A coordination language for mobile components

Farhad Arbab and Marcello M. Bonsangue
Centrum voor Wiskunde en Informatica, P.O. Box 94079, 1090 GB Amsterdam, The Netherlands
e-mails: farhad@cwi.nl and marcello@cwi.nl

Frank S. de Boer
Utrecht University, P.O. Box 80.089, 3508 TB Utrecht, The Netherlands
e-mail: frankb@cs.uu.nl

Keywords: Component, class, object, mobility, models for Internet applications, security, π-calculus.

Abstract

In this paper we present the $\sigma\pi$ coordination language, a core language for specifying dynamic networks of components. The language is inspired by the Manifold coordination language and by the π-calculus. The main concepts of the language are components, classes, objects and channels. A program in $\sigma\pi$ consists of a number of components, where each component is a collection of classes separable from its original context and re-usable in any other context. An object is an instance of a class that executes in parallel with the other objects active in the system. The $\sigma\pi$ language differs from other models of object-oriented systems mainly in its treatment of communication and mobility: communication is anonymous via synchronous or asynchronous channels, while mobility is obtained by moving channels in the virtual space of linked objects. Thus, a channel is a transferable capability of communication, and objects are mobile in the sense that their communication possibilities may change during a computation. The language $\sigma\pi$ itself does not impose exogenous coordination, meaning that the coordination primitives affecting each object can be executed within the computation of the object itself. However, only simple restrictions on the class-definitions of a $\sigma\pi$ program suffice to enforce a separation between computation and coordination. Interaction typically occurs anonymously and under the full control of the objects involved. This make it easier to deal with Internet application where security policies must be enforced in view of the possibilities of attacks.

1 Introduction

Most of the software produced today is purpose-built, monolithic, and one of a kind. Frequently, the systems built are so complex that individual pieces of software cannot be reused easily because their code strongly depends on that of the system in which they are integrated.

Component-based software engineering offers a solution to this problem by emphasizing modular architecture for partial development of systems and incremental enhancement of functionality by adding or replacing components.

A component is a basic unit of functionality describing a common structure and behavior of a system. It must be separable from its original context and usable in any other context; that is, a component must be designed to be composed [23]. The property of components that allows their use as separable elements of systems permits applications to be adapted by changing existing components or introducing new ones.

Component-based software is often implemented using a collection of components and a specific script (or coordination) language that prescribes how components are plugged together. Although it is possible to realize component-based software without employing object oriented technology, object oriented languages like C++ [27] and Java [16] provide a rather sophisticated support for component software.

In this paper, we present $\sigma\pi$, a component-based core-language for distributed and mobile processes. The language is based on the formal notions of component, class and object. A program in $\sigma\pi$ consists of a number of components, where each component is a collection of classes separable as a whole from its original context and re-usable in any other context. Objects are created dynamically, and are active and distributed meaning that their execution proceeds in parallel with that of all other objects active in the system. The $\sigma\pi$ language differs from other models of parallel object-oriented systems like POOL [4] or the actor model [3] mainly in its treatment of interaction. Objects communicate anonymously (in fact, objects do not even have identities) via synchronous or asynchronous channels. As such, the language $\sigma\pi$ provides a more appropriate basis for modeling the coordination of components.

In $\sigma\pi$ one can describe networks that reconfigure themselves because of a uniform treatment of ordinary values and references to channels and class names. A channel is a transferable capability of communication, and objects are mobile in the sense that their communication possibilities may change during the course of a computation. In this sense mobility is obtained as in the π-calculus [22] by moving channels in the virtual space of linked objects.

The language $\sigma\pi$ itself does not enforce exogenous coordination, meaning that the primitives for plugging the components together can be intermixed with the other primitives used for describing the behavior of a component [26]. In other words, the $\sigma\pi$ language itself does not embody specific constructs to enforce the dichotomy implied by the paradigm 'application = components + scripts'. Instead, $\sigma\pi$ focuses on a coordination mechanism wherein the plugging of components depends on the data exchanged through component interactions. However, simple restrictions may be imposed on the structure of the classes of a $\sigma\pi$ program in order to enforce a separation between computation and coordination.

The structure of the paper is as follows. In section 2, we present the language $\sigma\pi$. We briefly describe the language in terms of its different levels of abstraction for specifying an application, and then we discuss how coordination is achieved. In section 3, we define a formal operational semantics for the language in two steps. First, we define the semantics of each object in isolation in terms of a labeled transition system. Then, we combine all these transition systems in order to describe the behavior of a single program in $\sigma\pi$. In Section 4, we illustrate programming in the $\sigma\pi$ language by two small examples. In Section 5, we illustrate how $\sigma\pi$ conceptually supports the coordination needed for Internet applications. Finally, in Section 6, we present some perspective on future work and relate $\sigma\pi$ to other work on communicating and mobile agent systems.

2 The language $\sigma\pi$

The $\sigma\pi$ language is a basic language for describing dynamically changing networks of active concurrent processes. Its basic concepts are components, classes, objects, and communication channels.

2.1 Components, classes and objects

An *object* is a run-time entity with an encapsulated state, capable of offering services to its environment. In class-based object-oriented languages, like Java, the behavior of all objects is described by a finite collection of static entities, the classes. Thus, a *class* provides a generic description of the behavior of its instances, the objects. Components provide static building blocks similar to classes. A *component* is a collection of classes, together with an interface specifying the name of the classes that can be activated by other components. The classes in the interface play the role of the component application program interface, describing the data and channels that a component needs in order to correctly function and the channels where the return values are provided by the component. Thus, a component, as a collection of classes (some of which may be known to other components) provides an abstraction from its internal class-structure. Clearly, each class can be seen as a component, but by encapsulating several classes into a component one achieves more flexibility, since this facilitates the reuse of a substantial part of an application within another.

