A Formal Model of Service-Oriented Dynamic Object Groups ☆

Einar Broch Johnsen[a,*], Olaf Owe[a], Dave Clarke[b], Joakim Bjørk[a]

[a] *University of Oslo, Norway*
[b] *Uppsala University, Sweden & KU Leuven, Belgium*

Abstract

Services are autonomous, self-describing, technology-neutral software units that can be published, discovered, queried, and composed into software applications at runtime. Designing and composing software services to form applications or composite services, require abstractions beyond those found in typical object-oriented programming languages. This paper explores service-oriented abstractions such as service adaptation, discovery, and querying in an object-oriented setting. We develop a formal model of dynamic object-oriented groups which offer services to their environment. These groups fit directly into the object-oriented paradigm in the sense that they can be dynamically created, they have an identity, and they can receive method calls. In contrast to objects, groups are not used for structuring code. A group exports its services through interfaces and relies on objects to implement these services. Objects may join or leave different groups. Groups may dynamically export new interfaces, they support service discovery, and they can be queried at runtime for the interfaces they support. We define an operational semantics and a static type system for this model of dynamic object groups, and show that well-typed programs do not cause method-not-understood errors at runtime.

☆Partly funded by the EU projects FP7-610582 Envisage: Engineering Virtualized Services (`http://www.envisage-project.eu`) and FP7-612985 UpScale: From Inherent Concurrency to Massive Parallelism through Type-based Optimizations (`http://www.upscale-project.eu`).

*Corresponding author

Email addresses: `einarj@ifi.uio.no` (Einar Broch Johnsen), `olaf@ifi.uio.no` (Olaf Owe), `dave.clarke@it.uu.se` (Dave Clarke), `joakimbj@ifi.uio.no` (Joakim Bjørk)

Keywords: Object orientation, Object groups, Service orientation, Multithreading, Concurrency, Types, Semantics, Type safety

1. Introduction

Good software design often advocates a loose coupling between the classes and objects making up a system. Various mechanisms have been proposed to achieve this, including programming to interfaces, object groups, and service-oriented abstractions such as service discovery. By programming to interfaces, client code can be written independently of the specific classes that implement a service, using interfaces describing the services as types in the program. Object groups loosely organize a collection of objects that are capable of addressing a range of requests, reflecting the structure of real-world groups and social organizations in which membership is dynamic [1]; e.g., subscription groups, work groups, service groups, access groups, location groups, etc. Service discovery allows suitable entities (such as objects) that provide a desired service to be found dynamically, generally based on a query on some kind of interface. An advantage of designing software using these mechanisms is that the software is more readily adaptable. In particular, the structure of the groups can change and new services can be provided to replace old ones. The queries to discover objects are based on interface rather than class, so the software implementing the interface can be dynamically replaced by newer, better versions, offering improved services.

This paper explores service-oriented abstractions such as service adaptation, discovery, and querying in an object-oriented setting. Designing software services and composing services in order to form applications or composite services require abstractions beyond those found in typical object-oriented programming languages. To this end, we develop a formal model of dynamic object-oriented groups that also play the role of service providers for their environment. These groups can be dynamically created, they have identity, and they can respond to methods calls, analogously with objects in the object-oriented paradigm. In contrast to objects, groups are not used for executing code. A group exports its services through interfaces and relies on objects to implement these services. From the perspective of client code, groups may be used as if they were objects by programming to interfaces. However, groups support service-oriented abstractions not supported by objects. In particular, groups are *self-describing* in the sense that they

may dynamically export new interfaces, they support service discovery, and they can be queried at runtime for the interfaces they support. Groups are loosely assembled from objects: objects may dynamically join or leave different groups. Mechanisms for one-to-many communication and service replication for robustness are not the main focus of our model, but are to some degree supported. In this paper we develop an operational semantics and a static type system for a kernel language which captures this model of dynamic object groups, based on interfaces, interface queries, groups, and service discovery. The type system ensures that well-typed programs do not cause method-not-understood errors at runtime.

This paper extends a paper which appeared at FOCLASA 2012 [2]. In the extended version of the paper, the formalized kernel language includes a multithreaded concurrency model with reentrant method calls and a release mechanism which makes the language more expressive. These were not part of the language considered in [2]. The extended paper further expands on the use of inner groups for group management and discusses the diversity of object groups in object-oriented systems, and how different usages fit with the proposed kernel language.

The paper is organized as follows. Section 2 presents the language syntax and a small example. A type and effect system for the language is proposed in Section 3 and an operational semantics in Section 4. Section 5 defines a runtime type system and shows that the execution of well-typed programs is type-safe. Section 6 discusses different notions of groups from the perspective of the proposed kernel language. Section 7 discusses related work and Section 8 concludes the paper. The details of the type preservation proof are given in Appendix A.

2. A Kernel Language for Dynamic Object Groups

We study an integration of service-oriented abstractions in an object-oriented setting by defining a kernel object-oriented language with a Java-like syntax, in the style of Featherweight Java [3]. In contrast to Featherweight Java, types are different from classes in this language: interfaces describe services as sets of method signatures and classes generate objects which implement interfaces. By programming to interfaces, the client need not know how a service is implemented. For this reason, the language has a notion of *group* which dynamically connects interfaces to implementations. Groups are first-class citizens; they have identities and may be passed around. An object

Syntactic Categories.	*Definitions.*	
C : Class name	P	$::= \overline{IF}\ \overline{CL}\ B$
I : Interface name	T	$::= \mathsf{Void} \mid \mathsf{Null} \mid \mathsf{Bool} \mid \mathsf{Any} \mid I \mid \mathsf{Group}(\overline{I})$
T : Type name	IF	$::= \textbf{interface}\, I\, \textbf{extends}\, \overline{I}\, \{\overline{Sg}\}$
m : Method name	CL	$::= \textbf{class}\, C(\overline{T}\ \overline{w})\, \textbf{implements}\, \overline{I}\, \{\overline{T}\, \overline{w}; B\ \overline{M}\}$
x : Variable	Sg	$::= T\, m\, ([\overline{T}\, \overline{z}])$
z : Local variable	M	$::= Sg\, B$
w : Field	B	$::= \{\overline{T}\, \overline{z}; sr; \}$
f : Field reference	e	$::= v \mid x \mid \textbf{this}$
e : Expression	f	$::= \textbf{this}.w$
v : Value	x	$::= f \mid z$
s : Statement list	rhs	$::= e \mid \textbf{new}\ C(\overline{e}) \mid [\textbf{yield}]\, x.m(\overline{e}) \mid \textbf{newgroup}$
	$trial$	$::= z\ \textbf{subtypeOf}\ I \mid x = \textbf{acquire}\, I\, \textbf{in}\, x\, \textbf{except}\, \overline{x}\ \mid x\, \textbf{leaves}\, x\, \textbf{as}\, \overline{I}$
	s	$::= \textbf{skip} \mid x = rhs \mid s; s \mid \textbf{while}\, e\, \{s; \} \mid \textbf{if}\, e\, \{s; \}\, \textbf{else}\, \{s; \}$
		$\mid \textbf{try}\ trial\, \{s; \}\, \textbf{else}\, \{s; \} \mid x\, \textbf{joins}\, z\, \textbf{as}\, \overline{I} \mid \textbf{spawn}\, x.m(\overline{e})$
	sr	$::= [s;]\, \textbf{return}\, e$

Figure 1: Kernel language syntax. Square brackets [ ] denote optional elements, and overline denotes repetition (e.g., lists).

may dynamically join a group and thereby add new services to this group, extending the group's supported interfaces. Objects may belong to several groups. Both objects and groups may join and leave groups, thereby migrating their services between groups. Groups offer distribution of work between the implementations of the group's services, because a request to the group can be handled by any object in the group. To study the integration of these service-oriented abstractions, we consider a concurrent kernel language. For a seamless integration with standard object-oriented languages, the kernel language supports multithread concurrency (e.g., [4, 5]), but without shared access to objects. However, this concurrency aspect is largely orthogonal to the group abstraction, which would work equally well with the actor-like concurrency of active objects (e.g., [6, 7]).

2.1. The Syntax

The syntax of the kernel language is given in Figure 1. A type T is either a basic type, an interface describing a service, or a group of interfaces. The types T are the Null type with the value null, the unit type Void with the value void, the basic type Bool of Boolean expressions, the empty interface Any, the names I of the declared interfaces, and group types $\mathsf{Group}(\overline{I})$ which state that a group supports the set $\overline{I}$ of interfaces. The use of types is further detailed in Section 3, including the subtyping relation and the type system.

A *program* P consists of a list $\overline{IF}$ of interface declarations, a list $\overline{CL}$ of class declarations, and a main block $\{\overline{T}\ \overline{z}; s;\ \textsf{return void};\}$. We assume that classes and interfaces have distinct names. The main block introduces a scope with local variables $\overline{z}$ typed by the types $\overline{T}$, and a sequence s of program statements. We conventionally denote by $\overline{z}$ a list or set of the syntactic construct z (in this case, a local program variable), and furthermore we write $\overline{T}\ \overline{z}$ for the list of typed variable declarations $T_1\ z_1; \ldots; T_n\ z_n$ where we assume that the length of the two lists $\overline{T}$ and $\overline{z}$ is the same.

Interface declarations IF associate a name I with a set of method signatures. These method signatures may be inherited from other interfaces $\overline{I}$ or may be declared directly as $\overline{Sg}$. A method *signature* Sg associates a return type T with a name m and method parameters $\overline{z}$ with declared types $\overline{T}$.

Class declarations CL have the form $\textsf{class}\ C(\overline{T}_1\ \overline{w}_1)\ \textsf{implements}\ \overline{I}\ \{\ \overline{T}_2\ \overline{w}_2; B\ \overline{M}\}$ and associates a class name C to the services declared in the interfaces $\overline{I}$. In C, these services are realized using methods to manipulate the fields $\overline{w_2}$ of types $\overline{T_2}$. The constructor block B has the form $\{\overline{T}\ \overline{z}; s;\ \textsf{return void};\}$ and initializes the fields, based on the actual values of the formal class parameters $\overline{w}_1$ of types $\overline{T}_1$. The methods M have a signature Sg and a method body $\{\overline{T}\ \overline{z};\ s;\ \textsf{return}\ e;\ \}$ which introduces a *scope* with local variables $\overline{z}$ of types $\overline{T}$ where the sequence of statements s is executed, after which the value of the expression e is returned to the client. The value void is returned for methods with return type Void, reflecting a trivial return value. Class parameters are treated like fields with both read and write access. There is read-only access to the special variable **this** which refers to the current object. We use the syntax $\textsf{this}.w$ to refer to fields. Thus references to local variables z and fields f are syntactically distinct.

Expressions e of the kernel language consist of program variables x and Java-like expressions for constant values v, including void as well as the Boolean values true and false. *Right-hand-side expressions* (*rhs*) cover object creation $\textsf{new}\ C(\overline{e})$ where the actual constructor parameters are given by $\overline{e}$, and method calls $[\textsf{yield}]\, x.m(\overline{e})$ where the actual method parameters are given by $\overline{e}$. The optional keyword **yield** is attached to a method call to indicate a *release* mechanism such that the caller releases its lock for the duration of the method call. Method calls are synchronous and in contrast to Java all method calls are synchronized; i.e., a caller blocks until a method returns, and a callee will only accept a remote call when it is idle. In addition, we consider an expression related to service-oriented software: the *group creation* expression **newgroup** dynamically creates a new, empty group

which does not offer any service to the environment.

The *statements* s of the kernel language include standard statements such as **skip**, assignments $x = e$, sequential composition $s_1; s_2$, conditionals, and **while**-loops. *Trial* statements are statements that may fail. A trial is embedded in a **try-else** construct such that success in the trial selects the first branch and a fail selects the **else** branch. The trial z **subtypeOf** I is used to *query* a known group z about its supported interfaces. The query succeeds if z offers I (or a better interface), in which case the expanded knowledge of the group z becomes available. For typing reasons, we here require that z is a local variable (as discussed in Section 3.3).

Service discovery is localized to a named group x: The trial $y =$ **acquire** I **in** x **except** $\overline{x}$ tries to find some group g or object o which is in the group x and which offers a service better than I (in the sense of subtyping) and which is not in the set $\overline{x}$.

Service interfaces $\overline{I}$ are *dynamically exported* through a group z by the statement x **joins** z **as** $\overline{I}$, which states that an object or group x is used to implement the interfaces $\overline{I}$ in the group z. Consequently, z will support the interfaces $\overline{I}$ after x has joined the group. Objects and groups x_1 may try to withdraw service interfaces $\overline{I}$ from a group x_2 by the statement **try** x_1 **leaves** x_2 **as** $\overline{I}$ $\{s_1;\}$ **else** $\{s_2;\}$. The withdrawal succeeds if x_2 continues to offer all the interfaces of $\overline{I}$, exported by other objects or groups. Thus, removals do not affect the type of x_2. If the removal is successful then branch s_1 is taken, otherwise s_2 is taken.

Concurrency is obtained by **new** and **spawn** statements. The former creates a new object of a given class and a new thread performing the main block of the class, whereas a **spawn** statement makes a new thread performing the given method, which must be a void method. Thus a **spawn** statement executes the call asynchronously.

2.2. Example

We illustrate the dynamic organization of objects in groups by an example of software which provides text editing support (inspired by [8]). This software provides two interfaces: SpellChecker allows the spell-checking of a piece of text and Dictionary provides functionality to update the underlying dictionary with new words, alternate spellings, etc. Apart from an underlying shared catalog of words, these two interfaces need not share state and may be implemented by different classes. Let us assume that the overall system contains several versions of Dictionary, some of which may have an integrated

SpellChecker. Consider a class implementing a text editor factory, which manages groups implementing these two interfaces. The factory has two methods: makeEditor dynamically assembles such software into a text editor group and replaceDictionary allows the Dictionary to be dynamically replaced in such a group. These methods may be defined as follows:

```
Group(SpellChecker,Dictionary) makeEditor() {
  Group(∅) editor; SpellChecker s; Dictionary d;
  editor = newgroup;
  try d = acquire Dictionary except Nil {skip;} else {d = new Dictionary;};
  try d subtypeOf SpellChecker {
     d joins editor as Dictionary, SpellChecker;
  } else {
     d joins editor as Dictionary;
     s = new SpellChecker();
     s joins editor as SpellChecker;
  };
  return editor;
}

Void replaceDictionary(Group(SpellChecker,Dictionary) editor, Dictionary nd){
  Dictionary od;
  nd joins editor as Dictionary;
  try od = acquire Dictionary in editor except nd {skip;} else {skip;};
  try od leaves editor as Dictionary {skip;} else {skip;};
  return void;
}
```

The method makeEditor acquires a top-level service d which exports the interface Dictionary (by omitting the **in**-clause of the **acquire** statement, we mean that the service discovery happens in the global system). If d also supports the SpellChecker interface, we let d join the newly created group editor as *both* Dictionary and SpellChecker. As a consequence, the group editor will now support the two interfaces Dictionary and SpellChecker. Otherwise d joins the editor group only as Dictionary, and at this point only the interface Dictionary is supported by the group editor. In this case a new SpellChecker object is created and added to the group as SpellChecker, such that the group also supports both interfaces in this execution branch.

The method replaceDictionary will replace a Dictionary service in a text editor group. First we add a new Dictionary service nd to the editor group and then fetch an old service od in the group by means of an **acquire**, where the **except**-clause is used to avoid binding to the new service nd. Finally the old

$$\frac{}{\Gamma \vdash e : \Gamma(e)}\ (\text{T-Exp}) \qquad \frac{}{\Gamma \vdash \textbf{newgroup} : \mathsf{Group}(\emptyset)}\ (\text{T-Group}) \qquad \frac{\Gamma \vdash \overline{e} : \mathit{ptypes}(C)}{\Gamma \vdash \textbf{new}\ C(\overline{e}) : C}\ (\text{T-New}) \qquad \frac{T \prec T' \quad \Gamma \vdash e : T}{\Gamma \vdash e : T'}\ (\text{T-Sub})$$

Figure 2: The type system for expressions and for the creation of objects and groups.

service od is removed as Dictionary in the group by a **leaves** statement. The example illustrates group management by joining and leaving mechanisms as well as service discovery.

