Coordination Requirements Expressed in Types for Active Objects

Franz Puntigam

Technische Universität Wien, Institut für Computersprachen
Argentinierstraße 8, 1040 Vienna, Austria. E-mail: franz@complang.tuwien.ac.at

Abstract. An object's type is usually regarded as a contract between the object and each of its users. However, in concurrent (and sometimes also in sequential) systems it is more useful to regard a type as a contract between an object and the unity of all users: The users must be coordinated before sending messages to the object. Types in the proposed model express requirements on the coordination of users; objects must accept messages only in pre-specified orders. The model ensures statically that objects behave as specified by their types, and users are coordinated appropriately.

Keywords: Type model, concurrency, active objects.

1 Introduction

Each expression written in a statically typed programming language has a unique type specified explicitly or derivable at compile-time. Strong typing ensures that violations of type constraints (type errors) cannot occur during program execution [4]. Static and strong typing can increase the readability and reliability of programs and support optimizations. In the object-oriented paradigm, types specify contracts between objects (servers) and their users (clients) [14]. These contracts play an important role in the maintenance and reuse of software.

An object's type is usually viewed as the object's signature, often associated with a name or another entity used as (informal) description of the object's behavior. In some languages, programmers can formally express their expectations on the input and result parameters and give partial specifications of the behavior using assertions (preconditions, postconditions and invariants). Users get the promised results if they call the object's methods when the preconditions are satisfied. This type concept is quite useful for a large class of applications.

1.1 The Problem

Sometimes, methods shall be called only in certain circumstances depending on the object's current state and history. For example, let "iconify" be a method that replaces a window on a screen with an icon. Preconditions seem to be appropriate for specifying that "iconify" shall be called only when the window is displayed [13]. However, preconditions have limitations: They are not "history

sensitive" [12] and cannot always be checked statically, loosing the advantages of strongly typed languages.

In concurrent and distributed systems it is difficult for a user to know an object's state even at run-time; other users may cause unpredictable state changes. Unexpected effects may occur even if all preconditions are satisfied: For example, two concurrent users send "iconify" at about the same time to a displayed window. The window is replaced with an icon when the first message is handled. A further user sees the new state and sends a message causing the window to be displayed again. Then, the window is immediately replaced by an icon again when the second iconify-message is handled. This behavior is probably unexpected because one of the iconify-messages is dealt with in a context different from the one in which the message was sent.

There are several ways to prevent that messages are dealt with in wrong contexts. For example, the programmer implements a protocol ensuring that only one client can send "iconify" to a displayed window. But there is no support from the type system. A type in current type models is a contract between each individual user and an object, but not between the whole set of users and the object. It is not possible to express in the type that users must be coordinated before sending messages like "iconify". There is the implicit assumption that each object must be able to handle all supported messages from all users in arbitrary interleaving. As the windows example shows, this assumption can be too restrictive in practice.

1.2 The Proposal

A type model addressing this problem is proposed in the present paper. The type model is based on a process calculus in which each object has a unique identifier (mail address), a behavior and a queue of received messages, as in the actor model [1, 7]. Objects communicate by asynchronous message passing. Some actions in the calculus are annotated with type information. A type specifies all possible sequences of messages accepted by an object as well as type constraints on the messages' parameters. A type checker (or compiler) shall be able to ensure statically that

- objects can deal with all message sequences specified by the objects' types;
- users of an object are coordinated so that only messages specified by the object's type are sent to the object in an expected order.

In a type-consistent program, all objects can deal properly with all received messages. There are no "message-not-understood-errors", unintended behaviors or deadlocks caused by wrong messages or wrong message orders. (The type model cannot prevent all kinds of deadlocks. But it can prevent that an object blocks with a nonempty message queue, waiting for messages not in the queue.)

The messages each user can send to an object are controlled by *type marks*. A type mark allows a user to send some messages. Sometimes, the combination of several type marks allows a user to send a message, while one of these type marks

alone is not sufficient for that. Type marks are passed to other users as side-effects of sending messages. The programmer must provide code for coordinating the users by passing messages; the compiler checks if the users actually have all type marks they need.

The proposed type model supports subtyping and genericity. According to the principle of substitutability, an instance of a subtype can be used wherever an instance of a supertype is expected [11, 29]. A subtype extends a supertype by supporting additional messages and message orders so that each message accepted by an object of a supertype is also accepted by an object of a subtype.

In Sect. 2 we describe the kind of object systems considered in this paper and work out important requirements on types in these systems. In Sect. 3 we introduce types that can express coordination requirements and discuss some of their properties. In Sect. 4 we outline a type checking algorithm. An example in Sect. 5 shows how the type model can be used. A comparison with related work follows in Sect. 6. An extended version of this paper is available as a technical report [24].

2 Active Objects and Type Consistency

We consider systems composed of active objects that communicate through asynchronous message passing. An active object has its own (single) thread of execution, a behavior, a unique identifier (used as mail address), and an unlimited buffer of received messages. According to its behavior, an object can accept messages from its buffer, send messages to other objects, and create new objects. All messages are received and accepted in the same (logical) order they were sent; message buffers operate in a first-in-first-out manner. These restrictive assumptions make it easier to show how the type model works.

A program specifies the behavior of all objects in an object system. We assume that programs are written in a language based on a process calculus like those explored by Hoare [8], Milner [15, 16, 17] and others [2, 6, 9]. These calculi provide a theoretically well-founded and expressive basis for specifying the behavior of active objects. But they have to be adapted so that all messages sent to some address are handled sequentially by a single object, and atomic actions carry type information. The used calculus shall be simple enough to show static type checking, and expressive enough to support all important language concepts.

2.1 The Process Calculus

A *process* specifies the behavior of an object. Fig. 1 shows the syntax of processes denoted by θ. An object with behavior 0, the zero process, does nothing. There are three atomic actions for sending messages, accepting messages and creating new objects. Semicolons separate actions from the processes which shall be executed after the actions. A message consists of a constant name c (the message selector) and a list of arguments $a_1, \ldots, a_n$. (We write a line over an expression as

$\theta ::= 0$	(zero process; no action)
$\mid x.c[\overline{a}];\theta$	(send message c with arguments $\overline{a}$ to x; then execute θ)
$\mid c(\overline{x})^{\overline{\varphi}};\theta$	(accept message c with parameters $\overline{x}$ of types $\overline{\varphi}$; then θ)
$\mid (x)\$a^{[\overline{\sigma}]}[\overline{a}'];\theta$	(create new object x that executes $a^{[\overline{\sigma}]}[\overline{a}']$; then θ)
$\mid a{=}a'\,?\,\theta:\theta'$	(execute θ if $a = a'$; otherwise execute θ')
$\mid \theta + \theta'$	(alternatives; execute either θ or θ')
$\mid a^{[\overline{\sigma}]}[\overline{a}']$	(call a with type arguments $\overline{\sigma}$ and arguments $\overline{a}'$)
$a ::= x$	(parameter or object identifier)
$\mid {}^{(\overline{s})}(\overline{x})^{\overline{\varphi}}\theta^{\sigma}$	(closed process; does not contain free variable names)

Fig. 1. Syntax of Processes

an abbreviation of an indexed list of expressions; e.g., $\overline{a}$ stands for $a_1, \ldots, a_n$. Superscripts represent type annotations.) An accepting action $c(x_1, \ldots, x_n)^{\varphi_1, \ldots, \varphi_n}$ is executable only if the first message in the buffer of received messages has the selector c and arguments $a_1, \ldots, a_n$, where each a_i satisfies the type constraints of φ_i. The arguments are substituted for the parameters $x_1, \ldots, x_n$. All parameters will be replaced with arguments. Arguments are either object identifiers or *closed processes* which specify object behavior.

A closed process ${}^{(\overline{s})}(\overline{x})^{\overline{\varphi}}\theta^{\sigma}$ consists of

- type parameters $\overline{s}$ (to be substituted by object types),
- parameters $\overline{x}$ of types $\overline{\varphi}$ (standing for object identifiers and closed processes),
- and a process θ (specifying the behavior) which conforms to the type σ.

We differentiate between object types (denoted by $\sigma, \tau, \ldots$) and arbitrary types (i.e. types of closed processes or objects, denoted by $\varphi, \psi, \ldots$). Type parameters always stand for object types. (An extended version of this work [24] differentiates between object type parameters, closed process type parameters and general type parameters.)

Closed processes (that resemble generic procedures in imperative languages) can be called by providing types as substitutions for type parameters, and arguments of appropriate types as substitutions for parameters. New objects created by an atomic action get a closed-process call as behavior specification.

