1

A Formal Language for Composition

Markus Lumpe, Franz Achermann, Oscar Nierstrasz

Software Composition Group,
University of Berne,
Institute for Computer Science and Applied Mathematics
IAM, Neubrückstrasse 10, CH-3012 Bern, Switzerland.
{lumpe,acherman,oscar}@iam.unibe.ch

Abstract

A composition language based on a formal semantic foundation will facilitate precise specification of glue abstractions and compositions, and will support reasoning about their behaviour. The semantic foundation, however, must address a set of requirements like *encapsulation*, *objects as processes*, *components as abstractions*, *plug compatibility*, *a formal object model*, and *scalability*. In this work, we propose the $\pi\mathcal{L}$-calculus, an extension of the π-calculus, as a formal foundation for software composition, define a language in terms of it, and illustrate how this language can be used to plug components together.

1.1 Introduction

One of the key challenges for programming language designers today is to provide the tools that will allow software engineers to develop robust, flexible, distributed applications from plug-compatible software components [NM95]. Current object-oriented programming languages typically provide an ad hoc collection of mechanisms for constructing and composing objects, and they are based on ad hoc semantic foundations (if any at all) [Nie93]. A language for composing open systems, however, should be based on a rigorous formal foundation in which concurrency, communication, abstraction, and composition are primitives. In fact, the development of both the formal foundation and the language will be driven by a set of key requirements for a composition language [NM95]: *encapsulation*, *objects as processes*, *components as abstractions*, *plug compatibility*, *a formal object model*, and *scalability*.

Now, by defining the translation of higher-level language elements by means of a mapping to a mathematical calculus, we get non-ambiguous semantic interpretations of these constructs. Moreover, the language is open, since further language extensions can be handled the same way. Using this approach, we argue that it is possible to add newly defined higher-level abstractions and composition mechanisms. Finally, we are also able to reason about composition and check valid compositions.

We have previously used the π-calculus to model object composition and generic synchronization policies as an example for wrapping [LSN96, SL97, Var96]. The inherent problem, however, is the limited reusability and extensibility due to position dependent parameters. We propose $\pi\mathcal{L}$ as a variant of the π-calculus that is inherently extensible. In the polyadic π-calculus [Mil91], sender and receiver processes need to agree on the number of communicated names and the interpretation of each of these. This schema of contracts between two components is too rigid, since it often requires the propagation of extensions of one part to the other. This propagation, however, can be reduced in many cases, if we replace the polyadic communication of tuples by monadic communication of forms.

Dami has tackled a similar problem in the context of the λ-calculus, and has proposed λN [Dam94, Dam98], a calculus in which parameters are identified by names rather than positions. The resulting flexibility and extensibility can also be seen in HTML forms, whose fields are encoded as named (rather than positional) parameters in URLs, and in Python [vR96], where functions can be defined to take arguments by keywords.

In the next section, we informally present the $\pi\mathcal{L}$-calculus. In Section 1.3 we define Piccola by extending a core language step-by-step with higher-level language constructs that simplify programming with component agents. In the Sections 1.4 and 1.5 we show how to implement a composition mechanism and some glue paradigms. We conclude with some remarks about future work and directions.

1.2 The $\pi\mathcal{L}$-calculus

In this section we introduce the $\pi\mathcal{L}$-calculus [Lum99], an offspring of an asynchronous fragment of the (polyadic) π-calculus [Mil91, MPW92]. The asynchronous sublanguage was proposed first by Boudol [Bou92] and Honda and Tokoro [HT92]. Sangiorgi [San95] extended the proposal by allowing polyadic communication.

1.2.1 Syntax

In the $\pi\mathcal{L}$-calculus we replace the communication of names or tuples of names by communication of so-called *forms*, a special notion of extensible records. More precisely, the $\pi\mathcal{L}$-calculus is an offspring of the asynchronous π-calculus, where polyadic communication is replaced by monadic communication of forms.

We use a, b, c to range over the set $\mathcal{N}$ of names. As in the π-calculus literature, we use the words "name", "port", and "channel" interchangeably. Unlike the π-calculus where names are both subject and object of a communication in a sender or receiver, in the $\pi\mathcal{L}$-calculus names are only used as subject of a communication. The role of the object of a communication is taken by forms. Forms are finite mappings from an infinite set $\mathcal{L}$ of labels to an infinite set $\mathcal{N}^+ = \mathcal{N} \cup \{\mathcal{E}\}$, the set of names extended by $\mathcal{E}$ that denotes the empty binding. We use x, y, z to range over $\mathcal{N}^+$, the extended set of names, F, G, H to range over forms, X, Y, Z to range over form variables, and l, m, n to range over $\mathcal{L}$. The syntax for forms is defined as follows:

$$
\begin{array}{rclll}
F & ::= & \mathcal{E} & & \textit{empty binding} \\
  & | & X & & \textit{form variable} \\
  & | & F\langle l{=}V \rangle & & \textit{binding extension} \\
  & | & FX & & \textit{polymorphic extension} \\
\textit{where} & & & & \\
V & ::= & x & & \textit{simple name} \\
  & | & X_l & & \textit{name projection}
\end{array}
$$

The most notable elements of forms are *form variables* and *name projections*. *Form variables* are used for both as formal agent parameter in an input agent and as *polymorphic placeholders* in the *polymorphic extension*.

Projections denote locations of names in the $\pi\mathcal{L}$-calculus. A projection X_l has to be read as selection of the parameter named by l.

To formulate how projection works, we need the notion of *variables of a form* and *closed forms*.

Definition 1.2.1 (Variables of a form) *The set of variables of a form F, written $\mathcal{V}(F)$, is defined as:*

$$
\begin{array}{rcl}
\mathcal{V}(\mathcal{E}) & = & \emptyset \\
\mathcal{V}(X) & = & \{X\} \\
\mathcal{V}(F\langle l{=}x \rangle) & = & \mathcal{V}(F) \\
\mathcal{V}(F\langle l{=}X_k \rangle) = \mathcal{V}(FX) & = & \{X\} \cup \mathcal{V}(F)
\end{array}
$$

Definition 1.2.2 (Closed form) *We say that a form F is closed if it does not contain any form variable, so that $\mathcal{V}(F) = \emptyset$.*

Now we can define the notion of *name projection* as follows.

