A Theory of Compositional Concurrent Objects

UNSW-CSE-TR-0210

Xiaogang Zhang and John Potter
School of Computer Science and Engineering
University of New South Wales, Australia
{xzhang,potter}@cse.unsw.edu.au

Abstract

This paper presents the theory of composition for concurrent object systems, based on an object modelling in the κ-calculus. The behaviour of a concurrent object can be modelled as the composition of a process representing the functional behaviour of the object with no constraint on its concurrent interactions, or synchronisation, and a process representing concurrency constraints to reduce the allowable concurrency and to avoid the states of exception. With this model, we use the κ-calculus, a process algebra with polars, to study the theory of composition of concurrent behaviours, investigate when and how concurrent behaviours can (or should) be composed with and separated from functional behaviours or other concurrent behaviours, identify relevant patterns and properties of concurrent behaviours, etc. Some generic properties of the behaviour composition, such the Identity Law and Associative Law, have been proven in this study.

Keywords: *object models, π-calculus, κ-calculus, concurrency constraints, concurrency controls, composition, synchronisation*

A Theory of Compositional Concurrent Objects

Xiaogang Zhang and John Potter
School of Computer Science and Engineering
University of New South Wales, Australia
{xzhang,potter}@cse.unsw.edu.au

Abstract

This paper presents the theory of composition for concurrent object systems, based on an object modelling in the κ-calculus. The behaviour of a concurrent object can be modelled as the composition of a process representing the functional behaviour of the object with no constraint on its concurrent interactions, or synchronisation, and a process representing concurrency constraints to reduce the allowable concurrency and to avoid the states of exception. With this model, we use the κ-calculus, a process algebra with polars, to study the theory of composition of concurrent behaviours, investigate when and how concurrent behaviours can (or should) be composed with and separated from functional behaviours or other concurrent behaviours, identify relevant patterns and properties of concurrent behaviours, etc. Some generic properties of the behaviour composition, such the Identity Law and Associative Law, have been proven in this study.

Keywords: *object models, π-calculus, κ-calculus, concurrency constraints, concurrency controls, composition, synchronisation*

1 Introduction

With the rapid development of communication and networks technology, concurrent and distributed computing systems have been playing a more and more important role. However, concurrency is hard to reason about. Programming for concurrent systems is a complicated, difficult, and problematical task, requires experience and skill. The typical approach for reducing the complexity of problems and isolating difficulties is to separate different issues and recompose as needed. Object-orientation provides reasonable compositional properties with its support for modularity, extensibility and reusability. However, with conventional OO techniques the concurrency issues are usually mixed with the functionality issues. This limits composability and causes the *inheritance anomaly* problem ([Matsuoka93]). The anomaly arises from the attempting to inherit code implementing synchronisation controls -- typically the synchronisation code must be re-implemented in subclasses, negating much of the reuse benefits gained from inheritance ([McHale94]). To solve this problem, many compositional schemes have been proposed for separating concurrency from functionality ([Aksit92], [McHale94], [Holmes97]), but these still lack a formal foundation for describing the semantics of those schemes of composition, for studying principles of composition, and for reasoning about the correctness of composition.

The π-calculus is algebra for mobile processes ([Milner92], [Milner96]), and provides a formal foundation for modelling systems with dynamic structure. It can be used to mathematically model concurrent and distributed processes, analyse their behaviour and identify deadlocks. With its ability to directly model dynamic reference structures, the π-calculus has been applied to modelling concurrent object systems ([Walker95], [Jones93], [Sangiorgi96], [Hüttel96], [Zhang97]). Some researchers ([Schneider97], [Zhang98A], [Zhang98B]) also have applied it in modelling compositional concurrent objects. This paper presumes the readers familiar with the π-calculus.

In the concurrent object model of [Zhang98A] and [Zhang98B], the behaviour of a concurrent object was described as the parallel composition of a process F representing the object's functional behaviour with no constraint on its concurrent interactions, and a process C representing the constraints on the object's concurrent behaviour. For example, the functionality of a buffer object can be described by $F \stackrel{\text{def}}{=} !n_{\text{r}}(x).M_{\text{r}}\langle x\rangle \mid !n_{\text{w}}(x).M_{\text{w}}\langle x\rangle$, where $n_{\text{r}}(x).M_{\text{r}}\langle x\rangle$ and $n_{\text{w}}(x).M_{\text{w}}\langle x\rangle$ represent the behaviour of the read and write methods respectively; each of them can have unlimited

invocations executing in parallel with no concern for any potential interference. To discipline those invocations, assume a synchronisation behaviour modelled by the control process $C \stackrel{\text{def}}{=} m_{\text{r}}(x).\overline{n}_{\text{r}}\langle x\rangle + m_{\text{w}}(x).\overline{n}_{\text{w}}\langle x\rangle$, where the choice operator in fact represents a mutual exclusion lock on those methods. Then the parallel composition of the two processes, $(\nu\, n)(C \mid F)$, will be weakly bisimilar to $R \stackrel{\text{def}}{=} m_{\text{r}}(x).M_{\text{r}}\langle x\rangle + m_{\text{w}}(x).M_{\text{w}}\langle x\rangle$, as expected. One of the problems for using the π-calculus in this model is, it is difficult to present complicated exclusion relations where not all methods are either mutually exclusive or fully parallel.

To enhance the power in modelling the compositional concurrent objects, [Zhang02C] proposed the κ-calculus, an extended process algebra, which welds the mobility power of the π-calculus with the synchronisation expressiveness of the algebra of exclusion ([Noble00]). In the κ-calculus, the major change to the conventional π-calculus is that, the "+" operation, a mutually exclusive non-deterministic choice which eliminates all the un-chosen branches whenever a branch is chosen, is replaced by the conditional exclusive choice operator "⊗" where the un-chosen branches are only selectively and temperately blocked. To input-guarded processes, the '+' and ' | ' compositions can be viewed as two extrem cases of "⊗": permenately block all branches, and block none.

In addition to giving simpler and clearer descriptions of method exclusion while modelling dynamic behavour of concurrent objects, the κ-calculus provides a naturally separation of different aspects in concurrency behaviours such as locking status, method exclusion, scheduling synchronisation and the functionality. In this paper we adopt the κ-calculus as modelling and proving tool in study the compositional concurrent objects.

Another difficulty in this compositional concurrent object model was, while an equivalence between the comosed behaviour and the target behaviour could be determined in practice, it might not be theoritically recognised by existing bisimulation relations. This problem now has been solved by the responsive bisimulation, which was proposed in [Zhang02B] and extended to the κ-calculus in [Zhang02D]. The responsive bisimulation treats the buffering of a message globally as of the same effect as buffering locally. With the responsive bisimulation, composing an empty behaviour to an existing behaviour becomes possible, and more interestingly, similarity between responsive behaviours of general control processes can be captured.

This forms the base of the Theory of Composition for concurrent objects, a theory about when and how the behaviour of an concurrent object can be decomposed in to element behaviours and then composed back, and whether and how the correctness of a concurrent object can be reasoned about through the combination of three separated steps: the reasoning about the correctness of the functionality of the object, the reasoning about the correctness of concurrency controls, and the reasoning about the correctness of the composition of these element behaviours.

In this paper we apply the responsive bisimulation and the κ-calculus in the study of the Theory of Composition, and in establishing our compositional concurrent object model. This will begin with the defining, in term of processes, what is an object, a component of an object and a behaviour control in an object, and what is a composition of behaviours or a decomposition of a behaviour propose. Some generic properties of the composition are explored, including the identity law and associative law, as well as a characterising the existence of commutativity in some composition cases. Then a more detailed investigation in modelling component behaviours of concurrent objects in the κ-calculus is presented, which will lead to establish the semantic of a compositional concurrent object-oriented programming languege, and a methology for developing compositional concurrent object systems.

Structure of the report: The rest of this report is structured as follows: section 2 briefly introduces the κ-calculus and related notions; section 3 iutroduces the concept of responsive bisimulation in the κ-calculus; section 4 presents a compositional concurrent model in the κ-calculus; section 5 studies on the theory of composition for compositional object; section 6 concludes the paper.

2 The κ-calculus

The κ-calculus ([Zhang02C]) is a process calculus especially suitable for modelling the composition behaviours of concurrent objects. Like the asynchronous π-calculus ([Amadio96] and [Hüttel96]), it uses asynchronous communication, i.e. an output action does not block other actions. Like the polar π-calculus (π_{p}-calculus) of [Zhang02B], it adopts the concept of polarised names ([Odersky95a]), and the restriction that only output polar of a

name can be transmitted by communication ([Ravara97]). In addition, close to [Liu97], [Philippou96], [Zhang98A] and [Zhang98B], the κ-calculus has a higher-order extension which is only involved with higher-order process abstractions but excludes higher-order communication ([Sangiorgi92a], [Sangiorgi92b]), and therefore can employ the relatively simpler bisimilarity theory of the π–calculus while providing more power on behaviour separation.

The major significance in the κ-calculus is the inclusion of lock as primitive. In the conventional CCS or π-calculi, input-guarded processes can only be composed to play either a "one be chosen then all others have to die" game in the mutually exclusive choice (the sum operation "+"), or "no one minds others' business" game in the parallel composition "|". In the guarded exclusive choice of the κ-calculus, however, the exclusion between branches are explicitly defined, and the invocation of an input action can cause a lock on pre-specified branches, which may become available again when the lock is released. The "+" and "|" operations then are unified into the guarded exclusive choice as two extreme cases. This enables the κ-calculus to obtain the expressibility of the algebra of exclusion ([Noble00]) for methods exclusion of concurrent objects, allows the separation of some major concurrency behaviours of objects to be presented in a much more natural and clearer way.

Before start to introduce the , we introduce an abbreviation notation which is frequently used throughout this paper.

Notation 2−1: The notation $t_{i\in I}$ represents a list of terms t, which is indexed by the index set I. That is, $t_{i\in I}$ represents the list $t_1,t_2,\ldots,t_n$ when I={1,2,…,n}. If the details of I is not of interest, then the abbreviation notation $\tilde{t}$ can also be used interchangeable with $t_{i\in I}$.

Definition 2−2: A list $t_{i\in I}$ (or $\tilde{t}$) is called a *canonical list* if every element in it is distinguished.

2.1 The syntax of the κ-calculus

In the κ-calculus we distinguish two disjoint sets of label names in order to provent cross using by mistake: the communication channel names, and the key names for locking.

Let $\mathcal{M}$ be the set of all communication channel names, ranged over by expressions m,u,v and variables x,y. Let ${}^{+}\mathcal{M}\stackrel{\text{def}}{=}\{\dot{m}: m\in\mathcal{M}\}$ and ${}^{-}\mathcal{M}\stackrel{\text{def}}{=}\{\bar{m}: m\in\mathcal{M}\}$ be the sets of input polar and output polar of all channel names respectively. Let $\mathcal{K}$ be set of all release keys of locking, ranged over by κ. Let ${}^{+}\mathcal{K}\stackrel{\text{def}}{=}\{\dot{\kappa}: \kappa\in\mathcal{K}\}$ and ${}^{-}\mathcal{K}\stackrel{\text{def}}{=}\{\bar{\kappa}: \kappa\in\mathcal{K}\}$ be the sets of input polar and output polar of all keys respectively. Then the set of all label names is $\mathcal{N}\stackrel{\text{def}}{=}\mathcal{M}\cup\mathcal{K}$, ranged over by n. Consequently, we have various sets of polars, such as ${}^{\pm}\mathcal{M}\stackrel{\text{def}}{=}{}^{+}\mathcal{M}\cup{}^{-}\mathcal{M}$, ${}^{\pm}\mathcal{K}\stackrel{\text{def}}{=}{}^{+}\mathcal{K}\cup{}^{-}\mathcal{K}$, ${}^{+}\mathcal{N}\stackrel{\text{def}}{=}{}^{+}\mathcal{M}\cup{}^{+}\mathcal{K}$, ${}^{-}\mathcal{N}\stackrel{\text{def}}{=}{}^{-}\mathcal{M}\cup{}^{-}\mathcal{K}$, and ${}^{\pm}\mathcal{N}\stackrel{\text{def}}{=}{}^{+}\mathcal{N}\cup{}^{-}\mathcal{N}$. Let $a,b\in{}^{\pm}\mathcal{N}$ be polar constants, and $w\in{}^{\pm}\mathcal{N}$ be polar variables. The generic process terms P in the κ-calculus are generated by the following grammars:

$$P::=\mathbf{0_P}\,|\,m\langle\tilde{u}\rangle\,|\,\hat{\kappa}\,|\,(\nu\,\tilde{n})P\,|\,[\dot{m}\langle\bar{u}\rangle]P\,|\,P_1|P_2\,|\,\Lambda\circ(G)\,|\,A\langle\tilde{a}\rangle\,|\,\mathbb{P}\langle\!\langle\tilde{P}\rangle\!\rangle\,|\,\eta,\qquad A::=(\tilde{w})P,\qquad \mathbb{P}::=\langle\!\langle\tilde{\eta}\rangle\!\rangle P,$$

Most process terms (P–terms) are similar to those in normal π-calculi: $\mathbf{0_P}$ is the inactive (terminated) process; $m\langle\tilde{u}\rangle$ is the output action which sends output polars $\tilde{u}$ into the channel m; $(\nu\,\tilde{n})P$ binds the set of labels $\tilde{n}$, and therefore both polars of each of them, within the scope of P; $P_1\mid P_2$ indicates two processes run in parallel; $A\langle\tilde{a}\rangle$ is an instance of parameterised process agent, giving the process agent abstraction $A\stackrel{\text{def}}{=}(\tilde{w})P$ is obeying $((\tilde{w})P)\langle\tilde{a}\rangle\equiv P\{\tilde{a}/\tilde{w}\}$; η is a process variable. For the rest three terms, $\hat{\kappa}$ is the action which emits the *unlock signal* within the scope where the name κ is bound; higher order P-term agent $\mathbb{P}\stackrel{\text{def}}{=}\langle\!\langle\tilde{\eta}\rangle\!\rangle P$ accepts processes as parameters in the double angled brackets, and obeying $(\langle\!\langle\tilde{\eta}\rangle\!\rangle P)\langle\!\langle\tilde{R}\rangle\!\rangle\equiv P\{\tilde{R}/\tilde{\eta}\}$; $[\dot{m}\langle\bar{u}\rangle]P$ is called *localisation*, indicates an incoming message $\tilde{u}$ is buffered in the input polar $\dot{m}$, and becomes visible and consumable only by P; and $\Lambda\circ(G)$ is the *guarded exclusive choice* (*GEC* choice), where G defines the exclusion behaviour which can place some locks on the process itself, and Λ records the lock status. The syntax of the choice terms (G–terms) is

$$G::=B\,|\,(\nu\,\tilde{n})G\,|\,G_1\otimes G_2\,|\,D\langle\tilde{a}\rangle\,|\,\mathbb{G}\langle\!\langle\tilde{P}\rangle\!\rangle,\qquad \mathbb{G}::=\langle\!\langle\tilde{\eta}\rangle\!\rangle H,\qquad H::=\mathit{Б}\,|\,H_1\otimes H_2\,|\,(\nu\tilde{n})H\,|\,\mathbb{G}\langle\!\langle\tilde{\eta}\rangle\!\rangle,$$
$$B::=\mathbf{0_G}\,|\,!\beta.P\,|\,!(\nu\,\kappa)\beta.P,\qquad \beta::=\dot{m}(\tilde{x})L,\qquad \mathit{Б}::=\mathbf{0_G}\,|\,!\beta.\eta\,|\,!(\nu\,\kappa)\beta.\eta,$$
$$L::=\dot{\kappa}@J\,|\,(\nu)@J,\qquad J::=\{\tilde{\dot{m}}\}\,|\,\varnothing\,|\,\boldsymbol{M},\qquad D::=(\tilde{w})G$$