3. A Type and Effect System

The language distinguishes behavior from implementations by using an interface as a type that describes a service. Classes are not types in source programs. A class can implement a number of different service interfaces, so its instances can export these services to clients. A program variable typed by an interface can refer to an instance of any class that implements that interface. A group typed by $\mathsf{Group}(\overline{I})$ exports the services described by the set $\overline{I}$ of interfaces to clients, so a program variable of type I may refer to the group if $I \in \overline{I}$. We denote by Void the unit type and by Any the "empty" interface, which extends no interface and declares no method signatures. A service described by an interface may consist of only some of the methods defined in a class that implements the interface, so interfaces lead to a natural notion of hiding for classes. In addition to the source program types used by the programmer, class names are used to type the self-reference **this** in the context of the specific class; i.e., a class name is used as an interface type which exports *all* the methods defined in the class.

Subtyping. The subtyping relation applies to interfaces, classes and groups, with a top element Any and bottom element Null. For interfaces, the subtype relation $\prec$ is defined as the transitive closure of the extends-relation on interfaces: if I extends J' and $J' \prec J$ or $J' = J$, then $I \prec J$. We let a class be a subtype of all its implemented interfaces. We have $\mathsf{Null} \prec X \prec \mathsf{Any}$ where X is an interface, class, or group type. A group type $\mathsf{Group}(S)$ is a subtype of I if there is some $J \in S$ such that $J \prec I$. We also extend the subtype relation to lists of types in the second argument, such that $J \prec \overline{I}$ if $J \prec I$ for each $I \in \overline{I}$. The reflexive closure of $\prec$ is denoted $\preceq$. We let $\mathsf{Group}(S) \preceq \mathsf{Group}(S')$ if for all $J \in S'$ there is some $I \in S$ such that $I \preceq J$.

Finally, $\mathsf{Group}(S) \prec \mathsf{Group}(S')$ expresses that $\mathsf{Group}(S) \preceq \mathsf{Group}(S')$ but not $\mathsf{Group}(S') \preceq \mathsf{Group}(S)$.

Function types $\overline{T} \to T$ are used to type methods with parameter types $\overline{T}$ and return type T. Subtyping for function types is defined as follows: $\overline{T}' \to T' \preceq \overline{T} \to T$ if $\overline{T} \preceq \overline{T}'$ and $T' \preceq T$. We let $m_\omega \in I$ denote that interface I declares a method m with function type ω; and we let $m_\omega \preceq I$ denote $\omega \preceq \omega' \wedge m_{\omega'} \in I$ (for some ω'). Similarly, $I \preceq m_\omega$ denotes $\omega' \preceq \omega \wedge m_{\omega'} \in I$ (for some ω'). The same notation applies to classes. Finally $C \prec I$ denotes that $C \preceq m_\omega$ for each $m_\omega \in I$. Thus $C \prec I$ expresses that C offers all methods of I with the same or better function types.

Typing Contexts. A typing context Γ binds variable names to types. If Γ is a typing context, x a variable, and T a type, we denote by $dom(\Gamma)$ the set of names that are bound to types in Γ (the domain of Γ) and by $\Gamma(x)$ the type to which x is bound in Γ. Define the *update* $\Gamma[x \mapsto T]$ of a typing context Γ by $\Gamma[x \mapsto T](x) = T$ and $\Gamma[x \mapsto T](y) = \Gamma(y)$ if $y \neq x$. By extension, if $\overline{x}$ and $\overline{T}$ denote the lists $x_1, \ldots, x_n$ and $T_1, \ldots, T_n$, we may write $\Gamma[\overline{x} \mapsto \overline{T}]$ for the typing context $\Gamma[x_1 \mapsto T_1] \ldots [x_n \mapsto T_n]$ and $\Gamma[\overline{x_1} \mapsto \overline{T_1}, \overline{x_2} \mapsto \overline{T_2}]$ for $\Gamma[\overline{x_1} \mapsto \overline{T_1}][\overline{x_2} \mapsto \overline{T_2}]$. For typing contexts Γ_1 and Γ_2, we define $\Gamma_1 + \Gamma_2$ such that $\Gamma_1 + \Gamma_2(x) = \Gamma_2(x)$ if $x \in dom(\Gamma_2)$ and $\Gamma_1 + \Gamma_2(x) = \Gamma_1(x)$ if $x \notin dom(\Gamma_2)$. A typing context Γ is *better* than a typing context Δ, denoted $\Gamma \preceq \Delta$, if $dom(\Gamma) = dom(\Delta)$ and $\Gamma(x) \preceq \Delta(x)$ for all x.

The Type and Effect System. Let Γ_B denote the basic typing environment, providing types for the language constants (e.g., $\Gamma_B(\mathsf{null}) = \mathsf{Null}$, $\Gamma_B(\mathsf{void}) = \mathsf{Void}$ and $\Gamma_B(\mathsf{true}) = \mathsf{Bool}$). Programs in the kernel language are analyzed in the context of Γ_B by means of a type and effect system (e.g., [9–11]). Our effects deal with changes in the typing of local group variables only, dynamically modifying their type, depending on the program point. We do not consider changes in the typing of non-local group variables as this could cause the following soundness problem: if a field $\mathsf{Group}(S)$ w could get a smaller type at the point of a local call then the called method may contain an assignment $\mathsf{this}.w = e$ where e is of type $\mathsf{Group}(S)$ but not the smaller type. To obtain a simple type system, we do not here allow changes in the typing of fields. A possible extension is discussed at the end of the section. The inference rules for expressions are given in Figure 2 and for statements, methods, classes, and programs in Figure 3. Following standard conventions, variables appearing in a single premise of a rule are implicitly existentially quantified.

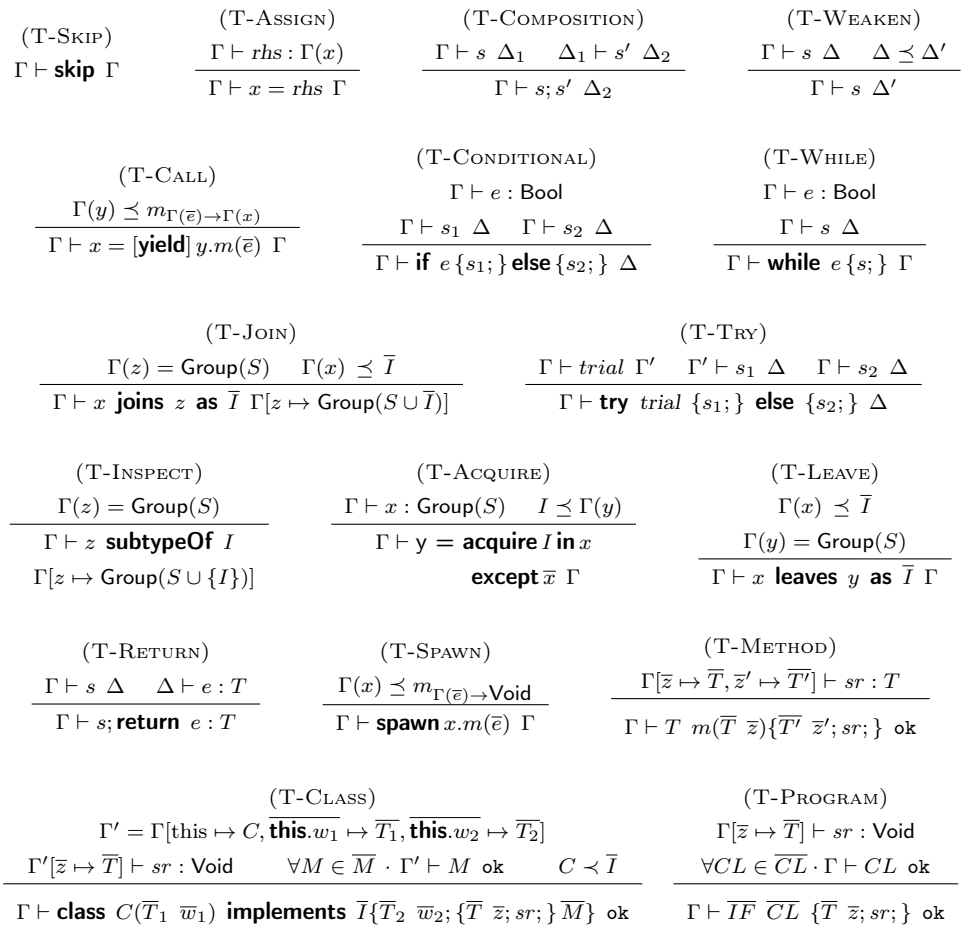

Figure 3: The type and effect system for statements, trials, methods, classes, and programs. Note that x and y denote variables and z local variables.

3.1. Expressions

Expressions are typed by the rules in Figure 2. Let Γ be a typing context. The typing judgment $\Gamma \vdash e : T$ states that the expression e has the type T if the variables in e are typed according to Γ. By T-Exp, constants and variables must be typed in Γ. By T-New, **new** C has type C if the types of the actual parameters to the class constructor can be typed to the declared types of the formal parameters of the class, as captured by the auxiliary function *ptypes*:

$$ptypes(\textsf{class}\ C(\overline{T}\ \overline{w})\ \textsf{implements}\ \overline{I}\{\overline{T}'\ \overline{w}'; B\ \overline{M}\}) = \overline{T}.$$

Note that an expression **new** C may only appear as the right-hand-side of an assignment statement; thus the type of the left-hand-side variable x will restrict C by the requirement $C \preceq \Gamma(x)$. By T-Group, a new group has the empty group type (with no exported interfaces). Rule T-Sub captures subtyping in the type system.

3.2. Statements

Statements are typed by the rules in Figure 3. Let Γ and Δ be typing contexts. The typing judgment $\Gamma \vdash s\ \Delta$ expresses that the statement or trial s is well-typed if the variables in s are typed according to Γ and that the *effect* Δ is the resulting typing context to be used for further analysis. The typing judgment $\Gamma \vdash sr : T$ expresses that the body sr is well-typed according to Γ with T as the resulting type. For a program, class, or method p, the typing judgment $\Gamma \vdash p$ `ok` means that p is well-typed according to Γ, and that the effect will not be needed in further analysis.

The typing of *standard statements* is conventional, but illustrates the effect systems. The statements **skip** and $x = e$ are typed by the rules T-Skip and T-Assign, respectively, and have no effects. The use of effects can be seen clearly in rule T-Composition, where the second statement is type checked in the typing context resulting from the first statement, and the effects are accumulated in the conclusion of the rule. Rule T-While has no effect, since no traversal of the loop is guaranteed. Rule T-Conditional propagates effects from the branches; the resulting effect is approximated by taking the intersection of the effects of the branches. Rule T-Weaken allows information to be discarded in the effect of a typing judgment; e.g., the two branches of a conditional can be unified by means of weakening.

Rule T-Try is similar to T-Conditional except that the effect of the trial is only propagated to the *then* branch, reflecting the effects of success in the trial. Rule T-Spawn is similar to T-Call except that the method must have Void as return type, since the new thread is executed asynchronously. The new thread has no effect on the current typing context.

By T-Call, a call to a method m on a variable y is well-typed if y offers an interface, say I, in which m has a function type ω such that $\omega \preceq \overline{T} \to T$, where $\overline{T}$ are the types of the actual parameters and T is the type of the left-hand-side variable. Using the keyword **yield**, a caller may release its lock for the duration of the method call. For both kinds of calls, the callee may be a group, in which case several interfaces I may satisfy the conditions above. The rule requires that there is at least one such interface I. To

ensure a type-correct binding at runtime, the static type analysis of a call $y.m(\overline{e})$ can associate the function type $\overline{T} \to T$ with the call, transforming it to $y.m_{\overline{T}\to T}(\overline{e})$. This function type gives the least type information needed to ensure well-typedness. Other forms of overloading can be considered, but these are not the focus here.

The typing of the *group manipulation statements* is as follows. By T-Acquire, service discovery is allowed on groups, with the obvious typing constraint on y. By T-Join, when an object joins a group z and contributes interfaces $\overline{I}$ to z, the type of z is extended with the interfaces $\overline{I}$ in the effect. The variable z referencing the group must be locally declared. Without this restriction, a field could dynamically extend its type, resulting in an unsound system; e.g., an assignment $f = e$ in a statically well-typed method could become unsound if the type of f were extended. However extending the type T of a local variable that copies the value of f to a type T' and assigning the result back to a field f' is allowed, as f' would need to be of the extended type T' and f would remain of type T as required by the other method. Rule T-Leave checks that the type of x is $\overline{I}$ (or better) and that y is a group, with *no overall effect*. Rule T-Inspect extends the typing context with the added information about the type of the local variable z, which must be a group.

Programs, classes, methods, and the main method of a program, are typed in the standard way. Methods do not have effects; this reflects that effects are constrained to local variables inside methods. By rule T-Return, the type of the body s; **return** e is the type of e, with no effect, but the effect of type checking s extends the typing environment for type checking of the returned expression. Likewise, classes and programs do not have effects. (For simplicity, we omit the standard type checking of interface declarations.)

Remarks. Since we associate function types with method calls, overloading is possible, and a class may even contain different implementations for the same method signature. In this case an implementation can be chosen non-deterministically, provided that the function type of the call is less than that of the chosen method.

The type system is non-deterministic due to the T-Weaken and T-Sub rules. However there is a least derivable type. The type rules may be used to define an algorithm to compute the least type. The type system in Figure 2 defines a *least type* for every expression and right-hand-side, ignoring T-Sub. Thus one may associate the least function type ω with each call. The type system in Figure 3 defines a *best effect* for every trial and statement, if T-

WEAKEN is removed and the effects of the rules for **if** and **try** statements are replaced by $\Delta_1 \cap \Delta_2$ where Δ_1 is the effect of the *then* branch and Δ_2 the effect of the *else* branch. We define $\Delta_1 \cap \Delta_2$ by $dom(\Delta_1 \cap \Delta_2) = dom(\Delta_1) \cap dom(\Delta_2)$ and by $(\Delta_1 \cap \Delta_2)(x) = \Delta_1(x) \cap \Delta_2(x)$, letting $\mathsf{Group}(S_1) \cap \mathsf{Group}(S_2) = \mathsf{Group}(S)$ where S is the set $\{I \mid \mathsf{Group}(S_1) \preceq I \wedge \mathsf{Group}(S_2) \preceq I\}$ without redundant interfaces (i.e., those that extend other subtypes in the set). This is sufficient as only the typing of local group variables may change.

The least type of a local group variable is initially the declared type and it may be improved by queries and by join statements. The other basic statements maintain the type, except branching statements (**if** and **try**) which make the least type worse, but not worse than the declared type, and not worse than the type prior to the branching construct. Thus, if $z \in dom(\Gamma)$ and $\Gamma \vdash s\ \Delta$, where Δ reflect least types, then $z \in dom(\Delta)$ and $\Delta(z) \preceq \Gamma(z)$.

3.3. Discussion

The type and effect system presented above does not allow changes in the typing of fields. This is because changes in the typing of fields may give an unsound typing system as illustrated by the example below (where the type of the field w is strengthened in method m):

```
interface I1 extends I { ... }
interface I2 extends I { ... }
class UnSound() { I w;
 Void m1(I1 x){ this.w = x; return void; }
 Void m(I1 x){ try this.w subtypeOf I2 { this.m1(x); } else { ... }; return void; }
}
```

Here method $m1$ is type correct. In method m, the trial will improve the type of w to $I2$, but when $m1$ is executed, w will get a value of type $I1$, violating type soundness since $I1$ and $I2$ are unrelated types. As a non-local call may implicitly lead to a local call, the problem illustrated here applies to calls in general, not only local calls; i.e, a call to a method n on another object may lead to a call on method $m1$ on the current object.

