Acta Informatica manuscript No.
(will be inserted by the editor)

MoMi: A Calculus for Mobile Mixins*

Lorenzo Bettini[1], **Viviana Bono**[2], **Betti Venneri**[1]

[1] Dipartimento di Sistemi e Informatica, Università di Firenze, e-mail: {bettini,venneri}@dsi.unifi.it
[2] Dipartimento di Informatica, Università di Torino, e-mail: bono@di.unito.it

Abstract. MOMI (Mobile Mixins) is a coordination language for mobile processes that communicate and exchange object-oriented code in a distributed context. MOMI's key idea is structuring mobile object-oriented code by using mixin-based inheritance. Mobile code is compiled and typed locally, and can interact successfully with code present on foreign sites only if its type is subtyping-compliant with the type of what is expected by the receiving site. The key feature of the paper is the definition of this subtyping relation on classes and mixins that enables a significantly flexible, yet still simple, communication pattern. We show that communication by subtyping is type-safe in that exchanged code is merged into local code without requiring further type analysis and recompilation.

1 Introduction

The Internet provides technologies that allow the transmission of resources and services among computers distributed geographically in wide area networks. The growing use of a network as a primary environment for developing, distributing, and running programs requires new supporting infrastructures. A possible answer to these requirements is the use of *mobile code* [49, 23] and in particular of *mobile agents* [38, 35, 52], which are software entities consisting of data and code that can migrate autonomously to a remote computer and execute automatically on arrival.

In parallel, the object-oriented paradigm has become established as a well suited technology for designing and implementing large software systems. In particular, it provides a high degree of modularity, flexibility and reusability, so that it is widely used also in distributed contexts (see, e.g., the Java Remote Method Invocation [48], and CORBA [42]).

The new scenario arising from mobility tests the *flexibility* of object-oriented code in a wider framework. Object-oriented components are often developed by different providers and may be downloaded on demand and assembled dynamically with local applications. As a consequence, they must be strongly adaptive to any local execution environment, so that they can be reconfigured dynamically in several ways: downloaded code can be specialized by locally defined operations,

* This work has been partially supported by EU within the FET - Global Computing initiative, project MIKADO IST-2001-32222 and DART project IST-2001-33477 and by MIUR project EOS. The funding bodies are not responsible for any use that might be made of the results presented here.

and, conversely, locally developed software can be extended with new operations available from the downloaded code.

Hence, a coordination language for programming object-oriented components as mobile code should provide specific mechanisms for coordinating not only the transmission, but also the local dynamic reconfiguration of the object-oriented code.

In this paper we address the above issue in the specific context of class-based languages, that "form the main stream of object-oriented programming" [1]. We use a *mixin*-based approach for structuring mobile object-oriented code, as an alternative to standard inheritance mechanisms, as discussed in Section 2. A *mixin* (a class definition parameterized over the superclass) can be viewed as a function that takes a class as a parameter and derives a new subclass from it. The same mixin can be applied to many classes (this operation is known as *mixin application*), obtaining a family of subclasses with the same set of methods added and/or redefined. A subclass can be implemented before its superclass has been implemented; thus mixins remove most of the dependencies of the subclass on the superclass, enabling dynamic development of class hierarchies. Mixins have become a focus of active research both in the software engineering [51,47,27] and programming language design [19,50,29,3] communities. In this paper the term *mixin* refers to *mixin classes*, as opposed to *mixin modules* (modules supporting deferred components [4,37]).

In our approach, we want to use mixins and mixin application as the mechanism for assembling mobile components in a flexible and safe way. Namely, classes and mixins (as well as processes) can be exchanged among different nodes, and then *mixed* into the local class hierarchies. To this aim, we propose a kernel language, MOMI (Mobile Mixins), integrating two distinct components: a core coordination calculus, which addresses the code exchange, and an object-oriented mixin-based calculus, which structures and composes mobile code. The focus of our proposal is on the aspects of communication and composition, therefore: *(i)* the coordination calculus is a simple low-level communication protocol that relies on a synchronous send/receive mechanism; *(ii)* the mixin-based calculus reduces to a minimal prototypical syntax, which does not describe a full language, but only the object-oriented features needed to realize mixin-based code composition. Most importantly, MOMI relies on typing, and in particular on a novel *subtype* relation on class and mixin types, which is the main novelty of the MOMI approach, and that is used as a flexible code communication pattern. The main distinguishing features of our proposal can be then summarized as follows:

- the static typing system is used for checking locally processes in their own locality, independently from other sites;
- classes and mixins travel together with their static type: during static type checking, object-oriented code that will be sent to remote sites is decorated with its type;
- when the code is received on a site (whose code has been successfully compiled, too), it is accepted only if its type is a subtype of the expected type;
- finally, if the retrieved code is successfully accepted then it can be composed within the local code, dynamically and automatically, in a type-safe way without requiring to type check the whole code again.

This last point is a crucial matter for mobility, since mobile code is expected to be autonomous: once the communication occurred successfully, transmitted code must behave remotely in a type safe way (no run-time errors due to type violations should occur). This makes the code exchange an *atomic* action. Moreover, there is general evidence that a flexible communication mechanism is of paramount importance in a distributed mobile code setting: on the one hand, it is important to be "open" enough to accept utilities and services that are not exactly as we would have dreamed

of (after all if we wanted something perfectly customized we would have written it ourselves, not searched for it on the net); on the other hand, it is also important to be "wise", that is, being able to single out possibly evil utilities or services from the ones that are harmless. The proposed MOMI subtype relation on class and mixin types aims at guaranteeing both the above issues, flexibility and type safety.

A mixin type describes a set of features that must be satisfied by any of its superclasses. Therefore, during mixin application, the type of the superclass must be checked against the mixin's requirements for type consistency. As a matter of fact, it is quite natural to perform this matching via a (standard) subtype check on record types (i.e., by relying on inclusion polymorphism, [22]), so that any "subtype" of the type of the superclass expected by the mixin satisfies the mixin requirements. Notice though that in our mobile scenario the situation is more complicated than in a sequential setting: mixins and classes are treated as fully-fledged polymorphic entities that are exchanged among processes. Hence, we need to define a suitable subtype relation both on classes and on mixins to be used as the open-wise communication pattern discussed above. While the former is still the standard one based on record types, the latter is a new concept with respect to similar approaches to mixins in sequential settings (see, for instance, [29, 18, 3, 2]), even though it still relies on inclusion polymorphism. It requires a special attention due to different roles played by the internal components of a mixin definition: new methods defined by the mixin (behaving co-variantly), expected methods from the superclass (behaving contra-variantly), and redefining ones (behaving invariantly). It is then important to notice that such novel class-mixin subtyping is never used in the local type inference but only at the communication level. The subtype relation on class and mixin types is formally defined in Section 4.3.

Since mobile classes and mixins are mixed together in various ways in local applications, the issue of type safety becomes the crucial point in which the coordination calculus and the object-oriented calculus interleave. Indeed, we must guarantee that processes remain well-typed after replacing formal parameters with mobile classes and mixins, where these received actual parameters have subtypes of the types of the parameters being substituted for. This substitution operation requires specific method renaming, in order to avoid name collision problems arising when classes and mixins are used as first-class data (as discussed in Session 5.1). Therefore, type safety of the communication results from the static type soundness of local and foreign code, and from a (global) subject reduction theorem relying on a substitution property based on this substitution operation (Section 5).

The issue of mobility also motivates our choice of adopting a *structural subtyping* (i.e., subtyping is defined directly on the structures of types) instead of a *nominal subtyping* (where names assigned to types are significant and subtyping is explicitly declared, like in Java and C++). Quoting from [44]: "In a structural setting, a type expression is a closed entity: it carries with it all the information that is needed to understand its meaning. In a nominal system, we are always working with respect to some global collection of type names and associated definitions". In our mobile context, these global agreements on type names are not realistic. In the distributed setting we need the full flexibility that is provided by the structural subtyping. This flexibility comes at some cost from the technical point of view, in order to maintain type safety: we must enforce some restrictions on method names occurring in method types of classes and mixins. This matter is discussed in details in Section 5.1. Another general reason for using structural subtyping in the presence of mixins comes from Bracha, who affirms in [5] that they work well together. Nevertheless, since most production languages use nominal type systems, a nominal type system for MOMI is the subject of ongoing studies.

The MOMI's approach has been firstly presented in [12], where a preliminary version of the calculus is sketched. The present paper is an extended version of [12]: the syntax of the calculus is revised and, foremost, the type system is fully formalized and the metatheory is investigated.

The paper outline is as follows. In Section 2 we motivate our approach by presenting some scenarios of object-oriented mobile code. We introduce the syntax of MOMI in Section 3, and the type system and the novel subtype relation on classes and mixins in Section 4. In Section 5 we prove the main properties of the typing and in particular the substitution property. Section 7 concerns the operational semantics and the subject reduction theorem on net evolution. In Section 8 we show a MOMI implementation of a motivating scenario. Section 9 describes briefly the application of the MOMI approach to the language KLAIM, and its implementation (freely available at `http://music.dsi.unifi.it`). Section 10 concludes the paper addressing some related works.

2 Mobility and Object-Oriented Code

In this section we discuss two different scenarios, where an object-oriented application is received from (sent to) a remote site. In this setting, we can assume that the application consists of a piece of code *A* that moves to a remote site, where it will be composed with a local piece of code *B*. These scenarios may occur in the development of an object-oriented software system in a distributed context with mobility.

Scenario 1 The local programmer may need to download dynamically classes in order to complete his own class hierarchy, without triggering off a chain reaction of changes over the whole system. For instance, he may want the downloaded class *A* to be a child class of a local class *B*. This generally happens in *frameworks* [32]: classes of the framework provide the general architecture of an application (playing the role of the local software), and classes that use the framework must specialize them in order to provide specific implementations. The downloaded class may want to use operations that depend on the specific site (e.g., system calls); thus the local base class must provide generic operations and the mobile code becomes a derived class containing methods that can exploit these generic operations.

Scenario 2 The site that downloads the class *A* for local execution may want to redefine some, possibly critical, operations that remote code may execute. This way, the access to some sensitive local resources is not granted to untrusted code (for example, some destructive "read" operations should be redefined as non-destructive ones in order to avoid that non-trusted code erases information). Thus the downloaded class *A* is seen, in this scenario, as a base class, that is locally specialized in a derived class *B*.

We note that in **1** the base class is the local code, while in **2** the base class is the mobile code. These scenarios are typical object-oriented compositions seen in a distributed mobile context. A major requirement is that composing local code with remote code should not affect existing code in a massive way. Namely, both components and client classes should not be modified nor recompiled.

Standard mechanisms of class extension and code specialization would solve these design problems in a static and local context, but they do not scale well to a distributed context with mobile code. The standard inheritance operation is essentially static in that it fixes the inheritance hierarchy, i.e., it binds derived classes to their parent classes, once and for all at compile time. If such a hierarchy must be changed, the program must be modified and then recompiled. This is quite unacceptable in a distributed mobile scenario, since it would be against its underlying

dynamic nature. Indeed, what we are looking for is a mechanism for providing a dynamic reconfiguration of the inheritance relation between classes, not only a dynamic implementation of some operations.

Let us go back and look at the above scenarios in more detail. We could think of implementing some kind of "dynamic inheritance" for specifying at run-time the inheritance relation between classes without modifying their code. Such a technique could solve the difficulty raised by scenario **1**. However, "dynamic inheritance" is not useful for solving scenario **2**, that would require a not-so-clear dynamic definition of the base class. Another solution would be releasing the requirement of not affecting the existing code, and permitting the modification the code of the local class (i.e., the local hierarchy). This could solve the second scenario, but not the first one that would require access to foreign source code. We are also convinced that the two scenarios should be dealt with by the same mechanism, allowing to use dynamically the same code in different environments, either as a base class for deriving new classes, or as derived class for being "adopted" by a parent class. We remark that a solution based on *delegation* could help solve these problems. However, delegation would destroy at least the dynamic binding and the reusability of the whole system [17]. Also some design patterns [32] can turn out to be useful in implementing these scenarios, but these patterns still require additional manual programming, while, by using the linguistic constructs we propose in this paper, one can easily implement these systems directly and can benefit further static type checks.

Summarizing, mobile object-oriented code needs to be more flexible than locally developed object-oriented applications. To this aim, we propose a novel solution which is based on a mixin approach with subtyping and we show that it enables us to achieve the desired dynamic flexibility. Indeed, mixin-based inheritance is more oriented to the concept of "completion" than to that of extendibility/specialization. Mixins are incomplete class specifications parameterized over superclasses, thus the inheritance relation between a derived and a base class is not established through a declaration (e.g., like `extends` in Java), instead it can be coordinated by the operation of *mixin application*, that takes place during the execution of a program.

3 MoMi: Mobile Mixin Calculus

The calculus MoMi is divided into three layers:

- the topmost layer is a *network*, seen as a composition of nodes;
- the intermediate layer is a set of processes that are the computational units and that run on nodes. Processes can also migrate to remote nodes and continue their execution there (e.g., they can implement many flavors of code mobility [23]);
- the lower lever is the object-oriented code. This is instrumental to processes that exchange it among the nodes of the network and exploit it to perform local operations.

This layer separation, apart from being standard in distributed and mobile calculi, in our context is even more crucial: MoMi is intended to be a small and general framework to experiment the integration of object-oriented features in calculi for mobility and distribution (for instance it could have a direct application to calculi such as, e.g., KLAIM, [24] and *DJoin*, [30]).

With this respect, the three layers can be adapted to the specific requirements of the language we intend to integrate with mobile and object-oriented features. This is the main reason why we keep the different layers of MoMi simple (e.g., we favor the simpler synchronous communication to the asynchronous one) and abstract as much as possible (e.g., we do not specify the syntax for

method bodies). All these simplifications imply less impositions when we integrate the MOMI features inside a concrete language, while still allowing us to inherit all the formal properties of MOMI that guarantee type safety. As an instance of MOMI, we extended KLAIM with object-oriented features (see O'KLAIM Section 9); in O'KLAIM we integrate the asynchronous communication mechanism of KLAIM with some mixin-based constructs using the MOMI approach fully.

We first describe the kernel syntax of the object-oriented code exchanged among distributed mobile processes. We then present the coordination part, which includes representative features for distribution, communication and mobility of processes and code.

3.1 Surface Object-Oriented Language

The object-oriented part of MOMI is defined as a standard class-based object-oriented language supporting mixin-based class hierarchies via *mixin definition* and *mixin application*. It was initially inspired by the core calculus of [18], and this is especially recognizable in MOMI's early version [12]. However, specific incarnations of most object-oriented notions (such as, e.g., functional or imperative nature of method bodies, object references, cloning, etc.) are irrelevant in this context, where the emphasis is on the structure of the object-oriented mobile code. Hence, we work here with a basic syntax forming a kernel calculus called here SOOL (*Surface Object-Oriented Language*, shown in Table 1) that includes the essential features a language or a calculus must support to be the MOMI's object-oriented component. It is of paramount importance to keep in mind that SOOL is *not* a specific object-oriented language or calculus, but just the set of mixin-based features we need in order to design the MOMI approach. Therefore, also SOOL typing and reduction rules (presented in following sections) are only a part of the rules defining a concrete language "implementing" SOOL. Moreover, we will prove properties assuming that we adopt as SOOL concrete instances only those (sequential) object-oriented calculi/languages underlying SOOL features (such as mixins) *and* enjoying a safety property with respect to message-not-understood run-time errors. One of such SOOL instances can be the calculus of [18].

exp	::=	v	(value)
	|	$\mathsf{new}\ exp$	(object instantiation)
	|	$exp \Leftarrow m$	(method call)
	|	$v \diamond exp$	(mixin application)
v	::=	$\{m_i : \tau_{m_i} = f_i\ ^{i \in I}\}$	(record)
	|	x	(variable)
	|	$\mathsf{class}\ [m_i : \tau_{m_i} = f_i\ ^{i \in I}]\ \mathsf{end}$	(class def)
	|	mixin	(mixin def)
		$\mathsf{expect}[m_i : \tau_{m_i}\ ^{i \in I}]$	
		$\mathsf{redef}[m_k : \tau_{m_k}\ \mathsf{with}\ f_k\ ^{k \in K}]$	
		$\mathsf{def}[m_j : \tau_{m_j} = f_j\ ^{j \in J}]$	
		end	

Table 1: Syntax of SOOL.

SOOL expressions offer object instantiation, method call and *mixin application* (to built class hierarchies); $\diamond$ denotes the mixin application operator and it always associates to the right. A SOOL value, to which an expression reduces, is an object, which is essentially a record[1] $\{m_i :$

[1] More precisely, we assume that an object is a recursive record where the self-references that are present in method bodies are solved by, for example, applying an appropriate fix-point operator, that binds *self* to the host object, as it happens, for instance, in [18].

$\tau_{m_i} = f_i \ ^{i\in I}\}$, a class definition class $[m_i : \tau_{m_i} = f_i \ ^{i\in I}]$ end, or a mixin definition mixin expect$[m_i : \tau_{m_i} \ ^{i\in I}]$ redef$[m_k : \tau_{m_k}$ with $f_k \ ^{k\in K}]$ def$[m_j : \tau_{m_j} = f_j \ ^{j\in J}]$ end. In class and mixin definitions, $[m_i : \tau_{m_i} = f_i \ ^{i\in I}]$ is a sequence of method definitions (also sometimes abbreviated as ρ) and I, J and K are sets of indexes. Method bodies, denoted here with f (possibly with subscripts), are closed terms/programs and they might have formal parameters (we abstract away from their actual form, relying on the form they may have in any concrete object-oriented languages underlying SOOL). Furthermore, m_i are method names and τ are types (types are described formally in Section 4).

A mixin is essentially an abstract class that is parameterized over a (super)class. Let us describe informally the mixin use through a tutorial example:

```
M = mixin
  expect [n : τ]
  redef [m2 : τ2 with ... next() ...]
  def [m1 : τ1 = ... n() ...]
end

C = class
  [n : τ = ...
  m2 : τ2 = ...]
end

(new (M ⋄ C)) ⇐ m1()
```

Each mixin consists of three parts: (*i*) methods defined in the mixins, like m_1; (*ii*) *expected methods*, like *n*, that must be provided by the superclass; (*iii*) *redefined methods*, like m_2, where *next* can be used to access the implementation of m_2 in the superclass[2]. A mixin can be applied to any class that fulfills all the mixin's requirements, thus, for instance, the application $M \diamond C$ is correct and constructs a class, which is a subclass of C.

