A Calculus for Reasoning about Software Composition

Franz Achermann and Oscar Nierstrasz

Software Composition Group
University of Bern, Switzerland
www.iam.unibe.ch/∼scg

Abstract

Although the term *software component* has become commonplace, there is no universally accepted definition of the term, nor does there exist a common foundation for specifying various kinds of components and their compositions. We propose such a foundation. The Piccola calculus is a process calculus, based on the asynchronous π-calculus, extended with *explicit namespaces*. The calculus is high-level, rather than minimal, and is consequently convenient for expressing and reasoning about software components, and different *styles* of composition. We motivate and present the calculus, and outline how it is used to specify the semantics of Piccola, a small composition language. We demonstrate how the calculus can be used to simplify compositions by partial evaluation, and we briefly outline some other applications of the calculus to reasoning about compositional styles.

Keywords: Software components, process calculi, software architecture

Preprint submitted to Elsevier Science. Final version published as:
Franz Achermann and Oscar Nierstrasz, “A Calculus for Reasoning about Software Components,” Theoretical Computer Science, vol. 331, no. 2-3, 2005, pp. 367-396.

1 Introduction

Component-Based Software Development (CBSD) offers us the promise of flexible applications being constructed from stable, robust software components. But how are components plugged into an application? How do we specify the way in which components are configured and composed?

In addition to components, we clearly need some means to specify compositions of components. A *composition language* [39] is a language for specifying operators for connecting components (*i.e.*, “connectors”), *glue* abstractions for adapting component interfaces, and *scripts* that instantiate and connect components. Piccola [4,6] is a prototype for such a composition language, and JPiccola is an implementation which targets the composition of Java software components [38].

A key challenge for a composition language is to offer a means to answer the question, *What, precisely, do we mean by composition?* There are many different notions of component composition and interconnection in practice, so a composition language must offer a neutral foundation in which different forms of composition can be expressed. We therefore need a *semantic foundation* for specifying compositional abstractions. Components may be configured and adapted in many different ways, which may or may not have an impact on the resulting composition. We therefore also need to reason about *equivalence* of different expressions of composition.

Drawing from our experience modeling various component models, we have developed the Piccola calculus as a tool for expressing the semantics of software composition and for reasoning about equivalence of compositions. The Piccola calculus extends the asychronous π-calculus [32,45] with *forms*—first-class, extensible namespaces [5]. Forms are not only convenient for expressing components, but play other important roles as well.

This calculus serves both as the semantic target and as an executable abstract machine for Piccola. In this paper we first motivate the calculus by establishing a set of requirements for modeling composition of software components in section 2. Next, we address these requirements by presenting the syntax and semantics of the Piccola calculus in section 3. In section 4 we provide a brief overview of the Piccola language, and summarize how the calculus helps us to define its semantics. In section 5, we show how the calculus helps us to reason about Piccola compositions and optimize the language bridge by partial evaluation while preserving its semantics. Finally, we conclude with a few remarks about related and ongoing work in sections 6 and 7.

2 Modeling Software Composition

We take as our starting point the view that

$$\mathsf{Applications} = \mathsf{Components} + \mathsf{Scripts}$$

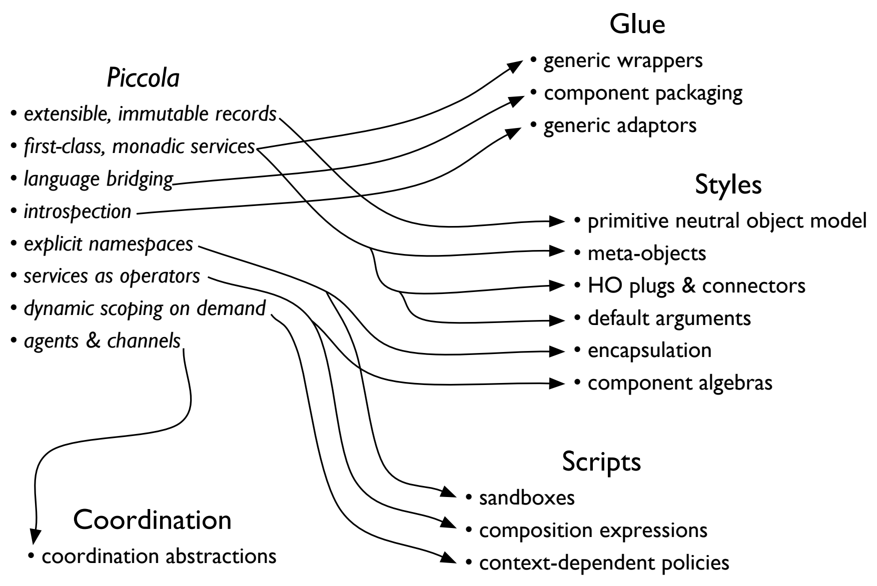

Fig. 1. How Piccola supports composition

that is, component-based applications are (ideally) made up of stable, off-the-shelf components, and scripts that plug them together [6]. Scripts (ideally) make use of high-level connectors that coordinate the services of various components [3,36,52]. Furthermore, complex applications may need services of components that depend on very different architectural assumptions [18]. In these cases, *glue code* is needed to adapt components to different architectural styles [50,51].

A foundation for modeling software components must therefore be suitable for expressing compositional styles, scripts, coordination abstractions and glue code. Figure 1 summarizes the requirements, and illustrates how Piccola and the Piccola calculus support them.

2.1 Compositional Styles

A *compositional style* allows us to express the structure of a software application in terms of components, connectors and rules governing their composition (cf. "architectural style" [52]). The following list summarizes the requirements we identified for a composition language to support the expression of different compositional styles:

- *Neutral object model:* There exists a wide variety of different object and component models. Components may also be bigger or smaller than objects. As a consequence, a general foundation for modeling components should make as few assumptions about objects, classes and inheritance as possi-

ble, namely, objects provide services, they may be instantiated, and their internal structure is hidden.
- *Meta-objects:* On the other hand, many component models depend on run-time reflection, so it must be possible to express dynamic generation of meta-objects.
- *Higher-order plugs and connectors:* In general, connectors can be seen as *higher-order* operators over components and other connectors.
- *Default arguments:* Flexibility in plugging together components is achieved if interface dependencies are minimized. Keyword-based rather than positional arguments to services enable both flexibility and extensibility.
- *Encapsulation:* Components are black-box entities that provide services, without exposing their structure. Sets of components and connectors should be packaged together, while encapsulating the implementation details of the connection mechanisms.
- *Component algebras:* Compositional styles are most expressive when compositions of components and connectors again yield components (or connectors). (The composition of two filters is again a filter.)

Based on these requirements, we conclude that we need (at least) records (to model components and their interfaces), higher-order functions, reflection, and (at some level) overloading of operators. Services may be monadic, taking records as arguments, rather than polyadic. To invoke a service, we just apply it to a record which bundles together all the required arguments, and possibly some optional ones.

These same records can serve as first-class *namespaces* which encapsulate the plugs and connectors of a given style. (A namespace is simply a scope within which certain definitions are visible.) For this reason we unify records and namespaces, and call them *"forms"*, to emphasize their special role.

A "form" is essentially a nested record, which binds labels to values. Let us consider the following script written in JPiccola, an implementation of Piccola for Java components [38]:

```
makeFrame
  title = "AWT Demo"
  x = 200
  y = 100
  hello = "hello world"
  sayHello: println hello
  component = Button.new(text=hello) ? ActionPerformed sayHello
```

This script invokes an abstraction `makeFrame`, passing it a form containing bindings for the labels `title`, `x`, and so on. The script makes use of a compositional style in which GUI components (*i.e.*, the Button) can be bound to events (*i.e.*, `ActionPerformed`) and actions (*i.e.*, `sayHello`) by means of the

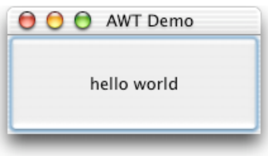

Fig. 2. Evaluating the `helloButton` script

`?` connector.

When we evaluate this code, it generates the button we see in Figure 2. When we click on the button, `hello world` is printed on the Java console.

2.2 Glue

Glue code is needed to package, wrap or adapt code to fit into a compositional style.

- *Generic wrappers:* Wrappers are often needed to introduce specific policies (such as thread-safe synchronization). Generic wrappers are hard to specify for general, polyadic services, but are relatively straightforward if all services are monadic.
- *Component packaging:* Glue code is sometimes needed to package existing code to conform to a particular component model or style. For this purpose, a language bridge is needed to map existing language constructs to the formal component model.
- *Generic adaptors:* Adaptation of interfaces can also be specified generically with the help of reflective or introspective features, which allow components to be inspected before they are adapted.

The JPiccola `helloButton` script only works because Java GUI components are wrapped to fit into our compositional style.

In addition to records and higher-order functions over records, we see that some form of language bridging will be needed, perhaps not at the level of the formal model, but certainly for a practical language or system based on the model.

2.3 Scripts

Scripts configure and compose components using the connectors defined for a style.

- *Sandboxes:* For various reasons we may wish to instantiate components only in a controlled environment. We do not necessarily trust third-party components. Sometimes we would like to adapt components only within a local context. For these and other reasons it is convenient to be able to instantiate and compose namespaces which serve as *sandboxes* for executing scripts.
- *Composition expressions:* Scripts instantiate and connect components. A practical language might conveniently represent connectors as operators. Pipes and filters are well-known, but this idea extends well to other domains.
- *Context-dependent policies:* Very often, components must be prepared to deploy services of the dynamic context. Transaction services, synchronization or communication primitives may depend on the context. For this reason, pure static scoping may not be enough, and *dynamic scoping on demand* will be needed for certain kinds of component models.

So, we see that explicit, manipulable namespaces become more important.

2.4 Coordination

CBSD is especially relevant in concurrent and distributed contexts. For this reason, a foundation for composition must be able to express coordination of interdependent tasks.

- *Coordination abstractions:* Both connectors and glue code may need to express coordination of concurrent activities. Consider a readers/writers synchronization policy as a generic wrapper.

We conclude that we not only need higher-order functions over first-class namespaces (with introspection), but also a way of expressing concurrency and communication [50].

3 The Piccola calculus

As a consequence of the requirements we have identified above, we propose as a foundation a process calculus based on the higher-order asynchronous π-calculus [32,45] in which tuple-based communication is replaced by communication of extensible records, or *forms* [5,37]. Furthermore, forms serve as first-class namespaces and support a simple kind of introspection.