Definition 1.2.3 (Name projection) *If a form F is closed, then the application of a label $l \in \mathcal{L}$ to form F (mapping from $\mathcal{L}$ to $\mathcal{N}^+$), written F_l, is called name projection and is defined as:*

$$\begin{aligned}
\mathcal{E}_l &= \mathcal{E} \\
(F\langle l{=}x\rangle)_l &= x \\
(F\langle m{=}x\rangle)_l &= F_l \quad \textit{if } m \neq l
\end{aligned}$$

If a binding is defined for label l then F_l yields x, otherwise it yields the empty binding ($\mathcal{E}$). The reader should note that projection F_l can yield the empty binding $\mathcal{E}$ either if $l \notin \mathcal{L}(F)$ or $x = \mathcal{E}$ since $x \in \mathcal{N}^+$. A form may have multiple bindings for label l. In this case F_l extracts the rightmost binding.

The class A of $\pi\mathcal{L}$-calculus agents is built using the operators of inaction, input prefix, output, parallel composition, restriction, and replication. We use A, B, C to range over the class of agents. The syntax for agents is defined as follows:

$$\begin{array}{llll}
A & ::= & \mathbf{0} & \textit{inactive agent} \\
& | & A \mid A & \textit{parallel composition} \\
& | & !V(X).A & \textit{replicated input} \\
& | & (\nu\ a)A & \textit{restriction} \\
& | & V(X).A & \textit{input \ (receive form in } X) \\
& | & \overline{V}(F) & \textit{output \ (send form } F)
\end{array}$$

$\mathbf{0}$ is the inactive agent. An input-prefixed agent $V(X).A$ waits for a form F to be sent along the channel denoted by value V and then behaves like $A\{F/X\}$, where $\{F/X\}$ is the substitution of all form variables X with form F. An output $\overline{V}(F)$ emits a form F along the channel denoted by value V. Unlike in the π-calculus, the value V in both the input prefix and the output particle can be either a simple name or a projection. Parallel composition runs two agents in parallel. The restriction $(\nu\ a)A$ makes name a local to A, i.e., it creates a *fresh* name a with scope A. A replication $!V(X).A$ stands for a countably infinite number of copies of $V(X).A$ in parallel.

1.2.2 Terminologies and notions

Both the input prefix and the restriction operator are binders for names in the π-calculus. In the $\pi\mathcal{L}$-calculus, however, only the operator $(\nu\ a)A$ acts as binder for names occurring free in an agent. In $\pi\mathcal{L}$ the input prefix $V(X)$ is the binding operator for form variables. We use $\mathrm{fn}(A)$ and $\mathrm{bn}(A)$ to denote the set of *free* and *bound names* of an agent and $\mathrm{fv}(A)$ and $\mathrm{bv}(A)$ to denote the set of *free* and *bound form variables* of an agent, respectively. Similarly, $\mathrm{n}(A)$ and $\mathrm{v}(A)$ stand for all names and variables in A, respectively.

The sets $\mathrm{fv}(A)$ and $\mathrm{bv}(A)$ give rise the the following definition.

Definition 1.2.4 (Closed agent) *We say that an agent A is closed if it does not contain any free form variables, so that* $\mathrm{fv}(A) = \emptyset$.

In contrast to the π-calculus, we strictly distinguish between constants and variables. In the $\pi\mathcal{L}$-calculus names are always constant, i.e., names are constant locations and are not subject of substitution (α-conversion is still possible). On the other side, *projections* denote variables in the $\pi\mathcal{L}$-calculus.

We write $A\{F/X\}$ for the substitution of all free occurrences of form variable X with form F in A. Substitutions have precedence over the operators of the language.

Finally, we adopt the usual convention of writing $x(X)$ when we mean $x(X).\mathbf{0}$. Additionally, an agent $\overline{x}(\mathcal{E})$ sending an empty form can just be written $\overline{x}$, a form $\langle\mathcal{E}\langle l{=}x\rangle\rangle$ is just written $\langle l{=}x\rangle$, and we abbreviate $(\nu\ x)(\nu\ y)A$ with $(\nu\ x,y)A$ and $(\nu\ x_1)...(\nu\ x_n)A$ with $(\nu\ \tilde{x})A$, respectively.

1.2.3 Operational semantics

The operational semantics is given using the reduction system technique proposed by Milner [Mil90, Mil91]. In the reduction system technique, axioms for a structural congruence relation are introduced prior the definition of the reduction relation. Basically, this allows us to separate the laws which govern the neighbourhood relation among agents for the rules that specify their interaction. Furthermore, this simplifies the presentation of the reduction relation by reducing the number of cases that we have to consider.

Definition 1.2.5 *The structural congruence relation, $\equiv$, is the smallest congruence relation over agents that satisfies the following axioms:*

(1) $A\mid B \equiv B\mid A$, $(A\mid B)\mid C \equiv A\mid(B\mid C)$, $A\mid\mathbf{0} \equiv A$;

(2) $(\nu\ a)\mathbf{0} \equiv \mathbf{0}$, $(\nu\ a)(\nu\ b)A \equiv (\nu\ b)(\nu\ a)A$;

(3) $(\nu\ a)A\mid B \equiv (\nu\ a)(A\mid B)$, *if a not free in B*;

(4) $!V(X).A \equiv V(X).A\mid !V(X).A$;

(5) $\mathcal{E}(X).A \equiv \mathbf{0}$, $\overline{\mathcal{E}}(F) \equiv \mathbf{0}$.

The axioms (1)–(4) are standard and are the same as for the π-calculus. The only "new" axiom is (5), which defines the behaviour if an *empty binding* appears in subject position of the leftmost prefix of an agent. For example, if we have the agent $a(X).X_l(Y)$ and this agent receives along channel a the form $\langle m{=}V_1\rangle\langle n{=}V_2\rangle$ then $(X_l(Y))\{\langle m{=}V_1\rangle\langle n{=}V_2\rangle/X\}$ yields $\mathcal{E}(Y)$ since label l is not defined in the received form. In this case the agent is identical with the *inactive agent*. This means that a system containing such an agent may reach a deadlock. In general, if the name $\mathcal{E}$ occurs as subject in the leftmost prefix of an agent, this may be interpreted as a run-time error.

