Dynamic Type Information in Process Types

Franz Puntigam

Technische Universität Wien, Institut für Computersprachen, Argentinierstr. 8,
A-1040 Vienna, Austria.
franz@complang.tuwien.ac.at

Abstract. Static checking of process types ensures that each object accepts all messages received from concurrent clients, although the set of acceptable messages can depend on the object's state. However, conventional approaches of using dynamic type information (e.g., checked type casts) are not applicable in the current process type model, and the typing of self-references is too restrictive. In this paper a refinement of the model is proposed. It solves these problems so that it is easy to handle, for example, heterogeneous collections.

1 Introduction

The process type model was proposed as a statically checkable type model for concurrent and distributed systems based on active objects [10, 11]. A process type specifies not only a set of acceptable messages, but also constraints on the sequence of these messages. Type safety can be checked statically by ensuring that each object reference is associated with an appropriate *type mark* which specifies all message sequences accepted by the object via this reference. The object accepts messages sent through different references in arbitrary interleaving. A type mark is a limited resource that represents a "claim" to send messages. It can be used up, split into several, more restricted type marks or forwarded from one user to another, but it must not be exceeded by any user.

Support of using dynamic type information shall be added. But, checked type casts break type safety: An object's dynamic type is not a "claim" to send messages. This problem is solved by checking against dynamic type marks.

Each object has better knowledge about its own state than about the other objects' states. The additional knowledge can be used in changing the self-references' type marks in a very flexible and still type-safe way.

The rest of the paper is structured as follows: An improved version of the typed process calculus presented in [11] is introduced in Sect. 2. Support of using dynamic type information is added in Sect. 3, a more flexible typing of self-references in Sect. 4. Static type checking is dealt with in Sect. 5.

2 A Typed Calculus of Active Objects

The proposed calculus describes systems composed of active objects that communicate through asynchronous message passing. An object has its own thread

of execution, a behavior, an identifier, and an unlimited buffer of received messages. According to its behavior, an object accepts messages from its buffer, sends messages to other objects, and creates new objects. All messages are received and accepted in the same logical order as they were sent.

We use following notation. *Constants* (denoted by $x, y, z, \ldots$) are used as message selectors and in types. *Identifiers* ($u, v, w, \ldots$) in the infinite set $\mathbb{V}$ are used as object and procedure identifiers as well as formal parameters. For each symbol e, $\tilde{e}$ is an abbreviation of $e_1, \ldots, e_n$; e.g., $\tilde{x}$ is a sequence of constants. $\{\tilde{e}\}$ denotes the smallest set containing $\tilde{e}$, and $|\tilde{e}|$ the length of the sequence. For each symbol $\cdot$, $\tilde{e}\cdot g$ stands for $e_1\cdot g, \ldots, e_n\cdot g$, and $\tilde{e}\cdot\tilde{g}$ for $e_1\cdot g_1, \ldots, e_n\cdot g_n$ ($|\tilde{e}| = |\tilde{g}|$).

Processes ($p, q, \ldots$) specify object behavior. Their syntax is:

$$
\begin{array}{rcl}
p & ::= & \{\tilde{s}\} \mid u.m;p \mid \underline{u} := (\langle\tilde{\underline{u}}\rangle p{:}\varphi); q \mid \underline{v}\$u\langle\tilde{\alpha},\tilde{u}\rangle; p \mid u\langle\tilde{\alpha},\tilde{u}\rangle \mid u{=}v\,?\,p \mid q \\
s & ::= & x\langle\tilde{\underline{u}}\rangle; p \\
m & ::= & x\langle\tilde{\alpha},\tilde{u}\rangle \\
\alpha & ::= & \tau \mid \varphi \\
\tau & ::= & \{\tilde{a}\}[\tilde{x}] \mid *\underline{u}\{\tilde{a}\}[\tilde{x}] \mid \sigma + \tau \mid u \\
a & ::= & x\langle\tilde{\underline{u}}{:}\tilde{\mu},\tilde{\alpha}\rangle[\tilde{y}]\triangleright[\tilde{z}] \\
\varphi & ::= & \langle\tilde{\underline{u}}{:}\tilde{\mu},\tilde{\alpha}\rangle\tau \mid *\underline{u}\langle\tilde{\underline{u}}{:}\tilde{\mu},\tilde{\alpha}\rangle\tau \mid u \\
\mu & ::= & \mathrm{t} \mid \mathrm{ot} \mid \mathrm{pt}
\end{array}
$$

A *selector* $\{\tilde{s}\}$ specifies a possibly empty, finite set of *guarded processes* ($r, s, \ldots$). Each s_i is of the form $x\langle\tilde{\underline{u}}\rangle; p$, where x is a message selector, $\tilde{u}$ a list of formal parameters, and p a process to be executed if s_i is selected. An s_i is selectable if the first received message in the object's buffer is of the form $x\langle\tilde{\alpha},\tilde{v}\rangle$, where $\tilde{\alpha},\tilde{v}$ are arguments to be substituted for the parameters $\tilde{u}$ ($|\tilde{\alpha},\tilde{v}| = |\tilde{u}|$); $\tilde{\alpha}$ is a list of types and $\tilde{v}$ a list of object and procedure identifiers. A process $u.m;p$ sends a message m to the object with identifier u, and then behaves as p. A *procedure definition* $\underline{u} := (\langle\tilde{\underline{u}}\rangle p{:}\varphi); q$ introduces an identifier u of a procedure of type φ and then behaves as q; the procedure takes $\tilde{u}$ as parameters and specifies the behavior p. A process $\underline{v}\$u\langle\tilde{\alpha},\tilde{u}\rangle; p$ creates a new object with identifier v and then behaves as p; the new object behaves as $u\langle\tilde{\alpha},\tilde{u}\rangle$. A *call* $u\langle\tilde{\alpha},\tilde{u}\rangle$ behaves as specified by the procedure with identifier u. A *conditional expression* $u{=}v\,?\,p \mid q$ behaves as p if the identifiers u and v are equal, otherwise as q.