A $\sigma\pi$ program consists of a collection of components, together with a designated root-component and a name of a root-class in its interface. The execution of the program starts by creating an object instance of the root-class of this root-component. Some of the class names available at the interfaces of the components may be provided to the object starting the computation by the initial environment.

2.2 The coordination space

Objects are the basic computational entities in the execution of a program. An object operates on its internal data which is not accessible to other objects. Objects interact only by means of sending and receiving data through channels. Channels are created dynamically. In fact, the creation of an object consists of the creation of a channel which connects it with its creator. This channel has a unique identity which is initially known only to the created object and its creator. As with any channel, the identity of this initial channel too can be communicated to other objects via other channels. Thus, we see that channels are dynamic entities that move in the virtual space of linked objects. As in the π calculus, mobility is obtained by abstracting from the notion of physical location: the identity of a channel can be sent via a channel to another active object. In this way, no interposition of specialized middleware – like CORBA [24] or DCOM [21] – is necessary.

We distinguish two kinds of channels: asynchronous and synchronous. Asynchronous channels enforce a decoupling between the parties engaged in communication in terms of identities (communication is anonymous as objects themselves do not have a referable identity), location (there is no notion of physical location), and time (communication is asynchronous). The same holds also for synchronous channels, except for the time decoupling, because the two interacting partners are forced to execute their synchronization actions at the same time. Any data can pass through the channels from one object to another, including class names. Passing a class name can be used to make an object aware of the existence of other parts of the system, while passing a reference to a channel supports dynamic reconfiguration and mobility.

2.3 The syntax

A program in the language $\sigma\pi$ is a collection of components. A component itself, in turn, is a collection of class definitions, where a class definition consists of an association of a unique name taken from a set CN of *class names* (with c as its typical element), with a statement describing the behavior of the objects that belong to that class. We require that *root* is the name of one of the classes, an instance of which will start the execution of the program. In this section we introduce the formal syntax of the class statements.

The statement of a class is executed by object instances of that class. Objects posses some internal data that are stored in their *variables*. Variables of objects are *private*, i.e., the data stored in the variable of an object is not accessible by another object, even if both objects are instances of the same class. We denote by Var, with its typical elements $x, y, \ldots$, the set of all variables of all instances of all classes. The value of a variable can be either an element of a specific data type, like integer or boolean, a class name, or a reference to a channel. We denote by T, with its typical element t, the set of *data types*, including bool, the boolean type, chn, the channel type, and cls, the class type. The set of variables Var is partitioned by the set of types T. We denote by Var_t the subset of variables in Var of type $t \in T$. For every object, we assume a designated variable (locally) named chn to be in Var_{chn}. This variable will, upon creation of an object, contain a reference to a channel connecting it to its creator.

Objects can interact only by sending and receiving messages via undirected channels. A message contains exactly one value; this can be of any type, including channel references or class names. There are two basic types of channels: *synchronous* and *asynchronous*. A synchronous channel is a zero-length buffer that can receive a value only if it can be delivered immediately. An asynchronous channel is an (unbounded) FIFO buffer. We denote by $CM = \{\mathsf{syn}, \mathsf{asy}\}$ the set of *communication modes*.

For each data type $t \in T$, there exists a set Exp_t of *local expressions*. Typical examples are expressions like $x + y$, $x = y$, or $x > 4$. The evaluation of a local expression by an object depends only on its current internal state. Moreover, we assume that the evaluation of an expression $e \in Exp_t$ by an object always terminates, resulting in a value of type t. Furthermore, no side-effects result from such evaluation.

The set *Act* of *primitive actions* of objects is defined by the following grammar:

$$
\begin{array}{lcll}
a & ::= & x{:}=e & x \in Var_t, e \in Exp_t \\
 & | & x{:}=\mathsf{new}(e,m) & x \in Var_{\mathsf{chn}}, e \in Exp_{\mathsf{cls}}, m \in CM \\
 & | & x!y & x \in Var_{\mathsf{chn}}, y \in Var_t \\
 & | & x?y & x \in Var_{\mathsf{chn}}, y \in Var_t .
\end{array}
$$

Primitive actions are the basic operations of $\sigma\pi$. The execution of an assignment $x{:}=e$ by an object consists of assigning the value resulting from evaluation of the expression e to its variable x.

The execution of the statement $x{:}=\mathsf{new}(e,m)$ by an object has the consequent of creating a new object and a new channel which, initially, forms a link between the two (creator and created) objects. The name of the class of which the new object is an instance is the result of the evaluation of the expression e. The communication mode of the new channel is specified by m. A reference to this new channel is assigned both to the variable x of the creator object, as well as to the variable chn of the newly created object.

The actions $x!y$ and $x?y$ are the only communication primitives in $\sigma\pi$. The execution of the output action $x!y$ sends the value stored in the variable y to the channel referred to by the variable x. The execution of the input action $x?y$ reads a value of a type compatible with the variable y from the channel referred to by the variable x. The read value is taken out of the channel and then stored in the variable y. The input action is blocking, meaning that its execution blocks until a value of the appropriate type is available through the specified channel.

