

OVERALL	2
DATA STRUCTURES	3
SPIN2CORE	3
SPIN	3
ALGORITHMS	5
OVERALL	5
PREPROCESSING	5
PARSING AND DATA STRUCTURES BUILDING	5
PREPARATIONS AND DATA STRUCTURE ADJUSTMENTS	5
TRANSLATION	5
TBD LIST	6
SYNCHRONIZATION OF A CHANNEL	6
RENDEZVOUS	6
ATOMIC AND D_STEP	6
<i>atomic</i>	6
<i>d_step</i>	7
LOG FILE	7
OPTIONAL ARGUMENTS	8

Overall

The spin2core is based on the files given from bell lab. The interface between the spin files and the spin2core file are reduced to three:

- a) S2C_DataStruct.h
- b) S2C_Translator.h
- c) S2C_Preprocess.h

All other files are for the use of spin2core and they are made for convenience.

The idea of the translation is to parse the spin file with the help of the spin parser, and using the spin data structure made during the parsing with as minimum changes as possible (described below).

Once the parsing is finished, the translation begins. During the translation some information is gathered (such as combined modules that have to be translated).

Changes to the Spin parser

Some of the files of the Bell Labs Spin parser where changed.

All changes begin with the line:

```
“/***** Eli - Spin2Core *****/”
```

and ends with the line:

```
“/*****/”
```

(look for “Eli - Spin2Core” to locate all the changes) .

The changed files are:

1. main.c
 - A. Optional arguments were removed.
 - B. S2C_preprocess was added in order to preprocess the input file, before the default preprocessor is activated.
 - C. S2C_closeFile was added.
2. spin.y (yacc file) (derives changes in y.tab.c)
 - A. S2C_initFile was added in order to open a new file for writing.
 - B. S2C_translateProgram was added.
 - C. Some more changes were made, look for: “Eli - Spin2Core”
3. spin.h
 - A. Several data structures were added, all start with the prefix S2C_

- B. Several fields were added in the original data structures, all changes can be found by looking for: “Eli - Spin2Core“

Data Structures

Spin2core

STRUCT CONSTANTSTABLE

ConstantsTable is an array of the names of the constants in program. In order to avoid the replacement of all the CONSTS with their value at the preprocessing, we need to keep them.

STRUCT COMBMODULE

There is three kinds of combination need to be done: async as the result of 'run' command, partial sync as the result of the channel abstraction and the async of all modules under the SYSTEM module.

Each module that was combined, will not appear with its original name, but with the combined module's name (unless another instantiation of the module is needed).

STRUCT S2C_ROOT

The root is a global singleton struct that holds all information needed in the translation process.

Spin

STRUCT LEXTOK

The most basic struct. It holds the info about a lexical token produced in the parsing process. The info can be such as a CONST, type of a statement, operators as so forth. The Lextok is used sometime (for inner use of spin) as a control flow data (such as do loops) or as a linking structure in order to create a linked list. Most of those tokens are not necessary (and sometime even disturbing) for the translation and therefore removed from data structure during the translation.

Another use of the Lextok struct is to simulate an expression tree, where 'lft' and 'rgt' represent the operands of the expression.

STRUCT SYMBOL

Another basic struct. The Symbol struct holds the information of a symbol such as variable, file name, process and so forth.

STRUCT PROCLIST

The struct that holds the information of the process. It holds a list of the parameters of the process, a list of all the sequences in its body, the provided logical statement that enables the process (true by default) and a list of all variables used in the process (both local and parameters).

STRUCT ELEMENT, STRUCT SEQUENCE, STRUCT SEQLIST

Three data structures that are used in order to hold the sequence of all the statements in a process. Together they combine a complicated (in the spin2core point of view) linked list. When translating a sequence list, the spin2core goes over the list both extracting all unnecessary Lextok structs and gathering information concerning the combination of modules that will be needed.

STRUCT ORDERED

A data structure that is used in order to combine another complicated (in the spin2core point of view) linked list.

