ETCH: An Enhanced Type Checking Tool for Promela

Alastair F. Donaldson and Simon J. Gay

Department of Computing Science
University of Glasgow
Glasgow, Scotland.
{ally,simon}@dcs.gla.ac.uk

Abstract. We present ETCH, an enhanced type checking tool for the Promela language. This tool uses standard type checking in conjunction with constraint-based type inference to detect type errors in Promela models which cannot currently be detected by SPIN before verification or simulation. ETCH allows for more rapid development of Promela code, and increased confidence in verification models used with SPIN. Since the utility of model checking depends heavily on the correctness of the model being verified, our tool is a significant contribution.

1 Introduction

Model checkers and type checkers have both been classed as *light weight* formal methods [9]. Type checkers for high level languages have been widely used in everyday software development for many years, and model checkers are increasingly being used in the development of reliable hardware and software. Verification models for use with a model checker are usually written in a high level language, and if this language includes type information, type checking can be used to aid the development of sensible models. The Promela language [6] includes a rich set of data types, including numeric types, enumerated message types, and types for communication channels. Before simulation or verification of a Promela model, the SPIN model checker performs some type checking to detect errors in the model. However, the type checking performed by SPIN is limited. Certain kinds of type errors which are not currently detected by SPIN could be detected in a straightforward manner using the type information included in a Promela model. More subtle errors involving dynamic channel passing (an attractive feature of the language) cannot be detected directly from this type information, since channel types in Promela are not fully specified. To be detected statically, such errors require additional type information to be inferred from the model.

We present ETCH (Enhanced Type CHecker), a type checking tool for Promela. This tool uses standard type checking in conjunction with constraint-based type inference to detect type errors in Promela models which cannot currently be detected by SPIN. ETCH allows for more rapid development of reliable Promela

code, and increased confidence in verification models used with SPIN. Our approach requires no modification to the syntax of Promela, and no extra type declarations are necessary—type checking is performed by type inference based on the existing type information in a Promela specification. Programmers in any language know that type errors are a frequent kind of mistake, and that compile-time type checking is very useful. Promela is no exception. For example, in an informal survey of Promela code produced for a student assignment, ETCH was used to detect numerous type errors which were not detected by SPIN. Since the utility of model checking depends heavily on the correctness of the model being verified, our tool is a significant contribution.

In Section 2 we give some examples of type errors in Promela code which are not detected by SPIN until verification time. In Section 3 we outline the design of ETCH. We discuss two interesting features of the tool—constraint-based type inference, and recursive channel types—in Sections 4 and 5 respectively. Conclusions and plans for future work are given in Section 6. ETCH can be downloaded from our website [3].

2 Example

To illustrate the kind of type errors which currently are not detected by SPIN before verification time, we consider a generic client-server model adapted from [6, Chapter 15]. The Promela code for this model is given below, annotated with asterisks which are for discussion purposes, and should otherwise be ignored.

```
mtype = {request,deny,hold,grant,return}
chan server = [0] of {mtype,chan}
chan null = [0] of {mtype,chan}

proctype Agent(chan listen, talk)
{ do
   :: talk!hold(listen)           (**)
   :: talk!deny(listen) -> break
   :: talk!grant(listen) ->       (***)
wait: listen?return(null); break
  od;
  server!return(listen)   }

active[2] proctype Client()
{ chan me = [0] of {mtype,chan};
  chan agent;
end: do
      :: timeout ->
         server!request(me);
         do
          :: me?hold(agent)
          :: me?deny(agent) -> break
          :: me?grant(agent) ->
             agent!return(null); break
         od
  od   }
```

```
active proctype Server()
{
  chan agents[2] = [0] of {mtype,chan};
  chan pool = [2] of {chan};
  chan client, agent;
  byte i;
  do
   :: i < 2 -> pool!agents[i]; i++
   :: else -> break
  od;
end:
  do
   :: server?request(client) ->
      if
       :: empty(pool) -> client!deny(null)
       :: nempty(pool) -> pool?agent;
          run Agent(agent,client)          (*)
      fi
   :: server?return(agent) -> pool!agent
  od
}
```

We now suggest three changes to the above model which introduce type errors, and discuss the implications of each error.

Error 1 *The statement* `run Agent(agent,client)` *at* `(*)` *is replaced with the statement* `run Agent(agent)`. This is clearly a type error since the *Agent*

proctype requires two parameters and only one has been supplied. The SPIN syntax checker does not detect this error. If this statement is executed during simulation then the message `Error: missing actual parameters: 'Agent'` is given, and simulation halts. During verification, an *Agent* process will be instantiated on execution of this statement, but the *talk* parameter of the *Agent* will be an uninitialised channel, resulting in an error from SPIN when this channel is used. ETCH detects this error without needing to use type inference.