Here B is a choice branch; $G_1 \otimes G_2$ is the choice composition; $(\tilde{w})G$ and $D\langle \tilde{a} \rangle$ are abstraction and instance of choice agent respectively, obeying $((\tilde{w})G)\langle \tilde{a} \rangle \equiv G\{\tilde{a}/\tilde{w}\}$; higher order G-term agent $\mathcal{G} \stackrel{\text{def}}{=} \langle\!\langle \tilde{\eta} \rangle\!\rangle H$ accepts processes as parameters in the double angled brackets, and obeying $(\langle\!\langle \tilde{\eta} \rangle\!\rangle H)\ \langle\!\langle \tilde{P} \rangle\!\rangle \equiv H\{\tilde{P}/\tilde{\eta}\}$; H is higher order G-term with some free process variable η; $\mathit{Б}$ is a branches of a higher-order GEC term; $\mathbf{0_G}$ is the unreachable choice, in the future we can omit the subscript of both $\mathbf{0_G}$ and $\mathbf{0_P}$ without any ambiguity. Unlike that in π, every branch B here always behaves as a (lazy) replication, among them, "!(ν κ)" creates a fresh key κ private to each replicated copy; in $\beta.P$ the action prefix operator "." indicates the execution of action β before the execution of the continuation process P; the action $\overset{\downarrow}{m}(\tilde{x})L$, where we stipulate that $\{\tilde{x}\} \cap n(L) = \varnothing$, produces two simultaneous events: receiving information $\tilde{x}$ from the input port of channel m, and triggering the lock L; the lock $L = \check{\kappa}@J$ read as "lock all input channels in J with key κ", where the exclusion set J specifies the channels to be locked within the *GEC* choice and κ is the key for unlocking the lock; abbreviation $L = (\nu)@J$ indicates a lock with an anonymous key, that is, $!\overset{\downarrow}{m}(\tilde{x})(\nu)@J.P \equiv !(\nu\ \kappa)\overset{\downarrow}{m}(\tilde{x})\check{\kappa}@J.P$ for $\kappa \notin fn(P)$, in other words, it is an unreleasable lock; $\boldsymbol{M}$ is the entire $^{+}\mathcal{M}$, the set of input polar of all channel names, and therefore enforces the locking of every channel within the *GEC* choice.

The set of all actions a process may take can be specified by $\alpha ::= \overset{\downarrow}{m}(\tilde{u}) \mid (\nu\ \tilde{v})\overline{m}\langle \tilde{u} \rangle \mid \hat{\kappa} \mid \check{\kappa} \mid \tau$, where $\tilde{v} \subseteq \tilde{u}$ and $m \notin \tilde{v}$.

The other part of the *GEC* choice, Λ, acts as a state machine maintaining and monitoring the current status of locks, and is described in an independent language. Different Λ grammars will give different locking schemes and locking status evolution paths, but will not interfere with semantic or syntax of the G language, and vice versa. In one of the simplest such locking scheme, where duplicate locks upon the same channel with the same key will have the same effect as such a single lock ([Zhang02C]), is defined by the grammar $\Lambda ::= \rfloor \mid \tilde{L}\rfloor \mid \Lambda\Lambda$ and the structural equivalencies rules are shown in Figure 2-1.

lstr-SMM (Summation)	:	$\rfloor\Lambda \equiv \Lambda;\quad \Lambda_1\Lambda_2 \equiv \Lambda_2\Lambda_1;\quad \Lambda_1(\Lambda\Lambda_2) \equiv (\Lambda_1\Lambda)\Lambda_2.$
lstr-EMP (Empty lock)	:	$\check{\kappa}@\varnothing\rfloor \equiv \rfloor;$
lstr-LKC (Combination)	:	$\check{\kappa}@J_1, \check{\kappa}@J_2\rfloor \equiv \check{\kappa}@(J_1 \cup J_2)\rfloor;$
lstr-GRP (Grouping)	:	$\tilde{L}_1\rfloor\ \tilde{L}_2\rfloor \equiv \tilde{L}_1, \tilde{L}_2\rfloor;$

Figure 2-1 Structual equivalence of locking status terms

Notation 2−3: Some auxiliary operations/functions (the formal definitions can be found in [Zhang02C]) are needed for integrating a Λ language into the κ-calculus:

guard(G) : gives the set of all branches' input prefix channel names in G, and defined by
$guard(!\overset{\downarrow}{m}(\tilde{x})L.P) \stackrel{\text{def}}{=} m;\quad guard((\nu\ \tilde{n})G) \stackrel{\text{def}}{=} guard(G);\quad guard(G_1 \otimes G_2) \stackrel{\text{def}}{=} \{guard(G_1)\} \cup \{guard(G_2)\}.;$

lock(J,κ,Λ) : gives the truth value for whether Λ indicates all the input polars appeared in J are locked by κ;

lset(Λ) : gives the set of all channel names for which their input polars are indicated by Λ as locked;

keys(Λ) : gives the set of all key κ for which there exists some $J \neq \varnothing$ such that *lock*(J,κ,Λ)=true;.

add(L,Λ) : gives the new locking status after adding L to the original locking status Λ.

Λ/L : gives the new locking status after removing L from the original locking status Λ.

We usually use $\Lambda \stackrel{\text{def}}{=} \rfloor$ to represent an empty lock, and for any Λ language, we always require that:
lock($J,\kappa,\rfloor$)≡*False*, *lset*($\rfloor$) ≡∅, *keys*($\rfloor$) ≡∅, *add*($L,\rfloor$) ≡ $L\rfloor$ and $\rfloor/L \equiv \rfloor$.

Notation 2−4: If $m \notin lset(\Lambda)$, we say that Λ *allows the commitment* on $\overset{\downarrow}{m}$, denoted as $\Lambda\downarrow\overset{\downarrow}{m}$;
if $m \in lset(\Lambda)$, we say that Λ *blocks* the channel $\overset{\downarrow}{m}$, denoted as $\Lambda\not\downarrow\overset{\downarrow}{m}$.
If for some $J' \subseteq J$, $J' \neq \varnothing$ and *lock*(J', κ, Λ), we say that Λ can commit $\check{\kappa}@J$, denoted as $\Lambda\downarrow\check{\kappa}@J$;
otherwise we say that he Λ cannot commit on $\check{\kappa}$ over J, denoted as $\Lambda\not\downarrow\check{\kappa}@J$.
If for some $J \supseteq lset(\Lambda)$, $\Lambda\downarrow\check{\kappa}@J$, we say that Λ can commit $\check{\kappa}$, denoted as $\Lambda\downarrow\check{\kappa}$;
otherwise we say that he Λ cannot commit on $\check{\kappa}$, denoted as $\Lambda\not\downarrow\check{\kappa}$.
In the form of labelled transition, we denote $\Lambda \xrightarrow{\overset{\downarrow}{m}\uparrow L} \Lambda'$ for $\overset{\downarrow}{m} \notin lset(\Lambda)$ and $\Lambda' = add(L,\Lambda)$; and
$\Lambda \xrightarrow{\check{\kappa}@J} \Lambda'$ for $\Lambda\downarrow\check{\kappa}@J$ and $\Lambda' = \Lambda/\kappa@J$.

Notation 2−5: Similar to the polar π-calculus, besides the functions *fn*, *bn* and *n* for identifying the sets of free, bound and all names respectively of a *P*–term, *G*–term or action, we also use more specified functions, such as *fin*, *bin*, *in*, *fon*, *bon* and *on* to identify free, bound and all input or out polars. Further more, as in the κ-calculus we distinguish communication channel names and keys, we also use finer grained functions, *fnc*, *bnc*, *nc*, *finc*, *binc*, *inc*, *fonc*, *bonc* and *onc* for communication channels only, and *fnk*, *bnk*, *nk*, *fink*, *bink*, *ink*, *fonk*, *bonk* and *onk* for keys only.

Notation 2−6: The following process abbreviations are for convenience and can simplify expressions:

$\overset{\cdot}{m}(\tilde{u}).P \overset{\text{def}}{\equiv} \rfloor\circ(!\overset{\cdot}{m}(\tilde{u})(\nu)_{@}[\overset{\cdot}{m}].P\rceil$, $\prod \overset{\cdot}{m}_i(\tilde{u}).P_i \equiv \rfloor\circ(\bigotimes !\overset{\cdot}{m}_i(\tilde{u})(\nu)_{@}[\overset{\cdot}{m}_i].P_i\rceil$, $\sum \overset{\cdot}{m}_i(\tilde{u}).P_i \overset{\text{def}}{\equiv} \rfloor\circ(\bigotimes !\overset{\cdot}{m}_i(\tilde{u})(\nu)_{@}\boldsymbol{M}.P_i\rceil$
$!\overset{\cdot}{m}(\tilde{u}).P \overset{\text{def}}{\equiv} \rfloor\circ(!\overset{\cdot}{m}(\tilde{u})(\nu)_{@}\varnothing.P\rceil$, $\prod !\overset{\cdot}{m}_i(\tilde{u}).P_i \equiv \rfloor\circ(\bigotimes !\overset{\cdot}{m}_i(\tilde{u})(\nu)_{@}\varnothing.P_i\rceil$.

These abbreviations give an illustration of that for input-prefixed processes, the standard parallel composition '$|$', the mutual exlcusive choice '+' and replication in conventional π-calculus all become merely special cases of the '⊗' operator in the κ-calculus. Further more, as these abbreviations suggested, encoding a polar π-calculus term into the κ-calculus is very simple, and has been done by [Zhang02C]. However, so far we have not found any straightforward technique for the opposite direction. In fact, the polar π-calculus can be considered as a sub-calculus of the κ-calculus.

2.2 The semantics of the κ-calculus

The structural equivalences and labelled transitions in the κ-calculus are shown in Figure 2-2 and Figure 2-3. The central idea of the operational semantics in this calculus is presented by rules tr-IN, tr-CHOI, tr-RELS, tr-SYNC1 and tr-SYNC2. Compare with the π-calculus, we can see that:

1. an input action $\overset{\cdot}{m}(\tilde{u})$ invokes a new copy of continuation process P from a *GEC* choice and triggers a lock L which may change Λ, the locking state of *GEC* choice, an unlock signal $\check{\kappa}$ may also change the locking state Λ, but does not change the *GEC* choice context;
2. expressions for different aspects, such as current state, exclusion relation and behaviour of the continuation, can therefore be separated naturally and intuitively.

Justification for our calculus is given in Section 5 where we further discuss modelling of composite objects.

As a normal treatment in this literature, throughout this paper the rule str-REN is often applied automatically and implicitly over fresh names to avoid name clash. For example, a name $n_2 \notin fn(P)$ may be picked up automatically so that the process $(\nu\, n_1)(A\langle \overset{\cdot}{n}_1, n_1\rangle \mid (\nu\, n_2)P_1\{^{n_2}/_{n_1}\})$ can be used to replace $(\nu\, n_1)(A\langle \overset{\cdot}{n}_1, n_1\rangle \mid (\nu\, n_1)P_1)$ without mention.

Remark 2−7: Similar to the polar π-calculus, in the the κ-calculus the τ action is truly internal, that is, neither visible nor interruptible by external observers. Therefore, the name restrictions in rule tr-SYNC1 and tr-SYNC2 are required. Without it, the synchronisation will not be considered as an internal action, but a two steps action, such as $P \mid Q \xrightarrow{(\nu \tilde{v})\overset{\cdot}{m}\langle \tilde{u}\rangle} \cdot \xrightarrow{\overset{\cdot}{m}(\tilde{u})} (\nu\, \tilde{v})(P' \mid Q')$ or $P \mid Q \xrightarrow{\hat{\kappa}} \cdot \xrightarrow{\check{\kappa}} P' \mid Q'$, where both steps are visible for external observers. This strong requirement on τ actions is necessary for guaranteeing the standard rule $fn(\tau) = bn(\tau) = \varnothing$ ([Amadio96]) valid, and is necessary for preserving τ actions in output polars substitution.

As usual, let ( )* represent that the contents in ( ) repeating zero or many times, then the weak transitions are defined as:

Definition 2−8: $P \overset{\tau}{\Longrightarrow} P'$ iff $P(\xrightarrow{\tau})^* P'$; $P \overset{\alpha}{\Longrightarrow} P'$ iff $P \overset{\tau}{\Longrightarrow} \cdot \xrightarrow{\alpha} \cdot \overset{\tau}{\Longrightarrow} P'$, where $\alpha \neq \tau$.

Reduction relation, a familiar concept in this literature, is defined in a non-standard way in the κ-calculus:

Definition 2−9: $P \longrightarrow P'$ iff $(\nu\, m)P \xrightarrow{\tau} (\nu\, m)P'$ for some m; $P \Longrightarrow P'$ iff $(\nu\, m)P \overset{\tau}{\Longrightarrow} (\nu\, m)P'$ for some m.