Following the standard convention, class and mixin definitions are closed terms in that all free variables occurring in a method's body are bound to the method's formal parameters. We do not consider explicit data fields, as they can be modeled in various ways.

We made the choice of making explicit: (*i*) names and types of the expected methods; and (*ii*) types of the defined and redefined methods. There are calculi/languages for which all this information is said to be inferred, for example in [18], so such names and types do not have to be explicitly given by the programmer. This would be acceptable from the point of view of MOMI, but we prefer to make for SOOL the simplest yet most general of the choices, since the focus is on using this type information for code mobility, rather than on deducing it.

3.2 Coordination language

MOMI's coordination component is similar to CCS [41] but also widely inspired by KLAIM [24], where physical nodes are denoted explicitly as localities. MOMI is higher-order in that processes can be exchanged as first-class data.

A node is denoted by its locality, ℓ, and by the processes P running on it. Informally, send(A, ℓ) sends A, that can be either a process P or code represented as an object-oriented value v, to a locality ℓ, where there may be a process waiting for it by means of a receive. The argument of receive, id, ranges over x (a variable of SOOL) and X (a process variable): for instance, a process can receive a class and integrate it into its local class hierarchy, or a process and spawn it for local execution. Our calculus is *synchronous* as it is meant to be a simple low-level communication protocol, but designing an asynchronous version would be straightforward.

In Table 2 *exp* and *v* respectively denote an object-oriented expression and an object-oriented value, according to Table 1.

[2] We use the keyword "next" instead of the more common keyword "super" not to cause conflict with the term "super" commonly referred to class-based inheritance. Furthermore, we choose to refer to the corresponding super-method only, instead of permitting the access to all of the superclass' methods (as **super** in Java does), because such extension is just

P	::=	**nil**	(null process)
	|	$a.P$	(action prefixing)
	|	$P_1 \mid P_2$	(parallel comp.)
	|	X	(process variable)
	|	$\mathsf{def}\ x = exp\ \mathsf{in}\ P$	(def)
a	::=	$\mathsf{send}(A, \ell)$	(send)
	|	$\mathsf{receive}(id : \tau)$	(receive)
A	::=	$v \mid P$	(send's arg.)
id	::=	$x \mid X$	(receive's arg.)
N	::=	$\ell :: P$	(node)
	|	$N_1 \parallel N_2$	(net composition)

Table 2: MOMI syntax.

We introduce the construct $\mathsf{def}\ x = exp\ \mathsf{in}\ P$ in order to pass to the sub-process P the result of the evaluation of an *exp*. In the processes $\mathsf{receive}(id : \tau).P$ and $\mathsf{def}\ x = exp\ \mathsf{in}\ P$, receive and def act as binders for, respectively, *id* and x in the process P. The receive action specifies, together with the formal parameter name, the type of the expected actual parameter. Notice that this is the only explicitly needed typed expression in the coordination language. In Section 4 a type system is introduced in order to assign a type to each well-behaved process. By convention, we use the form $\mathsf{def}\ x = exp_1\ \mathsf{in}\ exp_2$ as a shorthand for $\mathsf{def}\ x = exp_1\ \mathsf{in}\ \mathsf{def}\ y = exp_2\ \mathsf{in}$ **nil**.

3.3 Mixin Mobility in Action

We present two simple examples showing mobility of mixins in action. The examples represent a *remote evaluation* and a *code-on-demand* situation, respectively [23]. Let us observe that both situations can be seen as examples of mobile agents as well. A more complex example is presented in Section 8.

Example 1. Let `agent` represent the type of a mixin defining a mobile agent that must print some data by using the local printer on any remote site where it is shipped for execution. Obviously, since the *print* operation depends highly on the execution site (even only because of the printer drivers), it is sensible to leave such a method unimplemented. The mixin then can be applied, on the remote site, to a local class *printer* which will provide the specific implementation of the *print* method in the following way:

$$\begin{array}{l} \ell_1 :: \ldots \mid \mathsf{send}(my\_agent, \ell_2) \parallel \\ \ell_2 :: \ldots \mid \mathsf{receive}(mob\_agent : \texttt{agent}). \\ \qquad \mathsf{def}\ PrinterAgent = mob\_agent \diamond printer\ \mathsf{in} \\ \qquad\quad (\mathsf{new}\ PrinterAgent) \Leftarrow start() \end{array}$$

Example 2. Let `agent` be a class type defining a mobile agent that may access the file system of a remote site. If the remote site wants to execute this agent while restricting the access to its own file system, it can define locally a mixin *restricted*, redefining the methods accessing the file system according to specific restrictions. Then the arriving agent can be composed with the local mixin in the following way.

a technical matter and including it would complicate the operational semantics without adding anything relevant to the calculus.

$$\begin{array}{l}\ell_1 ::= \ldots \mid \mathsf{send}(\mathit{my\_agent}, \ell_2) \parallel \\ \ell_2 ::= \ldots \mid \mathsf{receive}(\mathit{mob\_agent} : \mathtt{agent}). \\ \qquad\qquad \mathsf{def}\ \mathit{RestrictedAgent} = \mathit{restricted} \diamond \mathit{mob\_agent}\ \mathsf{in} \\ \qquad\qquad\quad (\mathsf{new}\ \mathit{RestrictedAgent}) \Leftarrow \mathit{start}()\end{array}$$

This example can also be seen as an implementation of a "sandbox".

The above examples highlight how an object-oriented expression (*mob_agent*) can be used by the receiver site both as a mixin (Example 1) and as a base class (Example 2). Indeed, without any change to the code of the examples, one could also construct dynamically a class such as `restricted` $\diamond$ `mob_agent` $\diamond$ `printer`. It is important to remark that in these examples we assume that the sent code (argument of send) and the expected code (argument of receive) are "compatible". This will be guaranteed by the dynamic matching between the actual parameter type and the formal parameter type in the communication rule (see Table 10).

4 Typing

We present the type system for MOMI starting from SOOL, then we introduce the rules for typing processes. We conclude this section by introducing a novel subtype relation, $\sqsubseteq$, over class and mixin types, which is motivated by our goal of using classes and mixins in a mobile context, exchanged as first-class entities.

4.1 Typing SOOL *expressions*

The set $\mathcal{T}$ of types is defined in Table 3. Σ (possibly with a subscript) denotes a record type of the form $\{m_i : \tau_{m_i}{}^{i\in I}\}$. A record type can be viewed as a set of pairs *label*:*type*, with the additional requirement that labels are pairwise disjoint. Thus notations and operations on sets are easily extended to record types as in the following definitions:

- if $m_i : \tau_{m_i} \in \Sigma$ we say that the *subject* m_i *occurs* in Σ (with type τ_{m_i}). $\mathit{Subj}(\Sigma)$ is the set of the *subjects* of Σ and $\mathit{Meth}(\Sigma)$ is the set of all the method names occurring in Σ (e.g., if $\Sigma = \{m : \{n : \tau\}\}$ then $\mathit{Subj}(\Sigma) = \{m\}$, while $\mathit{Meth}(\Sigma) = \{m, n\}$).
- $\Sigma_1 \subseteq \Sigma_2$ is the standard set inclusion, i.e., $m : \tau_m \in \Sigma_1 \Rightarrow m : \tau_m \in \Sigma_2$;
- Σ_1 and Σ_2 are considered *equals*, denoted by $\Sigma_1 = \Sigma_2$, if $\Sigma_1 \subseteq \Sigma_2$ and $\Sigma_2 \subseteq \Sigma_1$ (i.e., they differ only for the order of their elements);
- $\Sigma_1 \cup \Sigma_2$ is the standard set union (defined only on Σ_1 and Σ_2 such that $\mathit{Subj}(\Sigma_1) \cap \mathit{Subj}(\Sigma_2) = \emptyset$, in order to guarantee $\Sigma_1 \cup \Sigma_2$ be a record type);
- $\Sigma_1 - \Sigma_2$ is the standard set difference;
- $\Sigma_1 / \Sigma_2 = \{m_i : \tau_{m_i} \mid m_i : \tau_{m_i} \in \Sigma_1 \ \wedge\ m_i \text{ occurs in } \Sigma_2\}$.

$$\begin{array}{rcl}\tau & ::= & \Sigma \\ & \mid & \mathsf{class}\langle\Sigma\rangle \\ & \mid & \mathsf{mixin}\langle\Sigma_{new}, \Sigma_{red}, \Sigma_{exp}\rangle \\ \Sigma & ::= & \{m_i : \tau_{m_i}{}^{i\in I}\}\end{array}$$

Table 3: Syntax of types.

As we left method bodies unspecified (see Section 3), we must assume that there is a type system for the underlying part of SOOL, i.e., method bodies, records and objects. We will denote this typing with $\Vdash$, i.e., we will write $\Gamma \Vdash f : \tau$ and $\Gamma \Vdash \{m_i : \tau_{m_i} = f_i{}^{i\in I}\} : \{m_i : \tau_{m_i}{}^{i\in I}\}$. Rules

for $\Vdash$ are obviously left implicit and $\Vdash$-statements are used as assumptions in other typing rules. SOOL *typing environments* are sets of typing assumptions of the form $x:\tau$ and $m:\tau$, where x is a variable and m is a method name. As it is standard, $\Gamma, \bigcup_{i\in I} m_i:\tau_{m_i}$ (resp. $\Gamma, x:\tau$) will denote the environment obtained by adding to Γ all the assumptions $m_i:\tau_{m_i}$ (resp. the assumption $x:\tau$), provided that the resulting environment is still a typing environment (i.e., typing assumptions concern distinct variables and method names).

Typing rules rely strongly on a subtype relation $<:$ whose judgments are of the form $\tau_1 <: \tau_2$. This subtype relation depends obviously on the nature of the SOOL calculus we choose (for instance, if SOOL is instantiated with a functional calculus, the subtype relation may have a rule for the co/contra-variance behavior of the arrow type), but as an essential constraint it must contain the standard *width subtyping* rule on record types, that is:

$$\frac{\Sigma_2 \subseteq \Sigma_1}{\Sigma_1 <: \Sigma_2}\ (\mathit{width})$$

Informally speaking, the (*width*) rule captures the intuition that the subtype is the one with more methods. Such a rule for record types is the only one we need to make explicit here.

Class types $\mathsf{class}\langle\Sigma\rangle$ and mixin types $\mathsf{mixin}\langle\Sigma_{new}, \Sigma_{red}, \Sigma_{exp}\rangle$ are formed over record types. A class type includes the type of its methods $\{m_i:\tau_{m_i}{}^{i\in I}\}$. Notice that this is just a "specimen" rule, since a concrete language underlying SOOL might have also private and protected methods (in these cases, the class type would also collect the names of private and protected methods, as it happens for protected ones in the calculus in [18]). Mixin types encode the following information:

- Σ_{new}, Σ_{red} are the exact types of mixin methods (which we call *new* and *redefined*, respectively).
- Σ_{exp} are the methods that are not redefined by the mixin but are still *expected* to be supported by the superclass since they may be called by other mixin methods.

Thus, $\Sigma_{exp} \cup \Sigma_{red}$ defines completely the type requirements for the superclass, while Σ_{new} defines the additional methods introduced by the mixin itself. Moreover, the following condition of well-formedness guarantees that name clashes among different families of methods do not occur.

Definition 1 (Well-formedness of mixin types).
A mixin type $\mathsf{mixin}\langle\Sigma_{new}, \Sigma_{red}, \Sigma_{exp}\rangle$ *is* well-formed *if and only if:*

- $\mathit{Subj}(\Sigma_{red}) \cap \mathit{Subj}(\Sigma_{exp}) = \emptyset$
- $\mathit{Meth}(\Sigma_{new}) \cap \mathit{Meth}(\Sigma_{red} \cup \Sigma_{exp}) = \emptyset$

In the above definition the first clause is a syntactic choice: any required method is either in Σ_{red} or in Σ_{exp} according to whether its body is redefined or not by the mixin. The second clause, instead, is an important consistency condition between defined and required methods, that must be totally disjoint also with respect to method names occurring in their types; this clause is justified in Section 5.1.

The typing rules for SOOL values are in Table 4. The most interesting one is the (*mixin*) rule: The clause (c_1) concerns the types of the new methods defined in the mixin and uses type assumptions about expected and redefined methods; the clause (c_2) checks, for each redefined method, that its body is well typed using the types of new methods, checked by (c_1). Notice that the body of a redefined method is well typed when its type is a subtype of the type of the original method of the superclass, which is explicitly given in the mixin definition and thus assumed for *next*. Let

$$\frac{}{\Gamma, x:\tau \vdash x:\tau}\ (proj1) \qquad \frac{}{\Gamma, m:\tau \vdash m:\tau}\ (proj2) \qquad \frac{\Gamma \Vdash \{m_i:\tau_{m_i} = f_i\ ^{i\in I}\} : \{m_i:\tau_{m_i}\ ^{i\in I}\}}{\Gamma \vdash \{m_i:\tau_{m_i} = f_i\ ^{i\in I}\} : \{m_i:\tau_{m_i}\ ^{i\in I}\}}\ (rec)$$

$$\frac{\Gamma \vdash \{m_i:\tau_{m_i} = f_i\ ^{i\in I}\} : \{m_i:\tau_{m_i}\ ^{i\in I}\}}{\Gamma \vdash \mathsf{class}\ [m_i:\tau_{m_i} = f_i\ ^{i\in I}]\ \mathsf{end} : \mathsf{class}\langle\{m_i:\tau_{m_i}\ ^{i\in I}\}\rangle}\ (class)$$

$$\frac{\begin{array}{ll}(c_1) & \Gamma, \bigcup_{i\in I} m_i:\tau_{m_i}, \bigcup_{k\in K} m_k:\tau_{m_k} \vdash \{m_j:\tau_{m_j} = f_j\ ^{j\in J}\} : \{m_j:\tau_{m_j}\ ^{j\in J}\} \\ (c_2) & \Gamma, \bigcup_{i\in I} m_i:\tau_{m_i}, \bigcup_{k\in K} m_k:\tau_{m_k}, \bigcup_{j\in J} m_j:\tau_{m_j}, next:\tau_{m_r} \Vdash f_r:\tau'_{m_r} \quad \tau'_{m_r} <: \tau_{m_r} \quad \forall r \in K \\ (c_3) & Subj(\Sigma_{red}) \cap Subj(\Sigma_{exp}) = \emptyset \qquad Meth(\Sigma_{new}) \cap Meth(\Sigma_{red} \cup \Sigma_{exp}) = \emptyset\end{array}}{\Gamma \vdash \begin{array}{l}\mathsf{mixin} \\ \quad \mathsf{expect}[m_i:\tau_{m_i}\ ^{i\in I}] \\ \quad \mathsf{redef}[m_k:\tau_{m_k}\ \mathsf{with}\ f_k\ ^{k\in K}] \\ \quad \mathsf{def}[m_j:\tau_{m_j} = f_j\ ^{j\in J}] \\ \mathsf{end}\end{array} : \mathsf{mixin}\langle\Sigma_{new}, \Sigma_{red}, \Sigma_{exp}\rangle}\ (mixin)$$

where $\Sigma_{new} = \{m_j:\tau_{m_j}\ ^{j\in J}\}, \Sigma_{red} = \{m_k:\tau_{m_k}\ ^{k\in K}\}, \Sigma_{exp} = \{m_i:\tau_{m_i}\ ^{i\in I}\}$

Table 4: Typing rules for SOOL values

$$\frac{\Gamma \vdash exp : \{m_i:\tau_{m_i}\ ^{i\in I}\} \qquad j \in I}{\Gamma \vdash exp \Leftarrow m_j:\tau_{m_j}}\ (lookup) \qquad \frac{\Gamma \vdash exp : \mathsf{class}\langle\{m_i:\tau_{m_i}\ ^{i\in I}\}\rangle}{\Gamma \vdash \mathsf{new}\ exp : \{m_i:\tau_{m_i}\ ^{i\in I}\}}\ (new)$$

$$\frac{\begin{array}{l}\Gamma \vdash v : \mathsf{mixin}\langle\Sigma_{new}, \Sigma_{red}, \Sigma_{exp}\rangle \\ \Gamma \vdash exp : \mathsf{class}\langle\Sigma_b\rangle \\ \Sigma_b <: (\Sigma_{exp} \cup \Sigma_{red}) \\ Meth(\Sigma_b) \cap Meth(\Sigma_{new}) = \emptyset\end{array}}{\Gamma \vdash v \diamond exp : \mathsf{class}\langle\Sigma_b \cup \Sigma_{new}\rangle}\ (mixin\ app)$$

Table 5: Typing rules for SOOL expressions.

us observe that, since depth subtyping[3] is not defined in this calculus, a redefined method keeps the type of its old version (the one of the superclass) in the mixin type, even though it is sound to assume that the method body can type check "internally" with a smaller type (clause $\tau'_{m_r} <: \tau_{m_r}$). Finally, ($c_3$) checks the well-formedness of the mixin type.

The typing rules for SOOL expressions are in Table 5. The first premise of the rule (*mixin app*), $\Sigma_b <: (\Sigma_{exp} \cup \Sigma_{red})$, requires that the superclass provides all the methods that the mixin expects and redefines. Notice that the superclass may have more methods than those required by the mixin constraints. Thus, the type of the mixin application expression is a class type containing both the signatures of all the methods supplied by the superclass (Σ_b) and those of the new methods defined by the mixin (Σ_{new}). Because of these additional new methods added by Σ_b, a further type check is needed in order to avoid accidental overriding during the mixin application, i.e., $Meth(\Sigma_b) \cap Meth(\Sigma_{new}) = \emptyset$. This clause checks that no unexpected method names of Σ_b produce conflicts with method names occurring in Σ_{new}; this requirement has the same purpose of the second clause of Definition 1 (about well-formed mixin types).