Primitive actions can be composed yielding statements S using the standard operations of sequential composition, conditional selection, alternative choice, and iteration. Additionally, we assume guarded statements of the form '$q \rightarrow S$', where q denotes an *external expression*, the evaluation of which involves external information about the state of one of the known channels. Informally, the execution of the statement S in a guarded statement $q \rightarrow S$ depends on the evaluation to true of the external expression q. This evaluation takes place in two phases: first local, and then external. The purpose of the local evaluation is to substitute values for all variables that appear in q. The result of the local evaluation is a single channel reference k, and a single closed external expression q'. The purpose of the external evaluation is to evaluate q' on the state of the channel k. The external evaluation always terminates and returns a boolean value. An example of an external expression is $\mathsf{empty}(x)$, for x a variable of type chn, which tests whether the channel referred to by the variable x is empty. We denote by *Query* the set of all external expressions, also called queries, by $CQuery \subseteq Query$ the set of all closed external expressions, and by *Stat* the set of statements, with its typical element S, which describe the behavior of objects.

3 A transition system semantics

We now define a formal operational semantics for programs in the coordination language $\sigma\pi$. Following [8] we proceed in two steps: first we define a labeled local transition system for the description of the behavior of an object in isolation, and then we define an unlabeled global transition system which describes the behavior of a dynamic network of objects. In the local transition system, the interactions of an object with the network involve assumptions about the behavior of the network. These assumptions are reflected in the labels of their corresponding local transitions. Different assumptions give rise to different transitions, and as such, the unknown global network induces a non-determinism on the local transition system of an object. At the level of the global transition system, this non-determinism is filtered out.

3.1 Semantics of objects

To define the semantics of objects, we assume for each type $t \in T$ a set Val_t of values of type t, with its typical element v. Thus, Val_{bool}, for instance, denotes the two-element boolean set $\{tt, \mathit{ff}\}$. Furthermore, $Val_{\mathsf{cls}} = CN$, i.e., class names denotes themselves, and Val_{chn} is some infinite set of channel identities. We denote by *Val* the set of all basic values, including the boolean values, the class names, and the channel identities.

The set of local states Σ of an object is given by $\Sigma = Var \rightarrow (Val \cup \{\perp\})$, where *Var* denotes the set of all variables and '$\perp$' stands for 'undefined' or 'uninitialized'. As usual, for every state $s \in \Sigma$, variable $x \in Var$ and value $v \in Val$, we denote by $s[v/x]$ the state which evaluates to $s(y)$ for every $y \neq x$ and evaluates to v otherwise.

We postulate the existence of a local evaluation function for local and external expressions

$$
\begin{array}{l}
\mathcal{E}{:}(Exp \rightarrow (\Sigma \rightarrow (Val \cup \{\perp\}))) \cup \\
\quad (Query \rightarrow (\Sigma \rightarrow ((Val \times CQuery) \cup \{\perp\}))) .
\end{array}
$$

The function $\mathcal{E}$ maps, for each state s, expressions e of type t to values of the same type t or $\perp$ (which stands for 'undefined'), and queries q to values of type chn together with a closed external expression or $\perp$. This function gives the semantics of the local expressions and queries with respect to the local state of an object: $\mathcal{E}(e)(s) = v$ means that in the state s, the local expression e has the value v, $\mathcal{E}(q)(s) = \langle v, q' \rangle$ means that in the state s, the external expression q has to be evaluated as closed expression q' in the channel v. The evaluation of an expression or a query in a certain state to $\perp$ means that it is undefined in that state.

The dynamics of an object of class c is described in terms of a labeled transition relation between configurations of the form $\langle S, s \rangle$, with $S \in Stat$ and $s \in \Sigma$. Such a pair $\langle S, s \rangle$ indicates that the statement S is to be executed in the internal state s.

Assignment:

$$\langle x{:}=e, s \rangle \xrightarrow{\tau} \langle \mathsf{nil}, s[\mathcal{E}(e)(s)/x] \rangle$$

The state that results from the execution of an assignment is obtained by changing the value of the variable x in the original state s to the value resulting from the evaluation of the expression e in the original state s. The label τ on the transition indicates that this action is local and has no effect on the environment.

Object creation: If $\mathcal{E}(e)(s) = c \in CN$ then for *any* $v \in Val_{\mathsf{chn}}$

$$\langle x{:}=\mathsf{new}(e,m), s \rangle \xrightarrow{\langle v:m,c \rangle} \langle \mathsf{nil}, s[v/x] \rangle$$

The local effect of creating a new object of class c is the assignment of a channel name v to the variable x. Here, v is an arbitrary channel name chosen under the assumption that it is indeed a globally new name. This assumption is verified at the level of the global transition system. The label on the transition also contains the information about the communication mode of this channel. Note that this action takes place only if the evaluation of the expression e returns a correct class name that is different than $\perp$.

Output: If $s(x) = v \in Val_{\mathsf{chn}}$ and $s(y) = w \neq \perp$ then

$$\langle x!y, s\rangle \xrightarrow{v!w} \langle \mathsf{nil}, s\rangle$$

A deadlock occurs when attempting to send an undefined value or to send a value to an undefined channel (i.e., when $s(x) = \perp$ or $s(y) = \perp$). The communication itself is recorded in the label of the transition.

Input: If $s(x) = v \neq \perp$ and $y \in Var_t$ then for *any* $w \in Val_t$,

$$\langle x?y, s\rangle \xrightarrow{v?w} \langle \mathsf{nil}, s[w/y]\rangle$$

As for the output, a deadlock occurs when attempting to receive a value from an undefined channel. Again, the communication itself is recorded in the label of the transition.