Summation

str-SUM1: $P_1 \mid \mathbf{0_P} \equiv P_1$; $\quad G_1 \otimes \mathbf{0_G} \equiv G_1$

str-SUM2: $P_1 \mid P_2 \equiv P_2 \mid P_1$; $\quad G_1 \otimes G_2 \equiv G_2 \otimes G_1$

str-SUM3: $P_1 \mid (P_2 \mid P_3) \equiv (P_1 \mid P_2) \mid P_3$; $\quad G_1 \otimes (G_2 \otimes G_3) \equiv (G_1 \otimes G_2) \otimes G_3$

str-SUM4: $[\natural m\langle \tilde{u}\rangle][\natural n\langle \tilde{v}\rangle]P \equiv [\natural n\langle \tilde{v}\rangle][\natural m\langle \tilde{u}\rangle]P$;

Null

str-NUL: $[\natural m\langle \tilde{u}\rangle]\,\mathbf{0} \equiv \mathbf{0}$; $\quad \Lambda\circ[\mathbf{0_G}] \equiv \mathbf{0_P}$

str-DISJ: $addl(\check{\kappa}@J, \Lambda)\circ[G] \equiv \Lambda\circ[G]$ if $guard(G) \cap J = \varnothing$

Instance

str-INS: $((\tilde{x})P)\langle \tilde{a}\rangle \equiv P\{\tilde{a}/\tilde{x}\}$; $\quad ((\tilde{x})G)\langle \tilde{a}\rangle \equiv G\{\tilde{a}/\tilde{x}\}$

str_HAR: $(\!|\tilde{\eta}|\!)P)\,\langle\!|\tilde{R}|\!\rangle \equiv P\{\tilde{R}/\tilde{\eta}\}$; $\quad (\!|\tilde{\eta}|\!)H)\,\langle\!|\tilde{P}|\!\rangle \equiv H\{\tilde{P}/\tilde{\eta}\}$

Scope

str-SCP1: $(\nu\, n)P \equiv P$, if $n \notin fn(P)$; $\quad (\nu\, n)\,G \equiv G$, if $n \notin fn(G)$;

$!(\nu\,\kappa)\beta.P \equiv !\beta.P$, if $\kappa \notin fn(\beta.P)$

str-SCP2: $(\nu\, n_1)(\nu\, n_2)P \equiv (\nu\, n_2)(\nu\, n_1)P$; $\quad (\nu\, n_1)(\nu\, n_2)P \equiv (\nu\, n_1, n_2)P$

str-SCP3: $(\nu\, m)\bar{m}\langle \tilde{y}\rangle \equiv \mathbf{0_P}$; $\quad (\nu\, m)!\natural m(\tilde{x})L.P \equiv \mathbf{0_G}$;

$(\nu\,\kappa)\hat{\kappa} \equiv \mathbf{0_P}$; $\quad (\nu\,\kappa)\Lambda\circ[G] \equiv (\nu\,\kappa)\Lambda\circ[\mathbf{0_G}]$, if $lock(guard(G), \kappa, \Lambda)$ is true;

$(\nu\,\kappa)\Lambda\circ[G \otimes G'] \equiv \Lambda\circ[G']$, if $lock(guard(G), \kappa, \Lambda)$ and $\kappa \notin fn(G')$

str-SCP4: $\Lambda\circ[(\nu\,\kappa)G] \equiv (\nu\,\kappa)\Lambda\circ[G]$, if $\kappa \notin keys(\Lambda)$;

$(\nu\, n)P_1 \mid P_2 \equiv (\nu\, n)(P_1 \mid P_2)$, if $n \notin fn(P_2)$; $\quad (\nu\, n)G_1 \otimes G_2 \equiv (\nu\, n)(G_1 \otimes G_2)$, if $n \notin fn(G_2)$

str-REN: $(\nu\, n_1)P \equiv (\nu\, n_2)(P\{n_2/n_1\})$, if $n_2 \notin fn(P)$

Localisation

str-LOC: $(\nu\, m)\,[\natural m\langle \tilde{u}\rangle]P \equiv (\nu\, m)\,(\natural m\langle \tilde{u}\rangle \mid P)$

str_IND: $([\natural m\langle \tilde{u}\rangle]P) \mid Q \equiv [\natural m\langle \tilde{u}\rangle](P \mid Q)$, if $m \notin fn(Q)$

Figure 2-2 Structural congruence rules for the κ-calculus

tr-OUT: $\dfrac{\cdot}{\bar{m}\langle \tilde{u}\rangle \xrightarrow{\bar{m}\langle \tilde{u}\rangle} \mathbf{0_P}}$, $\quad \dfrac{P \xrightarrow{\bar{m}\langle \tilde{u}\rangle} P', \quad m \notin \tilde{v}}{(\nu\,\tilde{v})P \xrightarrow{(\nu\,\tilde{v})\bar{m}\langle \tilde{u}\rangle} P'}$

tr-SIG: $\dfrac{\cdot}{\hat{\kappa} \xrightarrow{\hat{\kappa}} \mathbf{0_P}}$

tr-IN: $\dfrac{\Lambda \xrightarrow{\natural m\uparrow L} \Lambda'}{\Lambda\circ[!(\nu\,\kappa)\natural m(\tilde{x})L.P] \xrightarrow{\natural m(\tilde{u})} (\nu\,\kappa)(P\{\tilde{u}/\tilde{x}\} \mid \Lambda'\circ[!(\nu\,\kappa)\natural m(\tilde{x})L.P])}$

tr-CHOI: $\dfrac{\Lambda\circ[G_1] \xrightarrow{\alpha} (\nu\,\kappa)(P \mid \Lambda'\circ[G_1], \quad \kappa \notin fn(G_2)}{\Lambda\circ[G_1 \otimes G_2] \xrightarrow{\alpha} (\nu\,\kappa)(P \mid \Lambda'\circ[G_1 \otimes G_2])}$

tr-RELS: $\dfrac{\Lambda \xrightarrow{\check{\kappa}@J} \Lambda', \quad \text{where } J \supseteq guard(G)}{\Lambda\circ[G] \xrightarrow{\check{\kappa}} \Lambda'\circ[G]}$

tr-PARL: $\dfrac{P \xrightarrow{\alpha} P'}{P \mid Q \xrightarrow{\alpha} P' \mid Q}$

tr-SYNC1: $\dfrac{P \xrightarrow{(\nu\,\tilde{v})\bar{m}\langle \tilde{u}\rangle} P', \quad Q \xrightarrow{\natural m(\tilde{u})} Q', \quad \tilde{v} \cap fn(Q) = \varnothing}{(\nu\, m)(P \mid Q) \xrightarrow{\tau} (\nu\, m)(\nu\,\tilde{v})(P' \mid Q')}$

tr-SYNC2: $\dfrac{P \xrightarrow{\hat{\kappa}} P', \quad Q \xrightarrow{\check{\kappa}} Q'}{(\nu\,\kappa)(P \mid Q) \xrightarrow{\tau} (\nu\,\kappa)(P' \mid Q')}$

tr-SYNC3: $\dfrac{P \xrightarrow{\natural m(\tilde{u})} P'}{[\natural m\langle \tilde{u}\rangle]P \xrightarrow{\tau} P'}$,

tr-INV: $\dfrac{P \xrightarrow{\alpha} P' \quad \alpha \neq \natural m(\tilde{u})}{[\natural m\langle \tilde{u}\rangle]P \xrightarrow{\alpha} [\natural m\langle \tilde{u}\rangle]P'}$;

tr-RES: $\dfrac{P \xrightarrow{\alpha} P', \quad \tilde{n} \cap fn(\alpha) = \varnothing}{(\nu\ \tilde{n})P \xrightarrow{\alpha} (\nu\ \tilde{n})P'}$

tr-STRUC: $\dfrac{P_1' \equiv P_1, \quad P_1 \xrightarrow{\alpha} P_2, \quad P_2 \equiv P_2'}{P_1' \xrightarrow{\alpha} P_2'}$

Figure 2-3 Labelled transition rules for process terms in the κ-calculus

Clearly, $P \xrightarrow{\tau} P'$ implies $P \longrightarrow P'$, and $P \xRightarrow{\tau} P'$ implies $P \Longrightarrow P'$, and therefore a variant of the rule tr-SYNC1 can be written as: if $P \xrightarrow{(\nu \tilde{v})\overline{m}\langle \tilde{u} \rangle} P'$ and $Q \xrightarrow{m(\tilde{u})} Q'$ where $\tilde{v} \cap fn(Q) = \varnothing$, then $P \mid Q \longrightarrow (\nu \tilde{v})(P' \mid Q')$. Beside the reason we have just discussed, the distinguish between internal action and reduction is also necessary for the new bisimulation relation, and we will find out later.

Definition 2−10: The strong commitments are defined as:

Process P can *commit* the action α, denoted as $P \downarrow \alpha$, if there exists some P' such that $P \xrightarrow{\alpha} P'$.
Process P can *commit* on input polar m, denoted as $P \downarrow m$, if there exists some input action $\alpha = m(\tilde{u})$ s.t. $P \downarrow \alpha$;
Process P can *commit* on output polar $\overline{m}$, denoted as $P \downarrow \overline{m}$, if there is some output action $\alpha = (\nu \tilde{v})\overline{m}\langle \tilde{u} \rangle$ s.t. $P \downarrow \alpha$;
Process P can *commit* the action sequence ℓ, denoted as $P \downarrow \ell$, if $P \xrightarrow{a_1} \cdot \xrightarrow{a_2} \cdot \ldots \cdot \xrightarrow{a_n} P'$, or as an abbreviation, $P \xrightarrow{\ell} P'$;

The weak commitments $\Downarrow$, is obtained by replacing $\rightarrow$ with $\Rightarrow$ and $\downarrow$ with $\Downarrow$ though out.

Definition 2−11: Process P' is a *derivative* of process P, if there exists some finite sequence ℓ such that $P \xRightarrow{\ell} P'$.

2.3 The higher-order *GCE* terms

One of the most significants of using the higher-order choice term is that, it allows the expressions of exclusion to be separated from other aspects, and to be reasoned about separately. For example, in the process $\Lambda \circ [\mathcal{G}\langle\langle P_{i \in \{1,2,3\}} \rangle\rangle]$ we may have $\mathcal{G} \triangleq \langle\langle \eta_1, \eta_2, \eta_3 \rangle\rangle (!(\nu \kappa)\overline{m}_1(\tilde{x})\breve{\kappa}@\{\overline{m}_2\}.\eta_1\langle \tilde{x}, \hat{k} \rangle \otimes !(\nu \kappa)\overline{m}_2(\tilde{x})\breve{\kappa}@\{\overline{m}_1, \overline{m}_2\}.\eta_2\langle \tilde{x}, \hat{k} \rangle \otimes !(\nu \kappa)\overline{m}_3(\tilde{x})\breve{\kappa}@\varnothing.\eta_3\langle \tilde{x}, \hat{k} \rangle)$, which describes the same exclusion relations as the expression $m_1 \times \overline{m}_2 \mid \overline{m}_3$, in Noble's Algebra of Exclusion ([Noble00]), without rising any details about Λ or $P_{i \in \{1,2,3\}}$. [Zhang02C] has developed theory to enable this ability.

Notation 2−12: For convenience, we sometimes using symbol $\mathcal{B}$ to denote a higher-order GEC $\mathcal{G}$ with single arity and single branch, that is, it has the form of $\mathcal{B} \triangleq \langle\langle \eta \rangle\rangle B$.

Definition 2−13: Higher-order GEC terms $\mathcal{G}_1$ and $\mathcal{G}_2$ are structural equivalent, written $\mathcal{G}_1 \equiv \mathcal{G}_2$ if $arity(\mathcal{G}_1) = arity(\mathcal{G}_2)$, and $\mathcal{G}_1\langle\langle \tilde{P} \rangle\rangle \equiv \mathcal{G}_2\langle\langle \tilde{P} \rangle\rangle$ for all $\tilde{P}$ satisfying $arity(\tilde{P}) = arity(\mathcal{G}_1)$.

For example, $\mathcal{G}_1 \equiv \mathcal{G}_2$ if $\mathcal{G}_1 \triangleq \langle\langle \eta_1, \eta_2 \rangle\rangle (!\beta_1.\eta_1 \otimes !\beta_2.\eta_2)$ and $\mathcal{G}_2 \triangleq \langle\langle \eta_1, \eta_2 \rangle\rangle (!\beta_2.\eta_2 \otimes !\beta_1.\eta_1)$.

Corollary 2−14: $\mathcal{G} \equiv \langle\langle \tilde{\eta} \rangle\rangle \mathcal{G}\langle\langle \tilde{\eta} \rangle\rangle$.

With a slight notation abuse, whenever these is no ambiguity, we may simply write $\mathcal{G}\langle\langle \tilde{\eta} \rangle\rangle$ as $\mathcal{G}$ (and therefore write $\mathcal{B}\langle\langle \eta \rangle\rangle$ as $\mathcal{B}$), and therefore the symbol H for higher order GCE is no-longer necessary.

Definition 2−15: Let $arity(\tilde{\eta}_1) = arity(\mathcal{G}_1)$, $arity(\tilde{\eta}_2) = arity(\mathcal{G}_2)$ and $\tilde{\eta}_1 \cap \tilde{\eta}_2 = \varnothing$, then composition of higher order GCE choice is defined as $\mathcal{G}_1 \otimes \mathcal{G}_2 \triangleq \langle\langle \tilde{\eta}_1, \tilde{\eta}_2 \rangle\rangle (\mathcal{G}_1\langle\langle \tilde{\eta}_1 \rangle\rangle \otimes \mathcal{G}_2\langle\langle \tilde{\eta}_2 \rangle\rangle)$.

Even though we have $\mathcal{G}_1\langle\langle \tilde{P}_1 \rangle\rangle \otimes \mathcal{G}_2\langle\langle \tilde{P}_2 \rangle\rangle \equiv \mathcal{G}_2\langle\langle \tilde{P}_2 \rangle\rangle \otimes \mathcal{G}_1\langle\langle \tilde{P}_1 \rangle\rangle$ and $(\mathcal{G}_1 \otimes \mathcal{G}_2)\langle\langle \tilde{P}_1, \tilde{P}_2 \rangle\rangle \equiv (\mathcal{G}_2 \otimes \mathcal{G}_1)\langle\langle \tilde{P}_2, \tilde{P}_1 \rangle\rangle$, syntactically the summation $\mathcal{G}_1 \otimes \mathcal{G}_2 \equiv \mathcal{G}_2 \otimes \mathcal{G}_1$ does not hold, since $(\mathcal{G}_1 \otimes \mathcal{G}_2)\langle\langle \tilde{P}_1, \tilde{P}_2 \rangle\rangle \not\equiv (\mathcal{G}_2 \otimes \mathcal{G}_1)\langle\langle \tilde{P}_1, \tilde{P}_2 \rangle\rangle$. This is a disadvantage of the current form of the κ-calculus. However, as pointed out by [Zhang02C], this problem can be solved by introducing "labelled tuples" to κ-calculus, that is, each tuple uses a list of distinguished names called "labels" to indentify parameters and therefore a parameter will be no longer fixed to a position. For example, with labelled tuples, the three expressions $\mathcal{G}\langle\langle l_1 = P_1, l_2 = P_2, l_2 = P_3 \rangle\rangle$, $\mathcal{G}\langle\langle l_3 = P_3, l_2 = P_2, l_1 = P_1 \rangle\rangle$ and $\mathcal{G}\langle\langle l_1 = P_1, l_3 = P_3, l_2 = P_2 \rangle\rangle$ will be indifferent, and all can be used to instantiate $\mathcal{G} \triangleq \langle\langle l_1 = \eta_1, l_2 = \eta_2, l_2 = \eta_3 \rangle\rangle H$.

