Type Inference and Strong Static Type Checking for Promela

Alastair F. Donaldson
Oxford University Computing Laboratory, Oxford OX1 3QD, UK

Simon J. Gay
Department of Computing Science, University of Glasgow, Glasgow G12 8QQ, UK

Abstract

The SPIN model checker and its specification language Promela have been used extensively in industry and academia to check logical properties of distributed algorithms and protocols. Model checking with SPIN involves reasoning about a system via an abstract Promela specification, thus the technique depends critically on the soundness of this specification. Promela includes a rich set of data types including first-class channels, but the language syntax restricts the declaration of channel types so that it is not generally possible to deduce the complete type of a channel directly from its declaration. We present the design and implementation of ETCH, an enhanced type checker for Promela, which uses constraint-based type inference to perform strong type checking of Promela specifications, allowing static detection of errors that SPIN would not detect until simulation/verification time, or that SPIN may miss completely. We discuss theoretical and practical problems associated with designing a type system and type checker for an existing language, and formalise our approach using a Promela-like calculus. To handle subtyping between base types, we present an extension to a standard unification algorithm to solve a system of equality and subtyping constraints, based on bounded substitutions.

Key words: Promela, SPIN, type checking, type inference, model checking.

ACM Subject Classification: D.2.4 [Software/Program Verification]: Model checking, Formal methods, Validation; D.3.3 [Language Constructs and Features]: Constraints, Data types and structures, Recursion; F.3.3 [Studies of Program Constructs]: Type structure

1. Introduction

The SPIN model checker [16] and its specification language Promela have been used extensively in industry and academia to check logical properties of distributed algorithms and protocols (see e.g. [4,12,17,32]). Model checking with SPIN involves reasoning about a system via an abstract Promela specification, thus the technique depends critically on the soundness of this specification. Promela includes a rich set of data types, including first-class communication channels which can be transmitted along other channels in the style of the pi calculus [31]. However, although type information for program variables is declared and checked, the language allows information about the types of channels (i.e. the allowed types of messages) to be only partially specified, and there are situations in which a complete specification is not even permitted.

Channel declarations in Promela fall into the following three categories.

1. Declarations that specify complete type information, for example:

   `chan a = [0] of (int,int)`

   which declares a channel whose buffer has size 0 (a rendezvous channel) and which can carry messages consisting of pairs of integers.
2. Declarations that specify no message type information, for example:

```promela
d = [0] of {chan}
```

3. Declarations of channels that carry channels, for example:

```promela
c = [0] of {chan}
```

In this case the programmer does not have the option of replacing the innermost `chan` with a more precise type, e.g. the following declaration is not allowed:

```promela
d = [0] of {chan(int, int)}
```

Declarations in category (1) are unproblematic and allow the SPIN type checker to check the types of messages sent and received\(^1\) But declarations in categories (2) and (3) limit the scope of the type checker. For example, consider the Promela specification in Figure 1. Here \(a\) is a global channel, \(b\) is a channel sent from \(P\) to \(Q\), and \(c\) is the channel reference into which \(Q\) receives \(b\). At runtime, \(c\) becomes instantiated to \(b\), and the send command \(b!i\) synchronizes with the receive command \(c?j\). The result is that the value of \(i\), which is an int, is put into the variable \(j\), which is a byte. We regard this as a type error, since the range of values belonging to the byte type is smaller than those for the int type. However, the type checker built into SPIN is not able to detect this error, because the type of channel \(a\) is not fully specified and therefore both \(P\) and \(Q\) seem to be using it correctly.

Situations such as the example above frequently, although not always, signify errors in the Promela specification; to put it another way, they indicate errors in the modelling of the system which is to be analyzed. This can lead to errors during model checking, perhaps after a long period of checking; alternatively, it can lead to unexpected behaviour of the model and therefore to misleading results from model checking. In the example above, assigning an integer value into a byte variable might be allowed by SPIN, but the user might not know what the runtime behaviour (e.g. truncation or saturation) will be.

The essential problem is that Promela allows mobile channels, but does not use an appropriate type system for mobile channels. We have addressed this problem by designing E\(\text{TCH}\) (Enhanced Type CHecker), an enhanced type checking tool for Promela. E\(\text{TCH}\) facilitates the analysis of Promela specifications with mobile channels using an appropriate type system, but does not require changes to the syntax of the Promela language. E\(\text{TCH}\) is based on the following type-theoretic technology:

**Fully-specified channel types**, as found in type systems for the pi calculus\(^3\). For example, the type:

```promela
c=\{int, chan\{bool, chan\{int\}\}\}
```

could be associated with a channel which can only be declared in Promela as:

---

\(^1\)Even so, the type checking carried out by the SPIN tool remains minimal: the tool follows a weak type system where channels and integers are partially interchangeable.
chan a = [0] of (int, chan)

Constraint-based type inference, as found in functional languages such as ML \cite{22} and Haskell \cite{25}. This allows fully-specified channel types to be used internally without requiring the programmer to declare them explicitly. In particular, this means that we do not need to modify the syntax of Promela.

Recursive types, again as found in functional languages and in several type systems for the pi calculus. Recursive types naturally arise while solving constraints among channel types. For example, the declaration `chan a` together with the send command `a!a` requires channel `a` to have a recursive type defined by the equation \( X = \text{chan}\{X\} \). The programmer never needs to write a recursive type, but they can appear in error messages, and are presented in as palatable a form as possible.

Subtyping, which accounts for the relationships between the various numeric types supported by Promela. The subtyping relation is defined by:

\[
\text{bit} <: \text{byte} <: \text{short} <: \text{int} \quad \text{bit} <: \text{bool}.
\]

The constraint-based type inference system generates constraints involving subtyping relationships as well as equalities. In general this complicates constraint solving, but it turns out that in Promela there is always just enough explicit type information to enable us to obtain a straightforward algorithm.

\text{ETCH} is able to detect many situations which can reasonably be regarded as type errors but which are not detected by \text{SPIN}. However, because \text{ETCH} is implemented as a standalone tool, \text{SPIN} users are free to view its error messages as warnings and ignore them if they choose.

1.1. Contribution and structure of the paper

The main contribution of this paper is the design and implementation of an enhanced type checker for Promela, which can statically detect errors that \text{SPIN} would not detect until simulation/verification time, or that \text{SPIN} may miss completely. The tool, \text{ETCH}, is publicly available\footnote{\text{ETCH} can be downloaded from: http://www.allydonaldson.co.uk/etch, or from the \text{SPIN} website: http://spinroot.com} to the Promela/\text{SPIN} community, to aid the construction of robust specifications for formal verification. The paper also provides a case study in applying, and adapting in a non-trivial way, well-understood techniques from type theory to an existing language whose syntax we did not want to change.

We believe that by providing a concrete exposition of type-theoretic techniques for a practical, imperative language, our case study will be a useful reference for practitioners who are not experts in type theory.

We further motivate the need for enhanced type checking of Promela via a discussion of example Promela specifications, which also provides an overview of the language (Section 2). After this, we make the following contributions:

- We introduce Promela-calculus, a simple language based on Promela but with complete type information. We present a type system for Promela-calculus, and establish that well-typed Promela-calculus specifications are free of communication errors (Section 3).

- We consider a variant of Promela-calculus in which type information is incompletely specified, as is the case for Promela. We present a constraint-based type checking method that guarantees that specifications with soluble sets of constraints are free from communication errors. The approach is based on standard techniques for type reconstruction, extended to handle subtyping between base types via \textit{bounded substitutions} (Section 4).

- Although our results are presented in terms of Promela-calculus, our implementation \text{ETCH} is for the full Promela language. We present notes of some interesting practical issues, including the use of most a general unifier for inferring further properties of channel usage, and the way our implementation deals with recursive types (Section 5).

We conclude (Section 7) after a discussion of related work (Section 6). An operational semantics for Promela-calculus, as well as proofs of novel results, appear in the appendix.

2. Examples of Errors Detected by \text{ETCH}

To motivate the need for applying type inference to Promela, and to illustrate the kind of type errors which \text{ETCH} detects, we present two example Promela specifications from the literature, into each of which we inject two errors.
Table 2: Summary of the ability of SPIN to detect Errors 1–4 statically, during simulation, or during verification. In some cases, the error goes undetected and leads to an erroneous verification result in the sense that the number of reachable states explored by SPIN is affected by the presence of the error. Our enhanced type checker, ETCH, is able to detect all these errors statically.

In each case, SPIN does not detect the error before simulation or verification, and in some cases does not catch the error at all. This is summarised in Figure 2. ETCH, on the other hand, uses the techniques described later in the paper to detect these errors statically, before simulation or verification of the model, making it easier to eliminate them. We also discuss a scenario where static type checking can be restrictive. Results in this section were obtained using SPIN version 5.2.4.

2.1. Client-Server specification

We first consider a generic client-server specification adapted from [16, Chapter 15], where a detailed description of the protocol is provided. The Promela code for this specification is given in Figure 3.

The specification introduces an enumerated type, mtype, to represent messages in the protocol, and declares two global channel variables: server and null. Both channels are rendezvous (having length zero), and accept pairs of messages consisting of an mtype and a channel. Three process types are then declared, via the proctype construct.

The Client proctype is prefixed with active [2] to indicate that two instances of this proctype should be live when simulation or verification of the specification begins; the Server proctype is prefixed simply with active, indicating that one instance of this proctype should be live. The Agent proctype has no such prefix: this indicates that no instances of this proctype are live initially, instead instances are spawned via the run statement at line 37. This non-blocking statement instantiates an asynchronous Agent with the given parameters. Statements in Promela are separated either by ; or ->, which can be used interchangeably. However, it is common practice for -> to follow a boolean guard, e.g. timeout at line 18 (timeout is a built-in boolean variable which has value true if and only if every other process is blocked). Looping behaviour is specified using the do..od construct; this construct selects between a series of options, each prefixed ::, non-deterministically choosing between options for which the first statement is executable. The break keyword causes a jump to the end of the innermost do..od construct. The specification makes frequent use of channels, using the send (!) and receive (?) operators, as well as built-in functions to test whether or not a channel is empty.

The specification is interesting to study as it exhibits dynamic channel passing – the Server process holds a pool of channels which are used to connect Agent and Client processes, and Agent processes use the server channel to return their input channels to the pool. Also, the specification involves implicit recursive types.

Figure 4(a) shows the output generated by ETCH for this example. ETCH reports that the specification is well-typed, and displays the complete type for each variable. All channels with the exception of pool are found to have the recursive type satisfying the equation $X = \text{chan}\{X, \text{mtype}\}$. We write this type as rec $X$, chan{$X, \text{mtype}$}. The pool channel accepts messages which are channels of the above type, thus the type of pool is $\text{chan}\{\text{rec } X, \text{~chan}\{X, \text{mtype}\}\}$.

We consider two changes to the client-server specification which could conceivably arise due to programmer errors.

**Error 1** Statement talk!hold, listen at line 6 of Figure 3 is replaced with talk!listen, hold.
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)
        :: server!return, agent -> pool?agent
    od
}

Figure 3: Client-server specification.

From the specification, we can see that Agent processes are instantiated only by the Server process at line 37. The talk parameter of an Agent corresponds to the client variable of the Server process, which is defined at line 34 by a receive on the server channel. The corresponding sender for this statement is a Client process, which sends the channel me at line 18. Channel me accepts messages of the form {mtype, chan}. Thus the talk parameter of an Agent accepts messages of the form {mtype, chan}. The modification introduces an error since an attempt is made to send a message of the form {chan, mtype} on the channel talk.

As indicated in Figure 2, this error is not detected by SPIN at all. The reachable state-space associated with the specification differs depending on whether or not the error is present, thus Error 1 leads to an erroneous verification result.

In contrast, ETCH detects this error statically, producing the following error message: Error at line 6: "mtype" is not compatible with type "rec X.chan{X,mtype}".

Error 2 Statement talk!grant, listen at line 8 of Figure 3 is replaced with talk!grant.

By an argument similar to that for Error 1, this modification causes an error since only a single field has been sent on the channel talk. This error is not detected statically by SPIN. During simulation, SPIN does not report an error when this statement is executed. The message is received by a Client process via the statement me?grant(agent) at line 21. However, the received channel agent is uninitialised because no channel was actually sent by the Agent process. The Client process then attempts to execute agent!return(null). This causes an error during simulation as agent is not initialised. Thus SPIN indirectly catches the error during simulation. During verification, SPIN immediately halts with
an error when the statement of Error 2 is executed, thus this error does not lead to an erroneous verification result.
The behaviour for SPIN with respect to Error 2 is summarised in Figure 2.

ETCH detects this error statically, producing the following message:

Error at line 8: arguments of lengths 1 and 2 have been used with the same channel.

2.2. Telephone specification

The second specification we consider is shown, in part, in Figure 5. This specification models a telephone network, and is adapted from a specification presented in full in [4], in which the telephone system is augmented with a selection of features, and SPIN is used for feature interaction analysis. The specification illustrates a more explicit use of recursive channel types than for the client-server example of Section 2.1, shows the capability of ETCH to detect subtyping errors via type reconstruction, and shows how ETCH can be used to trap specification errors which alter the state-space of the associated model, but which are not detected by SPIN at all, even during verification.

This specification uses several Promela features which were not illustrated by Figure 3. Buffered channels are specified via a non-zero length – the channels null, zero, one, and two all have capacity 1. At line 4, an array of three channels is declared. The specification uses the if .. fi construct for conditional selection (this is similar to the do .. od construct discussed in Section 2.1 except that it does not cause looping behaviour), and includes labelled statements which may be the targets for goto instructions, in the style of the C language. At line 16, arguments to the receive operator are enclosed in angle brackets: this indicates that the receive should be non-destructive, copying values from the channel into the supplied arguments but not removing a message from the channel. Rather than specifying initial processes via the active keyword, this specification declares an init process, which is live by default when verification or simulation begins. The init process initialises the partner array, and instantiates three User processes. These statements are enclosed in an atomic block, to specify that they should be executed indivisibly.

Each User process in the specification takes a channel parameter, self, which accepts data in pairs consisting of a channel reference (the channel of a User process to whom the given User is connected), and a bit representing the connection status of the call. Statement self!self,0 at line 14 of Figure 5 shows that it permissible for the channel self to hold a reference to itself. This indicates that the corresponding User process is engaged, but not connected
1. mtype = { on, off, st_idle }

4. proctype User (byte selfid; chan self) {
5. chan messchan = null; bit messbit = 0;
6. mtype state = on, dev = on;
7. byte partnerid = 6;
8. O_T_Null:
9. if :: empty(self) -> state = on;
10. dev = off;
11. goto Auth_Orig_Att
12. :: full(self) -> self?<partner[0],messbit>;
13. if :: empty(partner[0]) -> self?messchan,messbit;
...}
14. fi;
15. if :: empty(partner[1]) ->
16. partner[1]=self,0;
17. self?messchan,messbit;
18. self?partner[1],0;
...}
19. fi
20. init {
21. atomic {
23. run User(0,zero); run User(1,one); run User(2,two);
24. } }

Figure 5: Part of a telephone specification, adapted from [3].