Query: If $\mathcal{E}(q)(s) = \langle v, q'\rangle$ then

$$\langle q \to S, s\rangle \xrightarrow{q'(v)} \langle S, s\rangle \quad \text{and} \quad \langle q \to S, s\rangle \xrightarrow{\neg q'(v)} \langle \mathsf{nil}, s\rangle$$

The execution of a statement guarded by a query starts by evaluating the query with respect to the local state of the object. If it returns a channel name v and an external query q', then q' is evaluated on the internal state of the channel v, otherwise a deadlock occurs. When no deadlock occurs, depending on whether q' evaluates to false or true, the execution of the guarded statement respectively terminates, or continues with the statement S. Locally, an assumption is made about this external evaluation, reflected in the two transitions above: the leftmost assumes that the query q' holds in the state of the channel v, whereas the one on the right assumes the contrary.

The local transition rules for sequential composition, conditional selection, alternative choice and iteration are standard and therefore we omit them here.

3.2 Semantics of a $\sigma\pi$ program

We conclude this section with the definition of the operational semantics of a program in $\sigma\pi$. The operational semantics of a program is defined in terms of a global transition system.

First, we introduce the notion of the state of the existing channels of a network of objects. Such a state σ specifies the existing channels, that is, the channels that have already been created, and the contents of their buffers. Formally, $\sigma(chn) \subseteq (Val_{\mathsf{chn}} \times CM)$ is a finite set of channel identities, each with an indication of its mode, i.e., either asynchronous or synchronous. Moreover, for every existing channel $\langle v, m\rangle \in \sigma(chn)$, a state σ specifies the contents of its buffer, that is, $\sigma(v) \subseteq Val^*$ (Val^* denotes the set of finite sequences of elements in Val). Clearly, if v is a synchronous channel, then its buffer will always be empty, thus we require that if $\langle v, \mathsf{syn}\rangle \in \sigma(chn)$ then $\sigma(v) = \varepsilon$.

For the formal treatment of external queries, we postulate the existence of an evaluation function

$$\mathcal{Q}:(CQuery \times Val_{chn}) \to (Val^* \to Val_{\mathsf{bool}})$$

mapping a closed query q and a channel name v to a boolean value for each content w of the buffer of v: $\mathcal{Q}(q, v)(w) = tt$ means that the query q holds in the buffer w of the channel v, and, similarly, $\mathcal{Q}(q, v)(w) = f\!f$ means that the query q does not hold in the buffer w of the channel v.

The behavior of a network of objects is described in terms of a transition relation between configurations of the form $\langle X, \sigma\rangle$, where σ is the state of the existing channels and X is a finite multiset of pairs of the form $\langle S, s\rangle$, for some local state s and statement S. A pair of the form $\langle S, s\rangle$ denotes an active object within the network: s denotes its current internal state and S the statement that is yet to be executed.

Given a $\sigma\pi$ program, we have the following global transition system.

Internal computation:

$$\frac{\langle S, s\rangle \xrightarrow{\tau} \langle S', s'\rangle}{\langle X \uplus \{\langle S, s\rangle\}, \sigma\rangle \longrightarrow \langle X \uplus \{\langle S', s'\rangle\}, \sigma\rangle}$$

(Here $\uplus$ denotes the union operation on multi-sets.) Internal computation of an object affects only the local state of that object.

Object creation: If $\langle v, m\rangle \notin \sigma(chn)$ for all $m \in CM$, and S'' is the body statement of class c then

$$\frac{\langle S, s\rangle \xrightarrow{\langle v:m,c\rangle} \langle S', s'\rangle}{\langle X \uplus \{\langle S, s\rangle\}, \sigma\rangle \longrightarrow \langle X \uplus \{\langle S', s'\rangle, \langle S'', s''\rangle\}, \sigma'\rangle}$$

where σ' results from σ by adding $\langle v, m\rangle$ to $\sigma(chn)$ and setting $\sigma(v) = \varepsilon$. Moreover, $s''(x) = \perp$, for every variable x, that is, s'' denotes the initial state of the newly created object.

In the semantics of object creation lies the main difference between $\sigma\pi$ and classical object-oriented models. Objects have no identity that can be used by other objects as a reference for communication. Communication happens only through channels that are created together with objects as links connecting each object with its creator object. However, references to channels can be passed from one object to another. A reference to a channel cannot be used as a reference to an object. Note that an object that does not know the reference to any channel, is isolated and as such cannot have any influence on the environment.

External queries: If $\mathcal{Q}(q, v)(\sigma(v)) = tt$ for an existing channel v (that is, $\langle v, m\rangle \in \sigma(chn)$ for some $m \in CM$), then

$$\frac{\langle S, s\rangle \xrightarrow{q(v)} \langle S', s'\rangle}{\langle X \uplus \{\langle S, s\rangle\}, \sigma\rangle \longrightarrow \langle X \uplus \{\langle S', s'\rangle\}, \sigma\rangle}$$

otherwise

$$\frac{\langle S, s\rangle \xrightarrow{\neg q(v)} \langle S', s'\rangle}{\langle X \uplus \{\langle S, s\rangle\}, \sigma\rangle \longrightarrow \langle X \uplus \{\langle S', s'\rangle\}, \sigma\rangle}$$

The state of a channel can be queried by an object. A query on the state of the buffer of a channel can either be true or false, resolving the nondeterminism in the local execution of an externally guarded statement.