The design of the Piccola calculus strikes a balance between minimalism and expressiveness. As a calculus it is rather large. In fact, it would be possible to express everything we want with the π-calculus alone, but the semantic gap between concepts we wish to model and the terms of the calculus would be

Table 1
Syntax of the Piccola calculus

$A, B, C ::=$	ϵ	*empty form*	$\mid$	$\mathbf{R}$	*current root*
$\mid$	$A; B$	*sandbox*	$\mid$	x	*variable*
$\mid$	$x\mapsto$	*bind*	$\mid$	$hide_x$	*hide*
$\mid$	$\mathbf{L}$	*inspect*	$\mid$	$A \cdot B$	*extension*
$\mid$	$\lambda x.A$	*abstraction*	$\mid$	AB	*application*
$\mid$	$\nu c.A$	*restriction*	$\mid$	$A \mid B$	*parallel*
$\mid$	$c?$	*input*	$\mid$	c	*output*
$F, G, H ::=$	ϵ	*empty form*	$\mid$	S	*service*
$\mid$	$x\mapsto F$	*binding*	$\mid$	$F \cdot G$	*extension*
$S ::=$	$F; \lambda x.A$	*closure*	$\mid$	$\mathbf{L}$	*inspect*
$\mid$	$x\mapsto$	*bind*	$\mid$	$hide_x$	*hide*
$\mid$	c	*output*			

rather large. With the Piccola calculus we are aiming for the smallest calculus with which we can conveniently express components, connectors and scripts.

3.1 Syntax

The Piccola calculus is given by agents A, B, C that range over the set of agents $\mathcal{A}$ in Table 1. There are two categories of identifiers: labels and channels. The set of labels $\mathcal{L}$ is ranged over by x, y, z. (Labels also play the role of *variables* in the calculus, so we use these two terms interchangeably.) Specific labels are also written in the *italic* text font. Channels are denoted by $a, b, c, d \in \mathcal{N}$. Labels are bound with bindings and λ-abstractions, and channels are bound by ν-restrictions.

The operators have the following precedence:

$$application > extension > restriction, abstraction > sandbox > parallel$$

Agent expressions normally reduce to static form values or simply *forms*.

Forms are ranged over by F, G, H. Notice that the set of forms is a subset of all agents. Forms are the first-class citizens of the Piccola calculus, *i.e.*, they are the values that get communicated between agents and are used to invoke services.

The set of forms is denoted by $\mathcal{F}$. Forms contain bindings and services. Services are essentially functional abstractions that may be invoked with arguments, and may possibly entail side-effects. We use S to range over services. User-defined services are closures. Primitive services are *inspect*, the *bind* and *hide* primitives, and the *output* service.

Before considering the formal reduction relation, we first give an informal description of the different agent expressions and how they reduce.

- The *empty form*, ϵ, does not reduce further. It denotes a form without any binding.
- The *current root* agent, $\mathbf{R}$, denotes the current lexical scope.
- A *sandbox* $A;B$ evaluates the agent B in the root context given by A. A binds all free labels in B. If B is a label x, we say that $A;x$ is a *projection* on x in A.
- A *label*, x, denotes the value bound by x in the current root context.
- The primitive service *bind* creates bindings. If A reduces to F, then $x\mapsto A$ reduces to the *binding* $x\mapsto F$.
- The primitive service $hide_x$ removes bindings. So, $hide_x(x\mapsto\epsilon \cdot y\mapsto\epsilon)$ reduces to $y\mapsto\epsilon$.
- The *inspect* service, $\mathbf{L}$, can be used to iterate over the bindings and services of an arbitrary form F. The result of $\mathbf{L}F$ is a service that takes as its argument a form that binds the labels *isEmpty*, *isLabel* and *isService* to services. Depending on whether F is the empty form, contains bindings, or is only a service, the appropriate one of these three services will be invoked.
- The values of two agents are concatenated by *extension*. In the value of $A\cdot B$ the bindings of B override those for the same label in A.
- An *abstraction* $\lambda x.A$ abstracts x in A.
- The *application* AB denotes the result of applying A to B. The Piccola calculus uses a call-by-value reduction order. In order to reduce AB, A must reduce to a service and B to a form.
- The expression $\nu c.A$ *restricts* the visibility of the channel name c to the agent expression A, as in the π-calculus.
- $A \mid B$ spawns off the agent A asynchronously and yields the value of B. Unlike in the π-calculus, the parallel composition operator is not commutative, since we do not wish parallel agents to reduce to non-deterministic values.
- The agent $c?$ *inputs* a form from channel c and reduces to that value. The reader familiar with the π-calculus will notice a difference with the input prefix. Since we have explicit substitution in our calculus it is simpler to specify the input by $c?$ and use the context to bind the received value instead of defining a prefix syntax $c(X).A$ as in the π-calculus.

Table 2
Free channels

$fc(\epsilon) = \emptyset$	$fc(\mathbf{R}) = \emptyset$
$fc(x) = \emptyset$	$fc(\mathbf{L}) = \emptyset$
$fc(x\mapsto) = \emptyset$	$fc(hide_x) = \emptyset$
$fc(A; B) = fc(A) \cup fc(B)$	$fc(A \cdot B) = fc(A) \cup fc(B)$
$fc(\lambda x.A) = fc(A)$	$fc(AB) = fc(A) \cup fc(B)$
$fc(\nu c.A) = fc(A) \backslash \{c\}$	$fc(A \mid B) = fc(A) \cup fc(B)$
$fc(c?) = \{c\}$	$fc(c) = \{c\}$

- The channel c is a primitive *output* service. If A reduces to F, then cA reduces to the *message* cF. The value of a message is the empty form ϵ. (The value F is only obtained by a corresponding input $c?$ in another agent.)

3.2 Free Channels and Closed Agents

As in the π-calculus, forms may contain free channel names. An agent may create a new channel, and communicate this new name to another agent in a separate lexical scope.

The *free channels* $fc(A)$ of an agent A are defined inductively in Table 2. α-conversion (of channels) is defined in the usual way. We identify agent expressions up to α-conversion.

We omit a definition of *free variables*. Since the Piccola calculus is a calculus with explicit environments, we cannot easily define α-conversion on variables. Such a definition would have to include the special nature of $\mathbf{R}$. Instead, we define a *closed agent* where all variables, root expressions, and abstractions occur beneath a sandbox:

Definition 1 *The following agents A are closed:*

- *$\epsilon, x\mapsto, hide_x, \boldsymbol{L}, c$ and $c?$ are closed.*
- *If A and B are closed then also $A \cdot B, AB, A \mid B$ and $\nu c.A$ are closed.*
- *If A is closed, then also $A; B$ is also closed for any agent B.*

Observe that any form F is closed by the above definition. An agent is *open* if it is not closed. Open agents are $\mathbf{R}$, variables x, abstractions $\lambda x.A$ and compositions thereof. Any agent can be closed by putting it into a sandbox with a closed context. Sandbox agents are closed if the root context is closed. In lemma 1 we show that the property of being closed is preserved by reduction.

Table 3
Congruences I

$\equiv$ is the smallest congruence satisfying the axioms in tables 3, 4 and 5:

		$F \cdot \epsilon$	$\equiv$	F	(ext empty right)
		$\epsilon \cdot F$	$\equiv$	F	(ext empty left)
		$(F \cdot G) \cdot H$	$\equiv$	$F \cdot (G \cdot H)$	(ext assoc)
		$S \cdot (x{\mapsto}F)$	$\equiv$	$(x{\mapsto}F) \cdot S$	(ext service commute)
$x \neq y$	implies	$x{\mapsto}F \cdot y{\mapsto}G$	$\equiv$	$y{\mapsto}G \cdot x{\mapsto}F$	(ext bind commute)
		$x{\mapsto}F \cdot x{\mapsto}G$	$\equiv$	$x{\mapsto}G$	(single binding)
		$S \cdot S'$	$\equiv$	S'	(single service)

Table 4
Congruences II

		$F; A \cdot B$	$\equiv$	$(F; A) \cdot (F; B)$	(sandbox ext)
		$F; AB$	$\equiv$	$(F; A)(F; B)$	(sandbox app)
		$A; (B; C)$	$\equiv$	$(A; B); C$	(sandbox assoc)
		$F; G$	$\equiv$	G	(sandbox value)
		$F; \mathbf{R}$	$\equiv$	F	(sandbox root)
		$\mathit{hide}_x(F \cdot x{\mapsto}G)$	$\equiv$	$\mathit{hide}_x F$	(hide select)
$x \neq y$	implies	$\mathit{hide}_y(F \cdot x{\mapsto}G)$	$\equiv$	$\mathit{hide}_y F \cdot x{\mapsto}G$	(hide over)
		$\mathit{hide}_x \epsilon$	$\equiv$	ϵ	(hide empty)
		$\mathit{hide}_x S$	$\equiv$	S	(hide service)
		$(F \cdot S)G$	$\equiv$	SG	(use service)

3.3 Congruence and Pre-forms

As in the π-calculus, we introduce *structural congruence* over agent expressions to simplify the reduction relation. The congruence allows us to rewrite agent expressions to bring communicating agents into juxtapositions, as in the Chemical Abstract Machine of Berry and Boudol [9].

The congruence rules constitute three groups. The first group (Table 3) specifies that extension is idempotent and associative on forms.

The rules *single service* and *single binding* specify that extension overwrites services and bindings with the same label.

We define $\mathit{labels}(F)$ as follows:

Table 5
Congruences III

		$(A \mid B) \mid C$	$\equiv$	$A \mid (B \mid C)$	(par assoc)
		$(A \mid B) \mid C$	$\equiv$	$(B \mid A) \mid C$	(par left commute)
		$(A \mid B) \cdot C$	$\equiv$	$A \mid B \cdot C$	(par ext left)
		$F \cdot (A \mid B)$	$\equiv$	$A \mid F \cdot B$	(par ext right)
		$(A \mid B)C$	$\equiv$	$A \mid BC$	(par app left)
		$F(A \mid B)$	$\equiv$	$A \mid FB$	(par app right)
		$(A \mid B); C$	$\equiv$	$A \mid B; C$	(par sandbox left)
		$F; (A \mid B)$	$\equiv$	$F; A \mid F; B$	(par sandbox right)
		$F \mid A$	$\equiv$	A	(discard zombie)
		cF	$\equiv$	$cF \mid \epsilon$	(emit)
		$\nu cd.A$	$\equiv$	$\nu dc.A$	(commute channels)
$c \notin fc(A)$	implies	$A \mid \nu c.B$	$\equiv$	$\nu c.(A \mid B)$	(scope par left)
$c \notin fc(A)$	implies	$(\nu c.B) \mid A$	$\equiv$	$\nu c.(B \mid A)$	(scope par right)
$c \notin fc(A)$	implies	$(\nu c.B) \cdot A$	$\equiv$	$\nu c.(B \cdot A)$	(scope ext left)
$c \notin fc(A)$	implies	$A \cdot \nu c.B$	$\equiv$	$\nu c.(A \cdot B)$	(scope ext right)
$c \notin fc(A)$	implies	$A; \nu c.B$	$\equiv$	$\nu c.(A; B)$	(scope sandbox left)
$c \notin fc(A)$	implies	$(\nu c.B); A$	$\equiv$	$\nu c.(B; A)$	(scope sandbox right)
$c \notin fc(A)$	implies	$(\nu c.B)A$	$\equiv$	$\nu c.BA$	(scope app left)
$c \notin fc(A)$	implies	$A(\nu c.B)$	$\equiv$	$\nu c.AB$	(scope app right)

Definition 2 *For each form F, the set of $labels(F) \subset \mathcal{L}$ is given by:*

$$\begin{array}{ll} labels(\epsilon) = \emptyset & labels(S) = \emptyset \\ labels(x \mapsto G) = \{x\} & labels(F \cdot G) = labels(F) \cup labels(G) \end{array}$$

Using the form congruences, we can rewrite any form F into one of the following three cases:

$$\begin{array}{ll} F \equiv \epsilon & \\ F \equiv S & \\ F \equiv F' \cdot x \mapsto G & \text{where } x \notin labels(F') \end{array}$$

This is proved by structural induction over forms [2]. This formalizes our idea that forms are extensible records unified with services. A form has at most one binding for a given label.