[3]. Figure 4(b) shows the complete type information for this specification as reconstructed by Etch. Notice the appearance of recursive types of a similar form to those detected by Etch in Figure 4(a).

Again, we consider the introduction of potential programmer errors to the specification.

Error 3 Declaration chan two = [1] of { chan, bit } at line 3 is replaced with chan two = [1] of { mtype, bit }.

This modification intuitively introduces a type error: the channel two is passed as parameter self to a User process at line 292, and at line 14 the statement self!self,0 indicates that the first field of a message on self should be a channel, not an mtype expression (which is the type of the first field of two in Error 3).

In practice, as indicated in Figure 2, SPIN does not flag this up as an error, statically or dynamically. SPIN uses a byte to represent the value of both chan and mtype variables, so assigning between chan and mtype variables does not cause any loss in precision. For this reason and because the channel self is consistently used as if its first field were a channel, the model associated with the specification of Figure 5 is the same whether or not Error 3 is introduced.

We view Error 3 as a genuine error for three reasons. Firstly, using the type name mtype where chan would be more sensible makes the specification harder to understand. Secondly, relying on the fact that SPIN treats mtype and chan variables identically provides little guarantee that a specification will behave as expected with future version of SPIN (or with other tools that use Promela as an input language (e.g. p2b [3]). Lastly, the XSPIN user interface provides a feature which allows the values of variables to be displayed during simulation. This feature replaces the numeric values associated (internally) with mtype variables by the symbolic names provided by the user via the mtype = {...} declarator. This means that in the presence of Error 3 messages on channel two will be displayed using mtype names rather than channel names for their first field, making simulation confusing.

Etch detects this error statically and displays the following message:
Error at line 17: "chan(mtype,bit)" is not compatible with type "mtype".

Error 4 Statement self!partner[0],0 at line 89 is replaced with self!partner[0],9.
mtype = { ho_cmd, ... };

proctype CC(chan f, ...) { /* communication controller */
    chan m_new;
    do :: l?m_new -> f!ho_cmd; f!m_new;
    ...
}

proctype BS(chan f, ...) { /* base station */
    chan v;
    do :: f?ho_cmd -> f?v; /* handover command */
    ...
}

Figure 6: A snippet of the mobile1 specification, provided with the SPIN distribution, which illustrates a limitation of static type checking with \textsc{Etch}. Proctypes \texttt{CC} and \texttt{BS} communicate both mtype and chan messages on their channel parameter, \texttt{f}, thus \textsc{Etch} reports a type error. Nevertheless, the full specification is free of communication errors.

In this final erroneous modification to the telephone specification, we assume that the user has mistyped the value 0 as 9. The SPIN tool includes the \texttt{bit} data type for convenience, but does not complain if a value outside \{0, 1\} is used in a \texttt{bit} context. SPIN uses an approach similar to that of C with respect to boolean expressions, and treats the value 0 as usual, and any other value as 1. This means that the statement \texttt{self!partner[selfid],9} means the same thing as \texttt{self!partner[selfid],1}, which is clearly different to the intended statement.

As with Error 1, this error leads to an erroneous verification result: SPIN does not detect the error statically or during verification, and the size of the reachable state-space associated with the telephone specification differs depending on whether or not Error 4 is present. During simulation, SPIN does report a truncation error when the statement of Error 4 is executed.

\textsc{Etch} detects the error statically, with the following message:

\texttt{Error at line 89: type "byte" occurs in a context where it is required to be a subtype of "bit".}

Errors 1 and 4 are arguably the most serious of the errors we have discussed, since they go undetected by SPIN but lead to a semantic difference in the associated models, illustrated by the change in state-space size shown in Figure 2.

In the presence of such errors, the user might try to verify properties of the erroneous specification, accepting that certain properties are true, when in fact the properties hold \textit{vacuously}: they have not been verified over the specification intended by the user.

2.3. The price of static type checking

Before we present the techniques on which our enhanced type checker is based, it is fair to point out that while static type checking can quickly identify genuine errors, a static type system always leads to the rejection of some correct programs. In our context, “correct” means Promela specifications that, when executed, do not exhibit communication errors.

An important example of this is the mobile1 specification provided with the SPIN distribution. This specification models a cell-phone hand-off strategy in a mobile network, and is translated from a pi calculus description [24]. A small fragment of the specification is shown in Figure 6. Instances of proctypes \texttt{CC} (communication controller) and \texttt{BS} (base station) are spawned, with the same channel passed as parameter \texttt{f}. The \texttt{CC} process receives a channel into local variable \texttt{m_new}, then sends two messages on channel \texttt{f}: a hand-off command (represented by mtype value \texttt{ho_cmd}), followed by the channel stored in \texttt{m_new}. Correspondingly, the \texttt{BS} process uses \texttt{f} to receive the hand-off command, followed by the channel.

\textsc{Etch} rejects this specification, since channel \texttt{f} in both proctypes is used as if it had both type \texttt{chan{mtype}} and \texttt{chan{chan{...}}}.

However, the specification executes without communication errors since, as discussed for Error 3 in Section 2.2, SPIN uses the same internal representation for mtype and chan values. Furthermore, the mobile1 specification is designed such that whenever a chan value is sent on a channel, the corresponding receive is into a chan variable, and similarly for mtype values.

The specification can be re-written to be accepted by \textsc{Etch}, by representing messages on \texttt{f} as (mtype, chan) pairs. Then each communication using an mtype is replaced with a communication using the same mtype and a null channel,
and each communication using a chan replaced with a communication using a null mtype and the same chan. The advantage of the representation is that the distinction between mtype and chan messages is made explicit, and the specification is deemed well typed. The price is that this representation requires a slightly larger state-vector, since the width of channel variables is increased. We note that this is the only example provided with the SPIN distribution that is beyond the scope of ETCH.

The scenario where the type of message to be communicated via a channel depends on the status of a communications protocol can often be captured using session types [18], which have been successfully applied to practical concurrent programming [7, 10]. An interesting area for future work is a formulation of session types suitable for Promela, to allow static checking of efficient protocols where channel fields are used with multiple message types in a structured way.

### 3. A Type System with Complete Type Information

Since no standard formal semantics for Promela is available, we present our type checking algorithm, and prove its correctness, with respect to Promela-calculus (PC), a small language based on the pi calculus. PC captures the features of Promela in which we are primarily interested: channel-based communication with support for first-class channels, recursive channel types, and subtyping between base types, and omits orthogonal language features which would add to the complexity of our proofs without providing further insight into the problems we are interested in solving. Note that our implementation, ETCH, supports the Promela language in full.

In this section we present a version of PC in which channel types are fully specified. We introduce a standard type system for this language, and state properties of interest: a type preservation theorem, and a result showing that execution of a well-typed specification cannot result in communication errors.

### 3.1. Syntax

The top-level syntax for PC is given in Figure 7 and an example specification presented in Figure 8.

---

<table>
<thead>
<tr>
<th>Values</th>
<th>v ::= true</th>
<th>false</th>
<th>numeric literal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Types T</td>
<td>bool</td>
<td>bit</td>
<td>byte</td>
</tr>
<tr>
<td></td>
<td>chan{T, ..., T}</td>
<td>channel (with product type for message fields)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>type variable</td>
<td></td>
</tr>
<tr>
<td></td>
<td>µX.T</td>
<td>recursive type</td>
<td></td>
</tr>
<tr>
<td>Expressions e ::= v</td>
<td>literal</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>x</td>
<td>variable</td>
<td></td>
</tr>
<tr>
<td></td>
<td>x = e</td>
<td>assignment</td>
<td></td>
</tr>
<tr>
<td></td>
<td>e → e : e</td>
<td>conditional</td>
<td></td>
</tr>
<tr>
<td></td>
<td>e == e</td>
<td>equality</td>
<td></td>
</tr>
<tr>
<td></td>
<td>x!e, ..., e</td>
<td>send</td>
<td></td>
</tr>
<tr>
<td></td>
<td>x?x, ..., x</td>
<td>receive</td>
<td></td>
</tr>
<tr>
<td></td>
<td>e; e</td>
<td>sequence</td>
<td></td>
</tr>
<tr>
<td></td>
<td>T x; e</td>
<td>declaration</td>
<td></td>
</tr>
<tr>
<td></td>
<td>run P(e, ..., e)</td>
<td>process instantiation</td>
<td></td>
</tr>
<tr>
<td>Definitions D ::= proctype P(T x; ...; T x) { e }</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Specification S ::= D ... D e</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

Figure 7: Top-level syntax for Promela-calculus with complete type information (PC_{full}).
A PC specification consists of a series of proctype declarations, followed by an expression, which can be thought of as a main process. A simple expression is a literal value (v), variable reference (x), or receive operation (x?x,...,x). Forms of compound expression are assignment (x=e), conditional (e→e:e), comparison (e=e), send (x!e,...,e), process instantiation (run P(e,...,e)) and sequence (e;e). A proctype is a named, parameterised process definition. Many instances of a proctype can be created to run in parallel via run expressions.

We use A to denote the set of PC base types: A = {bit, byte, short, int, bool}. The values of type bool are true and false as usual, and the values of bit are 0 and 1. We do not specify values for the other types, we merely require that the literals inhabiting the numeric types satisfy the following containment relation: bit ⊆ byte ⊆ short ⊆ int. Channel types are constructed by specifying an ordered list of field types. We follow the Promela convention of using curly braces to specify the list of field types. The language also provides recursive types, to cater for the sort of channel-based communication discussed in Section 2. We require that recursive types are contractive and types in declarations are closed.

A simple example PC specification is shown in Figure 8. The main process launches an instance of proctype P with a channel and an integer request. The instantiated proctype uses the provided channel to receive a channel through which to send a response. If the integer request is the expected value of 10000 then a value of 20000 is sent back via the response channel, otherwise the value 0 is sent. The main process waits for a response from the instance of P, and terminates with the value received.

3.2. Semantics and internal syntax

An operational semantics for Promela-calculus is provided as Appendix A. Figure 9 presents necessary syntax additional to that of Figure 7 used to describe the progress of a Promela calculus specification as it is evaluated using the operational semantics. The semantics track variables via store, σ which maps locations to values; store locations l are another form of expression. We use an undefined literal value ⊥ for variables which have been defined but not assigned, and a set of channel literals c to represent channel identifiers to which channel variables may refer. A configuration K is a parallel composition of threads, where each thread has the form ⟨e⟩. The expression e defines the behaviour of a given thread, and the parallel composition of threads e1 and e2 is denoted ⟨e1⟩∥⟨e2⟩. Combining a configuration K with a store σ yields a global configuration ⟨σ,K⟩, which corresponds to the notion of parallel threads sharing memory. Syntax for evaluation contexts [36] allow a concise presentation of the operational semantics.

Figure 9 also presents syntax used by the type systems of Sections 3.3 and 4. The Unit type, with domain {() } caters for expressions which do not yield numeric or boolean values. In Section 4 types Bot and Top are used to place bounds on type variables, and stand-alone product types are used to represent message field types in isolation. Syntax is also provided for environments, which bind variable names, store locations and channel literals to types, and for bounds, which associated upper and lower bound types with type variables and are used extensively in Section 4.

---

3In C-like programming languages, the notation e ? e : e is used for conditional expressions. Promela uses → in place of ? to avoid parsing conflicts, due to the use of ? as the receive operator.
Values
v ::= ... | c channel literal | () unit literal | ⊥ undefined value

Types
T ::= ... | {T, ..., T} product | Unit unit type | Bot bottom type | Top top type

Expressions
e ::= ... | l store location

Store
σ map from locations l to values v

Configurations
K ::= ⟨e⟩ thread | K || K parallel

Global configurations
G ::= (σ, K)

Evaluation contexts
E ::= [] | E → e : e | E => E | v => E | E! e, ..., e | E? x, ..., x | v! v, ..., v, E, e, ..., e | E: e | run P(v, ..., v, E, e, ..., e)

Environment
Γ ::= ∅ | Γ, x : T | Γ, l : T | Γ, c : T

Bounds
B map from type variables X to lower/upper bound type pairs (L, U)

Figure 9: Internal syntax for Promela-calculus.

3.3. Type system

A type system for PC is presented in Figure 10. Typing judgements are with respect to both an environment Γ (as standard) and a bounds function B, which assigns free type variables to lower and upper type bounds. This bounds function will be used extensively in Section 4 when we consider constraint-based typing, and until then can be ignored.

The T-BOOL-LIT, T-NUM-LIT and T-UNIT-LIT rules give types to literal values in the obvious way. The undefined literal value ⊥ may be assigned to any store location, so we need to be able to give this literal value any type: this requirement is satisfied by rule T-UNDEFINED. Rules T-VAR, T-LOCATION, T-CHANNEL allow us to look up the type of a variable, store location or channel literal based on the type recorded for the entity in the environment.

The typing rules T-ASSIGN, T-COND, T-EQ, T-SEND and T-RECEIVE formally specify requirements of PC operations, e.g. that the number of argument expressions sent on a channel must match the arity of the channel, and that the type of argument i must match the type of the i-th element in the product type associated with the channel. Our type system assumes that recursive channel types are unfolded sufficiently so that their outermost type constructor is chan. Sequences of expressions are handled by the rule T-SEQ, which states that the type of a typable sequence is the type of the last expression in the sequence. Declarations are checked by rule T-DECL, which requires that the type of the declared variable is not already in use and the remainder of the expression is typable under the environment extended with the new variable. Type Checking of proctype instantiations via run statements is achieved via the T-RUN. This rule works by using the OK-PROCTYPE rule to type check the associated proctype in the current environment, thus a single proctype declaration is type checked separately each time it is instantiated by a run statement.

The subtyping relation is presented by rules with conclusions of the form T <: U. Rules S-LOWER and S-UPPER allow subtyping information to be obtained for type variables based on the bounds function B. Rules S-BOT and S-TOP mean that Bot and Top can be used as default lower and upper bounds for type variables.
Figure 10: Type System for Promela-calculus.
The remainder of the typing rules are used to check that threads and parallel compositions of threads are well-typed, and that the store associated with a thread is well-formed. Rule T-\text{SPEC} allows us to assign a type to a specification based on the type of the main expression for the specification, if this expression is typable.

By standard techniques \cite{26} we can prove that typability is preserved by reductions in the operational semantics:

\textbf{Theorem 3.1} If $\emptyset; \Gamma \vdash (\sigma, K) \text{ OK}$ and $(\sigma, K) \rightarrow (\sigma', K')$ then there exists $\Gamma' \supseteq \Gamma$ such that $\emptyset; \Gamma' \vdash (\sigma', K') \text{ OK}$.

