Interpreting π-calculus with Spin/Promela

Peng Wu

Laboratory for Computer Science,
Institute of Software, Chinese Academy of Sciences,
Beijing, China, 100080
wp@ios.ac.cn

Abstract. In Promela, the specification language for Spin, messages passing between processes are able to contain channel identifiers, which intuitively provides a direct way to interpret π-calculus into Promela and therefore verify specifications in π-calculus with Spin. This paper presents a rule-based translation algorithm with an experimental tool *pi2promela*. Regarding the translation, two types of rules are introduced. One is to generate variables from names in π-calculus, while another is to generate statements from process expressions. A case study with Bluetooth Service Discovery Protocol shows its effectiveness.

1 Introduction

π-Calculus [4, 5] is characterized by describing the mobility with a well-built algebraic foundation. It enables to specify and verify concurrent systems with dynamically evolving communication topologies. Much theoretical research has been devoted to π-calculus, along with a number of industrial applications. Such research also contributes to many verification tools for model checking π-calculus, such as Mobility Workbench [1, 2]; however they're still not so practical.

On the other hand, Spin [3] has been regarded as an efficient software to specify and verify distributed systems. Furthermore, in Promela, the input language of Spin, messages passing between processes are able to contain channel identifiers, which actually provides a concrete way to model the mobility. So one question may arise whether it's possible to interpret π-calculus into Promela so as to verify specifications in π-calculus with Spin. If so, not only Spin can be extended indirectly to deal with mobile processes, but also π-calculus may gain much usability from Spin, i.e. to some extent, become executable.

Our work gives a positive answer. This paper presents a rule-based translation algorithm, which interprets π-calculus into Promela. Furthermore an experimental tool *pi2promela* has also been developed in ANSI C. To show its effectiveness, a case study with Bluetooth Service Discovery Protocol [6] is also presented.

The paper is organized as follows. Section 2 and 3 give a brief introduction on π-calculus and Promela respectively. The translation algorithm is illustrated in Section 4, while Section 5 shows some implementation details of the tool *pi2promela*. Section 6 presents the case study. The paper is concluded in Section 7 with some future work.

2 π-Calculus

Only two kinds of entities are involved in π-calculus. One is *name* that is always thought of as the name of a communication channel, while another is *process* that expresses some mobile system. Processes use names to interact, and especially pass names to one another in their interactions.

Let *p, q*...range over the set of processes, *x, y*...range over the set of names. The syntax of π-calculus is given as follows.

$$p ::= 0 \mid \alpha.p \mid p + p \mid p \mid p \mid (\nu x)p \mid [x = y]p \mid !p \mid A(\tilde{x})$$
$$\alpha ::= \tau \mid x(y) \mid \bar{x} < y >$$

0 is inaction that can do nothing. The prefix $\alpha.p$ evolves into p after performing the action α. There are totally three types of actions: the input action, denoted by $x(y)$ that receives any name y via x; the output action, denoted by $\bar{x} < y >$ that sends the name y via x; the silent action, denoted by τ that expresses an internal action.

The summation $p + q$ represents a nondeterministic choice between p and q. If any one of summands is exercised, another one is rendered void. On the contrary, for the composition $p \mid q$, the component p and q can proceed in a parallel way and can also interact via shared names. In the restriction $(\nu x)p$, the scope of the name x is restricted. For the match $[x = y]p$, the process p is executable only if x and y are the same names. The replication $!p$ can be thought of as an infinite composition $p \mid p \mid \cdots$. Finally, $A(\tilde{x})$ represents a process identifier of which the definition is in the form

$$A(\tilde{x}) \overset{def}{=} p$$

3 Promela

In Spin, a formal specification is built using Promela, which is an imperative language supporting nodeterministism and parallelism with

1) The selection statement **if** :: *sequence* [:: *sequence*]* **fi**, which describes a nondeterministic choice among those guarded conditions prefiexed by **::**, and
2) The predefined unary operator **run** *name* ([*argument list*]), which is used to create new process. The new process executes asynchronously with the current active processes from this point on.