Conditional execution is provided in two forms: $a{=}a'\,?\,\theta:\theta'$ corresponds to an if-then-else-statement, where the condition is true if a and a' are equal object identifiers or equivalent closed processes. $\theta + \theta'$ specifies two alternatives; one of them is selected nondeterministically for execution. If θ and θ' are headed by incompatible message accepting actions, the first message in the buffer determines the alternative to be executed. Special syntax was selected for equality (and equivalence) comparisons because type checking is easier if if-paths and else-paths occur pairwise.

As in the polyadic π-calculus [17] pairwise different names enclosed in round brackets represent parameters. They bind further occurrences of these names. An occurrence of a name n in a process is free if it is not bound, i.e., not preceded

by a name abstraction $(\ldots, n, \ldots)$. Constant names like those used as message selectors always are free. All free names in closed processes must be constant. Because of this restriction it is possible to pass closed processes as arguments between objects. Arguments enclosed in square brackets are substituted for parameters. We write $\theta\{\overline{a}/\overline{x}\}$ for a process constructed from θ by simultaneously substituting $\overline{a}$ for all free occurrences of $\overline{x}$, respectively. (Of course, the lists $\overline{a}$ and $\overline{x}$ must have the same length.)

In general, we allow closed processes to be defined recursively. In a closed process $p = {}^{(\overline{s})}(\overline{x})^{\overline{\varphi}}\theta^{\sigma}$, the name p can occur in θ. This name is silently replaced with the closed process wherever needed.

2.2 Examples

A set of simple running examples is used throughout this paper. The examples describe several kinds of data stores (buffers). The closed process S is called within the definition of S:

$$\mathrm{S} = {}^{(s)}()\mathrm{put}(x)^{s}; \mathrm{get}(y)^{\sigma_{\mathrm{B}}}; y.\mathrm{back}[x]; \mathrm{S}^{[s]}[]^{\sigma_{\mathrm{S}}}$$

S specifies the behavior of a data store with a capacity of at most one element of an object type given by the type parameter s. The closed process has no parameters. A data store first accepts a message "put" with a single argument of type s. This message inserts an element into the data store. Then, it accepts a message "get" with an argument of object type σ_{B} (which will be specified later). The argument is supposed to be the identifier of an object that wants to receive the element in the data store as an argument of a message "back". The next action sends this message. Finally, S is called recursively. An object behaving according to S accepts "put" and "get" in alternation, beginning with "put". Other sequences of messages are not acceptable.

The closed process Sd resembles S, but supports the additional message "del" when the buffer is empty. The execution terminates after accepting this message:

$$\mathrm{Sd} = {}^{(s)}()(\mathrm{put}(x)^{s}; \mathrm{get}(y)^{\sigma_{\mathrm{B}}}; y.\mathrm{back}[x]; \mathrm{Sd}^{[s]}[] + \mathrm{del}(); 0)^{\sigma_{\mathrm{Sd}}}$$

Data stores able to hold arbitrary numbers of elements can behave as Si:

$$\mathrm{Si} = {}^{(s)}(x)^{\sigma_{\mathrm{Si}}}(\mathrm{put}(y)^{s}; \mathrm{Sf}^{[s]}[x, y] + \mathrm{get}(y)^{\sigma_{\mathrm{B}}}; x.\mathrm{get}[y]; \mathrm{Si}^{[s]}[x])^{\sigma_{\mathrm{Si}}}$$
$$\mathrm{Sf} = {}^{(s)}(x, y)^{\sigma_{\mathrm{Si}}, s}(\mathrm{get}(z)^{\sigma_{\mathrm{B}}}; z.\mathrm{back}[y]; \mathrm{Si}^{[s]}[x] + \mathrm{put}(z)^{s}; x.\mathrm{put}[z]; \mathrm{Sf}^{[s]}[x, y])^{\sigma_{\mathrm{Si}}}$$

Objects with this behavior accept "put" and "get" in arbitrary order. Since the used process calculus does not support dynamic memory management, arrays, etc., the elements in the data stores are simply stored as messages in the objects' buffers. The messages are inspected; all get-messages are written back into the buffer by sending them to the self-reference x, until a put-message is accepted. Then, put-messages are written back into the buffer, and after accepting a get-message, "back" is sent, and this cycle is repeated. This implementation is not very efficient, but it is useful as an example for type checking.

2.3 Execution and Type Errors

An object system is essentially a set of active objects. Each object executes its thread according to its process as follows:

- If the process is 0, the object's execution halts.
- If the process is of the form $x.c[\overline{a}];\theta$, a message with selector c and arguments $\overline{a}$ is appended to the end of the buffer belonging to the object identified by x. Then, the process becomes θ.
- For a process of the form $c(\overline{x})^{\overline{\varphi}};\theta$, if the object's buffer is not empty and the first message has the selector c and arguments $\overline{a}$ satisfying the type constraints of $\overline{\varphi}$ (where $\overline{x}$, $\overline{\varphi}$ and $\overline{a}$ are lists of the same length), this message is removed from the buffer, and the process becomes $\theta\{\overline{a}/\overline{x}\}$. Otherwise the execution blocks until the condition is satisfied.
- For a process of the form $(x)\$a^{[\overline{\sigma}]}[\overline{a}'];\theta$, a new object with a new buffer and a new identifier y is constructed. The new object behaves according to $a^{[\overline{\sigma}]}[\overline{a}']\{y/x\}$, and the process of the creating object becomes $\theta\{y/x\}$. (The new object identifier is substituted for the parameter x.)
- A process ${}^{(\overline{s})}(\overline{x})^{\overline{\varphi}}\theta^{\sigma[\overline{\tau}]}[\overline{a}]$ (i.e. a call, where the called expression is a closed process) is executed as $\theta\{\overline{\tau}/\overline{s}\}\{\overline{a}/\overline{x}\}$ provided that the lists $\overline{s}$ and $\overline{\tau}$ as well as $\overline{x}$, $\overline{\varphi}$ and $\overline{a}$ have the same lengths, and the arguments $\overline{a}$ satisfy the type constraints $\overline{\varphi}$. (Types are substituted for type parameters, and arguments for parameters.)
- A conditional expression $a{=}a'\,?\,\theta:\theta'$ is executed as θ if $a = a'$. Otherwise it is executed as θ'.
- For a process $\theta + \theta'$, either θ or θ' is selected for execution. The selected alternative must be executable, i.e. neither blocked nor equal to 0. If no alternative is executable, the selection is deferred until an alternative becomes executable.

If the execution of an object cannot proceed for some unintended reason, a type error has been detected. These type errors can occur:

- In a call $a^{[\overline{\tau}]}[\overline{a}']$, the called expression a is not a closed process, or a is of the desired form ${}^{(\overline{s})}(\overline{x})^{\overline{\varphi}}\theta^{\sigma}$, but the lengths of the lists $\overline{s}$ and $\overline{\tau}$ or $\overline{x}$, $\overline{\varphi}$ and $\overline{a}'$ are different, or the arguments $\overline{a}'$ do not satisfy the type constraints $\overline{\varphi}$.
- There is at least one message in the buffer, but the execution still is blocked because no message accepting action in any alternative can deal with the message. The object does not understand the first message in the buffer because a message of this selector and number of arguments is not supported, or the arguments do not satisfy the parameters' type constraints.
- The process is 0, although there are messages in the buffer.

A program assigns a process (not containing free variables) to each initial object. All buffers are initially empty. Objects created later at run-time get their processes (as closed processes and argument lists) from the creating objects. Since closed processes cannot be created at run-time, programs contain all information needed for static type checking.

2.4 Type Consistency

Strong, static typing ensures that a system never gets into a state where type errors show up. A type error occurring in some system state already exists in the program specifying the system behavior. An intuitive definition of type consistency is:

Definition 1. A program P is weakly type-consistent if and only if no system with behavior P gets into a state where type errors show up.

Unfortunately, this definition is not strong enough: It does not support incremental software development processes and separate type checking. In practice, software components shall be compiled separately. And parts of a program—especially closed processes—shall be replaceable with new parts of compatible types without affecting the whole program's type consistency. The following definition seems to be more appropriate:

Definition 2. For a compatibility relation C on closed processes, a program P is type-consistent w.r.t. C if and only if P and each program Q are weakly type-consistent, where Q is constructed from P by substituting arbitrary closed processes $p_1, \ldots, p_n$ by closed processes $q_1, \ldots, q_n$ with $q_i \, C \, p_i$ $(1 \leq i \leq n)$.