The second group (Table 4) defines *preforms*. These are agent expressions that are congruent to a form. For instance, the agent $hide_x \epsilon$ is equivalent to the empty form ϵ. The set of all preforms is defined by:

$$\mathcal{F}^{\equiv} = \{A | \exists F \in \mathcal{F} \text{ with } F \equiv A\}$$

Clearly, all forms are preforms.

The last group (Table 5) defines the semantics of parallel composition and communication for agents. Note how these rules always preserve the position of the rightmost agent in a parallel composition, since this agent, when reduced to a form, will represent the value of the composition. In particular, the rule *discard zombie* garbage-collects form values appearing to the left of this position. The rule *emit*, on the other hand, spawns an empty form as the value, thus enabling the message to move around freely. For instance in

$$\begin{aligned} x{\mapsto}c() &\equiv x{\mapsto}(c() \mid \epsilon) && \text{by } \textit{emit} \\ &\equiv c() \mid x{\mapsto}\epsilon && \text{by } \textit{par ext right} \end{aligned}$$

the message $c()$ escapes the binding $x{\mapsto}$.

3.4 Reduction

We define the reduction relation $\rightarrow$ on agent expressions to reduce applications, communications and projections (see Table 6). $\Rightarrow$ is the reflexive and transitive closure of $\rightarrow$.

Especially noteworthy is the rule *reduce beta*. This rule does not substitute G for x in the agent A as in the classical λ-calculus. Instead, it extends the environment in which A is evaluated. This is essentially the beta-reduction rule found in calculi for explicit substitution [1,40]:

$$(F; \lambda x.A)G \;\rightarrow\; F \cdot x{\mapsto}G; A$$

The application of the closure $F; \lambda x.A$ to the argument G reduces to a sandbox expression in which the agent A is evaluated in the environment $F \cdot x{\mapsto}G$. Free occurrences of x in A will therefore be bound to G.

The property of being closed is respected by reduction:

Lemma 1 *If A is a closed agent and $A \rightarrow B$ or $A \equiv B$ then B is closed as well.*

PROOF. Easily checked by induction over the formal proof for $A \rightarrow B$.

Table 6
Reduction rules

$$
\begin{array}{rr}
(F; \lambda x.A)\ G \to F \cdot x{\mapsto}G; A & \text{(reduce beta)} \\
cF \mid c? \to F & \text{(reduce comm)} \\
F \cdot x{\mapsto}G; x \to G & \text{(reduce project)} \\
\mathbf{L}\epsilon \to \epsilon; \lambda x.(x; \mathit{isEmpty})\epsilon & \text{(reduce inspect empty)} \\
\mathbf{L}S \to \epsilon; \lambda x.(x; \mathit{isService})\epsilon & \text{(reduce inspect service)} \\
\mathbf{L}(F \cdot x{\mapsto}G) \to \epsilon; \lambda x.(x; \mathit{isLabel})\mathit{label}_x & \text{(reduce inspect label)} \\
\dfrac{A \equiv A' \quad A' \to B' \quad B' \equiv B}{A \to B} & \text{(reduce struct)} \\
\dfrac{A \to B}{\mathcal{E}[A] \to \mathcal{E}[B]} & \text{(reduce propagate)}
\end{array}
$$

where $\mathit{label}_x = \mathit{project}{\mapsto}(\epsilon; \lambda x.(x;x)) \cdot \mathit{hide}{\mapsto}\mathit{hide}_x \cdot \mathit{bind}{\mapsto}(x{\mapsto})$ and $\mathcal{E}$ is an evaluation context defined by the grammar:

$$\mathcal{E} ::= [\,] \mid \mathcal{E} \cdot A \mid F \cdot \mathcal{E} \mid \mathcal{E}; A \mid F; \mathcal{E} \mid \mathcal{E}A \mid F\mathcal{E} \mid A|\mathcal{E} \mid \mathcal{E}|A \mid \nu c.\mathcal{E}$$

3.5 Encoding Booleans

The following toy example actually illustrates many of the principles at stake when we model components with the Piccola calculus.

We can encode booleans by services that either project on the labels *true* or *false* depending on which boolean value they are supposed to model (cf. [15]). (This same idea is used by the primitive service **L** to reflect over the bindings and services of a form.)

$$
\begin{aligned}
\mathit{True} &\overset{\text{def}}{=} \epsilon; \lambda x.(x; \mathit{true}) \\
\mathit{False} &\overset{\text{def}}{=} \epsilon; \lambda x.(x; \mathit{false})
\end{aligned}
$$

Consider now:

$$\begin{aligned} \mathit{True}(\mathit{true}{\mapsto}1 \cdot \mathit{false}{\mapsto}2) &= (\epsilon; \lambda x.(x; \mathit{true}))(\mathit{true}{\mapsto}1 \cdot \mathit{false}{\mapsto}2) \\ &\rightarrow \epsilon \cdot x{\mapsto}(\mathit{true}{\mapsto}1 \cdot \mathit{false}{\mapsto}2); (x; \mathit{true}) \quad \text{by } \mathit{reduce\ beta} \\ &\equiv (\epsilon \cdot x{\mapsto}(\mathit{true}{\mapsto}1 \cdot \mathit{false}{\mapsto}2); x); \mathit{true} \\ &\qquad \text{by } \mathit{sandbox\ assoc} \\ &\rightarrow (\mathit{true}{\mapsto}1 \cdot \mathit{false}{\mapsto}2); \mathit{true} \quad \text{by } \mathit{reduce\ project} \\ &\equiv (\mathit{false}{\mapsto}2 \cdot \mathit{true}{\mapsto}1); \mathit{true} \quad \text{by } \mathit{ext\ bind\ commute} \\ &\rightarrow 1 \quad \text{by } \mathit{reduce\ project} \end{aligned}$$

Note how the bindings are swapped to project on *true* in the last step. A similar reduction would show $\mathit{False}(\mathit{true}{\mapsto}1 \cdot \mathit{false}{\mapsto}2) \Rightarrow 2$.

A key point is that a form may provide *additional bindings* that a client may ignore if it does not require them (cf. [15]). This same principle is applied to good effect in various scripting languages, such as Python [29]. For instance we can use *True* and provide an additional binding $\mathit{notused}{\mapsto}F$ for arbitrary form F:

$$\begin{aligned} &\mathit{True}(\mathit{true}{\mapsto}1 \cdot \mathit{false}{\mapsto}2 \cdot \mathit{notused}{\mapsto}F) \\ &\qquad \Rightarrow (\mathit{true}{\mapsto}1 \cdot \mathit{false}{\mapsto}2 \cdot \mathit{notused}{\mapsto}F); \mathit{true} \\ &\qquad \equiv (\mathit{false}{\mapsto}2 \cdot \mathit{true}{\mapsto}1 \cdot \mathit{notused}{\mapsto}F); \mathit{true} \quad \text{by } \mathit{ext\ bind\ commute} \\ &\qquad \equiv (\mathit{false}{\mapsto}2 \cdot \mathit{notused}{\mapsto}F \cdot \mathit{true}{\mapsto}1); \mathit{true} \quad \text{by } \mathit{ext\ bind\ commute} \\ &\qquad \rightarrow 1 \quad \text{by } \mathit{reduce\ project} \end{aligned}$$

Extending forms can also be used to *overwrite* existing bindings. For instance instead of binding the variable *notused* a client may override *true*:

$$\mathit{True}(\mathit{true}{\mapsto}1 \cdot \mathit{false}{\mapsto}2 \cdot \mathit{true}{\mapsto}3) \Rightarrow 3$$

A conditional expression is encoded as a curried service that takes a boolean and a case form. When invoked, it selects and evaluates the appropriate service in the case form:

$$\mathit{if} \overset{\text{def}}{=} \epsilon; \lambda uv.u(\mathit{true}{\mapsto}(v; \mathit{then}) \cdot \mathit{false}{\mapsto}(v; \mathit{else}))\epsilon$$

Now consider:

$$\begin{aligned} &\mathit{if}\ \ \mathit{True}\ \ (\mathit{then}{\mapsto}(F; \lambda x.A) \cdot \mathit{else}{\mapsto}(G; \lambda x.B)) \\ &\quad \Rightarrow F \cdot x{\mapsto}\epsilon; A \end{aligned}$$

The expression *if True* has triggered the evaluation of agent A in the environment $F \cdot x{\mapsto}\epsilon$.

The contract supported by *if* requires that the cases provided bind the labels *then* and *else*. We can relax this contract and provide default services if those bindings are not provided by the client. To do so, we replace in the definition of *if* the sandbox expression $v; \mathit{else}$ with a default service. This service gets triggered when the case form does not contain an *else* binding:

$$\mathit{if}_d \overset{\text{def}}{=} \epsilon; \lambda uv.u(\mathit{true}{\mapsto}(v; \mathit{then}) \cdot \mathit{false}{\mapsto}(\mathit{else}{\mapsto}(\lambda x.\epsilon) \cdot v; \mathit{else}))\epsilon$$

Now if_d *False* $(\mathit{then}{\mapsto}(F; \lambda x.A)) \Rightarrow \epsilon$.

3.6 Equivalence for Agents

Two agents are equivalent if they exhibit the same behaviour, *i.e.*, they enjoy the same reductions. We adopt Milner and Sangiorgi's notion of *barbed bisimulation* [33]. The idea is that an agent A is barbed similar to B if A can exhibit any reduction that B does and if B is a barb, then A is a barb, too. If A and B are similar to each other they are bisimilar. The advantage of this bisimulation is that it can readily be given for any calculus that contains barbs or values.