There are two kinds of types, object types ($\pi, \varrho, \sigma, \tau, \ldots$) and procedure types ($\varphi, \psi, \ldots$). A type is denoted by $\alpha, \beta, \gamma, \ldots$ if its kind does not matter. Furthermore, there are meta-types ($\mu, \nu, \ldots$): "ot" is the type of all object types, "pt" the type of all procedure types, and "t" the type of all types of any kind.

The *activating set* $[\tilde{x}]$ of an object type $\{\tilde{a}\}[\tilde{x}]$ is a multi-set of constants, the *behavior descriptor* $\{\tilde{a}\}$ a finite collection of *message descriptors* ($a, b, c, \ldots$). A message descriptor $x\langle\tilde{\underline{u}}{:}\tilde{\mu},\tilde{\alpha}\rangle[\tilde{y}]\triangleright[\tilde{z}]$ describes a message with selector x, type parameters $\tilde{u}$ of meta-types $\tilde{\mu}$, and parameters of types $\tilde{\alpha}$. The $\tilde{u}$ can occur in $\tilde{\alpha}$. The message descriptor is *active* if the activating set $[\tilde{x}]$ contains all constants in the multi-set $[\tilde{y}]$ (*in-set*). When a corresponding message is sent (or accepted), the type is updated by removing the constants in the in-set from the activating set and adding those in the multi-set $[\tilde{z}]$ (*out-set*). Using type updating repeatedly

for all active message descriptors, the type specifies a set of acceptable message sequences. An expression $*\underline{u}\{\tilde{a}\}[\tilde{x}]$ is a recursive version of an object type. A type combination $\sigma + \tau$ specifies a set of acceptable message sequences that includes all arbitrary interleavings of acceptable message sequences specified by σ and τ. An identifier u can be used as type parameter.

A procedure type $\langle \underline{\tilde{u}}{:}\tilde{\mu}, \tilde{\alpha}\rangle\tau$ specifies that a procedure of this type takes type parameters $\tilde{u}$ of meta-types $\tilde{\mu}$ and parameters of types $\tilde{\alpha}$. Objects behaving according to the procedure accept messages as specified by τ.

For example, a procedure $B := (\langle \underline{u}\rangle\{\mathrm{put}\langle \underline{v}\rangle; \{\mathrm{get}\langle \underline{w}\rangle; w.\mathrm{back}\langle v\rangle; B\langle u\rangle\}\}{:}\varphi_B)$ specifies the behavior of a buffer accepting "put" and "get" in alternation and sending "back" to the argument of "get" after receiving "get". The type of B is given by $\varphi_B =_{\mathrm{def}} \langle \underline{u}{:}\mathrm{t}\rangle\{\mathrm{put}\langle u\rangle[\mathrm{e}]\triangleright[\mathrm{f}], \mathrm{get}\langle\{\mathrm{back}\langle u\rangle[\mathrm{once}]\triangleright[]\}[\mathrm{once}]\rangle[\mathrm{f}]\triangleright[\mathrm{e}]\}[\mathrm{e}]$. The type of a procedure specifying the behavior of an infinite buffer that accepts as many "get" messages as there are elements in the buffer can be given by $\varphi_{BI} =_{\mathrm{def}} \langle \underline{u}{:}\mathrm{t}\rangle\{\mathrm{put}\langle u\rangle[]\triangleright[\mathrm{f}], \mathrm{get}\langle\{\mathrm{back}\langle u\rangle[\mathrm{once}]\triangleright[]\}[\mathrm{once}]\rangle[\mathrm{f}]\triangleright[]\}[]$.

Each underlined occurrence of an identifier binds this and all following occurrences of the identifier. An occurrence is free if it is not bound. *Free*(e) denotes the set of all identifiers occurring free in e. Two processes are regarded as equal if they can be made identical by renaming bound identifiers (α-conversion) and repeatedly applying the equations $\{\tilde{r}, s\} = \{\tilde{r}, s, s\}$ and $\{\tilde{r}, s, \tilde{s}\} = \{\tilde{r}, \tilde{s}, s\}$ to selectors. Two types are regarded as equal if they can be made identical by renaming bound identifiers (α-conversion) and applying these equations:

$$\{\tilde{a}, b\} = \{\tilde{a}, b, b\} \qquad \{\tilde{a}, c, \tilde{c}\} = \{\tilde{a}, \tilde{c}, c\} \qquad [\tilde{x}, y, \tilde{y}] = [\tilde{x}, \tilde{y}, y]$$

$$\sigma + \tau = \tau + \sigma \qquad (\varrho + \sigma) + \tau = \varrho + (\sigma + \tau) \qquad \tau + \{\}[] = \tau$$

$$*\underline{u}\alpha = [*\underline{u}\alpha/u]\alpha \qquad \{\tilde{a}\}[\tilde{x}, \tilde{y}] = \{\tilde{a}\}[\tilde{x}] \quad (\tilde{y} \notin \mathit{Eff}\{\tilde{a}\})$$