Input: If $\sigma(v) = w \cdot u$ for some $u \in Val$ then

$$\frac{\langle S, s\rangle \xrightarrow{v?u} \langle S', s'\rangle}{\langle X \uplus \{\langle S, s\rangle\}, \sigma\rangle \longrightarrow \langle X \uplus \{\langle S', s'\rangle\}, \sigma'\rangle}$$

where $\sigma' = \sigma[w/v]$, that is, σ' and σ are different only because $\sigma'(v) = w$.

An input primitive from an asynchronous channel v is executed only if the channel v can deliver a value of the right type to the requesting object. Otherwise, the execution of the primitive is delayed. If several objects require values from the same channel then only one such request is satisfied by each value in the channel buffer. Note that the non-determinism of an input in the local transition system is resolved here.

Output: If $\langle v, \mathsf{asy}\rangle \in \sigma(chn)$ then

$$\frac{\langle S, s\rangle \xrightarrow{v!w} \langle S', s'\rangle}{\langle X \uplus \{\langle S, s\rangle\}, \sigma\rangle \longrightarrow \langle X \uplus \{\langle S', s'\rangle\}, \sigma'\rangle}$$

where $\sigma' = \sigma[w \cdot \sigma(v)/v]$, that is σ' and σ are different only because $\sigma'(v) = w \cdot \sigma(v)$.

An output command to an asynchronous channel never blocks and its execution causes the introduction of a new value into the buffer of the channel. When several objects output to the same channel, the order in which their proposed values are introduced into the channel buffer is non-deterministic. Generally, an object does not need to be aware of the communication mode of the channel it is using for its output. The case where v is a synchronous channel is treated by the following rule.

Synchronization: If $\langle v, \mathsf{syn}\rangle \in \sigma(chn)$ then

$$\frac{\langle S, s\rangle \xrightarrow{v!w} \langle \bar{S}, \bar{s}\rangle \text{ and } \langle S', s'\rangle \xrightarrow{v?w} \langle \bar{S}', \bar{s}'\rangle}{\langle X \uplus \{\langle S, s\rangle, \langle S', s\rangle\}, \sigma\rangle \longrightarrow \langle X \uplus \{\langle \bar{S}, \bar{s}\rangle, \langle \bar{S}', \bar{s}'\rangle\}, \sigma\rangle}$$

An output command to a synchronous channel is performed only together with an input command on the same channel. In this case, the synchronous input and output commands (on the same channel) cannot be executed independently and they are delayed until such a synchronization is possible.

4 Examples

We now illustrate programming in the $\sigma\pi$ language by a few small examples. To keep the examples readable we write primitive actions without guards, meaning that they are implicitly guarded by an expression that always evaluate to true in every state. Furthermore, we do not write the statement nil at the end of a class body, and omit unnecessary parenthesis.

The first example shows a component consisting of two classes. The more interesting one of these two, when instantiated, returns back on its initial connection the identifier of a new asynchronous channel:

$$\begin{array}{lcl} NIL & \equiv & \mathsf{nil} \\ NEWCHN & \equiv & x := \mathsf{new}(NIL, \mathsf{asy})\ ;\ chn!x\,. \end{array}$$

The interface of the component consists only of the class $NEWCHN$. When an object creates an instance of this class it receives back a reference to a new asynchronous channel that is guaranteed to be empty and not known to any other active object. This reference can later be passed to another object, establishing a new communication link.

Next, we write a $\sigma\pi$ program using the above component for the specification of the Wide Mouthed Frog protocol [11]. There is a server $ROOT$ that is connected to two clients $ProcA$ and $ProcB$ by two different channels. In order to establish a private communication link between the two clients, $ProcA$ creates a new asynchronous channel using the above component, $NEWCHN$, and asks the server to communicate it to the other client. The $\sigma\pi$ description of the classes $ROOT$, $ProcA$ and $ProcB$ is as follows:

$$\begin{array}{lcl} ROOT & \equiv & x := \mathsf{new}(ProcA, \mathsf{syn})\ ; \\ & & y := \mathsf{new}(ProcB, \mathsf{syn})\ ; \\ & & x?z\ ; \\ & & y!z \\ ProcA & \equiv & x := \mathsf{new}(NEWCHN, SYN)\ ; \\ & & x?y\ ; \\ & & chn!y\ ; \\ & & y!msg_1\ ;\ldots y!msg_n \\ ProcB & \equiv & chn?x\ ; \\ & & \mathsf{do}\ x?msg\ ;\ \text{use message}\ msg\ \ \mathsf{od}\,. \end{array}$$

An object of the $ROOT$ class creates the two clients $ProcA$ and $ProcB$, receives a channel reference from the first and sends it to the second. The client $ProcA$ invokes an instance of the component $NEWCHN$ to receive a fresh reference to an asynchronous channel, sends it to the server via its initial communication link chn, and then sends n messages to the other client via this new channel. Finally, the client $ProcB$ receives a reference to a channel from the server and reads the messages coming through this channel.

Procedures can be modeled in the $\sigma\pi$ language as classes: the parameters are passed via a channel and the return values can be passed back to the calling process via another channel. We illustrate this by the following component that calculates the factorial of a number received via its initial connection:

$$\begin{array}{lll} FACT & \equiv & chn?n\ ; \\ & & \mathsf{IF}\ n \leq 1\ \ \mathsf{THEN}\ back := 1 \\ & & \qquad\qquad\ \ \mathsf{ELSE}\ child := \mathsf{new}(FACT, SYN)\ ; \\ & & \qquad\qquad\qquad\quad parameter1 := n - 1\ ; \\ & & \qquad\qquad\qquad\quad child!parameter1\ ; \\ & & \qquad\qquad\qquad\quad child?result\ ; \\ & & \qquad\qquad\qquad\quad back := n * result\ ; \\ & & \mathsf{FI}\ ; \\ & & chn!back\,. \end{array}$$

If an object executes the statement $x := NEW(FACT, SYN)$ and then sends an integer n via the channel x, it will receive back on the same channel the factorial of n.