For the asynchronous π-calculus, barbs are usually defined as having the capability of doing an output on a channel. A Piccola calculus agent reduces to a barb, *i.e.*, it returns a form. During evaluation the agent may spawn off new subthreads which could be blocked or still be running. We consequently define barbs as follows:

Definition 3 *A* barb V *is an agent expression* A *that is congruent to an agent generated by the following grammar:*

$$V ::= F \;\Big|\; A|V \;\Big|\; \nu c.V$$

We write $A{\downarrow}$ *for the fact that* A *is a barb, and* $A{\Downarrow}$ *when a barb* V *exists such that* $A \Rightarrow V$.

The following lemma relates forms, barbs and agents:

Lemma 2 *The following inclusion holds and is strict:*

$$\mathcal{F} \subset \mathcal{F}^{\equiv} \subset \{A | A{\downarrow}\} \subset \mathcal{A}$$

PROOF. The inclusions hold by definition. To see that the inclusion are strict, consider the empty form ϵ, the agent $\mathit{hide}_x\epsilon$, the barb $\mathbf{0} \mid \mathit{hide}_x\epsilon$ and the agent $\mathbf{0}$ (where $\mathbf{0} = \nu c.c?$ is the deadlocked *null* agent).

The following lemma gives a syntactical characterization of barbs.

Lemma 3 *For any form F, agent A, and label x, the following terms are barbs, given V_1 and V_2 are barbs.*

$$\begin{array}{ll} V_1 \cdot V_2 & \nu c.V_1 \\ V_1; V_2 & A \mid V_1 \\ x{\mapsto}V_1 & \end{array}$$

PROOF. By definition we have $V \equiv \nu\tilde{c}.A \mid F$. The claim follows by induction over F.

We now define barbed bisimulation and the induced congruence:

Definition 4 *A relation $\mathcal{R}$ is a* (weak) barbed bisimulation*, if $A \mathrel{\mathcal{R}} B$,* i.e., *$(A, B) \in \mathcal{R}$ implies:*

- *If $A \to A'$ then there exists an agent B' with $B \Rightarrow B'$ and $A' \mathrel{\mathcal{R}} B'$.*
- *If $B \to B'$ then there exists an agent A' with $A \Rightarrow A'$ and $A' \mathrel{\mathcal{R}} B'$.*
- *If $A\downarrow$ then $B\Downarrow$.*
- *If $B\downarrow$ then $A\Downarrow$.*

Two agents are (weakly) barbed bisimilar, written $A \stackrel{\cdot}{\approx} B$, if there is some (weak) barbed bisimulation $\mathcal{R}$ with $A \mathrel{\mathcal{R}} B$. Two agents are (weakly) barbed congruent, written $A \approx B$, if for all contexts $\mathcal{C}$ we have $\mathcal{C}[A] \stackrel{\cdot}{\approx} \mathcal{C}[B]$.

We define behavioural equality using the notion of barbed congruence. As usual we can define strong and weak versions of barbed bisimulation. The strong versions are obtained in the standard way by replacing $\Rightarrow$ with $\to$ and $\Downarrow$ with $\downarrow$ in Definition 4. We only concentrate on the weak case since it abstracts internal computation.

3.7 Erroneous Reductions

Not all agents reduce to forms. Some agents enjoy an infinite reduction [2]. Other agents may be stuck. An agent is stuck if it is not a barb and can reduce no further.

Definition 5 *An agent A is* stuck, *written $A\!\uparrow$, if A is not a barb and there is no agent B such that $A \rightarrow B$.*

Clearly it holds that $\mathbf{0}\!\uparrow$ and $\mathbf{R}\!\uparrow$. The property of being stuck is not compositional. For instance $c?\!\uparrow$ but obviously, $c() \mid c?$ can reduce to ϵ. We can put $\mathbf{R}$ into a context so that it becomes a barb, for instance $F; \mathbf{R} \equiv F$. Note that if an agent is stuck it is not a preform: $\mathcal{F}^{\equiv} \cap \{A | A\!\uparrow\} = \emptyset$ by definition.

Although $\mathbf{0}$ is arguably stuck by intention, in general a stuck agent can be interpreted as an error. The two typical cases which may lead to errors are (i) projection on an unbound label, *e.g.*, $\epsilon; x$, and (ii) application of a non-service, *e.g.*, $\epsilon\epsilon$.

3.8 π-Calculus Encoding

One may well ask what exactly the Piccola calculus adds over and above the asynchronous π-calculus. We have previously shown that the Piccola calculus can be faithfully embedded into the localized π-calculus Lπ of Merro and Sangiorgi [30,45].

The mapping $[\![]\!]_a$ encodes Piccola calculus agents as π-calculus processes. The process $[\![A]\!]_a$ evaluates A in the environment given by the empty form, and sends the resulting value along the channel a. A form (value) is encoded as a 4-tuple of channels representing projection, invocation, hiding and selection. The main result is that the encoding is sound and preserves reductions. We do not require a fully abstract encoding since that would mean that equivalent Piccola calculus agents translated into the π-calculus could not be distinguished by any π-processes. Our milder requirement means that we consider only π-processes which are translations of Piccola calculus agents themselves and state that they cannot distinguish similar agents:

Proposition 1 (Soundness) *For closed agents A, B and channel a the congruence $[\![A]\!]_a \approx [\![B]\!]_a$ implies $A \approx B$.*

Although it is comforting to learn that the π-calculus can serve as a foundation for modeling components, it is also clear from the complexity of the encoding that it is very distant from the kinds of abstractions we need to conveniently model software composition. For this reason we find that a richer calculus is more convenient to express components and connectors.

Applications:	Components + Scripts
Composition styles:	Streams, GUI composition, ...
Standard libraries:	Coordination abstractions, control structures, basic object model ...
Piccola language:	Host components, user-defined operators, dynamic namespaces
Piccola calculus:	Forms, agents and channels

Fig. 3. Piccola layers

4 From the Piccola calculus to Piccola

Piccola is a small composition language that supports the requirements summarized in Figure 1, and whose denotational semantics is defined in terms of the Piccola calculus [2].

Piccola is designed in layered fashion (see Figure 3). At the lowest level we have an abstract machine that implements the Piccola calculus.

At the second level, we have the Piccola language, which is implemented by translation to the abstract machine, following the specification of the denotational semantics.

Piccola provides a more convenient, Python-like syntax for programming than does the calculus, including overloaded operators to support component composition. It also provides a bridge to the host language (currently Java or Squeak). Piccola provides no basic data types other than forms and channels. Booleans, integers, floating point numbers and strings, for example, must be provided by the host language through the language bridge.

Curiously, the syntax of the Piccola calculus is actually larger than that of Piccola itself. This is because we need to represent all semantic entities, including agents and channels, as syntactic constructs in the calculus. In the Piccola language, however, these are represented only by standard library services, such as `run` and `newChannel`.

The third level provides a set of standard libraries to simplify the task of programming with Piccola. Not only does the Piccola language provide no built-in data types, it does not even offer any control structures of its own. These, however, are provided as standard services implemented in Piccola. Exceptions and try-catch clauses are implemented using agents, channels, and dynamic namespaces [5].

The first three layers constitute the standard Piccola distribution. The fourth layer is provided by the component framework designer. At this level, a domain expert encodes a compositional styles as a library of components, connectors, adaptors, coordination abstractions, and so on. Finally, at the top level, an application programmer may script together components using the abstractions provided by the lower layers [3,36].

Piccola has been used to experiment with the implementation of various compositional styles, including styles for GUI composition [6], styles for actor coordination [25], and styles for wrapping white-box frameworks as black-box components [49]. Tutorial examples are described in the JPiccola user's guide [38]

5 Partial Evaluation

Since Piccola is a *pure* composition language, evaluating scripts requires intensive *upping* and *downing* [31] between the "down" level of the host language and the "up" level of Piccola. If the language bridge were implemented naïvely, it would be hopelessly inefficient.

Partial evaluation [8,24,13] is a program transformation technique which, given a program and parts of its arguments, produces a specialized program with respect to those arguments. In this section we present a partial evaluation algorithm for Piccola [2,47,48]. This algorithm uses the fact that forms are immutable. We replace references to forms by the forms referred to. We can then specialize projections and replace applications of referentially transparent services by their results. However, most services in Piccola are not referentially transparent and cannot be inlined since that would change the order in which side-effects are executed. We need to separate the referentially transparent part from the non-transparent part in order to replace an application with its result and to ensure that the order in which the side-effects are evaluated is preserved.

The algorithm separates functional Piccola agents into *side-effect terms* and *lazy forms expressions*. The former contain applications that may cause side-effects and projections that may be undefined. For side-effect terms the order of evaluation is important. In contrast, lazy forms are referentially transparent. As subexpressions they contain deferred projections, bindings, and hidden forms. Dropping unnecessary subexpressions does not change the semantics of the lazy form. We call these expressions lazy since the bindings can be evaluated on demand.

In section 5.1 we give an overview of the algorithm. We formally define it and illustrate its use with examples in section 5.2. In section 5.3 we prove correctness and termination of the algorithm.

5.1 Overview

Before presenting the algorithm in detail, we give an informal account of the main idea. We separate each service s into two services s_p and s_r. The first service, s_p, is the side-effect part of the service. When we apply s_p to a form F, the side-effects of sF are evaluated. We refer to the result of s_pF as the side-effect. The service s_r is referentially transparent. It takes the side-effect and the argument F and returns the value of sF. Thus the service s is split into s_p and s_r such that the following holds:

$$sF = s_r(s_pF)F$$

As an example, consider the service `wrapRec` which wraps a value received from the channel `ch`:

```
wrapRec ch:
    received = ch.receive()        # side-effect
    value = wrap received          # wrap is referentially transparent
    channel = ch                   # return ch as part of the result
```

Here we assume that the (unspecified) `wrap` service is referentially transparent — for example, it might simply provide new names for the labels bound in the `received` form.