The reduction system describes the reduction of $\pi\mathcal{L}$-terms. In fact, the *reduction* rules define the interaction of $\pi\mathcal{L}$-agents. Note, however, that the reduction relation is only defined for closed agents (i.e., $\mathrm{fv}(A)=\emptyset$).

Definition 1.2.6 *Let A, B two $\pi\mathcal{L}$-agents and $\mathrm{fv}(A|B) = \emptyset$. Then the one-step reduction $A \longrightarrow B$ is the least relation closed under the following rules:*

$$\text{PAR: } \frac{A \longrightarrow A'}{A \mid B \longrightarrow A' \mid B} \qquad\qquad \text{RES: } \frac{A \longrightarrow A'}{(\nu\ a)A \longrightarrow (\nu\ a)A'}$$

$$\text{COM: } a(X).A \mid \overline{a}.(F) \longrightarrow A\{F/X\}$$

$$\text{STRUCT: } \frac{A \equiv A' \quad A' \longrightarrow B' \quad B' \equiv B}{A \longrightarrow B}$$

The first two rules state that we can reduce under both parallel composition and restriction. (The symmetric rule for parallel composition is redundant, because of the use of structural congruence.)

The communication rule takes two agents which are willing to communicate on the channel a, and substitutes all form variables X with form F in A. The communication rule is the only rule which directly reduces a $\pi\mathcal{L}$-term. A reduction is not allowed underneath a input prefix. Furthermore, closed agents evolve to closed agents [Lum99].

The communication assumes that agents are in a particular format. The structural congruence rule allows us to rewrite agents so that they have the correct format for the communication rules.

1.3 Towards a composition language

In this section we develop a first version of the composition language PICCOLA. In order to define PICCOLA, we use the scheme that has been successfully used for the definition of PICT [Pie95], i.e., we define a language core and extend it step by step with higher-level syntactic forms that are translated into the core language.

1.3.1 The core language

The core is presented in Table 1.1. For describing the syntax, we rely on a meta notation similar to the Backus-Naur Form that is commonly employed for language definitions. Keywords and symbolic constants appear inside quotes. Optional expressions are enclosed in square brackets. Furthermore, we reuse the PICT convention to denote the different types of agents using special symbols. We use ! to stand for output prefixes, ? to stand for input agents, and ?∗ to stand for replicated input agents.

The reader should note that a parallel composition of agents extends to the right as far as possible, i.e., the parallel agent is right associative. Therefore, if an input-prefixed agent is built using parallel composition of subagents, these subagents have

Declarations	::=	*Declaration* [*Declarations*]
Declaration	::=	'new' *NameList* 'run' *Agent*
Agent	::=	*PrimaryAgent* ['|' *Agent*]
PrimaryAgent	::=	'null' *Location* '!' *Form* *Location* '?' '(' [*Variable*] ')' 'do' *Agent* *Location* '?∗' '(' [*Variable*] ')' 'do' *Agent* 'let' *Declarations* 'in' *Agent* 'end' 'if' *BoolExpression* 'then' *Agent* ['else' *Agent*] 'end' '(' *Agent* ')'
NameList	::=	*Name* [, *NameList*]
BoolExpression	::=	*Value Built-in-BoolOperator Value*
Location	::=	*Name* *Variable* '.' *Label*
Form	::=	'<' [*FormElementList*] '>' *Variable*
FormElementList	::=	*FormElement* [',' *FormElementList*]
FormElement	::=	*Variable* *Label* '=' *Value*
Value	::=	*Location* *String* *Number*

Table 1.1. *The core of* PICCOLA.

to be written in parentheses (e.g., the agent $(\texttt{a?(X) do b!} <\texttt{Y}>) \mid \texttt{c!} <\texttt{Z}>$ is different from the agent $\texttt{a?(X) do b!} <\texttt{Y}> \mid \texttt{c!} <\texttt{Z}>$).

In PICCOLA the syntax for forms has been simplified. A form $\texttt{X}$ extended with a sequence of bindings $< l_1 = V_1 >< l_2 = V_2 >$ is now written as a list of form elements: $< X, l_1 = V_1, l_2 = V_2 >$.

In the following, we iteratively extend the syntax by defining higher-level abstractions. These higher-level abstractions come with a set of small examples that will motivate the newly introduced concepts or aspects of the higher-level language. Throughout this section, we use $[\![\cdot]\!]$ as interpretation of a higher-level construct denoted by $\cdot$ in terms of the core language. The complete syntax definition of PICCOLA is given in the Appendix.

1.3.2 Procedures

Assume we have implemented a simple person database service that is located at a global channel `lookup`. In order to query information about a person, we have to send a form containing the labels `name` and `result`. The label `name` binds a string denoting the name of the person while the label `result` maps a channel along which the query result is returned. In the following example, we query information about a person "Smith" and display the received information on the screen using a built-in display agent located at channel `print`.

```
new result                                          // create reply channel
run   lookup!<name="Smith",result=result>           // invoke query
    | result?(Info) do                              // wait for information
                    print!Info                      // print information
```

This definition, however, can immediately be simplified, because form variables can host arbitrary forms. In fact, instead using a newly created reply channel `result`, we can directly bind the channel `print` to label `result`, such that the query result is directly passed to the display agent. The simplified definition is shown in the following:

```
run lookup!<name="Smith",result=print>
```

Now, we would like to provide this behaviour as a single service that is parameterized with the person string to be queried. Therefore, we define a replicated input agent that listens at channel `printPerson` waiting for a form containing at least a label `name` and displays the corresponding information on the screen.

```
new printPerson                                       // service channel
run printPerson?*(ArgForm) do lookup!<ArgForm,result=print>    // perform lookup
```

In order to invoke this service, we can send a form to channel `printPerson` that must contain at least the label `name`:

```
printPerson!<name="Smith">
```

In fact, the above service definition is a parameterized agent abstraction. The structure of the definition is so common that we provide a new element for the syntactic domain *Declaration*; we extend the language with a *procedure declaration*:

$$\llbracket \text{ 'procedure' } Name \text{ '(' } [\ Variable\ ] \text{ ')' 'do' } Agent \rrbracket$$
$$= \text{ 'new' } Name$$
$$\text{ 'run' } Name?*(\ [\ Variable\ ]\ ) \text{ 'do' } Agent$$

Having the procedure declaration available, we also want to have a convenient way to invoke procedures. Therefore, we add an *application*† as additional syntactic form to primary agents:

$$\llbracket\ Location \text{ '(' } Form \text{ ')' } \rrbracket \quad = \quad Location!Form$$

---

† The reader should note that we allow arbitrary locations to denote procedure names. For example, an agent can get access to a procedure by receiving a form that contains a binding which maps to the procedure's name.