[3] Depth subtyping permits also the redefinition of methods' types. An algorithmic subtyping rule for depth and width subtyping is the following one (see, e.g., [44]):

$$\frac{J \subseteq I \qquad \tau_{m_j} <: \tau'_{m_j}\ \forall j \in J}{\{m_i:\tau_{m_i}\ ^{i\in I}\} <: \{m_j:\tau'_{m_j}\ ^{j\in J}\}}\ (width\text{-}depth)$$

It is straightforward to verify the following properties that derive directly from the syntax and from the type system.

Property 1 *If exp is a well-typed* SOOL *expression, and x is a variable of class type, then x can occur at most once in exp.*

Property 2 *If exp is a well-typed* SOOL *expression, and it is neither a value nor a variable, then it cannot have a mixin type.*

It is easy to prove that a well-typed application of a mixin to a class generates a subtype of the class argument.

Property 3 *If* $\Gamma \vdash v : \mathsf{mixin}\langle \Sigma_{new}, \Sigma_{red}, \Sigma_{exp} \rangle$, $\Gamma \vdash exp : \mathsf{class}\langle \Sigma_b \rangle$ *and* $\Gamma \vdash v \diamond exp : \mathsf{class}\langle \Sigma_d \rangle$, *then* $\mathsf{class}\langle \Sigma_d \rangle \sqsubseteq \mathsf{class}\langle \Sigma_b \rangle$.

Proof. By definition of (*mixin app*) rule, (Table 5), $(\Sigma_b \cup \Sigma_{new}) <: \Sigma_b$. □

4.2 Typing processes

In this section we present the type system for deciding whether a process is well typed. At this stage we are not interested in typing processes in detail, so we will assign simply to a well typed process the constant type proc. A process $\mathsf{receive}(X : \mathsf{proc}).X$ means that it is willing to receive any process and execute it. Observe that MOMI is a higher-order calculus where processes can exchange code that consists either of a process or of a SOOL value (shown in Table 2). Thus,

- the set $\mathcal{T}$ of types is extended to $\mathcal{T}^* = \mathcal{T} \cup \{\mathsf{proc}\}$; if the context is not ambiguous, τ will range over $\mathcal{T}^*$ for denoting a type of a process in general, while it ranges over $\mathcal{T}$ when denoting a type of object-oriented expressions and values;
- we assume $\mathsf{proc} <: \mathsf{proc}$;
- type environments are extended with assertions $id : \tau$ where *id* ranges over *x* and *X*;
- type judgments are of the shape $\Gamma \vdash P : \mathsf{proc}$ saying that the process *P* is well typed from Γ.

Typing rules are defined in Table 6.

$$\frac{}{\Gamma, X : \mathsf{proc} \vdash X : \mathsf{proc}}\ (proj) \qquad \frac{}{\Gamma \vdash \mathbf{nil} : \mathsf{proc}}\ (nil)$$

$$\frac{\Gamma \vdash A : \tau \qquad \Gamma \vdash P : \mathsf{proc}}{\Gamma \vdash \mathsf{send}(A, \ell).P : \mathsf{proc}}\ (send) \qquad \frac{\Gamma, id : \tau \vdash P : \mathsf{proc}}{\Gamma \vdash \mathsf{receive}(id : \tau).P : \mathsf{proc}}\ (receive)$$

$$\frac{\Gamma \vdash P_1 : \mathsf{proc} \qquad \Gamma \vdash P_2 : \mathsf{proc}}{\Gamma \vdash (P_1 \mid P_2) : \mathsf{proc}}\ (comp) \qquad \frac{\Gamma \vdash exp : \tau \qquad \Gamma, x : \tau \vdash P : \mathsf{proc}}{\Gamma \vdash \mathsf{def}\ x = exp\ \mathsf{in}\ P : \mathsf{proc}}\ (def)$$

Table 6: Typing rules for processes

The rule (*send*) states that a process performing a send is well typed if both its argument and the continuation are well typed. For typing a process performing a receive we type the continuation with the information about the type of *id* (rule (*receive*)). Rule (*def*) is standard. For parallel composition (rule (*comp*)) we require that both processes have the same type proc. Notice that if *P* has type proc then all object-oriented sub-expressions of *P* are typed.

Let us observe that we do not use an explicit *subsumption* rule for $<:$ in the type system, but rather an algorithmic discipline, where the use of $<:$ is performed when needed. This has the

important advantage that every typable expression has a unique type. Finally, we will require that a process, in order to be executed on a site, must be closed (i.e., be without free variables), so it must be well-typed under $\Gamma = \emptyset$. It is easy to verify that if a process P is closed, then, for any $\mathsf{send}(A, \ell)$ occurring in P, the free variables of A are bound by an outer def or by an outer receive. This implies that the exchanged code is closed when a send is executed.

Decidability of MOMI's type system, namely uniqueness of types, easily follows from the typing relation that defines a typing rule for each construct.

Lemma 1 (Decidability). *In a given context* Γ*, for any process* P *(with free variables all in the domain of* Γ*) it is decidable whether* $\Gamma \vdash P : \mathsf{proc}$.

Proof. Rules (*new*), (*lookup*) and (*mixin app*) are syntax driven and the subsumption rule for $<:$ is not explicit, thus if an expression is typable, then it is assigned a unique type. Concerning type environments, if *exp* has type τ, then there is a minimal environment Γ (assigning types to free variables of *exp*) such that $\Gamma \vdash exp : \tau$. For any other environment Γ' such that $\Gamma' \vdash exp : \tau$, we have that Γ is included in Γ'. Then, assuming that $<:$ is decidable, typability of processes follows from the shape of rules in Table 6, which are syntax driven. □

4.3 Subtyping on class and mixin types

One of the key ideas of the present paper is the introduction of a novel subtype relation, denoted by $\sqsubseteq$, defined on class and mixin types. This subtype relation will be used to match dynamically the actual parameter's type (send's argument) against the formal parameter's type (receive's argument) during communication. It is of paramount importance to notice that $\sqsubseteq$ is never used in the (local) static type inference. The operational semantics, where $\sqsubseteq$ is instead exploited, is presented in Section 7, but we anticipate here the introduction of $\sqsubseteq$ in order to state the substitution property, which will be a crucial step for proving the subject reduction theorem.

The subtype relation $\sqsubseteq$ is defined in Table 7. The rule ($\sqsubseteq$ *class*) is induced naturally by the structural subtyping on record types. The rule ($\sqsubseteq$ *mixin*): (*i*) allows the subtype to define more new methods; (*ii*) prohibits it from overriding more methods; (*iii*) allows a subtype to require less expected methods. Decidability of $\sqsubseteq$ follows trivially from decidability of $<:$. Note that the rule ($\sqsubseteq$ *mixin*) justifies the choice of keeping Σ_{new} and Σ_{red} records separated: in order to have a sound subtype relation, Σ_{new} must behaves "co-variantly", instead Σ_{red} must behave "invariantly".

$$\frac{\Sigma' <: \Sigma}{\mathsf{class}\langle\Sigma'\rangle \sqsubseteq \mathsf{class}\langle\Sigma\rangle}\ (\sqsubseteq \textit{class})$$

$$\frac{\Sigma'_{new} <: \Sigma_{new} \qquad \Sigma_{exp} <: \Sigma'_{exp} \qquad \Sigma'_{red} = \Sigma_{red}}{\mathsf{mixin}\langle\Sigma'_{new}, \Sigma'_{red}, \Sigma'_{exp}\rangle \sqsubseteq \mathsf{mixin}\langle\Sigma_{new}, \Sigma_{red}, \Sigma_{exp}\rangle}\ (\sqsubseteq \textit{mixin})$$

Table 7: Subtype on class and mixin types.

A common design principle for object-oriented programming languages is to keep the inheritance and the subtyping hierarchies completely separated so that subtyping only exists at the object level. Instead, in our mobile scenario, classes and mixins get a polymorphic nature during the mobile code exchange via $\sqsubseteq$. From this perspective, MOMI can be viewed as a sort of trade-off between the class-based approach and the object-based one: while object-oriented code is structured in a class-based way, mobile object-oriented code is exchanged among processes by a mechanism that is object-based-like.

The relation $\sqsubseteq$ plays the role of a communication pattern: send and receive synchronize (see Table 10) if and only if the sent code is subtyping-compliant with the expected receive's parameter type. Informally speaking, one can receive any class containing more resources than expected. Conversely, we would accept any mixin with weaker requests about methods expected from the superclass.

We aim at guaranteing that if some code was statically type-checked on a site, it can be sent to a different site by means of the send/receive mechanism, and, if its type is subtyping-compliant with the expected parameter type, it will not produce any "message-not-understood" run-time error when merged within the local (well-typed) code, and executed. In order to achieve this communication mechanism, we need to decorate the send's argument with its type. This can be easily done during type checking in the static analysis, in order to produce *compiled processes* to be evaluated (see Definition 6).

5 The substitution problem

The most important property of MOMI's type system is the *type safe substitution*, on which preservation of types during evaluation will rely (see Section 7). Informally speaking, a process of the shape $\mathsf{receive}(id:\tau).P$ will accept any argument of type τ_1, if τ_1 is a subtype of τ, where subtyping coincides with $\sqsubseteq$ in case of mixin and class definitions. To this aim, we need to prove that P remains well typed after replacing the free occurrences of id of type τ with any received code having a subtype of τ.

Thus, this substitution property turns out to be the key point in which MOMI's object-oriented component and coordination component interleave. This will be proved formally in the Section 6, Theorem 1.

In this section, firstly, we discuss informally the problem of accidental name clashes that can occur during the substitution operation based on the $\sqsubseteq$ subtype relation, secondly, we formalize the notion of the proposed substitution.

For simplicity, in the following we will use M and C (possibly with subscripts and superscripts) for denoting mixin definitions and class definitions, having mixin and class types respectively.

5.1 Dynamic name clashes

We recall that the (*mixin app*) rule checks the absence of unintentional overrides, for the mixin application operation, via the condition $Meth(\Sigma_b) \cap Meth(\Sigma_{new}) = \emptyset$ which implies, in particular, that $Subj(\Sigma_b) \cap Subj(\Sigma_{new}) = \emptyset$. This check is a repetition of the analogous check performed when typing the mixin definition in isolation (mixin well-formedness) and it is needed because the actual superclass the mixin will be applied to is not known in advance, namely, the actual superclass may contain more methods than expected. So far the problem at issue is present in general when type checking mixins, therefore also in a sequential context, and independently from code mobility. Its obvious solution, i.e., repeating the check at the moment of the mixin application, relies on the fact that all necessary information is provided by the mixin type and the class type during the type checking of the application.

Let us consider now the specific complications that arise from using mixin and class variables as formal parameters of a receive action. The static well-typedness of an expression like $x \diamond y$ is not sufficient: new unintentional overrides can occur again at run-time when receiving (and replacing) a mixin M and a class C for x and y, respectively, because of names of new methods possibly

added by M and C by $\sqsubseteq$ (defined in Table 7). In this context, the problem at issue differs from the previous one in two points.

Firstly, when receiving a class or mixin to be substituted in a process, we do not want to type check the whole process once again (remember that this is one of our main goals). Thus, the only run-time check consists of comparing the type of what we receive against what we expect. Secondly, differently from static name clashes that have to be considered as errors, dynamic name clashes should be allowed since they represent only an homonymy problem easily solvable with static bindings. Let us explain this matter with some examples. Suppose we have:

$$\mathsf{receive}(x:\mathsf{mixin}\langle\emptyset,\Sigma_{red},\emptyset\rangle).(\mathsf{new}\ (x\diamond C))\Leftarrow n$$

where $\Sigma_{red}=\{m:\tau\}$ and $C:\mathsf{class}\langle\{m:\tau,n:\tau_1\}\rangle$. Obviously, the code is statically well-typed, since n is implicitly inherited during the mixin application. At run-time, the mixin $M:\mathsf{mixin}\langle\Sigma_{new},\Sigma_{red},\emptyset\rangle$, with $\Sigma_{new}=\{n:\tau_2,p:\tau'\}$, can be accepted by receive since $\mathsf{mixin}\langle\Sigma_{new},\Sigma_{red},\emptyset\rangle\sqsubseteq\mathsf{mixin}\langle\emptyset,\Sigma_{red},\emptyset\rangle$. However, at run-time, $M\diamond C$ would generate a (dynamic) name clash due to n in Σ_{new}. Should this be considered an error just like a static name clash? We do not think so. At run-time a name clash is only an homonymy: the (remote) programmer of M defines a method n that can be used internally by M itself; the (local) programmer of C intends to invoke n from the class C when he wrote $(\mathsf{new}\ x\diamond C)\Leftarrow n$, and statically this is the only correct choice. The program of the local programmer would never access n of the received mixin M since it is not explicitly stated in any static type, therefore it is not known by the program. The two n's are completely unrelated, thus the only thing we should guarantee is that they are unrelated also during the execution. In other words, we have to make sure that, if p in M invokes n, the n defined by M is actually executed: in the methods of M, n must be statically bound to the n defined in M. Dynamic binding will be applied for the methods that are explicit in the static interface of mixins and classes, while for the other methods static binding will be adopted.

The same strategy will be applied also to classes received from the network. Consider the following code:

$$\mathsf{receive}(y:\mathsf{class}\langle\{m:\tau\}\rangle).(\mathsf{new}\ (M\diamond y))\Leftarrow n$$

where $M:\mathsf{mixin}\langle\Sigma_{new},\Sigma_{red},\emptyset\rangle$, $\Sigma_{red}=\{m:\tau\}$ and $\Sigma_{new}=\{n:\tau_2\}$. Now we can accept a class $C:\mathsf{class}\langle\{m:\tau,n:\tau_1\}\rangle$, since $\mathsf{class}\langle\{m:\tau,n:\tau_1\}\rangle\sqsubseteq\mathsf{class}\langle\{m:\tau\}\rangle$, and we have to make sure that the n in C is not accidentally overridden by the n in M (since this was not statically declared); indeed, inside the methods in C, n should be statically bound to the implementation provided by C.

This choice on dynamic/static binding will guarantee that dynamic name clashes, due to the subtyping-in-width-based matching at communication time, will not compromise the safety of the system and the reusability/flexibility of existing code. This matter is related to the "width subtyping versus method addition" problem (well known in the object-based setting, see for instance [28]), that in our case boils down to a careful management of these *dynamic name clashes*. This means that, from a technical point of view, we must define a suitable capture-avoid-substitution, indicated by $[\ /\ ]$, which performs all and only the needed renaming of methods in such a way that our type system enjoys the following property: any class or mixin matching by subtyping can be successfully received and safely substituted into the local code.

However, the renaming technique brings some difficulties from a technical point of view, because, as mentioned above, only the *types* of the formal parameter and of the actual parameter are known and compared when receiving some actual code. Let us discuss this point by a simple example, which highlights also some design principles of our type system. Assume we have:

$$\mathsf{receive}(x:\mathsf{mixin}\langle\Sigma_{new},\Sigma_{exp},\emptyset\rangle).\mathsf{receive}(y:\mathsf{class}\langle\Sigma_b\rangle).\ \ldots\ (x\diamond y)$$

where $\Sigma_{exp}=\{n:\tau\}$, $\Sigma_{new}=\{m:\{n:\tau\}\}$ and $\Sigma_b=\{n:\tau\}$. The simplest choice, during static type checking, would be to consider the above mixin definition well formed, and therefore also the application $x\diamond y$, since no method in the type of y is accidentally overridden by a method name of the mixin type. Formally, it looks like we could adopt a weaker condition, $Subj(\Sigma_{new})\cap Subj(\Sigma_{red}\cup\Sigma_{exp})=\emptyset$, for the mixin type well-formedness (and, consequently, $Subj(\Sigma_b)\cap Subj(\Sigma_{new})=\emptyset$ for the typing of mixin application).

Now, let us assume to receive the class $C:\mathsf{class}\langle\{n:\tau,m:\{n:\tau\}\}\rangle$ as the actual parameter for y. To avoid overriding accidentally the method m, we must rename m with a fresh name in order to do a correct substitution. What about renaming the method n occurring in the type of m? Clearly, we must avoid this renaming of n since n occurs in the type of y, in order to ensure that the renamed version of C, say C', still matches the expected requirements declared in the mixin type.

Concerning the mixin side, we assume to have to replace x with the received mixin $M:\mathsf{mixin}\langle\Sigma^1_{new},\emptyset,\emptyset\rangle$ where $\Sigma^1_{new}=\{n:\tau,m:\{n:\tau\}\}$. In principle, according to the choices made for the class, the method n added by M (w.r.t. the type of x) must be renamed with a fresh name, obtaining a new version of M, say $M':\mathsf{mixin}\langle\Sigma'_{new},\emptyset,\emptyset\rangle$, where $\Sigma'_{new}=\{n':\tau,m':\{n':\tau\}\}$. Notice that any occurrence of n in M has be renamed to ensure that typability of M implies typability of M', without requiring any further type checking. However, the type of M' turns out to be completely unrelated to the type of x, and so we cannot guarantee that the application of M to y is still well typed. The simplest choice is clearly not the appropriate one.

The above considerations show how the case of dynamic mixin application for composing local and foreign code in a type-safe way (without requiring any further type checking) requires to consider "mobile" mixins as distinct entities from standard mixins (in a sequential framework). The solution proposed in this paper consists of considering the mixin typing more restrictive, giving priority to the flexibility of communication by $\sqsubseteq$. Namely, we use the stronger constraint $Meth(\Sigma_{new})\cap Meth(\Sigma_{red}\cup\Sigma_{exp})=\emptyset$ when typing statically mixin definitions (and, consequently, $Meth(\Sigma_b)\cap Meth(\Sigma_{new})=\emptyset$, for mixin applications). In this way, we avoid name conflicts between methods occurring in types of methods defined and expected by the mixin (for instance, the code of the above example results not well typed in our system). Given this restriction, the substitution problem in our mobile setting has a smooth solution.