The less restrictive the compatibility relation C, the more freedom has the programmer in replacing software components without affecting type consistency, but the smaller is the set of type-consistent programs. A compromise between freedom in replacing components type-safely and freedom in programming (which influences the simplicity of developing components) must be found. An appropriate compromise is a compatibility relation defined just as restrictive as needed for separate compilation. Separate compilation is possible if

- all parameters are associated with static type constraints;
- no further assumptions than expressed in type constraints are made;
- arguments must satisfy the parameters' type constraints.

If these conditions hold (as in the proposed model), a closed process that satisfies some constraints can be replaced without further type checking by closed processes satisfying at least the same constraints. These constraints are expressed in the closed processes' (and the corresponding objects') types. In the rest of this work we use the compatibility relation $\sqsubseteq$:

Definition 3. Two closed processes p and q of types φ and ψ, respectively, are related by $p \sqsubseteq q$ if and only if φ is a subtype of ψ.

For example, it shall be possible to replace the closed process S (as defined in Sect. 2.2) with Sd or Si; Sd $\sqsubseteq$ S and Si $\sqsubseteq$ S shall hold because both, Sd and Si, accept all message sequences accepted by S.

When replacing a closed process, it is sufficient to check the type consistency of the new closed process, provided that the new closed process' type is a subtype of the replaced one's type.

$\varphi ::= @(\overline{s})\langle\overline{\varphi}\rangle\sigma$	(closed process type; without free type parameters)
$\mid \sigma$	(object type)
$\sigma ::= \{\overline{u}\}\mid b$	(descriptive object type)
$\mid \sigma \times \sigma'$	(combination of object types)
$\mid s$	(type parameter)
$u ::= c$	(simple state descriptor)
$\mid c^*$	(replicated state descriptor)
$b ::= 0$	(empty behavior descriptor)
$\mid c\langle\overline{\varphi}\rangle\{\overline{c}\}\rightarrow\{\overline{u}\}$	(message descriptor)
$\mid b + b'$	(combined behavior descriptor)

Fig. 2. Syntax of Types

3 Message Sequences Expressed in Types

3.1 Syntax and (Informal) Semantics of Types

The proposed model supports types of two kinds: Types of objects and types of closed processes. This distinction is reflected in the syntax of types as defined in Fig. 2, where $\varphi, \psi, \ldots$ denote types of any kind, and $\sigma, \tau, \ldots$ object types.

Descriptive object types partially specify the behavior of these types' instances (objects). They describe the acceptable messages as well as restrictions on their orders. During computation, an object's type may change in the same way as the object's process. Hence, types shall contain variable components. A type $\{\overline{u}\}\mid b$ consists of two parts: The *activating set* $\{\overline{u}\}$ represents an abstract state of the type's instances. The *behavior descriptor* b describes a set of messages; their acceptability depends on the activating set. (Alternative message descriptors in behavior descriptors are combined by $+$; 0 denotes a behavior descriptor not supporting any message.) When an object's process changes, the activating set can also change, but the behavior descriptor remains unchanged.

An activating set $\{\overline{u}\}$ is a multi-set of *state descriptors*, each denoted by $u, v, w, \ldots$. Different states are distinguished by the presence or absence of state descriptors. Some of the constant names used as state descriptors are marked with an asterisk indicating that an infinite number of the name's duplicates are contained in the multi-set.

For each supported message, the behavior descriptor contains a *message descriptor* $d\langle\overline{\varphi}\rangle\{\overline{c}\}\{\overline{v}\}$, where d is the message selector, $\overline{\varphi}$ the list of parameter types, $\{\overline{c}\}$ the *in-set* and $\{\overline{v}\}$ the *out-set*. The message descriptor is *active* if all names in the multi-set $\{\overline{c}\}$ are contained in the activating set. (A name c is regarded as being contained in the multi-set if c^* or c is in the multi-set.) An active message descriptor specifies an acceptable message. When a corresponding message is accepted, the names in $\{\overline{c}\}$ are removed from the activating set (but not state descriptors of the form c^*), and the state descriptors in the multi-set $\{\overline{v}\}$ are added.

A closed process type $@(\overline{s})\langle\overline{\varphi}\rangle\sigma$ specifies the type parameters $\overline{s}$ and the parameter types $\overline{\varphi}$ of its instances (closed processes), as well as the initial types of objects behaving according to these closed processes. The (most concrete) type of a closed process $p = {}^{(\overline{s})}(\overline{x})^{\overline{\varphi}}\theta^{\sigma}$ is $@(\overline{s})\langle\overline{\varphi}\rangle\sigma$, where σ (or more completely $\sigma\{\overline{\tau}/\overline{s}\}$) is a type of an object with behavior $p^{[\overline{\tau}]}[\overline{a}]$, i.e. a call of p.

For example, the object type σ_S of an empty simple data store with behavior S (as introduced in Sect. 2.2) can be:

$$\sigma_S = \{\text{empty}\} \mid \text{put}\langle s\rangle\{\text{empty}\}\rightarrow\{\text{full}\} + \text{get}\langle\sigma_B\rangle\{\text{full}\}\rightarrow\{\text{empty}\}$$

Only "put" is acceptable. After accepting "put", the activating set becomes {full}, and only "get" becomes acceptable. The parameter type of "get" may be the object type

$$\sigma_B = \{\text{once}\} \mid \text{back}\langle s\rangle\{\text{once}\}\rightarrow\{\}$$

i.e., the object accepts "back" with an argument of type s only once. ({} denotes an empty multi-set.)

The types σ_{Sd} and σ_{Si} of empty data stores with behavior Sd and Si (as introduced in Sect. 2.2), respectively, can be:

$$\begin{aligned}\sigma_{Sd} &= \{\text{empty}\} \mid \text{put}\langle s\rangle\{\text{empty}\}\rightarrow\{\text{full}\} + \text{get}\langle\sigma_B\rangle\{\text{full}\}\rightarrow\{\text{empty}\} \\ &\qquad + \text{del}\langle\rangle\{\text{empty}\}\rightarrow\{\} \\ \sigma_{Si} &= \{\} \mid \text{put}\langle s\rangle\{\}\rightarrow\{\} + \text{get}\langle\sigma_B\rangle\{\}\rightarrow\{\}\end{aligned}$$

An instance of a combination of object types $\sigma \times \tau$ accepts all messages specified by σ as well as those specified by τ in arbitrary interleaving. When accepting messages specified by σ, only σ is updated, and when accepting messages specified by τ, only τ is updated. Combinations of object types play an important role in subtyping, static type checking, and together with type parameters.

A type parameter s does not give any information about the messages acceptable by instances of s. Sometimes we need type parameters, but still want to have some information about acceptable messages. In object-oriented languages like Eiffel, bounded type parameters (where types substituted for type parameters must be subtypes of some type constants) can provide this type information. Unfortunately, bounded type parameters are not directly applicable in our model: If a message is sent to an instance of a type represented by a type parameter, the type may change; we have no representation of the changed type. Combinations of object types of the form $\{\overline{u}\}|b \times s_1 \times \cdots \times s_n$ provide a solution: Instances of these types are known to accept messages according to $\{\overline{u}\}|b$; the activating sets in the first parts of the types can be updated, while the type parameters remain unchanged.

3.2 Normalizable Object Types

If a user of an object knows that the object's type is $\{\overline{u}\}|b$, the user can safely send an acceptable message to the object. Then, the user updates his knowledge of the object's type, and the object updates its type after accepting the message.

Of course it is necessary that the object and its user update the type in the same way so that the user's knowledge remains valid. However, object types as described so far do not necessarily specify deterministically, how types shall be updated after sending or accepting a message: For example, if the type is of the form $\{c_1, c_2\} | \, d\langle\overline{\varphi}\rangle\{c_1\}\rightarrow\{c_3\} + d\langle\overline{\varphi}\rangle\{c_2\}\rightarrow\{c_4\} + \cdots$, the updated activating set is $\{c_2, c_3\}$ or $\{c_1, c_4\}$, depending on the considered message descriptor. In the rest of this paper we deal only with deterministic object types, where for each supported message there is only one possibility of updating the activating set. A sufficient (but not necessary) condition for deterministic object types is that all message descriptors have pairwise different message selectors.

State descriptors in activating sets and out-sets are useful only if there are message descriptors depending on these state descriptors. If a state descriptor does not occur in any in-set, it has no meaning. State descriptors without meaning are undesirable, especially when combining object types. Therefore, we define object types in normal form by:

Definition 4. An object type $\{\overline{u}\}|b \times s_1 \times \cdots \times s_m$ (where $m \geq 0$ and b is 0 or of the form $d_1\langle\overline{\varphi}_1\rangle\{\overline{c}_1\}\rightarrow\{\overline{v}_1\} + \cdots + d_n\langle\overline{\varphi}_n\rangle\{\overline{c}_n\}\rightarrow\{\overline{v}_n\}$) is in normal form if and only if all message selectors $d_1, \ldots, d_n$ are pairwise different and there exists a $c \in \bigcup_{1\leq i\leq n}\{\overline{c}_i\}$ for each $c, c^* \in \bigcup_{1\leq i\leq n}\{\overline{v}_i\} \cup \{\overline{u}\}$, and all object types occurring in $\overline{\varphi}_1, \ldots, \overline{\varphi}_n$ are in normal form.