The syntax of types in Figure 7 and the restrictions on well-formed types discussed in Section 3.1 mean that channel types are fully specified in a PC specification. As a result, the typing rules of Figure 10 are sufficient to enable checking of the correctness of channel operations directly. In particular, the rules ensure that the correct number of arguments are supplied to send and receive operations, and that these arguments have appropriate types. This leads to the following theorem, also proved using standard methods:

\textbf{Theorem 3.2} If $\emptyset; \Gamma \vdash (\sigma, [E[c? l_1, \ldots, l_m]] || [E'[c! v_1, \ldots, v_n]] || K) \text{ OK}$ then $m = n$ and $\emptyset; \Gamma \vdash v_i : \Gamma(l_i)$ for all $i \in \{1, \ldots, n\}$.

Theorems 3.1 and 3.2 together show that execution of a well-typed PC specification cannot lead to a communication error.

\section{Checking Under-specified Channel Types using Constraint-based Type Inference}

We now introduce a variant of Promela-calculus where, as in Promela, channel types are only partially specified. We show that full type checking is still possible, despite the lack of channel type information, using type reconstruction techniques. We adapt a standard constraint-based type inference methods, following the presentation given in \cite[Chapter 22]{26}. Our approach consists of two parts:

- \textit{A constraint typing relation}, defined by a set of syntax-directed typing rules. These rules give rise to a type checking algorithm that never fails: given a Promela-calculus specification, the algorithm generates a set of equality and subtyping constraints over type variables.

- \textit{A unification algorithm}, which processes the set of constraints generated by the constraint typing relation and attempts to find a solution—a substitution of type variables for concrete types which essentially “fills in” the missing types from the Promela-calculus specification such that it is typable in the standard type system of Section 3. If the constraints are unsatisfiable, the unification algorithm fails, meaning that the Promela-calculus specification is not typable in the system of Section 3 no matter how the missing type information is filled in.

The standard approach of \cite[Chapter 22]{26} applies to the simply-typed lambda calculus with integers and booleans, \textit{without} subtyping. In general, combining subtyping with type reconstruction is complicated. However, in the scenario where we only have subtyping between base types we show that type reconstruction is possible.

We shall henceforth refer to Promela-calculus with complete type information (see Section 3) as PC\textsubscript{full}, and Promela-calculus with partial type information (defined below) as PC\textsubscript{partial}, unless the variant of the calculus referred to is clear from the context.

The grammar for PC\textsubscript{partial} is the same as for PC\textsubscript{full} (see Figure 7), except for the syntax of types, which is presented for PC\textsubscript{partial} in Figure 11. The PC\textsubscript{partial} type syntax does not allow explicit specification of recursive types, and does not allow field types for channels to be specified beyond one level of nesting. A free type variable is
provided for the fields of each channel declaration when left unspecified. This syntax reflects the situation in Promela, where channel types cannot be specified beyond one level of nesting.

A $PC_{\text{full}}$ specification can be turned into a $PC_{\text{partial}}$ specification by unfolding all recursive types until any recursive type constructors appear deeper than one level of nesting, then replacing every channel field specifier deeper than one level of nesting with a distinct type variable. For example, unfolding the $PC_{\text{full}}$ type $\mu X.\text{chan}\{\text{int, } X\}$ gives $\text{chan}\{\text{int, chan}\{\text{int, }\mu X.\text{chan}\{\text{int, } X\}\}}$. Replacing the inner field specifier with a fresh type variable, $Y$ say, yields the $PC_{\text{partial}}$ type $\text{chan}\{\text{int, chan } Y\}$.

When analysing and reasoning about $PC_{\text{partial}}$ specifications we use the full language of types presented in Figure 7 augmented with the internal type syntax of Figure 9.

For ease of comparison, throughout this section we annotate a definition or result with (cf. [26, Chapter 22]) to indicate that it is analogous to definition or result $n$ in [26, Chapter 22].

4.1. Bounded substitutions

Recall from Section 3.1 that $\mathcal{A}$ denotes the set of Promela-calculus base types. We use $\mathcal{A}^*$ to denote $\mathcal{A} \cup \{\text{Bot, Top}\}$. Let $T$ denote the set of all Promela-calculus types and $TV$ the set of all type variable names.

Definition 4.1 (cf. [26, 22.1.1]) A bounded type substitution (or just bounded substitution) is a pair $(B, \sigma)$ where $B$ is a partial function from $TV$ to $\mathcal{A}^* \times \mathcal{A}^*$ and $\sigma$ is a partial function from $TV$ to $T$, satisfying:

1. If $B(X) = (L_X, U_X)$ then $\emptyset \vdash L_X \triangleleft U_X$, $L_X \neq \text{Top}$ and $U_X \neq \text{Bot}$
2. $\text{dom}(B) \cap \text{dom}(\sigma) = \emptyset$
3. $FV(\text{range}(\sigma)) \subseteq \text{dom}(B)$.

For a type variable $X$ and a bounded substitution $(B, \sigma)$ there are four possible cases:

1. $\sigma(X)$ is defined, say $\sigma(X) = T$, and $T$ contains no type variables. In this case $X$ is substituted for a concrete type.
2. $\sigma(X)$ is defined, say $\sigma(X) = T$, and $T$ contains type variables $X_1, \ldots, X_n$ such that $B(X_i)$ is defined for each $X_i$. In this case, $X$ is substituted for an abstract type, parameterised by pairs of allowable bounds.
3. $\sigma(X)$ is undefined, but $B(X)$ is defined as $(L_X, U_X)$. In this case, $X$ can be any type $T \in \mathcal{A}$ lying between $L_X$ and $U_X$. (Condition 1 of Definition 4.1 ensures that we do not have $L_X = U_X = \text{Bot}$ or $L_X = U_X = \text{Top}$.)
4. $\sigma(X)$ and $B(X)$ are both undefined. In this case $X$ can be any type.

We can obtain a concrete substitution for every type variable by composing $\sigma$ with a new substitution $\gamma$ which assigns each type variable $X$ satisfying case 3 above to a type $T \in \mathcal{A}$ between $L_X$ and $U_X$, and assigns each type variable satisfying case 4 above to any concrete type. In practice, we consider bounded substitutions where $B$ supplies bounds for every type variable of interest, so that case 4 does not apply.

4.2. Constraint-based typing

Definition 4.2 (cf. [26, 22.3.1]) A constraint set $\mathcal{C}$ is a set of equations $\{T_i \bowtie_i U_i \mid i \in 1..n\}$, where $\bowtie_i \in \{=, \lt\}$ $(1 \leq i \leq n)$. A bounded substitution $(B, \sigma)$ is said to unify (or satisfy) equation $T = U$ if the substitution instances $\sigma(T)$ and $\sigma(U)$ are identical (up to equivalence of recursive types). The substitution $(B, \sigma)$ is said to unify (or satisfy) equation $T \lt U$ if $B \vdash \sigma(T) \lt \sigma(U)$. We say that $(B, \sigma)$ is a unifier for $\mathcal{C}$ if it unifies every equation in $\mathcal{C}$, in which case we also say that $(B, \sigma)$ satisfies $\mathcal{C}$.
\[\Gamma \vdash e : T \mid_k C\]

\(C\) is a set of equality and subtyping constraints, \(\chi\) is a set of type variables

\(b\) is a boolean literal
\[\Gamma \vdash b : \text{bool} \mid_0 \{\}\] (CT-BOOL-LIT)

\(T_n\) is its type
\[R; \Gamma \vdash n : T_n \mid_0 \{\}\] (CT-NUM-LIT)

\(x \notin \text{dom}(\Gamma)\)
\(\chi_1\) is a type variable in \(T\)
\[\Gamma \vdash T ; x : T \vdash e : U \mid_{\chi_1 \cup U \chi_2} C\] (CT-DECL)

\(C\)'s are different in \(T\)
\[\Gamma \vdash C' \cup \{U : T\}\] (CT-VAR)

\(\Gamma \vdash x = e : T \mid_{\chi} C'\) (CT-ASSIGN)

\(\Gamma \vdash e_1 : T_1 \mid_{\chi_1} C_1\)
\[\Gamma \vdash e_2 : T_2 \mid_{\chi_2} C_2\] (CT-COND-BASE)

\(T_2 \notin A^*\) or \(T_3 \notin A^*\)
\[C' = C_1 \cup C_2 \cup \{T_1 : \text{bool}\} \cup \{T_2 = T_3\}\] (CT-COND-COMPONENT)

\(\chi_1 \neq \chi_2, T_1, T_2, C_1, C_2, \Gamma, e_1, e_2\)
\[x : \Gamma \vdash e_1 = e_2 : \text{bool} \mid_{\chi_1 \cup \chi_2 \cup (X)} C''\] (CT-EQ)

\(X \neq X_1, X_2, T_1, T_2, C_1, C_2, \Gamma, e_1, e_2\)
\[x : \Gamma \vdash \text{all_unif} \mid_{\chi_1 \cup \chi_2 \cup \{X_1 \rightarrow \chi_1, X_2 \rightarrow \chi_2\}} C'\] (CT-SEND)

\(\chi_i \neq X_i \neq \Gamma (1 \leq i \neq j \leq n)\)
\[\Gamma \vdash x_1 : T_1, \ldots, x_n : T_n \in \Gamma\]
\[X_i \neq \text{chan}[X_1, \ldots, X_n], X_i : T_i, X_i \leq \text{chan}[X_1, \ldots, X_n] \leq T_i\] (CT-RECEIVE)

\(\chi_1 \neq \chi_2, T_1, T_2, C_1, C_2, \Gamma, e_1, e_2\)
\[\Gamma \vdash e_1 : T_1 \mid_{\chi_1} C_1\]
\[\Gamma \vdash e_2 : T_2 \mid_{\chi_2} C_2\] (CT-SEQ)

\(\Gamma \vdash \text{run} \mid_{\chi_1 \cup \chi_2 \cup \{U_1 \rightarrow \chi_1, \ldots, U_n \rightarrow \chi_2\}} C''\) (CT-RUN)

\(\Gamma \vdash \text{D} \mid_{\chi} C\)
\[\Gamma \vdash \text{proctype} \mid_{\chi_1 \cup \chi_2 \cup \{U_1 \rightarrow \chi_1, \ldots, U_n \rightarrow \chi_2\}} C''\] (C-OK-PROCTYPE)

\(\forall i \in (1, \ldots, n), (\Gamma \vdash D_i \mid_{\chi_i} C_i)\)
\[\chi_1, \ldots, \chi_n\] are pairwise disjoint
\[\Gamma \vdash e : T \mid_{\chi} C\] (CT-SPEC)
4.2.1. Constraint typing relation

The constraint typing relation is defined by the rules of Figure 12. The notation for presenting constraints is adapted from [26], with the addition of subtyping constraints. We read $\Gamma \vdash e : T \mid _\chi C$ as “expression $e$ has type $T$ under assumptions $\Gamma$ whenever constraints $C$ are satisfied” [26]. The other forms of rule are interpreted similarly. The subscripts $\chi$ are used to track type variables appearing in sub-derivations, to ensure that distinct sub-derivations do not share type variable names.

The rules of Figure 12 are syntax-directed. Our type checker employs these rules as an algorithm that computes a type $T$, constraint set $C$ and set of type variables $\chi$ given an environment $\Gamma$ and $PC_{\text{partial}}$ specification $D_1 \ldots D_n$ $e$. We say that the constraint set $C$ is generated by the constraint typing relation. In Section 4.3, we present a unification algorithm which computes the most general unifier for a constraint set, if any unifier exists.

The constraint typing relation has two important properties on which our unification algorithm relies:

1. In any subtyping constraint of the form $T_1 <: T_2$ generated by the constraint typing relation, at least one of $T_1, T_2$ is not a type variable.

2. The rules never add to $\Gamma$ an assumption of the form $x : X$, where $X$ is a type variable.

We state property 1 formally as Lemma 4.1 below; property 2 is established as part of the proof of Lemma 4.1.

Since subtyping is always defined between base types, $Bot$ and $Top$, it is clear that $(\mathcal{A}^*, <:) \text{ forms a complete lattice. Thus the join } \lor \text{ and meet } \land \text{ operators are well-defined on } \mathcal{A}^*$. Rule CT-COND-BASE uses the join operator; and both operators are used when we present our unification algorithm in Section 4.3.

We discuss some of the more interesting rules in Figure 12.

Declaration of variables. Rule CT-DECL ensures that all type variables appearing in type $T$ associated with a declaration are distinct. The expression following the declaration is checked in an extended environment, and the type variables appearing in $T$ form part of the set of used type variables in the conclusion of the rule.

Conditional expressions. In a conditional expression of the form $e_1 \rightarrow e_2 : e_3$ we know that $e_1$ must have type bool, thus in both rules we post the constraint $T_1 <: \text{bool}$, where $T_1$ is the type computed for $e_1$ by the constraint typing relation. Intuitively, it makes sense to give the result type for the conditional expression the smallest type of which both types $T_2$ and $T_3$ are subtypes, where $T_2$ and $T_3$ are computed for $e_2$ and $e_3$ respectively. This is the join of $T_2$ and $T_3$, denoted $T_2 \lor T_3$. In general, we cannot compute the join while type checking, since $T_2$ and $T_3$ may involve free type variables. To overcome this, we use two rules to handle conditionals.

Rule CT-COND-BASE assumes that $T_2$ and $T_3$ are both base types, in which case $T_2 \lor T_3$ can be computed directly (allowing the case where $T_2 \lor T_3 = \text{Top}$, e.g. when $T_2 = \text{bool}$ and $T_3 = \text{int}$).

Rule CT-COND-COMPOUND assumes that at least one of $T_2$ and $T_3$ is not a base type. In this case we will show that the non base types must in fact be channel types, in which case it is sufficient to post the constraint $T_2 = T_3$ and give the conditional expression type $T_2$.

Send, receive and run operations. Rule CT-SEND in Figure 12 is analogous to the standard typing rule for send operations, rule T-SEND in Figure 10. The difference is that while T-SEND applies only if the arguments supplied for a send operation are subtypes of the field types for the associated channels, rule CT-SEND always applies, using distinct, fresh type variables to post constraints specifying this subtyping property.

Rules CT-RECEIVE and CT-RUN handle receive and run operations in a similar manner.

4.2.2. Relationship between constraint typing relation and standard type system

Given a $PC_{\text{partial}}$ specification $S$ and a $PC_{\text{full}}$ type $T$, if we can find a bounded substitution which, when applied to $S$, turns $S$ into a $PC_{\text{full}}$ specification with type $T$, then since the substitution does not affect the semantics of $S$, Theorem 3.2 tells us that $S$ will be free of communication errors.

This is the declarative characterization of possible solutions for a specification and a type.
Definition 4.3 (cf. [26] 22.2.1)

1. Let \( \Gamma \) be a context and \( e \) an expression. A solution for \((\Gamma, e)\) is a tuple \((B, \sigma, T)\) such that \((B, \sigma)\) is a bounded substitution and \(B; \sigma(\Gamma) \vdash \sigma(e) : T\).