$$\{x\langle \underline{\tilde{u}}{:}\tilde{\mu}, \tilde{\alpha}\rangle[\tilde{x}]\triangleright[\tilde{y}, \tilde{y}'], \tilde{a}\}[\tilde{z}] = \{x\langle \underline{\tilde{u}}{:}\tilde{\mu}, \tilde{\alpha}\rangle[\tilde{x}]\triangleright[\tilde{y}], \tilde{a}\}[\tilde{z}] \quad (\tilde{y}' \notin \{\tilde{x}\} \cup \mathit{Eff}\{\tilde{a}\})$$

$$\{\tilde{a}\}[\tilde{x}] + \{\tilde{c}\}[\tilde{y}] = \{\tilde{a}, \tilde{c}\}[\tilde{x}, \tilde{y}] \quad (\tilde{x} \in \mathit{Eff}\{\tilde{a}\};\ \tilde{y} \in \mathit{Eff}\{\tilde{c}\})$$

where $\mathit{Eff}\{x_1\langle \underline{\tilde{u}_1}{:}\tilde{\mu}_1, \tilde{\alpha}_1\rangle[\tilde{x}_1]\triangleright[\tilde{y}_1], \ldots, x_i\langle \underline{\tilde{u}_i}{:}\tilde{\mu}_i, \tilde{\alpha}_i\rangle[\tilde{x}_i]\triangleright[\tilde{y}_i]\}\} = \{\tilde{x}_1\} \cup \cdots \cup \{\tilde{x}_i\}$.

Subtyping on meta-types is defined by $\mathrm{ot} \leq \mathrm{t}$ and $\mathrm{pt} \leq \mathrm{t}$. Subtyping for types depends on an environment Π containing typing assumptions of the forms $u \leq \alpha$ and $\alpha \leq u$. The subtyping relation $\leq$ on types and the redundancy elimination relation $\coloneqq$ on object types are the reflexive, transitive closures of the rules:

$$\Pi \cup \{u \leq \alpha\} \vdash u \leq \alpha \qquad \Pi \cup \{\alpha \leq u\} \vdash \alpha \leq u$$

$$\frac{\Pi \vdash \tau + \varrho \coloneqq \sigma}{\Pi \vdash \sigma \leq \tau} \qquad \frac{\Pi \vdash \pi \leq \varrho \quad \Pi \vdash \sigma \leq \tau}{\Pi \vdash \pi + \sigma \leq \varrho + \tau} \qquad \frac{\Pi \vdash \tilde{\gamma} \leq \tilde{\alpha} \quad \Pi \vdash \sigma \leq \tau \quad \tilde{\nu} \leq \tilde{\mu}}{\Pi \vdash \langle \underline{\tilde{u}}{:}\tilde{\mu}, \tilde{\alpha}\rangle\sigma \leq \langle \underline{\tilde{u}}{:}\tilde{\nu}, \tilde{\gamma}\rangle\tau}$$

$$\Pi \vdash \{\tilde{a}, \tilde{c}\}[\tilde{x}] \coloneqq \{\tilde{a}\}[\tilde{x}] \quad (\forall c \in \{\tilde{c}\} \bullet \exists a \in \{\tilde{a}\} \bullet$$
$$c = x\langle \underline{\tilde{u}}{:}\tilde{\mu}, \tilde{\alpha}\rangle[\tilde{y}, \tilde{y}']\triangleright[\tilde{z}, \tilde{z}'] \ \wedge\ \tilde{z}' \notin \mathit{Eff}\{\tilde{a}\} \ \wedge$$
$$a = x\langle \underline{\tilde{u}}{:}\tilde{\nu}, \tilde{\gamma}\rangle[\tilde{y}]\triangleright[\tilde{z}, \tilde{z}'']\ \wedge\ \tilde{\mu} \leq \tilde{\nu}\ \wedge\ \Pi \vdash \tilde{\alpha} \leq \tilde{\gamma})$$

($\alpha \leq \beta$ and $\sigma \coloneqq \tau$ are simplified notations for $\emptyset \vdash \alpha \leq \beta$ and $\emptyset \vdash \sigma \coloneqq \tau$.) The relation $\coloneqq$ is used for removing redundant message descriptors from behavior descriptors. Using these definitions we can show, for example, $\varphi_{BI} \leq \varphi_B$.

A type $\{\tilde{a}\}[\tilde{x}]$ is *deterministic* if all message descriptors in $\{\tilde{a}\}$ have pairwise different message selectors. For each deterministic type σ and message m there is only one way of updating σ according to m.

A *system configuration* C contains expressions of the forms $u \mapsto \langle p, \tilde{m}\rangle$ and $u \mapsto \langle \tilde{u}\rangle p{:}\varphi$. The first expression specifies an object with identifier u, behavior p and sequence of received messages $\tilde{m}$. The second expression specifies a procedure of identifier u. C contains at most one expression $u \mapsto e$ for each $u \in \mathbb{V}$. $C[u_1 \mapsto e_1, \ldots, u_n \mapsto e_n]$ is a system configuration with expressions added to C, where $u_1, \ldots, u_n$ are pairwise different and do not occur to the left of $\mapsto$ in C.