However, it would be possible to allow inspect-, join-, and query-statements on local as well as non-local variables at the cost of a much more involved type analysis. The typing of such statements could extend the typing of fields in the same way as local variables, and such changes could be exploited inside a method. One would obtain a type sound system by weakening the type of each field to that of its declared type after a call. This way one could do a

subtype check on a field and then call a method of the added interface in the first branch, resuming with the declared type of all fields after the call. In the example above the type of w would be I after the call. Remark that the same call is accepted by the type system presented in this paper by copying the field to a local variable before the subtype check.

Another way of achieving type soundness is to require that all local methods can be retyped in the extended environment (possibly weakened as explained below). For a given class C one could first formulate a set of additional typing properties $P_C = (\Gamma_j, \Delta_j)$ common to all methods M in C in the sense that $\Gamma_j \vdash M\ \Delta_j$ for every j and every M in C. For each j and M one must then derive $\Gamma_j \vdash M\ \Delta_j$ using the adapted rules T-Method' and T-Call', where the latter makes use of P_C:

$$\frac{\begin{array}{c}\Gamma(y) \preceq m_{\Gamma(\overline{e})\to\Gamma(x)} \quad \Gamma \preceq \Gamma_j \\ (\Gamma_j, \Delta j) \in P_C \quad \Gamma \preceq \Delta j\end{array}}{\Gamma \vdash x = [\textsf{yield}]\ y.m(\overline{e})\ \Delta_j}\ \text{(T-Call')} \qquad \frac{\begin{array}{c}\Delta \vdash e : T \\ \Gamma[\overline{z} \mapsto \overline{T}, \overline{z}' \mapsto \overline{T'}] \vdash s\ \Delta\end{array}}{\Gamma \vdash T\ m(\overline{T}\ \overline{z})\{\overline{T'}\ \overline{z}'; s; \textsf{return}\ e;\}\ \Delta \setminus \{\overline{z}, \overline{z}'\}}\ \text{(T-Method')}$$

where C is the enclosing class and $\Delta \setminus \{\overline{z}, \overline{z}'\}$ is Δ without the local variables $\{\overline{z}, \overline{z}'\}$. Note that in such an approach each method must by type-checked several times with different typing environments, in order to allow T-Call' to flexibly exploit the type information in P_C depending on what is needed for a given call. It could happen that Γ is too strong (too good) for the body B to be well-typed. One may then use weakening to obtain a weaker environment Γ', in which case some of the effects prior to the call may be lost in the environment Δ after the call. A successful weakening is always possible since the method must be well-typed in the typing environment given by rule T-Class' below. Thus one may weaken to that environment if no better environment applies. It is here assumed that Γ, Γ', and Δ have the same domain. Note that Δ cannot be stronger than the current environment Γ since there may not be any call-back. In the example of class *UnSound* the type of w must be weakened to I. (A stronger rule for local calls $\textsf{this}.m(\overline{e})$ could be added, allowing the resulting environment Δ to be stronger than Γ provided the body of m takes Γ' to Δ.)

A somewhat similar discussion applies to class initiator blocks. The effect of the initiator block on fields, without local variable bindings, is used to type the methods. Thereby each method may rely on this effect. This gives the following modification of the class rule:

Syntactic Categories.	*Definitions.*		
o : Object name	cn	::=	$\epsilon \mid o(\sigma, \delta) \mid g(export) \mid t(pr; \rho) \mid t(\texttt{idle}) \mid cn \;\; cn$
g : Group name	σ	::=	$x \mapsto (T, v) \mid \sigma + \sigma$
t : Thread name	δ	::=	$\texttt{free} \mid (t, n)$
	v	::=	$o \mid g \mid \ldots$
	$export$	::=	$\emptyset \mid \{o : I\} \mid export \cup export$
	ρ	::=	$\texttt{idle} \mid \{\sigma \mid \texttt{block}(x); sr\}; \rho$
	pr	::=	$\{\sigma \mid sr\} \mid \texttt{error}$
	s	::=	$\texttt{lock}(n) \mid \ldots$

Figure 4: The runtime syntax, extending the language syntax for values v and statements s. We let n denote a number greater than zero.

$$\frac{\begin{array}{c}\text{(T-Class')} \\ \Gamma' = \Gamma[\text{this} \mapsto C, \overline{\textsf{this}.w_1} \mapsto \overline{T_1}, \overline{\textsf{this}.w_2} \mapsto \overline{T_2}] \\ \Gamma'[\overline{z} \mapsto \overline{T}] \vdash s \;\; \Delta \qquad \forall M \in \overline{M} \cdot (\Delta - \{\overline{z}\}) \vdash M \;\; \Delta' \qquad C \prec \overline{I}\end{array}}{\Gamma \vdash \textsf{class} \;\; C(\overline{T}_1 \;\; \overline{w}_1) \;\; \textsf{implements} \;\; \overline{I}\{\overline{T}_2 \;\; \overline{w}_2; \{\overline{T} \;\; \overline{z}; s; \textsf{return} \, \textsf{void}; \} \, \overline{M}\} \;\; \texttt{ok}}$$

4. Operational Semantics

The runtime syntax is given in Figure 4. A runtime configuration is seen in the context of the classes and interfaces defined by the given program. These definitions are fixed in our setting and give rise to auxiliary look-up functions depending on the class table, see Figure 5. A runtime configuration cn is either the empty configuration ε or a multiset of objects, groups, and threads. Objects $o(\sigma, \delta)$ have an identity o, a state σ defining the fields and class parameters, and a lock δ. We assume that the class name of an object is embedded in the object identity, such that $classOf(o)$ denotes the class of object o and $classOf(\textsf{null})$ is undefined. A state σ maps program variables x to their types T and runtime values v. The update notation of typing environments is reused for states; thus $\sigma[x \mapsto (T, v)]$ updates the binding of x in σ. At runtime, values v include object and group names, in addition to the language constants. The lock δ of an object is either `free` or a thread t has taken the lock n times, denoted (t, n). We let the predicate $\delta(t)$ denote that t has the form (t, n), expressing that the thread t holds the lock δ. Groups $g(export)$ have an identity g and contain a set $export$ of interfaces

I associated with the objects o implementing them, denoted $o : I$. Threads $t(\rho)$ have an identity t and a stack ρ of processes pr. When a thread has processes to execute, it executes the process at the top of its stack. The stack grows with method calls and shrinks at method returns. The empty stack is denoted `idle`.

A process pr has a local state σ, defining the local variables and method parameters, and a sequence sr of statements to be executed in that state, or it is the `error` process which denotes that the computation has gone wrong. To easily distinguish local states from object states, we let l denote the local state of a process, and a the state of an object. Thus $(a + l)$ represents the total state obtained from an object state a and a local state l. The look-up function for program variables x in a state σ is defined by $\sigma(x) = (T, v)$, with the corresponding projections $\sigma^{\mathcal{T}}(x) = T$ and $\sigma^{\mathcal{V}}(x) = v$ to types and values, respectively. Thus, for a state σ, $\sigma^{\mathcal{T}}$ gives the associated mapping of program variables to their current types and $\sigma^{\mathcal{V}}$ the mapping of program variables to their current values. The look-up function is extended to *values* v, such that $\sigma^{\mathcal{V}}(v) = v$ and $\sigma^{\mathcal{T}}(v)$ is the corresponding type. We reserve the name `block` for bookkeeping in the runtime typing rules (cf. Section 5), and assume that it is not in use as a variable name.

The runtime syntax extends the syntax of the surface language as follows. The runtime statement $\texttt{block}(x)$ encodes that the process is waiting for the return value of another process (which is above it on the stack); this return value will be assigned to variable x. Observe that the syntax enforces every frame below the executing frame on a non-idle stack to start with a block statement, these are the only possible occurrences of block statements. At runtime, the statements $\texttt{lock}(n)$ and `return` e are used to manipulate locks. As a simplifying assumption in this paper, a thread t will always lock the object in which it will execute a method activation, and unlock the object when the method activation returns. This is captured in the semantics by the premises $l^{\mathcal{V}}(\texttt{this}) = o$ and $\delta(t)$ when l is the local state of thread t and δ is the lock of object o. This assumption corresponds to synchronized methods in Java.

Structured operational semantics. The operational semantics is given by rules in the style of SOS [12], reflecting small-step semantics. Each rule describes one step in the execution of a thread. Concurrent execution is given by

$$
\begin{array}{ll}
\delta(t) & = \begin{cases} \mathit{true} & \text{if } \delta = (t, n) \\ \mathit{false} & \text{otherwise} \end{cases} \\
\mathit{init}(C, o) & = \{[\overline{z} \mapsto (\overline{T}, \texttt{default}(\overline{T})), \texttt{this} \mapsto (C, o)] \mid sr; \} \\
\mathit{atts}(C, \overline{v}) & = [\overline{\texttt{this}.w_1} \mapsto (\overline{T_1}, \overline{v}), \overline{\texttt{this}.w_2} \mapsto (\overline{T}_2, \texttt{default}(\overline{T}_2))] \\
CT(C) & = [\overline{\texttt{this}.w_1} \mapsto \overline{T_1}, \overline{\texttt{this}.w_2} \mapsto \overline{T_2}] \\
\mathit{export} \preceq I & = \exists\ v : J \in \mathit{export} \cdot J \preceq I \\
\mathit{export} \preceq \overline{I} & = \forall\ I \in \overline{I} \cdot \mathit{export} \preceq I \\
\mathit{export} - \overline{v} & = \{v : J \mid v : J \in \mathit{export} \wedge v \notin \overline{v}\}
\end{array}
$$

Figure 5: Auxiliary definitions in the operational semantics. Here C is $\texttt{class}\ C(\overline{T_1}\ \overline{w_1})\ \texttt{implements}\ \overline{I}\{\overline{T_2}\ \overline{w_2}; \{\overline{T}\ \overline{z}; sr; \}\ \overline{M}\}$. In addition, we use the predicate $\mathit{fresh}(t)$ to denote that thread (or group) name t is globally fresh, and $\mathit{fresh}_C(o)$ to denote that o is a globally fresh object name such that $\mathit{classOf}(\mathit{fresh}_C(o)) = C$, and $\texttt{default}(T)$ to denote some value of type T. A fresh name is not `null`.

standard SOS context and concurrency rules

$$
\frac{cn_1 \to cn_1'}{cn_1\ cn_2 \to cn_1'\ cn_2}\ (\textsc{Interleave}) \qquad \frac{cn_1 \to cn_1' \quad cn_2 \to cn_2'}{cn_1\ cn_2 \to cn_1'\ cn_2'}\ (\textsc{Parallel})
$$

We assume associative and commutative matching over configurations (as in rewriting logic [13]). Thus threads can execute in parallel in distinct parts of the configuration, which leads to the following restrictions on concurrent execution: Two threads which require the *same object* cannot execute in parallel. In a thread-based setting, a thread must take the lock of an object to access its state, and explicitly release this lock when it no longer requires the object's state. Two partial functions inc_t and dec_t capture the locking and unlocking discipline for a thread t; $\mathit{inc}_t(n, \delta)$ is applied whenever t wants to grab a lock δ (and grabs the lock n times) and $\mathit{dec}_t(\delta)$ is applied whenever t wants to release a lock δ. These functions are defined as follows:

$$
\begin{array}{ll}
\mathit{inc}_t(n, \texttt{free}) = (t, n) & \mathit{dec}_t((t, 1)) = \texttt{free} \\
\mathit{inc}_t(n, (t, n')) = (t, n + n') & \mathit{dec}_t((t, n + 1)) = (t, n)
\end{array}
$$

In the rules, if the function is undefined, then the rule cannot be applied.

The transition rules are given in Figures 6 and 7. All rules which make use of the object state require that the thread t has already taken the object lock δ, expressed by the auxiliary predicate $\delta(t)$. Rules involving a group will lock the group in question for one reduction step, thereby disallowing concurrent execution of other threads which require access to the interface

$$\frac{}{t(\{l \mid \texttt{skip}; sr\}; \rho) \to t(\{l \mid sr\}; \rho)}\ (\textsc{Skip})$$

$$\frac{}{t(\{l \mid \texttt{while}\, e\, \{s;\}; sr\}; \rho) \to t(\{l \mid \texttt{if}\, e\, \{s; \texttt{while}\, e\, \{s;\}\} \texttt{else}\, \{\texttt{skip}\}; sr\}; \rho)}\ (\textsc{While})$$

$$\frac{l^{\mathcal{V}}(\texttt{this}) = o \quad \delta(t) \quad (a+l)^{\mathcal{V}}(e) = \mathit{true}}{t(\{l \mid \texttt{if}\, e\, \{s_1;\}\, \texttt{else}\, \{s_2;\}; sr\}; \rho)\ o(a,\delta) \to t(\{l \mid s_1; sr\}; \rho)\ o(a,\delta)}\ (\textsc{Cond1})$$

$$\frac{l^{\mathcal{V}}(\texttt{this}) = o \quad \delta(t) \quad (a+l)^{\mathcal{V}}(e) = \mathit{false}}{t(\{l \mid \texttt{if}\, e\, \{s_1;\}\, \texttt{else}\, \{s_2;\}; sr\}; \rho)\ o(a,\delta) \to t(\{l \mid s_2; sr\}; \rho)\ o(a,\delta)}\ (\textsc{Cond2})$$

$$\frac{l^{\mathcal{V}}(\texttt{this}) = o \quad \delta(t) \quad l^{\mathcal{T}}(z) = T \quad (a+l)^{\mathcal{V}}(e) = v}{t(\{l \mid z = e; sr\}; \rho)\ o(a,\delta) \to t(\{l[z \mapsto (T,v)] \mid sr\}; \rho)\ o(a,\delta)}\ (\textsc{Assign1})$$

$$\frac{l^{\mathcal{V}}(\texttt{this}) = o \quad \delta(t) \quad a^{\mathcal{T}}(f) = T \quad (a+l)^{\mathcal{V}}(e) = v}{t(\{l \mid f = e; sr\}; \rho)\ o(a,\delta) \to t(\{l \mid sr\}; \rho)\ o(a[f \mapsto (T,v)],\delta)}\ (\textsc{Assign2})$$

$$\frac{l^{\mathcal{V}}(\texttt{this}) = o \quad \delta(t) \quad \delta' = \mathit{inc}_t(n,\delta)}{t(\{l \mid \texttt{lock}(n); sr\}; \rho)\ o(a,\delta) \to t(\{l \mid sr\}; \rho)\ o(a,\delta')}\ (\textsc{Lock-Object})$$

$$\frac{\mathit{fresh}(g) \quad \delta(t)}{t(\{l \mid x = \texttt{newgroup}; sr\}; \rho) \to t(\{l \mid x = g; sr\}; \rho)\ g(\emptyset)}\ (\textsc{New-Group})$$

$$\frac{\delta(t) \quad (a+l)^{\mathcal{V}}(x) = o' \quad \mathit{classOf}(o') = C \quad l^{\mathcal{V}}(\texttt{this}) = o \quad \mathit{fresh}(t') \quad pr = \mathit{bind}(m_\omega, o', C, (a+l)^{\mathcal{V}}(\overline{e}))}{t(\{l \mid \texttt{spawn}\, x.m_\omega(\overline{e}); sr\}; \rho)\ o(a,\delta) \to t(\{l \mid sr\}; \rho)\ t'(pr; \texttt{idle})\ o(a,\delta)}\ (\textsc{New-Thread})$$

$$\frac{\mathit{fresh}_C(o') \quad \mathit{fresh}(t') \quad l^{\mathcal{V}}(\texttt{this}) = o \quad pr = \mathit{init}(C,o') \quad \delta(t) \quad a' = \mathit{atts}(C, (a+l)^{\mathcal{V}}(\overline{e}))}{t(\{l \mid x = \texttt{new}\ C(\overline{e}); sr\}; \rho)\ o(a,\delta) \to t(\{l \mid x = o'; sr\}; \rho)\ o(a,\delta)\ t'(pr; \texttt{idle})\ o'(a', (t',1))}\ (\textsc{New-Object})$$

$$\frac{\delta(t) \quad (a+l)^{\mathcal{V}}(y) = o' \quad \mathit{classOf}(o') = C \quad l^{\mathcal{V}}(\texttt{this}) = o \quad pr = \mathit{bind}(m_\omega, o', C, (a+l)^{\mathcal{V}}(\overline{e}))}{t(\{l \mid x = y.m_\omega(\overline{e}); sr\}; \rho)\ o(a,\delta) \to t(pr; \{l \mid \texttt{block}(x); sr\}; \rho)\ o(a,\delta)}\ (\textsc{Call1})$$

$$\frac{\delta = (t,n) \quad (a+l)^{\mathcal{V}}(y) = o' \quad \mathit{classOf}(o') = C \quad l^{\mathcal{V}}(\texttt{this}) = o \quad pr = \mathit{bind}(m_\omega, o', C, (a+l)^{\mathcal{V}}(\overline{e}))}{t(\{l \mid x = \texttt{yield}\ y.m_\omega(\overline{e}); sr\}; \rho)\ o(a,\delta) \to t(pr; \{l \mid \texttt{block}(x); \texttt{lock}(n); sr\}; \rho)\ o(a, \texttt{free})}\ (\textsc{Call2})$$

Figure 6: The operational semantics (1).