To avoid to complicate the syntax and rules of the κ-calculus, we currently do not include the labelled tuples in the syntax of the κ-calculus. However, we may adopt it implicitly whenever the summation $\mathcal{G}_1 \otimes \mathcal{G}_2 \equiv \mathcal{G}_2 \otimes \mathcal{G}_1$ becomes necessary.

Lemma 2−16: $(\mathcal{G}_1 \otimes \mathcal{G}_2)$ «$\tilde{P}_1, \tilde{P}_2$» ≡ $(\mathcal{G}_2 \otimes \mathcal{G}_1)$ «$\tilde{P}_2, \tilde{P}_1$»; $\quad \mathcal{G} \otimes \mathbf{0} \equiv \mathbf{0} \otimes \mathcal{G} \equiv \mathcal{G}$;
$(\mathcal{G}_1 \otimes \mathcal{G}_2) \otimes \mathcal{G}_3 \equiv \mathcal{G}_1 \otimes (\mathcal{G}_2 \otimes \mathcal{G}_3)$; $\quad \mathcal{G}_1 \otimes \mathcal{G}_2 \equiv \mathcal{G}_1' \otimes \mathcal{G}_2'$ if $\mathcal{G}_1 \equiv \mathcal{G}_1'$ and $\mathcal{G}_2 \equiv \mathcal{G}_2'$.

There is still a problem of the ⊗ operator: it does not remove redundancy in composition as what the Algebra of Exclusion dose. For example, in the Algebra of Exclusion, given expressions $e_1 \stackrel{\text{def}}{\equiv} m_1 \times m_2$ and $e_2 \stackrel{\text{def}}{\equiv} m_2 \times m_3$, then the composition $e_1 \mid e_2 \equiv m_2 \times (m_2 \mid m_3)$. But when translated into the κ-calculus, then the arity of $\mathcal{G}_1 \otimes \mathcal{G}_2$, the composition of $\mathcal{G}_1 \stackrel{\text{def}}{\equiv} [\![e_1]\!]_\kappa$ and $\mathcal{G}_2 \stackrel{\text{def}}{\equiv} [\![e_2]\!]_\kappa$, will be 4, while the arity of $[\![e_1 \mid e_2]\!]$ is 3. We use a new operator to solve this problem.

Definition 2−17: An higher order *GEC* choice $\mathcal{G}$ is *canonical*, denoted as $\overline{\mathcal{G}}$, if the guard of every its branch is distinguished from other branchs. In other words, $\mathcal{G}$ satisfies that, either $\mathcal{G} = \mathbf{0}$ or $\mathcal{G} \equiv \mathcal{B} \otimes \overline{\mathcal{G}}_1$ where:

1. $\overline{\mathcal{G}}_1$ is a canonical higher order *GEC* choice; and
2. $guard(\mathcal{B}) \notin guard(\overline{\mathcal{G}}_1)$; and
3. The lock key of $\mathcal{B}$ is defined local to $\mathcal{B}$, that is, it is in the form of $\mathcal{B} \stackrel{\text{def}}{\equiv}$ «η» $!(\nu\,\kappa)\bar{m}(\tilde{x})\check{\kappa}@J\,.\,\eta$.

If $\mathcal{B}$ is the only branch of $\overline{\mathcal{G}}$, we may also use symbol $\overline{\mathcal{B}}$ to denote $\overline{\mathcal{G}}$.

For example, $\mathcal{G} \stackrel{\text{def}}{\equiv}$ «η_1, η_2» $!(\nu\,\kappa)\bar{m}_1(\tilde{x})\,\check{\kappa}@J_1.\,\eta_1 \otimes\ !(\nu\,\kappa)\bar{m}_2(\tilde{x})\,\check{\kappa}@J_2.\,\eta_2$ is canonical.

Lemma 2−18: If $(\mathcal{G}_1 \otimes \mathcal{G}_2)$ is a canonical higher order *GEC* choice, then both $\mathcal{G}_1, \mathcal{G}_2$ are also canonical higher order *GEC* choices, and $guard(\mathcal{G}_1) \cap guard(\mathcal{G}_2) = \varnothing$.

Proof: By the definition, if otherwise then $(\mathcal{G}_1 \otimes \mathcal{G}_2)$ cannot be a canonical higher order *GEC* choice. ■

Definition 2−19: The combination operator ⊞, mapping a pair of canonical higher order *GEC* choices to a single canonical higher order *GEC* choice, is defined as:

1). $\mathbf{0} \boxplus \overline{\mathcal{G}} \stackrel{\text{def}}{=} \overline{\mathcal{G}}$;

2). $\overline{\mathcal{G}} \boxplus \mathbf{0} \stackrel{\text{def}}{=} \overline{\mathcal{G}}$;

3). $\overline{\mathcal{B}}_1 \boxplus \overline{\mathcal{B}}_2 \stackrel{\text{def}}{=} \begin{cases} \overline{\mathcal{B}}_1 \otimes \overline{\mathcal{B}}_2, & \text{if } guard(\overline{\mathcal{B}}_1) \neq guard(\overline{\mathcal{B}}_2), \\ \text{«}\eta\text{»}\ (!(\nu\,\kappa)\bar{m}(\tilde{x})\check{\kappa}@(excl(\overline{\mathcal{B}}_1) \cup excl(\overline{\mathcal{B}}_2)).\eta), & \text{if } guard(\overline{\mathcal{B}}_1) = guard(\overline{\mathcal{B}}_2) = m; \end{cases}$

4). $(\overline{\mathcal{B}} \otimes \overline{\mathcal{G}}_1) \boxplus \overline{\mathcal{G}}_2 \stackrel{\text{def}}{=} \overline{\mathcal{B}} \otimes (\overline{\mathcal{G}}_1 \boxplus \overline{\mathcal{G}}_2)$ if $guard(\overline{\mathcal{B}}) \notin guard(\overline{\mathcal{G}}_2)$;

5). $(\overline{\mathcal{B}}_1 \otimes \overline{\mathcal{G}}_1) \boxplus (\overline{\mathcal{G}}_2' \otimes \overline{\mathcal{B}}_2 \otimes \overline{\mathcal{G}}_2'') \stackrel{\text{def}}{\equiv} (\overline{\mathcal{B}}_1 \boxplus \overline{\mathcal{B}}_2) \otimes (\overline{\mathcal{G}}_1 \boxplus (\overline{\mathcal{G}}_2' \otimes \overline{\mathcal{G}}_2''))$ if $guard(\overline{\mathcal{B}}_1) = guard(\overline{\mathcal{B}}_2)$.

Note, because the operands at both sides of the ⊞ operator should be canonical higher order *GEC* choices, in the clause 5) of the definition, we have $guard(\overline{\mathcal{B}}_1) \notin (guard(\overline{\mathcal{G}}_1) \cup guard(\overline{\mathcal{G}}_2') \cup guard(\overline{\mathcal{G}}_2''))$. Therefore the clause 1) to 5) in the above definition have covered all possible combinations of the operands.

Lemma 2−20: $\overline{\mathcal{G}}_1 \boxplus (\overline{\mathcal{G}}_2 \boxplus \overline{\mathcal{G}}_3) \equiv (\overline{\mathcal{G}}_1 \boxplus \overline{\mathcal{G}}_2) \boxplus \overline{\mathcal{G}}_3$ for all canonical higher order *GEC* choice $\overline{\mathcal{G}}_1$, $\overline{\mathcal{G}}_2$ and $\overline{\mathcal{G}}_3$.

Proof: The proof is found in [Zhang02C]. ■

From now on we restricte that only canonical *GEC* choice should be used in an object model to describe method exclusion relations. In later sections we will see that we model objects in the form of $\Lambda\circ(\overline{\mathcal{G}}\langle$«$\tilde{P}$»$\rangle)$ or a composition of such.

2.4 A simple type system

To displine the modelling of proecess, a simple type system, close to that in [Liu97], is included in the κ-calculus. A term H having type T is denoted as H:T. The *first-order types*, ranged over by ι, given by

$$\iota ::= \lambda \mid \delta \mid \beta, \qquad \lambda ::= \overset{\uparrow}{\lambda} \mid \overset{\downarrow}{\lambda}, \qquad \delta ::= \overset{\uparrow}{\delta} \mid \overset{\downarrow}{\delta},$$

where λ is a set of atomic types called *link sorts*, whose values are communication polars, and can be further devided to two subsets of types, the input link $\overset{\uparrow}{\lambda}$ and output link $\overset{\downarrow}{\lambda}$; δ is a set of atomic types called *signal sorts*, which has only two atomic types, the input signal type $\overset{\uparrow}{\delta}$ and output signal type $\overset{\downarrow}{\delta}$, whose values are input and output key polars respectively; β are some basic type such as integers, boolean, etc. However, with the same technique demonstrated by [Milner96] where basic types values are modelled by names in the π-calculus, we can always model basic types by polars in the κ-calculus. Therefore, in this paper we need not consider any basic types other than polars.

The *higher-order process types*, ranged over by ξ, are given by $\xi ::= \mathsf{pabs}(\iota_1, \iota_2, \ldots, \iota_n) \mid \mathsf{pabs}(\xi_1, \xi_2, \ldots, \xi_n)$;
and the *higher-order choice types*, ranged over by θ, are given by $\theta ::= \mathsf{gabs}(\iota_1, \iota_2, \ldots, \iota_n) \mid \mathsf{gabs}(\xi_1, \xi_2, \ldots, \xi_n)$.

To define the idea of *well-typing*, we introduce the sorting function lnk from link sorts to tuples of output link sorts, so that $lnk(\lambda) = (\tilde{\overset{\downarrow}{\lambda}})$ permits polars of sort λ to communicate tuples of values of type $\tilde{\overset{\downarrow}{\lambda}}$. For simplifying expressions, we use $\overset{\pm}{\iota}$ to stand a first order type in either $\overset{\uparrow}{\lambda}$ or $\overset{\uparrow}{\delta}$, and $\overset{\downarrow}{\iota}$ to stand a type in either $\overset{\downarrow}{\lambda}$ or $\overset{\downarrow}{\delta}$. we also use $m{:}\overset{\pm}{\lambda}$ to stand for a pair of communication polars $\overset{\uparrow}{m}{:}\overset{\uparrow}{\lambda}$ and $\overset{\downarrow}{m}{:}\overset{\downarrow}{\lambda}$, where $lnk(\overset{\uparrow}{\lambda}) = lnk(\overset{\downarrow}{\lambda})$; use $\kappa{:}\overset{\pm}{\delta}$ to stand for a pair of key polars $\overset{\uparrow}{\kappa}{:}\overset{\uparrow}{\delta}$ and $\overset{\downarrow}{\kappa}{:}\overset{\downarrow}{\delta}$; and use $n{:}\overset{\pm}{\iota}$ to stand for either $n{:}\overset{\uparrow}{\lambda}$ or $n{:}\overset{\downarrow}{\delta}$.

A communication action α is *well-typed* when it is in one of the following cases:

1) $\overset{\uparrow}{m}(\tilde{u})$, where $\overset{\uparrow}{m}{:}\overset{\uparrow}{\lambda}$, $lnk(\lambda) = (\tilde{\overset{\downarrow}{\lambda}})$ and $\tilde{u}{:}\tilde{\overset{\downarrow}{\lambda}}$;
2) $\overset{\downarrow}{m}\langle\tilde{u}\rangle$, where $\overset{\downarrow}{m}{:}\overset{\downarrow}{\lambda}$, $lnk(\lambda) = (\tilde{\overset{\downarrow}{\lambda}})$ and $\tilde{u}{:}\tilde{\overset{\downarrow}{\lambda}}$;
3) $\overset{\uparrow}{\kappa}{:}\overset{\uparrow}{\delta}$;
4) $\overset{\downarrow}{\kappa}{:}\overset{\downarrow}{\delta}$.

A lock L is *well-typed* whenever it is in the form:

5) $\overset{\uparrow}{\kappa}@J$, where $\overset{\uparrow}{\kappa}{:}\overset{\uparrow}{\delta}$ and either $J = \varnothing$, $J = \boldsymbol{M}$ or $J = [\tilde{\overset{\uparrow}{m}}]$ and $\tilde{\overset{\uparrow}{m}}{:}\tilde{\overset{\downarrow}{\lambda}}$.

A locking status Λ is *well-typed* when it is either empty or every element of Λ is well-typed:

6) $\rfloor$;
7) $add(L,\Lambda)$, where L and Λ are well-typed;
8) Λ/L, where L and Λ are well-typed.