2. Let \( \Gamma \) be a context and \( D \) a declaration. A solution for \((\Gamma, D)\) is a pair \((B, \sigma)\) such that \((B, \sigma)\) is a bounded substitution and \(B; \sigma(\Gamma) \vdash \sigma(D) \, OK\).

3. Let \( \Gamma \) be a context and \( S \) a specification. A solution for \((\Gamma, S)\) is a tuple \((B, \sigma, T)\) such that \((B, \sigma)\) is a bounded substitution and \(B; \sigma(\Gamma) \vdash \sigma(S) : T\).

The declarative characterization does not give a way of finding solutions. However, given a \( PC_{\text{partial}} \) specification \( S \), the constraint typing relation provides a type \( U \) and a set of constraints \( C \), where in general \( U \) and \( C \) may share type variables. If a unifier can be computed for \( C \), i.e., a bounded substitution \((B, \sigma)\) satisfying the conditions of Definition 4.2, then the unifier can be applied to \( U \) to provide a concrete type \( T \) for the specification.

Definition 4.4 (cf. [26] 22.3.4)

1. Let \( \Gamma \) be a context and \( e \) an expression. Suppose that \( \Gamma \vdash e : U \, |\chi \, C \). A solution for \((\Gamma, e, U, C)\) is a tuple \((B, \sigma, T)\) such that \((B, \sigma)\) is a bounded substitution and \((B, \sigma)\) satisfies \(C\) and \(\sigma(U) = T\).

2. Let \( \Gamma \) be a context and \( D \) a declaration. Suppose that \( \Gamma \vdash D \, OK \, |\chi \, C \). A solution for \((\Gamma, D, C)\) is a pair \((B, \sigma)\) such that \((B, \sigma)\) is a bounded substitution and \((B, \sigma)\) satisfies \(C\).

3. Let \( \Gamma \) be a context and \( S \) a specification. Suppose that \( \Gamma \vdash S : U \, |\chi \, C \). A solution for \((\Gamma, S, U, C)\) is a tuple \((B, \sigma, T)\) such that \((B, \sigma)\) is a bounded substitution and \((B, \sigma)\) satisfies \(C\) and \(\sigma(U) = T\).

This algorithmic characterization does allow us to find solutions, since we can apply the constraint typing relation algorithmically, and solve constraints using a unification algorithm presented in Section 4.3.

We show that the declarative and algorithmic characterizations are equivalent by showing that every solution for \((\Gamma, S, U, C)\) is also a solution for \((\Gamma, S)\), and that every solution for \((\Gamma, S)\) can be extended to a solution for \((\Gamma, S, U, C)\).

Theorem 4.1 (Soundness) (cf. [26] 22.3.5)

1. Let \( \Gamma \) be a context and \( e \) an expression, and suppose \( \Gamma \vdash e : U \, |\chi \, C \). If \((B, \sigma, T)\) is a solution for \((\Gamma, e, U, C)\) then it is also a solution for \((\Gamma, e)\).

2. Let \( \Gamma \) be a context and \( D \) a declaration, and suppose \( \Gamma \vdash D \, OK \, |\chi \, C \). If \((B, \sigma)\) is a solution for \((\Gamma, D, C)\) then it is also a solution for \((\Gamma, D)\).

3. Let \( \Gamma \) be a context and \( S \) a specification, and suppose \( \Gamma \vdash S : U \, |\chi \, C \). If \((B, \sigma, T)\) is a solution for \((\Gamma, S, U, C)\) then it is also a solution for \((\Gamma, S)\).

Definition 4.5 (cf. [26] 22.3.6) \( \sigma \setminus \chi \) is the substitution that is undefined for all the variables in \( \chi \) and otherwise behaves like \( \sigma \).

Theorem 4.2 (Completeness) (cf. [26] 22.3.7)

1. Let \( \Gamma \) be a context and \( e \) an expression, and suppose \( \Gamma \vdash e : U \, |\chi \, C \). If \((B, \sigma, T)\) is a solution for \((\Gamma, e)\) and \(\text{dom}(\sigma) \cap \chi = \emptyset\) then there is some solution \((B, \sigma', T')\) for \((\Gamma, e, U, C)\) such that \(\sigma' \setminus \chi = \sigma\) and \(\emptyset \vdash T' : < : T\).

2. Let \( \Gamma \) be a context and \( D \) a declaration, and suppose \( \Gamma \vdash D \, OK \, |\chi \, C \). If \((B, \sigma)\) is a solution for \((\Gamma, D)\) and \(\text{dom}(\sigma) \cap \chi = \emptyset\) then there is some solution \((B, \sigma')\) for \((\Gamma, D, C)\) such that \(\sigma' \setminus \chi = \sigma\).

3. Let \( \Gamma \) be a context and \( S \) a specification, and suppose \( \Gamma \vdash S : U \, |\chi \, C \). If \((B, \sigma, T)\) is a solution for \((\Gamma, S)\) and \(\text{dom}(\sigma) \cap \chi = \emptyset\) then there is some solution \((B, \sigma', T')\) for \((\Gamma, S, U, C)\) such that \(\sigma' \setminus \chi = \sigma\) and \(\emptyset \vdash T' : < : T\).

The previous two theorems establish the following:
Corollary 4.1 \textit{(cf. [26] 22.3.8)} Suppose \( \Gamma \vdash S : U \mid X. C \). There is a solution for \( (\Gamma, S) \) if and only if there is a solution for \( (\Gamma, S, U, C) \).

Thus if we can find solutions to the constraints generated by the constraint typing relation for a specification \( S \) then we have established that execution of \( S \) will not lead to communication errors.

4.3. Unifying constraints

We now turn to the problem of solving a set of subtyping and equality constraints. The basic idea is the Hindley-Milner [15, 21] approach of using unification [30] to find the most general solution. In general this is complicated by the presence of subtyping, but our constraint-based typing system for Promela has the property that every subtyping constraint contains at most one type variable. We can exploit this property to define an extension of the standard unification algorithm as presented by Pierce [26, Chapter 22]. The algorithm is given a set of constraints, generated by the constraint typing relation, and either fails or returns the most general bounded substitution which unifies the constraints. In Section 6 we discuss alternative approaches suggested by the literature on type inference with subtyping.

While it would be sufficient, for type safety, merely to show that the set of constraints is satisfiable, obtaining a concrete substitution allows us to present reconstructed types to the user, which aids understanding of Promela specifications. While computing the most general unifier is not strictly necessary (since Promela does not support polymorphic processes), we argue in Section 5.1 that type checking of a sensibly constructed Promela specification should yield a most general unifier where each type variable is assigned to a concrete type; a most general unifier without this property signals a potential flaw in the specification, to which the programmer can be alerted.

4.3.1. The unification algorithm

Our algorithm is presented in Figure 13. The top-level function \textit{unify} is mutually recursive with the functions \textit{unify\_subtype} and \textit{unify\_equality}.

The parameters of \textit{unify} are a set \( C \) of constraints and a bounds function \( B \), such that \( FV(C) \subseteq dom(B) \). For a bounds function \( B \) and a set \( X \subseteq dom(B) \) of type variables we use \( B \setminus X \) to denote the bounds function \( B' \) identical to \( B \) except that \( B'(X) \) is undefined for each \( X \in X \). For \( X \notin dom(B) \) we use \( B + [X \mapsto T] \) to denote the bounds function \( B' \) identical to \( B \) except that \( X \in dom(B') \) and \( B'(X) = T \).

If \( C \) is empty then \textit{unify} returns the bounded substitution \( (B, [ ]) \), consisting of the given bounds function and an empty substitution. Otherwise, constraints from \( C \) are processed one at a time, by calling \textit{unify\_subtype} or \textit{unify\_equality} as appropriate. The order of processing constraints does not matter.

Within \textit{unify\_subtype} there are several cases. If the subtyping constraint involves two base types, we simply check that the constraint is satisfied. The next case is for a constraint \( X <: T \), where \( T \) is a base type. In this case we adjust the bounds of \( X \) by lowering the upper bound so that it is at most \( T \), first checking that the desired new upper bound is not Bot, and solve the remaining constraint under this tighter bounds function. The next case is symmetrical and deals with a constraint \( T <: X \) by increasing the lower bound of \( X \). Finally, a subtyping constraint involving a non-base type is converted into an equality constraint.

Note that the \textit{unify\_subtype} function does not consider constraints between type variables (such as \( X <: Y \)); these are never generated by the constraint typing system:

Lemma 4.1 Suppose \( \Gamma \vdash S : U \mid X. C \). If \( T_1 <: T_2 \in C \) then at least one of \( T_1, T_2 \) is not a type variable.

The function \textit{unify\_equality} is similar to the standard unification algorithm, except that bounds must be checked and recomputed when solving constraints involving type variables.

For an equality constraint \( T_1 = T_2 \), if \( T_1 \) and \( T_2 \) are equal then the constraint is satisfied and can be discarded. The equality check must take into account the possibility that the types are recursive; we discuss how this is handled by our practical implementation in Section 5.2.1.

If \( T_1 \) and \( T_2 \) are both type variables, \( X \) and \( Y \) say, we check that the bounds for \( X \) and \( Y \) are compatible. If so, \( Y \) is substituted for \( X \), and the bounds of \( X \) are contracted with respect to the bounds for \( Y \).

If \( T_1 \) is a type variable \( X \) and \( T_2 \) is not a type variable then there are two cases, according to whether or not \( T_2 \) contains \( X \) as a free type variable. If so, then a recursive type is constructed and substituted for \( X \); if not, then \( T_2 \) is
substituted for $X$. In both cases we check that $T$ is within the bounds for $X$. The case where $T_2$ is a type variable and $T_1$ is not is handled symmetrically.

Two cases remain: a constraint of the form $\text{chan } U_1 = \text{chan } U_2$ is replaced with the constraint $U_1 = U_2$; a constraint of the form $\{U_1, \ldots, U_m\} = \{U'_1, \ldots, U'_n\}$, when $m = n$, is replaced with $m$ constraints, $U_i = U'_i$ ($1 \leq i \leq m$).

If the equality constraint does not correspond to one of these forms then unification fails.

If the top-level call $\text{unify}(T_1, T_2, C, B)$ function succeeds in processing all subtyping and equality constraints without failing, then it returns a bounded substitution $(B, \sigma)$ such that $\text{dom}(B) \subseteq \text{dom}(\sigma)$. It is possible for $B$ to be empty, if solving the constraints requires substituting for all of the type variables.

Figure 13: The unification function.

4.3.2. Properties of the unification algorithm

We will show that our unification algorithm can be applied in a way which ensures computation of the most general unifier for a set of constraints, if any unifier exists. To formalise the notion of “most general unifier”, we define an order on bounded substitutions.

**Definition 4.6** Let $(B, \sigma)$ be a bounded substitution. An instance of $(B, \sigma)$ is a mapping $\theta$ from $\text{dom}(B) \cup \text{dom}(\sigma)$ to closed type expressions such that

1. for every $X \in \text{dom}(B)$ with $B(X) = (L_X, U_X)$, $L_X \ll X \ll U_X$
2. there exists a substitution $\gamma$ such that $\theta = \gamma \circ \sigma$ (viewing $\theta$ as a substitution).

**Definition 4.7** (cf. [26, 22.4.1]) A bounded substitution $(B_1, \sigma_1)$ is less specific (or more general) than a bounded substitution $(B_2, \sigma_2)$, written $(B_1, \sigma_1) \sqsubseteq (B_2, \sigma_2)$, if and only if $\text{dom}(B_1) \cup \text{dom}(\sigma_1) = \text{dom}(B_2) \cup \text{dom}(\sigma_2)$ and every instance of $(B_2, \sigma_2)$ is also an instance of $(B_1, \sigma_1)$.

We now show that the unification procedure of Figure 13 computes the most general unifier for a set of constraints, if any unifier exists.

**Lemma 4.2** Let $C$ be a set of constraints and let $B_0$ be a bounds function such that $\text{FV}(C) \subseteq \text{dom}(B_0)$. Then $\text{unify}(C, B_0)$ terminates, either by failing or by returning a result of the form $(B, \sigma)$.

**Theorem 4.3** (cf. [26, 22.4.5]) Let $C$ be a set of constraints and let $B_0$ be the bounds function defined by $\text{dom}(B_0) = \text{FV}(C)$ and $\forall X \in \text{dom}(B_0). B_0(X) = (\text{Bot}, \text{Top})$. If there exists a unifier for $C$ then $\text{unify}(C, B_0)$ is a unifier for $C$, and for every unifier $(B, \sigma)$ for $C$, $\text{unify}(C, B_0) \sqsubseteq (B, \sigma)$. That is, $\text{unify}(C, B_0)$ is the most general unifier for $C$.

We have proved that, for a $PC_{\text{partial}}$ specification $S$, if the constraints generated by the constraint typing relation of Figure 12 have a solution then $\text{unify}$ can be used to compute the most general solution. Combining this with Corollary 4.1 this solution can be applied to $S$ to provide a typing under the standard type system of Section 3. This means that, by Theorem 3.2 execution of $S$ does not lead to communication errors.

5. **Practical Issues**

We have proved the correctness of our type checking methods for Promela-calculus, which captures enough of the interesting and relevant features of Promela to provide a rigorous theoretical basis for our practical implementation, Etch. We now address some practical issues: we show that computing a most general unifier allows us to infer some interesting properties related to channel usage in a specification (Section 5.1), describe the way in which we deal with recursive types (Section 5.2), sketch our approach to handling conditional receive statements, which are not part of Promela-calculus (Section 5.3), and address the problem of providing reasonable error messages with type inference (Section 5.4).

5.1. **Using most general unifiers for channel usage analysis**

Consider the following simple Promela example:

```promela
chan A = [3] of {int}
chan B = [1] of {byte}
chan C = [1] of {byte}
chan D = [1] of {byte}

proctype P() {
    A!4;
    B!5;
}

proctype Q() {
    byte x;
    A?x;
    C?x;
}
```

Messages on channel $A$ have type byte, but we can observe that this message type is wider than necessary: the literal value 4 is regarded as a byte in Promela, and variable $x$ has type byte, thus it would be sufficient for channel $A$ to be declared as: `chan A = [3] of {byte}`. Since int and byte fields require 4 and 1 bytes of storage respectively, and the channel has capacity 3, this modification reduces the space requirements for this channel, *per state*, from 12 to 3 bytes. For a realistic model with tens of millions of states this space reduction would clearly be significant. Note
also that in the above specification no message is ever received on channel \( B \), no message is ever sent on \( C \), and no communication whatsoever is performed on \( D \). Such a situation is likely to be unusual, and may signal a mistake in the specification. Nevertheless, the specification is well typed.