(Call3)

$$\frac{\begin{array}{c} l^{\mathcal{V}}(\texttt{this}) = o \quad (a+l)(y) = (\mathsf{Group}(S), g) \quad I \in S \\ \delta(t) \quad v : I \in \mathit{export} \quad I \preceq m_\omega \end{array}}{\begin{array}{c} t(\{l \mid [\texttt{yield}]\ x = y.m_\omega(\overline{e}); sr\}; \rho)\ \ o(a,\delta)\ \ g(\mathit{export}) \\ \rightarrow t(\{l \mid [\texttt{yield}]\ x = v.m_\omega(\overline{e}); sr\}; \rho)\ \ o(a,\delta)\ \ g(\mathit{export}) \end{array}}$$

(Join)

$$\frac{\begin{array}{c} (a+l)^{\mathcal{V}}(x) = v \quad l(z) = (\mathsf{Group}(S), g) \\ l^{\mathcal{V}}(\texttt{this}) = o \quad T = \mathsf{Group}(S \cup \overline{I}) \\ \delta(t) \quad \mathit{export}' = \bigcup_{I \in \overline{I}} \{v : I\} \cup \mathit{export} \end{array}}{\begin{array}{c} t(\{l \mid x\ \texttt{joins}\ z\ \texttt{as}\ \overline{I}; sr\}; \rho)\ \ o(a,\delta)\ \ g(\mathit{export}) \\ \rightarrow t(\{l[z \mapsto (T, g)] \mid sr\}; \rho)\ \ o(a,\delta)\ \ g(\mathit{export}') \end{array}}$$

(Return)

$$\frac{l^{\mathcal{V}}(\texttt{this}) = o \quad \delta(t) \quad \delta' = \mathit{dec}_t(\delta) \quad (a+l)^{\mathcal{V}}(e) = v}{\begin{array}{c} t(\{l \mid \texttt{return}\ e\}; \{l' \mid \texttt{block}(y); sr\}; \rho)\ \ o(a,\delta) \\ \rightarrow t(\{l' \mid y = v; sr\}; \rho)\ \ o(a,\delta') \end{array}}$$

(End)

$$\frac{l^{\mathcal{V}}(\texttt{this}) = o \quad \delta(t) \quad \delta' = \mathit{dec}_t(\delta)}{t(\{l \mid \texttt{return e}\}; \texttt{idle})\ \ o(a,\delta) \rightarrow o(a,\delta')}$$

(Acquire1)

$$\frac{\begin{array}{c} \delta(t) \quad l^{\mathcal{V}}(\texttt{this}) = o \quad (a+l)^{\mathcal{V}}(y) = g \\ (v : J) \in \mathit{export} - (a+l)^{\mathcal{V}}(\bar{x}) \quad J \preceq I \end{array}}{\begin{array}{c} t(\{l \mid \texttt{try}\ \ x = \texttt{acquire}\ I\ \texttt{in}\ y\ \texttt{except}\ \bar{x}\ \{s_1\} \\ \texttt{else}\ \{s_2\}; sr\}; \rho)\ \ o(a,\delta)\ \ g(\mathit{export}) \\ \rightarrow t(\{l \mid x = v; s_1; sr\}; \rho)\ \ o(a,\delta)\ \ g(\mathit{export}) \end{array}}$$

(Acquire2)

$$\frac{\begin{array}{c} \delta(t) \quad l^{\mathcal{V}}(\texttt{this}) = o \quad (a+l)^{\mathcal{V}}(y) = g \\ \mathit{export}' = \mathit{export} - (a+l)^{\mathcal{V}}(\bar{x}) \quad \mathit{export}' \not\preceq I \end{array}}{\begin{array}{c} t(\{l \mid \texttt{try}\ \ x = \texttt{acquire}\ I\ \texttt{in}\ y\ \texttt{except}\ \bar{x}\ \{s_1\} \\ \texttt{else}\ \{s_2\}; sr\}; \rho)\ \ o(a,\delta)\ \ g(\mathit{export}) \\ \rightarrow t(\{l \mid s_2; sr\}; \rho)\ \ o(a,\delta)\ \ g(\mathit{export}) \end{array}}$$

(Leave1)

$$\frac{\begin{array}{c} (a+l)^{\mathcal{V}}(y) = g \quad (a+l)^{\mathcal{V}}(x) = v \\ \delta(t) \quad \mathit{export}' = \mathit{export} \setminus \bigcup_{I \in \overline{I}} \{v : I\} \\ l^{\mathcal{V}}(\texttt{this}) = o \quad \mathit{export}' \preceq \overline{I} \end{array}}{\begin{array}{c} t(\{l \mid \texttt{try}\ \ x\ \ \texttt{leaves}\ \ y\ \ \texttt{as}\ \ \overline{I}\ \ \{s_1\} \\ \texttt{else}\ \ \{s_2\}; sr\}; \rho)\ \ o(a,\delta)\ \ g(\mathit{export}) \\ \rightarrow t(\{l \mid s_1; sr\}; \rho)\ \ o(a,\delta)\ \ g(\mathit{export}') \end{array}}$$

(Leave2)

$$\frac{\begin{array}{c} (a+l)^{\mathcal{V}}(y) = g \quad (a+l)^{\mathcal{V}}(x) = v \\ \delta(t) \quad \mathit{export}' = \mathit{export} \setminus \bigcup_{I \in \overline{I}} \{v : I\} \\ l^{\mathcal{V}}(\texttt{this}) = o \quad \mathit{export}' \not\preceq \overline{I} \end{array}}{\begin{array}{c} t(\{l \mid \texttt{try}\ \ x\ \ \texttt{leaves}\ y\ \ \texttt{as}\ \overline{I}\ \{s_1\} \\ \texttt{else}\ \{s_2\}; sr\}; \rho)\ \ o(a,\delta)\ \ g(\mathit{export}) \\ \rightarrow t(\{l \mid s_2; sr\}; \rho)\ \ o(a,\delta)\ \ g(\mathit{export}) \end{array}}$$

(Query1)

$$\frac{\begin{array}{c} \delta(t) \quad l^{\mathcal{V}}(\texttt{this}) = o \\ l(z) = (\mathsf{Group}(S), g) \quad \mathit{export} \preceq I \end{array}}{\begin{array}{c} t(\{l \mid \texttt{try}\ \ z\ \ \texttt{subtypeOf}\ I\ \{s_1\} \\ \texttt{else}\ \{s_2\}; sr\}; \rho)\ \ g(\mathit{export})\ \ o(a,\delta) \\ \rightarrow t(\{l[z \mapsto (\mathsf{Group}(S \cup \{I\}, g))] \mid s_1; sr\}; \rho) \\ o(a,\delta)\ \ g(\mathit{export}) \end{array}}$$

(Query2)

$$\frac{\begin{array}{c} \delta(t) \quad l^{\mathcal{V}}(\texttt{this}) = o \\ l^{\mathcal{V}}(z) = g \quad \mathit{export} \not\preceq I \end{array}}{\begin{array}{c} t(\{l \mid \texttt{try}\ \ z\ \ \texttt{subtypeOf}\ I\ \{s_1\} \\ \texttt{else}\ \{s_2\}; sr\}; \rho)\ \ o(a,\delta)\ \ g(\mathit{export}) \\ \rightarrow t(\{l \mid s_2; sr\}; \rho)\ \ o(a,\delta)\ \ g(\mathit{export}) \end{array}}$$

Figure 7: The operational semantics (2).

information of that group. This is crucial in the rules JOIN and LEAVE1, which may actually modify the interface information of the group.

The SKIP rule is standard; being independent of the object, it expresses that a skip has no effect on the object state. The two rules COND1 and COND2 handle the two cases of the conditional statement by evaluating the condition in the object state. The effect of assignment is described by two rules, ASSIGN1 for the assignment to local variables, updating l, and ASSIGN2 for the assignment to fields, updating a. A loop unwinds into a conditional, by rule WHILE. In rule NEW-GROUP, a globally unique group identifier is ensured by the auxiliary predicate $\mathit{fresh}(g)$. An empty group with this identifier is added to the configuration. Rule NEW-THREAD similarly adds a new thread to the configuration, with a globally unique identifier. The new thread gets a stack which consists of a single process frame, corresponding to the called method.

The **new** statement is handled by the NEW-OBJECT rule, where $\mathit{fresh}_C(o')$ and $\mathit{fresh}(t')$ assert that o' and t' do not previously occur in the global configuration and that $\mathit{classOf}(o') = C$. An object with this name is created. The auxiliary function $\mathit{atts}(C, \overline{v})$ maps the declared fields of C to their declared types and default values, and class parameters to declared types and actual values. (The default value of an interface or group type is null.) The auxiliary function $\mathit{init}(C, o')$ returns the process corresponding to the init-block of C, binding `this` to type C and value o'. This process is executed in a new thread with identity t', which holds the lock to o'.

Method calls are handled by rule CALL1 for calls to objects, rule CALL2 for releasing calls. and rule CALL3 for calls to groups. We assume that the function type ω of the call has been added during static analysis, as explained in Section 3. Let the auxiliary function $\mathit{bind}(m_\omega, o, C, \overline{v})$ return the process resulting from the activation of a method m of class C in object o such that the function type of m is equal or better than ω.

$$\mathit{bind}(m_\omega, o, C, \overline{v}) = \texttt{error} \text{ if } C \not\preceq m_\omega$$

The function $\mathit{bind}(m_\omega, o, C, \overline{v})$ is `error` unless C defines a method $T\ m(\overline{T}\ \overline{z})\ \{\overline{T}'\ \overline{z}'; sr;\}$ such that $\overline{T} \to T \preceq \omega$; and in this case we define

$$\mathit{bind}(m_\omega, o, C, \overline{v}) = \{[\overline{z} \mapsto (\overline{T}, \overline{v}), \overline{z}' \mapsto (\overline{T}', \texttt{default}(\overline{T}')), \texttt{this} \mapsto (C, o)] | \texttt{lock}(1); sr;\}$$

The local state maps the parameters of m to their declared types and values $\overline{v}$, the local variables to their declared types and default values, and `this` to

C and o. The *bind* function is deterministic if each class has distinct names for methods with a given number of arguments.

For simplicity, we assume that all methods are *synchronized* in the sense of Java and let the method body in the method activation be preceded by a runtime statement `lock`(n); i.e., the actual method body can only be executed once the thread has taken the lock to the object o n times. In rule Lock-Object a thread t takes the lock δ of an object n times by means of $inc_t(n, \delta)$. This operation succeeds if the lock is either *free* or already taken by t. When a call is made to an object in Call1, the new process $bind(m_\omega, o, C, (a + l)^{\mathcal{V}}(\overline{e}))$ is added to the stack of the thread, where C is the class of the callee. Rule Call2 implements the release mechanism associated with **yield**, which allows a thread to make a method call without keeping the lock of the caller object. The caller remains on the stack, and the thread must compete for the lock before it can execute when control returns. Although the thread may have taken the lock n times, the lock is `free` when the thread leaves the object, and the lock is taken n times upon return. In Call3, a call to a group is reduced to a call to a group or an object *inside* the callee which exports an appropriate interface to the original group in the sense that the called method is supported by the interface. The rules Call1 and Call3 are selected depending on the actual value of the callee y, which may be either an object or a group. Rule Return handles returns from method calls. Here the `block`(y) statement in the frame below the active frame (on the top of the stack) is *replaced* by $y = v$, assigning the value returned from the active frame to y, and the active frame is removed from the stack. Rule Return decrements the lock δ once by means of $dec_t(\delta)$. This operation succeeds if the lock counter is positive, if the lock counter is 1 then the decrement makes the lock `free`. Rule End is similar to rule Return, handling the completion of initialization blocks, main programs, and spawned threads. The associated thread is terminated, in which case the thread no longer holds any locks. This rule takes care of the garbage collection of threads.

The rule Join dynamically extends a group z with support for the services described in the interfaces $\overline{I}$. From the perspective of the thread, the type of the variable referencing the group is extended. From the group's perspective, the *export* set is extended. Observe that in the join statement, x may itself be a group; well-typedness ensures that x offers at least $\overline{I}$. Service discovery is handled by the Acquire rules. In Acquire1 the **acquire** statement is replaced by a value v, which is an object or group identifier satisfying the **in** and **except** clauses. The notation $export \preceq I$ means that the *export* list

has an interface J such that $J \preceq I$. If this condition is not satisfied the **else** branch is taken as defined in ACQUIRE2. An **acquire** statement which is not restricted to searching a specific group could be added, but requires constructing a "global" group, e.g., $global(\cup_i\ export_i)$ where $g_i(export_i)$ are all the groups in the configuration.

The **leaves** statement is handled by the rules LEAVE1 for a successful leave and LEAVE2 for an unsuccessful one. A group or object x succeeds in leaving a group if the group continues to provide the same interface support without x. To determine this, we use the subtype relation lifted to group export lists as defined in Figure 5. An entry is redundant if a subtype of the entry is present in the set. The type of the group does not change by a **leaves** statement and hence the object does not need to update information about the group. The branches s_1 or s_2 are chosen depending on whether the interfaces remain supported. The rules QUERY1 and QUERY2 handle the branching determined by a query. If the local group z exports a given interface, the query succeeds and the s_1 branch is taken, updating z with the new interface information. If the query fails the s_2 branch is chosen by QUERY2.

The initial configuration. For a program $P = \overline{IF}\ \overline{CL}\ \{\overline{T}\ \overline{z}; sr\}$, we define the initial configuration to be $o(\varepsilon, (t, 1))\ t(\{[\overline{z} \mapsto (\overline{T}, \texttt{default}(\overline{T})), \texttt{this} \mapsto (C, o)] \mid sr; \}; \texttt{idle})$ where o, t, and C are fresh names such that $classOf(o)$ is C and $\mathsf{Null} \prec C \prec \mathsf{Any}$. The fresh name C plays the role of a *Main* class.

Remarks. Our semantics covers runtime errors resulting from method binding. Runtime errors caused by null pointers are indirectly captured in the sense that execution of the process stops because no more rules can be applied to it. This is the case when a call is made to a null object or group, when joining, leaving, or querying a null group, and when acquiring an interface in a null group. We could add rules explicitly generating the `error` process in these cases; however, as these kinds of errors do not play a role in the results on type safety, we do not need them for our purposes. Static checks ensuring non-null pointer values would be a way to solve these kinds of errors (our semantics guarantees that **this** is not null).