Algorithms

Overall

The translation has four main stages:

1. Preprocessing.
2. Parsing and data structures building.
3. Preparations and data structure adjustments.
4. Translation.

As a rule, there is no need for more than one pass over the data structures, with the exception of the sequences lists (which we pass several times). The passes are usually linear (since most data structures are linked lists), with the exception of the expression tree, which is DFS.

Preprocessing

Before the Spin preprocessor is activated, there's a need to prevent it from replacing all constants defined into their values (in order to translate them as strings and not as values).

The idea is to replace all statements as `#define N 26` to a statement as `int N = 26;` This way the spin parser will not replace the tokens.

Parsing and data structures building

The parsing is done by Spin itself and therefore there is no information about the algorithms used.

Preparations and data structure adjustments

Before translating the sequences of a process, we need to go over them, detect loops in the data structure, and extracting unnecessary tokens. While going over the structure, we build the list of the combined modules that needs to be translated.

Raising a flag in every element that was visited and a look ahead to the next element do the detection of the loops.

Translation

The translation order is inspired by the grammar of the core/XML.

TBD List

Synchronization of a channel

The channels are identified, and declared properly, but they are not synchronized just yet. The goal is to use partial synchronization as follow:

```
MODULE SYNCHRONIZED_CHANNEL () {  
  VAR  
    rcv : msg_type;  
    snd : msg_type;  
  (SENDER(snd) | send, put | CHAN(snd, rcv) | get, receive | RECEIVER(rcv))  
}
```

Note: If the sender/receiver is combined with other modules, you should use the combined module rather than the module itself. The combined module identifies that a module should send/receive a message and it has the proper parameters.

[Is that meaning that we can't use channels?](#)

Rendezvous

A channel can have a zero size buffer. In this case it means we would like to make a rendezvous between two processes. It might be working as is, by using the channels, but if not adjustments should be made.

Atomic and d_step

The difference between the two statements is that `d_step` will not perform a block unless ALL statements are executable, while `atomic` will perform all executable statements sequentially as much as it can.

In both cases we should use a global variable that will be function as semaphores and will be initiated with TRUE. A local variable will be added with the initial value of FALSE. The COJOIN of each module will be $(\text{glbl_sem} - \text{lcl_sem})$.

When a `d_step/atomic` statement is reached, the global flag will be assigned with FALSE, and the local one with TRUE.

The flags will be restored in the end of the `atomic/d_step`.

atomic

For each `atomic` (only `atomic!!`) statement we will add another transition:

```
TRANS original_trans
```

```

    enable: pc /\ some_condition
    assign: pc'; some_statement;
    glbl_sem' := FALSE; lcl_sem' := TRUE;
TRANS added_trans
    enable: pc /\ !some_condition
    assign: glbl_sem' := TRUE; lcl_sem' := FALSE;

```

By adding this transition there difference between the atomic and d_step will be achieved.

d_step

If a sequence of logical statements followed by an assignments, the whole sequence can be translated into one transition. The idea is perform a look ahead in the preparation stage and to unite each two expression- trees of the logical statements into one expression. The implementation already exists, but documented in the S2C_SequenceTranslator.c file.

instead of:

```

TRANS trans1
    enable: pc /\ some_condition
    assign: pc';
TRANS trans2
    enable: pc /\ some_other_condition
    assign: pc';
TRANS trans3
    enable: pc
    assign: pc'; some_statement;

```

Will be joined into:

```

TRANS joined_trans
    enable: pc /\ some_condition /\ some_other_condition
    assign: pc'; some_statement;

```

LOG File

Currently there is no log file, but there are several remarks that should be logged:

1. The use of the channel abstraction.
2. which transition are transitions that were translated as part of a sub-sequence of loop/if.
3. The use of special global variables (such as the glbl_sem suggested above).

Optional Arguments

Currently, the spin2core has to be ran from the directory where the input is found, the name of the output file is also fixed and the place of the XML files needed (dtd, xsl) should be manually written after the translation

We want to allow the user to give the spin2core, as optional arguments, both the path of the input file, the name for the output file and the absolute/respective place of the XML files, needed for the pretty print.