Applying \( \text{TCH} \) to the above example, invoking an option to look for channel redundancy, automatically discovers these specification flaws. When invoked in this mode, \( \text{TCH} \) ignores channel initialisers in the input specification, treating the channels in the above example as if they were declared as follows:

\[
\begin{align*}
\text{chan} & \hspace{1em} A \\
\text{chan} & \hspace{1em} B \\
\text{chan} & \hspace{1em} C \\
\text{chan} & \hspace{1em} D
\end{align*}
\]

\( \text{TCH} \) then performs type reconstruction, and reports the following most general types for the channels:

\[
\begin{align*}
A & : \text{chan} \langle \text{byte} \rangle \\
B & : \text{chan} \langle \text{byte}; \text{int} \rangle \\
C & : \text{chan} \langle \text{byte}; \text{int} \rangle \\
D & : \text{chan} \langle \text{Z} \rangle
\end{align*}
\]

The output for \( A \) indicates that \( A \) could be safely declared with field type \( \text{byte} \) rather than \( \text{int} \). The output for \( B \) shows that \( B \)'s field type can be any supertype of \( \text{byte} \), but has no upper bound: this indicates that no messages are sent on \( B \); similarly the output indicates that no messages are received on \( C \). Finally, \( D \) is given type chan \( \text{Z} \), indicating that \( D \) is never used for communication (the arity of \( D \)'s message tuples is even left unspecified). While this additional information could also be obtained via dataflow analysis \( \text{II} \), it is an interesting by-product of computing a most general unifier.

5.2. Recursive types

Unlike Promela-calculus, Promela does not allow explicit declaration of recursive types, but recursive channel types can be implicitly introduced by channel usage as illustrated by the \text{client-server} and \text{telephone} examples of Section 5.1. Our constraint-based type checking algorithm may generate recursive types during unification (see Section 4.3).

Although the user never writes recursive type expressions, they may encounter them in error messages, or in the reconstructed type information which \( \text{TCH} \) generates. We describe the techniques \( \text{TCH} \) uses to compare recursive type expressions, and to minimise recursive types so that they are presented to the user in as palatable a form as possible.

5.2.1. Comparing recursive types

The \text{unify\_equality} procedure (Figure 13) in our unification algorithm (Section 4.3) takes types \( T_1 \) and \( T_2 \), and checks whether \( T_1 = T_2 \). This check for structural equivalence of types needs to handle the case where \( T_1 \) and/or \( T_2 \) are recursive.

To handle recursive types in a straightforward manner our uses techniques from a unification algorithm given in \( \text{II} \). Types are represented as directed graphs with a cyclic graph corresponding to a recursive type. Each graph node \( n \) has a representative \( \text{rep}(n) \), a pointer to another graph node. Initially each node \( n \) represents itself, \textit{i.e.} \( \text{rep}(n) = n \).

During unification, representatives are re-assigned. If unification succeeds then the substitution part of the bounded substitution \( (B, \sigma) \) is the mapping defined by, for each type variable \( X \), \( X \mapsto \text{rep}^* (X) \), where, for a node \( n \), \( \text{rep}^* (n) \) is computed by following \( \text{rep} \) pointers and returning the node \( m \) eventually reached for which \( \text{rep}(m) = m \).

Representatives are computed as follows. When two chan nodes are unified via a constraint \( \text{chan} \hspace{1em} U_1 = \text{chan} \hspace{1em} U_2 \), the representative for the node associated with the left hand side of the constraint is set to the representative for the right hand node. Product type nodes are handled similarly when solving constraints of the form \( \{U_1, \ldots U_m\} = \{U_1', \ldots, U_m'\} \). When two type variables are unified via a constraint \( X = Y \), \( \text{rep}(X) \) is set to \( \text{rep}(Y) \). When a type variable is unified with a non type variable, the representative for the type variable node is set to the representative for the non type variable node. The substitution of a type variable for a recursive type, specified as \( X \mapsto \mu X.F(Y) \) in Figure 13, is handled implicitly by introducing a cycle in the type graph: we do not use an explicit \( \mu \) constructor.

To check equality of types \( T_1 \) and \( T_2 \), it suffices for our algorithm to check whether \( \text{rep}^* (n_1) = \text{rep}^* (n_2) \), where \( n_1 \) and \( n_2 \) are the type graph nodes associated with \( T_1 \) and \( T_2 \) respectively. The types may not be identified as equal straight away: when we first compare \( T_1 \) with \( T_2 \) we may just set \( \text{rep}(n_1) \) to \( \text{rep}(n_2) \). The next time the types are
5.2.2. Recursive type minimisation

A recursive type can have infinitely many forms. Consider the type expression \( \text{rec} \, \text{X}. \, \text{chan} \{ \text{X}, \text{mtype} \} \) associated with the client-server specification of Section 2.1. Alternative ways to write this type include:

- \( \text{chan} \{ \text{rec} \, \text{X}. \, \text{chan} \{ \text{X}, \text{mtype} \}, \text{mtype} \} \)
- \( \text{chan} \, \text{rec} \, \text{X}. \{ \text{chan} \, \text{X}, \text{mtype} \} \)
- \( \text{chan} \{ \text{chan} \{ \text{chan} \{ \text{chan} \{ \text{chan} \, \text{rec} \, \text{X}. \, \text{chan} \{ \text{X}, \text{mtype} \}, \text{mtype} \}, \text{mtype} \}, \text{mtype} \}, \text{mtype} \}, \text{mtype} \} \)

One issue which motivates techniques for recursive type minimisation is that of storage: in a production compiler it is desirable to store type expressions as compactly as possible to avoid excessive memory overhead during compilation. Sophisticated minimisation techniques for recursive types have been developed for this purpose [5, 20]. Our motivation is to improve the readability of messages generated by \( \text{E} \text{TCH} \). The aim of an enhanced type checker is to assist with compile-time debugging, for which good quality error messages are essential.

To illustrate the improvement in output quality which type minimisation affords, consider the variables \( \text{pool} \) and \( \text{partner} \) from the client-server and telephone specifications respectively (see Sections 2.1 and 2.2). Figure 4 shows the complete types which \( \text{E} \text{TCH} \) reconstructs for these variables.

If we run \( \text{E} \text{TCH} \) without type minimisation then the reconstructed type information for these variables is presented as follows:

\[
\begin{align*}
\text{pool} & : \text{chan} \{ \text{chan} \, \text{rec} \, \text{X} . \{ \text{mtype}, \text{chan} \, \text{X} \} \\ 
\text{partner} & : \text{array(size 3) of chan rec} \, \text{X} . \{ \text{chan X}, \text{bit} \}
\end{align*}
\]

In each case, the displayed type expression is larger than need be. For a more extreme case where type minimisation can drastically improve readability, consider the contrived Promela example of Figure 14.

The code fragment is not well typed: the channel \( F \) accepts single field \( \text{byte} \) messages, but the channel \( E \) is supplied as an argument to \( F \) at line 14. \( \text{E} \text{TCH} \) reports this via a reasonable error message:

\[
\text{Error at line 14: "rec X.chan{X,X,rec Y.chan {Y}}" is not compatible with type "byte".}
\]

When minimisation is turned off the error message becomes much more difficult to understand:

\[
\text{Error at line 14: "rec Z.chan rec X.chan{chan x,chan x,chan rec Y,(chan(chan(chan Y))}, Z,chan rec A.(chan(chan(chan A))}" is not compatible with type "byte".}
\]

The \( \text{E} \text{TCH} \) recursive type minimisation algorithm is based on an algorithm for minimisation of deterministic finite automata (DFA) given in [19]. There are many algorithms for DFA minimisation in the literature (see [35] for a taxonomy). We settled on the algorithm of [19] for its simplicity and ease of correct implementation.
Figure 15: Typing rule illustrating the way conditional receive operations are handled by Etch. The rule shows an example of a variable argument and an eval argument. For ease of presentation, details of constraints and type variables for the other arguments are omitted.

5.3. Type Checking full Promela

Promela-calculus is deliberately much simpler than Promela, but from the perspective of implementing a type checker, most of the additional features of Promela can be handled easily. We briefly discuss one feature of full Promela which required some consideration when designing Etch: constraint typing for conditional receive statements.

A useful feature of Promela is the ability to conditionally receive messages on a channel. In the following example:

```
chan c = [0] of int;
...
byte x = ...;
c?eval(x)
```

the statement `c?eval(x)` blocks unless an integer message is offered on channel `c` such that the value of the message equals the runtime value of `x`. The conditional receive does not modify variables of the recipient; the only side-effect is that a message is removed from the channel, if buffered. Note that an arbitrary expression, not just a variable, can appear as an argument to eval.

Without the eval construct, we would deem the above example ill-typed, as it attempts to receive an int argument into a byte variable. However, with the eval construct it does not make sense to reject this example: the receive statement requires a value of type byte, and since the channel communicates int values, which contain all byte values, the statement will be executable if the correct int value is transmitted.

On the other hand, we could regard the following example as ill-typed:

```
chan c = [0] of byte;
...
int x = ...;
c?eval(x)
```

The receive statement requires that some value of type int be available on a channel which transmits values of the smaller type byte. Suppose that `x` has value -1 at runtime, which is an int but not a byte. Then the receive statement will never succeed, which could be viewed as a programming error.

Based on this discussion, we reverse the way in which subtyping constraints are posted for eval receive arguments. Recall that when the `i`-th argument to a receive operation is a variable with type `T_i`, we choose a fresh type-variable `X_i` for the argument type, and post the constraint `X_i <: T_i` (see rule CT-RECEIVE in Figure 12). When the `i`-th argument has the form eval(e), where `e` has type `T_i`, we again choose a fresh type variable `X_i`, but instead post the constraint `T_i <: X_i`. This captures the idea that a conditional receive construct should expect values belonging to a subset of the type assigned to the channel. Type rule CT-COND-RECEIVE in Figure 15 describes the way constraints are posted when both variable and eval arguments are used in a receive statement.

5.4. Providing reasonable error messages

Provision of informative error messages is an important feature of any practical type checker. Error reporting for standard type checking is relatively easy: a type error typically corresponds to a direct misuse of types, which can be isolated to a single line of source code. In theoretical terms, this corresponds to an expression for which no type rule applies. Error reporting with type inference is harder. The constraint typing relation does not fail: every expression in a program is successfully assigned a type, and type errors are reported if unification fails. It is difficult to relate a unification error to a specific line of source code, since the error may result from unification of several constraints, arising from distinct areas of the program. In the design of Etch, we use two simple techniques to provide reasonable error messages.

First, we do not implement the constraint typing relation strictly: when checking a compound expression, if concrete types are available for all sub-expressions then we apply standard type checking, logging a type error immediately if type checking fails. For example, if `x : bit and y : int`, on encountering the assignment `x = y`, rather
than posting constraint bit \(<\): \text{int} \) we immediately report a subtyping error, with a message tailored to the fact that the subtyping error occurs in the context of an assignment. Simply posting the constraint bit \(<\): \text{int} \) would lead to a subtyping error during unification, but without passing significant additional information to the unifier, we would not know the context in which the subtyping error had occurred, so could not give such a good error message.

Secondly, we associate with each constraint the source code line at which the constraint was generated. When a constraint causes unification to fail, this line number is used for error reporting. This is better than nothing: often a constraint corresponds to a direct type error at the point of constraint generation. However, if several incompatible constraints are generated, \(\text{e.g.}\) according to the way in which a channel is used, the specific constraint which causes unification to fail may result from \textit{correct} channel usage, if constraints arising from incorrect channel usage have been processed first. In this case, a type error will be generated referring to a seemingly correct line of source code, leaving the user to investigate the real cause of the error.

6. Related Work

The \textsc{Etch} type checker, together with an informal description of our approach to type checking, are described in a tool demonstration paper \[8\]. To our knowledge, there is no other work which concentrates directly on type checking for Promela. A paper introducing \textit{Promela+}, an extension of Promela for timed interactive simulation \[6\], describes a software tool which claims to provide better type checking than standard \textsc{Spin}, though no further details on the nature of this type checking are provided.

The problem of type inference in the presence of subtyping has been studied by several authors, including \[33, 27, 2, 9, 28, 23, 34, 14\]. We can observe that our approach, of finding the most general unifier as a bounded substitution, gives more than we really need for the specific problem of type checking Promela. It would be sufficient to solve the equality constraints in the standard way, apply the resulting substitution to the subtyping constraints, and then check satisfiability of the resulting subtyping constraints. Tiuryn \[33\] has shown that satisfiability can be decided in polynomial time, although his decision procedure does not construct an explicit solution of the constraints and does not reveal the bounds of each type variable in the way that our algorithm does. In practice, an explicit substitution allows us to present the modeller with reconstructed types for channels in a Promela specification, which aids program understanding. Furthermore, computing a most general substitution may reveal additional properties of interest related to channel usage, as discussed in Section \[5.1\].

The theory \textsc{HM(X)} \[23\] allows the Hindley-Milner system can be parameterised by a constraint system. The theory gives conditions for principal typings to exist, but does not automatically provide an algorithm for solving constraints. We believe that our system can be expressed as an instance of \textsc{HM(X)}, although we have not checked this in detail, but in any case we would still need to define an algorithm for solving subtyping constraints. Pottier \[27\] has also studied type inference with subtyping, focusing on efficient algorithms, although the algorithms themselves are not defined very explicitly. We have not made a detailed comparison with his work, as we found it easier to develop our own algorithm based on the specific properties of our constraint sets. However, in relation to efficiency, we can make two observations. First, our implementation uses the technique, crucial for making unification efficient, of representing type expressions by directed graphs. Second, Promela does not include let-polymorphism, which is the cause of potential inefficiency in ML-style type inference. In practice we have found our implementation to be acceptably efficient.

We note that the most general solutions produced by our algorithm could be used as the basis for a system of ML-style polymorphic typing for Promela, either to allow polymorphic process definitions or to support separate type checking of Promela code in multiple files. Our algorithm might be of interest for other languages with similar properties to Promela.

Sophisticated techniques have been developed for generating good error messages with type inference for functional languages: see \[13\] and references therein, as well as a recent tool for type error slicing in standard ML \[29, 11\]. Although we have found the simple methods for error reporting discussed in Section \[5.4\] to work reasonably well in practice, the usability of \textsc{Etch} could potentially be improved by implementing a more advanced system of error tracking.
7. Conclusions and Future Work

We have presented details of the design and implementation of ETCH, an enhanced type checker for Promela. ETCH is able to statically detect errors which are beyond the limited scope of type checking performed by the SPIN model checker, thus ETCH can aid the development of robust Promela specifications for formal verification. ETCH is publicly available for use by the Promela/SPIN community.

As discussed in Section [2,3], an avenue for future work is to extend ETCH with session types, to widen the class of specifications that can be successfully type checked. The main challenge here would be inferring session types from the way in which channels are used, so that ETCH can continue to be applied to Promela specifications without requiring additional type annotations.

Acknowledgements

Alastair F. Donaldson was supported by a PhD studentship from the Carnegie Trust for the Universities of Scotland, and subsequently by the grant “Advanced Formal Verification Techniques for Heterogeneous Multi-core Programming” (EP/G051100/1) from the UK Engineering and Physical Sciences Research Council (EPSRC). Simon Gay was partially supported by the grant “Engineering Foundations of Web Service: Theories and Tool Support” (EP/E065708/1) from the UK Engineering and Physical Sciences Research Council (EPSRC).