The basic idea is that, if x is of type $\mathsf{class}\langle\Sigma\rangle$, C is of type $\mathsf{class}\langle\Sigma'\rangle$ and $\mathsf{class}\langle\Sigma'\rangle\sqsubseteq\mathsf{class}\langle\Sigma\rangle$, method names are assumed fresh for all names belonging to $Meth(\Sigma')-Meth(\Sigma)$ in $[C/x]$. The case of mixin substitution is dealt with analogously, acting on Σ_{new}. Moreover, we must be sure that a different refreshed version of a class or a mixin value is substituted for each occurrence of the replaced variable. In order to do that formally, in Section 5.2 we define a *refresh* function that allows us to provide the actual substitution algorithm, preserving well-typedness of expressions.

In synthesis, our choice is based on the idea that it is better to rename as little as possible, to minimize the danger of changing the meaning of the code. However, this choice leads to a more restrictive use of types. Stricter types imply that less code is accepted at compile time; nevertheless this is directly under the programmer's control, while unexpected changes or failures taking place in a foreign site are clearly not. Our design decision is to limit such changes and avoid failures.

Remark 1. It is important to note that the restrictive condition discussed above influences only the typing of methods that return an object (i.e., typed with a record type where method names can occur), via a class instantiation. In general, any "new" mixin method can invoke any "expected" method in its body, and in fact the condition above does not affect this.

Moreover, we want to point out that, in the perspective of the implementation, we may want to adopt two distinct rules, using the weaker condition on $Subj()$ for type checking local mixin and mixin applications, while adopting our typing rules based on the $Meth()$ condition only for type checking mixin variables (formal parameters of receive's operations representing mixins, and consequently applications containing those variables). As the focus of this paper is the mobility of mixin-based code, we considered only one rule form (the most restrictive one), but dealing with both rule forms would not add any technical complication.

5.2 Substitution by refresh

We define firstly a *refresh* function that provides the actual substitution algorithm.

Definition 2 (Refresh). *Let v be a class or a mixin definition and $\mathcal{M}$ a set of method names such that methods in $\mathcal{M}$ are methods defined in v; we define $refresh_{\mathcal{M}}(v)$ as the definition v' obtained from v by lexically renaming all the labels belonging to $\mathcal{M}$ with fresh names.*

Then we define a substitution parameterized over a record type, which relies on the function *refresh*. In the following we use the notation $\{y \leftarrow x\}$ to denote the renaming of free occurrences of y with x.

Definition 3 (Parametric Substitution). *Let v denote a class or mixin definition. Let P be a well typed process and x be a free variable of type class or mixin, respectively. For any set of method names $\mathcal{M}$, $P[v/x]_{\mathcal{M}}$ is defined by structural induction on P in the following way:*

- $\mathbf{nil}[v/x]_{\mathcal{M}} \stackrel{def}{=} \mathbf{nil}$;
- $(P_1|P_2)[v/x]_{\mathcal{M}} \stackrel{def}{=} P_1[v/x]_{\mathcal{M}}\,|P_2[v/x]_{\mathcal{M}}$;
- $X[v/x]_{\mathcal{M}} \stackrel{def}{=} X$;
- $(\mathsf{def}\ y = x\ \mathsf{in}\ P)[v/x]_{\mathcal{M}} \stackrel{def}{=} (P\{y \leftarrow x\})[v/x]_{\mathcal{M}}$;
- $(\mathsf{def}\ y = exp\ \mathsf{in}\ P)[v/x]_{\mathcal{M}} \stackrel{def}{=} \mathsf{def}\ y = (exp[v/x]_{\mathcal{M}})\ \mathsf{in}\ P[v/x]_{\mathcal{M}} \qquad exp \not\equiv x$;
- $(\mathsf{def}\ x = exp\ \mathsf{in}\ P)[v/x]_{\mathcal{M}} \stackrel{def}{=} \mathsf{def}\ x = exp\ \mathsf{in}\ P$;
- $(\mathsf{receive}(y:\tau).P)[v/x]_{\mathcal{M}} \stackrel{def}{=} \mathsf{receive}(y:\tau).(P[v/x]_{\mathcal{M}})$;
- $(\mathsf{receive}(x:\tau).P)[v/x]_{\mathcal{M}} \stackrel{def}{=} \mathsf{receive}(x:\tau).P$;
- $(\mathsf{send}(A,\ell).P)[v/x]_{\mathcal{M}} \stackrel{def}{=} \mathsf{send}(A[v/x]_{\mathcal{M}},\ell).(P[v/x]_{\mathcal{M}})$,

where

- $y[v/x]_{\mathcal{M}} \stackrel{def}{=} y$;
- $x[v/x]_{\mathcal{M}} \stackrel{def}{=} refresh_{\mathcal{M}}(v)$;
- $(\mathsf{new}\ exp)[v/x]_{\mathcal{M}} \stackrel{def}{=} \mathsf{new}\ (exp[v/x]_{\mathcal{M}})$;
- $(exp \Leftarrow m)[v/x]_{\mathcal{M}} \stackrel{def}{=} (exp[v/x]_{\mathcal{M}}) \Leftarrow m$;
- $(v_1 \diamond exp)[v/x]_{\mathcal{M}} \stackrel{def}{=} (v_1[v/x]_{\mathcal{M}}) \diamond (exp[v/x]_{\mathcal{M}})$.

This substitution takes place only on well-typed terms, so that all the possible well-typed cases are those considered in the above definition. Moreover, it is also a standard capture-avoid-substitution since the variables of binders def and receive with the same name are ignored. Any

time a substitution on a free variable x actually takes place, *refresh* is used, thus fulfilling the requirement that a different refreshed version of a class or a mixin value is substituted for each occurrence of the variable x. Therefore, for instance, in the case:

$$(\mathsf{def}\ y = x\ \mathsf{in}\ P)[v/x]_{\mathcal{M}} \stackrel{def}{=} (P\{y \leftarrow x\})[v/x]_{\mathcal{M}}$$

we "skip" the *aliasing* step (being y the "alias" of x) by performing directly the desired substitutions in the continuation P, so using a different refreshed version of v for each substitution.

Thus, a general substitution can be defined using the above parametric substitution for dealing with crucial cases concerning the replacement of class and mixin variables.

Definition 4 (Substitution). *The substitution* $[\ /\ ]$ *is defined as follows:*

- *If x is of type* $\mathsf{class}\langle\Sigma\rangle$ *and C is a class definition of type* $\mathsf{class}\langle\Sigma'\rangle$, *with* $\mathsf{class}\langle\Sigma'\rangle \sqsubseteq \mathsf{class}\langle\Sigma\rangle$, *then*
$$[C/x] \stackrel{def}{=} [C/x]_{Meth(\Sigma')-Meth(\Sigma)}.$$
- *If x is of type* $\mathsf{mixin}\langle\Sigma_{new},\Sigma_{red},\Sigma_{exp}\rangle$ *and M is a mixin definition of type* $\mathsf{mixin}\langle\Sigma'_{new},\Sigma'_{red},\Sigma'_{exp}\rangle$, *with* $\mathsf{mixin}\langle\Sigma'_{new},\Sigma'_{red},\Sigma'_{exp}\rangle \sqsubseteq \mathsf{mixin}\langle\Sigma_{new},\Sigma_{red},\Sigma_{exp}\rangle$, *then*
$$[M/x] \stackrel{def}{=} [M/x]_{Meth(\Sigma'_{new})-Meth(\Sigma_{new})}.$$
- *In all the other cases the substitution* $[\ /\ ]$ *is intended as the standard replacement.*

The above definition of substitution highlights a simple property: new methods added by a class or a mixin by subtyping never change their original behavior, since they will never be accidentally redefined (due to a name clash). With our solution, these new methods are hidden by applying the *refresh* during substitution. This is very similar to the "privacy-via-subsumption" approach of [46]. Notice that we only rename methods that do not appear in the type of the variable x. This second constraint ensures a basic property: the refreshed version C' of C has a type τ' such that $\tau' \sqsubseteq \mathsf{class}\langle\Sigma\rangle$. The same holds for refreshed mixins.

¿From the point of view of the implementation, this formal treatment of "global" fresh names can be solved with static binding for the mentioned methods; the technique of using the static types of variables and the actual types of substituted mixin and class definitions may recall the approach of [29] of allowing overriding only for methods declared in the mixin's *inheritance interface*.

6 Properties of typing

The core of this section is Theorem 1, which proves the preservation of well-typedness of processes under our definition of substitution.

The following lemma is a preliminary technical step for proving that narrowing both class types and mixin types in a mixin application expression preserve well-typedness.

Lemma 2 (Type correctness of refreshing).

Case a. Let x and C be a variable and a class value respectively, such that:

- $x : \mathsf{class}\langle\Sigma\rangle$ *and* $C : \mathsf{class}\langle\Sigma'\rangle$, *where*
- $\mathsf{class}\langle\Sigma'\rangle \sqsubseteq \mathsf{class}\langle\Sigma\rangle$.

Then, $refresh_{Meth(\Sigma')-Meth(\Sigma)}(C) = C^*$, *where* $C^* : \mathsf{class}\langle\Sigma^*\rangle$ *satisfies the following conditions:*

(i) $\mathsf{class}\langle\Sigma^*\rangle \sqsubseteq \mathsf{class}\langle\Sigma\rangle$*;*
(ii) for any record type Σ'', $Meth(\Sigma'') \cap Meth(\Sigma) = \emptyset$ *implies* $Meth(\Sigma'') \cap Meth(\Sigma^*) = \emptyset$.

Case b. Let x *and* M *be a variable and a mixin value respectively, such that*

– $x : \mathsf{mixin}\langle\Sigma_{new}, \Sigma_{red}, \Sigma_{exp}\rangle$ *and* $M : \mathsf{mixin}\langle\Sigma'_{new}, \Sigma'_{red}, \Sigma'_{exp}\rangle$ *where*
– $\mathsf{mixin}\langle\Sigma'_{new}, \Sigma'_{red}, \Sigma'_{exp}\rangle \sqsubseteq \mathsf{mixin}\langle\Sigma_{new}, \Sigma_{red}, \Sigma_{exp}\rangle$.

Then, $refresh_{Meth(\Sigma'_{new}) - Meth(\Sigma_{new})}(M) = M^*$ *where* $M^* : \mathsf{mixin}\langle\Sigma^*_{new}, \Sigma'_{red}, \Sigma'_{exp}\rangle$ *satisfies the following conditions:*

(i) $\mathsf{mixin}\langle\Sigma^*_{new}, \Sigma'_{red}, \Sigma'_{exp}\rangle \sqsubseteq \mathsf{mixin}\langle\Sigma_{new}, \Sigma_{red}, \Sigma_{exp}\rangle$*;*
(ii) for any record type Σ'', $Meth(\Sigma'') \cap Meth(\Sigma_{new}) = \emptyset$ *implies* $Meth(\Sigma'') \cap Meth(\Sigma^*_{new}) = \emptyset$.

Proof. Case a. By definition of $\sqsubseteq$, $\mathsf{class}\langle\Sigma'\rangle \sqsubseteq \mathsf{class}\langle\Sigma\rangle$ means $\Sigma' \equiv \Sigma \cup \Sigma_2$ for some Σ_2. Since the refresh on C does not affect method names occurring in Σ, then $C^* : \mathsf{class}\langle\Sigma^*\rangle$ and $\Sigma^* \equiv \Sigma \cup \Sigma^*_2$, that is, $\mathsf{class}\langle\Sigma^*\rangle \sqsubseteq \mathsf{class}\langle\Sigma\rangle$. Point *(ii)* follows from the fact that method names of $Meth(\Sigma^*)$ either belong to $Meth(\Sigma)$ or are fresh names.

Case b. By (*mixin*) rule, $Meth(\Sigma'_{new}) \cap Meth(\Sigma'_{red} \cup \Sigma'_{exp}) = \emptyset$; by definition of $\sqsubseteq$, $\Sigma'_{new} \equiv \Sigma_{new} \cup \Sigma_2$ for some Σ_2 and $\Sigma_{exp} <: \Sigma'_{exp}$ (while $\Sigma'_{red} = \Sigma_{red}$). Thus, since renaming acts only on method names of Σ'_{new} that do not occur in Σ_{new}, we have $M^* : \mathsf{mixin}\langle\Sigma^*_{new}, \Sigma_{red}, \Sigma'_{exp}\rangle$, where $\Sigma^*_{new} \equiv \Sigma_{new} \cup \Sigma^*_2$. Then, we have $\mathsf{mixin}\langle\Sigma^*_{new}, \Sigma_{red}, \Sigma'_{exp}\rangle \sqsubseteq \mathsf{mixin}\langle\Sigma_{new}, \Sigma_{red}, \Sigma_{exp}\rangle$. Finally, point *(ii)* follows from the fact that method names of $Meth(\Sigma^*_{new})$ either belong to $Meth(\Sigma_{new})$ or are fresh names.

□

The above lemma shows that the refreshed version of a class or a mixin actually used in the substitution of Definition 4, preserves both the subtype relation $\sqsubseteq$ enjoyed by the original version and the absence of name clashes enjoyed by the variable to be replaced. This is a key point for proving that replacing classes and mixins with smaller types to variables inside mixin application expressions preserves well-typedness of the whole expression.

In the following, in an expression of the shape $M_n \diamond (\ldots \diamond (M_1 \diamond x) \ldots)$, when we need to refer to a specific record type of the mixin expression M_i we will use the notation Σ^i ($1 \leq i \leq n$). Moreover, if $\tau \equiv \mathsf{class}\langle\Sigma\rangle$, then $Meth(\tau)$ will be a short for $Meth(\Sigma)$.

Lemma 3 (Narrowing Class Types). *If*

$$\begin{array}{ll} \Gamma, x : \mathsf{class}\langle\Sigma_b\rangle \vdash M_n \diamond (\ldots \diamond (M_1 \diamond x) \ldots) : \tau & \textit{and} \\ \Gamma \vdash C : \mathsf{class}\langle\Sigma_c\rangle & \textit{with } \mathsf{class}\langle\Sigma_c\rangle \sqsubseteq \mathsf{class}\langle\Sigma_b\rangle, \end{array}$$

then $\Gamma \vdash (M_n \diamond (\ldots \diamond (M_1 \diamond x) \ldots))[C/x] : \tau'$ *where*

– $\tau' \sqsubseteq \tau$*;*
– $Meth(\tau') \cap Meth(\Sigma) = \emptyset$ *for any* Σ *such that* $Meth(\tau) \cap Meth(\Sigma) = \emptyset$.

Proof. By induction on the number n of mixin applications, taking into account that a refreshed version of C, say C^*, is actually replaced to x according to Definition 4. By Lemma 2-Case a-*(i)*, we have $C^* : \mathsf{class}\langle\Sigma^*_c\rangle$, where $\mathsf{class}\langle\Sigma^*_c\rangle \sqsubseteq \mathsf{class}\langle\Sigma_b\rangle$.

Base. $n = 1$. The (*mixin app*) rule still applies to M_1 and C^* since

– $\Sigma^*_c <: \Sigma_b$ and $\Sigma_b <: (\Sigma^1_{exp} \cup \Sigma^1_{red})$ (since $M_1 \diamond x$ is well-typed), then $\Sigma^*_c <: (\Sigma^1_{exp} \cup \Sigma^1_{red})$;
– $Meth(\Sigma_b) \cap Meth(\Sigma^1_{new}) = \emptyset$ (since $M_1 \diamond x$ is well-typed) implies $Meth(\Sigma^*_c) \cap Meth(\Sigma^1_{new}) = \emptyset$ by Lemma 2-Case a-*(ii)*.

Therefore, we obtain a deduction of $\Gamma \vdash M_1 \diamond x[C/x] : \tau'$, and $\tau' \equiv \mathsf{class}\langle \Sigma_b \cup \Sigma^1_{new} \cup \Sigma' \rangle$ for some Σ' containing new methods added by C, w.r.t. to x, possibly renamed. Then, $\tau' \sqsubseteq \mathsf{class}\langle \Sigma_b \cup \Sigma^1_{new} \rangle \equiv \tau$. Moreover, the thesis on $Meth(\tau')$ holds since $\tau \equiv \mathsf{class}\langle \Sigma_b \cup \Sigma^1_{new} \rangle$ and $\tau' \equiv \mathsf{class}\langle \Sigma_b \cup \Sigma^1_{new} \cup \Sigma' \rangle$, where any method name occurring in Σ' either belongs to $Meth(\Sigma_b)$ or is renamed with a fresh name (by definition of C^*).

Inductive Step. The last applied rule is:

$$\frac{\begin{array}{l}\Gamma, x : \mathsf{class}\langle \Sigma_b \rangle \vdash M_n : \mathsf{mixin}\langle \Sigma^n_{new}, \Sigma^n_{red}, \Sigma^n_{exp} \rangle \\ \Gamma, x : \mathsf{class}\langle \Sigma_b \rangle \vdash M_{n-1} \diamond (\ldots \diamond (M_1 \diamond x) \ldots) : \mathsf{class}\langle \Sigma_{n-1} \rangle\end{array}}{\Gamma, x : \mathsf{class}\langle \Sigma_b \rangle \vdash M_n \diamond (\ldots \diamond (M_1 \diamond x) \ldots) : \tau} \ (\textit{mixin app})$$

where $\tau \equiv \mathsf{class}\langle \Sigma_{n-1} \cup \Sigma^n_{new} \rangle$, $\Sigma_{n-1} <: (\Sigma^n_{exp} \cup \Sigma^n_{red})$ and $Meth(\Sigma_{n-1}) \cap Meth(\Sigma^n_{new}) = \emptyset$, by definition of (*mixin app*) rule.

By induction hypothesis:

- $\Gamma \vdash (M_{n-1} \diamond (\ldots \diamond (M_1 \diamond x) \ldots))[C/x] : \tau^*$ with $\tau^* \sqsubseteq \mathsf{class}\langle \Sigma_{n-1} \rangle$ and
- $Meth(\tau^*) \cap Meth(\Sigma) = \emptyset$ for any Σ such that $Meth(\Sigma_{n-1}) \cap Meth(\Sigma) = \emptyset$. Thus, in particular, since $Meth(\Sigma_{n-1}) \cap Meth(\Sigma^n_{new}) = \emptyset$ we have $Meth(\tau^*) \cap Meth(\Sigma^n_{new}) = \emptyset$.