Now, assuming that the service `lookup` was also defined as procedure, the above `printPerson` service can be rewritten using the new syntax:

```
procedure printPerson(ArgForm) do lookup(<result=print,ArgForm>)
```

The reader should note that in the above declaration we have changed the order of the elements in the form used as argument for the procedure *lookup*. In fact, using this scheme we can define so-called *default arguments*. In case of the procedure *lookup* we have one default argument represented by the binding *result = print* which guarantees that the result of *lookup* is passed to an agent located at channel *print*. To override this default behaviour we can simply add a new binding for label *result* to the argument form of the procedure *printPerson*. For example, if we have a user-defined print agent located at channel *printnew*, we can redirect the output produced by procedure *lookup* with the following invocation of *printPerson*:

```
printPerson(<ArgForm,result=printnew>)
```

This example uses polymorphic extension which is technically spoken *asymmetric record concatenation* [CM94] and which is a feature that is extremely useful to model compositional abstractions [Sch99].

1.3.3 Value declaration

So far, new form variables can only be introduced by input-prefixed agents or procedures. However, it is often convenient to make some variables globally accessible, such that they appear "free" somewhere in the program text and provide a value throughout the rest of that program. To define such variables and to assign them values, we introduce a *value declaration* which is defined as follows:

$$'\mathbf{let}' \; [\![\; '\mathbf{value}' \; \textit{Variable} \; '=' \; \textit{Form} \;]\!] \; '\mathbf{in}' \; \textit{Agent} \; '\mathbf{end}'$$
$$= \; '\mathbf{let}' \; '\mathbf{new}' \; r \; '\mathbf{in}' \; r!\textit{Form} \; | \; r?(\; \textit{Variable} \;) \; '\mathbf{do}' \; \textit{Agent} \; '\mathbf{end}'$$

1.3.4 Complex forms

All form expressions that we have encountered so far have been built up in a simple way, using just variables and bindings from labels to core values. If we, however, want to define updatable data structures, we need a packing technique that allows us to define data and operations over them in one syntactic construct. Therefore, we extend the syntax of forms and allow local declarations and call these construct *complex forms*:

$$\textit{Location}! \; [\![\; '\mathbf{let}' \; \textit{Declaration} \; '\mathbf{in}' \; \textit{Form} \; '\mathbf{end}' \;]\!]$$
$$= \; '\mathbf{let}' \; \textit{Declaration} \; '\mathbf{in}' \; \textit{Location}!\textit{Form} \; '\mathbf{end}'$$

For example, this syntactic construct together with the value declaration allows us to define a storage cell as follows:

```
value AStorageCell =
  let
    new cell
    run cell!<>
    procedure Read(Args) do cell?(Val) do (Args.result!Val | cell!Val)
    procedure Update(Val) do cell?(OldVal) do cell!Val
  in <Read=Read,Update=Update> end
```

In fact, this declaration defines a simple object **AStorageCell** that maintains a private instance variable denoted by channel *cell* and provides two member functions **Read** and **Update** to get and set the contents of the object **AStorageCell**, respectively. The reader should note that this object encoding uses asynchronous method invocation. In the next section, we define function abstractions. Using functions to implement **Read** and **Update** will allow us to execute these methods synchronously.

1.3.5 Functions

Each time we have applied a form in an output agent or a call expression, we have used "static" forms. To get a greater flexibility and even a more compact specification, we add abstractions for "dynamic" forms. The term "dynamic" means that forms are either generated by agents that act as functions or that have themselves dynamic elements. For functions, we extend *Declarations* as follows:

$$\llbracket \text{ 'function' } Name \text{ '(' } [\text{ } Variable \text{ }] \text{ ')' '=' } Form \rrbracket$$
$$= \quad \text{'new' } Name$$
$$\text{'run' } Name?*(\text{ X }) \text{ 'do' } \texttt{X.result!}Form$$

with variable **X** being either *Variable* or a fresh wildcard if *Variable* has been omitted in the function declaration. The reader should note that by convention the label **result** maps the name (channel) along which the function result is sent.

In order to call a function, we also add the syntactic domain *Application* to forms (see Section 1.3.2). However, in contrast to procedures, the translation of functions uses **result** as default label to return the function result. Therefore, a function call

```
AFun( AForm )
```

is transformed into

```
AFun( <AForm,result=AResultChannel> )
```

where **AResultChannel** denotes the actual result channel to be used.

The result of a function may be available before the complete function body has been processed. To stress this fact, we extend the syntax domain *Form* with a *return expression*:

$$\llbracket \text{ 'return' } Form \rrbracket = \texttt{X.result!}Form$$

where X denotes the actual formal function parameter.

This expression, however, is only allowed to be used within a function declaration, because the return expression is translated to an output process that sends the form expression along a channel mapped by label `result`. If a return expression is specified within a procedure it will evaluate to the `null` agent, because the necessary label `result` is not defined.

1.3.6 Nested forms

Up to now, forms are flat values, i.e., there is no support to add an additional structure to forms. However, it is often necessary to keep things separated. For example, if a service expects both a channel and a value, we have to send both in one form which is typically written like

```
<channel=cname,AFormValue>
```

Unfortunately, this definition merges the channel and the form value, such that the original structure information is lost. Therefore, we introduce so-called *nested forms*, i.e., we extend *Value* to denote also forms:

$$\begin{aligned} & \text{'<'}\ Label\ \text{'='}\ [\![\ Value ::= Form\ ]\!]\ \text{'>'} \\ = \ \ & \text{'\textbf{let}'}\ [\![\ \text{'\textbf{function}'}\ \texttt{f}\ (\ \texttt{X}\ ) = Form\ ]\!]\ \text{'\textbf{in}'}\ \text{'<'}\ Label\ \text{'='}\ \texttt{f}\ \text{'>'}\ \text{'\textbf{end}'} \end{aligned}$$

with `f` being a fresh function name.