If we separate the side-effect from the referentially transparent part, we obtain services `wrapRec`$_p$ and `wrapRec`$_s$ as follows:

```
wrapRec_p ch: ch.receive()
wrapRec_r side ch:
    received = side
    value = wrap side
    channel = ch
```

We can now defer invocation of the referentially transparent service. Assume we use the result of an invocation of `wrapRec` and project on the `received` label. In that case, the invocation of `wrap` is not necessary anymore. The code

```
a = (wrapRec ch).received
```

is equivalent to “`a = wrapRec`$_r$ `(wrapRec`$_p$ `ch) ch`” which has the same effect as

Table 7
Functional Piccola agents

$A, B, C ::= \mathbf{L}$	*inspect*	$\mid$ **new**	*new Channel*
$\mid$ **run**	*run*	$\mid hide_x$	*hide*
$\mid \epsilon$	*empty form*	$\mid \mathbf{R}$	*(static) root*
$\mid A; B$	*sandbox*	$\mid A \cdot B$	*extension*
$\mid x$	*variable*	$\mid x \mapsto A$	*binding*
$\mid \lambda x.A$	*abstraction*	$\mid AB$	*application*

$$\mathbf{new} \;=\; \epsilon; \lambda x.\nu c.send{\mapsto}c \cdot receive{\mapsto}(\lambda y.c?) \qquad \mathbf{run} \;=\; \epsilon; \lambda x.(x() \mid \epsilon)$$

Table 8
Side-effect terms and lazy forms

$P ::= \epsilon$	*empty form*	$\mid x \mapsto P$	*nested side-effect*
$\mid P \cdot P$	*extension*	$\mid x \mapsto R.x$	*projection*
$\mid x \mapsto RR$	*side-effect application*		
$R ::= \epsilon$	*empty form*	$\mid x$	*variable*
$\mid R \cdot R$	*extension*	$\mid x \mapsto R$	*binding*
$\mid R.x$	*projection*	$\mid \lambda x.P \star R$	*lazy abstraction*
$\mid \mathbf{side}(A)$	*side-effect service*		

```
a = ch.receive()
```

when we inline the referentially transparent service.

The algorithm not only splits services but also any *functional agent*: an agent written using a functional subset of the Piccola calculus [2] (see Table 7). We also do not represent the lazy part as curried services. Instead the root context consists of the side-effects bound by unique labels.

For simplicity, we do not consider the hide primitive and treat it as a service with a side-effect. The present approach can easily be extended to handle hide more effectively [2].

The partial evaluation algorithm *partial* : $\mathcal{A} \rightarrow \mathcal{A}$ is expressed in two steps. First an agent A is split into a side-effect term and a lazy form expression. Then, the side-effect term and the lazy form are combined back into a specialized agent. The set of side-effect terms is denoted by $\mathcal{P}$ and ranged over by P. The set of lazy forms $\mathcal{R}$ ranged over by R. Some helper predicates are defined over lazy forms and side-effects. In that case we use Q to range over

$\mathcal{P} \cup \mathcal{R}$. The grammar for side-effect terms and lazy forms is given in Table 8. We adopt the same precedence as for functional agents: projection is stronger than binding which is stronger than application.

The two functions of the algorithm are *split* and *combine*:

- The function $\mathit{split} : (\mathcal{A} \times \mathcal{R}) \rightarrow (\mathcal{P} \times \mathcal{R})$ separates a functional agent into a side-effect and a lazy form. We split the agent A in the context given by the lazy form R' and get a side-effect term P and a lazy form R, written $\mathit{split}(A, R') = (P, R)$.
- The function $\mathit{combine}(\mathcal{P} \times \mathcal{R}) \rightarrow \mathcal{A}$ combines the side-effect and the lazy form back into a functional agent.

The partial evaluation algorithm $\mathit{partial} : \mathcal{A} \rightarrow \mathcal{A}$ is defined as:

$$\mathit{partial}(A) = \mathit{combine}(\mathit{split}(A, \epsilon))$$

Observe that we assume the empty form as initial context for specializing an agent A.

Lazy form expressions are referentially transparent. They contain unevaluated projections that are guaranteed to succeed as we will see. Evaluating lazy forms can be deferred. Lazy forms contain references to side-effects or to formal parameters. Lazy abstractions $\lambda x.P \star R$ contain their side-effect P and the referentially transparent result R.

Side-effect services are arbitrary agents A. Partial evaluation does not specialize them. The primitive services $\mathbf{new}, \mathbf{run}$, and $\mathbf{L}$ are side-effect services. Side-effect terms contain applications and projections that may fail. Side-effect terms have a specific structure. Atomic side-effects are bound by unique labels. Side-effect terms can be nested and sequentially composed. In $P_1 \cdot P_2$ we may refer to side-effects of P_1 from P_2.

This algorithm has been used extensively in the implementation of SPiccola [47,48], an implementation of Piccola for components written in Squeak [23], an open-source Smalltalk.

5.2 The Algorithm

We now present and discuss the functions *split* and *combine* in detail. We assume that $\cdot$ is associative and ϵ is the neutral element. This allows us to reduce the number of defining equations. For instance, when defining projection in Table 11 we write $\mathit{project}(R \cdot x \mapsto R_1, x) = R_1$ assuming that we can rewrite any form with several bindings into a form extended with a single binding.

Table 9
Free variables and defined labels

$fv(\epsilon) = \emptyset$	$fv(x) = \{x\}$
$fv(R_1R_2) = fv(R_1) \cup fv(R_2)$	$fv(R.x) = fv(R)$
$fv(Q_1 \cdot Q_2) = fv(Q_1) \cup fv(Q_2)$	$fv(x{\mapsto}Q) = fv(Q)$
$fv(\lambda x.P \star R) = (fv(P) \cup fv(R)) \backslash (labels(P) \cup \{x\})$	$fv(\mathbf{side}(A)) = \emptyset$
$labels(\epsilon) = \emptyset$	$labels(x) = \emptyset$
$labels(x{\mapsto}Q) = \{x\}$	$labels(R.x) = \emptyset$
$labels(Q_1 \cdot Q_2) = labels(Q_1) \cup labels(Q_2)$	$labels(\mathbf{side}(A)) = \emptyset$
$labels(R_1R_2) = \emptyset$	$labels(\lambda x.P \star R) = \emptyset$

We need some helper predicates for the free and bound variables and we define substitution.

The set of free variables in a lazy or side-effect term, $fv(Q)$ is defined in Table 9. Note that the definition of free labels of an ordinary Piccola agent is meaningless as the free labels of $\mathbf{R}$ are undefined. For lazy forms and side-effects, a recursive definition can be given since sandbox expression are inlined and lazy forms do not contain $\mathbf{R}$. The interesting case is the free variables for abstractions $\lambda x.P \star R$. They are constructed by taking the free variables of P and R and removing x and the labels that are defined by P. This definition reflects the fact that R will be evaluated in a context defined by P as we will see.

The predicate $labels(Q)$ denotes the set of labels that are bound by Q. The label of a binding $x{\mapsto}Q$ is the set $\{x\}$. The set of labels of an extension is the union of the labels of the subexpressions. We are conservative when the set of labels cannot be inferred in a straightforward manner. For instance, the set of labels of any application or projection is empty.

The expression $Q[x/R]$ denotes the expression Q where all free x are replaced by R. Substitution is defined in Table 10. Note that there is no special definition for the side-effect $y{\mapsto}R_1R_2$. This case is defined by the binding and by the application, thus $(y{\mapsto}R_1R_2)[x/R] = y{\mapsto}(R_1R_2)[x/R] = R_1[x/R]\ R_2[x/R]$. Note that $R[x/R]' \in \mathcal{R}$ and $P[x/R] \in \mathcal{P}$. This means that a substitution on a lazy form denotes a lazy form, and a substitution on a side-effect term denotes a side-effect term. As usual we replace bound variables to avoid name capture [20].

The helper predicate $project : (\mathcal{R} \times \mathcal{L}) \rightarrow \mathcal{Q}$ denotes the value bound by a label (Table 11). If the projection can be performed at specialize time, we do the actual lookup. If the value of the projection is not known, an uneval-

Table 10
Substitution

$$\begin{aligned}
\epsilon[x/R] &= \epsilon \\
(P \cdot Q)[x/R] &= P[x/R] \cdot Q[x/R] \\
x[x/R] &= R \\
y[x/R] &= y && \text{where } x \neq y \\
(y \mapsto Q)[x/R] &= y \mapsto Q[x/R] \\
(R_1 R_2)[x/R] &= R_1[x/R]\ R_2[x/R] \\
\mathbf{side}(A)[x/R] &= \mathbf{side}(A) \\
(\lambda x.P \star R_1)[x/R] &= \lambda x.P \star R_1 \\
(\lambda y.P \star R_1)[x/R] &= \lambda y.P[x/R] \star R_1[x/R] && \text{where } x \neq y \text{, and} \\
&&& \quad y \notin \mathit{fv}(R) \text{ or } x \notin \mathit{fv}(P, R_1) \\
(\lambda y.P \star R_1)[x/R] &= \lambda z.P[x/z][x/R] \star R_1[x/z][x/R] && \text{where } x \neq y \text{ and} \\
&&& \quad y \in \mathit{fv}(R) \text{ and } x \in \mathit{fv}(P, R_1) \\
(R_1.y)[x/R] &= \mathit{project}(R_1[x/R], y)
\end{aligned}$$

Table 11
Projection

$$\begin{aligned}
\mathit{project}(\epsilon, x) &= \text{error} \equiv \epsilon.x \\
\mathit{project}(R \cdot x \mapsto R_1, x) &= R_1 \\
\mathit{project}(R \cdot y \mapsto R_1, x) &= \mathit{project}(R, x) && \text{if } x \neq y \\
\mathit{project}(R \cdot (\lambda y.P \star R_1), x) &= \mathit{project}(R, x) \\
\mathit{project}(R \cdot \mathbf{side}(A), x) &= \mathit{project}(R, x) \\
\mathit{project}(R, x) &= R.x && \text{otherwise}
\end{aligned}$$

uated projection is returned. For instance $\mathit{project}(R_1 \cdot x \mapsto R_2, x) = R_2$ and $\mathit{project}(y, x) = y.x$.

If the form is extended to its right with a binding $x \mapsto R$, projection on x returns R. This is the important case that simplifies a projection expression. If the form is an extension with a service or an extension with a binding with a different label, projection proceeds recursively. In any other case, projection cannot be determined at specialization time and $\mathit{project}(R, x)$ denotes the projection $R.x$. Note that $R.x$ is a lazy form if $x \in \mathit{labels}(R)$, for instance $(x \mapsto \epsilon \cdot y).x \in \mathcal{R}$.