Promela also provides a direct means to describe interprocess communication via an explicit message passing channel. Both synchronous and asynchronous communication are supported. For synchronous communication, the channel works in a rendezvous mode with the capacity of zero; while for asynchronous communication, the channel works as a FIFO buffer with a nonzero capacity.

For example, **chan** c = [0] of {chan}; where *chan* is the keyword for channel declaration. It defines a single rendezvous (indicated by 0) channel c for messages that contain just one field of type *chan*. It's because the ability to pass channel identifiers in messages that Promela can serve as an optimal candidate for interpreting π-calculus. The sending and receiving operations are denoted respectively by

3) The sent statement name ! arguments, which sends messages to the channel specified by name, and
4) The receive statement name ? arguments, which receives messages from the channel specified by name.

For detailed Promale grammer, please refer to [7].

4 Rule-based Algorithm

The basic idea of our algorithm comes from the correspondences between concepts in π-calculus and in Promela.

- Statements in Promela are the counterpoint of process expressions in π-calculus.
- Variables in Promela are the counterpoint of names in π-calculus;

Therefore our algorithm works by structural reduction on process expressions. Two types of translation rules are introduced:

- Rules for generating statements from actions and processes;
- Rules for generating variables via name synthesis.

As mentioned above, messages in π-calculus contain only names, no values. So the only variables concerned during translation are those of type *chan*. In addition, only synchronous channels are involved according to the semantics model of π-calculus.

4.1 Statement Generation

Actions are correspondent with atomic statements in Promela. Messages in the Promela code contain only zero or more channel identifiers. The rules to translate actions are shown in Fig 1.

$$SILENT-ACT\ \frac{\tau}{\texttt{true}} \quad INPUT-ACT\ \frac{x(y)}{\texttt{x?y}} \quad OUTPUT-ACT\ \frac{\overline{x}<y>}{\texttt{x!y}}$$

Fig. 1. Rules for Actions

The silent action τ gives its continuation a high priority. For process $\tau.p$, the continuation p is enabled unconditionally at the time. So τ is interpreted as `true`. It's obvious that the input prefix and output prefix should be translated into the receive and send statement respectively.

Based on action translation, the algorithm interprets process expressions according to their syntactical structure. The rules to translate process expressions are shown in Fig 2.

$$INACTION\ \frac{0}{-} \quad PREFIX\ \frac{\alpha.p}{\sigma(\alpha);\sigma(p)} \quad MATCH\ \frac{[x=y]p}{(\mathrm{x}==\mathrm{y})\rightarrow\sigma(p)}$$

$$PAR_1\ \frac{p\mid q}{\texttt{run Id}[p];\sigma(q)} \quad \begin{array}{l}none\ of\ p\ or\ q\ is\ an\ identifier\ ,or\\ p\ is\ an\ identifer\ \ but\ \ q\ is\ not.\end{array}$$

$$PAR_2\ \frac{p\mid q}{\texttt{run Id}[q];\sigma(p)} \quad q\ is\ an\ identifer\ \ but\ \ p\ is\ not.$$

$$PAR_3\ \frac{p\mid q}{\texttt{run Id}[p];\ \&\ \&\ \texttt{run Id}[q];} \quad p\ and\ \ q\ are\ both\ \ identifier\ s.$$

$$SUM\ \frac{p+q}{\texttt{if}\ ::\sigma(p)\ ::\sigma(q)\ \texttt{fi;}} \quad RES\ \frac{(\nu x)p}{\sigma(p)} \quad IDE\ \frac{A(\tilde{x})}{\texttt{proctype A(}\overline{\texttt{chan x}}\texttt{)}\ \{\ \text{chan } \overline{z};\ \sigma(p)\ \}} \quad A(\tilde{x}) \overset{def}{=} p$$

Fig. 2. Rules for Processes

1) For inaction *0*, no statement is generated.

2) The prefix and match both are interpreted as sequential statement. Note that the semi-colon and the arrow are equivalent statement separators in Promela.