References


A. Operational semantics for Promela-calculus

Operational semantics for PC are presented in Figure 16. Rules of the form \((\sigma, e) \rightarrow_{\gamma} (\sigma', e')\) show how expressions are evaluated with respect to a store. The E-Lookup and E-Assign rules show respectively how the store \(\sigma\) is referenced to look up the value of a store location, and updated when a store location is written to. Evaluation of conditional expressions are handled naturally by rules E-True-Cond and E-False-Cond, while analogous rules E-1-Cond and E-0-Cond show how the bit literals 1 and 0 can be used in place of true and false. Rules E-True-Eq and E-False-Eq allow numeric values to be compared to yield a boolean result, and rule E-Seq shows how sequences of statements are evaluated. Declaration of variables with base types is handled by the E-Base-Type-Decl rule, which picks a fresh store location for each such declaration. When a variable is declared with a type \(T \notin \mathcal{A}\), rule E-Chan-Decl introduces a new channel value. We introduce type rules in Section 3.3 such that for any type \(T \in \mathcal{A}\) or \(T\) is a (possibly recursive) channel type.

The evaluation context syntax of Figure 9, together with the E-Context rule, allow more complex forms of expressions to be evaluated in terms of rules of the form \((\sigma, e) \rightarrow_{\gamma} (\sigma', e')\). For example, the pair of evaluation context \(E \vdash e_1, \ldots, e_n \land v \land \ldots, v, E, e_1, \ldots, e\) specify that the expression to the left of a \(!\) operator should be evaluated to a value before the expressions on the right are evaluated, and that these expressions should be evaluated in left-to-right order. The other forms of evaluation context are similar.

Semantics for communication are provided by rule E-Comms. The rule states that if a thread is ready to send \(n\) values on channel \(e\) and another thread is ready to receive into \(n\) store locations, also via \(e\), then the store should be updated so that each location \(l_i\) is set to the corresponding value \(v_i\) \((1 \leq i \leq n)\). The send and receive operations reduce to the \((\_\_\_\_\_\_)\) literal value of type Unit. Rule E-Run provides call-by-value semantics for proctype instantiation.

The remaining rules provide algebraic identities which allow parallel compositions of threads to be maneuvered in order for an evaluation rule to apply.

B. Proofs omitted from the text

For convenience, we re-state each result.

B.1. Proofs of results in Section 4.2

Theorem 4.1 (soundness), page 17

1. Let \(\Gamma\) be a context and \(e\) an expression, and suppose \(\Gamma \vdash e : U \mid \chi\ C\). If \((B, \sigma, T)\) is a solution for \((\Gamma, e, U, C)\) then it is also a solution for \((\Gamma, e)\).

2. Let \(\Gamma\) be a context and \(D\) a declaration, and suppose \(\Gamma \vdash D \text{ OK} \mid \chi\ C\). If \((B, \sigma)\) is a solution for \((\Gamma, D, C)\) then it also is a solution for \((\Gamma, D)\).

3. Let \(\Gamma\) be a context and \(S\) a specification, and suppose \(\Gamma \vdash S : U \mid \chi\ C\). If \((B, \sigma, T)\) is a solution for \((\Gamma, S, U, C)\) then it also is a solution for \((\Gamma, S)\).

Proof In each case, a straightforward induction on the typing derivation, with a case analysis on the last rule. We give details for the rules CT-Cond-Base and CT-Send in case 1.

(CT-Cond-Base): In this case \(e\) is \(e_1 \rightarrow e_2 : e_3\) and from the hypotheses of the instance of CT-Cond-Base we have \(\Gamma \vdash e_1 : T_1 \mid C_1, \Gamma \vdash e_2 : T_2 \mid C_2, \Gamma \vdash e_3 : T_3 \mid C_3\), and \(T_2, T_3 \in \mathcal{A}\), with \(U = T_2 \lor T_3\), and \(C = C_1 \cup C_2 \cup C_3 \cup \{T_1 <: \text{bool}\}\). We omit the \(\chi\) annotations as they are not used in this proof.

Because \((B, \sigma)\) is a solution for \((\Gamma, e_1 \rightarrow e_2 : e_3, U, C)\), we have that \((B, \sigma)\) satisfies \(C\) and \(\sigma(U) = T\). Therefore \((B, \sigma)\) satisfies \(C_1, C_2, C_3\), as well as \(T_1 <: \text{bool}\), meaning that \(B; \sigma(T_1) <: \text{bool}\).

From the algorithmic definition of a solution (Definition 4.4), \((B, \sigma, \sigma(T_1))\) is a solution for \((\Gamma, e_1, T_1, C_1)\), and similarly for \(e_2\) and \(e_3\). Applying the induction hypothesis, \((B, \sigma, \sigma(T_1))\) is a solution for \((\Gamma, e_1)\) in the declarative sense (Definition 4.3), i.e. \(B; \sigma(\Gamma) \vdash \sigma(e_1) : \sigma(T_1)\) is derivable in the standard type system. The same reasoning applies to \(e_2\) and \(e_3\).
\[
\begin{align*}
& (\sigma, e) \rightarrow v (\sigma', e') \\
& (\sigma, l) \rightarrow v (\sigma, \sigma(l)) \quad \text{(E-LOOKUP)} \\
& (\sigma, \text{true} \rightarrow e : e') \rightarrow v (\sigma, e) \quad \text{(E-TRUE-COND)} \\
& (\sigma, 1 \rightarrow e : e') \rightarrow v (\sigma, e) \quad \text{(E-1-COND)} \\
& v_1, v_2 \text{ are equal integers} \quad \frac{(\sigma, v_1 = v_2) \rightarrow v (\sigma, \text{true})}{(\sigma, v_1 = v_2) \rightarrow v (\sigma, \text{false})} \quad \text{(E-TRUE-EQ)} \\
& v_1, v_2 \text{ are unequal integers} \quad \frac{(\sigma, v_1 = v_2) \rightarrow v (\sigma, \text{true})}{(\sigma, v_1 = v_2) \rightarrow v (\sigma, \text{false})} \quad \text{(E-FALSE-EQ)} \\
& T \in A \quad l \notin \sigma \quad \frac{(\sigma, T \ x ; e) \rightarrow v (\sigma + [l \mapsto \perp], e[l/x])}{(\sigma, T \ x ; e) \rightarrow v (\sigma + [l \mapsto c], e[l/x])} \quad \text{(E-BASE-TYPE-DECL)} \\
& c \notin \sigma \quad \frac{(\sigma, T \ x ; e) \rightarrow v (\sigma + [l \mapsto c], e[l/x])}{(\sigma, T \ x ; e) \rightarrow v (\sigma, e)} \quad \text{(E-SEQ)} \\
& (\sigma, e) \rightarrow (\sigma', e') \quad \frac{(\sigma, e) \rightarrow v (\sigma', e')}{(\sigma, E[e]) \rightarrow (\sigma', E[e'])} \quad \text{(E-CONTEXT)} \\
& (\sigma, e) \rightarrow v (\sigma', e') \quad \frac{(\sigma, E[e]) \rightarrow (\sigma', E[e'])}{(\sigma, E[e]) \rightarrow (\sigma', E[e'])} \quad \text{(E-CONTEXT)} \\
& G \rightarrow G' \quad \frac{(\sigma, E[e] \ | \ l_1, \ldots, l_n) \parallel (E'[e \ | \ v_1, \ldots, v_n]) \rightarrow (\sigma[l_1 := v_1, \ldots, l_n := v_n], (E[1]) \parallel (E'[1]))}{(\sigma, E[e] \ | \ l_1, \ldots, l_n) \parallel (E'[e \ | \ v_1, \ldots, v_n]) \rightarrow (\sigma[l_1 := v_1, \ldots, l_n := v_n], (E[1]) \parallel (E'[1]))} \quad \text{(E-COMMS)} \\
& \text{proctype } P(T_1 \ x_1, \ldots, T_n \ x_n) \{ e \} \text{ is defined} \quad l_1, \ldots, l_n \notin \sigma \quad \frac{(\sigma, (E[\text{run } P(v_1, \ldots, v_n)]) \rightarrow (\sigma + [l_1 \mapsto v_1, \ldots, l_n \mapsto v_n], (E[1]) \parallel (e[l_1/x_1, \ldots, l_n/x_n]))}{(\sigma, (E[\text{run } P(v_1, \ldots, v_n)]) \rightarrow (\sigma + [l_1 \mapsto v_1, \ldots, l_n \mapsto v_n], (E[1]) \parallel (e[l_1/x_1, \ldots, l_n/x_n]))} \quad \text{(E-RUN)} \\
& (\sigma, K) \rightarrow (\sigma', K') \quad (\sigma, K) \rightarrow (\sigma', K') \quad (\sigma, K) \rightarrow (\sigma', K'') \quad (\sigma, K) \rightarrow (\sigma', K''') \quad \text{(E-TRANS)} \\
& K \cong K' \quad K' \cong K'' \quad K'' \cong K''' \quad (\sigma, K') \rightarrow (\sigma', K'') \quad (\sigma, K') \rightarrow (\sigma', K''') \quad \text{(E-TRANS)} \\
& \parallel \text{ is commutative and associative}
\end{align*}
\]

Figure 16: Operational semantics for Promela-calculus.
Because $T_2$ and $T_3$ are base types we have $\sigma(T_2) = T_2$ and $\sigma(T_3) = T_3$. It follows that, $U = T_2 \lor T_3$ is either a base type or Top and so $\sigma(U) = U$; hence $U = T = T_2 \lor T_3$.

From $B; \sigma(\Gamma) \vdash \sigma(e_1) : \sigma(T_1)$ and $\sigma(T_1) : \text{bool}$ we can derive $B; \sigma(\Gamma) \vdash \sigma(e_1) : \text{bool}$ by using rule T-Sub.

Similarly, since $B; \sigma(\Gamma) \vdash \sigma(e_2) : \sigma(T_2)$, $\sigma(T_2) = T_2$ and $T_2 < T_2 \lor T_3$, we have $B; \sigma(\Gamma) \vdash \sigma(e_2) : T_2 \lor T_3$. By a symmetric argument, $B; \sigma(\Gamma) \vdash \sigma(e_3) : T_2 \lor T_3$. Finally, rule T-Cond gives $B; \sigma(\Gamma) \vdash \sigma(e_1) \rightarrow \sigma(e_2) : \sigma(e_3) : T_2 \lor T_3$. Because $\sigma(e_1) \rightarrow \sigma(e_2) : \sigma(e_3) = \sigma(e_1 \rightarrow e_2 : e_3)$ and $T_2 \lor T_3 = T$, this means that $(B, \sigma, T)$ is a solution for $(\Gamma, e)$, as required.

(CT-SEND): In this case $e$ is $x_1e_1, \ldots, e_n$. From the hypothesis of the instance of CT-SEND we have $x : T' \in \Gamma$, $\Gamma \vdash e_i : T_i \mid C_i (1 \leq i \leq n)$, $U = \text{Unit}$ and $C = C_1 \cup \cdots \cup C_n \cup \{T' = \text{chan}\{X_1, \ldots, X_n\}, T_1 < : X_1, \ldots, T_n < : X_n\}$. Again, we omit the $\chi$ annotations.

Because $(B, \sigma, T)$ is a solution for $(\Gamma, x\!\!e_1, \ldots, e_n, U, C)$ we have $(B, \sigma)$ satisfies $C$ and $\sigma(U) = T$. Therefore $(B, \sigma)$ satisfies $C_1, \ldots, C_n$ as well as $T' = \text{chan}\{X_1, \ldots, X_n\}$ and $T_1 < : X_i (1 \leq i \leq n)$. Since $U = \text{Unit}$, $T = \sigma(U) = \text{Unit}$.

For $1 \leq i \leq n$, combining the fact that $(B, \sigma, \sigma(T_i))$ is a solution for $(\Gamma, e_i, T_i, C_i)$ with the induction hypothesis, we have $(B, \sigma, \sigma(T_i))$ is a solution for $(\Gamma, e_i)$, i.e. $B; \sigma(\Gamma) \vdash \sigma(e_i) : \sigma(T_i)$ is derivable in the original type system. Since $(B, \sigma)$ satisfies $T_i < : X_i$, we have $B; \sigma(T_i) < : \sigma(X_i)$. Using rule T-Sub gives $B; \sigma(\Gamma) \vdash \sigma(e_i) : \sigma(X_i)$.

We have $x : T' \in \Gamma$, so $x : \sigma(T') \in \Gamma$. Now $(B, \sigma)$ satisfies $T' = \text{chan}\{X_1, \ldots, X_n\}$, so $\sigma(T') = \sigma(\text{chan}\{X_1, \ldots, X_n\}) = \sigma(\text{chan}\{X_1, \ldots, X_n\}) \in \sigma(\Gamma)$.  

Rule T-SEND now applies, to yield $B; \sigma(\Gamma) \vdash x\!\! \{e_1, \ldots, e_n\} : \text{Unit}$. Because $x\!\! \{e_1, \ldots, e_n\} = x\!\! \{e_1, \ldots, e_n\}$ and Unit = $T$, this means $(B, \sigma, T)$ is a solution for $(\Gamma, e)$ as required.

Theorem 4.2 (completeness), page 17

1. Let $\Gamma$ be a context and $e$ an expression, and suppose $\Gamma \vdash e : U \mid \chi. C$. If $(B, \sigma, T)$ is a solution for $(\Gamma, e)$ and $\text{dom}(\sigma) \cap \chi = \emptyset$ then there is some solution $(B, \sigma', T')$ for $(\Gamma, e, U, C)$ such that $\sigma' \setminus \chi = \sigma$ and $0 \vdash T' < : T$.

2. Let $\Gamma$ be a context and $D$ a declaration, and suppose $\Gamma \vdash D \triangleleft \chi. C$. If $(B, \sigma)$ is a solution for $(\Gamma, D)$ and $\text{dom}(\sigma) \cap \chi = \emptyset$ then there is some solution $(B, \sigma')$ for $(\Gamma, D, C)$ such that $\sigma' \setminus \chi = \sigma$.

3. Let $\Gamma$ be a context and $S$ a specification, and suppose $\Gamma \vdash S : U \mid \chi. C$. If $(B, \sigma, T)$ is a solution for $(\Gamma, S)$ and $\text{dom}(\sigma) \cap \chi = \emptyset$ then there is some solution $(B, \sigma', T')$ for $(\Gamma, S, U, C)$ such that $\sigma' \setminus \chi = \sigma$ and $0 \vdash T' < : T$.

Proof

Again, the proof is by straightforward induction on typing derivations. We give details for the rules CT-COND-COMPUND, CT-SEND and CT-RECEIVE in case 1.