The configurations defined by the operational semantics have state information for each object and thread, and these states include explicit type information. As for the static type system, the types of fields do not change, and the types of local variables that are not group variables do not change. The basic statements have the same effect on the types as in the static type

system. The typing effect of `if`, `try`, and `while` statements is different from that of the static type system, as it uses the typing effect from the chosen branch, whereas the static typing of `if` and `try` statements uses weakening, and the static typing of `while` ignores the effect of the loop. As a consequence the types at runtime are better than (or equal to) than the corresponding types defined by the static type system, which again are better than (or equal to) the declared types.

Notice that all lock handling is managed by the operational semantics, letting lock statements be inserted by *bind* and the rule for **yield**, and letting **return** statements decrease the lock. The initial block of a thread takes the lock of **this** object and releases it by rule End. Assuming we start from an initial configuration, the lock of **this** object is continuously held by the thread, and with the same lock value. Under this assumption, the premise $\delta(t)$ is redundant in all rules except the rule for lock statements.

5. Type Safety

This section extends the type system of Section 3 to runtime configurations and shows type preservation for the execution of well-typed programs.

5.1. Well-Typed Configurations

The extension of the type system to runtime configurations is given in Figure 8. The typing context Γ contains the types of all constant values (including object, group, and thread identities) at runtime. By RTT-Config, a configuration is well-typed if all objects, groups, and threads are well-typed. By RTT-Group, a group is well-typed if all the objects which export interfaces through the group implement these interfaces and that each interface supported by the group according to the type system is exported by an object in the group (checked by RTT-Exps and RTT-Exp). We denote by $CT(C)$ the typing context which maps the fields of C to their declared types. By RTT-Object, an object o is well-typed if its fields a are well-typed in $\Gamma + CT(\Gamma(o))$ and its lock is well-typed. This means that the typing of fields is invariant over execution. Substitutions (the state of fields and local variables) are checked by RTT-Subs and RTT-Sub. By RTT-ObjectLock, the object's lock is well-typed if a thread t holds the lock n times, where n is an integer.

A thread is well-typed by RTT-Thread1 if its stack is well-typed and by RTT-Thread2 if it is idle. A stack is well-typed by RTT-Idle if it is `idle`

and by RTT-Stack if all its processes are well-typed by RTT-Frame1 and RTT-Frame2; i.e., the state of local variables, the `block` statement if the stack is not active, and the method body sr are well-typed. The runtime syntax enforces the usage of the `block` statement only once in a frame, at the head of the frame. Observe that due to the query-mechanism of the language, the types of local program variables in two processes which stem from activations of the same method, may differ at runtime. This is in contrast to the typing of fields. For this reason, the typing context used for typing runtime configurations cannot rely on the statically declared types of program variables. This explains why RTT-Frame1 and RTT-Frame2 extend Γ with the *locally stored typing information* $l^{\mathcal{T}}$ to type check $l^{\mathcal{V}}$, sr, and $\texttt{block}(x)$. The effects of the static type system are not needed here, as they are reflected by how the operational semantics updates this local type information. The additional runtime statement $\texttt{lock}(n)$ is well-typed by RTT-Lock. The rules from the static type checking are reused as appropriate. The remaining rules RTT-Empty, RTT-Free, RTT-Def, and RTT-EmptyGroup are straightforward.

5.2. Subject Reduction

The type system guarantees that the type of *fields* in an object never changes at runtime. The static typing of methods in well-typed programs allows us to establish as Lemma 1 that method binding, if successful, results in a well-typed process at runtime. To show that the `error` process cannot occur in the execution of well-typed programs, it suffices to show that substitutions are always well-typed. Lemma 2 shows that this is the case for the initial configuration and Lemma 3 shows that one execution step preserves runtime well-typedness. Together, these lemmas establish a subject reduction theorem for the language, expressing that well-typedness is preserved during the execution of well-typed programs and in particular that method binding always succeeds. Here, $\rightarrow^*$ denotes the reflexive and transitive closure of the reduction relation $\rightarrow$.

Lemma 1. *Let $m_{\overline{T}\rightarrow T} \in C$ be declared in a well-typed program and let o be an object of that program such that $classOf(o) = C$. If $\Gamma \vdash \overline{v} : \overline{T}'$, $\overline{T}' \preceq \overline{T}$, and $T \preceq T'$, then $\Gamma + CT(C) \vdash bind(m_{\overline{T'}\rightarrow T'}, o, C, \overline{v}) : T'$.*

Lemma 2. *Let P be a program such that $\Gamma \vdash P$ `ok` and let cn be the initial configuration of P. Then there is a Γ' such that $\Gamma' \vdash cn$ `ok` and $\Gamma \subseteq \Gamma'$.*

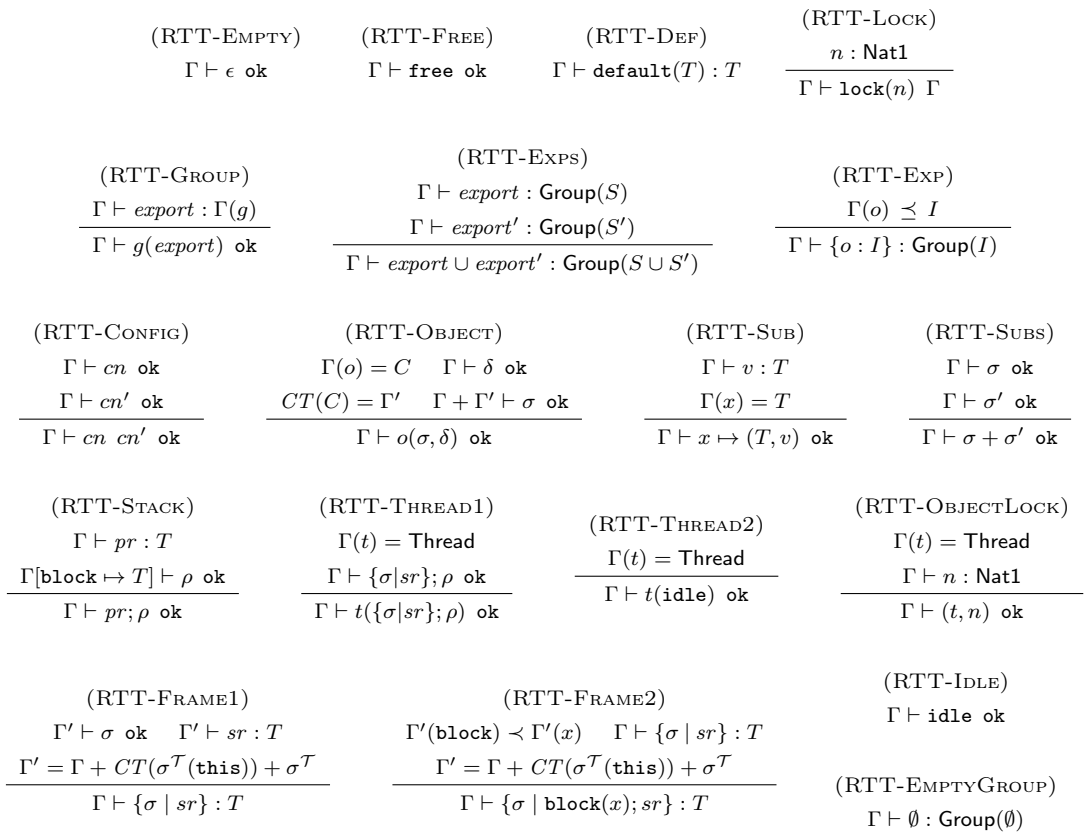

Figure 8: The runtime type system. Nat1 denotes the natural numbers greater than zero.

Lemma 3. *If* $\Gamma \vdash cn$ `ok` *and* $cn \to cn'$ *then there is a* Γ' *such that* $\Gamma' \vdash cn'$ `ok` *and* $\Gamma \subseteq \Gamma'$.

The proofs of the lemmas are found in Appendix A.

Theorem 1 (Subject reduction)**.** *Let* $\Gamma \vdash P$ `ok` *and let* cn *be the initial runtime configuration of* P. *If* $cn \to^* cn'$ *then there is a* Γ' *such that* $\Gamma' \vdash cn'$ `ok` *and* $\Gamma \subseteq \Gamma'$.

Proof. The proof is by induction over the length of the reduction sequence. The two cases follow directly from Lemmas 2 and 3. □

Notice that unsuccessful method binding gives the special process `error`, which is not well-typed since there is no type rule for `error`. Thus our notion of subject reduction implies that `error` may not occur, i.e., method binding

will succeed. However, the execution of a thread may stop if the thread is blocked, in the sense that it does not have the lock to the object on top of the process stack, or is accessing a null pointer. A statement s is said to be *null-free* in a given state if no variable in the statement has the value `null` apart from left-hand-side variables and actual parameters, ignoring inner blocks. A list of statements sr is null-free if the first statement (which may be the **return** statement) is null-free. For example, a **try** statement is null-free if the trial is null-free. We prove a notion of *local progress*, expressing that a thread can proceed when it is not blocked and the next statement is null-free.

Lemma 4 (Local progress)**.** *Consider a well-typed configuration containing a thread* $t(\{l \mid sr\}; \rho)$ *and an object* $o(\sigma, \delta)$*, such that* $l^{\mathcal{V}}(\texttt{this}) = o$ *and* $\delta(t)$*. If* sr *is null-free in the state* $(\sigma + l)$*, there will be exactly one applicable operational rule modifying the thread.*

Together, Theorem 1 and Lemma 4 ensure that "well-typed programs do not cause method-not-understood errors at runtime" in the sense of [14]. Our language deals with concurrent execution and deadlocks may occur. The type system does not restrict deadlocks, but ensures that each non-blocked process in a well-typed configuration can make a transition, and in particular, method binding will succeed, provided the callee is not `null`. The condition of null-freeness may be extended to cover well-defined expressions, i.e., the absence of errors in expressions such as division by zero, and then the progress property would assume well-defined expressions. (Alternatively an exception mechanism could be added to the language.)

6. Diversity in Object Groups

The kernel language formalized in this paper proposes a lightweight notion of dynamic object groups in which the main emphasis is on how service-oriented abstractions combine with groups to allow service discovery, migrating a service from one service provider to another, etc. In our model, objects seen as service providers may offer their services through several groups. The proposed mechanisms are lightweight in that the required additional machinery to handle the groups is very small.

Object groups in object-oriented systems are not a uniform concept, but have been used for a variety of purposes. In this section, we consider an additional construct for communication in object groups and discuss how these constructs fit with different ways of using object groups. For simplicity, we

have not opted for integrating the additional construct in the kernel language. However, the extension of the kernel language is fairly straightforward and does not lead to complications from the typing perspective. For convenience, variable names in the examples will be kept disjoint and we refer to fields without dot-notation; e.g., we write publicGroup rather than **this**.publicGroup.

6.1. Unicast vs. Broadcast

The dynamic object group model considered in the kernel language of this paper is based on *unicast* communication following the standard call and return structure of object-oriented languages. A different notion of dynamic object group can be obtained by considering *broadcast* communication. In such a scenario, all objects in a group who provide the service, receive the call from the group's client. Our model can be adapted to such a scenario, for example by introducing an explicit primitive into the language to express broadcast communication: **broadcast** $y.m(\overline{e})$. Dynamic object groups allow concurrent activities between their objects, so in the multithread setting broadcast is most naturally captured by spawning new threads for the calls to each object. The main issue here is how to handle the reply values from multiple calls. An interesting solution is to collect the replies from the group into a list: if the return type from a method m is type T then the return type to a broadcast to method m would be List<T>. If we assume that the compiler introduces an auxiliary method $\texttt{broadcast}_{m_\omega}$ for every broadcasted method m_ω, this would lead to the following rule in the semantics:

$$\frac{\begin{array}{c}(\textsc{Broadcast})\\ l^{\mathcal{V}}(\texttt{this}) = o \quad \delta(t) \quad (a+l)^{\mathcal{V}}(y) = g \\ S = \{o' | o' : J \in \mathit{export} \wedge J \preceq m_{\overline{T}\to T}\} \quad (a+l)^{\mathcal{T}}(x) = \mathsf{List<}T\mathsf{>}\end{array}}{\begin{array}{c} o(a,\delta)\ \ t(\{l \mid x = \texttt{broadcast}\ \ y.m_{\overline{T}\to T}(\overline{e}); sr\}; \rho)\ \ g(\mathit{export}) \\ \to o(a,\delta)\ \ t(\{l \mid x = \texttt{broadcast}_{m_{\overline{T}\to T}}(S,\overline{e}); sr\}; \rho)\ \ g(\mathit{export})\end{array}}$$

The auxiliary method $\texttt{broadcast}_{m_\omega}$ unwraps method calls to m to a set S of receivers, as illustrated in Figure 9. Here, some(S) denotes some value in the set S, Nil denotes the empty list, and Cons(l,e) denotes the list constructor which appends an element e to the list l. The auxiliary method uses **yield** to invoke concurrent execution of m for the different members of S. Observe that with this approach to broadcast, one cannot access a partial response from the group as the replies to all method calls need to be returned before the broadcast succeeds. This restriction would be naturally circumvented

```
List<T> broadcast_{m_ω}(Set<J> S, T_1 x_1, ..., T_n x_n){
    T tmp; J o; List<T> replies;
    if S==∅ {
      replies = Nil;
    } else {
      o = some(S);
      tmp = yield o.m(x̄);
      replies = broadcast_{m_ω}(S\{o}, x_1, ..., x_n);
      replies = Cons(replies,tmp);
    };
    return replies;
}
```

Figure 9: Example of broadcast collecting replies.

in a concurrent object setting with futures (e.g., [6]) rather than with the standard multithreaded concurrency model.

6.2. Channels

Groups can be seen as channels, reminiscent of stubs for remote method calls (RMI); the type system of our kernel language guarantees that the service described by the type of the group is implemented by (at least) one object inside the group. A channel provides an abstraction of who is at either end, but enables communication. There may be more than one sender, and more than one possible receiver. The client makes calls to the group, the receiver is in the group. The group provides anonymity for the receiver end of the channel. The example below illustrates how the receiver in a channel can be dynamically replaced in a way which is transparent to the client, using the kernel language of this paper.

```
Void replace(Service receiver1, Service receiver2, Group(Service) channel){
    receiver2 joins channel as Service;
    try receiver1 leaves channel as Service { skip; } else { skip; };
    return void;
}
```

Channel names are here represented by interface names, and the receiver identity is hidden and may even change dynamically. The two communication models for group communication discussed above, result in different channel behavior. With unicast, only one server will receive the call. With broadcast, all servers will receive the call.

6.3. Roles

One way of managing groups is through different roles, which define available services without specifying the specific implementation of these services. In a type system, roles can be captured through nominal types. Objects may adopt roles or be assigned roles, which associate different nominal types with the object. Roles can be managed at the group level, so an object may have one role in one group and a different role in a different group. Lea [1] distinguishes public and private roles in a group, and relates these to the public and private parts of an object.

In the model of this paper, roles are naturally captured by means of interfaces. These may be dynamically associated with groups. Hence, a group which distinguishes private and public interfaces may be modeled in the kernel language by means of two groups, one which exports public interfaces and another which exports private (group-level) interfaces. In particular, inner groups provide a means to distinguish public calls from intra-group calls. By adding different interfaces to the two groups, an object may provide different services to the group's clients and to the group's members.

Figure 10 illustrates how policies for group membership can be achieved in the kernel language of this paper. In the example, we see how a *group administrator* may create a group with an inner group which functions as a private channel for intra-object communication. In this structure, the inner group has different interfaces from the public group. The two groups are linked via the GroupManager. Whereas publicGroup only makes the methods of the Service1 interface available to clients, the inner group has objects providing the interfaces Administrator, Service1, and Service2. Any object which joins the group publicGroup gets the identity of the inner group privateGroup and may use the inner group for intra-group communication. By modifying the joinGroup method such that the state of the GroupManager determines whether the caller may join the group, the GroupManager can seal a group to prevent new members.