Then, the (*mixin app*) rule still applies to

$$\Gamma \vdash M_n : \mathsf{mixin}\langle \Sigma^n_{new}, \Sigma^n_{red}, \Sigma^n_{exp} \rangle$$

and

$$\Gamma \vdash M_{n-1} \diamond (\ldots \diamond (M_1 \diamond x) \ldots)[C/x] : \tau^*$$

and the thesis follows as in the Base case.

□

Corollary 1 (Narrowing Class Assumptions). *If*

$$\Gamma, x : \mathsf{class}\langle \Sigma_b \rangle \vdash M_n \diamond (\ldots \diamond (M_1 \diamond x) \ldots) : \tau$$

and $\mathsf{class}\langle \Sigma_1 \rangle \sqsubseteq \mathsf{class}\langle \Sigma_b \rangle$ *is such that all method names belonging to* $Meth(\Sigma_1) - Meth(\Sigma_b)$ *are (globally) fresh names, then* $\Gamma, x : \mathsf{class}\langle \Sigma_1 \rangle \vdash M_n \diamond (\ldots \diamond (M_1 \diamond x) \ldots) : \tau'$ *where*

- $\tau' \sqsubseteq \tau$;
- $Meth(\tau') \cap Meth(\Sigma) = \emptyset$ *for any* Σ *such that* $Meth(\tau) \cap Meth(\Sigma) = \emptyset$.

Proof. Let C be any class of type $\mathsf{class}\langle \Sigma_1 \rangle$. Since x has only one occurrence in the mixin application (Property 1), then by Lemma 3 $\Gamma \vdash (M_n \diamond (\ldots \diamond (M_1 \diamond C) \ldots)) : \tau'$, where $Meth(\Sigma_1) - Meth(\Sigma_b)$ are already fresh names. Then we can replace the assumption $x : \mathsf{class}\langle \Sigma_1 \rangle$ to the deduction of $C : \mathsf{class}\langle \Sigma_1 \rangle$ in the deduction of $\Gamma \vdash (M_n \diamond (\ldots \diamond (M_1 \diamond C) \ldots)) : \tau'$. The rule (*mixin app*) still applies, therefore we obtain a deduction of $\Gamma, x : \mathsf{class}\langle \Sigma_1 \rangle \vdash M_n \diamond (\ldots \diamond (M_1 \diamond x) \ldots) : \tau'$. □

Lemma 4 (Narrowing Mixin Types). *Let exp be a mixin application expression. If*

$\Gamma, x : \mathsf{mixin}\langle \Sigma_{new}, \Sigma_{red}, \Sigma_{exp} \rangle \vdash exp : \tau$ *and*
$\Gamma \vdash M : \mathsf{mixin}\langle \Sigma'_{new}, \Sigma'_{red}, \Sigma'_{exp} \rangle$, *where* $\mathsf{mixin}\langle \Sigma'_{new}, \Sigma'_{red}, \Sigma'_{exp} \rangle \sqsubseteq \mathsf{mixin}\langle \Sigma_{new}, \Sigma_{red}, \Sigma_{exp} \rangle$

then $\Gamma \vdash exp[M/x] : \tau'$, *where*

- $\tau' \sqsubseteq \tau$;

– $Meth(\tau') \cap Meth(\Sigma) = \emptyset$ *for any* Σ *such that* $Meth(\tau) \cap Meth(\Sigma) = \emptyset$.

Proof. By structural induction on *exp*. For simplicity, we only consider the interesting case when x occurs in *exp*.

Base. $exp \equiv v_1 \diamond v_2$ where $v_1 \equiv x$ and v_2 is a class value.
By looking at the typing rules, $\Gamma, x : \mathsf{mixin}\langle \Sigma_{new}, \Sigma_{red}, \Sigma_{exp} \rangle \vdash exp : \tau$ implies that the last applied rule is a (*mixin app*) rule, that is:

$$\frac{\begin{array}{l}(1)\ \Gamma, x : \mathsf{mixin}\langle \Sigma_{new}, \Sigma_{red}, \Sigma_{exp} \rangle \vdash x : \mathsf{mixin}\langle \Sigma_{new}, \Sigma_{red}, \Sigma_{exp} \rangle \\ (2)\ \Gamma, x : \mathsf{mixin}\langle \Sigma_{new}, \Sigma_{red}, \Sigma_{exp} \rangle \vdash v_2 : \mathsf{class}\langle \Sigma_c \rangle \\ (3)\ \Sigma_c <: (\Sigma_{exp} \cup \Sigma_{red}) \\ (4)\ Meth(\Sigma_c) \cap Meth(\Sigma_{new}) = \emptyset\end{array}}{\Gamma, x : \mathsf{mixin}\langle \Sigma_{new}, \Sigma_{red}, \Sigma_{exp} \rangle \vdash x \diamond v_2 : \mathsf{class}\langle \Sigma_c \cup \Sigma_{new} \rangle}\ (\textit{mixin app})$$

Now we replace to x a refreshed version of M, say M^*, such that $M^* : \mathsf{mixin}\langle \Sigma^*_{new}, \Sigma'_{red}, \Sigma'_{exp} \rangle$ where $\mathsf{mixin}\langle \Sigma^*_{new}, \Sigma'_{red}, \Sigma'_{exp} \rangle \sqsubseteq \mathsf{mixin}\langle \Sigma_{new}, \Sigma_{red}, \Sigma_{exp} \rangle$ by Lemma 2-Case b-*(i)*, i.e., $\Sigma^*_{new} <: \Sigma_{new}$, and $\Sigma_{exp} <: \Sigma'_{exp}$ by hypothesis (while $\Sigma'_{red} = \Sigma_{red}$).
The (*mixin app*) rule still applies since:
– $\Sigma_c <: (\Sigma'_{exp} \cup \Sigma_{red})$ follows from condition (3) and $\Sigma_{exp} <: \Sigma'_{exp}$;
– $Meth(\Sigma_c) \cap Meth(\Sigma^*_{new}) = \emptyset$ follows from (4) using Lemma 2-Case b-*(ii)*.
So we obtain a deduction of

$$\Gamma \vdash M^* \diamond v_2 : \mathsf{class}\langle \Sigma_c \cup \Sigma^*_{new} \rangle,$$

where $\tau' \equiv \mathsf{class}\langle \Sigma_c \cup \Sigma^*_{new} \rangle \sqsubseteq \tau \equiv \mathsf{class}\langle \Sigma_c \cup \Sigma_{new} \rangle$ since $\Sigma^*_{new} <: \Sigma_{new}$.
Moreover, according to the definition of the refreshed version M^* used in the substitution, any method name occurring in Σ^*_{new} either occurs in Σ_{new} or has a fresh name. Then, the thesis on $Meth(\tau')$ easily follows.

Inductive Step. $exp \equiv v \diamond exp_1$ where exp_1 is a mixin application expression. We have two possible cases:

1. $v \not\equiv x$. Let v be such that $v : \mathsf{mixin}\langle \Sigma^v_{new}, \Sigma^v_{red}, \Sigma^v_{exp} \rangle$. By (*mixin app*) rule, $\Gamma \vdash exp_1 : \mathsf{class}\langle \Sigma_c \rangle$, where $\Sigma_c <: (\Sigma^v_{exp} \cup \Sigma^v_{red})$ and $Meth(\Sigma_c) \cap Meth(\Sigma^v_{new}) = \emptyset$, so that

$$\Gamma, x : \mathsf{mixin}\langle \Sigma_{new}, \Sigma_{red}, \Sigma_{exp} \rangle \vdash v \diamond exp_1 : \tau \equiv \mathsf{class}\langle \Sigma_c \cup \Sigma^v_{new} \rangle$$

By induction hypothesis:
– $\Gamma \vdash exp_1[M/x] : \mathsf{class}\langle \Sigma^* \rangle$ where $\mathsf{class}\langle \Sigma^* \rangle \sqsubseteq \mathsf{class}\langle \Sigma_c \rangle$;
– $Meth(\Sigma_c) \cap Meth(\Sigma^v_{new}) = \emptyset$ implies $Meth(\Sigma^*) \cap Meth(\Sigma^v_{new}) = \emptyset$.
Thus, (*mixin app*) rule applies and we obtain

$$\Gamma \vdash v \diamond exp_1[M/x] : \mathsf{class}\langle \Sigma^* \cup \Sigma^v_{new} \rangle$$

In particular, we have $\tau' \equiv \mathsf{class}\langle \Sigma^* \cup \Sigma^v_{new} \rangle \sqsubseteq \tau \equiv \mathsf{class}\langle \Sigma_c \cup \Sigma^v_{new} \rangle$ since $\mathsf{class}\langle \Sigma^* \rangle \sqsubseteq \mathsf{class}\langle \Sigma_c \rangle$.
The thesis on $Meth(\tau')$ trivially holds since any method name occurring in Σ^* either occurs in Σ_c or is a fresh name.

2. $v \equiv x$. Firstly, we use the proof of the above case to type $x \diamond (exp_1[M/x])$, and then use the proof of the Base case to replace M to x and type the application $(x[M/x]) \diamond (exp_1[M/x])$. Notice that this proof relies on the fact that a new refreshed version of M is replaced to x, independently from any substitution of x with M inside exp_1, according to Definition 4.

□

We observe that an analogous corollary as Corollary 1 does not hold in the case when the assumption has a mixin type, because the assumption $x : \mathsf{mixin}\langle \Sigma_{new}, \Sigma_{red}, \Sigma_{exp} \rangle$ can have multiple occurrences inside the same mixin application. For instance, $x \diamond (x \diamond C)$ can be typed only if in the type of $x : \mathsf{mixin}\langle \Sigma_{new}, \Sigma_{red}, \Sigma_{exp} \rangle$ we have $\Sigma_{new} = \emptyset$. Instead, using the assumption $x : \mathsf{mixin}\langle \Sigma'_{new}, \Sigma'_{red}, \Sigma'_{exp} \rangle$ with $\Sigma'_{new} \neq \emptyset$, we can type $(x \diamond C)$ but not $x \diamond (x \diamond C)$. Thus, we cannot narrow the types of assumptions in the case of mixin variables. However, Lemmas 3 and 4, together with Corollary 1, are all we need for proving the substitution property that is relevant for our purpose (see Theorem 1). This substitution property will be crucial for proving the subject reduction theorem in Section 7.

For the sake of simplicity, in order to deal with $<:$ and $\sqsubseteq$ at the same time, we introduce the meta notation:

$$\tau_1 \preccurlyeq \tau_2 = \begin{cases} \tau_1 \sqsubseteq \tau_2 & \text{if } \tau_1 \text{ is a mixin or a class type} \\ \tau_1 <: \tau_2 & \text{otherwise} \end{cases}$$

We firstly consider the case when we replace values inside expressions (the generalization to replacement of expressions inside expressions would be straightforward but it is not necessary for our purpose, since only values and processes can be exchanged in the operational semantics). Then, we generalize the property to well-typed processes, that is, processes having type proc from a suitable context Γ ($\Gamma = \emptyset$ if the process is closed): namely, an identifier *id* can be replaced with an argument *A*, being a value or a process (see Table 2), inside a process, preserving well-typedness.

Lemma 5. *Let exp and v be an object-oriented expression and a closed object-oriented value, respectively. If* $\Gamma, x : \tau_x \vdash exp : \tau$ *and* $\Gamma \vdash v : \tau_1$ *where* $\tau_1 \preccurlyeq \tau_x$, *then* $\Gamma \vdash exp[v/x] : \tau'$ *with* $\tau' \preccurlyeq \tau$.

Proof. By induction on the definition of *exp* (Table 1), which coincides with the induction on a deduction of $\Gamma, x : \tau_x \vdash exp : \tau$. The only interesting case of a mixin application expression follows from Lemmas 3 and 4. □

Corollary 2. *If* $\Gamma, x : \mathsf{class}\langle \Sigma_x \rangle \vdash exp : \tau$ *and* $\mathsf{class}\langle \Sigma_1 \rangle \sqsubseteq \mathsf{class}\langle \Sigma_x \rangle$ *is such that all method names belonging to* $Meth(\Sigma_1) - Meth(\Sigma_x)$ *are fresh names, then* $\Gamma, x : \mathsf{class}\langle \Sigma_1 \rangle \vdash exp : \tau'$ *where*

- $\tau' \sqsubseteq \tau$;
- $Meth(\tau') \cap Meth(\Sigma) = \emptyset$ *for any* Σ *such that* $Meth(\tau) \cap Meth(\Sigma) = \emptyset$.

Proof. Let C be any class of type $\mathsf{class}\langle \Sigma_1 \rangle$. Since x has a class type, then there is only one occurrence of x in *exp*, by Property 1 (the case when it does not occur at all is trivial). By Lemma 5, we have $\Gamma \vdash exp[C/x] : \tau'$ where the substitution of $C : \mathsf{class}\langle \Sigma_1 \rangle$ does not require any renaming since $Meth(\Sigma_1) - Meth(\Sigma_x)$ are already fresh names. Thus, we can replace the assumption $x : \mathsf{class}\langle \Sigma_1 \rangle$ to $C : \mathsf{class}\langle \Sigma_1 \rangle$ in the type derivation of $\Gamma \vdash exp[C/x] : \tau'$. □

Lemma 6. *If* $\Gamma, X : \mathsf{proc} \vdash P : \mathsf{proc}$ *and* $\Gamma \vdash Q : \mathsf{proc}$, *then* $\Gamma \vdash P[Q/X] : \mathsf{proc}$.

Proof. Trivial, by an inspection of typing rules for processes (Table 6). □

Theorem 1 (Substitution Property). *Let P be a process and let A be either a closed* SOOL *value or a process. If*

$$\Gamma, id:\tau \vdash P:\mathsf{proc} \qquad \textit{and} \qquad \Gamma \vdash A:\tau_1,$$

for some Γ and $\tau_1 \preccurlyeq \tau$, then $\Gamma \vdash P[A/id]:\mathsf{proc}$.

Proof. We distinguish two cases:

Case (a) A is a process. By Lemma 6;

Case (b) A is an object-oriented value. By structural induction on P which coincides with the induction on the deduction of $\Gamma, id:\tau \vdash P:\mathsf{proc}$ according to the rules of Table 6.
Base. The only cases are $P \equiv \mathbf{nil}$ and $P \equiv X$, thus $P[A/id] \equiv P$.
Inductive Step.

(a) $P \equiv \mathsf{send}(A', \ell).P'$. The only interesting case is when id occurs in A', that is $A' \equiv id$, since A' is a value. By rule (*send*), we have to prove that (i) $A \equiv A'$ is well typed and (ii) $\Gamma \vdash P'[A/id]:\mathsf{proc}$. We obtain (i) by hypothesis and (ii) by induction hypothesis.

(b) $P \equiv \mathsf{receive}(id':\tau').P'$. We take into account that receive is a binder for id', thus, if $id' \equiv id$, then $(\mathsf{receive}(id':\tau').P')[A/id] \stackrel{def}{=} \mathsf{receive}(id':\tau').P'$; otherwise, if $id' \neq id$, then $(\mathsf{receive}(id':\tau').P')[A/id] \stackrel{def}{=} \mathsf{receive}(id':\tau').(P'[A/id])$ and the proof follows from the induction hypothesis that $P'[A/id]$ is well-typed.

(c) $P \equiv P_1|P_2$. By induction hypothesis, $\Gamma \vdash P_1[A/id]:\mathsf{proc}$ and $\Gamma \vdash P_2[A/id]:\mathsf{proc}$, therefore $\Gamma \vdash (P_1|P_2)[A/id] \stackrel{def}{=} (P_1[A/id])|(P_2[A/id]):\mathsf{proc}$.

(d) $P \equiv \mathsf{def}\ x = exp\ \mathsf{in}\ P'$. We distinguish three subcases according to the definition of parametric substitution (Definition 3).

i. $exp \equiv y$ where $y \not\equiv id$: then $P[A/id] \stackrel{def}{=} P'[A/id]$ and $\Gamma \vdash P'[A/id]:\mathsf{proc}$ by induction hypothesis.

ii. $exp \equiv id$: then $(\mathsf{def}\ x = id\ \mathsf{in}\ P')[A/id] \stackrel{def}{=} (P'\{x \leftarrow id\})[A/id]$ and the proof follows from the induction hypothesis.

iii. exp is not a variable: the interesting case is when exp is not a closed value (otherwise id cannot occur in exp) and id occurs in exp, that is

$$(\mathsf{def}\ x = exp\ \mathsf{in}\ P)[A/id] \stackrel{def}{=} \mathsf{def}\ x = exp[A/id]\ \mathsf{in}\ P'[A/id].$$

By rule (*def*), $\Gamma, id:\tau \vdash \mathsf{def}\ x = exp\ \mathsf{in}\ P':\mathsf{proc}$ implies (i) $\Gamma, id:\tau \vdash exp:\tau'$, for some τ', and (ii) $\Gamma, id:\tau, x:\tau' \vdash P':\mathsf{proc}$. By Lemma 5 on (i) we have $\Gamma \vdash exp[A/id]:\tau'_1$, where $\tau'_1 \preccurlyeq \tau'$ and $Meth(\tau'_1) - Meth(\Sigma) = \emptyset$ for any Σ such that $Meth(\tau') - Meth(\Sigma) = \emptyset$. Moreover, τ' (and thus also τ'_1) can only be class or record type (they cannot be a mixin type, see Property 2). Let us consider the deduction of $\Gamma, id:\tau, x:\tau' \vdash P':\mathsf{proc}$ in the case when τ' is a class type. By induction hypothesis, there is a deduction $\mathcal{D}$ of $\Gamma, x:\tau' \vdash P'[A/id]:\mathsf{proc}$. Moreover, for any expression exp' occurring in P', such that $\Gamma, x:\tau' \vdash exp':\tau''$ is used in $\mathcal{D}$, by Corollary 1, we have also a deduction $\Gamma, x:\tau'_1 \vdash exp':\tau''_1$ for some $\tau''_1 \preccurlyeq \tau''$ such that $Meth(\tau''_1) - Meth(\tau'')$ have fresh names. Thus, it is easy to verify that we can replace such deduction of $\Gamma, x:\tau'_1 \vdash exp':\tau''_1$ to each $\Gamma, x:\tau' \vdash exp':\tau''$ in $\mathcal{D}$, and we obtain a deduction $\mathcal{D}^*$ of $\Gamma, x:\tau'_1 \vdash P'[A/id]:\mathsf{proc}$. Finally, from $\Gamma \vdash exp[A/id]:\tau'_1$ and $\Gamma, x:\tau'_1 \vdash P'[A/id]:\mathsf{proc}$ we obtain $\Gamma \vdash \mathsf{def}\ x = exp[A/id]\ \mathsf{in}\ P'[A/id]:\mathsf{proc}$ by using the rule (*def*).