The specific form of object types in normal form is not always useful. For example, the types σ_{S}, σ_{Sd}, σ_{Si} and σ_{B} (shown in Sect. 3.1) are not in normal form because s is not in normal form. Sometimes we need object types only semantically equivalent to object types in normal form. The type parameter s is semantically equivalent to the object type $\{\}|0 \times s$ in normal form:

Definition 5. An object type σ is normalizable if and only if σ can be reduced to an object type in normal form by

- using associativity, commutativity and 0 as neutral element of $+$,
- using associativity, commutativity and $\{\}|0$ as neutral element of $\times$,
- and repeatedly applying the rules R1 to R4:

R1 For a type $\{\overline{u}\}|b$: Remove a state descriptor c or c^* from the activating set or an out-set in b if c does not occur in any in-set.

R2 For a multi-set of state descriptors: Remove all duplicates of state descriptors of the form c^*; if the multi-set contains c^*, remove all c.
This rule is sound because c^* stands for an infinite number of copies of c.

R3 For a type $\{\overline{u}\}|b$: Simultaneously remove message descriptors of the form $d\langle\varphi_1, \ldots, \varphi_n\rangle\{\overline{c}\}\rightarrow\{\overline{v}\}$ from b and then apply R1 to the remaining rules if (for each removed message descriptor) b contains a further message descriptor $d\langle\psi_1, \ldots, \psi_n\rangle\{\overline{c}'\}\rightarrow\{\overline{v}'\}$, where each φ_i is a subtype of ψ_i $(1 \leq i \leq n)$, $\{\overline{c}'\}$ is a sub-multi-set of $\{\overline{c}\}$, and (with w and w' constructed from v and v', respectively, by applying R1) $\{\overline{w}\}$ is a sub-multi-set of $\{\overline{w}'\}$.
This rule removes unnecessary message descriptors dealing with messages also dealt with by other message descriptors:

- Each argument type conforming to a parameter type of a removed message descriptor also conforms to the corresponding parameter type of a remaining message descriptor.
- A remaining message descriptor always is active when a removed message descriptor was active.
- When corresponding messages are accepted, the remaining message descriptors add all state descriptors added by the removed ones.

R4 For a type $\{\overline{u}\}|b \times \{\overline{v}\}|b'$, where $\{\overline{u}\}|b$ and $\{\overline{v}\}|b'$ are in normal form: Replace this type with $\{\overline{u}, \overline{v}\} \mid b + b'$.
This rule reflects the informal semantics: Instances of $\sigma \times \tau$ accept at least all messages acceptable by σ and those acceptable by τ in arbitrary interleaving. ($\{\overline{u}\}|b \times \{\overline{v}\}|b'$ specifies more acceptable messages than interleavings of $\{\overline{u}\}|b$ and $\{\overline{v}\}|b'$ if message descriptors in $b+b'$ depend on state descriptors in both, $\{\overline{u}\}$ and $\{\overline{v}\}$.) Since $\{\overline{u}\}|b$ and $\{\overline{v}\}|b'$ are in normal form, $\{\overline{u}\}$ and $\{\overline{v}\}$ cannot contain state descriptors useful in $b + b'$ but not in b and b', respectively.

All object types that can be reduced to the same object type using the algebraic properties and R1 to R4 are regarded as equivalent. We write $\sigma \equiv \tau$ if σ and τ are equivalent. It is easy to see that the rules are terminating and confluent. Hence, there exists an effective algorithm for deciding whether or not two object types are equivalent.

Rule R4 expresses an equivalence between combinations of object types and descriptive object types. An important equivalence for all behavior descriptors b and activating sets $\{\overline{u}_1\}, \ldots, \{\overline{u}_n\}$ is $\{\overline{u}_1\}|b \times \cdots \times \{\overline{u}_n\}|b \equiv \{\overline{u}_1, \ldots, \overline{u}_n\}|b$. (Since all combined object types have the same behavior descriptor, the same set of state descriptors is useful for them. Replicated message descriptors in $b+\cdots+b$ can be removed by applying R3.) Type checking is based on this equivalence. In the specific case of activating sets $\{\overline{u}\}$ containing only replicated state descriptors of the form c^*, the equivalence $\{\overline{u}\}b \times \{\overline{u}\}b \equiv \{\overline{u}\}b$ holds. Combinations of object types with different behavior descriptors are used in subtyping.

Object type equivalence is easily extended to closed process types: Two closed process types are equivalent if they have the same numbers of type parameters, and the parameter types as well as the types specifying the initial object behavior are pairwise equivalent (after renaming type parameters). The notions of normal forms and normalizable types also can easily be adapted to closed process types: A closed process type φ is normalizable (or in normal form) if all object types occurring in φ are normalizable (in normal form).

We assume that object types and closed process types can be defined recursively: For a type $\sigma = \{\overline{u}\}|b$, the type name σ can occur as parameter type in b; and for a type $\psi = @(\overline{s})\langle\overline{\varphi}\rangle\sigma$, the name ψ can occur in $\overline{\varphi}$. These names are silently replaced with the corresponding type expressions wherever needed. Type equivalence is decidable even when using recursive types. (The corresponding rules are not shown here because of their rather involved technicalities.)

The normalizability property of object types is preserved when types are updated: After accepting a message, only useful state descriptors are added to activating sets; behavior descriptors and type parameters remain unchanged.

3.3 Subtyping

According to the principle of substitutability, an instance of a subtype can be used wherever an instance of a supertype is expected [11, 29]. Especially, an instance of a subtype must accept all message sequences as promised by a supertype. This consideration immediately leads to the definition:

Definition 6. An object type σ is a subtype of an object type τ (formally $\sigma \leq \tau$) if and only if there exists an object type σ' such that $\sigma \equiv \tau \times \sigma'$.

A closed process type $@(\overline{s})\langle\varphi_1, \ldots, \varphi_n\rangle\sigma$ is a subtype of a closed process type ψ if and only if ψ (after renaming bound type parameters) is of the form $@(\overline{s})\langle\psi_1, \ldots, \psi_n\rangle\tau$ and $\psi_i \leq \varphi_i$ (for all $1 \leq i \leq n$) and $\sigma \leq \tau$.

The subtype relation on object types directly reflects the following property: A subtype σ extends a supertype τ by supporting additional messages and message orders (as specified by σ'). Each message accepted by an instance of τ is also accepted by an instance of $\tau \times \sigma'$ (and equivalently σ).

Types of closed processes have contravariant parameter types: If an instance p of a type $@(\overline{s})\langle\overline{\varphi}\rangle\sigma$ is expected in a call $p^{[\overline{\tau}]}[\overline{a}]$, each instance of a subtype ψ can be used instead of p and must be able to deal with all arguments $\overline{a}$. An object executing this call must accept all messages specified by σ.

It is easy to verify that subtyping is an antisymmetric, reflexive and transitive relation on equivalence classes of types.

An important step in deciding whether or not an object type σ is a subtype of a type τ is to find an appropriate type σ' such that $\sigma \equiv \tau \times \sigma'$. Fortunately, σ' can easily be found for object types in normal form: If σ is of the form $\{\overline{u}\}|b \times s_1 \times \cdots \times s_m \times \cdots \times s_n$, and τ of the form $\{\overline{v}\}|b' \times s_1 \times \cdots \times s_m$, then σ' is $\{\overline{w}\}|b \times s_{m+1} \times \cdots \times s_n$, where $\{\overline{w}\}$ contains all state descriptors in $\{\overline{u}\}$ except those in $\{\overline{v}\}$. Under these conditions, $\sigma \leq \tau$ can be decided by deciding whether or not $\{\overline{u}\}|b \equiv \{\overline{v}, \overline{w}\} \mid b + b'$. If σ and τ cannot be brought into these forms using commutativity of $\times$, σ is not a subtype of τ.