The above form can now be rewritten as follows:

```
<channel=cname,form=AFormValue>
```

Nested form add, however, one level of indirection. Therefore, to access the form bound by label $form$ we need to use function application. For example, if form variable X denotes the above form then the application $X.form()$ yields the form mapped by label $form$.

1.3.7 Active forms

Active forms are forms that contain active elements, i.e., they have function or procedure specifications or function calls in place of bindings. This concept allows us to define form expressions that can act as objects. Therefore, we add *procedures*, *functions*, and *function call* to the syntax category of *FormElement*:

'<' ⟦ '**procedure**' *Name* '(' [*Variable*] ')' '**do**' *Agent* ⟧ '>'
= '**let**' ⟦ '**procedure**' *Name* '(' [*Variable*] ')' '**do**' *Agent* ⟧
        '**in**' '<' *Name* '=' *Name* '>' '**end**'

'<' ⟦ '**function**' *Name* '(' [*Variable*] ')' '=' *Form* ⟧ '>'
= '**let**' ⟦ '**function**' *Name* '(' [*Variable*] ')' '=' *Form* ⟧
        '**in**' '<' *Name* '=' *Name* '>' '**end**'

'<' ⟦ *Location* '(' *Form* ')' ⟧ '>'
= '**let**' ⟦ '**value**' X '=' *Location* '(' *Form* ')' '**in**' '<' X '>' '**end**'

The functions and procedures are translated using the scheme for nested forms. Function calls are transform into a complex form using X as fresh variable.

However, we allow not only form elements to be active, but also forms. Therefore, we extend forms with *conditionals*.

'<' ⟦ '**if**' *BoolExpression* '**then**' $Form_1$ ['**else**' $Form_2$] '**end**' ⟧ '>'
= '<' ⟦ PrIf '(' bool '=' ⟦ *BoolExpression* ⟧','
            t '=' $Form_1$ ',' f '=' $Form_2$ ')' ⟧ '>'

Roughly spoken, conditional forms are transformed into a function call. The function PrIf is a builtin function (i.e., it is part of the runtime system of Piccola) that expects a form that contains the labels bool, t, and f. If the else part has been omitted, then the translation replaces $Form_2$ with the empty form. Further information on the complete treatment of conditional forms can be found in [Lum99].

1.3.8 Sequencing

In fact, the form <> can be considered as a *continuation signal*, i.e., it carries no information but tells the calling agent that its request has been satisfied. Using this signal, we can synchronize parallel running agents. Therefore, we provide a convenient syntax to specify "invoke operation, wait for a signal as a result, and continue":

⟦ *Form* ';' *PrimaryAgent* ⟧
= '**let**' '**value**' X '=' *Form* '**in**' *PrimaryAgent* '**end**'
⟦ $Form_1$ ';' $Form_2$ ⟧
= '**let**' '**value**' X '=' $Form_1$ '**in**' $Form_2$ '**end**'

The first abstraction denotes an agent while the latter denotes a form. In both, however, the lefthand-side form (the variable X is fresh) is evaluated before the righthand-side agent or form becomes active. This means that the value of the lefthand-side form is lost; we are only interested in the fact that this form has been evaluated, so that it is now safe to proceed.

1.3.9 External services

In order to incorporate components that have not been developed in PICCOLA, we provide an *external declaration*:

$$Declaration \quad ::= \quad \text{'\textbf{extern}'} \; String \; Name$$

In this declaration *String* denotes a Java class that provides the interface implementation to the external service while *Name* can be an arbitrarily chosen name that maps the Java class to a channel name. For example, a print service located at `print` is declared as follows:

```
extern "Piccola.builtin.print" print
```

The PICCOLA runtime system provides several Java classes to map external components. However, a detailed description of them is beyond the scope of this work.

1.3.10 Composition scripts

Finally, we add the facility to define separate modules or composition scripts. A script is itself a component, i.e, it can be composed with other composition scripts.

Composition scripts can be separately compiled. The result is stored in a composition library. Composition scripts can load other scripts that have been previously compiled. This implies, however, that composition scripts cannot have circular dependencies which is, in our opinion, the most natural way to support *black-box* reuse of components.

A composition script is defined as follows:

$$Script \quad ::= \quad \text{'\textbf{module}'} \; ModuleName \; [\; Imports \;] \; Declarations \; [\; Main \;]$$

$$Imports \quad ::= \quad \text{'\textbf{load}'} \; NameList \; \text{';'}$$

$$Main \quad ::= \quad \text{'\textbf{main}'} \; Agent$$

Inspired by Python [Lut96], the main declaration specifies the agent that has to be started in the PICCOLA environment when the script is executed at the top level. Main declarations of imported scripts are ignored.

1.4 An object model

In this section we show how objects can be implemented in PICCOLA. Our development is based on the basic object model of Pierce and Turner [PT95]. Adapting this model, we specify an object as a composition of parallel running agents which are bundled with some restricted locations that serve as the state of the object.

Using the higher-level syntax, a storage cell can be implemented as follows:

```
load Blackboard;

function newStorageCell( X ) =
    let
        value cell = newBlackboard()
    in
        cell.Write( <val = X.init> );
        <
            function Update( Args ) = cell.Remove(); cell.Write( <val = Args> ),
            function Read() = cell.Read()
        >
    end
```

In fact, the function `newStorageCell` implements an object generator. The instance variable `cell`, in contrast to the example in Section 1.3.4, is implemented using a Linda-like blackboard abstraction [CG89] (imported by the load statement). Since each new storage cell has its own restricted blackboard we guarantee the invariant that a storage cell can store only a single value.

Now we are ready to define a class abstraction. This abstraction implements a simple metaobject protocol that allows us to create objects. Basically, the definition of a class abstraction follows the scheme similar to those found in [AC96] and [SL97]. Furthermore, the class abstraction can be solely implemented in Piccola. There is no need to extend the language†.