Error 2 *The statement* `talk!hold(listen)` *at* `(**)` *is replaced with the statement* `talk!listen(hold)`. From the model, we can see that *Agent* processes are instantiated only by the *Server* process. The *talk* parameter of an *Agent* corresponds to the *client* variable of the *Server* process, and this is in turn received on the channel *server*, from a *Client* process. The channel which the *Client* process sends is *me*, which accepts messages of the form $\{mtype, chan\}$. Thus the *talk* parameter of an *Agent* accepts messages of the form $\{mtype, chan\}$. Our modification introduces a type error since an attempt is made to send a message of the form $\{chan, mtype\}$ on the channel *talk*. This error is not picked up by the SPIN syntax checker. During simulation, SPIN reports a type-clash if this statement is executed. An invalid end state is reported during verification of the model with this error, and the corresponding counterexample contains a warning of a type-clash. In Section 4, we describe how ETCH detects this error using constraint-based type inference.

Error 3 *The statement* `talk!grant(listen)` *at* `(***)` *is replaced with the statement* `talk!grant`. By the same argument presented for Error 2, this modification causes an error since only a single field has been sent on the channel *talk*. This error is not detected by the syntax checker. Since the argument *grant* has type *mtype*—the correct type for the first message field—SPIN does not flag up an error when this statement is executed, even though a field is missing from the message (during simulation a warning is given). The message is received by a *client* process via the statement `me?grant(agent)`. However, the received channel *agent* is uninitialised since no channel was actually sent by the *Agent* process. The *Client* process then attempts to execute the statement `agent!return`, which causes an error since *agent* is not initialised. This error is less easy to isolate than Errors 1 and 2 since it has effect at a later stage in system execution. In Section 4 we describe how ETCH detects this error using constraint-based type inference.

ETCH detects each of these errors statically, before simulation or verification of the model, making it easier to eliminate them.

3 Overview of ETCH

We have implemented ETCH in Java, using the compiler generation framework SableCC [5] to generate a parser for Promela based on the grammar provided in [6]. The type system used by ETCH is based on type systems for the π-calculus

[10, Chapter 6]. The core grammar which ETCH uses to represent types is as follows:

$$
\begin{array}{lll}
T ::= \mathsf{numeric\ type} \mid \mathsf{pid} \mid \mathsf{mtype} \mid \mathsf{bool} & & \textit{basic types} \\
\quad \mid \mathsf{chan}\, C & & \textit{channels} \\
\quad \mid \mu X.T & & \textit{recursive types} \\
C ::= X & & \textit{type variables} \\
\quad \mid \{T_1, \ldots, T_n\} & & \textit{tuples}
\end{array}
$$

Array and record types are also supported, with the same restrictions as are imposed by SPIN. In themselves they do not generate any complications for type checking. Type checking of Promela code is performed by ETCH using the type information included with variable declarations. Each time a channel declaration is encountered, ETCH chooses a fresh type variable to represent the type of messages which may be sent on the channel. Constraints on the form of type variables are generated based on applied occurrences of channel identifiers. A standard constraint-based type inference algorithm [1, Chapter 6] is used to solve these constraints in order to determine values for the type variables. Recursive channel types are introduced in order to solve constraints of the form $X = \mathit{chan}\, C$ where X occurs in C. We discuss constraint-based type inference and recursive channel types in Sections 4 and 5 respectively.

Promela has a variety of numeric types representing different numeric ranges, including a type of unsigned integers which is parameterised by the word length. ETCH implements the natural subtyping relations, e.g. $\mathit{byte} <: \mathit{unsigned}(12) <: \mathit{short} <: \mathit{int}$. For programming convenience, $\mathit{bit} <: \mathit{bool}$, although *bool* is not related to other numeric types. By default, ETCH (like SPIN) treats the types *pid* and *byte* as equivalent, but there is an option to regard them as different types. This is useful in e.g. work by the first author on symmetry detection [4].

Since ETCH is not part of SPIN, any errors reported by ETCH are really just warnings. Therefore we have taken a strict approach in deciding what constitutes a type error. SPIN treats enumerated *mtype* variables and values as if they were numeric, and thus allows *mtype* variables and values to be used in any context in which numeric variables and values can be used. For example, if a, b and c are *mtype* variables, SPIN allows a statement such as $a = b + c$. This use of message types in an arithmetic context is usually bad practice—arithmetic on *mtype* values is also disallowed in [7]—and so ETCH will report a type error in such cases. Running ETCH on the client-server model, modified to include Error 1, results in the following error message:

```
Line 47: Error - the proctype "Agent" expects 2 arguments but
1 has been supplied
```

The message corresponding to Error 3 is:

```
Line 9: Error - arguments of different lengths have been used
for the same channel.  Unable to unify "{mtype,chan X7}"
and "{mtype}"
```

Here `X7` is the name of a type variable used by the type inference algorithm (see Section 4). Error 2 generates a similar warning.