For each $u \in \mathbb{V}$ the relation $\xrightarrow{u}$ on system configurations (defined by the rules given below) specifies all possible execution steps of object u. ($[\tilde{e}/\tilde{u}]f$ is the simultaneous substitution of the expressions $\tilde{e}$ for all free occurrences of $\tilde{u}$ in f.)

$$
\begin{aligned}
C[u \mapsto \langle \{x\langle \underline{\tilde{u}}\rangle; p,\ \tilde{s}\},\ x\langle \tilde{\alpha}, \tilde{v}\rangle,\ \tilde{m}\rangle] &\xrightarrow{u} C[u \mapsto \langle ([\tilde{\alpha}, \tilde{v}/\tilde{u}]p),\ \tilde{m}\rangle] \\
C[u \mapsto \langle v.m; p,\ \tilde{m}\rangle,\ v \mapsto \langle q, \tilde{m}'\rangle] &\xrightarrow{u} C[u \mapsto \langle p, \tilde{m}\rangle,\ v \mapsto \langle q, \tilde{m}', m\rangle] \\
C[u \mapsto \langle \underline{v} := (\langle \underline{\tilde{u}}\rangle p{:}\varphi); q,\ \tilde{m}\rangle] &\xrightarrow{u} C[u \mapsto \langle q, \tilde{m}\rangle,\ v \mapsto \langle \underline{\tilde{u}}\rangle p{:}\varphi] \\
C[u \mapsto \langle \underline{v}\$w\langle \tilde{\alpha}, \tilde{u}\rangle; p,\ \tilde{m}\rangle] &\xrightarrow{u} C[u \mapsto \langle p, \tilde{m}\rangle,\ v \mapsto \langle w\langle \tilde{\alpha}, \tilde{u}\rangle\rangle] \\
C[u \mapsto \langle v\langle \tilde{\alpha}, \tilde{u}\rangle,\ \tilde{m}\rangle,\ v \mapsto \langle \underline{\tilde{v}}\rangle p{:}\varphi] &\xrightarrow{u} C[u \mapsto \langle ([\tilde{\alpha}, \tilde{u}/\tilde{v}]p),\ \tilde{m}\rangle,\ v \mapsto \langle \underline{\tilde{v}}\rangle p{:}\varphi] \\
C[u \mapsto \langle v{=}v\,?\,p \mid q,\ \tilde{m}\rangle] &\xrightarrow{u} C[u \mapsto \langle p, \tilde{m}\rangle] \\
C[u \mapsto \langle v{=}w\,?\,p \mid q,\ \tilde{m}\rangle] &\xrightarrow{u} C[u \mapsto \langle q, \tilde{m}\rangle] \quad (v \neq w)
\end{aligned}
$$

$\longrightarrow$ denotes the closure of these relations over all $u \in \mathbb{V}$, and $\xrightarrow{*}$ the reflexive, transitive closure of $\longrightarrow$. $\xrightarrow{*}$ defines the operational semantics of object systems.

3 Dynamic Type Information

It seems to be easy to extend the calculus with a dynamic type checking concept: Processes of the form $\alpha{\leq}\beta\,?\,p \mid q$ are introduced, where α or β initially is a type parameter to be replaced with a (dynamic) type during computation. The typing assumption $\alpha \leq \beta$ (for α or β in $\mathbb{V}$) is statically known to hold in p. The relation $\xrightarrow{u}$ (for each $u \in \mathbb{V}$) is extended by the rules:

$$
\begin{aligned}
C[u \mapsto \langle \alpha{\leq}\beta\,?\,p \mid q,\ \tilde{m}\rangle] &\xrightarrow{u} C[u \mapsto \langle p, \tilde{m}\rangle] \quad (\alpha \leq \beta) \\
C[u \mapsto \langle \alpha{\leq}\beta\,?\,p \mid q,\ \tilde{m}\rangle] &\xrightarrow{u} C[u \mapsto \langle q, \tilde{m}\rangle] \quad (\alpha \not\leq \beta)
\end{aligned}
$$

For example, assume that a reference w is of type mark St[f], where St $=_{\text{def}}$ $\{\text{put}\langle \underline{u}{:}\text{t}, u\rangle[\text{e}]\triangleright[\text{f}],\ \text{get}\langle \text{R[one]}\rangle[\text{f}]\triangleright[\text{e}]\}$ and R $=_{\text{def}}$ $\{\text{back}\langle \underline{v}{:}\text{t}, v\rangle[\text{one}]\triangleright[]\}$. The process $w.\text{get}\langle w'\rangle; \{\text{back}\langle \underline{u}, \underline{v}\rangle; u{\leq}\text{St[e]}\,?\,v.\text{put}\langle \text{St[f]}, v\rangle; \{\} \mid \{\}\}$ (w' denotes the object's self-reference) is type-safe. First, "get" is sent to w; then "back" is accepted. If the type mark of v is a subtype of St[e], v is put into v. The type used as argument of "put" is St[f] because the type is updated when sending "put".