A defining equation $E \overset{\text{def}}{=} R$ is *well-typed* if R has the same type as E. Suppose that each agent variable and agent constant is assigned with a higher-order type, then each well-typed process expression and abstraction acquires a unique types as follows:

9) $\mathbf{0_P} : \mathsf{pabs}(\,)$;
10) $\overset{\downarrow}{m}\langle\tilde{u}\rangle : \mathsf{pabs}(\,)$, if $\overset{\downarrow}{m}\langle\tilde{u}\rangle$ is a well-typed action
11) $\overset{\downarrow}{\kappa} : \mathsf{pabs}(\,)$, if $\overset{\downarrow}{\kappa}$ is a well-typed action;
12) $\prod_{i\in I} P_i : \mathsf{pabs}(\,)$ if for each $i \in I$, $P_i : \mathsf{pabs}(\,)$;
13) $\Lambda\circ[G] : \mathsf{pabs}(\,)$ if Λ is well-typed and $G : \mathsf{gabs}(\,)$;
14) $(\nu\ \tilde{n})P : \mathsf{pabs}(\,)$ if $P : \mathsf{pabs}(\,)$ and $\tilde{n}{:}\tilde{\iota}$;
15) $P\langle\tilde{a}\rangle : \mathsf{pabs}(\tilde{\iota})$ if $P : \mathsf{pabs}(\tilde{\iota}_a, \tilde{\iota})$ and $\tilde{a}{:}\tilde{\iota}_a$; and $P\langle\tilde{a}\rangle : \mathsf{pabs}(\,)$ if $P : \mathsf{pabs}(\tilde{\iota}_a)$ and $\tilde{a}{:}\tilde{\iota}_a$;
16) $(\tilde{w})P : \mathsf{pabs}(\tilde{\iota}_n, \tilde{\iota})$ if $P : \mathsf{pabs}(\tilde{\iota})$ and $\tilde{w}{:}\tilde{\iota}_w$; and $(\tilde{w})P : \mathsf{pabs}(\tilde{\iota}_w)$ if $P : \mathsf{pabs}(\,)$ and $\tilde{w}{:}\tilde{\iota}_w$;
17) $\mathbb{P}\langle\!\langle\tilde{P}\rangle\!\rangle : \mathsf{pabs}(\tilde{\xi})$ if $\mathbb{P} : \mathsf{pabs}(\tilde{\xi}_p, \tilde{\xi})$ and $\tilde{P}{:}\tilde{\xi}_p$; and $\mathbb{P}\langle\!\langle\tilde{P}\rangle\!\rangle : \mathsf{pabs}(\,)$ if $\mathbb{P} : \mathsf{pabs}(\tilde{\xi}_p)$ and $\tilde{P}{:}\tilde{\xi}_p$;
18) $\langle\!\langle\tilde{\eta}\rangle\!\rangle\mathbb{P} : \mathsf{pabs}(\tilde{\xi})$ if $\mathbb{P} : \mathsf{pabs}(\,)$, and $\tilde{\eta} : \tilde{\xi}$.

and each well-typed GEC expression and abstraction acquires a unique types as follows:

19) $\mathbf{0_G} : \mathsf{gabs}(\,)$;
20) $\bigotimes_{i\in I} G_i : \mathsf{gabs}(\,)$ if for each $i \in I$, $G_i : \mathsf{gabs}(\,)$;
21) $\overset{\downarrow}{m}(\tilde{x})L.P : \mathsf{gabs}(\,)$ if $P : \mathsf{pabs}(\,)$ and action $\overset{\downarrow}{m}(\tilde{x})$ and lock L are well-typed;

22) $(\nu\ \tilde{n})G : \mathsf{gabs}(\ )$ if $G : \mathsf{gabs}(\ )$ and $\tilde{n}:\tilde{\iota}$;
23) $G\langle\tilde{a}\rangle : \mathsf{gabs}(\tilde{\iota}\ )$ if $G : \mathsf{gabs}(\ \tilde{\iota}_a, \tilde{\iota})$ and $\tilde{a}: \tilde{\iota}_a$; and $G\langle\ \tilde{a}\rangle : \mathsf{gabs}(\ )$ if $G : \mathsf{gabs}(\ \tilde{\iota}_a)$ and $\tilde{a}: \tilde{\iota}_a$;
24) $(\tilde{w})G : \mathsf{gabs}(\tilde{\iota}\ )$ if $G : \mathsf{gabs}(\ )$ and $\tilde{w}: \tilde{\iota}$;
25) $\mathscr{G}\langle\langle\tilde{P}\rangle\rangle : \mathsf{gabs}(\ )$ if $\mathscr{G} : \mathsf{gabs}(\tilde{\xi})$, and $\tilde{P} : \tilde{\xi}$;
26) $\langle\langle\tilde{\eta}\rangle\rangle\mathscr{H} : \mathsf{gabs}(\tilde{\xi})$ if $\mathscr{H} : \mathsf{gabs}(\ )$, and $\tilde{\eta} : \tilde{\xi}$.
27) $\mathscr{G}_1 \otimes \mathscr{G}_2 : \mathsf{gabs}(\tilde{\xi}_1, \tilde{\xi}_2)$ if $\mathscr{G}_1 : \mathsf{gabs}(\tilde{\xi}_1)$, and $\mathscr{G}_2 : \mathsf{gabs}(\tilde{\xi}_2)$.

Where the clause 15, 17 and 24 base on the fact that the type of $S \stackrel{\text{def}}{=} (\tilde{w})(((\tilde{x},\tilde{y})R)\ \langle\tilde{a},\tilde{w}\rangle\ )$ can be determined by that of the partial instanced *R*. The type of a canonical higher order *GEC* choice $\overline{\mathscr{G}}$ is covered by clause 25 and 26, since $\overline{\mathscr{G}}$ is a special case of $\mathscr{G}$. The type of a term $\overline{\mathscr{G}}_1 \boxplus \overline{\mathscr{G}}_2$ depends on the expression of both $\overline{\mathscr{G}}_1$ and $\overline{\mathscr{G}}_2$, and can be individually derived by applying the clause 19 and 25 to 27 above to the definition of operator $\boxplus$.

3 Responsive bisimulation in the κ-calculus

Bisimulation is an important concept in the process algebra community, and the most common idea utilised to mathematically set the behavioural equivalence on processes. It considers two processes are behaviourly the same if both can perform the same action at every step in the every evolution path. Existing bisimulations can fail to capture the behaviour equivalence between the composed object and the target object. As an example, the process O_1 and O_2 illustrated in Figure 3-1 represent two different versions of the internal structure of the same composed object in a state where its only method is blocked by the lock of key κ. The only difference between these two is that O_1 has an extra "empty" control $Ctrl_e$ which does nothing but forwards whatever message received from channel *m* to the next control $Ctrl_1$. The body of these two can always give the same response if fed with the same message. If an unlocking signal is received via channel κ, both O_1 and O_2 can accept incoming messages and process them immediately, as shown in Figure 3-1a. If some message arrives before the unlocking, O_1 will store it in an internal buffer and delay the process until unlocked, but O_2 will leave the message in the external buffer as it was while waiting for unlocking, as shown in Figure 3-1b. To the client who sent the message, the two objects O_1 and O_2 are behaviourly the same because they always give the same response. However, existing bisimulations usually distinguish the two because in the situation of Figure 3-1b, O_1 can input the message but O_2 does not. To solve this problem, Zhang and Potter proposed the responsive bisimulation ([Zhang02A], [Zhang02B]) and extended to it the κ-calculus ([Zhang02D]). In this section we only give a very brief introduction about responsive bisimulation, for which the detailed study it can be found in [Zhang02A], [Zhang02B] and [Zhang02D].

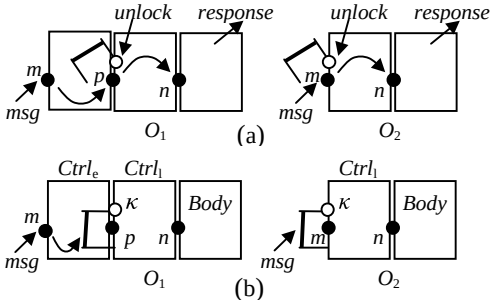


Figure 3-1

In object-oriented systems, the lock/unlock actions are usually internal activities of objects, and therefore may not be visible from outside. However, while study on a component process of a system or object, these activities have to be observed. In the κ-calculus, the distinction between names for locking keys and for communication allows us to take two different positions in observing processes interactive behaviours:

1. ignore all locking/releasing actions, and adopted the same set bisimulation relations developed in the polar π-calculus;
2. take locking/releasing actions into account and therefore produce the "κ-variation", an even finer version, for each of those bisimulation relations.

Thus, variations of bisimulation relations will be doubled. For every those bisimulations, each κ-version bisimulation is a subset of its non-κ-version counterpart. And in the polar π-calculus, which is a sub-calculus of the κ-calculus, the κ-version and non-κ-version bisimulations will coincide respectively.

Generally say, the κ-version bisimulations are needed for measuring properties of object components, when non-κ-version bisimulations are intersted in measuring overal behaviour of composd objects.

To measure the observation behaviours, we need to distinguish the similarity in responses perceived by outsiders, butignore the unrelated information. We must note that the state changes of a process caused by internal actions, and we must also be able to detect which communication channels are available for output in all evolved states. What is more, in order to distinguish states, we need to be able to observe what each of the messages output by the process is. The οτ-bisimulation, can provide this degree of observation:

Definition 3−21: The (strong) *οτ*-**bisimulation** is a symmetric relation $\mathcal{S}$ on processes such that whenever $P\mathcal{S}Q$ then $P\xrightarrow{\alpha}P'$ implies $Q\xrightarrow{\alpha}Q'$ and $P'\mathcal{S}Q'$ for all action α in the form of either $\alpha=(\nu\,\tilde{v})\overline{m}\langle\tilde{u}\rangle$ or $\alpha=\tau$, and $bn(\alpha)\cap fn(Q)=\varnothing$.

The κ-version, (strong) *κοτ*-**bisimulation**, is a strong οτ-bisimulation $\mathcal{S}$ such that whenever $P\mathcal{S}Q$ then $P\xrightarrow{\alpha}P'$ implies $Q\xrightarrow{\alpha}Q'$ and $P'\mathcal{S}Q'$ for all $\alpha\in\mathcal{K}^{+}\cup\bar{\mathcal{K}}^{-}$.

The weak οτ-bisimulation and weak κοτ-bisimulation are obtained by replacing $\xrightarrow{\alpha}$ with $\overset{\alpha}{\Rightarrow}$ everywhere above respectively. We denote $\sim_{o\tau}$ be the largest οτ-bisimulation, and $\approx_{o\tau}$ be the largest weak οτ-bisimulation, $\sim_{\kappa o\tau}$ be the largest κοτ-bisimulation, and $\approx_{\kappa o\tau}$ be the largest weak κοτ-bisimulation.

The οτ-bisimulation gives a measurement on processes' states by observing available reductions and output actions, but can not determine how a process responses to incoming messages, since communicating input actions are not observed. To determine responsive behaviours, we introduce a new behaviour equivalence relations.

Definition 3−22: Let $\mathcal{T}[.]$ be the **responsive testing context** of syntax $\mathcal{T}::=[.]\mid[\dagger m\langle\tilde{u}\rangle]\mathcal{T}$, then we define
the strong and weak **responsive equivalence**: $P\simeq_r Q$ iff $\forall\mathcal{T}.(\mathcal{T}[P]\sim_{o\tau}\mathcal{T}[Q])$, $P\cong_r Q$ iff $\forall\mathcal{T}.(\mathcal{T}[P]\approx_{o\tau}\mathcal{T}[Q])$;
the strong and weak ***κr*- equivalence**: $P\simeq_{\kappa r}Q$ iff $\forall\mathcal{T}.(\mathcal{T}[P]\sim_{\kappa o\tau}\mathcal{T}[Q])$, $P\cong_{\kappa r}Q$ iff $\forall\mathcal{T}.(\mathcal{T}[P]\approx_{\kappa o\tau}\mathcal{T}[Q])$.

This definition gives a quite clear description about the meaning of equivalence in responsive behaviour, but is not so useful since it requires the exhaustive testing over the infinite set of responsive testing contexts. A more practical definition is the r1-bisimulation, named so because the structurally comparable to the 1-bisimulation in [Amadio96].

Definition 3−23: The strong (or weak) *r*1-**bisimulation** is a strong (or weak, respectively) οτ-bisimulation $\mathcal{S}$ if whenever $P\mathcal{S}Q$ then $[\dagger m\langle\tilde{u}\rangle]P\mathcal{S}[\dagger m\langle\tilde{u}\rangle]Q$ for all $[\dagger m\langle\tilde{u}\rangle]$.

We denote the largest strong r1-bisimulation as $\sim_{r1}$, and the largest weak r1-bisimulation as $\approx_{r1}$.

The κ-versions, strong and weak *κr*1-**bisimulation** $\sim_{\kappa r1}$ and $\approx_{\kappa r1}$, are defined by replacing οτ-bisimulation with its κ-version, the κοτ-bisimulation, in the above definition.

While responsive equivalences and r1-bisimulations provide a good base for describing similarities of responsive behaviours, they tell little about why or when two processes may offer similar behaviours. For closer study, we need an inside view observing input actions.

Definition 3−24 : The (strong) **responsive bisimulation** is a (strong) οτ-bisimulation $\mathcal{S}$ such that whenever $P\mathcal{S}Q$ then $P\xrightarrow{\dagger m(\tilde{u})}P'$ implies either $Q\xrightarrow{\dagger m(\tilde{u})}Q'$ and $P'\mathcal{S}Q'$, or $Q\xrightarrow{\tau}Q'$ and $P'\mathcal{S}[\dagger m\langle\tilde{u}\rangle]Q'$.

The weak responsive bisimulation is obtained by replacing transitions with weak transitions everywhere. We denote $\sim_r$ and $\approx_r$ be the largest strong and weak responsive bisimulation respectively. Clearly, $\sim_r\subseteq\approx_r$.

The κ-versions, strong and weak *κr*-**bisimulation** $\sim_{\kappa r}$ and $\approx_{\kappa r}$, are defined by replace οτ-bisimulation with κοτ-bisimulation in the above definitions. Clearly, $\sim_{\kappa r}\subseteq\approx_{\kappa r}$.

Lemma 3−25: The responsive bisimulation, r1-bisimulation and responsive equivalence are coincide for both κ-version and non-κ-version, i.e., $\sim_{\kappa r}\equiv\sim_{\kappa r1}\equiv\simeq_{\kappa r}$, $\approx_{\kappa r}\equiv\approx_{\kappa r1}\equiv\cong_{\kappa r}$, $\sim_r\equiv\sim_{r1}\equiv\simeq_r$ and $\approx_r\equiv\approx_{r1}\equiv\cong_r$.

The responsive bisimulation has the following properties.

Lemma 3−26: The responsive bisimulations are equivalences, in other words, it is reflexive, symmetric and transitive.

Proposition 3−27: The responsive bisimulations are preserved by restriction, localisation, and output polarity name substitution. That is, let $\mathcal{S}$ be any of $\sim_r$, $\approx_r$, $\sim_{\kappa r}$ or $\approx_{\kappa r}$, then $P\mathcal{S}Q$ implies

$(\nu\,\tilde{n})P\mathcal{S}(\nu\,\tilde{n})Q$ for all $\tilde{n}$;
$[\natural m\langle\tilde{u}\rangle]P\ \mathcal{S}\ [\natural m\langle\tilde{u}\rangle]Q$ for all $[\natural m\langle\tilde{u}\rangle]$; and
$P\sigma\mathcal{S}Q\sigma$ for all $\sigma=\{\tilde{u}/\tilde{x}\}$.