3) For the composition, there are three cases. `Id[p]` represents the identifier of the process *p*.

a) If one of components is an identifier while another is not, a `run` statement is used to fork a process, for the component that is an identifier, proceeding in a parallel way with current active processes, as rule PAR_1 and PAR_2 show.

b) If none of components is an identifier, a new identifier is to be introduced for the first one, i.e. *p*, as rule PAR_1 shows. Such identifier is generated randomly, but certain to be non-conflicting from others. Then such case can be dealt with as a).

c) If both components are identifiers, two `run` statements are used to fork two concurrent processes, as rule PAR_3 shows.

4) Rule *SUM* interprets the summation as a guarded selection statement.

5) For the restriction, a local channel variable *x* is declared for process *p* in the Promela code.

6) Finally, with rule *IDE*, the complete definition of a process can then be generated, i.e. the body of the definition is combined with its interface. Here $\overline{\texttt{chan x}}$ represents a channel declaration sequence **chan x$_1$, chan x$_2$, …, chan x$_n$**, where $\tilde{x} = x_1, x_2, \ldots, x_n, n \geq 0$.

Furthermore, these rules for actions and processes can be easily extended for polyadic π-calculus. For example,

$$INPUT - ACT\ \frac{x(\tilde{y})}{\texttt{x?}\overline{\texttt{y}}}$$

where $\tilde{y} = y_1, y_2, \ldots, y_n, n \geq 0$; $\overline{\texttt{y}}$ represents an argument list **y$_1$, y$_2$, …, y$_n$**,

4.2 Name Capture Avoidance

Generally speaking, all names can be reused for the definition of variables in the Promela code. However in π-calculus, α-conversion provides a basic but implicit means to keep names from conflict. The algorithm has to perform such conversion explicitly, whenever necessary, so that no name conflict may take place during code generation. Remember that such α-conversion should be injective so that different names wouldn't be converted to the same one. Let $[S]_\alpha$ be the image of α-conversion on name set *S*.

Definition (name capture avoidance) Let *S*, *T* be name sets,

$$S \setminus T = \begin{cases} (S-T) \cup [S \cap T]_\alpha & \textbf{if } S \cap T \neq \varnothing \\ S & \textbf{otherwise} \end{cases}$$

To be informal, *S**T* is just the result by performing α-conversion in *S* on those names shared between *S* and *T*.

As a side effect of rule *INPUT-ACT, OUTPUT-ACT,* two sets of variables are introduced: one is of free variables, denoted by $FV(\alpha)$, while another is of local variables, denoted by $BV(\alpha)$. Here a variable is said to be *free* if its scope is not specified yet. Fig 3 shows the definition of $FV(\alpha)$ and $BV(\alpha)$.

(1) $FV(\tau) = BV(\tau) = \varnothing$

(2) $FV(x(y)) = \{x\}, BV(x(y)) = \{y\}$

(3) $FV(\overline{x} < y >) = \{x, y\}, BV(\overline{x} < y >) = \varnothing$

Fig. 3. Definition of $FV(\alpha)$ and $BV(\alpha)$

After each translation step based on rules in Fig 2, name checking is necessary to avoid semantic errors in the Promela code. Let $FV(p)$ be the set of free variables in the Promela code for process *p*, $BV(p)$ be the set of local variables in the Promela code for process *p*. The definition for $FV(p)$ and $BV(p)$ is shown in Fig 4, where $V(\alpha) = FV(\alpha) \cup BV(\alpha)$, $V(p) = FV(p) \cup BV(p)$.

(4) $FV(0) = BV(0) = \varnothing$

(5) $FV(\alpha.p) = FV(\alpha) \cup (FV(p) - BV(\alpha)),$
$BV(\alpha.p) = BV(\alpha) \cup (BV(p) \setminus V(\alpha))$

(6) $FV(p+q) = FV(p \mid q) = FV(p) \cup FV(q),$
$BV(p+q) = BV(p \mid q) = (BV(p) \setminus FV(q)) \cup (BV(q) \setminus V(p))$

(7) $FV([x = y]p) = FV(p) \cup \{x, y\},$
$BV([x = y]p) = BV(p) \setminus \{x, y\}$