(CT-COND-COMPUND): In this case $e = e_1 \rightarrow e_2 : e_3$, $U = T_2$, $C = C_1 \cup C_2 \cup C_3 \cup \{T_1 < : \text{bool}, T_2 = T_3\}$. From the premises of the rule we have $\Gamma \vdash e_1 : T_1 \mid \chi_1 C_1, \Gamma \vdash e_2 : T_2 \mid \chi_2 C_2, \Gamma \vdash e_3 : T_3 \mid \chi_3 C_3$. We also have $T_2 \notin A'$ or $T_3 \notin A'$.

Since $(B, \sigma, T)$ is a solution for $(\Gamma, e_1 \rightarrow e_2 : e_3)$ we have $B; \sigma(\Gamma) \vdash \sigma(e_1) \rightarrow \sigma(e_2) : \sigma(e_3) : T$, i.e. $B; \sigma(\Gamma) \vdash \sigma(e_1) \rightarrow \sigma(e_2) : T$. Inverting rule T-COND tells us:

- $B; \sigma(\Gamma) \vdash \sigma(e_1) : \text{bool}$
- $B; \sigma(\Gamma) \vdash \sigma(e_2) : T$
- $B; \sigma(\Gamma) \vdash \sigma(e_3) : T$

This means that $(B, \sigma, \text{bool})$ is a solution for $(\Gamma, e_1)$ and $(B, \sigma, T)$ is a solution for both $(\Gamma, e_2)$ and $(\Gamma, e_3)$. By the induction hypothesis there are solutions $(B, \sigma_1, K_1)$ for $(\Gamma, e_1, T_1, C_1)$, $(B, \sigma_2, K_2)$ for $(\Gamma, e_2, T_2, C_2)$ and $(B, \sigma_3, K_3)$ for $(\Gamma, e_3, T_3, C_3)$ such that $\sigma_1 \setminus \chi_1 = \sigma_2 \setminus \chi_2 = \sigma_3 \setminus \chi_3 = \sigma_0 \setminus K_1 < : \text{bool}, 0 \vdash K_2 < : T$ and $0 \vdash K_3 < : T$.

Because for $1 \leq i \neq j \leq 3$ we have $\sigma_i \setminus \chi_i = \sigma_j \setminus \chi_j$ and $\chi_i \cap \chi_j = \emptyset$ we can extend $\sigma_1, \sigma_2, \sigma_3$ to $\sigma'$ such that $\sigma' \setminus \chi_1 \cup \chi_2 \cup \chi_3 = \sigma$. To complete the proof for case CT-COND-COMPUND we show that $(B, \sigma', T)$ is a solution for $(\Gamma, e_1 \rightarrow e_2 : e_3, T_2, C_3 \cup C_1 \cup C_2 \cup \{T_1 < : \text{bool}, T_2 = T_3\})$.

We need to show that $\sigma'$ satisfies $C_1 \cup C_2 \cup C_3 \cup \{T_1 < : \text{bool}, T_2 = T_3\}$. We have $\sigma'$ satisfies $C_1$ from the fact that $\sigma_i$ satisfies $C_i (1 \leq i \leq 3)$, Since $T_1$ does not include any type variables contained in $\chi_1$, $\sigma'_1(T_1) = \sigma_1(T_1) = K_1 < :
bool = σ'(bool), so σ' satisfies $T_1: bool$. For $i \in \{2, 3\}$, no variable in $\chi_i$ appears in $T_i$, so $σ'(T_i) = σ(T_i)$. So we have $σ'(T_2) = K_2$ and $σ'(T_3) = K_3$. We know that at least one of $T_2, T_3$ is not a base type. It follows that at least one of $σ'(T_2), σ'(T_3)$, i.e. one of $K_2, K_3$, is not a base type. Since subtyping can only occur between base types, $∅ ⊢ K_2 < T$ and $∅ ⊢ K_3 < T$ implies that $K_2 = K_3 = T$. Thus $σ'$ satisfies $T_2 = T_3$.

(CT-SEND): In this case $e = x_ι e_1, \ldots, e_n$, $U = Unit$, $C = C_ι \cup \cdots \cup C_n \cup \{V = chan\{X_1, \ldots, X_n\}, T_1: X_1, \ldots, T_1: X_n\}$. From the premises of the rule we have: $x: V ∈ Γ, Γ ⊢ e_i: T_i |_{X_i}, C_i (1 ≤ i ≤ n)$.

Since $(B, σ, T)$ is a solution for $(Γ, (x_ι e_1, \ldots, e_n))$ we have $B; σ(Γ) ⊢ σ(x_ι e_1, \ldots, e_n) : T$, i.e. $B; σ(Γ) ⊢ x_ι σ(e_1), \ldots, σ(e_n) : T$. Inverting rule T-SEND we have:

- $T = Unit$
- $B; σ(Γ) ⊢ σ(e_i) : W_i (1 ≤ i ≤ n)$
- $x: chan\{W_1, \ldots, W_n\} ∈ σ(Γ)$, i.e. $x: chan\{V_1, \ldots, V_n\} ∈ Γ$, where $σ(V_i) = W_i (1 ≤ i ≤ n)$.

This tells us that $V = chan\{V_1, \ldots, V_n\}$ and $(B, σ, W_i)$ is a solution for $(Γ, e_i) (1 ≤ i ≤ n)$.

For each $i (1 ≤ i ≤ n)$ by the induction hypothesis there is a solution $(B, σ_i, K_i)$ for $(Γ, e_i, T_i, C_i)$ such that $σ_i \setminus \chi_i = σ' \setminus \chi_j$ and $χ_i \cap \chi_j = ∅$. Therefore $(B, σ_i)$ satisfies $K_i$.

Because for any $1 ≤ i ≠ j ≤ n$, $σ_i \setminus \chi_i = σ_j \setminus \chi_j$ and $χ_i \cap \chi_j = ∅$, we can extend $σ_1, \ldots, σ_n$ to $σ'$ such that $σ' \cap \chi = σ'$ and $σ'(X_i) = W_i (1 ≤ i ≤ n)$. To complete the proof for CT-SEND we must show that $(B, σ', Unit)$ is a solution for $(Γ, (x_ι e_1, \ldots, e_n), Unit, C_ι \cup \cdots \cup C_n \cup \{V = chan\{X_1, \ldots, X_n\}, T_1: X_1, \ldots, T_n: X_n\})$. This requires $σ'$ to satisfy $C_ι \cup \cdots \cup C_n \cup \{V = chan\{X_1, \ldots, X_n\}, T_1: X_1, \ldots, T_n: X_n\}$.

Therefore we have $V = chan\{V_1, \ldots, V_n\}, σ(T_1) = U$, and $B; σ(Γ) ⊢ σ(T_i) (1 ≤ i ≤ n)$.

Define $σ'$ as follows: $σ'(Y) = σ(K)$ if $Y ∉ \{X_1, \ldots, X_n\}$ and $Y ∈ dom(σ); σ(X_i) = σ(V_i) (1 ≤ i ≤ n)$. It clearly that $σ' \setminus \{X_1, \ldots, X_n\} = σ$. To complete the proof for CT-RECEIVE we must show that $(B, σ', Unit)$ is a solution for $(Γ, (x_ι x_j, \ldots, x_n), Unit, \{V = chan\{X_1, \ldots, X_n\}, X_1: T_1, \ldots, X_n: T_n\})$, which requires $σ'$ to satisfy $V = chan\{X_1, \ldots, X_n\}, X_1: T_1, \ldots, X_n: T_n\}$. We have $σ'(V) = σ'(chan\{X_1, \ldots, X_n\}) = chan\{σ(V_1), \ldots, σ(V_n)\}$ (since none of $X_1, \ldots, X_n$ appear in any of $V_1, V_2, \ldots, V_n$).

B.2. Proofs of results in Section 4.3

Lemma 4.1 page 18 Suppse $Γ ⊢ S: U | C$. If $T_1: T_2 ∈ C$ then at least one of $T_1, T_2$ is not a type variable.

Proof: The result follows from the fact that for any $PC_{part}$ expression $e$, if $Γ ⊢ e: U | C$ then $U$ is not a type variable. To see this, observe that all typical typing rules for expressions in Figure 12 either assign a type from $A^*$ (bool,
Unit, or $T_1 \lor T_2$, where $T_1, T_2 \in \mathcal{A}^\ast$, or pass through a type computed by other rules. Exceptions to this are “source” rules CT-BOOL-LIT, CT-NUM-LIT and CT-VAR. The first two of these rules generate base types. Rule CT-VAR deduces the type for a variable reference $x$ from the environment $\Gamma$. An entry in $\Gamma$ comes from the rule CT-DECL, which adds $x : T$ to $\Gamma$ based on the type specified for $x$ at declaration. This type is restricted to the syntax of Figure 11 and thus cannot be a type variable.

Every rule in Figure 12 which generates subtyping constraints produces constraints with one of three forms: $T_1 \ll T_2$, $T_1 \ll X$, or $X \ll T_2$. In each case $T_1, T_2$ are types generated by the constraint typing relation for expressions, and thus are not type variables by the above argument.

Lemma 3.2 Let $C$ be a set of constraints and let $B_0$ be a bounds function such that $FV(C) \subseteq dom(B_0)$. Then $\text{unify}(C, B_0)$ terminates, either by failing or by returning a result of the form $(B, \sigma)$.

Proof The argument is the same as in Pierce [26] Theorem 22.4.5. Given a set of constraints $C$, let $m$ be the number of distinct free type variables in $C$ and let $n$ be the total size of the types in $C$. Then each recursive call of $\text{unify}$ moves strictly downwards in the lexicographic order on $(m, n)$.

We prove Theorem 4.3 via a series of technical lemmas.

Lemma B.1 If $(B, \sigma)$ unifies $\{X = T\} \cup C$ then $(B, \sigma)$ unifies $\{X \rightarrow T'\} \cup C$, where $T'$ is either $\mu Y.F(Y)$ if $T$ is of the form $F(X)$ (for some fresh type variable $Y$) or $T$ otherwise.

Proof By induction on the size of $C$. The base case, $C = \emptyset$, is trivial because $[X \rightarrow T']\emptyset$ is $\emptyset$.

If $C = \{U \ll U'\} \cup C'$ then $[X \rightarrow T']C = \{[X \rightarrow T']U \ll [X \rightarrow T']U' \cup [X \rightarrow T']C'\}$. Since $(B, \sigma)$ unifies $\{X = T\} \cup C$ and $C' \subseteq C$ we have that $(B, \sigma)$ unifies $\{X = T\} \cup C'$ and so, by the induction hypothesis, $(B, \sigma)$ unifies $[X \rightarrow T']C'$. It remains to show that $(B, \sigma)$ unifies $[X \rightarrow T']U \ll [X \rightarrow T']U'$, i.e. $B \vdash \sigma([X \rightarrow T']U) : [X \rightarrow T']U'$. Because $(B, \sigma)$ unifies $X = T$, we have $\sigma(X) = \sigma(T')$ (if $T$ is of the form $F(X)$ then this requires equivalence of a recursive type and its unfolding) and therefore $\sigma([X \rightarrow T']U) = \sigma(U)$, similarly $\sigma([X \rightarrow T']U') = \sigma(U')$. Since $(B, \sigma)$ unifies $U : U'$ we have $B \vdash \sigma(U) : \sigma(U')$. This completes the argument.

If $C = \{U = U'\} \cup C'$ then $[X \rightarrow T']C = \{[X \rightarrow T']U = [X \rightarrow T']U' \cup [X \rightarrow T']C'\}$ and we require $\sigma([X \rightarrow T']U) = \sigma([X \rightarrow T']U')$. This follows from $\sigma(X) = \sigma(T')$ as before.

Lemma B.2 If $(B, \sigma) \nsubseteq (B', \sigma')$ and $\sigma'(X) = Y$ and $B(X) = (L_X, U_X)$ and $B'(Y) = (L_Y, U_Y)$ then $L_X \ll : L_Y \ll : U_Y \ll : U_X$.

Proof Let $\theta$ be an instance of $(B', \sigma')$ such that $\theta(Y) = U_Y$. There exists $\gamma$ such that $\theta = \gamma \circ \sigma'$, so $\theta(X) = \gamma(\sigma'(X)) = \gamma(Y)$ by hypothesis. Because $(B', \sigma')$ is a bounded substitution and $\sigma'(X) = Y$, $Y \notin dom(\sigma')$, and so $\theta(Y) = \gamma(Y)$. Therefore $\theta(X) = \theta(Y)$. We have that $\theta$ is also an instance of $(B, \sigma)$ and so $\theta(X) \ll : U_X$, i.e. $U_Y \ll : U_X$ by choice of $\theta$. Similarly by taking $\theta(Y) = L_Y$ we obtain $L_X \ll : L_Y$ and we also have $L_Y \ll : U_Y$ by definition of $\ll$. The result follows.

Lemma B.3 If $(B, \sigma) \nsubseteq (B', \sigma')$ and $B(X) = (L_X, U_X)$ and $B'(X) = (L'_X, U'_X)$ then $L_X \ll : L'_X \ll : U'_X \ll : U_X$.

Proof Similar to the proof of Lemma B.2 consider an instance $\theta$ of $(B', \sigma')$ such that $\theta(X)$ is either $L'_X$ or $U'_X$, and use the fact that $\theta$ is also an instance of $(B, \sigma)$.

Lemma B.4 Let $L, U \in \mathcal{A}^\ast$, $T$ a type which is not a type variable, and $\sigma$ a substitution. Suppose $L \ll : T \ll : U$. Then $L \ll : \sigma(T) \ll : U$.

Proof If $T \in \mathcal{A}^\ast$ then $\sigma(T) = T$ and the result follows. Otherwise, $T$ is either a channel type or a product type. The only types in $\mathcal{A}^\ast$ to which $T$ is comparable via the subtyping relation are therefore Bot and Top, so $L \ll : T \ll : U$ forces $L = \text{Bot}$ and $U = \text{Top}$, from which we have $L \ll : \sigma(T) \ll : U$ as required.

Lemma B.5 Let $C$ be a set of constraints and let $B_0$ be a bounds function such that $FV(C) \subseteq dom(B_0)$. Suppose that $\text{unify}(C, B_0) = (B, \sigma)$. Then:

1. $\text{dom}(\sigma) \subseteq FV(C)$.
2. $\text{dom}(B) \cup \text{dom}(\sigma) = \text{dom}(B_0)$.
3. $(B, \sigma)$ is a bounded substitution.
For every $X \in \text{dom}(B_0)$, $B \vdash L_X <: \sigma(X) <: U_X$ where $B_0(X) = (L_X, U_X)$.

5. $(B, \sigma)$ is a unifier for $C$.

**Proof** Each part is proved by induction on the number of recursive calls resulting from the initial call unify($C, B_0$). The assumption that unify($C, B_0$) = $(B, \sigma)$ implies that unify($C, B_0$) does not fail, enabling us to assume various conditions checked by the algorithm. Parts (1) and (2) are straightforward. The proof of part (3) uses (1) to show that when unifying $Y \rightarrow X$ for $C$, $Y$ is not in the domain of the resulting substitution.