5.2.1 Combining Side-effects and Lazy Forms

The function *combine* gives a denotational semantics to pairs of side-effects and lazy forms. It does so by translating them to Piccola agents. The function

Table 12
Embedding side-effects and lazy terms

$$\begin{aligned}
embed(\epsilon) &= \epsilon \\
embed(x) &= x \\
embed(R_1 \cdot R_2) &= embed(R_1) \cdot embed(R_2) \\
embed(\mathbf{side}(A)) &= A \\
embed(x \mapsto R) &= x \mapsto embed(R) \\
embed(P.x) &= embed(P); x \\
embed(\lambda x.P \star R) &= \lambda x.combine(P, R)
\end{aligned}$$

$$\begin{aligned}
combine'(\epsilon) &= \mathbf{R} \\
combine'(P_1 \cdot P_2) &= combine'(P_1); combine'(P_2) \\
combine'(x \mapsto R_1 R_2) &= \mathbf{R} \cdot x \mapsto embed(R_1) embed(R_2) \\
combine'(x \mapsto R.x) &= \mathbf{R} \cdot x \mapsto embed(R.x) \\
combine'(x \mapsto P) &= \mathbf{R} \cdot x \mapsto combine'(P)
\end{aligned}$$

---

combine is defined as:

$$combine(P, R) = combine'(P); embed(R)$$

A side-effect and a lazy form are combined into a sandbox expression where the root context is the combined side-effect and the value is the embedded lazy form. The functions *embed* and $combine'$ are given in Table 12. The embedding is compositional except for abstractions that respect the special nature of lazy closures. Since an abstraction $\lambda x.P \star R$ itself contains a side-effect part and a lazy form value, the embedding is $\lambda x.combine(P, R)$.

The function $combine'(P)$ translates a side-effect into a functional agent. It replaces the sequential composition operator of the side-effect with a sandbox. Nested side-effects, applications and projections are combined into extensions of $\mathbf{R}$ with the embedded expression. Recall from the introduction that the root context $\mathbf{R}$ will contain the side-effects.

5.2.2 Separating Side-effects

We now discuss splitting of agents which is the heart of the specialization algorithm. The function $split(A, R)$ is defined in Table 13. The first few rules of the definition are straightforward. In rules (1-3), for example, the side-effect part is empty since the construction of a service has no side-effect. The lazy forms are the primitive services marked as side-effect services.

Table 13
Split Function

$$
\begin{aligned}
\mathit{split}(\mathbf{L}, R') &= (\epsilon, \mathbf{side}(\mathbf{L})) && && (1)\\
\mathit{split}(\mathbf{new}, R') &= (\epsilon, \mathbf{side}(\mathbf{new})) && && (2)\\
\mathit{split}(\mathbf{run}, R') &= (\epsilon, \mathbf{side}(\mathbf{run})) && && (3)\\
\mathit{split}(\mathit{hide}_x, R') &= (\epsilon, \mathbf{side}(\mathit{hide}_x)) && && (4)\\
\mathit{split}(\epsilon, R') &= (\epsilon, \epsilon) && && (5)\\
\mathit{split}(\mathbf{R}, R') &= (\epsilon, R') && && (6)\\
\mathit{split}(x{\mapsto}A, R') &= (P, x{\mapsto}R) && \text{where } \mathit{split}(A, R') = (P, R) && (7)\\
\mathit{split}(A; B, R') &= (P_1 \cdot P_2, R_2) && \text{where } \mathit{split}(A, R') = (P_1, R_1) && \\
&&& \text{and } \mathit{split}(B, R_1) = (P_2, R_2) && (8)\\
\mathit{split}(A \cdot B, R') &= (P_1 \cdot P_2, R_1 \cdot R_2) && \text{where } \mathit{split}(A, R') = (P_1, R_1) && \\
&&& \text{and } \mathit{split}(B, R') = (P_2, R_2) && (9)\\
\mathit{split}(\lambda x.A, R') &= (\epsilon, \lambda x.P \star R) && \text{where } \mathit{split}(A, R' \cdot x{\mapsto}x) = (P, R) && (10)
\end{aligned}
$$

$$
\mathit{split}(x, R') = \begin{cases} (\epsilon, \mathit{project}(R', x)) & \text{if } x \in \mathit{labels}(R') \\ (y{\mapsto}\mathit{project}(R', x), y) & \text{otherwise} \end{cases} \qquad (11)
$$

$$
\mathit{split}(AB, R') = \begin{cases} (P_1 \cdot P_2, R_3[x/R_2]) & \text{if } \mathit{service}(R_1) = \lambda x.\epsilon \star R_3 \\ (P_1 \cdot P_2 \cdot y{\mapsto}P_3[x/R_2], & \text{if } \mathit{service}(R_1) = \lambda x.P_3 \star R_3 \\ \quad \mathit{nest}(R_3, y, P_3)[x/R_2]) & \quad \text{and } P_3 \neq \epsilon \\ (P_1 \cdot P_2 \cdot y{\mapsto}\mathit{service}(R_1)R_2, y) & \text{otherwise} \end{cases} \qquad (12)
$$

where $(P_1, R_1) = \mathit{split}(A, R')$, $(P_2, R_2) = \mathit{split}(B, R')$ and y denotes a unique identifier.

The side-effect of evaluating the empty form (5) is the empty form and the result is the empty form. The result of evaluating **R** (6) is the current context R' and the side-effect is empty. Splitting a binding (7) works as follows: we first split A which yields a side-effect P and a result R. The side-effects are propagated and the resulting lazy form is the lazy binding $x{\mapsto}R$.

Evaluation of x (11) is done by projecting x in the current root context R'.

The most interesting case is specializing an application (equation 12 on Table 13). First — as is the case with extension — we split the agents A and B in the context of R'. This gives us two side-effects P_1 and P_2 and two lazy forms R_1 and R_2, respectively. The side-effects are composed in the right order, first P_1 then P_2. The predicate *service* : $\mathcal{R} \rightarrow \mathcal{R}$ (see Table 14) extracts the service bound in a term. For instance, $\mathit{service}(y{\mapsto}R_1 \cdot \lambda x.P \star R)$ is the abstraction $\lambda x.P \star R$.

Table 14
Service selection and nesting

$$\begin{aligned}
\mathit{service}(\epsilon) &= \text{error} \\
\mathit{service}(R \cdot \lambda x.P \star R_1) &= \lambda x.P \star R_1 \\
\mathit{service}(R \cdot \mathbf{side}(A)) &= \mathbf{side}(A) \\
\mathit{service}(R \cdot x \mapsto R_1) &= \mathit{service}(R) \\
\mathit{service}(R, x) &= R && \text{otherwise} \\
\\
\mathit{nest}(R, x, \epsilon) &= R \\
\mathit{nest}(R, x, P \cdot y \mapsto Q) &= \mathit{nest}(R[x/x.y], x, P)
\end{aligned}$$

Now, there are three possible ways to proceed, depending on the service of the application:

(1) If the service is referentially transparent, *i.e.*, its side-effect is empty, as in $\lambda x.\epsilon \star R_3$, then we inline the application.
(2) If the service contains side-effects, *i.e.*, $\lambda x.P_3 \star R_3$ where $P_3 \neq \epsilon$, then we introduce a new label y and bind it to the side effect term. We need to ensure that we can refer to the result of this application even if we invoke the same abstraction several times. Consequently, the lazy form has to lookup the result in the nested form by using a projection.
The function *nest* does the nesting of side-effects (see Table 14). The term $\mathit{nest}(R, x, P)$ is R where all y that are defined in P are replaced by the projection $x.y$.
(3) When the service cannot be determined at specialization time, we simply put the application into the side-effect and bind it to a unique label y.

This concludes the predicate *split* and the partial evaluation algorithm.

5.2.3 Example

Let us consider an example of splitting an application with a side effect.

Assume the application $f()$ appears within an abstraction where f is the passed argument. We have:

$$\mathit{split}(\lambda f.f(), \epsilon) \;=\; (\epsilon, \lambda f.y_1 \mapsto f() \star y_1)$$

Now we apply this abstraction to a form F. In the side effect and the lazy form we have to replace the variable f with the concrete argument F. The substitution yields $y_1 \mapsto F()$ and y_1.

Using such a unique label y_2, the side effect of the above application is

$y_2 \mapsto (y_1 \mapsto F())$ and the lazy form $y_2.y_1$.

Consider the following agent A. It defines a service f which calls a service g. The service f is applied twice, once on the empty form and once on the form u.

$$A \;=\; \mathbf{R} \cdot f \mapsto \lambda x.(c \mapsto (\underbrace{g(a \mapsto x)}_{y_1})); a \mapsto \underbrace{f()}_{y_2} \cdot b \mapsto \underbrace{fu}_{y_3}$$

The agent A contains three static applications. We associate unique identifier $y_{1...3}$ with each invocation. Let r be the initial context that contains the bindings for the unknown forms $r = g \mapsto g \cdot u \mapsto u$. Splitting the three applications yields:

$$\begin{aligned}
split(g(a \mapsto x), r \cdot x \mapsto x) &= (y_1 \mapsto g(a \mapsto x), c \mapsto y_1) \\
split(f(), r \cdot f \mapsto (...) &= (y_2 \mapsto (y_1 \mapsto g(a \mapsto \epsilon)), c \mapsto y_2.y_1) \\
split(fu, r \cdot f \mapsto (...) &= (y_3 \mapsto (y_1 \mapsto g(a \mapsto u)), c \mapsto y_3.y_1)
\end{aligned}$$

which gives

$$\begin{aligned}
split(A, r) = (&y_2 \mapsto (y_1 \mapsto g(a \mapsto \epsilon)) \cdot y_3 \mapsto (y_1 \mapsto g(a \mapsto u)), \\
&a \mapsto (c \mapsto y_2.y_1) \cdot b \mapsto (c \mapsto y_3.y_1))
\end{aligned}$$

The partial evaluation has inlined f and bound the side-effects to y_2 and y_3, respectively.

Observe that the nesting of side-effects ensures that we can access the side-effects from within the lazy form expression. If we apply the partial evaluation algorithm twice on the above expression, the nested side effects and projection are specialized:

$$split(combine(split(A, r)), r) = (y_1 \mapsto g(a \mapsto \epsilon) \cdot y_2 \mapsto g(a \mapsto u), a \mapsto (c \mapsto y_1) \cdot b \mapsto (c \mapsto y_2))$$

However, applying *split* twice does not linearize all nested side-effects since recursive service applications would introduce new nested side-effects at each specialization step.

5.3 Termination and Correctness

We now show that the partial evaluation algorithm is correct and terminates for all expressions. While termination is straightforward to show, correctness requires a bit of work. The important definition is that of referential transparency.

5.3.1 Termination

We can readily verify by structural induction on the domains for $\mathcal{A}, \mathcal{P}$ and $\mathcal{R}$ that the algorithm terminates. The important aspect for termination is the definition of the substitution given in Table 10. Consider the application xR_1 where we replace x with a user defined abstraction. For example

$$(xR_1)[x/\lambda z.P \star R] = (\lambda z.P \star R)R_1[x/\lambda z.P \star R]$$

It might be tempting to define the result of such a substitution as the result of splitting the application, as we have done for projection. However, this may lead to an infinite loop during the specialization process. Consider the term xx where we substitute the service $\lambda y.y_1 \mapsto yy \star y_1$ for x. When we split the substitute term, the process loops since the substitute contains another instance of the same expression.