The next example illustrates how Dijkstra's semaphores can be implemented using a shared asynchronous channel.

$$\begin{array}{lcl} SEMAPHORE & \equiv & sem := \mathsf{new}(NIL, \mathsf{asy})\ ; \\ & & x := \mathsf{new}(USER, \mathsf{syn})\ ;\ x!sem \\ & & y := \mathsf{new}(USER, \mathsf{syn})\ ;\ y!sem\ ; \\ & & token := true\ ;\ sem!token \\ USER & \equiv & chn?sem\ ; \\ & & \ldots \\ & & sem?x\ ; \\ & & \text{critical section}\ ; \\ & & sem!x\ ; \\ & & \ldots\,. \end{array}$$

Finally, we write a $\sigma\pi$ program with three components: a sender, a receiver and a root, which establishes a communication link between the two.

$$\begin{array}{lcl} ROOT(y1, y2) & \equiv & x1 := \mathsf{new}(y1, \mathsf{syn})\ ; \\ & & x2 := \mathsf{new}(y2, \mathsf{syn})\ ; \\ & & x1!x2 \\ SEND & \equiv & y := 0\ ; \\ & & chn?x\ ; \\ & & \mathsf{do}\ x!y\ ;\ y := y + 1\ \mathsf{od} \\ REC & \equiv & \mathsf{do}\ chn?y\ ;\ \cdots\ \mathsf{od} \end{array}$$

The sender receives through its initial connection a reference to a channel through which it sends the values it produces. The receiver uses its initial connection for receiving values. The connection between the sender and the receiver is established by an instance of the $ROOT$ class, which can start its computation in an initial state where the class name $SEND$ is stored in $y1$ and REC in $y2$, respectively. The communication between the sender and the receiver is synchronous, but the two objects are not aware of it. Indeed, asynchronous communication could be established by adapting the body of the class $ROOT$: replace $x1 := \mathsf{new}(y1, \mathsf{syn})$ by $x1 := \mathsf{new}(y1, \mathsf{asy})$. Note that the two components $SEND$ and REC are not affected by this adaptation.

5 Coordinating mobile components

In Internet applications, there is a growing interest for software applications that execute computation while moving around the network. Not surprisingly, the structure of such applications is highly distributed. Since distributed systems are very complex in general, software abstractions are needed to simplify the task [20]. In $\sigma\pi$, components are natural units of distribution that abstract from the difference between local and remote communication. Local interaction takes places amongst objects that are instances of classes within the same component (i.e., likely to be located at the same site), while remote interaction takes place amongst objects of classes of different components.

The language $\sigma\pi$ itself does not impose exogenous coordination, meaning that the coordination primitives affecting each object can be executed within the computation of the object itself. A system described by a program consists of a dynamically evolving collection of processes and channels which run in parallel and communicate with each other by maintaining and passing around channel references. Interaction typically occurs anonymously but under the full control of the objects involved. While this ensures security in the communication (as we argue below), it is sometimes preferable to have a clear separation between computation and coordination to facilitate the reuse of code.

The separation of the computational aspects from the coordination in $\sigma\pi$ can be obtained by specifying for each class in the component *interface* its *external* channels. An external channel x of a component can be used only by its objects as a reference in an input or output communication. Within the context of a component, these external channels play the role of *constants*, that is, their values are not affected by the computation of its objects. The interface of a component is controlled by a class , which is described as an *implementation* of the interface. Plugging an entire computational component, therefore, simply consists of implementing its interface class. Such an implementation itself acts like a *port-manager*. This port-manager has access only to the channels of the interface class and thus, it cannot affect the computation of the component. A port-manager basically communicates its controlled channels to other port-managers. The communication between different port-managers in turn can be prescribed by a coordination protocol in $\sigma\pi$. Such a coordination protocol basically describes a *coordinator* whose only task is to create components and their associated port-managers, and transmit channels received from one port-manager to another, thereby linking and relinking different components. Such a coordination protocol abstracts from the internal computation of the components. Moreover, it does not even know the external channels of is components, because these are controlled by their respective port-managers. The use of such pure coordinators makes it easier to adapt a coordination policy in different applications.

Thus, the language $\sigma\pi$ supports the methodology of component-based software engineering. Furthermore, the feature of $\sigma\pi$ that objects can have full control over their communication, is of particular interest in Internet applications where security policies must be enforced to regulate interprocess communication in view of the possibility of attacks. A reference to a channel shared by two objects may be known to a third party only if one of the two objects divulges it via another channel. In such circumstances, channels may be seen as private domains containing data accessible only to trusted processes that have the right to access them. In this view, channels in $\sigma\pi$ are related to shared-key cryptography.

Analogs to other forms of cryptography can also be easily supported in the $\sigma\pi$ calculus. For example, a sub-language of $\sigma\pi$ can be obtained by imposing a more refined type discipline on its channel variables, so that a channel can be used by a process in either input or output mode only (but not both). In such a sub-language, channels support a security mechanism analogous to public-key cryptography based on key-pairs for encryption and decryption, respectively.