The object model defined by the abstraction `Class` that supports single inheritance and private and protected features. This abstraction provides three functions: `newInstance`, `selfbinder`, and `allocate`. The function `newInstance` is responsible for creating new object instances while the others are metaobject operations that establish inheritance and the correct binding of `self`.

```
function Class( Body ) =
    let
        value super = Body.super()

        function allocate ( Init ) =
            Body.allMethods( <super.allocate( Init ), init = Init> )

        function selfbinder( Self ) =
            Body.public( <super = super.selfbinder( Self ), self = Self> )

    in
        <
            function newInstance( Init ) = selfbinder( <allocate(Init)> ),
            selfbinder = selfbinder,
            allocate = allocate
        >
    end
```

The (function) abstraction `Class` expects a form that has three labels: `super`, `allMethods` and `public`. In fact, all labels denote functions. The function `super` returns the actual super object.

The function `allMethods` returns a "pre-object", i.e., `allMethods` extends its

---

† The reader should note that future extensions of Piccola will contain builtin language features for classes and objects.

argument form (i.e, the super object) with all newly defined class features (methods and instance variables).

The function `public` is built in the same way as `allMethods`. The function `public` expects a form that maps `super` and `self`. In fact, the values bound to `super` and `self` are also forms. Basically, the function `public` returns the public interface of an object and establishes the correct binding of `self`.

Finally, we have to define the root of the inheritance tree. The root of a class hierarchy is the class `Object`. The class metaobject for `Object` is defined as follows:

```
value Object =
    <
        function newInstance() = <>,
        function selfbinder() = <>,
        function allocate() = <>
    >
```

Using the function `Class`, a metaobject for a storage cell can be implemented as follows. A storage cell class inherits from class `Object`. The function `allMethods` returns a "pre-object" that extends `super` with the pre-methods `preSet` and `preGet` and a binding for the protected feature `blackboard`. Finally, the function `public` returns a fresh storage cell object.

```
value StorageCellClass = Class(
    <
      super = Object,

      function allMethods( Super ) =
        let
          value Init = Super.init()
          value Blackboard = newBlackboard()
        in
          Blackboard.Write( <val = Init> );
          <
            Super,
            function preSet( Args ) =
              let
                value B = Args.self().blackboard()
              in
                B.Remove(); B.Write( Args )
              end,
            function preGet( Args ) = Args.self().blackboard().Read(),
            blackboard = Blackboard
         >
        end,

      function public( Pre ) =
        <
          Pre.super(),
          function set( A ) = Pre.self().preSet( <Pre, val = A> ),
          function get() = Pre.self().preGet( Pre )
        >
   > )
```

The abstraction `Class` illustrates how a complex composition mechanism can be defined in Piccola. Compared with the object encodings in the polyadic π-calculus [LSN96, SL97], the new formulation of inheritance is much more compact.

Furthermore, in [LSN96, SL97] we had to explicitly name all methods in the process of binding self, even those methods that were not modified in the subclass.

1.5 Generic Wrapper

In the previous section we illustrated how to define a composition mechanism in Piccola. In this section we present the definition of a wrapping mechanism. More precisely, we show how a Readers-Writers synchronization policy can be defined using a generic wrapper.

The Readers-Writers policy is a family of concurrency control designs that apply when any number or readers can be executing simultaneously as long as there are no writers, but writers require exclusive access.

It is relatively straightforward to implement a Readers-Writers policy for a given set of reader and writer methods. However, it is problematic to implement it generically, but not impossible. For example, Doug Lea [Lea96] has given a generic implementation of a Readers-Writers policy using an abstract Java class. This implementation defines pre- and post-methods for both reader and writer methods. Using a template method `read_()`, the method `read()` executes `beforeRead()`, possibly blocking until the reader is allowed to enter its critical region. The critical region is implemented in `read_()` and must be provided by a subclass. Then a method `afterRead()` does some cleanup. By subclassing, a programmer provides its own implementation of `read_()`. Unfortunately, it is not possible to subclass the read method so that it can take arguments. However, it is possible to add new methods, but then it is the programmer's responsibility to correctly call `beforeRead()` and `afterRead()` for each reading method and analogously for writer methods.

Using forms we can define a generic wrapper which wraps any service with arbitrary arguments with pre- and post-methods. A wrapped service is created using the following steps:

- The pre-function `pre` is invoked with the arguments for the service. The returned form is stored in `PreService`.
- The service is invoked with the arguments and its result is returned.
- Finally, the post-function `post` is invoked using `PreService` (the result of the pre-function) as argument.

We define this wrapping operation by the function `wrap` as follows:

```
function wrap( Service ) =
   < function val( X ) =
       let
           value PreService = Service.pre( X )
       in
           return Service.val( X ); Service.post( PreService )
       end >
```

Now we want to build a Readers-Writers policy wrapper. This wrapper enables

the wrapping of arbitrary agents by either declaring them as `reader` or `writer`. A agent wrapped with `reader` (`writer`) becomes then a reader (writer) agent. They apply to the following constraints:

- When no writer agent is active, an arbitrary number of reader agents can be concurrently active,
- There is at most one writer active. When a writer is active, no active reader is allowed.

A Readers-Writers policy can now be built up using four agents. These agents work on two private resources, one with a semaphore for the writers, and the other containing the number of currently active reader agents. Initially, the semaphore allows to enter, and the value `0` is written onto the blackboard `count` since no reader is active.

The function `preRead` removes the count. If the count is zero, then the writer semaphore is grabbed. Finally, an incremented count is written back to the channel keeping track that one reader-agent is active.

The function `postRead` removes the count from the blackboard. If the count equals one, then the last reader has finished and the writer semaphore is released. Finally, the decremented count is written back to the blackboard.

The function `preWrite` grabs the writer semaphore and the function `postWrite` releases the semaphore.