Type information is lost when types are updated. St[e] is a supertype of v's type mark u; but, $\text{put}\langle u, v\rangle$ cannot be sent to v: So far it is not possible to specify that v's type mark is equal to u, except that u is updated. To solve the problem,

processes of the form $\sigma{\leq}\tau\,?\,\underline{v}; p \mid q$ are introduced, where σ initially is a type parameter to be replaced with an object type. The relation $\xrightarrow{u}$ is extended:

$$C[u \mapsto \langle \sigma{\leq}\tau\,?\,\underline{v}; p \mid q, \tilde{m}\rangle] \xrightarrow{u} C[u \mapsto \langle([\varrho/v]p), \tilde{m}\rangle] \quad (\tau + \varrho \doteq \sigma)$$
$$C[u \mapsto \langle \sigma{\leq}\tau\,?\,\underline{v}; p \mid q, \tilde{m}\rangle] \xrightarrow{u} C[u \mapsto \langle q, \tilde{m}\rangle] \quad (\sigma \not\leq \tau)$$

v in p is replaced with a dynamically determined object type ϱ which specifies the difference between σ and τ. This version of the above process keeps type information: $w.\mathrm{get}\langle w'\rangle; \{\mathrm{back}\langle \underline{u}, \underline{v}\rangle; u{\leq}\mathrm{St[e]}\,?\,\underline{u}'; v.\mathrm{put}\langle \mathrm{St[f]} + u', v\rangle; \{\} \mid \{\}\}$. The type $\mathrm{St[f]} + u'$ equals u after removing e and adding f to the activating set.

4 Typing Self-References

Without special treatment, static typing of self-references is very restrictive: If an object behaves according to a type τ, the type mark σ of a self-reference is a subtype of τ. Both, σ and τ specify all messages that may return results of possibly needed services. In general, it is impossible to determine statically which services are needed. Hence, σ and τ usually specify much larger sets of message sequences than needed. This problem can be solved because each object has much more knowledge about its own state than other objects: Before a message with a self-reference as argument is sent to a server, this reference's type mark is selected so that it supports the messages returned by the server. At the same time the object's type is extended correspondingly. The object's type and self-reference's type mark are determined dynamically as needed in the computation.

We extend the calculus: The distinguished name "self" can be used as arguments in procedures; "self" is replaced with a self-reference when the procedure is called. The fifth rule in the definition of $\xrightarrow{u}$ has to be replaced with:

$$C[u \mapsto \langle v\langle\tilde{\alpha}, \tilde{u}\rangle, \tilde{m}\rangle, v \mapsto \langle\tilde{\underline{v}}\rangle p{:}\varphi] \xrightarrow{u} C[u \mapsto \langle([u/\mathrm{self}][\tilde{\alpha}, \tilde{u}/\tilde{v}]p), \tilde{m}\rangle, v \mapsto \langle\tilde{\underline{v}}\rangle p{:}\varphi]$$

The type mark of an occurrence of "self" as argument is equal to the type of the corresponding formal parameter. (Elements of $\mathbb{V} \cup \{\mathrm{self}\}$ are denoted by $o, \ldots$)

An example demonstrates the use of self and dynamic type comparisons:

$$\underline{w} := (\langle \underline{u}, \underline{u}', \underline{v}, \underline{w}'\rangle\, u{\leq}\mathrm{St[f]}\,?\,v.\mathrm{get}\langle\mathrm{self}\rangle; \{\mathrm{back}\langle \underline{u}, \underline{v}\rangle; w\langle u, u', v, w'\rangle\} \mid w'\langle u, v\rangle) \quad : \quad \langle \underline{u}{:}\mathrm{t}, \underline{u}'{:}\mathrm{ot}, u, \langle \underline{u}{:}\mathrm{t}, u\rangle u'\rangle u'); \ \ldots$$

The procedure w takes as parameters a type u, an object type u', an identifier v of type u and a procedure identifier w' of type $\langle \underline{u}{:}\mathrm{t}, u\rangle u'$. If v is a non-empty store, an element is got from the store and w is called recursively with the element and its type as arguments. Otherwise w' is called with u and v as arguments. The type mark of self is R[one], as specified by St. The corresponding message "back" is accepted if the expression containing self is executed.

5 Static Type Checking

An important result is that the process type model with the proposed extensions supports static type checking. Type checking rules are given in Appendix A.

Theorem 1. *Let* $C = \{u_1 \mapsto \langle p_1 \rangle, \ldots, u_n \mapsto \langle p_n \rangle\}$ *be a system configuration and* $\sigma_{i,j}$ *and* τ_i $(1 \leq i, j \leq n)$ *object types such that* $\tau_i \leq \sigma_{i,1} + \cdots + \sigma_{i,n}$ *and* $\emptyset, \{u_1{:}\sigma_{1,i}, \ldots, u_n{:}\sigma_{n,i}\} \vdash \langle p_i{:}\tau_i \rangle$ *as defined by the type checking rules. Then, for each system configuration* D *with* $C \xrightarrow{*} D$, *if* D *contains an expression* $u \mapsto \langle p, m, \tilde{m} \rangle$, *there exists a system configuration* E *with* $D \xrightarrow{u} E$.

(The proof is omitted because of lack of space.) Provided that type checking succeeds for an initial system configuration, if an object has a nonempty buffer of received messages, the execution of this object cannot be blocked. Especially, the next message in the buffer is acceptable. Theorem 1 also implies that separate compilation is supported: If a process p_i in C is replaced with another p_i' satisfying the conditions, the consequences of the theorem still hold.

6 Related Work

Much work on types for concurrent languages and models was done. The majority of this work is based on Milner's π-calculus [3, 4] and similar calculi. Especially, the problem of inferring most general types was considered by Gay [2] and Vasconcelos and Honda [14]. Nierstrasz [5], Pierce and Sangiorgi [8], Vasconcelos [13], Colaco, Pantel and Sall'e [1] and Ravara and Vasconcelos [12] deal with subtyping. But their type models differ in an important aspect from the process type model: They cannot represent constraints on message sequences and ensure statically that all sent messages are acceptable; the underlying calculus does not keep the message order.