There is a problem: the responsive bisimulations are not be preserved by parallel composition in general. For instance, with the O_1 and O_2 of the previous example, we have $O_1\sim_r O_2$, but $(O_1\,|\,O_3)\not\sim_r (O_2\,|\,O_3)$ for $O_3\stackrel{\text{def}}{=} \rfloor\circ(!\natural m(\tilde{v})L.R]$, because the occurrence of input polar $\natural m$ in O_3 has changed the ability of O_1 on receiving message from $\natural m$. However, as mentioned at the beginning of this paper, the purpose of our study is about object modelling, and as the nature of object systems, the ownership of each input port should be unique. For example, the object identity of an object is uniquely owned by no one else but that object; each method of each object is also uniquely identified so that no message would be delivered to wrong destination. In general, as mentioned in the previous session, each input polar has a static scope (or ownership), and will never appears outside this scope.

When responsive bisimulation is strictly restricted within the problem domain, objects modelling, where the responsive bisimulation is needed, then its preservation in parallel composition can be guaranteed, as shown later.

Definition 3−28: Let $\natural m$ be the input polar of a communication channel name m, P be a process for which $m\in fin(P)$, and $\mathcal{E}$ be the context $\mathcal{E}[.]\stackrel{\text{def}}{=}(\nu\,\tilde{n})\,(Env\,|\,[.]\,)$ where $m\notin fin(Env)$ while m may or may not be a member of $\tilde{n}$. We say that, P is an *owner* of $\natural m$ (or say, $\natural m$ is owned by P) with respect to the *environment Env*;
Env is an environment free of $\natural m$ (or say, *$\natural m$-free environment*);
$\mathcal{E}[.]$ is an *$\natural m$-safe* environment context, or *$\natural m$*-safe environment for short.

An *$\natural m$*-safe environment only allows the process in the hole to consume a message sent along the channel m, ensuring no interference from the environment. It reflects the fact that the responsive behaviour of a process can be measured only when messages sent to it are guaranteed not to be intercepted by some other process.

Definition 3−29 A process P is *safe* for *Env*, and the environment *Env* is said to be *safe* for P, if P is the owner of all $m\in fin(P)$ respect to the environment *Env*, i.e., $fin(P)\cap fin(Env)=\varnothing$. We may call P an *safe process*, when the behaviour of P is only considered within environments which are safe for P.

A process P is *autonomous* if $fin(P)=\varnothing$.

Lemma 3−30: The process safety is preserved by evolution. That is, if $fin(P)\cap fin(Env)=\varnothing$ holds for processes P and *Env*, then $fin(P')\cap fin(Env')=\varnothing$ holds for all $P'$and Env', which are derivatives of P and *Env* respectively.

Proof: Simply because the input polar of a channel cannot be transmitted by communication. ■

Corollary 3−31: An autonomous process and all its derivatives are safe to any system.

When modelling objects in the κ-calculus, all method bodies can be considered as autonomous, since after parameters passed through the method interface, further input (if any) can only be performed via channels that were initially private and informed to the senders by the forked method body. An object itself is initially autonomous while creation, until its name, the unique identification, is exported to its environment. Its method names can also be considered as initially private to the object, and then exported to the caller during each method call. For example, similar to [Walker95] and

[Zhang97] amongst others, the method call `o.m1(a1,a2)` may be modelled as $(\nu\, mset)(\bar{o}\langle mset\rangle \mid \sharp mset(\tilde{m}).\bar{m}_1\langle a_1,a_2\rangle)$, and on the object side the encoding will look like $(\nu\ \tilde{m})\ (!o(mset).\ \overline{mset}\langle\tilde{m}\rangle \mid\ \Lambda\circ[\bigotimes !\sharp m_i(\tilde{v})L_i.Body_i]\ )$.

Proposition 3−32: The responsive bisimulations are preserved by parallel composition for safe processes. That is, to each of the κ-version or non-κ-version responsive bisimulations $\mathcal{S}$, whenever $P_1\mathcal{S}P_2$ implies $(P_1|P)\mathcal{S}(P_2|P)$ for all P which satisfying $fin(P)\cap(fin(P_1)\cup fin(P_2))=\varnothing$.

The following proposition is equivelant to say, in the term of ordinary π–calculi, the responsive bisimulations are preserved by input prefix, replication, choice and, outside the π–calculi scope, lock, for autonomous processes.

Proposition 3−33: The responsive bisimulations are preserved by *GEC* choice for autonomous processes. That is, to each of the κ-version or non-κ-version responsive bisimulations $\mathcal{S}$, if P_1 and P_2 are autonomous processes, then $P_1\mathcal{S}P_2$ implies $\mathcal{D}[P_1]\mathcal{S}\mathcal{D}[P_2]$ for all context of the form $\mathcal{D}[.]\stackrel{def}{=}\Lambda\circ[!(\nu\,\kappa)\sharp m(\tilde{x})L.[.]\otimes G]$.

Proposition 3−34: For autonomous processes, the responsive bisimulations are congruences. That is, for each of the κ-version or non-κ-version responsive bisimulations $\mathcal{S}$, if P_1 and P_2 are autonomous processes, then $P_1\mathcal{S}P_2$ implies $\mathcal{C}[P_1]\mathcal{S}\mathcal{C}[P_2]$ for all **process context** given by syntax $\mathcal{C}::=[.]\mid(\nu\,\tilde{n})\mathcal{C}\mid\mathcal{C}|P\mid\Lambda\circ[!(\nu\,\kappa)\beta.\,\mathcal{C}\otimes G]$.

4 Object Model in the κ-calculus

In this section we will demonstrate that the idear of [Zhang98A] and [Zhang98B] in modelling compositional concurrent objects can be better presented in the κ-calculus, and show that the κ-calculus can naturally divide different aspects of concurrency into structurally different parts of an expression, and reason about them separately.

4.1 Compositional object model

In the concurrent object models of [Zhang98A] and [Zhang98B], which was described in the π-calculus version, the behaviour of a concurrent object can be represented as the parallel composition of a process F representing the object's functional behaviour with no constraint on its concurrent interactions, and a process C representing the constraints on the object's concurrent behaviour. For example, the functionality of a buffer object can be described by

$$F \stackrel{def}{=} !m_r(x).M_r\langle x\rangle \mid !m_w(x).M_w\langle x\rangle,$$

where $m_r(x).M_r\langle x\rangle$ and $m_w(x).M_w\langle x\rangle$ represent the behaviour of the read and write methods respectively; each of them can have unlimited invocations executing in parallel with no concern for any potential interference. To discipline those invocations, assume a synchronisation behaviour modelled by the control process

$$C \stackrel{def}{=} n_r(x).\overline{m}_r\langle x\rangle + n_w(x).\overline{m}_w\langle x\rangle,$$

where the choice operator in fact represents a mutual exclusion lock on those methods. Then the parallel composition of the two processes, $(\nu\, m)(C\mid F)$, will be weakly bisimilar to

$$R \stackrel{def}{=} n_r(x).M_r\langle x\rangle + n_w(x).M_w\langle x\rangle$$

which describes the combined behaviours as expected.

When presenting the similar model in the κ-calculus, the functionality of an object may be written as :

$$\begin{aligned} F &\stackrel{def}{=} (\sharp\tilde{m},\tilde{m})\ \ ]\circ[\bigotimes_{i\in 1} !\sharp m_i(\tilde{y})(\nu)@\varnothing.M_i\langle\tilde{m},\tilde{y}\rangle\,] \\ &\equiv (\sharp\tilde{m},\tilde{m})\prod_{i\in 1} !\sharp m_i(\tilde{y}).M_i\langle\tilde{m},\tilde{y}\rangle \end{aligned} \qquad 4-1$$

The first presentation in this equation emphases that for every method, there is no exclusion constraint has been specified, and therefore unlimit copies of each M_i, which represents the behaviour of the body of the i^{th} method, can be

"forked" by arriving messages in the corresponding input polar $\overline{m}_I$, the identifier of the methods. The process abstraction F can also be presented in a higher-order form

$$F \equiv (\tilde{m},\tilde{m}) \quad ⌋\circ(\overline{\tilde{E}}\langle\overline{\tilde{m}}\rangle \mathbin{«} M_{i\in I}\langle\tilde{m}\rangle \mathbin{»}) \qquad 4\text{−}2$$

where $\overline{\tilde{E}}$ is an empty exclusion control, which does nothing except passes messages to corresponding mathod bodies:

$$\overline{\tilde{E}} \stackrel{\text{def}}{=} (\overline{\tilde{m}}) « \tilde{\eta} » \bigotimes_{i\in I} !\overline{m}_i(\tilde{y})(\nu)@\varnothing.\,\eta_i\langle\tilde{y}\rangle \qquad 4\text{−}3$$

Here $\overline{\tilde{E}}$ can be considered as an abstraction of the functional object, which provides the interface, a set of the input polar of channels (methods), of the object. In other words, the equation 4−2 describes an object's functionality in an form with separation of the object's method interface and the "implementation" of method bodies (Figure 4-1).

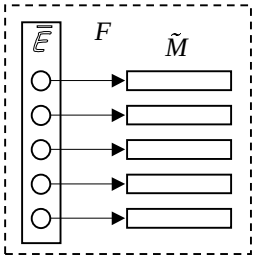


Figure 4-1

Now we can say, for the generic form of objects, the functionality $F:\mathtt{pabs}(\tilde{\lambda}_m,\tilde{\lambda}_m)$, is a process with an interface, an empty exclusion control, and a set of method body definitions.

While $\overline{\tilde{E}}$ is presented in the form of $\overline{\tilde{E}} \equiv \overline{\tilde{E}}_1 ⊞ \overline{\tilde{E}}_2 ⊞ \ldots ⊞ \overline{\tilde{E}}_n$, we may consider that each $\overline{\tilde{E}}_i$ describes a portion of the interface.

A single control process C, which represents a concurrent behaviours, may be modelled in the κ-calculus as:

$$C \stackrel{\text{def}}{=} (\tilde{n},\tilde{m}) \quad ⌋\circ(\overline{\tilde{G}}\langle\tilde{n}\rangle \mathbin{«} C_{T\,i\in I}\langle\tilde{m}\rangle \mathbin{»}) \qquad 4\text{−}4$$

The control process C itself can be divided into several parts, each determines a difference aspect of behaviour. ⌋ is an empty locking list. The GEC choice $\overline{\tilde{G}}$ purely describes the exclusion.

$$\overline{\tilde{G}} \stackrel{\text{def}}{=} (\tilde{n}) « \tilde{\eta} » \bigotimes_{i\in I} !(\nu\,\kappa)\,\overline{n}_i(\tilde{y})\;\check{\kappa}@J_i.\;\eta_i\langle\tilde{x},\hat{\kappa}\rangle \qquad 4\text{−}5$$

The role of $\overline{\tilde{G}}$ here is like an exclusion table for methods, where each J_i lists all the methods should be locked when the i[th] method being invoked. For example

$$\overline{\tilde{G}}_1 \stackrel{\text{def}}{=} (\overline{m}_a,\overline{m}_b,\overline{m}_c) « \eta_a,\eta_b,\eta_c » (!(\nu\,\kappa)\,\overline{m}_a(\tilde{y})\,\check{\kappa}@[\overline{m}_a,\overline{m}_b].\,\eta_a\langle\tilde{x},\hat{\kappa}\rangle \otimes !(\nu\,\kappa)\,\overline{m}_b(\tilde{y})\,\check{\kappa}@[\overline{m}_c].\,\eta_b\langle\tilde{x},\hat{\kappa}\rangle \otimes !(\nu\,\kappa)\,\overline{m}_c(\tilde{y})\,\check{\kappa}@[\overline{m}_b].\,\eta_c\langle\tilde{x},\hat{\kappa}\rangle)$$

indicates that an invocation of method m_a will lock method m_b and another copy of m_a, and an invocation of method m_b will lock method m_c, an invocation of m_c will lock m_b. The exclusion table which $\overline{\tilde{G}}_1$ represents is shown in Figure 4-3.

While $\overline{\tilde{G}}$ is presented in the form $\overline{\tilde{G}} \equiv \overline{\tilde{G}}_1 ⊞ \overline{\tilde{G}}_2 ⊞ \ldots ⊞ \overline{\tilde{G}}_n$, then each indicates a portion of the interface and the exclusion relation among the methods within this portion.

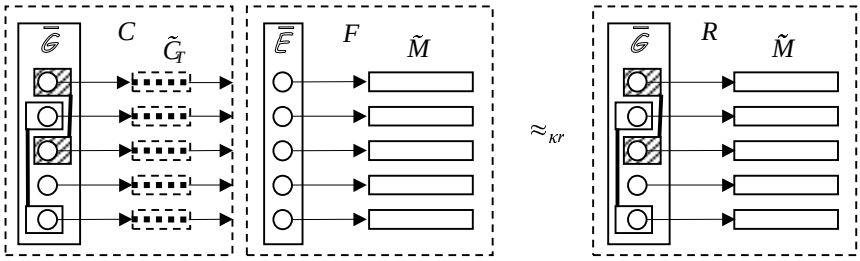


Figure 4-2 A visualised explanation of Equation 4−7

$C_{Ti\in I}$, the continuations of $\overline{\tilde{G}}$, can be used to specify what other synchronisation should be done after the invocaion of each method, including when a lock should be released. That is, $C_{Ti\in I}$ may play a role as "*scheduler*", which we will demonstrate in details in later sections. The simplest example of C_{Ti} can be defined as

$$C_{Ti} \stackrel{\text{def}}{=} (\tilde{m})(\tilde{y},\hat{\kappa})\overline{m}_i\langle\tilde{y}\rangle, \qquad 4\text{−}6$$

which simply forwards whatever message received to the method body to process, and provides no mechanism for releasing locked methods. In other words, all locks laid by $\overline{\tilde{G}}$ will become permanent. For this case, if write $R \stackrel{\text{def}}{=} (\tilde{n},\tilde{n}) \quad ⌋\circ(\overline{\tilde{G}}\langle\tilde{n}\rangle « (\tilde{y},\hat{\kappa})M_{i\in I}\langle\tilde{n},\tilde{y}\rangle »)$, then we have

$(\nu\ \tilde{m})\,(C\langle\tilde{n},\tilde{m}\rangle \,|\, F\langle\tilde{m},\tilde{n}\rangle) \approx_{Kr} R\langle\tilde{n},\tilde{n}\rangle,$

or $(\nu\ \tilde{m})\,(\ \rfloor\circ[\overline{\mathcal{G}}\langle\tilde{n}\rangle \ll(\tilde{y},\hat{\kappa})m_{i\epsilon I1}\langle\tilde{y}\rangle\gg]\ |\ \rfloor\circ[\overline{\mathcal{E}}\langle\tilde{m}\rangle \ll(\tilde{y})M_{i\epsilon I1}\langle\tilde{n},\tilde{y}\rangle\gg]\ ) \approx_{Kr} \rfloor\circ[\overline{\mathcal{G}}\langle\tilde{n}\rangle\ll(\tilde{y},\hat{\kappa})M_{i\epsilon I1}\langle\tilde{n},\tilde{y}\rangle\gg].$ 4−7

That is, the exclusion relation described by C is enforced into F. The diagram illustrated in Figure 4-2 is another way to present the meaning of Equation 4−7.