Creating a reader writer wrapper then means to define two wrappers `reader` and `writer` which wrap an arbitrary agent using the corresponding pre- and post agents:

```
load Blackboard, Semaphore;

function newReaderWriter() =
  let
    value Writer = newSemaphore()
    value count = newBlackBoard()
    run count.Write( <val = <val = 0>> );

    function preRead() =
      let
        value n = count.Remove()
      in
        if n.val = 0 then Writer.p() end; count.Write( <val = <inc( n )>> )
      end

    function postRead() =
      let
        value n = count.Remove()
      in
        if n.val = 1 then Writer.v() end; count.Write( <val = dec( n )> )
      end

    function preWrite() = Writer.p()
    function postWrite() = Writer.v()
  in
    < function reader( X ) = wrap( <X, pre = preR, post = postR> ),
      function writer( X ) = wrap( <X, pre = preW, post = postW> ) >
  end
```

In this implementation the `count` blackboard acts as lock for the functions `preRead` and `postRead`. However, it is important to note that we keep access to the pre- and post functions restricted to the policy. A malicious agent cannot call a post function in isolation.

As an example of using the policy we assume a concurrent queue with methods `put`, `get`, and `getcount`. This queue is fully concurrent meaning that `put` and `get` can be executed concurrently. In order to add a `clone` method we can wrap the whole queue with a Readers-Writers policy, and declare the original methods as readers and the `clone` method as a writer. This way we can guarantee that the clone function has a locked queue, can safely use the count, and use put and get methods to create a clone of the queue. In the following we only show the wrapping part of the definition of a "clonable queue":

```
function newCloneQueue() =
  let
    value queue = newQueue()
    value policy = newReaderWriter()

    function clone() = ...                     // clone functionality
  in
    < put = policy.reader( <val = queue.put> ).val,
      get = policy.reader( <val = queue.get> ).val,
      getCount = policy.reader( <val = queue.getCount> ).val,
      clone = policy.writer( <val = clone> ).val  >
  end
```

1.6 Conclusion and future work

We have presented the $\pi\mathcal{L}$-calculus and a first design of the composition language Piccola based on the $\pi\mathcal{L}$-calculus. Furthermore, Piccola provides support such that (i) an application may run on a variety of hardware and software platforms (the runtime system is built in Java), and (ii) open applications may be inherently concurrent and distributed (Piccola's formal semantics is based on a process calculus).

We have presented some Piccola component scripts that facilitate the specification and modeling of generic glue abstractions for adaptation and composition of software components. Although our examples are neither exhaustive nor canonical, we think they represent the essence of glue code, which is typically concerned with (i) adapting interfaces and behaviour of software components, and (ii) composing or connecting components to achieve a combined behaviour.

Ultimately we are targeting the development of open, hence distributed systems [NSL96]. Given the ad hoc way in which the development of open systems is supported in existing languages, we have identified the need for composing software from predefined, plug-compatible software components. The overall goal of our work, and hence the development of Piccola, is the development of a formal model for software composition, integrating a black-box framework for modelling objects

and components, and an executable composition language for specifying components and applications as compositions of software components.

We are planning a further development of the $\pi\mathcal{L}$-calculus to address various other practical problems. For example: Should labels be first-class values? Generic glue code may need to learn about new labels representing extended interfaces of components. Do first-class forms suffice to model a general reflective behaviour? Glue code is often reflective in nature. Is it enough to reflect over messages or do we need more?

For the moment PICCOLA is untyped. However, most component approaches (e.g., COM [Rog97], CORBA [OMG96], or Darwin [MDEK95]) equip partly or fully the interface specifications with type annotations. One argument for this decision is that only fully and explicitly typed interfaces can benefit from type checking. Furthermore, an independent development of both the client and the provider side may be more or less impossible without appropriate type information (e.g., the current version of PICCOLA already records which names have been used to denote forms). Therefore, a next extension of PICCOLA will support type annotations. Furthermore, missing type annotations will be inferred by a type inference algorithm.

In the field of concurrent and distributed systems, various process calculi have recently been proposed [CG98, FG96, VC98] that incorporate other aspects of distributed computation, such as communication failure, distributed scopes, and security. For the moment, we focus on composed systems within one administrative domain and do not take into account other distribution aspects. But a proper solution that addresses these aspects will play a crucial role when we want to model composition between distributed components.

Acknowledgements

Bibliography

[AC96] Abadi, M. and Cardelli, L. *A Theory of Objects*. Springer, 1996.

[Bou92] Boudol, G. Asynchony and the π-calculus. Technical Report 1702, INRIA Sophia-Antipolis, May 1992.

[CG89] Carriero, N. and Gelernter, D. How to Write Parallel Programs: A Guide to the Perplexed. *ACM Computing Surveys*, 21(3):323–357, September 1989.

[CG98] Cardelli, L. and Gordon, A. D. Mobile Ambients. In Nivat, M., editor, *Foundations of Software Science and Computational Structures*, LNCS 1378, pages 140–155. Springer, 1998.

[CM94] Cardelli, L. and Mitchell, J. C. Operations on Records. In Gunter, C. and Mitchell, J. C., editors, *Theoretical Aspects of Object-Oriented Programming*. MIT Press, 1994. Also appeared as SRC Research Report 48, and in *Mathematical Structures in Computer Science*, 1(1):3–48, March 1991.

[Dam94] Dami, L. *Software Composition: Towards an Integration of Functional and Object-Oriented Approaches.* PhD thesis, Centre Universitaire d'Informatique, University of Geneva, CH, 1994.

[Dam98] Dami, L. A Lambda-Calculus for Dynamic Binding. *Theoretical Computer Science*, 192:201–231, February 1998.

[FG96] Fournet, C. and Gonthier, G. The Reflexive Chemical Abstract Machine and the Join-Calculus. In *Proceedings of the 23rd ACM Symposium on Principles of Programming Languages*, pages 372–385. ACM, January 1996.

[HT92] Honda, K. and Tokoro, M. On asynchronous Communication Semantics. In *ECOOP'91*, LNCS 612. Springer, June 1992.

[Lea96] Lea, D. *Concurrent Programming in Java: Design Principles and Patterns.* The Java Series. Addison-Wesley, October 1996.

[LSN96] Lumpe, M., Schneider, J.-G., and Nierstrasz, O. Using Metaobjects to Model Concurrent Objects with PICT. In *Proceedings of Langages et Modèles à Objets '96*, pages 1–12, Leysin, October 1996.