5.3.2 Correctness

This property specifies that any closed agent is behaviourally equivalent to its specialized agent. This means:

$$\mathit{partial}(A) \approx A \qquad \text{for } A \text{ closed} \tag{1}$$

In order to prove this equation, we show by induction over A that for all functional agents A and lazy form expressions R, the following holds

$$\mathit{combine}(\mathit{split}(A, R)) \approx \mathit{embed}(R); A \tag{2}$$

Then, equation (1) is a special case of equation (2) where R is the empty form. However, in order to prove the induction steps for this equation we need a stronger property, namely that for all A and R, there are two agents A_1 and A_2 such that:

$$\mathit{combine}(\mathit{split}(A, R)) \approx A_1; A_2$$

and all free labels in A_2 are defined by A_1 and A_2 does not contain any applications which cause side-effects or undefined projections. Whenever A_1 reduces to a barb with value F, there exists a form value G such that the expression $F; A_2$ is equivalent to G. The formal definition of this property is that A_2 is referentially transparent in A_1.

Definition 6 *A Piccola agent B is* referentially transparent *in an agent A, if for any agent C and vector of names $\tilde{c}$ with $\nu\tilde{c}.C \mid A\Downarrow$, written as canonical* *agent:*

$$\nu\tilde{c}.(C \mid A) \ \Rightarrow\ \nu\tilde{c}'.(M_1 \mid ... \mid M_n \mid A_1 \mid ... \mid A_{k-1} \mid F)$$

there exists a form G such that:

$$F; B \approx G$$

The fact that B is referentially transparent within A is written $A \vdash B$. $\epsilon \vdash B$ is written as $\vdash B$.

Referential transparency formalizes the idea behind lazy forms. Whenever A is reduced to a barb with value F, the agent $F; B$ is equivalent to a form G. In other words, when A reduces to F then $A; B$ reduces to G. This notion rules out the possibility of B containing a side-effect. It also guarantees that all required labels of B are defined by A.

The word *all* in the above definition is important. It is not enough to find an equivalent G just for some possible reductions. For instance

$$c(x\mapsto\epsilon) \mid c() \mid c?; x \rightarrow c() \mid x\mapsto(); x \approx c() \mid \epsilon$$

But $c(x\mapsto\epsilon) \mid c() \mid c? \not\vdash x$.

Obviously, all forms are referentially transparent, thus $\vdash F$ for any form F. We can prove by induction on A that $\mathit{split}(A, R)$ generates tuples that are referentially transparent. If $\mathit{split}(A, R) = (P, R)$ then $\mathit{combine}'(P) \vdash \mathit{embed}(R)$ [2].

6 Related Work

The Piccola calculus extends the asynchronous π-calculus with higher-order abstractions and first-class environments.

6.1 π-calculus

The π-calculus [32] is a calculus of communicating systems in which one can naturally express processes with a changing structure. Its theory has been thoroughly studied and many results relate other formalisms or implementations to it. The affinity between objects and processes, for example, has been treated by various authors in the context of the π-calculus [21,53]. The Pict experiment has shown that the π-calculus is a suitable basis for programming many high-level constructs by encodings [42].

For programming and implementation purposes, synchronous communication seems uncommon and can generally be encoded by using explicit acknowledgments (cf. [21]). Moreover, asynchronous communication has a closer correspondence to distributed computing [54]. Furthermore, in the π-calculus the asynchronous variant has the pleasant property that equivalences are simpler than for the synchronous case [16]. Input-guarded choice can be encoded and is fully abstract [34]. For these reasons we adopt asynchronous channels in the Piccola calculus.

6.2 Higher-order abstractions

Programming directly in the π-calculus is often considered like programming a concurrent assembler. When comparing *programs* written in the π-calculus with the lambda-calculus it seems like lambda abstractions scale up, whereas sending and receiving messages does not scale well. There are two possible solutions proposed to this problem: we can change the metaphor of communication or we can introduce abstractions as first-class values.

The first approach is advocated by the Join-calculus [17]. Communication does not happen between a sender and a receiver, instead a join pattern triggers a process on consumption of several pending messages. The Blue calculus of Boudol [10] changes the receive primitive into a definition which is defined for a scope. By that change, the Blue calculus is more closely related to functions and provides a better notion for higher-order abstraction. Boudol calls it a continuation-passing calculus.

The other approach is adopted by Sangiorgi in the HOπ-calculus. Instead of communicating channels or tuples of channels, processes can be communicated as well. Surprisingly, the higher-order case has the same expressive power as the first-order version [44,45]. In the Piccola calculus we take the second approach and reuse existing encodings of functions into the π-calculus as in Pict. The motivation for this comes from the fact that the HOπ-calculus itself can be encoded in the first-order case.

6.3 Asymmetric parallel composition

The semantics of asynchronous parallel composition is used in the concurrent object calculus of Gordon and Hankin [19] or the (asymmetric) Blue calculus studied by Dal-Zilio [14]. In the higher-order π-calculus the evaluation order is orthogonal to the communication semantics [45]. In Piccola, evaluation strategy interferes with communication, therefore we have to fix one for meaningful terms. For Piccola, we define strict evaluation which seems appropriate and more common for concurrent computing.

6.4 Record calculus

When modeling components and interfaces, a record-based approach is the obvious choice. We use *forms* [27,28] as an explicit notion for extensible records. Record calculi are studied in more detail for example in [12,43].

In the λ-calculus with names of Dami [15] arguments to functions are named. The resulting system supports records as arguments instead of tuples as in the classical calculus. The λN-calculus was one of the main inspiration for our work on forms without introspection. An issue omitted in our approach is record typing. It is not clear how far record types with subtyping and the runtime acquisition can be combined. An overview of record typing and the problems involved can be found for example in [12].

6.5 Explicit environments

An explicit environment generalizes the concept of explicit substitution [1] by using a record like structure for the environment. In the environment calculus of Nishizaki, there is an operation to get the current environment as a record and an operator to evaluate an expression using a record as environment [40,46]. Projection of a label x in a record R then corresponds to evaluating the script x in an environment denoted by R. The reader may note that explicit environments subsume records. This is the reason why we call them forms in Piccola instead of just records. Handling the environment as a first-class entity allows us to define concepts like modules, interfaces and implementation for programming in the large within the framework.

To our knowledge, the language Pebble of Burstall and Lampson was the first to formally show how to build modules, interfaces and implementation, abstract data types and generics on a typed lambda calculus with bindings, declarations and types as first-class values [11].

6.6 Other approaches

Zenger has developed a component calculus [55] that extends Featherweight Java [22] with primitives to dynamically build, extend and compose software components. The novelty of Zenger's calculus is that components are composed *implicitly* on the basis of the type compatibility of component interfaces, rather than by establishing explicit connections.

Pahl has developed composition and replacement calculus based on the π-calculus [41]. In this approach, types are used to characterize values that may be passed along ports, and *contracts* (pre- and post-conditions) are used to determine whether provided and required services match. The calculus is used to reason about when components can be replaced in dynamically evolving systems.

A very different model is offered by $\rho\epsilon\omega$ (AKA Reo) [7], a calculus of component connectors. Reo is algebraic in flavour, and provides various *connectors* that coordinate and compose streams of data. Primitive connectors can be composed using the Reo operators to build higher-level connectors. In contrast to process calculi, Reo is well-suited to compositional reasoning, since connectors can be composed to yield new connectors, and properties of connectors can be shown to compose. Data communicated along streams are uninterpreted in Reo, so it would be natural to explore the application of Reo to streams of forms.

6.7 Precursors to the Piccola calculus

In our earlier work on the foundations of Piccola, we specified the semantics of Piccola in terms of translations to $\pi\mathcal{L}$ (the π calculus with labels) [27,28] or to the form calculus [50]. The difference between $\pi\mathcal{L}$ and the form calculus is that the latter allows hiding of labels and forms and it contains a testing primitive for labels.

The Piccola calculus is better suited to give a direct semantics to the Piccola language. The enhanced expressiveness of the Piccola calculus with respect to the form- and the $\pi\mathcal{L}$-calculus are as follows:

- *Form extension.* In $\pi\mathcal{L}$ and the form calculus we have separate primitives to extend a form with either a single binding or a separate form. In the Piccola calculus there is a single extension operator $\cdot$ for asymmetric form concatenation.
- *Label hiding.* The Piccola calculus introduces label hiding as a primitive service. It cannot be expressed in $\pi\mathcal{L}$ or the form calculus.
- *Nested forms.* The syntax for binding is simplified in the Piccola calculus since nested forms are primitive. In the form calculus and $\pi\mathcal{L}$ nested forms must be encoded as constant services.
- *Label matching.* The form calculus provides a matching construct that allows an agent to check whether a given form contains a label. In the Piccola calculus we do not need this primitive since inspect is more expressive and

allows us to build a checking predicate within the language. In contrast, neither in the form calculus nor in $\pi\mathcal{L}$ can we iterate over all the labels in a form.

- *Higher-order abstractions.* In Piccola, lambda abstractions are specified as user services. In $\pi\mathcal{L}$ and the form calculus, such abstractions must be encoded using a replicated agent.
- *Value semantics.* In the Piccola calculus, agents that are not stuck reduce to value. ("Everything is a form.") In $\pi\mathcal{L}$ and in the form calculus, a (parallel) process does not denote a value.

7 Concluding Remarks

We have presented the Piccola calculus, a high-level calculus for modeling software components that extends the asynchronous π-calculus with explicit namespaces, or *forms.* The calculus serves as the semantic target for Piccola, a language for composing software components that conform to a particular compositional style. JPiccola, the Java implementation of Piccola, is realized by translation to an abstract machine that implements the Piccola calculus.

The Piccola calculus is not only helpful for modeling components and connectors, but it also helps to reason about the Piccola language implementation and about compositional styles. Efficient language bridging between Piccola and the host language (Java or Squeak) is achieved by means of partial evaluation of language wrappers. The partial evaluation algorithm is proved correct with the help of the Piccola calculus.

Different compositional styles make different assumptions about software components. Mixing incompatible components can lead to compositional mismatches. The Piccola calculus can help to bridge mismatches by supporting reasoning about wrappers that adapt component contracts from one style to another. We have studied two extended examples. The first concerns synchronization wrappers that express the synchronization constraints assumed by a component. The second study compares push- and pull-flow filters and demonstrates how to adapt pull-filters so that they work correctly in a push-style [2].

One shortcoming of our work so far is the lack of a type system. We have been experimenting with a system of *contractual types* [35] that expresses both the *provided* as well as the *required* services of a software component. Contractual types are formalized in the context of the *form calculus*, which can be seen as the Piccola calculus minus agents and channels. Contractual types have been integrated into the most recent distribution of JPiccola [26].