□

7 Operational Semantics

The operational semantics of MOMI is based on two sets of rules. The first one describes how SOOL expressions reduce to values. This reduction relation, denoted by $\twoheadrightarrow$, is the reflexive and transitive closure of the relation $\rightarrow$, defined in Table 8 and it would be part of the operational semantics of any concrete language underlying SOOL.

The only interesting rule concerns mixin application, that produces a new class containing all the methods defined and redefined by the mixin, and those defined by the superclass. The function *override* takes care of introducing in the new class the overridden methods from the superclass, and of binding the *next* in the mixin's redefining methods' bodies to their "old" implementations: such "old" methods are given a fresh name, denoted by $m_{i'}$. Dynamic binding is then implemented for redefining methods, and old implementations of the base class are hidden in the derived class, since they are given a fresh name (i.e., they are *private* to the class).

Definition 5. *Let ρ_1 and ρ_2 be two method sets, such that for each $m_i : \tau_{m_i} = f_i$ belonging to ρ_1, $m_i : \tau_{m_i} = f_i$ belongs to ρ_2. The result of override(ρ_1, ρ_2) is the method set ρ_3 defined as follows:*

- *for all $m_i : \tau_{m_i} = f_i \in \rho_1$, let $m_i : \tau_{m_i} = f'_i \in \rho_2$ and let $m_{i'}$ be a fresh method name, then $m_{i'} : \tau_{m_i} = f'_i \in \rho_3$ and $m_i : \tau_{m_i} = f_i[m_{i'}/next] \in \rho_3$;*
- *for all $m_i : \tau_{m_i} = f_i \in \rho_2$ such that $m_i \neq m_j$ for all $m_j : \tau_{m_j} = f_j \in \rho_1$, then $m_i : \tau_{m_i} = f_i \in \rho_3$.*

$$\frac{exp \rightarrow \{m_i : \tau_{m_i} = f_i \ ^{i \in I}\}}{exp \Leftarrow m_j \rightarrow f_j}$$

$$\frac{exp \rightarrow \mathsf{class}\ [m_i : \tau_{m_i} = f_i \ ^{i \in I}]\ \mathsf{end}}{\mathsf{new}\ exp \rightarrow \{m_i : \tau_{m_i} = f_i \ ^{i \in I}\}}$$

$$\frac{exp \rightarrow \mathsf{class}\ [m_l : \tau_{m_l} = f_l \ ^{l \in L}]\ \mathsf{end}}{\left(\begin{array}{l}\mathsf{mixin}\\ \ \mathsf{expect}[m_i : \tau_{m_i} \ ^{i \in I}]\\ \ \mathsf{redef}[m_k : \tau_{m_k}\ \mathsf{with}\ f_k \ ^{k \in K}]\\ \ \mathsf{def}[m_j : \tau_{m_j} = f_j \ ^{j \in J}]\\ \mathsf{end}\end{array}\right) \diamond exp \rightarrow \left(\begin{array}{l}\mathsf{class}\\ \ [m_j : \tau_{m_j} = f_j \ ^{j \in J}] \cup\\ \ override([m_k : \tau_{m_k} = f_k \ ^{k \in K}], [m_l : \tau_{m_l} = f_l \ ^{l \in L}])\\ \mathsf{end}\end{array}\right)}$$

Table 8: Operational semantics of object-oriented expressions.

The second set of rules (presented in Table 10) describes the evolution of a MOMI net, showing how distributed processes communicate and exchange data and code by means of send and receive. It is based on a standard structural congruence $\equiv$ (defined as the least congruence relation closed under the rules in Table 9) that allows the rearrangement of the syntactic structure of a term, so that reduction rules may be applied.

$$\begin{array}{rcl} N_1 \parallel N_2 & = & N_2 \parallel N_1 \\ (N_1 \parallel N_2) \parallel N_3 & = & N_1 \parallel (N_2 \parallel N_3) \\ \ell :: \overline{P} & = & \ell :: \overline{P} \mid \mathbf{nil} \\ \ell :: (\overline{P_1} \mid \overline{P_2}) & = & \ell :: \overline{P_1} \parallel \ell :: \overline{P_2} \end{array}$$

Table 9: Congruence laws

$$\frac{match(\tau_1,\tau_2)}{\ell_1 :: \mathsf{send}(\overline{A}^{\tau_1},\ell_2).\overline{P'} \parallel \ell_2 :: \mathsf{receive}(id^{\tau_2}).\overline{Q} \rightarrowtail \ell_1 :: \overline{P'} \parallel \ell_2 :: \overline{Q}[\overline{A}^{\tau_1}/id]}\ (comm)$$

$$\frac{exp \twoheadrightarrow v}{\ell :: \mathsf{def}\ x = exp\ \mathsf{in}\ \overline{P} \rightarrowtail \ell :: \overline{P}[v/x]}\ (def)$$

$$\frac{N_1 \rightarrowtail N_1'}{N_1 \parallel N \rightarrowtail N_1' \parallel N}\ (par) \qquad \frac{N \equiv N_1 \quad N_1 \rightarrowtail N_2 \quad N_2 \equiv N'}{N \rightarrowtail N'}\ (net)$$

Table 10: Net and process operational semantics

Reduction rules for MOMI nets are displayed in Table 10. We recall that the key idea of MOMI is that, except for the dynamic checking required during the communication, the type analysis of processes is totally static and performed in each site independently. No further re-compilation and type-checking must be performed after a communication successfully took place.

To this aim, the semantics is defined on *compiled processes* $\overline{P}$, i.e., processes statically decorated (by the compiler) as follows:

– any send's argument is decorated with its type that is derived by the type analysis algorithm (see (*send*) rule in Table 6);
– any receive's argument is decorated with its type explicitly given by the programmer.

Definition 6. *The* compiled version *of a process P, denoted by* $\overline{P}$*, is defined as follows:*

– $\overline{\mathbf{nil}} \stackrel{def}{=} \mathbf{nil}$
– $\overline{\mathsf{send}(A,\ell).P'} \stackrel{def}{=} \mathsf{send}(\overline{A}^{\tau},\ell).\overline{P'}$
  where $\overline{A} \stackrel{def}{=} A$ *and* τ *is the type assigned to A by the static type system.*
– $\overline{\mathsf{receive}(id:\tau).P'} \stackrel{def}{=} \mathsf{receive}(id^{\tau}).\overline{P'}$
– $\overline{P_1 \mid P_2} \stackrel{def}{=} \overline{P_1} \mid \overline{P_2}$
– $\overline{\mathsf{def}\ x = exp\ \mathsf{in}\ P'} \stackrel{def}{=} \mathsf{def}\ x = exp\ \mathsf{in}\ \overline{P'}$

The crucial point is the dynamic matching of types. The operations send and receive synchronize only if the type of the delivered expression *matches* the one expected according to the following matching predicate:

$$match(\tau_1,\tau_2) = \begin{cases} \tau_1 \sqsubseteq \tau_2 & \text{if } \tau_1 \text{ and } \tau_2 \text{ are mixin or class types} \\ \tau_1 <: \tau_2 & \text{otherwise} \end{cases}$$

The type τ_1 of the argument A of send has been statically built and checked during the compilation. The (*comm*) rule uses this type information, delivered together with the argument A, in order to check dynamically that the received item is correct w.r.t. the formal argument of type τ_2. In particular, since A and id are decorated with their types in the compiled process, we use the parametric substitution with refresh (Definitions 3 and 4); namely, if $\tau_1 \equiv \mathsf{class}\langle\Sigma_1\rangle$ for some Σ_1 and $\tau_2 \equiv \mathsf{class}\langle\Sigma_2\rangle$ for some Σ_2, then $\overline{Q}[\overline{A}^{\tau_1}/id] \stackrel{def}{=} \overline{Q}[\overline{A}^{\tau_1}/id]_{Meth(\Sigma_1)-Meth(\Sigma_2)}$; if $\tau_1 \equiv \mathsf{mixin}\langle\Sigma^1_{new},\ldots\rangle$ and $\tau_2 \equiv \mathsf{mixin}\langle\Sigma^2_{new},\ldots\rangle$ then $\overline{Q}[\overline{A}^{\tau_1}/id] \stackrel{def}{=} \overline{Q}[\overline{A}^{\tau_1}/id]_{Meth(\Sigma^1_{new})-Meth(\Sigma^2_{new})}$.

Rule (*def*) says that *exp* is evaluated and the resulting value is replaced for x in P. The other rules are straightforward.

Preservation of types during evaluation is assumed for any concrete (sequential) language underlying the kernel SOOL, since it is standard for object-oriented calculi and languages. In particular, this property holds for SOOL expressions.

Property 4 (Preservation of types for SOOL) *If* $exp \twoheadrightarrow v$ *and* $\Gamma \vdash exp : \tau$, *then* $\Gamma \vdash v : \tau$.

This property would be proved for any concrete SOOL language by structural induction on the definition of the reduction relation. Notice that preservation of types holds for the mixin application evaluation rule (last rule in Table 8) because the original implementation of the overriding methods taken from the superclass and renamed (which are included in the reduct for technical purposes, see Definition 5) are private, therefore their types do not need to appear in the class type.

Analogously, a *progress* theorem is assumed as a standard property of any concrete language including SOOL expressions (i.e., for any well-typed expression, either it is a value or it can take a step according to the evaluation rules). *Type safety* is commonly known as the conjunction of the progress and the type preservation properties. However, in our mobile framework, the crucial issue we are interested in is concerned with a *global type safety* property rather than a local one. Roughly speaking, we aim at guaranteeing that a net preserves well-typedness when processes and object-oriented code move among different nodes during the net evolution. To this aim, we need to formalize the notion of *well-typedness* w.r.t. a net.

Definition 7. *A net* N *is* well typed *iff for any node* $\ell :: \overline{P}$ *in* N, $\Gamma \vdash P : \mathsf{proc}$ *for some* Γ.

Then we state the *subject reduction* property for nets:

Theorem 2 (Subject Reduction). *If* N *is well typed and* $N \rightarrowtail N'$, *then* N' *is well typed.*

Proof. By induction on $\rightarrowtail$. First observe that if $\Gamma \vdash a.P' : \mathsf{proc}$, then $\Gamma \vdash P' : \mathsf{proc}$ (see rules in Table 6).

- (*comm*). The interesting part concerns the receiver process on ℓ_2, $\mathsf{receive}(id^{\tau_2}).\overline{Q}$; by hypothesis we have $\Gamma \vdash \mathsf{receive}(id : \tau_2).Q : \mathsf{proc}$, which implies that $\Gamma, id : \tau_2 \vdash Q : \mathsf{proc}$ (by rule (*receive*) in Table 6). Thus we use the substitution property (Theorem 1) and therefore obtain $\Gamma \vdash Q[A/id] : \mathsf{proc}$.
- (*def*). Consider $\mathsf{def}\ x = exp\ \mathsf{in}\ \overline{P}$; by hypothesis $\Gamma \vdash \mathsf{def}\ x = exp\ \mathsf{in}\ P : \mathsf{proc}$, thus there exists τ' such that $\Gamma \vdash exp : \tau'$ and $\Gamma, x : \tau' \vdash P : \mathsf{proc}$. Since $exp \twoheadrightarrow v$, by Property 4 we have also $\Gamma \vdash v : \tau'$. Then, we can replace v to all the free occurrences of x in P, therefore obtaining a well-typed process. Namely, a deduction of $\Gamma \vdash P[v/x] : \mathsf{proc}$ is obtained by replacing $\Gamma \vdash v : \tau'$ to the assumptions $x : \tau'$ in the deduction of $\Gamma, x : \tau' \vdash P : \mathsf{proc}$ (as a special case of Theorem 1, where no type information is needed during substitution since the two types coincide).
- (*par*). By induction hypothesis.
- (*net*). It is easy to verify that structural congruence (Table 9) preserves well-typedness.

□

The above theorem, together with Property 4, leads to a global safety property, saying that merging code received from a remote site into local code does not harm local type safety (local evaluation of well-typed object-oriented expressions cannot produce errors like "message not understood").

Finally, we observe that the dynamic checking during communication is the only dynamic use of types; it essentially consists in checking subtyping between record, class or mixin types, which is of linear complexity on the argument types.

We conclude this section with an example that shows how dynamic name clashes are dealt with, even in the presence of an *aliasing* introduced by def. Let us consider this process:

$$\mathsf{receive}(id^{\tau}).\mathsf{def}\ x = id\ \mathsf{in}\ \mathsf{def}\ y = x\ \mathsf{in}\ x \diamond y \diamond C$$

where $\tau \equiv \mathsf{mixin}\langle \emptyset, \ldots \rangle$ and C is a class value that provides everything requested by the mixin. This process is well-typed. Let us now suppose that we receive a (matching) mixin value M with (sub)type $\tau' \equiv \mathsf{mixin}\langle \{m : \tau_m\}, \ldots \rangle$. By applying the (*comm*) rule we have:

$$(\mathsf{def}\ x = id\ \mathsf{in}\ \mathsf{def}\ y = x\ \mathsf{in}\ x \diamond y \diamond C)[M/id]_{\{m\}}$$

Now, by applying the definition of parametric substitution (Definition 3) we obtain:

$$\begin{array}{l}(\mathsf{def}\ x = id\ \mathsf{in}\ \mathsf{def}\ y = x\ \mathsf{in}\ x \diamond y \diamond C)[M/id]_{\{m\}} \stackrel{def}{=} \\ ((\mathsf{def}\ y = x\ \mathsf{in}\ x \diamond y \diamond C)\{x \leftarrow id\})[M/id]_{\{m\}} \stackrel{def}{=} \\ (\mathsf{def}\ y = id\ \mathsf{in}\ id \diamond y \diamond C)[M/id]_{\{m\}} \stackrel{def}{=} \\ ((id \diamond y \diamond C)\{y \leftarrow id\})[M/id]_{\{m\}} \stackrel{def}{=} \\ (id \diamond id \diamond C)[M/id]_{\{m\}} \stackrel{def}{=} \\ id[M/id]_{\{m\}} \diamond id[M/id]_{\{m\}} \diamond C.\end{array}$$

As explained in Section 5.2, a different refreshed version is substituted for each occurrence of id, thus we obtain:

$$M' \diamond M'' \diamond C$$

where M' is M where m is renamed with the globally fresh name m' and M'' is M where m is renamed with the globally fresh name m''. Thus, the resulting mixin application is still well-typed since no name clash arises.

8 A scenario for mixin mobility

We will use here a slightly simplified syntax: (*i*) we will list the methods' parameters in between "()"; (*ii*) $exp_1; P$ is interpreted as $\mathsf{def}\ x = exp_1\ \mathsf{in}\ P$, $x \notin FV(P)$, with a call-by-value semantics; also the let construct $\mathsf{let}\ x = exp_1\ \mathsf{in}\ exp_2$ is interpreted as usual in a call-by-value way; (*iii*) even though our calculus uses structural subtyping (and not nominal), in this section we will refer to expressions related via "$\sqsubseteq$" with names as shortcuts[4].

The example is about a client and a server, executing on two different nodes, that want to communicate, e.g., by means of a common protocol. They both use a *Socket* to this aim, however the server is willing to abstract from the implementation of such a socket, by allowing the client to provide a custom implementation. This can be useful, for instance, because the client may decide to use a customized socket; in this example the client implements a socket that sends and receives compressed data (alternatively it could implement a *multicast* socket, or even a combination of the two). However, the code sent by the client may rely on some low-level system calls, that may be different on the server's site: indeed, the two sites may run different operating systems and have different architectures. These low-level system calls are then to be provided by each site (the client's and the server's sites). The customized socket of the client is then a mixin requiring the existence of such system calls, that will be provided by two different (yet compliant) superclasses, one resident on each site. The code executed in the two nodes (`client` and `server`) is in Listing 1.

Both `ZipSocket` and `Socket` rely on a superclass that provides (at least) methods `write_to_net` and `read_from_net`. The client, in its site, completes its mixin `ZipSocket` with

[4] We refer to [44] for a wider discussion on the advantages and drawbacks of structural and nominal subtyping, in theory and in language implementations.

```
client:: let ZipSocket =
  mixin
    def zip = ...
    def unzip = ...
    def write = write_to_net(zip(data))
    def read  = unzip(read_from_net())
    expect write_to_net
    expect read_from_net
  end in
let NetChannel =
  class
    send_data =
      // <send through the net>
    receive_data =
      // <receive from the net>
    write_to_net = send_data(data)
    read_from_net = receive_data()
  end in
let channel = new
    (ZipSocket ◇ NetChannel) in
  (
    send( ZipSocket, server ).
    ( channel⇐write("hello") ;
      channel⇐read() )
  )
```

```
server:: let Socket =
  mixin
    def write = write_to_net(data)
    def read  = read_from_net()
    expect write_to_net
    expect read_from_net
  end in
let NetFile =
  class
    write_to_net =
      // <send through the net>
    read_from_net =
      // <receive from the net>
  end in
(
  receive( sock : Socket ).
  let client_channel = new
      (sock ◇ NetFile) in
  (
    client_channel⇐read() ;
    client_channel⇐write("welcome")
  )
)
```

Listing 1: Example code for client and server communication.