The proposals of Nielson and Nielson [6] can deal with constraints on message sequences. As in the process type model, a type checker updates type information while walking through an expression. However, their type model cannot ensure that all sent messages are understood, and dealing with dynamic type information is not considered.

The process type model can ensure statically that all sent messages are understood, although the set of acceptable messages can change. Therefore, this model is a promising approach to strong, static types for concurrent and distributed applications based on active objects. However, it turned out that the restrictions caused by static typing are an important difficulty in the practical applicability of the process type model. So far it was not clear how to circumvent this difficulty without breaking type safety. New techniques for dealing with dynamic type information (as presented in this paper) had to be found.

The approach of Najm and Nimour [7] has a similar goal as process types. However, this approach is more restrictive and cannot deal with dynamic type information, too.

The work presented in this paper refines previous work on process types [9–11]. The major improvement is the addition of dynamic type comparisons and a special treatment of self-references. Some changes to earlier versions of the process type model were necessary. For example, the two sets of type checking rules presented in [11] had to be modified and combined into a single set so that it was possible to introduce "self".

Dynamic type comparisons exist in nearly all recent object-oriented programming languages. However, the usual approaches cannot be combined with process types because these approaches do not distinguish between object types and type marks and cannot deal with type updates. Conventional type models do not have problems with the flexibility of typing self-references.

7 Conclusions

The process type model is a promising basis for strongly typed concurrent programming languages. The use of dynamic type information can be supported in several ways so that the process type model becomes more flexible and useful. An important property is kept: A type checker can ensure statically that clients are coordinated so that all received messages can be accepted in the order they were sent, even if the acceptability of messages depends on the server's state.

References

1. J.-L. Colaco, M. Pantel, and P. Sall'e. A set-constraint-based analysis of actors. In *Proceedings FMOODS '97*, Canterbury, United Kingdom, July 1997. Chapman & Hall.
2. 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.
3. Robin Milner. The polyadic π-calculus: A tutorial. Technical Report ECS-LFCS-91-180, Dept. of Comp. Sci., Edinburgh University, 1991.
4. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes (parts I and II). *Information and Computation*, 100:1–77, 1992.
5. Oscar Nierstrasz. Regular types for active objects. *ACM SIGPLAN Notices*, 28(10):1–15, October 1993. Proceedings OOPSLA'93.
6. Flemming Nielson and Hanne Riis Nielson. From CML to process algebras. In *Proceedings CONCUR'93*, number 715 in Lecture Notes in Computer Science, pages 493–508. Springer-Verlag, 1993.
7. E. Najm and A. Nimour. A calculus of object bindings. In *Proceedings FMOODS '97*, Canterbury, United Kingdom, July 1997.
8. Benjamin Pierce and Davide Sangiorgi. Typing and subtyping for mobile processes. In *Proceedings LICS'93*, 1993.
9. Franz Puntigam. Flexible types for a concurrent model. In *Proceedings of the Workshop on Object-Oriented Programming and Models of Concurrency*, Torino, June 1995.

10. Franz Puntigam. Types for active objects based on trace semantics. In Elie Najm et al., editor, *Proceedings FMOODS '96*, Paris, France, March 1996. IFIP WG 6.1, Chapman & Hall.
11. Franz Puntigam. Coordination requirements expressed in types for active objects. In Mehmet Aksit and Satoshi Matsuoka, editors, *Proceedings ECOOP '97*, number 1241 in Lecture Notes in Computer Science, Jyväskylä, Finland, June 1997. Springer-Verlag.
12. António Ravara and Vasco T. Vasconcelos. Behavioural types for a calculus of concurrent objects. In *Proceedings Euro-Par '97*, Lecture Notes in Computer Science. Springer-Verlag, 1997.
13. Vasco T. Vasconcelos. Typed concurrent objects. In *Proceedings ECOOP'94*, number 821 in Lecture Notes in Computer Science, pages 100–117. Springer-Verlag, 1994.
14. Vasco T. Vasconcelos and Kohei Honda. Principal typing schemes in a polyadic pi-calculus. In *Proceedings CONCUR'93*, July 1993.

A Type Checking Rules

$$\frac{\Pi \vdash \{\tilde{a}\}[\tilde{y}] \leq \tau \quad \forall 1 \leq i \leq n \bullet \Pi, \Gamma[\tilde{u}_i{:}\tilde{\mu}_i, \tilde{v}_i{:}\tilde{\alpha}_i] \vdash \langle p_i{:}\{\tilde{a}\}[\tilde{z}_i]\rangle}{\Pi, \Gamma \vdash \langle \{x_1\langle \underline{\tilde{u}}_1, \underline{\tilde{v}}_1\rangle; p_1, \ldots, x_n\langle \underline{\tilde{u}}_n, \underline{\tilde{v}}_n\rangle; p_n\}{:}\tau\rangle} \quad (1) \qquad \text{SEL}$$

$$\frac{\Pi \vdash \sigma \leq \{a\}[\tilde{x}] + \varrho \quad \Pi, \Gamma[u{:}\{a\}[\tilde{y}] + \varrho] \vdash \langle \tilde{\alpha}{:}\tilde{\mu}, \tilde{o}{:}([\tilde{\alpha}/\tilde{u}]\tilde{\gamma}), p{:}\tau\rangle}{\Pi, \Gamma[u{:}\sigma] \vdash \langle (u.x\langle \tilde{\alpha}, \tilde{o}\rangle; p){:}\tau\rangle} \quad (2) \qquad \text{SEND}$$