Acknowledgments

We gratefully acknowledge the financial support of the Swiss National Science Foundation for projects No. 20-61655.00, “Meta-models and Tools for Evolution Towards Component Systems”, and 2000-067855.02, “Tools and Techniques for Decomposing and Composing Software”.

References

[1] Martín Abadi, Luca Cardelli, Pierre-Louis Curien, and Jean-Jacques Lévy. Explicit substitutions. *Journal of Functional Programming*, 1(4):375–416, October 1991.

[2] Franz Achermann. *Forms, Agents and Channels — Defining Composition Abstraction with Style.* PhD thesis, University of Berne, January 2002.

[3] Franz Achermann, Stefan Kneubühl, and Oscar Nierstrasz. Scripting coordination styles. In António Porto and Gruia-Catalin Roman, editors, *Coordination '2000*, volume 1906 of *LNCS*, pages 19–35, Limassol, Cyprus, September 2000. Springer-Verlag.

[4] Franz Achermann, Markus Lumpe, Jean-Guy Schneider, and Oscar Nierstrasz. Piccola — a small composition language. In Howard Bowman and John Derrick, editors, *Formal Methods for Distributed Processing — A Survey of Object-Oriented Approaches*, pages 403–426. Cambridge University Press, 2001.

[5] Franz Achermann and Oscar Nierstrasz. Explicit Namespaces. In Jürg Gutknecht and Wolfgang Weck, editors, *Modular Programming Languages*, volume 1897 of *LNCS*, pages 77–89, Zürich, Switzerland, September 2000. Springer-Verlag.

[6] Franz Achermann and Oscar Nierstrasz. Applications = Components + Scripts — A Tour of Piccola. In Mehmet Aksit, editor, *Software Architectures and Component Technology*, pages 261–292. Kluwer, 2001.

[7] Farhad Arbab and Farhad Mavaddat. Coordination through channel composition. In F. Arbab and C. Talcott, editors, *Coordination Languages and Models: Proc. Coordination 2002*, volume 2315 of *Lecture Notes in Computer Science*, pages 21–38. Springer-Verlag, April 2002.

[8] Kenichi Asai, Hidehiko Masuhara, and Akinori Yonezawa. Partial evaluation of call-by-value λ-calculus with side-effects. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 12–21, Amsterdam, the Netherlands, June 1997.

[9] Gérard Berry and Gérard Boudol. The chemical abstract machine. *Theoretical Computer Science*, 96:217–248, 1992.

[10] Gérard Boudol. The pi-calculus in direct style. In *Conference Record of POPL '97*, pages 228–241, 1997.

[11] Rod Burstall and Butler Lampson. A kernel language for abstract data types and modules. *Information and Computation*, 76(2/3), 1984. Also appeared in Proceedings of the International Symposium on Semantics of Data Types, Springer, LNCS (1984), and as SRC Research Report 1.

[12] Luca Cardelli and John C. Mitchell. Operations on records. In Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming. Types, Semantics and Language Design*, pages 295–350. MIT Press, 1993.

[13] Charles Consel and Olivier Danvy. Tutorial notes on partial evaluation. In *Conference Record of POPL '93*, pages 493–501. ACM, January 1993.

[14] Silvano Dal-Zilio. *Le calcul bleu: types et objects.* Ph.D. thesis, Université de Nice — Sophia Antipolis, July 1999. In french.

[15] Laurent Dami. *Software Composition: Towards an Integration of Functional and Object-Oriented Approaches.* Ph.D. thesis, University of Geneva, 1994.

[16] Cédric Fournet and Georges Gonthier. A hierarchy of equivalences for asynchronous calculi. In *Proceedings of ICALP '98*, pages 844–855, 1998.

[17] Cédric Fournet, Georges Gonthier, Jean-Jacques Lévy, Luc Maranget, and Didier Rémy. A calculus of mobile agents. In *Proceedings of the 7th International Conference on Concurrency Theory (CONCUR '96)*, volume 1119 of *LNCS*, pages 406–421. Springer-Verlag, August 1996.

[18] David Garlan, Robert Allen, and John Ockerbloom. Architectural mismatch: Why reuse is so hard. *IEEE Software*, 12(6):17–26, November 1995.

[19] Andrew D. Gordon and Paul D. Hankin. A concurrent object calculus: Reduction and typing. In *Proceedings HLCL '98*. Elsevier ENTCS, 1998.

[20] J. Roger Hindley and Jonathan P. Seldin. *Introduction to Combinatory Logic and Lambda Calculus.* Cambridge University Press, 1986.

[21] Kohei Honda and Mario Tokoro. An object calculus for asynchronous communication. In Pierre America, editor, *Proceedings ECOOP '91*, volume 512 of *LNCS*, pages 133–147, Geneva, Switzerland, July 1991. Springer-Verlag.

[22] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM TOPLAS*, 23(3):396–450, May 2001.

[23] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. Back to the future: The story of Squeak, A practical Smalltalk written in itself. In *Proceedings OOPSLA '97*, pages 318–326. ACM Press, November 1997.

[24] Neil J. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice-Hall, 1993.

[25] Stefan
Kneubühl. Implementing coordination styles in piccola. Informatikprojekt, University of Bern, February 2001.

[26] Stefan Kneubühl. Typeful compositional styles. Diploma thesis, University of Bern, April 2003.

[27] Markus Lumpe. *A Pi-Calculus Based Approach to Software Composition.* Ph.D. thesis, University of Bern, Institute of Computer Science and Applied Mathematics, January 1999.

[28] Markus Lumpe, Franz Achermann, and Oscar Nierstrasz. A Formal Language for Composition. In Gary Leavens and Murali Sitaraman, editors, *Foundations of Component Based Systems*, pages 69–90. Cambridge University Press, 2000.

[29] Mark Lutz. *Programming Python.* O'Reilly & Associates, Inc., 1996.

[30] Massimo Merro and Davide Sangiorgi. On asynchrony in name-passing calculi. In Kim G. Larsen, Sven Skyum, and Glynn Winskel, editors, *25th Colloquium on Automata, Languages and Programming (ICALP) (Aalborg, Denmark)*, volume 1443 of *LNCS*, pages 856–867. Springer-Verlag, July 1998.

[31] Wolfgang De Meuter. Agora: The story of the simplest MOP in the world — or — the scheme of object–orientation. In J. Noble, I. Moore, and A. Taivalsaari, editors, *Prototype-based Programming.* Springer-Verlag, 1998.

[32] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, part I/II. *Information and Computation*, 100:1–77, 1992.

[33] Robin Milner and Davide Sangiorgi. Barbed bisimulation. In *Proceedings ICALP '92*, volume 623 of *LNCS*, pages 685–695, Vienna, July 1992. Springer-Verlag.

[34] Uwe Nestmann and Benjamin C. Pierce. Decoding choice encodings. In Ugo Montanari and Vladimiro Sassone, editors, *CONCUR '96: Concurrency Theory, 7th International Conference*, volume 1119 of *LNCS*, pages 179–194, Pisa, Italy, August 1996. Springer-Verlag.

[35] Oscar Nierstrasz. Contractual types. Technical Report IAM-03-004, Institut für Informatik, Universität Bern, Switzerland, 2003.

[36] Oscar Nierstrasz and Franz Achermann. Supporting Compositional Styles for Software Evolution. In *Proceedings International Symposium on Principles of Software Evolution (ISPSE 2000)*, pages 11–19, Kanazawa, Japan, November 2000. IEEE.

[37] Oscar Nierstrasz and Franz Achermann. A calculus for modeling software components. In *FMCO 2002 Proceedings*, volume 2852 of *LNCS*, pages 339–360. Springer-Verlag, 2003.

[38] Oscar Nierstrasz, Franz Achermann, and Stefan Kneubühl. A guide to JPiccola. Technical Report IAM-03-003, Institut für Informatik, Universität Bern, Switzerland, June 2003.

[39] Oscar Nierstrasz and Theo Dirk Meijler. Requirements for a composition language. In Paolo Ciancarini, Oscar Nierstrasz, and Akinori Yonezawa, editors, *Object-Based Models and Langages for Concurrent Systems*, volume 924 of *LNCS*, pages 147–161. Springer-Verlag, 1995.

[40] Shin-ya Nishizaki. Programmable environment calculus as theory of dynamic software evolution. In *Proceedings ISPSE 2000*. IEEE Computer Society Press, 2000.

[41] Claus Pahl. A pi-calculus based framework for the composition and replacement of components. In *Workshop on Specification and Verification of Component-Based Systems (OOPSLA 2001)*, 2001.

[42] Benjamin C. Pierce and David N. Turner. Pict: A programming language based on the pi-calculus. In G. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT Press, May 2000.

[43] Didier Rémy. *Typing Record Concatenation for Free*, chapter 10, pages 351–372. MIT Press, April 1994.

[44] Davide Sangiorgi. *Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigms*. Ph.D. thesis, Computer Science Dept., University of Edinburgh, May 1993.

[45] Davide Sangiorgi. Asynchronous process calculi: the first-order and higher-order paradigms (tutorial). *Theoretical Computer Science*, 253, 2001.

[46] Masahiko Sato, Takafumi Sakurai, and Rod M. Burstall. Explicit environments. In Jean-Yves Girard, editor, *Typed Lambda Calculi and Applications*, volume 1581 of *LNCS*, pages 340–354, L'Aquila, Italy, April 1999. Springer-Verlag.

[47] Nathanael Schärli. Supporting pure composition by inter-language bridging on the meta-level. Diploma thesis, University of Bern, September 2001.

[48] Nathanael Schärli and Franz Achermann. Partial evaluation of inter-language wrappers. In *Workshop on Composition Languages, WCL '01*, September 2001.

[49] Andreas Schlapbach. Enabling white-box reuse in a pure composition language. Diploma thesis, University of Bern, January 2003.

[50] Jean-Guy Schneider. *Components, Scripts, and Glue: A conceptual framework for software composition*. Ph.D. thesis, University of Bern, Institute of Computer Science and Applied Mathematics, October 1999.

[51] Jean-Guy Schneider and Oscar Nierstrasz. Components, scripts and glue. In Leonor Barroca, Jon Hall, and Patrick Hall, editors, *Software Architectures — Advances and Applications*, pages 13–25. Springer-Verlag, 1999.

[52] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, 1996.

[53] David Walker. Objects in the π-calculus. *Information and Computation*, 116(2):253–271, February 1995.

[54] Pawel T. Wojciechowski. *Nomadic Pict: Language and Infrastructure Design for Mobile Computation.* PhD thesis, Wolfson College, University of Cambridge, March 2000.

[55] Matthias Zenger. Type-safe prototype-based component evolution. In *Proceedings ECOOP 2002*, volume 2374 of *LNCS*, pages 470–497, Malaga, Spain, June 2002. Springer Verlag.