4 Constraint-Based Type Inference

Channel types are only partially specified in Promela. For example, the declaration `chan server = [0] of {mtype,chan}`, in the client-server model of Section 2, specifies that the second field of a message to be sent on *server* should be a channel, but does not specify the type of this channel. ETCH represents channel types fully, using type-variables for types which are not known (see the grammar for types presented in Section 3). To illustrate the approach of constraint-based type inference used by ETCH, consider the client-server example, modified to include Error 2. The `listen` and `talk` parameters of the *Agent* proctype are assigned types $chan\,X$ and $chan\,Y$ respectively, since the types of messages which they accept are not specified. The (modified) statement `talk!listen(hold)` causes the constraint $chan\,Y = chan\{chan\,X, mtype\}$ to be stored, while the statement `talk!deny(listen)` causes the constraint $chan\,Y = chan\{mtype, chan\,X\}$ to be stored. Attempting to unify these constraints results in the constraint $chan\,X = mtype$, which cannot be unified, thus a type error is generated.

Now consider the client-server example modified to include Error 3. Again, `listen` and `talk` are assigned types $chan\,X$ and $chan\,Y$ respectively. The statements `talk!hold(listen)` and `talk!deny(listen)` both result in the constraint $chan\,Y = chan\{mtype, chan\,X\}$. However, the (modified) statement `talk!grant` results in the constraint $chan\,Y = chan\{mtype\}$. Unification of these constraints fails since the tuples $\{mtype, chan\,X\}$ and $\{mtype\}$ cannot be unified, being of different lengths.

For a more general description of constraint-based type inference, see [9, Chapter 22]. Our implementation is based on an algorithm described by Aho et al. [1, Chapter 6].

5 Recursive Channel Types

Consider the following Promela code: `chan A = [1] of {chan,bit}; A!A,0`. The channel A accepts messages with two fields. The second field should be of type *bit*, and the first must be a channel of *the same type as* A. This because the channel A is sent on itself by the statement `A!A,0`. Thus the type of channel A is recursive, and using standard notation, we can express the type of A as $\mu X.chan\{X, bit\}$. The above Promela code fragment is legitimate, and this kind of channel usage has been employed in realistic Promela models, e.g. a model of a telephone system [2]. We have incorporated recursive types into ETCH. A recursive type expression has infinitely many equivalent syntactic representations, depending on where the recursive μ construct appears in the expression, and on how far the type expression has been *unfolded*. A recursive type expression resulting from the unification of a set of constraints may look very complex, even though it is equivalent to a much shorter expression. When presenting types to the user in type errors, it is desirable to convert recursive types into their simplest forms. ETCH incorporates a minimisation algorithm for recursive types.

This algorithm is based on an algorithm for minimising a deterministic finite automaton [8, Chapter 2], and involves finding the largest bisimulation on a type expression. For example, the types

$$\mu X.chan\{chan\{X, bit\}, bit\} \quad \text{and} \quad chan\,\mu X.\{chan\,X, bit\}$$

are the same, and are equivalent to the type $\mu X.chan\{X, bit\}$, which is the minimal form to which ETCH converts both type expressions.

6 Conclusions and Future Work

We have presented ETCH, an enhanced type checker for Promela. ETCH extends the capabilities of SPIN by detecting type errors in a model before simulation or verification. Eliminating errors using ETCH results in more reliable verification models for use with SPIN, and thus increased confidence in SPIN verifications. The standard Promela language is handled in full by ETCH. Inline macros, not part of the language grammar given in [6], cannot be handled at present. Extending ETCH to handle inline macros should be straightforward. We also intend to improve the quality of error messages arising due to errors detected by the type inference algorithm.

References

1. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers—Principles, Techniques and Tools*. Addison-Wesley, 1986.
2. M. Calder and A. Miller. Using SPIN for Feature Interaction Analysis - a Case Study. In *Proceedings of the 8th International SPIN Workshop on Model Checking Software*, LNCS 2057, pages 143–162. Springer, 2001.
3. A. F. Donaldson and S. J. Gay. ETCH Website: `http://www.dcs.gla.ac.uk/people/personal/ally/etch/`.
4. A. F. Donaldson and A. Miller. Automatic Symmetry Detection for Model Checking Using Computational Group Theory. In *Proceedings of the 13th International Symposium on Formal Methods*, LNCS 3582, pages 418–496. Springer, 2005.
5. E. Gagnon and L. J. Hendren. SableCC, an object-oriented compiler framework. In *26th Technology of Object-Oriented Languages and Systems*, pages 140–154. IEEE Computer Society Press, 1998.
6. G. J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, 2003.
7. S. Leue, R. Mayr, and W. Wei. A Scalable Incomplete Test for Message Buffer Overflow in Promela Models. In *Proceedings of the 11th International SPIN Workshop on Model Checking Software*, LNCS 2989, pages 216–233. Springer, 2004.
8. P. Linz. *An Introduction to Formal Languages and Automata*. Jones and Bartlett, 1986.
9. B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.
10. D. Sangiorgi and D. Walker. *The π-calculus: a Theory of Mobile Processes*. Cambridge University Press, 2001.