(8) $FV((\nu x)p) = FV(p) - \{x\},$
$BV((\nu x)p) = \{x\} \cup (BV(p) \setminus \{x\})$

(9) $FV(A(\tilde{x})) = FV(p) - \{x \mid x \ \texttt{in} \ \tilde{x}\},$
$BV(A(\tilde{x})) = BV(p) \cup \{x \mid x \ \texttt{in} \ \tilde{x}\}$

Fig. 4. Definition of $FV(p)$ and $BV(p)$

Two types of name checking are mandatory here. One is name binding, i.e. to determine the scope of a free variable. For example, if $FV(p) \cap BV(\alpha) \neq \varnothing$, assuming $x \in FV(p) \cap BV(\alpha)$ without loss of generality, x is free in p, however its scope is restricted to $\alpha.p$ at this time, namely $x \notin FV(\alpha.p), x \in BV(\alpha.p)$, as shown in (5). Such case can also be found in (9) and (10).

Another one is to avoid name capture by α-conversion. It works only on local variables. For example, if $BV(p) \cap V(\alpha) \neq \varnothing$, assuming $x \in BV(p) \cap V(\alpha)$ without loss of generality, although the name is same, however the scope is different. The scope of $x \in BV(p)$ is p while the one of $x \in V(\alpha)$ is $\alpha.p$. With the definition above, the former x has to be α-converted, as shown in (5). Such case can also be found in (6), (7), (9) and (10).

5 *pi2promela* – An Experimental Tool

An experimental tool *pi2promela* has been developed to implement the algorithm above. This section will address some detail on the implementation.

The input language of *pi2promela* borrows notions from Mobility Workbench to define process expressions. Within *pi2promela*, each process expression is interpreted as a syntax tree internally to gather enough information for code generation.

As far as name processing is concerned, each process expression is associated with a name table to record all names that occur in it. In order to make α-conversion convenient, an index table is built for each name *x* to keep track of its occurrence. In this way, when a name has to be α-converted, only the corresponding item in the name table should be modified without interference on its occurrence.

In addition, type consistency is also an important issue for the implementation because the mismatch on name type may cause confusion on code generation. For example, $x(y).\bar{x} < y, y > .0$ where the type of the first *x* is 1, but the type of the second one is 2. In such case, *pi2promela* will report a warning error and assume the type of *x* as the bigger one by default. Note that such example is acceptable in Mobility Workbench.

6 A Case Study

To show the effectiveness of the algorithm and the tool, a quantity of experiments has been performed, including those sample codes distributed with Mobility Workbench. The result of these experiments shows that *pi2promela* works well with specifications in π-calculus and the generated Promela codes are also acceptable in Spin.

One of these experiments is Bluetooth Service Discovery Protocol (SDP). Bluetooth is a short-range wireless communication technology. SDP is one of its protocols. It works at a high level, providing a means for applications to discover which services are available and to determine the characteristics of those available services. Note that the model given here is only an abbreviated one.

6.1 SDP Specification in π-calculus

SDP is consisted of two entities, SDP Client and SDP Server. A typical workflow of service discovery is:

1. A Service application registers its characteristics into SDP server so that it can be discovered in Bluetooth environment, as illustrated in Fig 5.

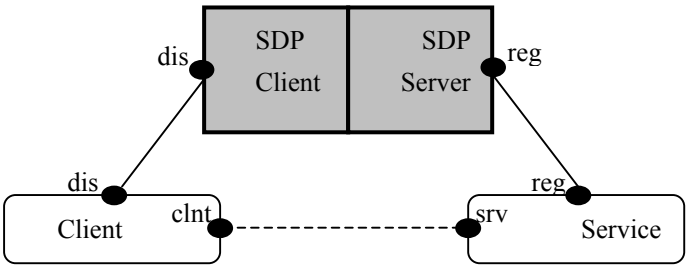

Fig. 5. A Work Model for SDP

$$Service(reg, sid) \overset{def}{=} (\nu\ srv)\ \overline{reg} < sid, srv > .Daemon(srv) \qquad \text{<1>}$$

Name *reg* represents the channel that is used for service registry. It is shared with *SDP* server. Name *sid* represents the identifier of *Service*, as one of its characteristics.