Several control processes may be compounded to from a new one with the composed behaviours. For example, assume $C_k \triangleq^{def} (\tilde{n},\tilde{m})\ \rfloor\circ[\overline{\mathcal{G}}_k\langle\tilde{n}\rangle\ll G_{\Gamma i\epsilon I}\langle\tilde{m}\rangle\gg]$ for k=1,2, then with the same $G_{\Gamma i\epsilon I}$ definded in equation 4-6, the composed control C constructed by

$C \triangleq^{def} (\tilde{n},\tilde{m})\,(\nu\ \tilde{p})\,(C_1\langle\tilde{n},\tilde{p}\rangle \,|\, C_2\langle\tilde{p},\tilde{m}\rangle)$ 4−8

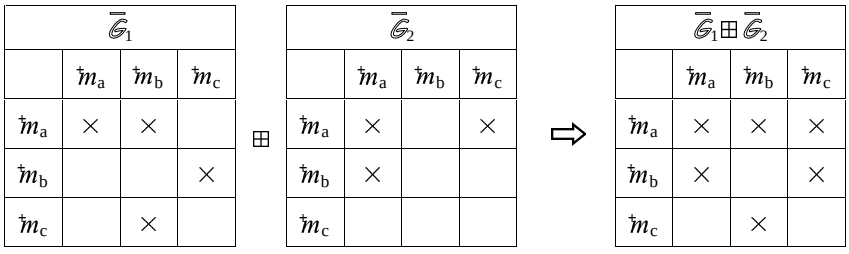
will satisfy $C\langle\tilde{n},\tilde{m}\rangle \approx_{Kr} C'\langle\tilde{n},\tilde{m}\rangle$ for $C' \triangleq^{def} (\tilde{n},\tilde{m})\ \rfloor\circ[(\overline{\mathcal{G}}_1\langle\tilde{n}\rangle \boxplus \overline{\mathcal{G}}_2\langle\tilde{n}\rangle)\ll G_{\Gamma i\epsilon I}\langle\tilde{m}\rangle\gg].$

$C\langle\tilde{n},\tilde{m}\rangle \approx_{Kr} C'\langle\tilde{n},\tilde{m}\rangle$ for $C' \triangleq^{def} (\tilde{n},\tilde{m})\ \rfloor\circ[(\overline{\mathcal{G}}_1\langle\tilde{n}\rangle \boxplus \overline{\mathcal{G}}_2\langle\tilde{n}\rangle)\ll G_{\Gamma i\epsilon I}\langle\tilde{m}\rangle\gg].$

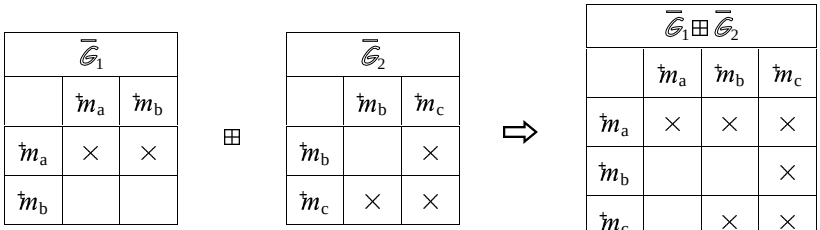
That is, by the definition of $\boxplus$, the compositionl effect on exclusion is to union the lock sets of each corresponding branch of $\overline{\mathcal{G}}_1$ and $\overline{\mathcal{G}}_2$:

$(\bigotimes_{i\epsilon I}!(\nu\ \kappa)\natural n_i(\tilde{y})\ \check{\kappa}@J_{1i}.\eta_i\langle\tilde{x},\hat{\kappa}\rangle) \boxplus (\bigotimes_{i\epsilon I}!(\nu\ \kappa)\natural n_i(\tilde{y})\ \check{\kappa}@J_{2i}.\eta_i\langle\tilde{x},\hat{\kappa}\rangle) \equiv \rfloor\circ[\bigotimes_{i\epsilon I}!(\nu\ \kappa)\natural n_i(\tilde{y})\ \check{\kappa}@(J_{ni}\cup J_{pi}).\eta_i\langle\tilde{x},\hat{\kappa}\rangle].$

Figure 4-3 (a) is a detailed example of such a composition.



(a)



(b)

Figure 4-3 Examples of superposing exclusion

In fact, C_1 can C_2 do not have to be of the same arity or of the same type (Figure 4-3b). The following proposition describes such a composition more generically:

Proposition 4−35: Let $\tilde{m}{:}\tilde{\lambda}$, $\tilde{n}{:}\tilde{\lambda}$ and $\tilde{p}{:}\tilde{\lambda}$ be name sets of the same set of link types, write $G_{\Gamma i} \triangleq^{def} (\tilde{m})(\tilde{y},\hat{\kappa})m_i\langle\tilde{y}\rangle$, $\overline{\mathcal{G}}_k \triangleq^{def} (\tilde{n})\ll\tilde{\eta}\gg\bigotimes_{i\epsilon I_k}!(\nu\ \kappa)\natural n_i(\tilde{y})\ \check{\kappa}@J_{ki}.\eta_i\langle\tilde{y},\hat{\kappa}\rangle$, and given

$G_1 \triangleq^{def} \overline{\mathcal{G}}_1\langle\{\natural n_{i\epsilon I_1}\}\rangle\ll\{G_{\Gamma i\epsilon I_1}\langle\{m_{i\epsilon(I_1-I_2)}, p_{i\epsilon(I_2\cap I_1)}\}\rangle\}\gg$,

$G_2 \triangleq^{def} \overline{\mathcal{G}}_2\langle\{p_{i\epsilon(I_2\cap I_1)}, \natural m_{i\epsilon(I_2-I_1)}\}\rangle\ll\{G_{\Gamma i\epsilon I_2}\langle\{m_{i\epsilon I_2}\}\rangle\}\gg$, and

$G \triangleq (\overline{\mathcal{G}}_1\langle\{\natural n_{i\epsilon I_1}\}\rangle \boxplus \overline{\mathcal{G}}_2\langle\{\natural n_{i\epsilon I_2}\}\rangle)\ll\{G_{\Gamma i\epsilon(I_1\cup I_2)}\langle\tilde{m}\rangle\}\gg$, then

$(\nu\ \tilde{p})(\ \rfloor\circ[G_1]\ |\ \rfloor\circ[G_2]) \approx_{Kr} \rfloor\circ[G].$

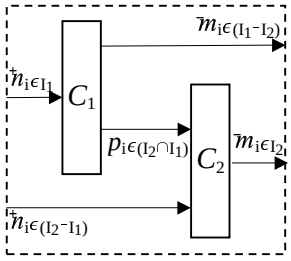


Figure 4-4

Proof: Assume some locking states Λ, Λ_1 and Λ_2 and some autonomous processes P and P_1 which satisfy the following conditions:

1. $(lset(\Lambda_1)\cap guard(G_1))\cup((lset(\Lambda_2)\cap guard(G_2))\{\tilde{n}/\tilde{p}\}) \equiv lset(\Lambda)\cap guard(G)$;

2. P is either $\mathbf{0}$ or of the form $P \stackrel{\text{def}}{\equiv} \prod_{I'} \overline{m}_i\langle\tilde{y}\rangle$, where $I' \subseteq I_1 \cup I_2$;
3. P_1 is either $\mathbf{0}$, or of the form $P_1 \stackrel{\text{def}}{\equiv} \prod_{I'_1} \overline{p}_i\langle\tilde{y}\rangle$ and $\{\overline{p}_{i\in I'_1}\} \subseteq I\!lset(\Lambda_2)$, where $I'_1 \subseteq I_1 \cap I_2$.

Write $\tilde{\kappa}_1 = keys(\Lambda_1) \cup keys(\Lambda_2)$ and $\tilde{\kappa}_2 = keys(\Lambda)$. Now, if we can prove

$$(\nu\ \tilde{p}, \tilde{\kappa}_1)\,((\Lambda_1 \circ [G_1] \mid \Lambda_2 \circ [G_2])) \mid P \mid P_1) \approx_{kr} (\nu\ \tilde{\kappa}_2)\,(\Lambda \circ [G] \mid P)$$

then this proposition is obtained by letting $\Lambda = \Lambda_1 = \Lambda_2 = \rfloor$ and $P = P_1 = \mathbf{0}$.

Let $R_1 \stackrel{\text{def}}{\equiv} (\nu\ \tilde{p}, \tilde{\kappa}_1)\,((\Lambda_1 \circ [G_1] \mid \Lambda_2 \circ [G_2])) \mid P_1 \mid P_2)$ and $R_2 \stackrel{\text{def}}{\equiv} (\nu\ \tilde{\kappa}_2)\,(\Lambda \circ [G] \mid P)$, and let $\mathcal{S}$ be the symmetric relation such that $R_1 \mathcal{S} R_2$. Write $L_{ki} = k @ J_{ki}$. For both R_1 and R_2, the only actions can be taken are inputs via a channel in $\tilde{n}$, and outputs via a channel in $\tilde{m}$:

$R_1 \xrightarrow{\overline{m}_i\langle\tilde{y}\rangle} R'_1$: It can only possible when $P \xrightarrow{\overline{m}_i\langle\tilde{y}\rangle} P'$, therefore $R_2 \xrightarrow{\overline{m}_i\langle\tilde{y}\rangle} R'_2$ and
$R'_1 \equiv (\nu\ \tilde{p}, \tilde{\kappa}_1)\,((\Lambda_1 \circ [G_1] \mid \Lambda_2 \circ [G_2])) \mid P' \mid P_1)$ and $R'_2 \equiv (\nu\ \tilde{\kappa}_2)\,(\Lambda \circ [G] \mid P')$.
It is easy to see from condition 2, that P' also satisfies the condition 2. That is, $(R'_1, R'_2) \in \mathcal{S}$.

$R_2 \xrightarrow{\overline{m}_i\langle\tilde{y}\rangle} R'_2$: In the same way as the above, we have $P \xrightarrow{\overline{m}_i\langle\tilde{y}\rangle} P'$, $R_1 \xrightarrow{\overline{m}_i\langle\tilde{y}\rangle} R'_1$ and $(R'_1, R'_2) \in \mathcal{S}$.

$R_1 \xrightarrow{\dagger n_i(\tilde{y})} R'_1$: It can only be one of the following cases ($R_1 \xrightarrow{\dagger n_i(\tilde{y})} R'_1$:)

$i \in (I_1 - I_2)$, $R'_1 \equiv (\nu\ \tilde{p}, \tilde{\kappa}_1, key(L_{1i}))\,((addl(L_{1i}, \Lambda_1) \circ [G_1] \mid \Lambda_2 \circ [G_2])) \mid P \mid \overline{m}_i\langle\tilde{y}\rangle \mid P_1)$, and $R_2 \xrightarrow{\dagger n_i(\tilde{y})} R'_2$, where
$R'_2 \equiv (\nu\ \tilde{\kappa}_2, key(L_i))\,((addl(L_i, \Lambda) \circ [G] \mid P \mid \overline{m}_i\langle\tilde{y}\rangle)$. Clearly, $(R'_1, R'_2) \in \mathcal{S}$;

$i \in (I_2 - I_1)$, $R'_1 \equiv (\nu\ \tilde{p}, \tilde{\kappa}_1, key(L_{2i}))\,(\Lambda_1 \circ [G_1] \mid (addl(L_{2i}, \Lambda_2) \circ [G_2])) \mid P \mid \overline{m}_i\langle\tilde{y}\rangle \mid P_1)$, and $R_2 \xrightarrow{\dagger n_i(\tilde{y})} R'_2$, where
$R'_2 \equiv (\nu\ \tilde{\kappa}_2, key(L_i))\,((addl(L_i, \Lambda) \circ [G] \mid P \mid \overline{m}_i\langle\tilde{y}\rangle)$. Clearly, $(R'_1, R'_2) \in \mathcal{S}$;

$i \in (I_1 \cap I_2)$, $R'_1 \equiv (\nu\ \tilde{p}, \tilde{\kappa}_1, key(L_{1i}))\,((addl(L_{1i}, \Lambda_1) \circ [G_1] \mid \Lambda_2 \circ [G_2])) \mid P \mid \overline{p}_i\langle\tilde{y}\rangle \mid P_1)$. There are two cases.
$p_i \notin lset(\Lambda_2)$, then $R'_1 \xrightarrow{\tau} R''_1$, and by condition 1, $n_i \notin lset(\Lambda)$, and therefore $R_2 \xrightarrow{\dagger n_i(\tilde{y})} R'_2$, where
$R''_1 \equiv (\nu\ \tilde{p}, \tilde{\kappa}_1, key(L_{1i}), key(L_{2i}))\,((addl(L_{1i}, \Lambda_1) \circ [G_1] \mid addl(L_{2i}, \Lambda_2) \circ [G_2])) \mid P \mid \overline{m}_i\langle\tilde{y}\rangle \mid P_1)$,
$R'_2 \equiv (\nu\ \tilde{\kappa}_2, key(L_i))\,((addl(L_i, \Lambda) \circ [G] \mid P \mid \overline{m}_i\langle\tilde{y}\rangle)$.
From definition of G, we have $lset(L_i) = lset(L_{1i}) \cup (lset(L_{2i}\{\tilde{n}/\tilde{p}\})$, that is, condition 1 is maintained. Clearly, $(R''_1, R'_2) \in \mathcal{S}$;
$p_i \in lset(\Lambda_2)$, then by condition 1, $n_i \in lset(\Lambda)$ and therefore $R_2 \not\downarrow \overline{n}_i$. Writer $P'_1 \stackrel{\text{def}}{\equiv} \overline{p}_i\langle\tilde{y}\rangle \mid P_1$, clearly, $(R'_1, [\dagger n_i\langle\tilde{y}\rangle] R_2) \in \mathcal{S}$.