Intuitively, the object types σ_S, σ_{Sd} and σ_{Si} (defined in Sect. 3.1) shall be related by $\sigma_{Si} \leq \sigma_S$ and $\sigma_{Sd} \leq \sigma_S$, but σ_{Si} and σ_{Sd} shall not be related by subtyping. These types are (after replacing s with $\{\}|0 \times s$) in normal form. We show $\sigma_{Si} \leq \sigma_S$ by showing $\sigma_{Si} \equiv \sigma_S \times \sigma'_{Si}$, where $\sigma'_{Si} = \sigma_{Si}$:

$$\begin{aligned}\sigma_S \times \sigma_{Si} &\equiv \{\text{empty}\} \mid \text{put}\langle s\rangle\{\text{empty}\}\rightarrow\{\text{full}\} + \text{put}\langle s\rangle\{\}\rightarrow\{\} \\ &\quad + \text{get}\langle\sigma_B\rangle\{\text{full}\}\rightarrow\{\text{empty}\} + \text{get}\langle\sigma_B\rangle\{\}\rightarrow\{\} \\ &\equiv \{\} \mid \text{put}\langle s\rangle\{\}\rightarrow\{\} + \text{get}\langle\sigma_B\rangle\{\}\rightarrow\{\} = \sigma_{Si}\end{aligned}$$

First, the type combination is resolved by using R4. Then, a message descriptor for "put" and one for get are removed simultaneously by applying R3. (The two message descriptors cannot be removed sequentially because the out-sets would not be appropriate.) The proof of $\sigma_{Sd} \leq \sigma_S$ is equally simple:

$$\begin{aligned}\sigma_S \times \sigma_{Sd} &\equiv \{\text{empty}\} \mid \text{put}\langle s\rangle\{\text{empty}\}\rightarrow\{\text{full}\} + \text{put}\langle s\rangle\{\text{empty}\}\rightarrow\{\text{full}\} \\ &\quad + \text{get}\langle\sigma_B\rangle\{\text{full}\}\rightarrow\{\text{empty}\} + \text{get}\langle\sigma_B\rangle\{\text{full}\}\rightarrow\{\text{empty}\} \\ &\quad + \text{del}\langle\rangle\{\text{empty}\}\rightarrow\{\} \\ &\equiv \sigma_{Sd}\end{aligned}$$

4 Type Checking

Static type checking is divided into two logical parts. One part checks whether objects (regarded as servers) are actually able to accept all messages as promised by the object's type. The other part checks whether objects (regarded as users) send only type-conforming messages. Before we can give a type checking algorithm, we have to state which messages users can safely send to an object.

4.1 Type Marks

As mentioned in Sect. 1.2, the messages each user can send to an object are controlled by type marks. In a process, the initial type mark for an object is the type annotation of the parameter standing for the object. (We will usually say "type mark of a parameter" instead of "type mark for an object represented by a parameter".) Type annotations of parameters bound in message receiving actions or closed processes (parameter types) are specified explicitly. The type annotation of the parameter bound in an object creating action is the annotation of the closed process specifying the new object's behavior. If a parameter of type $@(\overline{s})(\overline{\varphi})\sigma$ is used for this closed process, the initial type mark for the created object is σ.

A type mark of x always reflects the user's knowledge about the type of the object represented by x. A process $x.c[\overline{a}];\theta$ is type-conforming only if the type mark of x specifies an acceptable message with selector c and an appropriate number of parameters with appropriate types. The type mark of x in θ can be different: The type mark must be updated by removing the state descriptors in the corresponding message descriptor's in-set from the activating set and adding the state descriptors in the out-set.

An object can have several users who send messages concurrently. The object must be able to accept all messages from concurrent users in arbitrary interleaving. Combinations of object types have the required properties: If there are n parameters $x_1, \ldots, x_n$ (standing for the same object) with type marks $\sigma_1, \ldots, \sigma_n$, respectively, the object must understand all messages according to $\sigma_1 \times \cdots \times \sigma_n$. Especially, if the object is of type $\{\overline{u}_1, \ldots, \overline{u}_n\}|b$, each parameter x_i can have a type mark σ_i with $\{\overline{u}_i\}|b \leq \sigma_i$.

An object creating action $(x)\$a^{[\overline{\tau}]}[\overline{a}']$ binds a single parameter x; no other parameter stands for the new object. Hence, the type mark σ of x can be equal to the new object's type. When an alias of x (i.e. a further parameter standing for the same object as x) is introduced by using x as an argument, x's type mark σ must be split into σ_1 and σ_2, where $\sigma \equiv \sigma_1 \times \sigma_2$; x gets the new type mark σ_2, and the new parameter's initial type mark is σ_1. So, the condition stated in the previous paragraph remains satisfied. In general, type splitting has to be applied whenever a new alias is introduced, i.e. for each use of a parameter of an object type as an argument. The original type mark σ of a parameter x used as argument is known as well as the explicitly specified type mark σ_1 (parameter type) of the new alias. The new type mark σ_2 of x can be computed from the equivalence $\sigma \equiv \sigma_1 \times \sigma_2$ in the same way as the object type needed in proving

$\sigma \leq \sigma_1$ (see Sect. 3.3). If $\sigma \leq \sigma_1$ does not hold, the types of x and its new alias are incompatible and the program is not type-consistent.

A type $\sigma \times \tau$ can specify more acceptable messages than all interleavings of acceptable messages specified by σ and τ. A single parameter with a type mark $\sigma \times \tau$ can allow a user to send more messages than two parameters with type marks σ and τ. Therefore, the process calculus contains conditional expressions of the form $x{=}y\,?\,\theta : \theta'$ (where x's type mark is σ, and y's type mark is τ). In the if-path θ, x and y are regarded as equal with the combined type mark $\sigma \times \tau$, whereas in the else-path θ', x and y still have separate type marks.

No updating and splitting of types is necessary for closed process types. Closed process types are handled in the same way as types in conventional object-oriented languages.

For a human reader it is rather easy to understand the type annotations of the closed processes S, Sd and Si (as introduced in Sect. 2.2 with types defined in Sect. 3.1). Directly after accepting a message $\mathrm{get}(y)^{\sigma_B}$, y has the type mark $\{\mathrm{once}\} \mid \mathrm{back}\langle s\rangle\{\mathrm{once}\}{\rightarrow}\{\}$, and x has the type mark s. After an action $y.\mathrm{back}[x]$, y's type mark is updated to $\{\} \mid \mathrm{back}\langle s\rangle\{\mathrm{once}\}{\rightarrow}\{\}$, and x's type mark is split such that x's new type mark is $\{\}|0$. Neither x nor y accept any further message. The type mark σ_{Si} resembles a type in a conventional object-oriented language: It remains unchanged after sending messages to its instances, and it can be split into σ_{Si} and σ_{Si} because $\sigma_{\mathrm{Si}} \equiv \sigma_{\mathrm{Si}} \times \sigma_{\mathrm{Si}}$.

4.2 Checking Type Marks

Since each occurrence of a parameter is associated with a type mark, it is not difficult for a compiler to ensure that users send only messages as specified in the type marks to an object. A checker of type marks walks (from left to right) through each (closed) process in a program. Thereby it has to

- initialize the type mark of each parameter where it is bound; free parameters must not occur in a program;
- for each message sending action $x.d[a_1, \ldots, a_n]$:
  1. ensure that x's type mark specifies an active message descriptor of the form $d\langle\varphi_1, \ldots, \varphi_n\rangle\{\overline{c}\}{\rightarrow}\{\overline{u}\}$, and update x's type mark;
  2. for each $1 \leq i \leq n$: if a_i is a parameter, ensure that a_i's type mark is a subtype of φ_i and (if φ_i is an object type) split a_i's type mark;
     otherwise (a_i is a closed process ${}^{(\overline{s})}(\overline{x})^{\overline{\psi}}\theta^{\sigma}$) check the type marks and the object behavior of a_i and ensure $@(\overline{s})\langle\overline{\psi}\rangle\sigma \leq \varphi_i$;
- for each action $a^{[\sigma_1,\ldots,\sigma_m]}[a_1, \ldots, a_n]$ or $(x)\$a^{[\sigma_1,\ldots,\sigma_m]}[a_1, \ldots, a_n]$ (after initializing x's type mark):
  1. ensure that a is a closed process ${}^{(s_1,\ldots,s_m)}(\overline{x})^{\varphi_1,\ldots,\varphi_n}\theta^{\sigma}$ for which a check of the type marks and the object behavior succeeds, or a is a parameter with a type mark $@(s_1, \ldots, s_m)\langle\varphi_1, \ldots, \varphi_n\rangle\sigma$;
  2. for each $1 \leq i \leq n$: if a_i is a parameter, ensure that a_i's type mark is a subtype of φ_i and (if φ_i is an object type) split a_i's type mark;

otherwise (a_i is a closed process $^{(\overline{s}')}(\overline{x}')^{\overline{\psi}}\theta^{\tau}$) check the type marks and the object behavior of a_i and ensure $@(\overline{s}')\langle\overline{\psi}\rangle\tau \leq \varphi_i$;

- for each expression $\theta + \theta'$: check θ and θ' independently (with the same type marks at the beginning);
- for each expression $a{=}a'\,?\,\theta : \theta'$: if both, a and a' are parameters standing for object types, check θ' with the current type marks, and check $\theta\{a/a'\}$ independently with updated type marks, where a's updated type mark is a combination of a's and a''s current type marks;
  otherwise (a or a' is a closed process or a parameter standing for a closed process) check θ and θ' independently.