Gateways to impose access domain restriction can be easily implemented in $\sigma\pi$. Such a gateway would be a special object that maintains the initial connection with all other objects it creates, none of which will be allowed to be accessible from the rest of the environment. This gateway object will not allow channel references to be communicated through other channels. This allows all objects within a domain to communicate with each other, but they will not have any direct communication links with any other object outside the domain.

6 Conclusion and related work

In $\sigma\pi$, objects of different components may interact (via channels) without a priori knowledge of the internal class structure of the component to which the other object belongs. Thus, one component can be replaced by another, provided that they both exhibit the same communication behavior. The communication behavior of a component consists of the set of sequences generated by the operational semantics of communications of the form $u!v$ and $u?v$, where $u!v$ indicates that the value of v has been sent along channel u and $u?v$ indicates that the value v has been received from channel u. Since we do not want to distinguish identical programs that may assign different names to a channel because of different naming mechanism, we must abstract from the specific identities of the channels. Therefore, we in fact must consider sets of sequences of communications that are closed under renaming, i.e., injective functions in $Val_{\mathsf{chn}} \to Val_{\mathsf{chn}}$. Currently, we are investigating an appropriate logic for describing and reasoning about such sets of communication sequences. This logic of communication sequences would then form a basis for a programming logic for proving that the set of communication sequences of a component satisfies certain properties. Such a programming logic for components will provide a formal basis for the actual practice of component-based software engineering.

In this paper, we have studied two basic type of object connectors: synchronous links and unbounded FIFO buffers. Other kinds of connectors can also be added to $\sigma\pi$ without major modifications to its semantics. For example, one could create channels with a multiset or a set structure (i.e., a Linda-like tuple space [13] or a Splice-like data-space [9]) when the ordering among data items is insignificant. In this way, several generic software architectures can be described by the $\sigma\pi$ language.

Currently, we are implementing the $\sigma\pi$ language using Java [16]. The basic idea is to implement channels as objects of a class chn in Java. This class contains appropriate data structures for representing the buffer of a channel and method definitions for adding elements to and removing elements from the buffer. The class definitions of $\sigma\pi$ will then correspond to (the subclasses of) a certain kind of class in Java which takes into account the particular way objects interact and are created in $\sigma\pi$. The object creation of $\sigma\pi$ can then be implemented as the object creation of its respective Java class, together with the creation of an object instance of the class chn. Only a reference to this latter object will be returned to the creator object.

The communication mechanism of the $\sigma\pi$ language is inspired by that of the π calculus [22], uses primitives for process interaction similar to those of CSP [17], and it is implemented using a notion of channel similar to that used by the coordination language Manifold [5, 8]. The π calculus is a process calculus for communicating and mobile systems which abstract from the notion of state by considering atoms as symbols, whereas the atomic constituents of the $\sigma\pi$ language are interpreted as state transformers. The communication mechanism of the language $\sigma\pi$ in fact originates from the introduction of a notion of state in the π-calculus. Manifold is a coordination language that supports dynamic (re)connection of input and output ports of components, depending on the current state of an application. As in the Manifold language, coordination in $\sigma\pi$ consists of dynamic creation of objects and connections among them, and dynamic changes of these connection in reaction to the data received.

The π-calculus has been used as a semantic foundation for object-oriented languages [28], for cryptographic calculi like the Spi calculus [1] and the Ambient calculus [12], and for several languages for composing components. Examples include Piccola [2], a language for composing components that communicate by passing extensible records, and Darwin [19], a configuration language that models composition of distributed agents in terms of data-flows. A CCS-like calculus for the Olan configuration language [15] is also based on the same concept, supporting the specification of applications as hierarchies of components glued together by connectors. These languages, however, differ from $\sigma\pi$ in that, contrary to the π-calculus, they do not involve the notion of a state.

The notion of component in $\sigma\pi$ is similar to that of Manifold's, and makes it possible for to acquire and replace the implementation of a component at run-time, possibly from some independent vendor over the net [6]. This notion supports the same kind of pluggability as obtained in the Java language [16] by separating interfaces from classes.

The $\sigma\pi$ language is very similar to the verification modeling language Promela, used in SPIN, a tool for analyzing the logical consistency of distributed systems, specifically of data communication protocols [18]. The main difference is that when processes are instantiated in Promela, they receive a unique identity like in ordinary object-oriented languages. Furthermore, in Promela channels and variables can be declared globally.

From a more theoretical point of view, the semantics of the $\sigma\pi$ language is a direct extension of the one presented in [7] for dynamic networks of asynchronously communicating objects. The use of a two-level transition system to define the formal semantics of $\sigma\pi$ programs stems from [8].

The $\sigma\pi$ language can also be compared with a number of models for Internet applications, most of which use decoupled and associative coordination typical of the Linda coordination language [13]. For example, TuCSoN [25] is a model for programming Internet agents based on the notion of programmable local tuple spaces; Secure Object Space [10] is a model for mobile agents based on multiple tuple spaces supporting privacy of data, security and logical channels; and Klaim [14] supports mobility of code via the notion of physical location and the use of multiple tuple spaces with types for representing access rights.

Acknowledgments: We want to thank the anonymous referees and all the members of the Amsterdam Coordination Group for their comments on a preliminary version of this paper.

References

[1] M. Abadi and A.G. Gordon. A calculus for cryptographic protocols: the Spi calculus. In *Proceedings of the 4th. ACM Conference on Computer and Communication Security*, 1997, pp. 36–47.

[2] F. Achermann, M. Lumpe, J.-G. Schneider, and O. Nierstrasz. Piccola - a small composition language. Available on line at the URL: `http://www.iam.unibe.ch/~scg/Research/Piccola`.

