER_BUFFER_dok;
    next(trans) = BUFFER$RECEIVER_BUFFER_move : 1;
    next(trans) = BUFFER$RECEIVER_BUFFER_put$RECEIVER_receive : {0,1};
    1 : BUFFER$RECEIVER_BUFFER_dok;
  esac;
  init(BUFFER$RECEIVER_RECEIVER_readyr) := 1;
  next(BUFFER$RECEIVER_RECEIVER_readyr) := case
    terminate : BUFFER$RECEIVER_RECEIVER_readyr;
    next(trans) = BUFFER$RECEIVER_RECEIVER_consume : 1;
    next(trans) = BUFFER$RECEIVER_BUFFER_put$RECEIVER_receive : {0,1};
    1 : BUFFER$RECEIVER_RECEIVER_readyr;
  esac;
  next(BUFFER$RECEIVER_RECEIVER_vr) := case
    terminate : BUFFER$RECEIVER_RECEIVER_vr;
    (next(trans) = BUFFER$RECEIVER_RECEIVER_consume) & (0 <= t) & (t <= 4) : t;
    1 : BUFFER$RECEIVER_RECEIVER_vr;
  esac;
  init(SENDER_readys) := 0;
  next(SENDER_readys) := case
    next(trans) = SENDER_produce | next(trans) = SENDER_send$BUFFER$RECEIVER_BUFFER_get : {0,1};
    1 : SENDER_readys;
  esac;
  next(fail) := case
    terminate : 0;
    ((next(trans) = BUFFER$RECEIVER_RECEIVER_consume) & BUFFER$RECEIVER_RECEIVER_readyr) | ((next(trans) = SENDER_produce) & SENDER_readys) | ((next(trans) = SENDER_send$BUFFER$RECEIVER_BUFFER_get) & !(SENDER_readys & BUFFER$RECEIVER_BUFFER_cok)) | ((next(trans) = BUFFER$RECEIVER_BUFFER_put$RECEIVER_receive) & !(BUFFER$RECEIVER_BUFFER_dok & BUFFER$RECEIVER_RECEIVER_readyr)) | ((next(trans) = BUFFER$RECEIVER_BUFFER_move) & !(BUFFER$RECEIVER_BUFFER_cok & !BUFFER$RECEIVER_BUFFER_dok)) : 1;
    1 : 0;
  esac;
```

```

next(s) := case
  terminate : s;
  next(trans) = SENDER_produce : {0,1,2,3,4};
  1 : s;
esac;
next(t) := case
  terminate : t;
  (next(trans) = BUFFER$RECEIVER_BUFFER_move) & (0 <= s) & (s <= 4):s;
  1 : t;
esac;
terminate := case
  (!(BUFFER$RECEIVER_RECEIVER_readyr | !SENDER_readys | (SENDER_readys
& BUFFER$RECEIVER_BUFFER_cok) | (BUFFER$RECEIVER_BUFFER_dok &
BUFFER$RECEIVER_RECEIVER_readyr) | (!(BUFFER$RECEIVER_BUFFER_cok &
!BUFFER$RECEIVER_BUFFER_dok)) : 1;
  1 : 0;
esac;
TRANS
(next(trans) = BUFFER$RECEIVER_RECEIVER_consume) | ((next(trans) =
SENDER_produce) & (next(SENDER_readys))) | ((next(trans) =
SENDER_send$BUFFER$RECEIVER_BUFFER_get) & (!next(SENDER_readys) &
!next(BUFFER$RECEIVER_BUFFER_cok))) | ((next(trans) =
BUFFER$RECEIVER_BUFFER_put$RECEIVER_receive) &
(!next(BUFFER$RECEIVER_BUFFER_dok) &
!next(BUFFER$RECEIVER_RECEIVER_readyr))) | (next(trans) =
BUFFER$RECEIVER_BUFFER_move) | terminate
INVAR
!fail

```

## Acknowledgments

Several discussions were held with Prof. Shmuel Katz and Prof. Orna Grumberg, from the CS faculty at the Technion. Their insight and suggestions aided my reasoning in the complex and tricky parts.

Mr. Shahar Dag (lab engineer at SSDL – Systems and Software Development Laboratory) was deeply involved in all aspects of this project. Dozens of meetings helped my progress. His advice, patience, and encouragement made this work possible.