Proposition 7. *Let P be a program passing the above checks. Then, no system with behavior P can get into a state where an object's type does not contain an active message descriptor corresponding to the first message in the object's buffer. Furthermore, the checks ensure that each called expression is an appropriate closed process.*

This proposition holds because each object's type is a subtype of the combination of the type marks of all parameters standing for this object. Checking type marks ensures that type marks are actually used as described in Sect. 4.1.

4.3 Checking Object Behavior

A type checker also has to ensure that each object always can deal with all messages corresponding to an active message descriptor in the object's type. A checker of object behavior walks (from left to right) through each closed process $^{(\overline{s})}(\overline{x})^{\overline{\psi}}\theta^{\sigma}$ in a program. Thereby it has to ensure for θ with σ that

- σ is a normalizable object type $\{\overline{u}\}|b$; then, σ is reduced to normal form;
- if θ is 0, no message descriptor in σ is active;
- if θ is $d_1(\overline{x}_1)^{\overline{\varphi}_1};\theta_1 + \cdots + d_n(\overline{x}_n)^{\overline{\varphi}_n};\theta_n$ ($n \geq 1$), σ contains no active message descriptor $d\langle\overline{\varphi}'\rangle\{\overline{c}\}\rightarrow\{\overline{v}\}$ with $d \notin \{d_1, \ldots, d_n\}$ (i.e., there is at least one message accepting action for each active message descriptor), and checking succeeds for each $d_i(\overline{x}_i)^{\overline{\varphi}_i};\theta_i$ with σ ($1 \leq i \leq n$);
- if θ is $\theta_1 + \cdots + \theta_n$ ($n \geq 1$), where at least one $\theta_i \neq 0$ has no message receiving action in the head, checking succeeds for each θ_i with σ ($1 \leq i \leq n$);
- (in addition to the above two items) if θ is $d(\overline{x})^{\varphi_1,\ldots,\varphi_n};\theta'$ and σ contains an active message descriptor $d\langle\varphi_1', \ldots, \varphi_n'\rangle\{\overline{c}\}\rightarrow\{\overline{v}\}$, each $\varphi_i' \leq \varphi_i$ ($1 \leq i \leq n$) and checking succeeds for θ' with σ', where σ' is constructed by updating σ;
- if θ is $d(\overline{x})^{\overline{\varphi}};\theta'$ and σ contains no appropriate active message descriptor, this dead code (θ) is eliminated (replaced with 0);
- if θ is $x.d[\overline{a}];\theta'$ or $(x)\$a^{[\overline{\tau}]}[\overline{a}'];\theta'$, checking succeeds for θ' with σ;
- if θ is $a{=}a'\,?\,\theta' : \theta''$, checking succeeds for θ' with σ and for θ'' with σ;
- if θ is $a^{[\overline{\tau}]}[\overline{a}']$ and a is a closed process $^{(\overline{s}')}(\overline{x}')^{\overline{\psi}'}\theta'^{\tau}$ or a parameter of type $@(\overline{s}')\langle\overline{\psi}'\rangle\tau$, then $\tau\{\overline{\psi}'/\overline{s}'\} \leq \sigma$.

Proposition 8. *Let P be a program passing the above checks. Then, each object in a system with behavior P accepts all messages specified by the object's type.*

The checks ensure that there is a message accepting action for each active message descriptor.

The main result of this paper directly follows from the definition of $\sqsubseteq$ (given in Sect. 2.4), Proposition 7 and Proposition 8:

Theorem 9. *A program passing the above checks of type marks and object behavior is type-consistent w.r.t. $\sqsubseteq$.*

For each $q \sqsubseteq p$, the closed process q substituted for p is checked in the same way as p. Since q's type is a subtype of p's type, q satisfies all constraints promised by p's type.

Programs accepted by our type checker do not suffer from an important kind of deadlocks:

Theorem 10. *Let P be a program passing the above checks of type marks and object behavior. Then, in a system with behavior P, the execution of an object with a nonempty message buffer cannot be blocked.*

Deadlocks can occur only if a user does not send a message needed by an object.

We shall estimate the complexity of type checking: A type checker runs through each process once for checking object behavior and once for checking type marks. (Of course, these phases can be combined.) No part of a process must be checked several times in each phase. The type checker always knows which checks must be applied to show type consistency. Thus, the type checker runs through the code at most twice. Assuming an appropriate type representation and considering type comparisons with a worst case time complexity of at most quadratic order, the worst case time complexity of the whole type checker is of at most quadratic order.

5 Discussion

5.1 Coordination with Type Marks

The solution of the dining philosophers problem in Fig. 3 shall demonstrate how synchronization restrictions can be expressed in the proposed type model. Philosophers sit around a table and spend their time with thinking and eating. Each philosopher has a plate of spaghetti and a fork. However, two forks are needed for eating this kind of spaghetti. When a philosopher becomes hungry, he has to borough a fork from his neighbor. All philosophers shall get a fair chance to eat. The dining philosophers problem is a well-known example from a class of problems, where several concurrent users (philosophers) need exclusive access to limited, shared resources (forks).

The behavior descriptor b_P specifies the messages each philosopher can deal with. A philosopher always is in one of four abstract states: thinking, asking his

$$
\begin{aligned}
b_{\mathrm{P}} = {} & \mathrm{eat}\langle\{\}|b_{\mathrm{P}}, \{\mathrm{down}\}|b_{\mathrm{F}}\rangle\{\mathrm{thinking}\}\rightarrow\{\mathrm{asking}\} + \\
& \mathrm{ask}\langle\{\mathrm{asking}\}|b_{\mathrm{P}}\rangle\{\}\rightarrow\{\} + \\
& \mathrm{yes}\langle\{\mathrm{eating}\}|b_{\mathrm{P}}, \{\mathrm{nice}\}|b_{\mathrm{P}}, \{\mathrm{down}\}|b_{\mathrm{F}}\rangle\{\mathrm{asking}\}\rightarrow\{\mathrm{eating}\} + \\
& \mathrm{no}\langle\{\mathrm{thinking}\}|b_{\mathrm{P}}\rangle\{\mathrm{asking}\}\rightarrow\{\mathrm{thinking}\} + \\
& \mathrm{think}\langle\rangle\{\mathrm{eating}\}\rightarrow\{\mathrm{thinking}\} + \\
& \mathrm{grant}\langle\{\}|b_{\mathrm{P}}\rangle\{\mathrm{thinking}\}\rightarrow\{\mathrm{nice}\} + \\
& \mathrm{back}\langle\{\mathrm{thinking}\}|b_{\mathrm{P}}, \{\mathrm{down}\}|b_{\mathrm{F}}\rangle\{\mathrm{nice}\}\rightarrow\{\mathrm{thinking}\}
\end{aligned}
$$

$$
\begin{aligned}
b_{\mathrm{F}} = {} & \mathrm{get}\langle\rangle\{\mathrm{down}\}\rightarrow\{\mathrm{up}\} + \\
& \mathrm{put}\langle\rangle\{\mathrm{up}\}\rightarrow\{\mathrm{down}\}
\end{aligned}
$$