[3] G. Agha. *Actors: A model of concurrent computation in distributed systems.* MIT Press, 1986.

[4] P. America, J. W. de Bakker, J.N. Kok, and J.J.M.M. Rutten. Operational semantics of a parallel object-oriented language. In *Proc. 13th ACM Symposium on Principles of Programming Languages*, pages 194–208, 1986.

[5] F. Arbab. Manifold version 2: Language reference manual. Technical report, CWI, Amsterdam, The Netherlands, 1996. Available on-line at the URL: `http://www.cwi.nl/ftp/manifold/refman.ps.Z`.

[6] F. Arbab, C.L. Blom, F.J. Burger, and C.T.H. Everaars. Reusable coordinator modules for massively concurrent applications. *Software: Practice and Experience* 28:7, 1998, pp. 703-735.

[7] F. de Boer. Reasoning about asynchronous communication in dynamically evolving object structures. In D.Sangiorgi and R. de Simone (eds.) *Proceedings of CONCUR'98*, volume 1466 of *Lecture Notes in Computer Science*, Springer-Verlag, 1998.

[8] M.M. Bonsangue, F. Arbab, J.W. de Bakker, J.J.M.M. Rutten, A. Scutella', and G. Zavattaro. A transition system semantics for the control-driven coordination language MANIFOLD. In *Theoretical Computer Science* 240:1, 2000.

[9] M.M. Bonsangue, J.N. Kok, M. Boasson, and E. de Jong. A software architecture for distributed control systems and its transition system semantics. In J. Carroll, G.B. Lamont, D. Oppenheim, K.M. George, and B. Bryant, editors, *Proceedings of the 1998 ACM Symposium on Applied Computing (SAC '98)*, ACM press, 1998, pp. 159 – 168.

[10] C. Bryce, M. Oriol, and J. Vitek. Secure object spaces: A coordination model for agents. In P. Ciancarini and A. Wolf (eds.), *Proceedings of COORDINATION'99*, volume 1594 of *Lecture Notes in Computer Science*, Springer-Verlag, 1999, pp. 4–20.

[11] M. Burrows, M. Abadi, and R.M. Needham. A logic of authentication. *ACM Transaction on Computer Systems* 8:1, 1990, pp. 18–36.

[12] L. Cardelli and A.D. Gordon. Mobile ambients In M. Nivat (ed.), *Proceedings of Foundation of Software Science and Computational Structure*, volume 1378 of *Lecture Notes in Computer Science*, Springer-Verlag, 1998, pp. 140–155.

[13] N. Carriero and D. Gelernter. Linda in context. In *Communications of the ACM* 32:4, pages 444–458, 1989.

[14] R. De Nicola, G. Ferrari, and R. Pugliese. Coordinating mobile agents via blackboards and access rights. In D. Garlan and D. Le Métayer (eds.), *Proceedings of COORDINATION'97*, volume 1282 of *Lecture Notes in Computer Science*, Springer-Verlag, 1997, pp. 220–236.

[15] J.-Y. V. Dury, L. Bellissard, and V. Marangozov. A component calculus for modelling the Olan configuration language. In D. Garlan and D. Le Métayer (eds.), *Proceedings of COORDINATION'97*, volume 1282 of *Lecture Notes in Computer Science*, Springer-Verlag, 1997, pp. 392–409.

[16] J. Gosplin and H. McGilton. The Java language environment. Sun Microsystems Computer Company, 1995.

[17] C.A.R. Hoare. Communicating sequential processes. *Communication of ACM* 21, 1978, pp. 666–677.

[18] G.J. Holzmann. The model checker SPIN *IEEE Transactions on Software Engineering* 23:5, 1997.

[19] J. Magee, J. Kramer, and N. Dulay. Darwin/mp: An environment for parallel and distributed programming. In *Proceedings 26th Annual Hawwaii Int. Conf. on System Sciences*, volume 2, IEEE computer Society Press, 1993.

[20] T.D. Meijler and O. Nierstrasz. Beyond objects: components. In M.P. Papazoglou and G. Schlageter (eds.), *Cooperative Information Systems: Current Trends and Directions*, Academic Press, 1997.

[21] Microsoft Corporation. Distributed component object model protocol-DCOM/1.0, 1996 Available on line at the URL: `http://www.microsoft.com/oledev`.

[22] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, parts I and II. *Information and Computation* 100:1, 1992, pp. 1–77.

[23] O. Nierstrasz and L. Dami. Component-oriented software technology. In O. Nierstrasz and D. Tsichritzis (eds.), *Object-Oriented Software Composition*, Prentice Hall, 1995, pp. 3–38.

[24] OMG. CORBA 2.1 specifications, 1997. Available on line at the URL:`http://www.omg.org`.

[25] A. Omicini and F. Zambonelli. Tuple centre for the coordination of Internet agents. In J. Carroll, H. Haddad, D. Oppenheim, B. Bryant, and G.B. Lamont (eds.), *Proceedings of the 1999 ACM Symposium on Applied Computing (SAC '99)*, ACM press, 1999, pp. 183–190.

[26] G. Papadopoulos and F. Arbab. Coordination models and languages. *Advances in Computers*, volume 46: The Engineering of Large Systems, Academic Press, 1998.

[27] B. Stroustrup. *The C++ programming language* Addison-Wesley, 1991.

[28] D. Walker. π-calculus semantics of object-oriented programming languages. *Information and Computation* 116:2, 1995, pp. 253–271.