$$\frac{\Pi, \emptyset[\tilde{w}_1{:}\tilde{\nu}, \tilde{w}_2{:}\tilde{\varphi}, u{:}\psi, \tilde{u}{:}\tilde{\mu}, \tilde{v}{:}\tilde{\alpha}] \vdash \langle \psi{:}\mathrm{pt}, p{:}\sigma\rangle \quad \Pi, \Gamma[u{:}\psi] \vdash \langle q{:}\tau\rangle}{\Pi, \Gamma \vdash \langle (\underline{u} := (\langle \underline{\tilde{u}}, \underline{\tilde{v}}\rangle p{:}{*}\underline{u}\psi); q){:}\tau\rangle} \quad (3) \qquad \text{DEF}$$

$$\frac{\Pi \vdash \varphi \leq \langle \underline{\tilde{u}}{:}\tilde{\mu}, \tilde{\gamma}\rangle\sigma \quad \Pi, \Gamma[u{:}([\tilde{\alpha}/\tilde{u}]\sigma), v{:}\varphi] \vdash \langle \tilde{\alpha}{:}\tilde{\mu}, \tilde{o}{:}([\tilde{\alpha}/\tilde{u}]\tilde{\gamma}), p{:}\tau\rangle}{\Pi, \Gamma[v{:}\varphi] \vdash \langle (\underline{u}\$v\langle \tilde{\alpha}, \tilde{o}\rangle; p){:}\tau\rangle} \qquad \text{NEW}$$

$$\frac{\Pi \vdash \varphi \leq \langle \underline{u}{:}\tilde{\mu}, \tilde{\gamma}\rangle\sigma \quad \Pi \vdash ([\tilde{\alpha}/\tilde{u}]\sigma) \leq \tau \quad \Pi, \Gamma[u{:}\varphi] \vdash \langle \tilde{\alpha}{:}\tilde{\mu}, \tilde{o}{:}([\tilde{\alpha}/\tilde{u}]\tilde{\gamma}), \{\}{:}\{\}[]\rangle}{\Pi, \Gamma[u{:}\varphi] \vdash \langle u\langle \tilde{\alpha}, \tilde{o}\rangle{:}\tau\rangle} \qquad \text{CALL}$$

$$\frac{\Pi, \Gamma[u{:}\varrho + \sigma] \vdash \langle ([u/v]p){:}\tau\rangle \quad \Pi, \Gamma[u{:}\varrho, v{:}\sigma] \vdash \langle q{:}\tau\rangle}{\Pi, \Gamma[u{:}\varrho, v{:}\sigma] \vdash \langle (u{=}v\,?\,p \mid q){:}\tau\rangle} \qquad \text{EQU}_1$$

$$\frac{\Pi, \Gamma[u{:}\varphi, v{:}\psi] \vdash \langle p{:}\tau\rangle \quad \Pi, \Gamma[u{:}\varphi, v{:}\psi] \vdash \langle q{:}\tau\rangle}{\Pi, \Gamma[u{:}\varphi, v{:}\psi] \vdash \langle (u{=}v\,?\,p \mid q){:}\tau\rangle} \qquad \text{EQU}_2$$

$$\frac{\Pi \cup \{u \leq \alpha\}, \Gamma[u{:}\mu] \vdash \langle p{:}\tau\rangle \quad \Pi, \Gamma[u{:}\mu] \vdash \langle \alpha{:}\mathrm{t}, q{:}\tau\rangle}{\Pi, \Gamma[u{:}\mu] \vdash \langle (u{\leq}\alpha\,?\,p \mid q){:}\tau\rangle} \quad (u \notin \mathit{Free}(\alpha)) \qquad \text{SUB}_1$$

$$\frac{\Pi \cup \{\alpha \leq u\}, \Gamma[u{:}\mu] \vdash \langle p{:}\tau\rangle \quad \Pi, \Gamma[u{:}\mu] \vdash \langle \alpha{:}\mathrm{t}, q{:}\tau\rangle}{\Pi, \Gamma[u{:}\mu] \vdash \langle (\alpha{\leq}u\,?\,p \mid q){:}\tau\rangle} \quad (u \notin \mathit{Free}(\alpha)) \qquad \text{SUB}_2$$

$$\frac{\Pi \cup \{u \leq \sigma\}, \Gamma[u{:}\mathrm{ot}, v{:}\mathrm{ot}] \vdash \langle p{:}\tau\rangle \quad \Pi, \Gamma[u{:}\mathrm{ot}] \vdash \langle \sigma{:}\mathrm{ot}, q{:}\tau\rangle}{\Pi, \Gamma[u{:}\mathrm{ot}] \vdash \langle (u{\leq}\sigma\,?\,\underline{v}; p \mid q){:}\tau\rangle} \quad (u \notin \mathit{Free}(\sigma)) \qquad \text{SPLIT}$$

$$\frac{\Pi \vdash \sigma \leq \varrho + \tau \quad \Pi, \Gamma[u{:}\varrho] \vdash \langle \tilde{e}{:}\tilde{g}\rangle}{\Pi, \Gamma[u{:}\sigma] \vdash \langle u{:}\tau, \tilde{e}{:}\tilde{g}\rangle} \qquad \text{OBJ}$$