$$
\begin{aligned}
\mathrm{Pth} = {} & {}^{()}(x,y,z)^{\{\mathrm{thinking}\}|b_{\mathrm{P}},\{\}|b_{\mathrm{P}},\{\mathrm{down}\}|b_{\mathrm{F}}}(\\
& \mathrm{ask}(y')^{\{\mathrm{asking}\}|b_{\mathrm{P}}}; x.\mathrm{grant}[y]; x'.\mathrm{yes}[x',x,z]; \mathrm{Pf}^{[]}[] + \\
& x.\mathrm{eat}[y,z]; y.\mathrm{ask}[x]; \mathrm{Pf}^{[]}[])^{\{\mathrm{thinking}\}|b_{\mathrm{P}}} \\
\mathrm{Pf} = {} & {}^{()}(\mathrm{ask}(y)^{\{\mathrm{asking}\}|b_{\mathrm{P}}}; y.\mathrm{no}[y]; \mathrm{Pf}^{[]}[] + \\
& \mathrm{grant}(y)^{\{\}|b_{\mathrm{P}}}; \mathrm{Pn}^{[]}[y] + \\
& \mathrm{eat}(y,z)^{\{\}|b_{\mathrm{P}},\{\mathrm{down}\}|b_{\mathrm{F}}}; \mathrm{Pa}^{[]}[y,z])^{\{\mathrm{thinking}\}|b_{\mathrm{P}}} \\
\mathrm{Pn} = {} & {}^{()}(y)^{\{\}|b_{\mathrm{P}}}(\mathrm{ask}(y')^{\{\mathrm{asking}\}|b_{\mathrm{P}}}; y'.\mathrm{no}[y']; \mathrm{Pn}^{[]}[y] + \\
& \mathrm{back}(x,z)^{\{\mathrm{thinking}\}|b_{\mathrm{P}},\{\mathrm{down}\}|b_{\mathrm{F}}}; \mathrm{Pth}^{[]}[x,y,z])^{\{\mathrm{nice}\}|b_{\mathrm{P}}} \\
\mathrm{Pa} = {} & {}^{()}(y,z)^{\{\}|b_{\mathrm{P}},\{\mathrm{down}\}|b_{\mathrm{F}}}(\mathrm{ask}(y')^{\{\mathrm{asking}\}|b_{\mathrm{P}}}; y'.\mathrm{no}[y'], \mathrm{Pa}^{[]}[y,z] + \\
& \mathrm{yes}(x,y,z')^{\{\mathrm{eating}\}|b_{\mathrm{P}},\{\mathrm{nice}\}|b_{\mathrm{P}},\{\mathrm{down}\}|b_{\mathrm{F}}}; z.\mathrm{get}[]; z'.\mathrm{get}[]; \mathrm{Pe}^{[]}[x,y,z,z'] + \\
& \mathrm{no}(x)^{\{\mathrm{thinking}\}|b_{\mathrm{P}}}; \mathrm{Pth}^{[]}[x,y,z])^{\{\mathrm{asking}\}|b_{\mathrm{P}}} \\
\mathrm{Pe} = {} & {}^{()}(x,y,z,z')^{\{\mathrm{eating}\}|b_{\mathrm{P}},\{\}|b_{\mathrm{P}},\{\mathrm{up}\}|b_{\mathrm{F}},\{\mathrm{up}\}|b_{\mathrm{F}}}(\\
& \mathrm{ask}(y')^{\{\mathrm{asking}\}|b_{\mathrm{P}}}; y'.\mathrm{no}[y']; \mathrm{Pe}^{[]}[x,y,z,z'] + \\
& x.\mathrm{think}[]; z.\mathrm{put}[]; z'.\mathrm{put}[]; y.\mathrm{back}[y,z']; \mathrm{Pg}^{[]}[x,y,z])^{\{\mathrm{eating}\}|b_{\mathrm{P}}} \\
\mathrm{Pg} = {} & {}^{()}(x,y,z)^{\{\mathrm{thinking}\}|b_{\mathrm{P}},\{\}|b_{\mathrm{P}},\{\mathrm{down}\}|b_{\mathrm{F}}}(\\
& \mathrm{ask}(y')^{\{\mathrm{asking}\}|b_{\mathrm{P}}}; y'.\mathrm{no}[y']; \mathrm{Pg}^{[]}[x,y,z] + \\
& \mathrm{think}(); \mathrm{Pth}^{[]}[x,y,z])^{\{\mathrm{eating}\}|b_{\mathrm{P}}}
\end{aligned}
$$

Fig. 3. Closed Processes for Dining Philosophers

right neighbor for a fork, being nice by giving his fork to his left neighbor, and eating with two forks. The acceptable messages depend on the abstract state. As the behavior descriptor b_{F} shows, a fork can be in two states: down on the table or up in a philosopher's hand. The abstract states of philosophers and forks as well as the acceptable messages for each abstract state and the corresponding state changes are shown graphically in Fig. 4.

When asked for a fork, a philosopher with behavior Pth gives his fork to a neighbor by sending "grant" to himself and "yes" to the philosopher who asked. (Parameter x stands for the philosopher itself, y for his right neighbor, y' for his left neighbor who asks for the fork, and z for the own fork.) When a philosopher becomes hungry, he sends "eat" to himself and "ask" to his right neighbor. All further requests for a fork are answered with "no". When receiving "grant", the (nice) philosopher waits for the message "back" from his left neighbor who returns the fork, and then continues with thinking. When receiving "eat", the

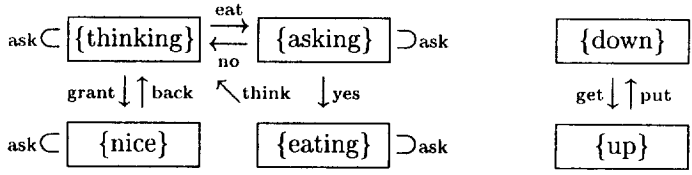

Fig. 4. The Abstract States of a Dining Philosopher and a Fork

(asking) philosopher waits for an answer from his right neighbor. If the answer is "no", he continues with thinking. Otherwise he takes his and his neighbor's fork and begins to eat. When he is no longer hungry, he puts the forks on the table, returns one of them to his neighbor and sends himself a message "think". When receiving this message, he begins to think again.

As the example shows, the proposed model can actually deal with rather difficult synchronization problems in a type-safe manner. Types express in which circumstances messages are supported, and a type checker ensures that only supported messages are received and each object can handle all supported messages. For example, each philosopher can be sure that he does not receive a message "eat" while he is "nice".

The type model cannot ensure that the solution of this problem is free of deadlocks because there is no way to ensure that all users send messages as expected. For example, a philosopher always remains in state "nice" if his neighbor does not return his fork. The type model also cannot prevent live-locks. But with this type model it is impossible that an object does not understand a message in its buffer.

The example in Fig. 3 also shows that synchronization sometimes requires additional messages and message arguments. A message "ask" contains a reference to the sending philosopher as a parameter; the answer shall be returned to this parameter. The answer also contains this parameter, although the receiver of the answer knows it. But it is still necessary to have this parameter: The parameter of "ask" is associated with a type mark which allows the receiver to send a reply. In order to avoid uncontrolled aliasing (since appropriate type splitting is impossible), the philosopher cannot keep the type mark describing his own abstract state. The answer returns the (updated) type mark so that the philosopher can continue to send messages to himself.

5.2 Object Types as Automata

Usually, descriptive object types correspond to finite automata: Activating sets represent states, acceptable messages represent state transitions. Fig. 4 shows the automata corresponding to the types of philosophers and forks.

It is not possible to represent all protocols as finite automata. Even if possible, it may not be desirable to represent an involved protocol as an object type because it can be difficult for a programmer to handle many different type

marks. But, it is always possible to use arbitrary approximations to such protocols: A type of the form $\{\} \mid d_1\langle\overline{\varphi}_1\rangle\{\}\rightarrow\{\} + \cdots + d_n\langle\overline{\varphi}_n\rangle\{\}\rightarrow\{\}$ is a simple first approximation of a protocol supporting the messages $d_1, \ldots, d_n$; the corresponding automaton has only one state. The type model ensures that an object of this type always accepts each supported message. If the object accepts a message that should not be received because of a constraint on message orders not expressed in the type, the object can raise an exception. A constraint on message orders expressed in a type is checked statically, while one not expressed in the type is checked dynamically. The first approximation can be improved by adding further abstract states (and constraints on message orders) to the type.

Since activating sets can hold an unlimited number of state descriptors, an automaton corresponding to an object type can, in principle, have an unlimited number of states. However, without some extensions of the model, we cannot make much use of this flexibility. If we extend the model with integer variables (over given ranges) as generic parameters and allow the corresponding integers to specify the number of occurrences of state descriptors in activating sets and be used in conditions of if-then-else-expressions, we can actually use object types that correspond to automata with an unlimited number of states. For example, σ_{Su} is the type of a data store that always accepts put-messages, but only as many get-messages as there are elements in the store:

$$\sigma_{\mathrm{Su}} = \{\} \mid \mathrm{put}\langle s\rangle\{\}\rightarrow\{\mathrm{full}\} + \mathrm{get}\langle\sigma_{\mathrm{B}}\rangle\{\mathrm{full}\}\rightarrow\{\}$$

This type corresponds to an automaton with an unlimited number of states. It is no more difficult to deal with σ_{Su} than with types corresponding to finite automata. For example, it is easy to show $\sigma_{\mathrm{Si}} \leq \sigma_{\mathrm{Su}} \leq \sigma_{\mathrm{S}}$. But integer variables as generic parameters are needed, for example, to specify a closed process that performs different actions on such data stores, depending on the number of elements.