6.4. Group Variants

The following list illustrates the diversity of group usage in object-oriented systems [1]. We relate these to the proposed kernel language and to the discussion in Section 6.1 of unicast and broadcast communication in groups.

- *Subscription Groups*: Groups where all calls from clients are broadcast to all group members, without imposing a restriction on how calls are

```
class GroupManager(Service1 o1, Service2 o2) implements Administrator {
   Group(Service1) publicGroup;
   Group(Administrator,Service2) privateGroup;

   { // Initialization block
     Group(∅) g1; Group(∅) g2;
     g1 = newgroup; g2 = newgroup;
     o1 joins g1 as Service1; publicGroup = g1;
     o2 joins g2 as Service2;
     this joins g2 as Administrator;
     privateGroup = g2;
     g2 joins g1 as Any; return void;
   }

   Group(Administrator,Service1,Service2) joinGroup(Service1 o){
     Group(∅) g1; Group(Administrator,Service2) g2;
     g1 = publicGroup; g2 = privateGroup;
     o joins g1 as Service1;
     o joins g2 as Service1;
     return g2;
   } ...
}
```

Figure 10: A group manager linking an external and an internal group.

handled. This can be modeled in our setting using the broadcast model discussed in Section 6.1 and a group manager that only accepts objects as members if they implement the particular interface.

- *Work Groups*: Groups where the different members provide different interfaces, such that work tasks and subtasks can be distributed among the members of the group. This is directly supported by unicast in the kernel language.

- *Service Groups*: Groups where any member can handle a call from a client. These are directly supported by unicast in the kernel language where all group members implement the same interface, and similar to subscription groups except that they use unicast instead of broadcast.

- *Resource Groups*: Groups of functionally identical services that may be acquired and released by clients. These are unicast groups, similar to subscription and service groups. They differ in that the intention

is that members of a resource group are held by the respective client while the client is using the resource. In the approach of this paper, resources would leave the group when acquired by a client and rejoin the group when released.

- *Access Groups*: Groups where the members have special privileges. These groups may be either based on unicast or broadcast communication. Access groups based on unicast are directly supported by the kernel language, as illustrated by the GroupManager example which grants group members access to the internal group.

- *Replication Groups*: Groups where all members receive the request, typically to ensure fault tolerance. Replication groups are based on broadcast, but they may require that a reply is returned even if some members fail. Replication groups can be achieved by adapting the broadcast mechanism of Section 6.1 to use, e.g., a list of future variables or a shared return variable instead of a list.

- *Transaction Groups*: Groups where the members follow a particular protocol. These are unicast groups and may in principle be encoded in the kernel language by a group manager for an internal group. Transaction groups are not particularly supported by the approach taken in this paper.

- *Property Groups*: Groups where objects are added or removed based on a particular property (for example a location). In the kernel language, this would require a proactive group controller which monitors objects for the particular property. Property groups are not particularly supported by the approach taken in this paper.

7. Related Work

Object orientation is well-suited for designing small units that encapsulate state with behavior, but it does not directly address the organization of more complex software units with rich interfaces. Two approaches to building flexible and adaptive complex software systems involve, independently, object groups and service discovery. Two main uses of groups appear in the literature: *group communication* and *groups as components*. In the first case,

a group is used to facilitate one-to-many communication, and to provide support for, e.g., load balancing. Members of such groups tend to offer the same interface. In the second case, the members of a group typically offer different, complimentary interfaces and the group acts as a means for composition of objects beyond what standard classes and objects offer. Service discovery allows the binding of a client and a service object via an interface, rather than requiring that the two objects know each others' identity. Our work unifies these approaches in a formal, type-safe setting.

The most common use of object groups is to provide replicated services in order to offer better fault tolerance. Communication to members of a group is via multicast. This idea originated in the Amoeba operating system [15]. The component model Jgroup/ARM [16] adopts this idea to provide autonomous replication management using distributed object groups. In this setting, members of a group maintain a replicated state for reasons of consistency. The ProActive active object programming model [17] supports abstractions for object groups, which enable group communication—via method calls—and various means for synchronizing on the results of such method calls. ProActive is formalized in Caromel and Henrio's Theory of Distributed Objects [7]. A core difference with our work is that ProActive's groups lack a notion of service discovery.

Another early work on groups is ActorSpaces [18], which combine Actors with Linda's pattern matching facility, allowing both one-to-one communication, multicast, and querying. Unlike our approach, groups in ActorSpaces are intensional: all actors with the same interface belong to the same group. No formalisation is given, nor is typing discussed.

Object groups have been investigated as a modularization unit for objects which is complementary to components. Groups meet the needs of organizing and describing the statics and dynamics of networks of collaborating objects [1]; groups can have many threads of control, they support roles (or interfaces), and objects may dynamically join and leave groups. Lea [1] presents a number of common usages for groups and discusses their design possibilities, inspired from CORBA. Groups have been used to provide an abstraction akin to a notion of component. For example, in Oracle Siebel 8.2 [19], groups are used as units of deployment, units of monitoring, and units of control when deploying and operating components on Siebel servers. Our approach abstracts from most of these details, though groups are treated as first class entities in our calculus.

ProActive's components [7] are similar to our notion of group in that

they consist of a collection of active objects working together, that objects can be accessed via interface and that communication can be performed in a one-to-one or many-to-one fashion. One key difference is that ProActive relies on client and server ports to connect components, whereas our model lacks ports and instead regular interfaces and service discovery are used to connect objects.

Object groups have further been used for coordination purposes. For example, CoLaS [20] is a coordination model based on groups in which objects may join and leave groups. CoLaS goes beyond the model in our paper by allowing very intrusive coordination of message delivery based on a coordinator state. In our model, the groups do not have any state beyond the state of their objects. Similar to our model, objects enroll to group roles (similar to our interfaces). However, unlike our model, objects may leave a group at any time and the coordinator may access the state of participants. The model is implemented in Smalltalk and neither formalization nor typing are discussed [20].

Concurrent object groups have been proposed to define collaborating objects with a single thread of control in programming and modeling languages [21, 22]. In contrast to our dynamic object groups, these concurrent object groups do not have identity and function as runtime restrictions on concurrency rather than as a linguistic concept.

Microsoft's Component Object Model (COM) allows querying a component to check whether it supports a specific interface, similar to the query-mechanism considered in this paper. A component in COM may also have several interfaces, which are independent of each other. In contrast to the model presented in our paper, COM is not object-oriented and the interfaces of a component are stable (i.e., they do not change). COM has proven difficult (or perhaps, uninteresting) to formalize. Pucella develops λ^{COM} [8], a typed λ-calculus which addresses COM components in terms of their interfaces, and discusses extensions to the calculus to capture subtyping, querying for interfaces, and aggregation.

A wide range of service discovery mechanisms exist [23]. The programming language AmbientTalk [24] has built-in service discovery mechanisms, integrated in an object-oriented language with asynchronous method calls and futures. In contrast to our work, AmbientTalk is an untyped language, and lacks any compile time guarantees. Various works formalise the notion of service discovery [25], but they often do so in a formalism quite far removed from the standard setting in which a program using service discovery would

be written, namely, an object-oriented setting. For example, Fiadeiro et al.'s model of service discovery and binding takes an algebraic and graph-theoretic approach [26], but it lacks the concise operational notion of service discovery formalized in our model. No type system is presented either.

Some systems work has been done that combines groups and service discovery mechanisms, such as group-based service discovery mechanisms in mobile ad-hoc networks [27, 28]. In a sense our approach provides language-based abstractions for such a mechanism, except that ours is also tied to interface types to ensure type soundness and includes a notion of exclusion to filter matched services.

Our earlier work [29] enabled objects to advertise and retract interfaces to which other objects could bind, using a primitive service discovery mechanism. A group mechanism was also investigated as a way of providing structure to the services. In that work services were equated with single objects, whereas in the present work a group service is a collection of objects exporting their interfaces. In particular, this means that the type of a group can change over time as it comes to support more functionality.

The key differences with most of the discussed works is that the model in this paper remains within the object-oriented approach, multiple groups may implement an advertised service in different ways, and our formalism offers a transparent group-based service discovery mechanism with primitive exclusion policies. Furthermore, our notion of groups has an implicit and dynamically changing interface.

8. Conclusion

This paper has proposed a formal model for adaptive service-oriented systems, based on a notion of object-oriented groups. We develop a kernel object-oriented language in which groups are first-class citizens in the sense that they may play the role of objects; i.e., a reference typed by an interface may refer to an object or to a group. A main advantage is that several objects may be collected into a group, thereby obtaining a rich interface reflecting a complex service, which can be seen as a single object from the outside. Although objects in our language are restricted to executing one method activation at the time, the language itself is multithreaded and a group may serve many clients at the same time due to inner concurrency.

In contrast to objects, our groups may dynamically add support for an increasing number of interfaces. Group formation is dynamic; join and leave

primitives allow the migration of services provided by objects and inner groups as well as software upgrade, provided that interfaces are not removed from a group. An object or group may be part of several groups at the same time. This gives a very flexible notion of group. By combining inner groups and release using **yield**, the language allows complex group behavior to be expressed in a simple and elegant way, including inner encapsulation and intra-object communication, access control, and mechanisms for intercepting and filtering messages.

In this paper, dynamic object groups are combined with service discovery by means of **acquire** and **subtypeOf** primitives. This allows a programmer to discover services in an open and unknown environment or in a known group, and to query interface support of a given object or group. These mechanisms are formalized in a general object-oriented setting, based on experiences from a prototype implementation of the group and service discovery primitives in Maude [13]. The presented model provides expressive mechanisms for adaptive services in the setting of object-oriented programming with modest conceptual additions. We have developed an operational semantics and type and effects system for the kernel language, and shown the soundness of the approach by a proof of type safety.

The combination of features proposed in this paper suggests that our notion of a group can be made into a powerful programming concept. The work presented may be further extended in a number of directions. The overall goal of our work has been to study an integration of service-oriented and object-oriented paradigms based on a formal foundation. In this paper, we have explored an approach to groups in the setting of multithread concurrency. However, the **yield** mechanism proposed for release takes inspiration from asynchronous method calls with implicit futures in the context of cooperatively scheduled concurrent objects [30]. To better support groups based on broadcast communication and different ways to collect and use results from an *unknown* number of objects in the group, it would be a natural extension of our work to introduce asynchronous method calls, futures, and cooperative scheduling into the language (e.g., [6, 30]). It is also interesting to study the integration into the kernel language of more service-oriented concepts such as for example error propagation and handling, as well as high-level group management operations such as group aggregation.

Acknowledgement. We thank the anonymous reviewers for their detailed comments, which contributed to a significant improvement of the paper.

References

[1] D. Lea, Objects in Groups, available at `http://gee.cs.oswego.edu/dl/groups/groups.html`, 1993.

[2] J. Bjørk, D. Clarke, E. B. Johnsen, O. Owe, A Type-Safe Model of Adaptive Object Groups, in: N. Kokash, A. Ravara (Eds.), Proc. 11th Intl. Workshop on Foundations of Coordination Languages and Self Adaptation (FOCLASA'12), vol. 91 of *Electronic Proceedings in Theoretical Computer Science*, 1–15, 2012.

[3] A. Igarashi, B. C. Pierce, P. Wadler, Featherweight Java: a minimal core calculus for Java and GJ, ACM Transactions on Programming Languages and Systems 23 (3) (2001) 396–450.

[4] C. Flanagan, S. N. Freund, Type-based race detection for Java, in: M. S. Lam (Ed.), Proceedings Conference on Programming Language Design and Implementation (PLDI), ACM, 219–232, 2000.

[5] J. Östlund, T. Wrigstad, Welterweight Java, in: J. Vitek (Ed.), 48th International Conference on Objects, Models, Components, Patterns (TOOLS), vol. 6141 of *Lecture Notes in Computer Science*, Springer, 97–116, 2010.

[6] F. S. de Boer, D. Clarke, E. B. Johnsen, A Complete Guide to the Future, in: R. de Nicola (Ed.), Proc. 16th European Symposium on Programming (ESOP'07), vol. 4421 of *Lecture Notes in Computer Science*, Springer, 316–330, 2007.

[7] D. Caromel, L. Henrio, A Theory of Distributed Objects - Asynchrony, Mobility, Groups, Components, Springer, 2005.

[8] R. Pucella, Towards a formalization for COM part I: the primitive calculus, in: M. Ibrahim, S. Matsuoka (Eds.), Proceedings of the 2002 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'02), ACM, 331–342, 2002.

[9] J.-P. Talpin, P. Jouvelot, Polymorphic Type, Region and Effect Inference., Journal of Functional Programming 2 (3) (1992) 245–271.

[10] T. Amtoft, H. R. Nielson, F. Nielson, Type and effect systems - behaviours for concurrency, Imperial College Press, 1999.

[11] J. M. Lucassen, D. K. Gifford, Polymorphic effect systems, in: Proceedings of the 15th Symposium on Principles of Programming Languages (POPL'88), ACM Press, 47–57, 1988.

[12] G. D. Plotkin, A structural approach to operational semantics, Journal of Logic and Algebraic Programming 60-61 (2004) 17–139.

[13] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, C. L. Talcott (Eds.), All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic, vol. 4350 of *Lecture Notes in Computer Science*, Springer, 2007.

[14] B. C. Pierce, Types and Programming Languages, The MIT Press, 2002.

[15] M. F. Kaashoek, A. S. Tanenbaum, K. Verstoep, Group communication in Amoeba and its applications, Distributed Systems Engineering 1 (1) (1993) 48–58.

[16] H. Meling, A. Montresor, B. E. Helvik, Ö. Babaoglu, Jgroup/ARM: a distributed object group platform with autonomous replication management, Software: Practice and Experience 38 (9) (2008) 885–923.

[17] L. Baduel, F. Baude, D. Caromel, Efficient, flexible, and typed group communications in Java, in: J. E. Moreira, G. Fox, V. Getov (Eds.), Proc. Joint ACM-ISCOPE Conference on Java Grande, ACM, 28–36, 2002.

[18] G. Agha, C. J. Callsen, ActorSpaces: An Open Distributed Programming Paradigm, in: M. C. Chen, R. Halstead (Eds.), Proceedings of the Fourth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP), ACM, 23–32, 1993.

[19] Oracle Corporation, Siebel Business Applications Documentation, 2010.

[20] J. C. Cruz, S. Ducasse, A Group Based Approach for Coordinating Active Objects, in: P. Ciancarini, A. L. Wolf (Eds.), Third International Conference on Coordination Languages and Models (COORDINATION'99), vol. 1594 of *Lecture Notes in Computer Science*, Springer, 355–370, 1999.

[21] J. Schäfer, A. Poetzsch-Heffter, JCoBox: Generalizing Active Objects to Concurrent Components, in: T. D'Hondt (Ed.), Proc. European Conference on Object-Oriented Programming (ECOOP 2010), vol. 6183 of *Lecture Notes in Computer Science*, Springer, 275–299, 2010.

[22] E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, M. Steffen, ABS: A Core Language for Abstract Behavioral Specification, in: B. Aichernig, F. S. de Boer, M. M. Bonsangue (Eds.), Proc. 9th International Symposium on Formal Methods for Components and Objects (FMCO 2010), vol. 6957 of *Lecture Notes in Computer Science*, Springer, 142–164, 2011.

[23] P. Hasselmeyer, On Service Discovery Process Types, in: B. Benatallah, F. Casati, P. Traverso (Eds.), Proceedings of the Third International Conference on Service-Oriented Computing (ICSOC 2005), vol. 3826 of *Lecture Notes in Computer Science*, Springer, 144–156, 2005.

[24] J. Dedecker, T. V. Cutsem, S. Mostinckx, T. D'Hondt, W. D. Meuter, Ambient-Oriented Programming in AmbientTalk, in: D. Thomas (Ed.), Proc. 20th European Conference on Object-Oriented Programming, (ECOOP'06), vol. 4067 of *Lecture Notes in Computer Science*, Springer, 230–254, 2006.