For part (4) some cases require more reasoning. If $C = \{X \leftarrow T_2\} \cup C'$ then let $(L_X, U_X) = B_0(X)$ and so $(B, \sigma) = \text{unify}(C', B_0 \setminus \{X\} + [X \mapsto (L_X, U_X \cup T_2)])$. Consider a type variable $Z$. If $Z \neq X$ then the induction hypothesis gives $B \vdash L_Z <: \sigma(Z) <: U_Z$ where $B_0(Z) = (L_Z, U_Z)$, as required for the conclusion. If $Z = X$ then the induction hypothesis gives $B \vdash L_X <: \sigma(X) <: U_X \cup T_2$, and the conclusion follows because $U_X \cup T_2 <: U_X$.

The case where $C = \{T_1 <: X\} \cup C'$ is symmetric. If $C = \{X = Y\} \cup C'$ then let $(L_X, U_X) = B_0(X)$ and $(L_Y, U_Y) = B_0(Y)$, and so $(B, \sigma) = \{B', \sigma' [Y \mapsto X]\}$ where $(B', \sigma') = \text{unify}(Y \mapsto X)[C', B_0 \setminus \{X, Y\} + [X \mapsto (L_X \cup L_Y, U_X \cup U_Y)])$. Consider a type variable $Z$. If $Z \neq X$ and $Z \neq Y$ then the induction hypothesis gives $B \vdash L_Z <: \sigma'(Z) <: U_Z$ where $B_0(Z) = (L_Z, U_Z)$, as required for the conclusion because $\sigma'(Z) = \sigma(Z)$. If $Z = X$ then the induction hypothesis gives $B \vdash L_X \cup L_Y <: \sigma'(X) <: U_X \cup U_Y$, and the conclusion follows because $\sigma'(X) = \sigma(X)$ and $L_X <: L_X \cup L_Y$ and $U_X \cup U_Y <: U_X$. If $Z = Y$ then similar reasoning concludes the argument, noting that $\sigma(Y) = \sigma'(X)$.

If $C = \{X = T_2\} \cup C'$ then the proof is straightforward induction.

For part (5) we again show the more interesting cases. If $C = \{X <: T_2\} \cup C'$ then let $(L_X, U_X) = B_0(X)$ and so $(B, \sigma) = \text{unify}(C', B_0 \setminus \{X\} + [X \mapsto (L_X, U_X \cup T_2)])$. By the induction hypothesis, $(B, \sigma)$ unifies $C'$. By (4), $B \vdash L_X <: \sigma(X) <: U_X \cup T_2$. Hence $B \vdash \sigma(X) <: T_2$, and so $(B, \sigma)$ unifies $C$. The case where $C = \{T_1 <: X\} \cup C'$ is symmetric.

If $C = \{X = Y\} \cup C'$ then let $(L_X, U_X) = B_0(X)$ and $(L_Y, U_Y) = B_0(Y)$, and so $(B, \sigma) = \{B', \sigma' [Y \rightarrow X]\}$ where $(B', \sigma') = \text{unify}(Y \mapsto X)[C', B_0 \setminus \{X, Y\} + [X \mapsto (L_X \cup L_Y, U_X \cup U_Y)])$. The induction hypothesis $(B', \sigma')$ unifies $[X \mapsto Y][C'$ and therefore it is easy to see that $(B, \sigma)$ unifies $C'$; also it clearly unifies $X = Y$. If $C = \{X = T_2\} \cup C'$ or $C = \{T_1 = X\} \cup C'$ then similar reasoning applies.

Let $\mathcal{C}$ be a set of constraints and let $B_0$ be a bounds function such that $FV(\mathcal{C}) \subseteq \text{dom}(B_0)$. Suppose that $(B, \sigma)$ is a unifier for $\mathcal{C}$ and $(B_0[\mathcal{C}]) \subseteq (B, \sigma)$. Then unify($C, B_0$) does not fail.

**Proof** By induction on the number of recursive calls in unify($C, B_0$). The base case is trivial.

If $C = \{T_1 <: T_2\} \cup C'$ and $T_1, T_2 \in A$ then because $(B, \sigma)$ unifies $C$ we have $B \vdash \sigma(T_1) <: \sigma(T_2)$. Because $T_1 \in A$, $\sigma(T_1) = T_1$, and similarly for $T_2$. So $T_1 <: T_2$, and unify($C, B_0$) does not fail.

If $C = \{X <: T_2\} \cup C'$ and $T_2 \in A$ then let $(L_X, U_X) = B_0(X)$. Because $(B, \sigma)$ unifies $C$ we have $B \vdash \sigma(X) <: T_2$. Now consider whether or not $X \in \text{dom}(\sigma)$. If $X \notin \text{dom}(\sigma)$ then the proof is complete. Let $(L_X', U_X') = B(X)$, and then $U_X' <: T_2$. Because $(B_0[\mathcal{C}]) \subseteq (B, \sigma)$, Lemma B.3 gives $L_X <: L_X' <: U_X' <: U_X$. Therefore $L_X <: U_X \cup T_2 \neq \bot$ and so unify($C, B_0$) does not fail. If $X \in \text{dom}(\sigma)$ then there are two further sub-cases. If $\sigma(X) = Y$ then let $(L_Y, U_Y) = B(Y)$ (because $(B, \sigma)$ is a bounded substitution, $Y \notin \text{dom}(\sigma)$ and so $Y \in \text{dom}(B)$) by Lemma B.3. We have $B \vdash Y <: T_2$, so $U_Y <: T_2$. By Lemma B.2, $L_X <: L_Y <: U_Y <: U_X$ and so $L_X <: U_X \cup T_2 \neq \bot$ as before. Finally, if $\sigma(X)$ is not a type variable then for a suitable $\gamma$, define $\theta = \gamma \circ \sigma$ so that $\theta$ is an instance of $(B, \sigma)$. Because $(B_0[\mathcal{C}]) \subseteq (B, \sigma)$, $\theta$ is also an instance of $(B_0[\mathcal{C}])$. Therefore $L_X <: \theta(X) <: U_X$. If $\sigma(X) \in A$ then $\theta(X) = \sigma(X)$. If $\sigma(X) \notin A$ then $\sigma(X)$ has the same subtyping relationships as $\theta(X)$ because its top-level type constructor is invariant for subtyping. In either case, we have $L_X <: \sigma(X) <: U_X$ and so $L_X <: \sigma(X) <: U_X \cup T_2 \neq \bot$ as required.

If $C = \{T_1 <: X\} \cup C'$ and $T_1 \in A$ then the reasoning is symmetrical to the previous case.

If $C = \{X <: T_2\} \cup C'$ and $T_2 \notin A$ then unify replaces the subtyping constraint with $X = T_2$, which is handled by the argument below; a similar argument applies when $C = \{T_1 = X\} \cup C'$.

If $C = \{X = Y\} \cup C'$ then let $(L_X, U_X) = B_0(X)$ and $(L_Y, U_Y) = B_0(Y)$. Because $(B, \sigma)$ unifies $X = Y$, we have $\sigma(X) = \sigma(Y)$. We now consider three cases. If $\sigma(X) = Y$ (and so $Y \notin \text{dom}(\sigma)$) then let $(L_Y, U_Y') = B(Y)$. By Lemma B.2, $L_X <: L_Y <: U_Y' <: U_X$. By Lemma B.2, $L_Y <: U_Y' <: U_Y$. Therefore $L_Y <: L_X <: U_X \cup U_Y \neq \bot$ and unify($C, B_0$) does not fail. If $\sigma(Y) = X$ then symmetrical reasoning applies. Finally, suppose that $X, Y \in \text{dom}(\sigma)$ and $\sigma(X) = \sigma(Y)$. There are two further sub-cases. If $\sigma(X) = \sigma(Y) = Z$, a type variable, then let $(L_Z, U_Z) = B(Z)$. Lemma B.2 gives $L_X <: L_Z <: U_Z <: U_Y$ and $L_Y <: L_Z <: U_Z <: U_Y$, which is sufficient by similar reasoning to the previous case. If $\sigma(X) = \sigma(Y) = \top$, not a
type variable, then for a suitable \( \gamma \) define \( \theta = \gamma \circ \sigma \) so that \( \theta \) is an instance of \((B, \sigma)\). Because \((B_0, \{\})\) \(\subseteq (B, \sigma)\), \(\theta\) is also an instance of \((B_0, \{\})\), so we have \(L_X <: \theta(X) = \theta(Y) <: U_X\) and \(L_Y <: \theta(X) = \theta(Y) <: U_Y\) which again is sufficient.

If \( C = \{X = T_2\} \cup C' \) where \( T_2 \) is not a type variable, then let \((L_X, U_X) = B_0(X)\). Since \((B, \sigma)\) unifies \(X = T_2\) we have \(\sigma(X) = \sigma(T_2)\). Since \(T_2\) is not a type variable this implies that \(X \in \text{dom}(\sigma)\). For a suitable \( \gamma \) define \( \theta = \gamma \circ \sigma \) so that \( \theta \) is an instance of \((B, \sigma)\). Because \((B_0, \{\}) \subseteq (B, \sigma)\), \(\theta\) is also an instance of \((B_0, \{\})\), so we have \(L_X <: \theta(X) <: U_X\), i.e. \(L_X <: \gamma(\sigma(X)) <: U_X\), i.e. \(L_X <: \gamma(\sigma(T_2)) <: U_X\). Applying Lemma B.4 we have \(L_X <: T_2 <: U_X\), which is what is needed for unify to succeed.

If \( C = \{T_1 = X\} \cup C' \), \(T_1\) is not a type variable, then the reasoning is symmetrical to the previous case.

\[\square\]

Lemma B.7 Let \( C \) be a set of constraints and let \( B_0 \) be a bounds function such that \( \text{FV}(C) \subseteq \text{dom}(B_0) \). Suppose that \((B, \sigma)\) is a unifier for \( C \) and \((B_0, \{\}) \subseteq (B, \sigma)\). Then \(\text{unify}(C, B_0) \subseteq (B, \sigma)\).

**Proof** Again the proof is by induction on the number of recursive calls in \(\text{unify}(C, B_0)\). We show the non-trivial cases.

If \( C = \{X <: T_2\} \cup C' \) and \( T_2 \in A \) then let \((L_X, U_X) = B_0(X)\). We have \(\text{unify}(C, B_0) = \text{unify}(C', B_0 \setminus \{X\} + [X \mapsto (L_X, U_X \land T_2)]) = (B', \sigma')\) as expected. By hypothesis, \((B, \sigma)\) unifies \(C'\). To use the induction hypothesis, which will complete the argument, we need \((B \setminus \{X\} + [X \mapsto (L_X, U_X \land T_2)]) \subseteq (B, \sigma)\). Let \(\theta\) be an instance of \((B, \sigma)\).

If \( C = \{X = Y\} \cup C' \) then let \((L_X, U_X) = B_0(X)\) and \((L_Y, U_Y) = B_0(Y)\). We have \(\text{unify}(C, B_0) = (B', \sigma' \circ [Y \mapsto X])\) where \((B', \sigma') = \text{unify}([Y \mapsto X]C', B_0 \setminus \{X, Y\} + [X \mapsto (L_X \lor L_Y, U_X \lor U_Y)])\). We need to know that \((B', \sigma' \circ [Y \mapsto X]) \subseteq (B, \sigma)\). By Lemma B.1 \((B, \sigma)\) unifies \([Y \mapsto X]C'\). To use the induction hypothesis we need \((B_0 \setminus \{X, Y\} + [X \mapsto (L_X \lor L_Y, U_X \lor U_Y)]) \subseteq (B, \sigma)\). Let \(\theta\) be an instance of \((B, \sigma)\). Then \(\theta = \gamma \circ \sigma\) for some \(\gamma\). We need \(L_Y \lor L_X <: \theta(X) <: U_X \lor U_Y\). \(\theta\) is also an instance of \((B_0, \{\})\), so \(L_X <: \theta(X) <: U_X\) and \(L_Y <: \theta(Y) <: U_Y\). But \(\sigma\) unifies \(X = Y\), so \(\sigma(X) = \sigma(Y)\) and \(\theta(X) = \theta(Y)\). Therefore \(L_X \lor L_Y <: \theta(X) = \theta(Y) <: U_X \lor U_Y\). Now, the induction hypothesis gives \((B', \sigma') \subseteq (B, \sigma)\). Let \(\phi\) be an instance of \((B, \sigma)\); it is therefore also an instance of \((B', \sigma')\). We need to show that it is an instance of \((B', \sigma' \circ [Y \mapsto X])\). The condition on bounds follows directly from the fact that \(\phi\) is an instance of \((B', \sigma')\). We also need the existence of \(\gamma\) such that \(\phi = \gamma \circ \sigma \circ [Y \mapsto X]\). Therefore \(\phi\) is an instance of \((B', \sigma')\), there exists \(\delta\) such that \(\phi = \delta \circ \sigma \circ \sigma\). We will show that \(\phi \circ [X \mapsto Y] = \phi\) and then take \(\gamma = \delta\). Because \(\phi\) is an instance of \((B', \sigma')\), there exists \(\eta\) such that \(\sigma = \eta \circ \sigma\) and so, because \(\sigma(X) = \sigma(Y)\), we have \(\phi(Y) = \phi(Y)\). Now consider the action of \(\phi \circ [X \mapsto Y]\) on any type variable \(Z\). If \(Z \neq X\) then \((\phi \circ [X \mapsto Y])(Z) = \phi(Z)\). And \((\phi \circ [X \mapsto Y])(X) = \phi(Y) = \phi(X)\). Therefore \(\phi \circ [X \mapsto Y] = \phi\), as required.

If \( C = \{X = T_2\} \cup C' \) and \( T_2 \) is not a type variable, then the reasoning is similar, and this reasoning also applies if \( C = \{X <: T_2\} \cup C' \) and \( T_2 \not\in A \), since \(\text{unify}\) replaces the subtyping constraint with \(X = T_2\); a similar argument applies when \( C = \{T_1 = X\} \cup C'\).

\[\square\]

Lemmas B.5, B.6 and B.7 prove that the \(\text{unify}\) function computes the most general unifier for a set of constraints, when instantiated with a bounds function which assigns trivial bounds (\(\text{Bot}, \text{Top}\)) to every free type variable occurring in the constraints.

**Theorem 4.3** Let \( C \) be a set of constraints and let \( B_0 \) be the bounds function defined by \(\text{dom}(B_0) = \text{FV}(C)\) and \(\forall X \in \text{dom}(B_0), B_0(X) = (\text{Bot}, \text{Top})\). If there exists a unifier for \( C \) then \(\text{unify}(C, B_0)\) is a unifier for \( C \), and for every unifier \((B, \sigma)\) for \( C \), \(\text{unify}(C, B_0) \subseteq (B, \sigma)\). That is, \(\text{unify}(C, B_0)\) is the most general unifier for \( C \).

**Proof** Directly from Lemmas B.5, B.6 and B.7.

\[\square\]