2. A client application issues a request for *SDP* client to discover services with specified constraints on their characteristics.

$$Client(dis, sid) \overset{def}{=} (\nu\ clnt)\ \overline{dis} < sid, clnt > .Wait(clnt) \qquad \text{<2>}$$

Name *dis* represents the channel to request for service discovery. It is also shared with SDP. Name *sid* represents a constraint that specifies the identifier of expected service.

3. *SDP* Client responses the client application with services that satisfy those constraints, or otherwise notification of no service available.
4. If success in step 3, the client application can request for services directly without the interference of SDP.

$$Wait(clnt) = clnt(srv).Call(srv) \qquad \text{<3>}$$

SDP specification can be formalized in π-calculus as

$$\begin{aligned} SDP(reg,dis) &= reg(sid,srv).(SDP(reg,dis) \mid Proxy(sid,srv)) \\ &\quad + dis(sid,c\ln t).(SDP(reg,dis) \mid \overline{sid} < clnt > .0) \\ Proxy(sid,srv) &= sid.(clnt).(Proxy(sid,srv) \mid \overline{clnt} < srv > .0) \end{aligned} \qquad \text{<4>}$$

6.2 Promela Code

Given the specification shown above, *pi2promela* generates the following Promela code (Only the segment for process *SDP* is demonstrated here). As shown in a verification experiment with Spin, this Promela code works well.

```
/* Agent SDP */
proctype SDP(chan reg, dis)
{
          chan sid = [0] of { chan };
          chan srv;
          chan _sid_0 = [0] of { chan };
          chan clnt;

          if
          :: reg?sid, srv;
           run SDP(reg, dis);
           run Proxy(sid, srv);
          :: dis?_sid_0, clnt;
           run SDP(reg, dis);
           _sid_0!clnt;
          fi
}
```

In this segment, the variable *_sid_0* comes from $dis.(sid,clnt).(...)$ in <4> by α-conversion.

7 Conclusion

The main contribution of this paper is to propose a rule-based algorithm that interprets specifications in π-calculus into Promela, so as to make it possible to verify them with Spin. An experimental tool, *pi2promela*, is also presented in this paper. The experimental result shows that the algorithm and *pi2promela* work well as expected. Anyway, *pi2promela* is still immature. The following aspects should be paid more attention to.

(1) Type Consistency. As shown above, the input language of Mobility Workbench is not strictly typed, which may cause confusion on code generation.

(2) Channel Initialization. In *pi2promela,* not all type information can be retrieved from process expressions. The variable *srv* and *clnt* in the sample Promela code above is only declared without initialization. Sometime it may cause a verification error of sending messages to or receiving messages from a channel that is not initialized yet. To make the generated Promela code works seamlessly with Spin, it's desirable to gather more information from process expressions.

Acknowledgment

The author would like to thank Prof. Wenhui Zhang for stimulating discussion on this paper.

References

1. Victor, B., Moller, F.: The Mobility Workbench: A Tool for the π-Calculus. In Dill, D. ed., Proceedings of the Conference on Computer-Aided Verification (CAV'94), LNCS 818:428-440, Springer-Verlag, 1994.
2. Dam, M.: Model Checking Mobile Process. Information and Computation, 129(1):35-51, 1996.
3. Holzmann, G. J.: The Model Checker Spin. IEEE Trans. on Software Engineering, 23(5):279-295, 1997.
4. Milner, R., Parrow, J., Walker, D.: A Calculus for Mobile Processes (Parts I/II). Information and Computation. 100:1-77, 1992.
5. Sangiorgi, D., Walker, D.: The π-Calculus - A Theory of Mobile Processes. Cambridge University Press, 2001.
6. Specification of the Bluetooth System: Core, Part E: Service Discovery Protocol. Bluetooth SIG, Version 1.1, 2001.
7. Spin – Formal Verfication. http://spinroot.com/spin/whatispin.html