[25] A. Lapadula, R. Pugliese, F. Tiezzi, Service Discovery and Negotiation With COWS, Electronic Notes in Theoretical Computer Science 200 (3) (2008) 133–154.

[26] J. L. Fiadeiro, A. Lopes, L. Bocchi, An abstract model of service discovery and binding, Formal Aspects of Computing 23 (4) (2011) 433–463.

[27] D. Chakraborty, A. Joshi, Y. Yesha, T. W. Finin, GSD: a novel group-based service discovery protocol for MANETS, in: Proceedings of The Fourth IEEE Conference on Mobile and Wireless Communications Networks, IEEE, 140–144, 2002.

[28] Z. Gao, L. Wang, M. Yang, X. Yang, CNPGSDP: An efficient group-based service discovery protocol for MANETs, Computer Networks 50 (16) (2006) 3165–3182.

[29] D. Clarke, E. B. Johnsen, O. Owe, Concurrent Objects à la Carte, in: D. Dams, U. Hannemann, M. Steffen (Eds.), Concurrency, Composition-

ality, and Correctness, vol. 5930 of *Lecture Notes in Computer Science*, Springer, 185–206, 2010.

[30] E. B. Johnsen, O. Owe, An Asynchronous Communication Model for Distributed Concurrent Objects, Software and Systems Modeling 6 (1) (2007) 35–58.

Appendix A. Proof of Type Preservation

This appendix includes proofs of the lemmas from Section 5.2.

Lemma 1. *Let $m_{\overline{T}\to T} \in C$ be declared in a well-typed program and let o be an object of that program such that classOf$(o) = C$. If $\Gamma \vdash \overline{v} : \overline{T}'$, $\overline{T}' \preceq \overline{T}$, and $T \preceq T'$, then $\Gamma + CT(C) \vdash bind(m_{\overline{T'}\to T'}, o, C, \overline{v}) : T'$.*

Proof. Since *classOf*$(o) = C$, o cannot be null. Since $m_{\overline{T}\to T} \in C$, $bind(m_{\overline{T}\to T}, o, C, \overline{v})$ must give a non-error process $\{l \,|\, \texttt{lock}(1); s; \texttt{return}\ e\}$ such that C has the method $T'\ m(\overline{T_z}\ \overline{z})\,\{\overline{T_{z'}}\ \overline{z}'; s; \texttt{return}\ e;\}$ where l is $[\overline{z} \mapsto (\overline{T_z}, \overline{v}), \overline{z}' \mapsto (\overline{T_{z'}}, \texttt{default}(\overline{T_{z'}})), \texttt{this} \mapsto (C, o)]$. So the local typing context is $l^{\mathcal{T}} = [\overline{z} \mapsto \overline{T_z}, \overline{z}' \mapsto \overline{T_{z'}}, \texttt{this} \mapsto C]$. We need to show that $\Gamma \vdash \{l|\texttt{lock}(1); s; \texttt{return}\, e;\} : T$. Let $\Gamma' = \Gamma + CT(C) + l^{\mathcal{T}}$.

We first show $\Gamma' \vdash l$ $\texttt{ok}$. By assumption, $\Gamma \vdash \overline{v} : \overline{T}$, and $\Gamma(o) = C$. Thus $\overline{v} : \overline{T_z}$ since $\overline{T} \preceq \overline{T_z}$. Since $\overline{v}, o$ are values, $\Gamma' \vdash \overline{v} : \overline{T_z}$ and $\Gamma'(o) = C$. Since $\Gamma'(\overline{z}) = \overline{T_z}$ and $\Gamma'(\overline{v}) = \overline{T_z}$, by RTT-Sub and RTT-Subs we have $\Gamma' \vdash \overline{z} \mapsto (\overline{T_z}, \overline{v})$ $\texttt{ok}$. By RTT-Def, $\Gamma' \vdash \texttt{default}(\overline{T_{z'}}) : \overline{T_{z'}}$, so by RTT-Sub and RTT-Subs we have $\Gamma' \vdash \overline{z'} \mapsto (\overline{T_{z'}}, \texttt{default}(\overline{T_{z'}}))$ $\texttt{ok}$. Since $\Gamma'(\texttt{this}) = C$ and $\Gamma'(o) = C$, by RTT-Sub we have $\Gamma' \vdash \texttt{this} \mapsto (C, o)$ $\texttt{ok}$, and by RTT-Subs $\Gamma' \vdash l$ $\texttt{ok}$ follows by composition.

We next show $\Gamma' \vdash \texttt{lock}(1); s; \texttt{return}\ e : T$. Since m is well-typed in C we have from T-Class, T-Method, and RTT-Lock that $\Gamma[\texttt{this} \mapsto C] + CT(C) + [\overline{z} \mapsto \overline{T_z}, \overline{z'} \mapsto \overline{T_{z'}}] \vdash \texttt{lock}(1); s; \texttt{return}\ e : T'$. Since $l^{\mathcal{T}} = \overline{z} \mapsto \overline{T_z}, \overline{z'} \mapsto \overline{T_{z'}}, \texttt{this} \mapsto C$, we get $\Gamma + CT(C) + l^{\mathcal{T}} \vdash \texttt{lock}(1); s; \texttt{return}\ e : T'$. By RTT-Frame1 we get $\Gamma \vdash \{l|\texttt{lock}(1); s; \texttt{return}\ e\} : T$, since $T' \preceq T$. □

Lemma 2. *Let P be a program such that $\Gamma \vdash P$ $\texttt{ok}$ and let cn be the initial configuration of P. Then there is a Γ' such that $\Gamma' \vdash cn$ $\texttt{ok}$ and $\Gamma \subseteq \Gamma'$.*

Proof. Let us assume that $P = \overline{IF}\ \overline{CL}\ \{\overline{T}\ \overline{z}; sr;\}$, then the initial state is $o(\varepsilon, (t, 1))\ t([\{\overline{z} \mapsto (\overline{T}, \texttt{default}(\overline{T})), \texttt{this} \mapsto (C, o)]|sr;\}; \texttt{idle})$ for fresh

names o, t, and C. By assumption, $C \prec \mathsf{Any}$. We assume that $CT(C) = \varepsilon$. Let $\Gamma' = \Gamma[o \mapsto C, t \mapsto \mathsf{Thread}]$.

Obviously, by RTT-Object, $\Gamma' \vdash o(\varepsilon, (t, 1))$ $\mathtt{ok}$. By T-Program, $\Gamma'[\overline{z} \mapsto \overline{T}] \vdash sr : \mathsf{Void}$. By RTT-Def, $\Gamma'[\overline{z} \mapsto \overline{T}] \vdash z_i \mapsto (T_i, \mathtt{default}(T_i))$. By RTT-Sub, $\Gamma'[\mathtt{this} \mapsto (C, o)] \vdash \mathtt{this} \mapsto (C, o)$ $\mathtt{ok}$. Then by RTT-Subs, RTT-Frame1, RTT-Stack, RTT-Thread1, and RTT-Config, $\Gamma' \vdash cn$ $\mathtt{ok}$. □

Lemma 3. *If* $\Gamma \vdash cn$ $\mathtt{ok}$ *and* $cn \to cn'$ *then there is a* Γ' *such that* $\Gamma' \vdash cn'$ $\mathtt{ok}$ *and* $\Gamma \subseteq \Gamma'$.

Proof. The proof is by cases over the reduction rules of the semantics. To help readability, we let σ denote the typing context $CT(\Gamma(\mathtt{this}))$, i.e., the declared types of the fields of the current object. We use the notation $\Gamma \vdash s$ $\mathtt{ok}$ instead of $\Gamma \vdash s\ \Delta$ when the effect Δ is uninteresting.

Case Skip. Let $cn = cn_0\ t(\{l|\mathtt{skip}; sr; \}; \rho)$ and $cn' = cn_0\ t(\{l|sr; \}; \rho)$. By assumption, $\Gamma \vdash cn$ $\mathtt{ok}$, so $\Gamma \vdash cn_0$ $\mathtt{ok}$ and $\Gamma \vdash t(\{l|\mathtt{skip}; sr; \}; \rho)$ $\mathtt{ok}$. Then, by RTT-Thread1, RTT-Stack, RTT-Frame1, RTT-Frame2, and T-Composition, we know that $\Gamma \vdash t(\{l|sr; \}; \rho)$ $\mathtt{ok}$, and $\Gamma \vdash cn'$ $\mathtt{ok}$.

Case While. Let $cn = cn_0\ o(a, \delta)\ t(\{l|\mathtt{while}\, e\, \{s; \}; sr; \}; \rho)$ and $cn' = cn_0\ o(a, \delta)\ t(\{l|\mathtt{if}\, e\, \{s; \mathtt{while}\, e\, \{s; \}\}\, \mathtt{else}\, \{\mathtt{skip}\}; sr; \}; \rho)$. By assumption, $\Gamma \vdash cn$ $\mathtt{ok}$, so $\Gamma + \sigma + l^{\mathcal{T}} \vdash e : \mathsf{Bool}$ and $\Gamma + \sigma + l^{\mathcal{T}} \vdash s$ $\mathtt{ok}$. It follows that $\Gamma + \sigma + l^{\mathcal{T}} \vdash \mathtt{if}\, e\, \{s; \mathtt{while}\, e\, \{s; \}\}\, \mathtt{else}\, \{\mathtt{skip}\}$ $\mathtt{ok}$, and we get that $\Gamma \vdash cn'$ $\mathtt{ok}$.

Case Cond1. Let $cn = cn_0\ o(a, \delta)\ t(\{l|\mathtt{if}\, e\, \{s_1; \}\, \mathtt{else}\, \{s_2; \}; sr; \}; \rho)$ and $cn' = cn_0\ o(a, \delta)\ t(\{l|s_1; sr; \}; \rho)$. By assumption, $\Gamma \vdash cn$ $\mathtt{ok}$, so $\Gamma + \sigma + l^{\mathcal{T}} \vdash s_1$ $\mathtt{ok}$ and $\Gamma + \sigma + l^{\mathcal{T}} \vdash sr$ $\mathtt{ok}$. Then, $\Gamma + \sigma + l^{\mathcal{T}} \vdash s_1; sr$ $\mathtt{ok}$ and it follows that $\Gamma \vdash cn'$ $\mathtt{ok}$.

Case Cond2. Let $cn = cn_0\ o(a, \delta)\ t(\{l|\mathtt{if}\, e\, \{s_1; \}\, \mathtt{else}\, \{s_2; \}; sr; \}; \rho)$ and $cn' = cn_0\ o(a, \delta)\ t(\{l|s_2; sr; \}; \rho)$. By assumption, $\Gamma \vdash cn$ $\mathtt{ok}$, so $\Gamma + \sigma + l^{\mathcal{T}} \vdash s_2$ $\mathtt{ok}$ and $\Gamma + \sigma + l^{\mathcal{T}} \vdash sr$ $\mathtt{ok}$. Then, $\Gamma + \sigma + l^{\mathcal{T}} \vdash s_2; sr$ $\mathtt{ok}$ and it follows that $\Gamma \vdash cn'$ $\mathtt{ok}$.

Case Assign1. Let $cn = cn_0\ o(a, \delta)\ t(\{l|x = e; sr; \}; \rho)$ and $cn' = cn_0\ o(a, \delta)\ t(\{l[x \mapsto (T, v)]|sr; \}; \rho)$, where $l^{\mathcal{T}}(x) = T$ and $(a + l)^{\mathcal{V}}(e) = v$. By assumption, $\Gamma \vdash cn$ $\mathtt{ok}$, so $\Gamma + \sigma + l^{\mathcal{T}} \vdash x = e$ $\mathtt{ok}$. Since $\Gamma \vdash o(a, \delta)$ $\mathtt{ok}$ and $\Gamma \vdash t(\{l|x = e; sr; \}; \rho)$ $\mathtt{ok}$, we know that $\Gamma + \sigma + l^{\mathcal{T}} \vdash v : T$. Then $\Gamma + \sigma + l^{\mathcal{T}} \vdash x \mapsto (T, v)$ $\mathtt{ok}$, so $\Gamma \vdash t(\{l[x \mapsto (T, v)]|sr; \}; \rho)$ $\mathtt{ok}$, and finally $\Gamma \vdash cn'$ $\mathtt{ok}$.

Case Assign2. Let $cn = cn_0\ o(a, \delta)\ t(\{l|x = e; sr; \}; \rho)$ and $cn' = cn_0\ o(a[x \mapsto (T, v)], \delta)\ t(\{l|sr; \}; \rho)$, where $\sigma(x) = T$ and $(a + l)^{\mathcal{V}}(e) = v$. By

assumption, $\Gamma \vdash cn$ $\mathtt{ok}$, so $\Gamma + \sigma + l^{\mathcal{T}} \vdash x = e$ $\mathtt{ok}$. Since $\Gamma \vdash o(a,\delta)$ $\mathtt{ok}$ and $\Gamma \vdash t(\{l|x = e; sr;\}; \rho)$ $\mathtt{ok}$, we know that $\Gamma + \sigma + l^{\mathcal{T}} \vdash v : T$. Then $\Gamma + \sigma + l^{\mathcal{T}} \vdash x \mapsto (T, v)$ $\mathtt{ok}$, so $\Gamma \vdash o(a[x \mapsto (T,v)], \delta)$ $\mathtt{ok}$. Since $\Gamma \vdash t(\{l|sr;\}; \rho)$ $\mathtt{ok}$, we get $\Gamma \vdash cn'$ $\mathtt{ok}$.

Case Lock-Object. Let $cn = cn_0\ o(a,\delta)\ t(\{l|\mathtt{lock}(n); sr;\}; \rho)$ and $cn' = cn_0\ o(a,\delta')\ t(\{l|sr;\}; \rho)$. By assumption, $\Gamma \vdash cn$ $\mathtt{ok}$, so $\Gamma + \sigma \vdash a$ $\mathtt{ok}$ and $\Gamma + \sigma + l^{\mathcal{T}} \vdash \mathtt{lock}(n); sr : T$ for some T. Since $\Gamma \vdash \delta$ $\mathtt{ok}$, we get from RTT-ObjectLock that $\delta' = inc_t(1, \delta) = (t, n)$ for some n, so $\Gamma \vdash \delta'$ $\mathtt{ok}$ and it follows that $\Gamma \vdash cn'$ $\mathtt{ok}$.

Case New-Group. Let $cn = cn_0\ o(a,\delta)\ t(\{l|x = \mathtt{newgroup}; sr;\}; \rho)$ and $cn' = cn_0\ o(a,\delta)\ t(\{l|x = g; sr;\}; \rho)\ g(\emptyset)$. By assumption, $\Gamma \vdash cn$ $\mathtt{ok}$, so $\Gamma + \sigma + l^{\mathcal{T}} \vdash x = \mathtt{newgroup}; sr : T$. In particular, by T-Assign, $\Gamma + \sigma + l^{\mathcal{T}} \vdash x : \mathsf{Group}(\emptyset)$. Since g is fresh, $g \notin dom(\Gamma)$. Let $\Gamma' = \Gamma[g \mapsto \mathsf{Group}(\emptyset)]$. Then $\Gamma' \vdash cn_0\ o(a,\delta)$ $\mathtt{ok}$, $\Gamma' + \sigma + l^{\mathcal{T}} \vdash x = g$ $\mathtt{ok}$, and $\Gamma' + \sigma + l^{\mathcal{T}} \vdash sr : T$. It follows that $\Gamma' \vdash t(\{l|x = g; sr;\}; \rho)$ $\mathtt{ok}$. By RTT-Group, $\Gamma' \vdash g(\emptyset)$ $\mathtt{ok}$, and it follows that $\Gamma' \vdash cn'$ $\mathtt{ok}$.

