In the name of God

**1:**

we must solve some problems of representation in Promela.

* Promela does not support dynamically allocated data structures. To overcome this, we must explicitly model dynamically allocated memory, e.g., by arrays.
* Promela does not support functions. We must therefore emulate function calls, e.g., by creation of processes, and represent the parameter passing and function return by appropriate synchronizations.
* We must consider the issue of creating suitable test harnesses to exercise different conﬁgurations of concurrent processes.

Data Structures and Statements:

Promela has C-like syntax (actually it is compiled to C), but supports limited data-types. Our current translator handles a subset of ANSI C. Integer and array variables will be preserved in the same way as in C, while pointers and structures (and linked lists) will be handled separately.

Simple statements are translated straight-forwardly because of the similarity between statements in C and Promela. Control structures, including jumps (except return), selection, and iteration are translated to corresponding structures in Promela.

There is an example of these statements in article, page 3.

Dynamically Allocated Memory:

Promela does not support dynamically allocated data structures or pointers, hence we must emulate them.

Translating Pointer operations:

Assignments to pointers is translated to normal integer assignments.

There is an example with a figure of it in article, page 4.

Translating functions:

Since there is no function concept in Promela, they must be emulated. The only structuring concept available in Promela is the process, whence we will use them to emulate functions.

Each function deﬁnition will be translated to a Promela process type declaration which uses the corresponding local variables.

A function call will be translated into the creation of a process.

Parameters that are passed in the function call will be translated into parameters that are instantiated when the function is created.

Return values will be passed by passing a message back with the corresponding value as parameter to the calling context.

Function deﬁnitions:

in C will be translated into process type declarations in Promela, whose parameters will be the formal parameters of the function, plus an additional channel parameter that is used to communicate the return value. The body of the function is translated “as usual”. A return statement in C is translated into the transmission of a return message, after which process should be terminated immediately.

Function call:

will be translated into a sequence of statements in Promela. Before calling a function, a channel must be deﬁned for the transmission of return values. If the same function is called several times within the same context, only one channel is necessary. The actual call is translated into the creation of a process (using the run statement of Promela), to which the actual parameters, and the recently deﬁned channel are passed. The created process will execute asynchronously. Meanwhile, the calling context will be blocked on its next statement, which is a (blocking) receive of the return value on the return channel. The two functions, lock and unlock, which are used to represent the locking and unlocking mechanisms in C, will not be translated into process creations but into an atomic structure simulation.

Return statements:

A return statement in C passes a value to its caller and terminates the function. So we have to emulate both of these two properties. The ﬁrst property is emulated by transmitting a return value over the synchronous return channel. If a void is returned, a dummy value 0 will be sent. The second property is emulated by a goto end statement immediately after the transmission.

Recursive functions:

The above scheme for translating function deﬁnitions and calls automatically handles recursion. The main reason is that the declared return channels are local to the calling process, and that therefore multiple independent local copies of the return channels are created by the scoping mechanism of Promela. Promela allows at most 255 processes to be created in an execution, so this puts a limit on the recursion depth: this limitation is not signiﬁcant for the analyses that we envisage.

There is an example of it in article, page 5.

\*Related work number 5 seems related to our work!

+++++++++++++++++++++++++++++++++++++++++

+++++++++++++++++++++++++++++++++++++++++