6 Related Work

Much work on types for concurrent languages and models has been done. The majority of this work is based on Milner's π-calculus [15, 17] and similar calculi. Especially, the problem of inferring most general types was considered by Gay [5] and Vasconcelos and Honda [28]. Nierstrasz [19], Pierce and Sangiorgi [20], Vasconcelos [27] and Kobayashi and Yonezawa [10] deal with subtyping in such calculi. But their type models differ in an important aspect from the one presented in this work: They cannot represent constraints on the order of messages and ensure statically that all sent messages will be processed.

Several proposals [10, 20, 27] support subtyping in a similar way as sequential object-oriented languages based on the typed λ-calculus: A type of an active object specifies the set of messages that will be accepted by all instances; a subtype specifies an extended set of messages. Some of these proposals [20, 27] ensure that all sent messages will be processed, but do not support constraints on

the sequence of messages. The proposal of Kobayashi and Yonezawa [10] ensures neither message processing nor constraints on message sequences.

A large amount of work based on "path expressions" [3, 26] and, more recently, process algebra [2, 15] shows that reasoning about the order of messages in concurrent systems is quite difficult. Not much work was done on type models able to deal with constraints on message sequences because of the difficulty of this problem. Nierstrasz [19] argues that it is essential for a type model to regard an object as a process in a process calculus. He proposes "regular types" and "request substitutability" as foundations of subtyping. However, his very general results are not concrete enough to develop a static type system from them, especially because his approach does not consider aliases.

The proposal of Nielson and Nielson [18] can deal with constraints on message sequences. As in the type model proposed in this paper, types in their proposal are based on a process algebra, and a type checker updates type information while walking through a process expression. However, their proposal does not control aliases; types are regarded as contracts between an object and a single user, not as a contract between an object and the whole set of its users. Thus, their type model cannot ensure that all sent messages are understood. But subtyping is supported so that instances of subtypes preserve the properties expressed in supertypes; if a program corresponding to a supertype sends only understood messages, also a program corresponding to a subtype does so. Because types in their model specify the communication between processes completely, subtyping is rather restricted.

The present work improves earlier work on the process type model [21, 22, 23]. These earlier type models also support subtyping and ensure statically that all sent messages are understood, although constraints on message sequences are considered. The type model in the present work uses a new type representation that has several advantages over the old one:

- It provides better support for the coordination of users. In the earlier models, a user sometimes had to ask the object if some messages were acceptable. In the present model, users coordinate themselves without asking the object.
- The present model supports more efficient type checking. In some versions of the process type model, type checking was exponential in time, or even undecidable. Now, type checking time is quadratic in the worst case.
- The earlier models did not support genericity.

A type representation slightly similar to the one used in this work was proposed in [25]. But that proposal does not deal with subtyping and genericity and does not provide a formal analysis. A more thorough and formal treatment of the present work can be found in [24].

Future work. The work on this type model is not yet finished. Currently, it is not possible to specify in types which response is expected from the receiver of a message. There is ongoing work to make it possible for a type checker to ensure statically that the receiver of a message sends an appropriate reply. More expressive behavior descriptions as in [11] shall also be considered. Furthermore,

the type model shall be adapted for different kinds of communication, including synchronous message passing, restricted buffer sizes, and reordering of messages in message buffers. Several kinds of "fine tuning", especially considering explicit self-references, shall help to reduce the necessary syntactical overhead of using this type model.

7 Conclusions

The results of this work show that it is indeed feasible to regard a type as a contract between an (active) object and the unity of all users. Types specify constraints on the expected sequences of messages. A type checker can ensure statically that concurrent users are actually coordinated so that all sequences of messages sent to an object conform to the object's type, and the object accepts all type-conforming messages. Subtyping, genericity and separate compilation can be supported.

References

1. Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* The MIT Press, 1986.
2. J. C. M. Baeten and W. P. Weijland. *Process Algebra*, volume 18 of *Cambridge Tracts in Theoretical Computer Science.* Cambridge University Press, 1990.
3. R. H. Campbell and A. N. Habermann. The specification of process synchronization by path expressions. In E. Gelenbe, editor, *Proceedings of the International Symposium on Operating Systems*, volume 16 of *Lecture Notes in Computer Science*, pages 89–102. Springer-Verlag, 1974.
4. Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–522, 1985.
5. Simon J. Gay. A sort inference algorithm for the polyadic π-calculus. In *Conference Record of the 20th Symposium on Principles of Programming Languages*, January 1993.
6. Matthew Hennessy. *Algebraic Theory of Processes.* The MIT Press, 1988.
7. Carl Hewitt. Viewing control structures as patterns of passing messages. *Journal of Artificial Intelligence*, 8(3), 1977.
8. C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.
9. Kohei Honda and Mario Tokoro. An object calculus for asynchronous communication. In Pierre America, editor, *Proceedings ECOOP'91*, volume 512 of *Lecture Notes in Computer Science*, pages 141–162, Geneva, Switzerland, July 1991. Springer-Verlag.
10. Naoki Kobayashi and Akinori Yonezawa. Type-theoretic foundations for concurrent object-oriented programming. *ACM SIGPLAN Notices*, 29(10):31–45, October 1994. Proceedings OOPSLA'94.
11. Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, November 1994.

12. Satoshi Matsuoka and Akinori Yonezawa. Analysis of inheritance anomaly in object-oriented concurrent programming languages. In Gul Agha, Peter Wegner, and Akinori Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*. The MIT Press, 1993.
13. Bertrand Meyer. Systematic concurrent object-oriented programming. *Communications of the ACM*, 36(9):56–80, September 1993.
14. Bertrand Meyer. *Reusable Software: The Base Object-Oriented Component Libraries*. Prentice-Hall, Englewood Cliffs, NJ, 1994.
15. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes (parts I and II). *Information and Computation*, 100:1–77, 1992.
16. Robin Milner. *Communication and Concurrency*. Prentice-Hall, New York, 1989.
17. Robin Milner. The polyadic π-calculus: A tutorial. Technical Report ECS-LFCS-91-180, Dept. of Comp. Sci., Edinburgh University, 1991.
18. Flemming Nielson and Hanne Riis Nielson. From CML to process algebras. In *Proceedings CONCUR'93*, volume 715 of *Lecture Notes in Computer Science*, pages 493–508. Springer-Verlag, 1993.
19. Oscar Nierstrasz. Regular types for active objects. *ACM SIGPLAN Notices*, 28(10):1-15, October 1993. Proceedings OOPSLA'93.
20. Benjamin Pierce and Davide Sangiorgi. Typing and subtyping for mobile processes. In *Proceedings LICS'93*, 1993.
21. Franz Puntigam. Flexible types for a concurrent model. In *Proceedings of the Workshop on Object-Oriented Programming and Models of Concurrency*, Torino, June 1995.
22. Franz Puntigam. Type specifications with processes. In *Proceedings FORTE'95*. IFIP WG 6.1, October 1995.
23. Franz Puntigam. Types for active objects based on trace semantics. In Elie Najm et al., editor, *Proceedings of the 1st IFIP Workshop on Formal Methods for Open Object-based Distributed Systems*, Paris, France, March 1996. IFIP WG 6.1, Chapman & Hall.
24. Franz Puntigam. Coordination requirements expressed in types for active objects. Technical report, Institut für Computersprachen, Technische Universität Wien, Vienna, Austria, 1997. Electronically available under http://www.complang.tuwien.ac.at/franz/papers/ecoop97tr.ps.gz.
25. Franz Puntigam. Types that reflect changes of object usability. In *Proceedings of the Joint Modular Languages Conference*, volume 1204 of *Lecture Notes in Computer Science*, Linz, Austria, March 1997. Springer-Verlag.
26. Jan van den Bos, Rinus Plasmeijer, and Jan Stroet. Process communication based on input specifications. *ACM Transactions on Programming Languages and Systems*, 3(3):224–250, July 1981.
27. Vasco T. Vasconcelos. Typed concurrent objects. In *Proceedings ECOOP'94*, volume 821 of *Lecture Notes in Computer Science*, pages 100–117. Springer-Verlag, 1994.
28. Vasco T. Vasconcelos and Kohei Honda. Principal typing schemes in a polyadic pi-calculus. In *Proceedings CONCUR'93*, July 1993.
29. Peter Wegner and Stanley B. Zdonik. Inheritance as an incremental modification mechanism or what like is and isn't like. In S. Gjessing and K. Nygaard, editors, *Proceedings ECOOP'88*, volume 322 of *Lecture Notes in Computer Science*, pages 55–77. Springer-Verlag, 1988.