Case New-Thread. Let $cn = cn_0\ o(a,\delta)\ t(\{l|\mathtt{spawn}\, x.m(\overline{e}); sr;\}; \rho)$ and $cn' = cn_0\ o(a,\delta)\ t(\{l|sr;\}; \rho)\ t'(pr; \mathtt{idle})$. Let $\Gamma' = \Gamma + \sigma + l^{\mathcal{T}}$ and $\Gamma'' = \Gamma[t' \mapsto \mathsf{Thread}]$. By assumption, $\Gamma + \sigma \vdash \{l|\mathtt{spawn}\, x.m(\overline{e}); sr;\} : T$, so $\Gamma + \sigma \vdash \{l|sr;\} : T$ and by T-Spawn we have $\Gamma'(x) \preceq m_{\Gamma'(\overline{e})} \to \mathsf{Void}$. By Lemma 1 we then get $\Gamma \vdash pr : T$, and by RTT-Stack $\Gamma \vdash pr; \mathtt{idle}$ $\mathtt{ok}$. We have $\Gamma'' \vdash t'(pr; \mathtt{idle})$ $\mathtt{ok}$ and $\Gamma'' \vdash cn'$ $\mathtt{ok}$ follows.

Case New-Object. Let $cn = cn_0\ o(a,\delta)\ t(\{l|x = \mathtt{new}\ C(\overline{e}); sr;\}; \rho)$ and $cn' = cn_0\ o(a,\delta)\ t(\{l|x = o'; sr;\}; \rho)\ o'(a', (t', 1))\ t'(pr; \mathtt{idle})$, where $pr = \{\overline{z} \mapsto (\overline{T}, \mathtt{default}(\overline{T})), \mathtt{this} \mapsto (C, o')|sr';\}$ where $\overline{z}$ of type $\overline{T}$ are the local variables of the initiator of C. By assumption, $\Gamma \vdash cn$ $\mathtt{ok}$, so $\Gamma + \sigma + l^{\mathcal{T}} \vdash x = \mathtt{new}\ C(\overline{e}); sr : T$. From T-New, we can assume that $\Gamma(x) = I$ such that $C \prec I$, and $\Gamma + \sigma + l^{\mathcal{T}} \vdash \overline{e} : ptypes(C)$. Since o' is fresh, $o' \notin dom(\Gamma)$. Let $\Gamma' = \Gamma[o \mapsto C]$. Then $\Gamma' \vdash cn_0\ o(a,\delta)$ $\mathtt{ok}$. We need to show that t, t', and o' are well-typed in Γ'.

It follows from the induction hypothesis that $\Gamma' + \sigma + l^{\mathcal{T}} \vdash x = o'$ $\mathtt{ok}$ and consequently $\Gamma' + \sigma + l^{\mathcal{T}} \vdash x = o'; sr : T$. Then, $\Gamma' + \sigma \vdash \{l|x = \mathtt{new}\, C(\overline{e}); sr;\} : T$ and t is well-typed in Γ'. Since $\Gamma + \sigma \vdash a$ $\mathtt{ok}$ and $\Gamma + \sigma + l^{\mathcal{T}} \vdash l$ $\mathtt{ok}$, we have that $\Gamma + \sigma + l^{\mathcal{T}} \vdash (a + l)^{\mathcal{V}}(\overline{e}) : ptypes(C)$. It follows that $\Gamma' \vdash atts(C, (a + l)^{\mathcal{V}}(\overline{e}))$ $\mathtt{ok}$ and o' is well-typed in Γ'. Since C is well typed in Γ', we have $\Gamma' + CT(C)[\overline{z} \mapsto \overline{T}] \vdash sr' : \mathsf{Void}$. It follows that $\Gamma' \vdash pr$ $\mathtt{ok}$ and the thread t' is well-typed in Γ'.

Case Call1. Let $cn = cn_0\ o(a,\delta)\ t(\{l|x = y.m_\omega(\overline{e}); sr;\}; \rho)$ and $cn' =$

$cn_0\ o(a,\delta)\ t(pr; \{l|\texttt{block}(x); sr; \}; \rho)$. By assumption, $\Gamma \vdash \{l|sr\}; \rho\ \texttt{ok}$. Let Γ' denote $\Gamma + \sigma + l^{\mathcal{T}}$ and assume that $\Gamma'(x) = T$. By assumption, $\Gamma \vdash cn\ \texttt{ok}$, so by T-Call we have $\Gamma'(y) \preceq m_{\Gamma'(\overline{e})\to T}$. By Lemma 1 we then get $\Gamma \vdash pr : T$, and it follows from RTT-Stack that $\Gamma \vdash pr; \{l|\texttt{block}(x); sr\}; \rho\ \texttt{ok}$.

Case Call2. By RTT-ObjectLock, $\Gamma \vdash (t, n)\ \texttt{ok}$. The case is then similar to *Case* Call1.

Case Call3. Let $cn = cn_0\ o(a,\delta)\ t(\{l|x = y.m_\omega(\overline{e}); sr; \}; \rho)\ g(export)$ and $cn' = cn_0\ o(a,\delta)\ t(\{l|x = v.m_\omega(\overline{e}); sr; \}; \rho)\ g(export)$. Let Γ' denote $\Gamma + \sigma + l^{\mathcal{T}}$. By assumption, $\Gamma \vdash cn\ \texttt{ok}$, so by T-Call we have $\Gamma' \vdash x = y.m(\overline{e})\ \texttt{ok}$ where $\Gamma'(y) \preceq m_{\Gamma'(\overline{e})\to\Gamma'(x)}$. Since $v : I \in export$ we know by RTT-Exp that $\Gamma'(v) \preceq I$. Hence, $\Gamma' \vdash v : I$ and $\Gamma' \vdash x = v.m(\overline{e})\ \texttt{ok}$. It follows that $\Gamma \vdash cn'\ \texttt{ok}$.

Case Join. Let $cn = cn_0\ o(a,\delta)\ t(\{l|x\ \texttt{joins}\ z\ \texttt{as}\ \overline{I}; sr; \}; \rho)\ g(export)$ and $cn' = cn_0\ o(a,\delta)\ t(\{l[z \mapsto (T, g)]|sr; \}; \rho)\ g(export')$, where $T = \mathsf{Group}(S \cup \overline{I})$. By assumption, $\Gamma \vdash cn\ \texttt{ok}$, so $\Gamma + \sigma + l^{\mathcal{T}} \vdash x\ \texttt{joins}\ z\ \texttt{as}\ \overline{I}; sr : T$. Assume that $l(z) = (\mathsf{Group}(S), g)$ and that $(a + l)^{\mathcal{V}}(x) = o'$. By T-Join, $(\Gamma + \sigma + l^{\mathcal{T}})(x) \preceq \overline{I}$, and since $\Gamma + \sigma + l^{\mathcal{T}} \vdash l\ \texttt{ok}$, we know that $\Gamma(o') \preceq \overline{I}$. It follows from T-Join and T-Composition that $\Gamma + \sigma + l^{\mathcal{T}}[z \mapsto \mathsf{Group}(T)] \vdash sr : T$. Let $export' = \bigcup_{I \in \overline{I}} \{o' : I\} \cup export$. By assumption, $\Gamma \vdash export\ \texttt{ok}$ and since $\Gamma(o') \preceq \overline{I}$ we know by RTT-Exp that $\Gamma \vdash g(export')\ \texttt{ok}$. It follows that $\Gamma \vdash cn'\ \texttt{ok}$.

Case Return. Let $cn = cn_0\ o(a,\delta)\ t(\{l|\texttt{return}\ e\}; \{l'|\texttt{block}(y); sr; \}; \rho)$ and $cn' = cn_0\ o(a,\delta')\ t(\{l'|y = v; sr; \}; \rho)$. By assumption, $\Gamma \vdash cn\ \texttt{ok}$, so by RTT-Stack $\Gamma + \sigma + l^{\mathcal{T}} \vdash e : T$ for some T such that $\Gamma[\texttt{block} \mapsto T] \vdash \{l'|\texttt{block}(y); sr; \}; \rho\ \ \texttt{ok}$. Since $\Gamma + \sigma \vdash a\ \texttt{ok}$ and $\Gamma + \sigma + l^{\mathcal{T}} \vdash l\ \texttt{ok}$, we have that $\Gamma + \sigma + l^{\mathcal{T}} \vdash v : T$. It follows that $\Gamma \vdash \{l'|y = v; sr; \}; \rho\ \ \texttt{ok}$. Since $dec_t(\delta)$ is well-defined, either $\delta' = \texttt{free}$, which is well-typed by RTT-Free, or $\delta = (t, n + 1)$ which is well-typed by RTT-ObjectLock since $\Gamma(t) = \mathsf{Thread}$ by assumption. It follows that $\Gamma \vdash cn'\ \texttt{ok}$.

Case End. Follows directly from the induction hypothesis.

Case Leave1. Let $cn = cn_0\ o(a,\delta)\ t(\{l|\texttt{try}\ x\ \texttt{leaves}\ y\ \texttt{as}\ \overline{I}\ \{s_1\}\ \texttt{else}\ \{s_2\}; sr; \}; \rho)\ g(export)$ and $cn' = cn_0\ o(a,\delta)\ t(\{l|s_1; sr; \}; \rho)\ g(export')$. By assumption, $\Gamma \vdash cn\ \texttt{ok}$, so $\Gamma \vdash g(export)\ \texttt{ok}$ and $\Gamma + \sigma \vdash \{l|\texttt{try}\ x\ \texttt{leaves}\ y\ \texttt{as}\ \overline{I}\ \{s_1\}\ \texttt{else}\ \{s_2\}; sr; \} : T$. It is obvious that $\Gamma + \sigma \vdash \{l|s_1; sr; \} : T$ and that $\Gamma \vdash g(export')\ \texttt{ok}$ where $export' \subseteq export$. Since $export' \preceq \overline{I}$, we know that if $\Gamma \vdash g : \overline{I}$ with $g(export)$ then $\Gamma \vdash g : \overline{I}$ still holds with $g(export')$. It follows that $\Gamma \vdash cn\ g(export')\ \texttt{ok}$.

Case LEAVE2. Follows directly from the induction hypothesis.

Case QUERY1. Let $cn = cn_0\ o(a,\delta)\ t(\{l|\texttt{try}\ z\ \texttt{subtypeOf}\ I\ \{s_1\}\ \texttt{else}\ \{s_2\}; sr;\};\rho)\ g(export)$ and $cn' = cn_0\ o(a,\delta)\ t(\{l[z \mapsto (\mathsf{Group}(S \cup \{I\}, g))]|s_1; sr;\};\rho)\ g(export)$. By assumption, $\Gamma \vdash cn\ \texttt{ok}$, so $\Gamma + \sigma \vdash \{l|\texttt{try}\ z\ \texttt{subtypeOf}\ I\ \{s_1\}\ \texttt{else}\ \{s_2\}; sr;\} : T$. Let $(l)^{\mathcal{T}}(z) = \mathsf{Group}(S)$. By T-INSPECT, we know that $\Gamma + \sigma + l^{\mathcal{T}}[z \mapsto \mathsf{Group}(S \cup I)] \vdash s_1\ \texttt{ok}$, and consequently $\Gamma + \sigma + l^{\mathcal{T}}[z \mapsto \mathsf{Group}(S \cup I)] \vdash s_1; sr : T$. It follows that $\Gamma \vdash cn'\ \texttt{ok}$.

Case QUERY2. Follows directly from the induction hypothesis.

Case ACQUIRE1. Let $cn = cn_0\ o(a,\delta)\ t(\{l|\texttt{try}\ x = \texttt{acquire}\ I\ \texttt{in}\ y\ \texttt{except}\ \bar{x}; \{s_1\}\ \texttt{else}\ \{s_2\}; sr;\};\ \rho)\ g(export)$ and $cn' = cn_0\ o(a,\delta)\ t(\{l|x = v; s_1; sr;\};\rho)\ g(export)$, where $(v : J) \in export$. By assumption, $\Gamma \vdash cn\ \texttt{ok}$, so $\Gamma + \sigma \vdash \{l|\texttt{try}\ x = \texttt{acquire}\ I\ \texttt{in}\ y\ \texttt{except}\ \bar{x}; \{s_1\}\ \texttt{else}\ \{s_2\}; sr;\} : T$ and $\Gamma \vdash g((v : J) \cup export)\ \texttt{ok}$. It follows from RTT-EXP that $\Gamma(v) \preceq J$ and since $J \preceq I$, we have $\Gamma + \sigma \vdash \{l|x = v; s_1; sr;\} : T$ and consequently $\Gamma \vdash cn'\ \texttt{ok}$.

Case ACQUIRE2. Follows directly from the induction hypothesis. □

Lemma 4. *Consider a well-typed configuration containing a thread* $t(\{l|sr\};\rho)$ *and an object* $o(\sigma,\delta)$*, such that* $l^{\mathcal{V}}(\texttt{this}) = o$ *and* $\delta(t)$*. If* sr *is null-free in the state* $(\sigma + l)$*, there will be exactly one applicable operational rule modifying the thread.*

Proof. The proof is by cases over the first statement of the statement list sr, which may be the **return** statement. We may assume that sr is well-typed. In our core language a well-typed expression e can be evaluated to a well-defined value, using $(\sigma + l)^{\mathcal{V}}(e)$. If we were to consider errors in expressions, we would need a strengthened notion of null-free, so that we could assume error-free expressions in the proof.

Case **skip**. When sr starts with **skip**, the SKIP rule can be applied since there are no premises on the rule.

Case **while**. When sr starts with a **while** statement, the WHILE rule can be applied since there are no premises on the rule.

Case **if** statement. A well-typed Boolean expression (in our core language) will evaluate to true or false. In either case, one rule will apply.

Case z=e. Rule ASSIGN1 applies. (Here null-freeness is not needed.)

Case f=e. Rule ASSIGN2 applies. (Again null-freeness is not needed.)

Case lock(n) statement. The LOCK-OBJECT rule can be applied since $\delta(t)$ implies that $inc_t(n,\delta)$ is well-defined.

Case **newgroup**, **spawn**, and **new** object. The corresponding rule can be applied, given that fresh names (of each category) can be generated. For **spawn**, the callee is not null by the assumption of null-freeness.

Case method call 1: a normal call on an object. By null-freeness, the callee is not null and Rule CALL1 applies.

Case method call 2: a yielding call on an object. By null-freeness, the callee is not null and Rule CALL2 applies.

Case method call 3: a call on a group. By null-freeness, the callee is a non-null group, say $(\mathsf{Group}(S), g)$, and by well-typedness of the call, the callee supports an interface I with a method m of the function type ω generated during typing. Since the configuration is well-typed $(\sigma + l)^{\mathcal{V}}(g)$ must contain an object v of type (better than) I. Thus the premises of Rule CALL3 can be satisfied, and the rule can be applied.

Case x **joins** z **as** $\overline{I}$. By null-freeness and well-typedness, z is a non-null group, and rule JOIN applies.

Case **return**. Observe that in a well-typed configuration, a stack consists of an active process at the top of the stack and of (suspended) frames below the active process such that the frame at the bottom of the stack is `idle`. A **return** may only occur as the last statement in the active process or in a frame on the stack. The active frame cannot contain a `block` statement, whereas a frame below the active process must begin with a `block` statement or it is `idle`. Thus, it suffices to consider these two cases for **return** and the rules RETURN or END will apply, respectively. ($\delta(t)$ implies that $dec_t(\delta)$ is defined.)

Case x= **acquire** l **in** y **except** $\overline{x}$. By null-freeness and well-typedness, y will bind to a group. This group may or may not have an object exporting I (other than objects in $\overline{x}$). Rule ACQUIRE1 and rule ACQUIRE2 cover the two cases.

Case x **leaves** y **as** $\overline{I}$. By null-freeness and well-typedness, y will bind to a group. The group may or may not have objects exporting each of $\overline{I}$ (ignoring the object x). Rule LEAVE1 and rule LEAVE2 cover the two cases.

Case z **subtypeOf** l. By null-freeness and well-typedness, z will bind to a group. The group may or may not export I. Rule QUERY1 and rule QUERY2 cover the two cases. □