$$\frac{\Pi, \Gamma \vdash \langle \tilde{e}{:}\tilde{g}, p{:}\sigma + \tau\rangle}{\Pi, \Gamma \vdash \langle \mathrm{self}{:}\sigma, \tilde{e}{:}\tilde{g}, p{:}\tau\rangle} \qquad \text{SELF}$$

$$\frac{\Pi \vdash \psi \leq \varphi \quad \Pi, \Gamma[u{:}\psi] \vdash \langle \tilde{e}{:}\tilde{g}\rangle}{\Pi, \Gamma[u{:}\psi] \vdash \langle u{:}\varphi, \tilde{e}{:}\tilde{g}\rangle} \qquad \text{PROC}$$

$$\frac{\Pi, \Gamma[u{:}\nu] \vdash \langle \tilde{e}{:}\tilde{g}\rangle \quad \nu \leq \mu}{\Pi, \Gamma[u{:}\nu] \vdash \langle u{:}\mu, \tilde{e}{:}\tilde{g}\rangle} \qquad \text{TPAR}$$

$$\frac{\forall 1 \leq i \leq n \bullet \Pi, \Gamma[u{:}\mathrm{ot}, \tilde{u}_i{:}\tilde{\mu}_i] \vdash \langle \tilde{\alpha}_i{:}\mathrm{t}, \{\}{:}\{\}[]\rangle \quad \Pi, \Gamma \vdash \langle \tilde{e}{:}\tilde{g}\rangle \quad \mathrm{ot} \leq \nu}{\Pi, \Gamma \vdash \langle {*}\underline{u}\{x_1\langle \underline{\tilde{u}}_1{:}\tilde{\mu}_1, \tilde{\alpha}_1\rangle[\tilde{y}_1]{\triangleright}[\tilde{z}_1], \ldots, x_n\langle \underline{\tilde{u}}_n{:}\tilde{\mu}_n, \tilde{\alpha}_n\rangle[\tilde{y}_n]{\triangleright}[\tilde{z}_n]\}[\tilde{y}]{:}\nu, \tilde{e}{:}\tilde{g}\rangle} \qquad \text{OBJT}_1$$

$$\frac{\Pi, \Gamma \vdash \langle \sigma{:}\mathrm{ot}, \tau{:}\mathrm{ot}, \tilde{e}{:}\tilde{g}\rangle \quad \mathrm{ot} \leq \mu}{\Pi, \Gamma \vdash \langle (\sigma + \tau){:}\mu, \tilde{e}{:}\tilde{g}\rangle} \qquad \text{OBJT}_2$$

$$\frac{\Pi, \Gamma[u{:}\mathrm{pt}, \tilde{u}{:}\tilde{\mu}] \vdash \langle \tilde{\alpha}{:}\mathrm{t}, \tau{:}\mathrm{ot}, \{\}{:}\{\}[]\rangle \quad \Pi, \Gamma \vdash \langle \tilde{e}{:}\tilde{g}\rangle \quad \mathrm{pt} \leq \nu}{\Pi, \Gamma \vdash \langle ({*}\underline{u}\langle \underline{\tilde{u}}{:}\tilde{\mu}, \tilde{\alpha}\rangle\tau){:}\nu, \tilde{e}{:}\tilde{g}\rangle} \qquad \text{PROCT}$$

(1) $\mathit{Act}\{\tilde{a}\}[\tilde{y}] = \{x_1\langle \underline{\tilde{u}}_1{:}\tilde{\mu}_1, \tilde{\alpha}_1\rangle[]{\triangleright}[\tilde{z}_1], \ldots, x_n\langle \underline{\tilde{u}}_n{:}\tilde{\mu}_n, \tilde{\alpha}_n\rangle[]{\triangleright}[\tilde{z}_n]\}$ where

$$\begin{aligned} \mathit{Act}\{\}[\tilde{x}] &= \{\} \\ \mathit{Act}\{x\langle \underline{\tilde{u}}{:}\tilde{\mu}, \tilde{\alpha}\rangle[\tilde{x}]{\triangleright}[\tilde{y}'], \tilde{a}\}[\tilde{x}, \tilde{z}] &= \{x\langle \underline{\tilde{u}}{:}\tilde{\mu}, \tilde{\alpha}\rangle[]{\triangleright}[\tilde{y}', \tilde{z}], \tilde{c}\} \quad (\mathit{Act}\{\tilde{a}\}[\tilde{x}, \tilde{z}] = \{\tilde{c}\}) \\ \mathit{Act}\{x\langle \underline{\tilde{u}}{:}\tilde{\mu}, \tilde{\alpha}\rangle[\tilde{x}]{\triangleright}[\tilde{y}'], \tilde{a}\}[\tilde{z}] &= \mathit{Act}\{\tilde{a}\}[\tilde{z}] \quad (\forall \tilde{x}' \bullet [\tilde{z}] \neq [\tilde{x}, \tilde{x}']) \end{aligned}$$

and $\{\tilde{a}\}[\tilde{y}]$ is deterministic.

(2) $a = x\langle \underline{\tilde{u}}{:}\tilde{\mu}, \tilde{\gamma}\rangle[\tilde{x}]{\triangleright}[\tilde{y}]$

(3) $\psi = \langle \underline{\tilde{u}}{:}\tilde{\mu}, \tilde{\alpha}\rangle\sigma$ and $\Gamma = \{\tilde{w}_1{:}\tilde{\nu}, \tilde{w}_2{:}\tilde{\varphi}, \tilde{w}_3{:}\tilde{\varrho}\}$