[Lum99] Lumpe, M. *A π-Calculus Based Approach to Software Composition.* PhD thesis, University of Bern, Institute of Computer Science and Applied Mathematics, January 1999.

[Lut96] Lutz, M. *Programming Python: Object-Oriented Scripting.* O'Reilly & Associates, October 1996.

[MDEK95] Magee, J., Dulay, N., Eisenbach, S., and Kramer, J. Specifying Distributed Software Architectures. In Schäfer, W. and Botella, P., editors, *Proceedings ESEC '95*, LNCS 989, pages 137–153. Springer, September 1995.

[Mil90] Milner, R. Functions as Processes. In *Proceedings ICALP '90*, LNCS 443, pages 167–180. Springer, July 1990.

[Mil91] Milner, R. The Polyadic Pi-Calculus: a Tutorial. Technical Report ECS-LFCS-91-180, Computer Science Department, University of Edinburgh, UK, October 1991.

[MPW92] Milner, R., Parrow, J., and Walker, D. A Calculus of Mobile Processes, Part I/II. *Information and Computation*, 100:1–77, 1992.

[Nie93] Nierstrasz, O. Composing active objects. In Agha, G., Wegner, P., and Yonezawa, A., editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 151–171. MIT Press, 1993.

[NM95] Nierstrasz, O. and Meijler, T. D. Requirements for a Composition Language. In Ciancarini, P., Nierstrasz, O., and Yonezawa, A., editors, *Object-Based Models and Languages for Concurrent Systems*, LNCS 924, pages 147–161. Springer, 1995.

[NSL96] Nierstrasz, O., Schneider, J.-G., and Lumpe, M. Formalizing Composable Software Systems – A Research Agenda. In *Proceedings 1st IFIP Workshop on Formal Methods for Open Object-based Distributed Systems*, pages 271–282. Chapmann & Hall, 1996.

[OMG96] Object Management Group. *The Common Object Request Broker: Architecture and Specification*, July 1996.

[Pie95] Pierce, B. C. Programming in the Pi-Calculus: An experiment in concurrent language design. Technical report, Computer Laboratory, University of Cambridge, UK, May 1995. Tutorial Notes for Pict Version 3.6k.

[PT95] Pierce, B. C. and Turner, D. N. Concurrent Objects in a Process Calculus. In Ito, T. and Yonezawa, A., editors, *Theory and Practice of Parallel Programming (TPPP)*, LNCS 907, pages 187–215. Springer, April 1995.

[Rog97] Rogerson, D. *Inside COM: Microsoft's Component Object Model.* Microsoft Press, 1997.

[San95] Sangiorgi, D. Lazy functions and mobile processes. Technical Report RR-2515, INRIA Sophia-Antipolis, April 1995.

[Sch99] Schneider, J.-G. *Components, Scripts, and Glue: A conceptual framework for*

software composition. PhD thesis, University of Bern, Institute of Computer Science and Applied Mathematics, 1999. to appear.

[SL97] Schneider, J.-G. and Lumpe, M. Synchronizing Concurrent Objects in the Pi-Calculus. In Ducournau, R. and Garlatti, S., editors, *Proceedings of Langages et Modèles à Objets '97*, pages 61–76, Roscoff, October 1997. Hermes.

[Var96] Varone, P. Implementation of "Generic Synchronization Policies" in PICT. Technical Report IAM-96-005, University of Bern, Institute of Computer Science and Applied Mathematics, April 1996.

[VC98] Vitek, J. and Castagna, G. Towards a Calculus of Secure Mobile Computations. In Tsichritzis, D., editor, *Electronic Commerce Objects*, pages 31–46. Centre Universitaire d'Informatique, University of Geneva, CH, July 1998.

[vR96] van Rossum, G. Python Reference Manual. Technical report, Corporation for National Research Initiatives (CNRI), October 1996.

Appendix − Piccola **language definition**

Script   ::=   '**module**' *Name* [*Imports*] *Declarations* [*Main*]

Imports   ::=   '**load**' *NameList* ';'

NameList   ::=   *Name* [',' *NameList*]

Declarations   ::=   *Declaration* [*Declarations*]

Declaration   ::=   '**extern**' *String Name*
                      '**new**' *NameList*
                      '**run**' *Agent*
                      '**procedure**' *Name* '(' [*Variable*] ')' '**do**' *Agent*
                      '**value**' *Name* '=' *Form*
                      '**function**' *Name* '(' [*Variable*] ')' '=' *Form*

Main   ::=   '**main**' *Agent*

Agent   ::=   *PrimaryAgent* ['|' *Agent*]

PrimaryAgent   ::=   '**null**'
                      Location '!' *Form*
                      Location '?' '(' [*Variable*] ')' '**do**' *Agent*
                      Location '?∗' '(' [*Variable*] ')' '**do**' *Agent*
                      '**let**' *Declarations* '**in**' *Agent* '**end**'
                      '**if**' *BoolExpression* '**then**' *Agent* ['**else**' *Agent*] '**end**'
                      Application
                      '(' *Agent* ')'
                      PrimaryForm ';' [*Agent*]

BoolExpression   ::=   *Value Built-in-BoolOperator Value*

Location   ::=   *Name*
                   Variable '.' *Label*
                   PrimaryForm '.' *Label*

Application   ::=   *Location* '(' [*Form*] ')'

Form   ::=   *SeqForm*

SeqForm   ::=   *PrimaryForm* [';' *SeqForm*]

PrimaryForm   ::=   '<' [*FormElementList*] '>'
                      '**let**' *Declarations* '**in**' *Form* '**end**'
                      Application
                      '**return**' *PrimaryForm*
                      '**if**' *BoolExpression* '**then**' *Form* ['**else**' *Form*] '**end**'
                      '(' *Form* ')'
                      Variable

FormElementList   ::=   *FormElement* [',' *FormElementList*]

FormElement   ::=   *Variable*
                      Application
                      Label '=' *Value*
                      '**procedure**' *Name* '(' [*Variable*] ')' '**do**' *Agent*
                      '**function**' *Name* '(' [*Variable*] ')' '=' *Form*

Value   ::=   *Location*
               Number
               String
               Form