`NetChannel` that provides these two methods for writing data on the net, by using its operating system low-level system calls. Sending the class `ZipSocket` ◇ `NetChannel` directly to the server may be nonsense, since the server may use a different operating system (or a different version of the same operating system). Instead, only the mixin `ZipSocket` is sent to the remote server. In the server this mixin will be received as a `Socket` mixin (and this succeeds since `ZipSocket` $\sqsubseteq$ `Socket`) and it will be completed with `NetFile`, which corresponds to the `NetChannel` of the client. The server will then use such socket independently from the particular client's implementation. Notice that the use of subtyping in the communication (instead of a simpler type equality) completely relieves the receiver (and especially its programmer) from the real complete interface of the clients' code.

Let us now consider an alternative implementation of the same scenario, in order to show other features of MOMI: suppose that on the client's side `ZipSocket` is written like in Listing 2 on the left. In this case the class does not rely on `write_to_net` and `read_from_net` (instead it expects the superclass to provide methods `write` and `read` that the mixin redefines), and thus it is not a subtype of `Socket` in the server. In the server, the code would be like in Listing 2 on the right. Since also `Channel` relies on a super class that provides `write` and `read`, we have that `ZipSocket` $\sqsubseteq$ `Channel`. So the server receives a `ZipSocket` (as a `Channel`) that it completes with `Socket` completed, in turn, with `NetFile`.

Finally, as hinted in Section 3.1, other implementations of such a socket can be created, simply by using more than one mixin, such as `UUEncode`, `Encrypt`, and so on.

```
ZipSocket =
 mixin
  def zip = ...
  def unzip = ...
  redef write = next(zip(data))
  redef read  = unzip(next())
 end
```

```
Channel =
 mixin
  redef write = next(data)
  redef read  = next()
 end
 ...
 receive( chan : Channel ).
 let client_channel =
  new (chan ◇ (Socket ◇ NetFile)) in
 (
  client_channel⇐read() ;
  client_channel⇐write("welcome")
 )
```

Listing 2: An alternative implementation.

9 A case study: O'KLAIM (Object-Oriented KLAIM)

In order to test the applicability of the MOMI approach, in [13] we presented O'KLAIM, a mixin-oriented version of KLAIM [24,9]. Here we want to summarize the main steps of the extension that led to O'KLAIM, since they can be seen as a general methodology for applying the MOMI approach. This section is completed with a brief description of the implementation of O'KLAIM, freely available at `http://music.dsi.unifi.it`.

O'KLAIM is also interesting because it highlights that the design motivation of keeping the MOMI object-oriented calculus and the MOMI processes separated is not a limitation, but, on the contrary, it gives extra flexibility. We recall that MOMI consists of three components: the "surface" object-oriented component, a mixin/class subtype relation, and a coordination calculus. If the object-oriented component is chosen to be an object-oriented concurrent/mobile language, the two components (object-oriented and concurrent/mobile) may interleave in a deeper way. A good example is the implementation of O'KLAIM (Section 9.3): the "surface" object-oriented calculus and the coordination language are melt together, so that method bodies can perform KLAIM actions. The matching mechanism that allows safe interactions during code exchange is based on the MOMI's subtype relation, that acts as a general glue to glue together the two language components, of whichever nature they are (as long as the object-oriented one implements classes and mixins).

9.1 The basics of KLAIM

KLAIM (*Kernel Language for Agent Interaction and Mobility*) [24,9] is a coordination language inspired by the Linda model [33], hence it relies on the concept of *tuple space*. A tuple space is a multiset of *tuples*; these are sequences of information items (called *fields*). There are two kinds of fields: *actual fields* (i.e., expressions, processes, localities, constants, identifiers) and *formal fields* (i.e., variables). Syntactically, a formal field is denoted with !*ide*, where *ide* is an identifier. Tuples are anonymous and content-addressable; *pattern-matching* is used to select tuples in a tuple space:

- two tuples match if they have the same number of fields and corresponding fields have matching values or formals;
- formal fields match any value of the same type, but two formals never match, and two actual fields match only if they are identical.

P	$::=$	**nil**	(null process)	N	$::=$	$l :: p$	(single node)
	$\mid$	$act.P$	(action prefixing)		$\mid$	$N_1 \parallel N_2$	(net composition)
	$\mid$	$P_1 \mid P_2$	(parallel composition)	p	$::=$	$P \mid \langle t \rangle \mid p_1 \mid p_2$	(located item)
	$\mid$	X	(process variable)				
	$\mid$	def $x = exp$ in P	(OO expression)				

$act ::= \mathbf{out}(t)@\ell \mid \mathbf{in}(t)@\ell \mid \mathbf{read}(t)@\ell \mid \mathbf{eval}(P)@\ell \qquad \ell ::= l \mid \chi \qquad t ::= f \mid f,t$

$f ::= arg \mid !id:\sigma \qquad id ::= x \mid X \mid \chi \qquad arg ::= id \mid e \mid P \mid l \mid v$

Table 11: O'KLAIM syntax (see Table 1 for the syntax of *exp* and *v*, and Section 4 for types σ).

For instance, tuple (`"foo"`, `"bar"`, 300) matches with (`"foo"`, `"bar"`, !*val*). After matching, the variable of a formal field gets the value of the matched field: in the previous example, after matching, *val* (an integer variable) will contain the value 300.

Tuple spaces are placed on *nodes* (or *sites*), which are part of a *net*. Each node contains a single tuple space and processes in execution, and can be accessed through its *locality*. KLAIM processes may run concurrently, both at the same node or at different nodes, and can perform four basic operations over nodes. The $\mathbf{in}(t)@\ell$ operation looks for tuple t' that matches with t in the tuple space located at ℓ. Whenever the matching tuple t' is found, it is removed from the tuple space. The corresponding values of t' are then assigned to the formal fields of t and the operation terminates. If no matching tuple is found, the operation is suspended until one is available. The $\mathbf{read}(t)@\ell$ operation differs from $\mathbf{in}(t)@\ell$ only because the tuple t', selected by pattern-matching, is not removed from the tuple space located at ℓ. The $\mathbf{out}(t)@\ell$ operation adds the tuple t to the tuple space located at ℓ. The $\mathbf{eval}(P)@\ell$ operation spawns process P for execution at node ℓ.

KLAIM is higher-order in that processes can be exchanged as primary-class data. While $\mathbf{eval}(P)@\ell$ spawns a process for (remote) evaluation at ℓ, processes sent with an **out** must be retrieved explicitly at the destination site. The receiver can then execute the received process locally, as in the following process: $\mathbf{in}(!X)@\texttt{self}.\mathbf{eval}(X)@\texttt{self}$.

9.2 O'KLAIM*: processes and nets*

O'KLAIM syntax is defined in Table 11[5]. In order to obtain O'KLAIM, we extend the KLAIM syntax of tuples t to include any object-oriented value v (defined in Table 1). In particular, differently from KLAIM, formal fields are now explicitly typed. Actions $\mathbf{in}(t)@\ell$ (and $\mathbf{read}(t)@\ell$) and $\mathbf{out}(t)@\ell$ can be used to move object-oriented code (together with the other KLAIM items) from/to a locality ℓ, respectively. Moreover, we add to KLAIM processes the construct def $x = exp$ in P in order to pass to the sub-process P the result of computing an object-oriented expression *exp* (for the *exp* syntax see Table 1). We use the following syntactic convention: x, X and χ are variables representing object-oriented values, processes and localities, respectively. A constant locality (e.g., IP:port) is denoted by l. Moreover, e is a basic expression (i.e., not object-oriented).

A *Net* is a finite collection of *nodes*. A node is a pair where the first component is a (constant) locality and the second component is either a process P or a tuple $\langle t \rangle$ or a composition of processes and tuples. Thus, a tuple space is represented by the parallel composition of located tuples. Notice

[5] The distinction between logical and physical locality (and thus the concept of "allocation environment"), and the creation of new nodes, which are in the original KLAIM definition, are not relevant in the O'KLAIM context, thus, for the sake of simplicity, we omit them in the present formal presentation. Notice, however, that their integration, being them orthogonal with respect to the object-oriented features, is completely smooth.

$$\frac{}{\Gamma, id:\tau \vdash id:\tau}\ (proj) \qquad \frac{}{\Gamma \vdash l:\mathsf{loc}}\ (loc) \qquad \frac{}{\Gamma \vdash \mathbf{nil}:\mathsf{proc}}\ (nil)$$

$$a \equiv \mathbf{in}, \mathbf{read}, \mathbf{out} \qquad \frac{\begin{array}{l}\Gamma \vdash \ell:\mathsf{loc} \\ \Gamma \vdash f_i:\tau_i \qquad i=1,\ldots,n \ \wedge\ f_i \equiv arg \\ \Gamma \cup ftypes(f_1,\ldots,f_n) \vdash P:\mathsf{proc}\end{array}}{\Gamma \vdash a(f_1,\ldots,f_n)@\ell.P:\mathsf{proc}}\ (action)$$

$$ftypes(f,t) = \begin{cases} \{id:\tau\} \cup ftypes(t) & \text{if } f \equiv !id:\tau \\ ftypes(t) & \text{otherwise} \end{cases}$$

$$\frac{\Gamma \vdash Q:\mathsf{proc} \qquad \Gamma \vdash \ell:\mathsf{loc}}{\Gamma \vdash \mathbf{eval}(Q)@\ell.P:\mathsf{proc}}\ (eval)$$

$$\frac{\Gamma \vdash P_1:\mathsf{proc} \qquad \Gamma \vdash P_2:\mathsf{proc}}{\Gamma \vdash (P_1 \mid P_2):\mathsf{proc}}\ (comp) \qquad \frac{\Gamma \vdash exp:\tau \qquad \Gamma, x:\tau \vdash P:\mathsf{proc}}{\Gamma \vdash \mathsf{def}\ x = exp\ \mathsf{in}\ P:\mathsf{proc}}\ (let)$$

Table 12: Typing rules for processes

$$\frac{}{l :: \mathbf{out}(t)@l'.P \parallel l' :: P' \rightarrowtail l :: P \parallel l' :: P' \mid \langle t \rangle}\ (\textsc{out})$$

$$\frac{match(t,t')}{l :: \mathbf{in}(t)@l'.P \parallel l' :: \langle t' \rangle \rightarrowtail l :: P[t'/t] \parallel l' :: \mathbf{nil}}\ (\textsc{in})$$

$$\frac{match(t,t')}{l :: \mathbf{read}(t)@l'.P \parallel l' :: \langle t' \rangle \rightarrowtail l :: P[t'/t] \parallel l' :: \langle t' \rangle}\ (\textsc{read})$$

$$\frac{}{l :: \mathbf{eval}(P)@l'.P' \parallel l' :: P'' \rightarrowtail l :: P' \parallel l' :: P' \mid P}\ (\textsc{eval})$$

$$\frac{exp \twoheadrightarrow v}{l :: \mathsf{def}\ x = exp\ \mathsf{in}\ P \rightarrowtail l :: P[v/x]}\ (\textsc{let}) \qquad \frac{N \equiv N_1 \qquad N_1 \rightarrowtail N_2 \qquad N_2 \equiv N'}{N \rightarrowtail N'}\ (\textsc{net})$$

Table 13: O'KLAIM operational rules

that programmers write only located processes, while located tuples are produced at run-time by evaluating **out** actions (see Table 13).

In order to type processes and nets, we extend the set of types $\mathcal{T}^*$ of Section 4.2 with the type constant loc, used to type localities. Typing rules for O'KLAIM processes are defined in Table 12. Following the MOMI requirement, every tuple item t_i (which may be an object-oriented value) that takes part in the information exchange must be decorated with its type information, denoted by ${t_i}^{\tau_i}$. The types of the tuples are built statically by the compiler, while the types of tuple formal fields must be written explicitly by the programmer. In a process of the form $\mathbf{in}(!\,id:\tau)@\ell.P$, the type τ is used to statically type check the continuation P, where *id* is possibly used. More generally, concerning (*action*) rule, in a process performing an operation with a tuple (i.e., **out**, **read** and **in**), the actual fields of the tuple are type checked, and the types of formal fields (collected by the function *ftypes*) are used to type check the continuation. Thus, **in** and **read** play the role of MOMI's receive and **out** plays the role of MOMI's send. Notice that, instead of the synchronous communication mechanism of MOMI, we use in O'KLAIM the original KLAIM asynchronous (tuple space-based) communication.

$$match(e,e) \qquad match(l,l) \qquad \frac{match(t_2,t_1)}{match(t_1,t_2)}$$

$$\frac{match(t_1,t_2) \quad match(t_3,t_4)}{match((t_1,t_3),(t_2,t_4))} \qquad \frac{match(\tau,\tau_i)}{match(!id:\tau,t_i{}^{\tau_i})}$$

$$match(\tau_1,\tau_2) = \begin{cases} \tau_1 \sqsubseteq \tau_2 & \text{if } \tau_1 \text{ and } \tau_2 \text{ are mixin or class types} \\ \tau_1 <: \tau_2 & \text{otherwise} \end{cases}$$

Table 14: Matching rules (with proc $<:$ proc and loc $<:$ loc)

The operational semantics of O'KLAIM involves two sets of rules. The first set of rules describes how object-oriented expressions reduce to values (the corresponding reduction relation is denoted by $\twoheadrightarrow$), and they are inherited from MOMI (Table 8). The second set of rules for O'KLAIM, shown in Table 13, concerns processes and it is an extension of the operational semantics of KLAIM. Notice that the O'KLAIM's operational semantics must be defined on typed *compiled* processes, i.e., processes where each object-oriented value and tuples are decorated with their types, because the crucial point is the dynamic matching of types. In fact, an **out** operation adds a tuple decorated with a (static) type to a tuple space, and a process can perform an **in** action by synchronizing with a process which represents a matching typed tuple. The rule for def $x = exp$ in P relies on the reduction relation for object-oriented expressions $\twoheadrightarrow$.

The predicate for tuples, *match*, is presented in Table 14. The matching rules exploit the static type information, delivered together with the tuple items, in order to check dynamically that the received item is correct with respect to the type of the formal field, say τ. Therefore, an item is accepted if and only if it is subtyping-compliant with the expected type of the formal field. Thus, the original pattern mechanism of KLAIM, which was based on type equality for formal fields, is extended in O'KLAIM in order to use the MOMI subtype relation.

Finally, the semantics for the distributed part is based on structural congruence and reduction relations (just like in MOMI, Table 9). As a final remark, let us observe that we do not define a matching predicate for actual fields containing object-oriented values and processes since this would require to decide equalities on classes, mixins and objects (e.g., equality on their interfaces) and on processes (e.g., a bisimulation). This issue is out of the scope of the present work, since matching between two actual fields does not involve any substitution and then does not cause problems with respect to typing.

9.3 The implementation

The original implementation of KLAIM consisted of a Java package, KLAVA [16], implementing the run-time system for KLAIM operations and communications among distributed nodes, and the programming language X-KLAIM [15], which extends KLAIM with high-level programming constructs. A compiler translates X-KLAIM programs into Java programs that use the package KLAVA. Here we hint how we extended these two items with MOMI object-oriented features, in order to implement O'KLAIM. Let us observe that the object-oriented version of X-KLAIM is used both as the "surface" object-oriented calculus and as the coordination language, with the added bonus of being able to write methods that can perform KLAIM actions, all the same inheriting the properties of Section 5, thus guaranteeing absence of run-time type errors.

The implementation of the O'KLAIM object-oriented component in Java consists in a package `momi` presented in [7], and described in details in [8]. This package provides the run-time system,

or the virtual machine, for classes, mixins and objects that can be downloaded from the network and composed dynamically (via the mixin application operation). It thus provides functionalities for checking the subtyping among class types and among mixin types, and for building at run-time new subclasses. Since we abstract from the specific communication and mobility features, this package does not provide means for code mobility and network communication, so that `momi` can be smoothly integrated into the existing Java mobility frameworks. We would like to stress that this package should be thought of as an "assembly" language that is the target of a compiler for a high-level language (in our case the language is X-KLAIM). If `momi`, as it is, were used for writing directly object-oriented expressions, the programmer would be left with the burden of writing methods containing Java statements dealing with `momi` objects, classes and mixins, and to check manually that they are well typed. Basically these are the same difficulties a programmer must face when using an assembly language directly, instead of a high-level language. We could say that `momi` enhances the functionalities of the Java virtual machine: while the latter already provides useful mechanisms for dynamically loading new classes into a running application, the former supplies means for dynamically building class hierarchies (based on mixins) and for inserting new subclasses into existing hierarchies (which is not possible in Java).

In order to implement O'KLAIM we extended the KLAIM programming framework: the package KLAVA already provided all the primitives for network communication, through distributed tuple spaces, and, in particular, for code mobility, not supplied by `momi`. Thus the package has been modified in order to be able to exchange code that is based on `momi`, and for performing subtyping on `momi` elements during pattern matching by relying on the `MoMiType` classes and the associated subtyping. Notice that the X-KLAIM compiler generates code that uses both the KLAVA package and `momi`. Obviously, before generating code, the compiler also performs type checking according to the type system defined by MOMI. Notice that only the run-time system of MOMI can be implemented as a Java package: a compilation phase (performed by the compiler of the high-level language) is still required, since the static type checking of Java is not enough for MOMI's features (i.e., mixins and mixin applications).

The programming example shown here is based on the first introductive example of Section 3.3. The agent that wants to print is designed as a mixin, that expects a method for actually printing, `print_doc`, and defines a method `start_agent` through which the site can start its execution. The actual instance of the printing agent is instantiated from a class dynamically generated by applying such mixin to a local superclass that provides the method `print_doc` acting as a wrapper for the printer driver. However, the system is willing to accept any agent that has a compatible interface, i.e., any mixin that is a subtype of the one used for describing the printing agent. Thus any client wishing to print on this site can send a mixin that is subtyping compliant with the one expected. In particular such a mixin can implement finer printing formatting capabilities.

