Prism2Promela

Christopher Power, Alice Miller
Department of Computing Science
University of Glasgow
Email: {power, alice}@dcs.gla.ac.uk

Abstract—The Prism model checker [1] facilitates the formal
modelling and analysis of systems that exhibit random or
probabilistic behaviour. Prism lacks the provision of methods for
specification debugging. To help alleviate this problem, we present
a tool Prism2Promela for translating Prism into Promela [2].
Once in this form, the Spin model checker can be used to identify
errors in the Promela specification, which can be traced back
to corresponding errors in the original Prism specification.

I. INTRODUCTION

Prism [1] is a probabilistic model checker that employs
automatic formal verification techniques to enable the analysis
of stochastic systems specified in the Prism language. These
specifications give rise to models which take the form of
variants of Markov chains.

While Prism presents a powerful analytical framework, it
lacks the facility to track down the causes of simple errors,
such as the existence of deadlock states, or variable bound
violations. Prism can be used to show that deadlock states
exist, but not to identify behaviour leading to the error.
Similarly, if a variable bound is violated, no indication as to
the reason for the violation is provided. This can make the
process of model debugging slow and problematic.

By translating a specification to Promela, the Spin model
checker can be used to conduct random simulations or perform
an exhaustive search over all possible behaviours of a model.
Error behaviour is reported as a trace which can be analysed
to detect the cause of the error. If the error is the result of
an ill-judged assumption, it can be traced back to the original
Prism specification.

In this paper we present an overview of Prism2Promela,
a tool to automatically convert Prism specifications to
Promela. We illustrate features of the tool through the use of
two examples. Table I(A) shows a client-server specification in
Prism, adapted from [3] and its translation, while Table I(B)
gives a simple mutual exclusion model and its corresponding
translation. The underlying models are both Markov Decision
Processes (MDPs), but our technique applies equally to other
model types. Note some lines are marked with an * for ex-
planatory reasons only. Prism2Promela together with Prism
examples and their translations, can be downloaded from [4].

II. FEATURES

Prism2Promela can be used to translate a large subset
of the Prism language and is designed to ensure that no be-
vaviour is lost during the translation process. This is essential
as lost behaviour may correspond to lost errors.

All update statements of the form:

\[ \text{guard} \rightarrow \text{prob}_1 : \text{update}_1 + \ldots + \text{prob}_n : \text{update}_n; \]

are replaced by a sequence of \( n \) individual (non-deterministic)
choices in the translated model. This makes it easier to see
which update leads to a particular error. In our Promela
translation the statements of each proctype are enclosed within
a \( \text{do} \ldots \text{od} \) (repetitive choice) statement. This is common
practice (but not obligatory) in Promela specifications, and
mirrors the Prism specification style.

One feature of the Prism language that does not have an
obvious direct translation is the multi-way synchronisation of
modules. This means that two or more commands labelled
with the same action tag are executed simultaneously. This is
a frequently used language construct in Prism specifications.
To represent this behaviour in a Promela model we use
rendezvous channels chaining, as presented in (the appendix
to) [5].

The initial step of this translation is to concatenate the
guards of commands labelled with the same action tag into a
statement \( S \). Then \( S \) is used within the guard of the equivalent
statement in the Promela specification where the action tag
first appeared. Once \( S \) has been selected for execution all other
statements should be blocked until those associated with the
synchronisation have been processed.

This isolated execution of synchronised statements is guided
by the chaining of channels. Once a synchronised statement
has completed its update, a message is sent via a synchronous
channel to the process executing the next synchronised state-
ment, prompting its execution. A large number of statements
can be synchronised using this method, and if deadlock occurs
it will be clear which statement caused the error. For every
process that is involved in a synchronisation, an additional
choice is added to the corresponding proctype to allow a read
from the associated synchronous channel. This translation is
illustrated in Table I(A). Note that all statements in the Prism
specification with the serve tag are synchronised. In this
simple example, as only two processes are involved in the
synchronisation, only one synchronising channel is used. This
is not true in general. When \( j > 2 \) processes are involved in
the synchronisation, a chain of \( j - 1 \) synchronising channels
are used. The \( k \)th synchronising process reads from the \( k \)th
channel and writes to the \( (k + 1) \)th channel (modulo \( j - 1 \)).

Another common problem in model debugging is the break-
ing of Prism variable bounds. When this occurs, an error
message is returned to the user but no contextual information
provided. To provide this context, assertions are used in the
PROMELA specification. Assert statements allow one to trap the violation of simple safety properties during verification and simulation runs with SPIN. In our translation, for each bounded variable defined in a PRISM statement of the form var: [b1..b2], two assertions assert (var>=b1) and assert (var<=b2) are generated and placed within a single Monitor process. The transitions of the underlying model associated with the Monitor process are interleaved with those of the other processes, enabling any violation of an assertion to be caught.

We use SableCC [6] to produce a strongly typed Abstract Syntax Tree (AST) associated with the PRISM specification. A single pass of the AST allows us to produce an intermediate output which is, in turn, used to produce the PROMELA output.

### III. Results

Table I(A) contains PRISM code representing a simple client server model. In this specification the client sends periodic requests to the server for processing. The variable queue represents the number of requests currently waiting on a queue of size $N$ to be processed.

By running SPIN on the translated code, the first error is a violation of the statement assert (queue<=N). A guided simulation shows that this error occurs when the client process sends a request to an already full queue (i.e. when queue has value $N$). The solution is to amend the guard in the first statement in the PRISM code marked with an asterisk to queue<N. A similar assertion leads one to tighten the guard on the other marked statement.

The example contained in Table I(B) is a (flawed) mutual exclusion specification. Two processes should not be simultaneously in their critical state (defined by $(x = y = 2)$).

By running SPIN on this translated code, we discover that there is a deadlock state when $x = y = 1$. Inspection of the error trace shows the system deadlocks as there are no defined commands to deal with $x$ or $y$ when they have value 1.

### IV. Conclusion

PRISM2PROMELA is useful for uncovering simple errors in PRISM models. It can be used to translate a large subset of the PRISM language into PROMELA without losing any of the original behaviour.

A language feature that currently can not be translated is that of user-defined formulae. This could be overcome to some extent by using a library of the most common used formulæ and inserting where appropriate.

Currently a drawback of the tool is that a PRISM user must be an expert in interpreting the error traces produced by SPIN. He/she must first identify the source of an error in the PROMELA specification, and then map it back to an error in the original PRISM specification. One way to avoid this would be to parse the error trace and return one presented in a more (non SPIN-specific) readable form. This is the subject of current work.

### REFERENCES