Listing 3, where **rec** is the X-KLAIM keyword for defining a process, presents a possible implementation of the printing client node (on the left) and of the printer server node (on the right). We refer to [8] for the complete syntax of X-KLAIM, which is quite self-explicative (we just say that following the approach of mainstream object-oriented languages we use **this** for the implicit invocation object and `.` instead of $\Leftarrow$ for method invocation). The printer client sends to the server a mixin `MyPrinterAgent` that complies with (it is a subtype of) the mixin that the server expects to receive, `PrinterAgent`. In particular, `MyPrinterAgent` mixin will print a document on the printer of the server after preprocessing it (method `preprocess`). On the server, once the mixin is received, it is applied to the local (super)class `LocalPrinter`, and an object (the agent) is instantiated from the resulting class and started, so that it can actually print its document. The result of the printing task is then retrieved and sent back to the client.

```
mixin MyPrinterAgent
 expect print_doc(doc : str) : str;
 def start_agent() : str
 begin
  return
   this.print_doc
    (this.preprocess("my document"))
 end;
 def preprocess(doc : str) : str
 begin
  return "preprocessed(" + doc +")"
 end
end

rec SendPrinterAgent[server : loc]
 declare
  var response : str
 begin
  out(MyPrinterAgent)@server;
  in(!response)@server;
  print "response is " + response
 end
```

```
mixin PrinterAgent
 expect print_doc(doc : str) : str;
 def start_agent() : str;
end

class LocalPrinter
 print_doc(doc : str) : str
 begin
  # real printing code omitted :−)
  return "printed " + doc
 end;
 init()
 begin
  nil # foo init
 end
end

rec ReceivePrinterAgent[]
 declare
  var rec_mixin : mixin PrinterAgent;
  var result : str
 begin
  in(!rec_mixin)@self;
  result :=
   (new rec_mixin <> LocalPrinter).start_agent();
  out(result)@self
 end
```

Listing 3: The printer agent example.

We observe that the sender does not actually know the mixin name `PrinterAgent`: it only has to be aware of the mixin type expected by the server. Furthermore, the mixin which is received can also define more methods than those specified in the receiving site, thanks to the mixin subtype relation. This adds flexibility to such a system, while hiding these additional methods to the receiving site (since they are not specified in the receiving interface they are actually statically unknown to the compiler). Furthermore, we stress that type names in X-KLAIM are only shortcuts for the actual type structures, thus subtyping is still structural.

10 Conclusions

In this paper we presented MOMI, that is a general framework for integrating a mobile and distributed language with some flexible and dynamic object-oriented features. The three main ingredients of MOMI are:

- the definition of an object-oriented "surface calculus" (SOOL) containing essential object-oriented features to write mixin-based code;
- the definition of a new subtype relation on class and mixin types to be exploited dynamically at communication time;
- a very primitive coordination language consisting in a synchronous send/receive mechanism, to study the communication of the mixin-based code among different sites at a basic level and to prove the main correctness properties.

There already exists an extension of MOMI: we studied how to extend the presented width subtyping to depth subtyping on record types. This extension permits achieving a more powerful mixin and class subtype relation, and therefore increases the communication flexibility in a

substantial way. However, this powerful subtyping raises some technical difficulties and requires further static analysis. We introduced this problem and its solution in [14], but we still need to analyze in some detail the meta-theory, and, in particular, the complexity of such a solution.

10.1 Related work

In the literature, there are several proposals of combining objects with processes and/or mobile agents. *Obliq* [21] is a lexically-scoped language providing distributed object-oriented computation; mobile code maintains network references and provides transparent access to remote resources. In [20], a general model for integrating object-oriented features in calculi of mobile agents is presented: agents are extended with method definitions and constructs for remote method invocations. Other works, such as, e.g., [26,45,34,31] are concerned with merging concurrency and object orientation and do not deal explicitly with mobile distributed code. In our calculus no remote method call functionality is considered, and, instead of formalizing remote procedure calls (like most of the above mentioned approaches), MOMI provides the introduction of safe and scalable distribution of object-oriented code, in a calculus where communication and coordination facilities are already provided.

Our approach is more related to papers, such as, e.g., [43,53], where properties of distributed systems are enforced by a typing system equipped with subtyping. In our case the property we enforce is a flexible and type-safe coordination for exchanging code among processes, up- and down-loading classes and mixins from different sites. Thus the crucial feature of MOMI is that classes and mixins are themselves mobile code, i.e., code that is dynamically down- and up-loaded to/from a remote source. At run time, the actual implementation of mixins and classes may differ from their expected (static) interface (obviously formal and actual parameter types must be compliant by subtyping, as described in Section 4.3). This highlights the main difference between MOMI and other mixin-based languages in the literature (see, e.g., [29,3]): in most mixin-based languages a mixin can be applied to several base classes (and this shows the "dynamic" inheritance nature of mixins) but when the application takes place in a specific part of a program, both the code of the mixin and the one of the classes are available for the compiler. This does not hold in MOMI since mixin application can also act on mixin and class variables. So the tasks of generating a new class, when a mixin is applied to a class, and of creating an object from a class (performing all the right bindings) must be done at run-time in MOMI, when the class and mixin variables have been actually instantiated with effective code (possibly downloaded from the network). This is similar to what happens in calculi (such as, e.g., [18]) where classes and mixins are "first-class citizens", i.e., they can be passed as parameters to functions. It is important to notice, though, that in such calculi the matching between actual and formal parameters is always based on equality (at least to the best of our knowledge), because the burden of checking extra constraints at parameter-passing time, to avoid inheritance/subtyping conflicts, is not worthwhile in a sequential scenario. In a mobile distributed scenario instead, where flexibility is a must because the nature of the downloaded code cannot always be estimated a priori, the use of such extra subtyping constraints at communication time is a small price to pay, especially since this allows us to combine local and foreign code without any recompilation.

In the language MIXGEN presented in [2] first class generic types are added to Java, with nominal subtyping, in such a way that mixin definitions are formulated as a particular case of generic classes. Some points relate [2] to the MOMI approach:

– mixins are given first-class generic types that are enforced through static type checking;

– subtyping on class types are used in mixin applications;
– in the context of separate compilation, the problem of accidental overrides during mixin applications arises and is solved by renaming all new methods in mixin classes.

However, differently from MOMI, no subtype relation on mixin types is exploited in [2], apart from mixin instantiations being subtypes of their superclasses.

10.2 Open issues and further development

We want to stress once again that the MOMI communication protocol is a low level one. Indeed, we make the following assumptions:

– there are no failures (in particular network connectivity is always available[6]);
– types attached to migrating code are computed correctly (i.e., it does not prevent malicious sites from sending code with a forged/fake type).

Upon this low level protocol, one can build and implement higher level protocols. With this respect we hint some directions:

– implement a protocol where the processes that want to exchange code must first agree on the types of items they intend to exchange and then rely on the MOMI matching mechanism for code exchange;
– consider a digital signature mechanism to certify types that are attached to migrating code (e.g., similarly to digitally-signed Java applets).

Furthermore, the O'KLAIM case study (Section 9) shows that switching from the synchronous communication mechanism of MOMI to an asynchronous (tuple space based) one is straightforward.

Another remark concerns the relation between inheritance and synchronization constraints, that often conflict in a concurrent object-oriented setting: their simultaneous use tends to require many method redefinitions and to break encapsulation, so that typical practical advantages of object-oriented programming are lost. In [40] this phenomenon is studied and is given a name, *inheritance anomaly*. In MOMI, we do not provide explicit means for concurrency since we think that this is an orthogonal issue and should be dealt with by the underlying concrete language. However, if direct means for synchronization would be added to MOMI, either in the language underlying SOOL or in the coordination language, we believe that some strategies could be adopted in order to avoid deadlocks and inheritance anomaly's drawbacks. We refer to [39, 6] for some techniques for avoiding these problems in concurrent object-oriented languages.

Further extensions of MOMI are topics for developments: the SOOL component can be enriched with *incomplete objects*, i.e., objects instantiated from mixins [11] and *higher-order mixins*, i.e., mixins that can be composed with mixins [10]. Another research direction could be the study of a MOMI with a SOOL instance based on nominal subtyping instead of the structural one. Moreover, the type system presented in this paper can be modified in order to refine proc types, so that finer types can be assigned to processes, e.g., according to the capability-based type system for access control developed for KLAIM [25], or to the type system presented in [36].

[6] Most calculi for distributed and mobile computations make the same assumptions and simply do not handle failures: processes that fail to perform a specific actions are simply stuck in computation. Indeed, both network and communication failures are implementation issues.

Acknowledgments We thank the anonymous referees for many helpful remarks and suggestions.

References

1. M. Abadi and L. Cardelli. *A Theory of Objects*. Springer, 1996.
2. E. Allen, J. Bannet, and R. Cartwright. A first-class approach to genericity. *ACM SIGPLAN Notices*, 38(11):96–114, 2003.
3. D. Ancona, G. Lagorio, and E. Zucca. Jam - A Smooth Extension of Java with Mixins. In *Proc. of ECOOP'00*, volume 1850 of *LNCS*, pages 145–178. Springer-Verlag, 2000.
4. D. Ancona and E. Zucca. A Theory of Mixin Modules: Algebraic Laws and Reduction Semantics. *Mathematical Structures in Computer Science*, 12(6):701–737, 2001.
5. L. Bak, G. Bracha, S. Grarup, R. Griesemer, D. Griswold, and U. Hölzle. Mixins in Strongtalk. In *Proc. of the "Inheritance Workshop" at ECOOP'02*, 2002.
6. N. Benton, L. Cardelli, and C. Fournet. Modern Concurrency Abstractions for C♯. In *Proc. of ECOOP'02*, volume 2374 of *LNCS*, pages 415–440. Springer-Verlag, 2002.
7. L. Bettini. A Java package for class and mixin mobility in a distributed setting. In *Proc. of FIDJI'03*, volume 2952 of *LNCS*, pages 12–22. Springer-Verlag, 2003.
8. L. Bettini. *Linguistic Constructs for Object-Oriented Mobile Code Programming & their Implementations*. PhD thesis, Dipartimento di Matematica, Università di Siena, 2003. Available at `http://music.dsi.unifi.it`.
9. L. Bettini, V. Bono, R. De Nicola, G. Ferrari, D. Gorla, M. Loreti, E. Moggi, R. Pugliese, E. Tuosto, and B. Venneri. The klaim project: Theory and practice. In C. Priami, editor, *Global Computing – Programming Environments, Languages, Security, and Analysis of Systems, IST/FET International Workshop GC 2003*, volume 2874 of *LNCS*. Springer-Verlag, 2003.
10. L. Bettini, V. Bono, and S. Likavec. A Core Calculus of Higher-Order Mixins and Classes. In *Post-Proc. of TYPES'03*, volume 3085 of *LNCS*. Springer-Verlag, 2004.
11. L. Bettini, V. Bono, and S. Likavec. A Core Calculus of Mixin-Based Incomplete Objects. In *FOOL 11*, 2004.
12. L. Bettini, V. Bono, and B. Venneri. Coordinating Mobile Object-Oriented Code. In *Proc. of Coordination Models and Languages (COORDINATION'02)*, volume 2315 of *LNCS*, pages 56–71. Springer-Verlag, 2002.
13. L. Bettini, V. Bono, and B. Venneri. O'KLAIM: a coordination language with mobile mixins. In *Proc. of Coordination Models and Languages (COORDINATION'04)*, volume 2949 of *LNCS*, pages 20–37. Springer-Verlag, 2004.
14. L. Bettini, V. Bono, and B. Venneri. Subtyping-Inheritance Conflicts: The Mobile Mixin Case. In *Proc. of Third IFIP International Conference on Theoretical Computer Science (TCS'04)*, pages 451–464. Kluwer Academic Publishers, 2004.
15. L. Bettini, R. De Nicola, G. Ferrari, and R. Pugliese. Interactive Mobile Agents in X-KLAIM. In *Proc. of the 7th Int. IEEE Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE'98)*, pages 110–115. IEEE Computer Society Press, 1998.
16. L. Bettini, R. De Nicola, and R. Pugliese. KLAVA: a Java package for distributed and mobile applications. *Software – Practice and Experience*, 32(14):1365–1394, 2002.
17. L. Bettini, M. Loreti, and B. Venneri. On Multiple Inheritance in Java. In *Proc. of Technology of Object-Oriented Languages, Systems and Architectures (TOOLS'02)*, pages 1–15. Kluwer Academic Publishers, 2003.
18. V. Bono, A. Patel, and V. Shmatikov. A Core Calculus of Classes and Mixins. In *Proc. of ECOOP'99*, volume 1628 of *LNCS*, pages 43–66. Springer-Verlag, 1999.
19. G. Bracha and W. Cook. Mixin-based inheritance. In *Proc. of OOPSLA '90*, pages 303–311, 1990.
20. M. Bugliesi and G. Castagna. Mobile Objects. In *FOOL 7*, 2000.
21. L. Cardelli. A Language with Distributed Scope. *Computing Systems*, 8(1):27–59, 1995.
22. L. Cardelli and P. Wegner. On Understanding Types, Data Abstraction, and Polymorphism. *ACM Computing Surveys*, 17(4):471–522, 1985.
23. A. Carzaniga, G. Picco, and G. Vigna. Designing Distributed Applications with Mobile Code Paradigms. In *Proc. of the 19th Int. Conf. on Software Engineering (ICSE '97)*, pages 22–33. ACM Press, 1997.
24. R. De Nicola, G. Ferrari, and R. Pugliese. KLAIM: a Kernel Language for Agents Interaction and Mobility. *IEEE Transactions on Software Engineering*, 24(5):315–330, 1998.
25. R. De Nicola, G. Ferrari, R. Pugliese, and B. Venneri. Types for Access Control. *Theoretical Computer Science*, 240(1):215–254, 2000.

26. P. Di Blasio and K. Fisher. A Calculus for Concurrent Objects. In *Proc. of 7th Int. Conf. on Concurrency Theory (CONCUR'96)*, volume 1119 of *LNCS*, pages 655–670. Springer-Verlag, 1996.
27. R. Findler and M. Flatt. Modular object-oriented programming with units and mixins. In *Proc. of ICFP '98*, pages 94–104, 1998.
28. K. Fisher and J. C. Mitchell. A Delegation-based Object Calculus with Subtyping. In *Proc. of FCT'95*, volume 965 of *LNCS*, pages 42–61. Springer-Verlag, 1995.
29. M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and mixins. In *Proc. of POPL '98*, pages 171–183. ACM Press, 1998.
30. C. Fournet, G. Gonthier, J. J. Levy, L. Maranget, and D. Remy. A Calculus of Mobile Agents. In *Proc. of 7th Int. Conf. on Concurrency Theory (CONCUR'96)*, volume 1119 of *LNCS*, pages 406–421. Springer-Verlag, 1996.
31. C. Fournet, C. Laneve, L. Maranget, and D. Rémy. Inheritance in the Join Calculus. In *Proc. of Foundations of Software Technology and Theoretical Computer Science (FSTTCS'00)*, volume 1974 of *LNCS*, pages 397–408. Springer-Verlag, 2000.
32. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
33. D. Gelernter. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.
34. A. Gordon and P. Hankin. A Concurrent Object Calculus: Reduction and Typing. In *Proc. of High-Level Concurrent Languages (HLCL'98)*, volume 16.3 of *ENTCS*. Elsevier, 1998.
35. C. Harrison, D. Chess, and A. Kershenbaum. Mobile agents: Are they a good idea? Research Report 19887, IBM Research Division, 1994.
36. M. Hennessy and J. Riely. Type-Safe Execution of Mobile Agents in Anonymous Networks. In J. Vitek and C. Jensen, editors, *Secure Internet Programming: Security Issues for Distributed and Mobile Objects*, volume 1603 of *LNCS*, pages 95–115. Springer-Verlag, 1999.
37. T. Hirschowitz and X. Leroy. Mixin modules in a call-by-value setting. In *Proc. of ESOP'00*, volume 2305 of *LNCS*, pages 6–20. Springer-Verlag, 2002.
38. F. Knabe. An overview of mobile agent programming. In *Proc. of the 5th workshop on Analysis and Verification of Multiple-Agent Languages (LOMAPS'96)*, volume 1192 of *LNCS*. Springer-Verlag, 1996.
39. C. Laneve. Inheritance in Concurrent Objects. In H. Bowman and J. Derrick, editors, *Formal Methods for Distributed Processing, An Object Oriented Approach*. Cambridge University Press, 2001.
40. S. Matsuoka and A. Yonezawa. Analysis of inheritance anomaly in object-oriented concurrent programming languages. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 107–150. MIT Press, 1993.
41. R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.
42. Object Management Group. Corba: Architecture and specification. `http://www.omg.org`, 1998.
43. B. Pierce and D. Sangiorgi. Typing and Subtyping for Mobile Processes. In *Proc. of 8th IEEE Logics in Computer Science (LICS'93)*, pages 376–385. IEEE, 1993.
44. B. C. Pierce. *Types and Programming Languages*. The MIT Press, 2002.
45. B. C. Pierce and D. N. Turner. Concurrent Objects in a Process Calculus. In *Proc. of Theory and Practice of Parallel Programming (TPPP 94)*, volume 907 of *LNCS*, pages 187–215. Springer, 1995.
46. J. Riecke and C. Stone. Privacy via Subsumption. *Information and Computation*, 172:2–28, 2002. 3rd special issue of Theory and Practice of Object-Oriented Systems (TAPOS).
47. Y. Smaragdakis and D. Batory. Implementing layered designs with mixin layers. In *Proc. of ECOOP '98*, pages 550–570, 1998.
48. Sun microsystems. RMI, Remote Method Invocation. `http://java.sun.com/products/jdk/rmi`.
49. T. Thorn. Programming Languages for Mobile Code. *ACM Computing Surveys*, 29(3):213–239, 1997.
50. M. Van Limberghen and T. Mens. Encapsulation and composition as orthogonal operators on mixins: a solution to multiple inheritance problems. *Object Oriented Systems*, 3(1):1–30, 1996.
51. M. VanHilst and D. Notkin. Using role components to implement collaboration-based designs. In *Proc. of OOPSLA '96*, pages 359–369. ACM Press, 1996.
52. J. E. White. Mobile Agents. In J. Bradshaw, editor, *Software Agents*. AAAI Press and MIT Press, 1996.
53. N. Yoshida and M. Hennessy. Subtyping and Locality in Distributed Higher Order Mobile Processes (extended abstract). In *Proc. of 10th Int. Conf. on Concurrency Theory (CONCUR'99)*, volume 1664 of *LNCS*, pages 557–572. Springer-Verlag, 1999.