$R_2 \xrightarrow{\dagger n_i(\tilde{y})} R'_2$: Similar to the previous one, it can be possible only when in the following cases

$i \in (I_1 - I_2)$, $R'_2 \equiv (\nu\ \tilde{\kappa}_2, key(L_i))\,((addl(L_i, \Lambda) \circ [G] \mid P \mid \overline{m}_i\langle\tilde{y}\rangle)$, and $R_1 \xrightarrow{\dagger n_i(\tilde{y})} R'_1$, where
$R'_1 \equiv (\nu\ \tilde{p}, \tilde{\kappa}_1, key(L_{1i}))\,((addl(L_{1i}, \Lambda_1) \circ [G_1] \mid \Lambda_2 \circ [G_2])) \mid P \mid \overline{m}_i\langle\tilde{y}\rangle \mid P_1)$. Clearly, $(R'_1, R'_2) \in \mathcal{S}$;

$i \in (I_2 - I_1)$, $R'_2 \equiv (\nu\ \tilde{\kappa}_2, key(L_i))\,((addl(L_i, \Lambda) \circ [G] \mid P \mid \overline{m}_i\langle\tilde{y}\rangle)$, and $R_2 \xrightarrow{\dagger n_i(\tilde{y})} R'_2$, where
$R'_1 \equiv (\nu\ \tilde{p}, \tilde{\kappa}_1, key(L_{2i}))\,(\Lambda_1 \circ [G_1] \mid (addl(L_{2i}, \Lambda_2) \circ [G_2])) \mid P \mid \overline{m}_i\langle\tilde{y}\rangle \mid P_1)$. Clearly, $(R'_1, R'_2) \in \mathcal{S}$;

$i \in (I_1 \cap I_2)$, $R'_2 \equiv (\nu\ \tilde{\kappa}_2, key(L_i))\,((addl(L_i, \Lambda) \circ [G] \mid P \mid \overline{m}_i\langle\tilde{y}\rangle)$. By condifion 1, we have $R_1 \xrightarrow{\dagger n_i(\tilde{y})} R'_1 \xrightarrow{\tau} R''_1$,
$R'_1 \equiv (\nu\ \tilde{p}, \tilde{\kappa}_1, key(L_{1i}))\,((addl(L_{1i}, \Lambda_1) \circ [G_1] \mid \Lambda_2 \circ [G_2])) \mid P \mid \overline{p}_i\langle\tilde{y}\rangle \mid P_1)$,
$R''_1 \equiv (\nu\ \tilde{p}, \tilde{\kappa}_1, key(L_{1i}), key(L_{2i}))\,((addl(L_{1i}, \Lambda_1) \circ [G_1] \mid addl(L_{2i}, \Lambda_2) \circ [G_2])) \mid P \mid \overline{m}_i\langle\tilde{y}\rangle \mid P_1)$.
Clearly, $(R''_1, R'_2) \in \mathcal{S}$.

Put them together, by the definition, $\mathcal{S}$ is a $\approx_{kr}$. ■

This proposition indicates that, when composing two controls two together, it when every scheduler $G_{\vec{n}}$ in these two controls is a simply forwarder, then the composition effect on exclusion can be calculated or reasoned about using their $\overline{\mathcal{G}}$-terms, the describer of exclusion relations, only. In the next subsection we will see that, a scheduler $G_{\vec{n}}$ describes about synchronisation activities, may include the timing of unlock activity, after the exclusion lock (including "lock nothing") has been acquired, and therefore it will not affect the descrition of exclusion relations. In other words, the conclusion from the above proposition is still valid for even a non-simple scheduler $G_{\vec{n}}$. That is, for any situation, the composition effect on exclusion relations can be always described and reasoned about by using $\overline{\mathcal{G}}$-terms only .

However, with the "simple forwarder" version of scheduler $C_{\overline{1}}$, all the locks generated by $\overline{\mathscr{E}}$ will be un-releasable, and there is also no control over other synchronisation amongst methods.

To enable unlocking and enforce more controls over synchronisation, some signals manipulation within $C_{\overline{1}}$ is necessary.

4.2 A more sophisticated compositional object model

McHale ([McHale94]) has considered that a method invocation as a series of events, and the synchronisation performed between the events of *arrival* and *start* (Figure 4-5). But we consider that during the execution of method body, the produsing of return value and the termination of execution can be considered as two separated events and both can also be used in synchronisation control (Figure 4-6). That is, during the invocation and execution of a method, we can have four natural events: message arrival, start execution, return required value and end execution, to be used as the synchronisation points. With this idea, we designed a more sophisticated object model which uses four extra channels, *s*, s_{f}, *r* and *t*, to signalise these events respectively, and uses them to manipulate the synchronisation. As we will see soon, this model allow us keep the unlocking signal channel and other manipulation of synchronisation completely encapsulated within the control components.

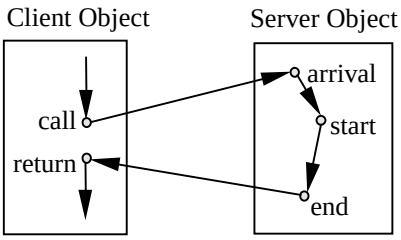


Figure 4-5
McHale model

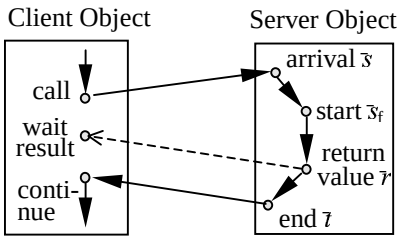


Figure 4-6
Our synchronisation modle

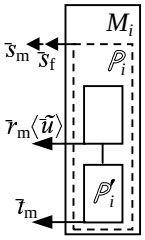


Figure 4-7
Method body

Now, by including these four synchronisation channels into the details of the variable list $\tilde{y}$ in equation 4−1, a refined version of process *F*, which represents the functionality of an object, can be expressed as:

$$F \stackrel{\text{def}}{=} (\tilde{m},\tilde{m}) \quad]\circ[\bigotimes_{i\in I} !\mathfrak{m}_i(\mathfrak{s}_{\mathrm{m}},\mathfrak{s}_{\mathrm{f}},\mathfrak{r}_{\mathrm{m}},\mathfrak{t}_{\mathrm{m}},\tilde{x})(\nu)@\varnothing.M_i\langle\tilde{m},\mathfrak{s}_{\mathrm{m}},\mathfrak{s}_{\mathrm{f}},\mathfrak{r}_{\mathrm{m}},\mathfrak{t}_{\mathrm{m}},\tilde{x}\rangle\,]$$
$$\equiv (\tilde{m},\tilde{m}) \prod_{i\in I} !\mathfrak{m}_i(\mathfrak{s}_{\mathrm{m}},\mathfrak{s}_{\mathrm{f}},\mathfrak{r}_{\mathrm{m}},\mathfrak{t}_{\mathrm{m}},\tilde{x}).M_i\langle\tilde{m},\mathfrak{s}_{\mathrm{m}},\mathfrak{s}_{\mathrm{f}},\mathfrak{r}_{\mathrm{m}},\mathfrak{t}_{\mathrm{m}},\tilde{x}\rangle. \qquad 4-9$$

Here each $\mathfrak{m}_i$ refers to a method, $\tilde{x}$ the arguaments to the method call, $\mathfrak{s}_{\mathrm{m}}$ acknowledges the receiving of the call, $\mathfrak{s}_{\mathrm{f}}$ indicates the start of method body execution, $\mathfrak{r}_{\mathrm{m}}$ is the link to the required return value, and $\mathfrak{t}_{\mathrm{m}}$ signalises the termination of method body execution. With the previously defined abbreviation $\mathfrak{t}.P \stackrel{\text{def}}{=} \mathfrak{t} \mid P$, the generic form of M_i may look like

$$M_i \stackrel{\text{def}}{=} (\tilde{m},\mathfrak{s}_{\mathrm{m}},\mathfrak{s}_{\mathrm{f}},\mathfrak{r}_{\mathrm{m}},\mathfrak{t}_{\mathrm{m}},\tilde{x})\ (\mathfrak{s}_{\mathrm{m}} \mid \mathfrak{s}_{\mathrm{f}} \mid \mathscr{P}_i\text{«}(\tilde{u})(\mathfrak{r}_{\mathrm{m}}\langle\tilde{u}\rangle \mid \mathscr{P}'_i\text{«}\mathfrak{t}_{\mathrm{m}}\text{»})\text{ » })$$
$$\equiv (\tilde{m},\mathfrak{s}_{\mathrm{m}},\mathfrak{s}_{\mathrm{f}},\mathfrak{r}_{\mathrm{m}},\mathfrak{t}_{\mathrm{m}},\tilde{x})\ \mathfrak{s}_{\mathrm{m}}.\ \mathfrak{s}_{\mathrm{f}}.\ \mathscr{P}_i\text{«}(\tilde{u})(\mathfrak{r}_{\mathrm{m}}\langle\tilde{u}\rangle \mid \mathscr{P}'_i\text{«}\mathfrak{t}_{\mathrm{m}}\text{»})\text{ »} \qquad 4-10$$

where, $\mathfrak{r}_{\mathrm{m}}\langle\tilde{u}\rangle \mid \mathscr{P}'_i$ «$\mathfrak{t}_{\mathrm{m}}$», the continuation of $\mathscr{P}_i$, indicates that the requested value $\tilde{u}$ may be obtained and immediately returned via $\mathfrak{r}_{\mathrm{m}}$ in the middle of the excution (called *early return*), and the rest of execution (represented by $\mathscr{P}'_i$) will finally ended with the termination signal $\mathfrak{t}_{\mathrm{m}}$ (see Figure 4-7).

Note in the Equation 4−10, each of signals $\mathfrak{s}_{\mathrm{m}}$, $\mathfrak{s}_{\mathrm{f}}$, $\mathfrak{r}_{\mathrm{m}}$ and $\mathfrak{t}_{\mathrm{m}}$ is restricted to be sent once only.

For the generic form of the object functionality *F* in the Equation 4−9, assume $\tilde{m}{:}\tilde{\lambda}_m$, $\tilde{x}{:}\tilde{\lambda}_x$, $\mathfrak{s}_{\mathrm{m}}{:}\lambda_s$, $\mathfrak{s}_{\mathrm{f}}{:}\lambda_f$, $\mathfrak{r}_{\mathrm{m}}{:}\lambda_r$, $\mathfrak{t}_{\mathrm{m}}{:}\lambda_t$, then $M_i{:}\mathsf{pabs}(\tilde{\lambda}_m,\lambda_s,\lambda_f,\lambda_r,\lambda_t,\tilde{\lambda}_x)$.

To present *F* in its higher-order represention of Equation 4−2, we also need to refine the empty exclusion control of Equation 4−3 to a detailed form:

$$\overline{E} \stackrel{\text{def}}{=} (\tilde{m})\text{«}\tilde{\eta}\text{»} \bigotimes_{i\in I} !\mathfrak{m}_i(\mathfrak{s}_{\mathrm{m}},\mathfrak{s}_{\mathrm{f}},\mathfrak{r}_{\mathrm{m}},\mathfrak{t}_{\mathrm{m}},\tilde{x})(\nu)@\varnothing.\eta_i\langle\mathfrak{s}_{\mathrm{m}},\mathfrak{s}_{\mathrm{f}},\mathfrak{r}_{\mathrm{m}},\mathfrak{t}_{\mathrm{m}},\tilde{x}\rangle \qquad 4-11$$

and both $\overline{E}{:}\,\mathsf{gabs}(\tilde{\lambda}_m,\tilde{\xi})$ and $F{:}\,\mathsf{pabs}(\tilde{\lambda}_m)$ will be well-typed, where $\xi_i \stackrel{\text{def}}{=} \mathsf{pabs}(\lambda_s,\lambda_f,\lambda_r,\lambda_t,\tilde{\lambda}_x)$.

For the control process $C \triangleq (\tilde{n},\tilde{m})\ \rfloor\circ(\overline{\mathscr{G}}\langle\tilde{n}\rangle \mathbin{\text{«}} G_{Ti\in I}\langle\tilde{m}\rangle \text{»})$, which has been presented in Equation 4−4 and represents the concurrent behaviours, we can refine the higher-ordered GEC choice $\overline{\mathscr{G}}$ from Equation 4−5 to

$$\overline{\mathscr{G}} \triangleq (\tilde{n})\text{«}\tilde{\eta}\text{»} \bigotimes_{i\in I} !(\nu\ \kappa)\ \natural n_i(\mathfrak{z}_n,\mathfrak{z}_f,\mathfrak{r}_n,\mathfrak{r}_n,\tilde{x})\ \check{\kappa}@J_i.\ \eta_i\langle\mathfrak{z}_n,\mathfrak{z}_f,\mathfrak{r}_n,\mathfrak{r}_n,\tilde{x},\hat{\kappa}\rangle \qquad 4\text{−}12$$

where $\tilde{n}{:}\tilde{\lambda}_m,\mathfrak{z}_n{:}\lambda_s,\mathfrak{r}_n{:}\lambda_r,\mathfrak{l}_n{:}\lambda_t,\kappa{:}^{+}\delta$ and $\overline{\mathscr{G}}{:}\,\mathsf{gabs}(\tilde{\lambda}_m,\tilde{\xi}_c)$, with $\xi_{ci}\triangleq\mathsf{pabs}(\lambda_s,\lambda_f,\lambda_r,\lambda_t,\tilde{\lambda}_x,\delta)$.

If use the simplest version of G_{Ti} shown in Equation 4−6, which can be refined as $G_{Ti}\triangleq(\mathfrak{z}_n,\mathfrak{z}_f,\mathfrak{r}_n,\mathfrak{l}_n,\tilde{x},\hat{\kappa})\,\overline{m}_i\langle\mathfrak{z}_n,\mathfrak{z}_f,\mathfrak{r}_n,\mathfrak{l}_n,\tilde{x}\rangle$, then again we have the same conclusion of Equation 4−7:

$$(\nu\ \tilde{m})\,(C\langle\tilde{n},\tilde{m}\rangle \mid F\langle\tilde{m},\tilde{n}\rangle)\ \approx_{\kappa r}\ \rfloor\circ(\overline{\mathscr{G}}\langle\tilde{n}\rangle\text{«}(\tilde{y},\hat{\kappa})M_{i\in I}\langle\tilde{n},\tilde{y}\rangle\text{»}),$$

where all the locks laid by $\overline{\mathscr{G}}$ are un-releasable, and there is no other synchronisation control amongst methods.

However, we usually do want to get control on unlocking and other synchronisation activities. This can be achieved by including manipulations of singals $\mathfrak{z},\mathfrak{r},\mathfrak{l}$ and κ in G_{Ti}. Equation 4−13 gives such an example of G_{Ti}, which describes a behaviour where early return during the execution of method body is enabled, and the exclusion lock is released only after the execution of method body is terminated.

$$\begin{aligned} G_{Ti} &\triangleq (\tilde{m},\mathfrak{z}_n,\mathfrak{z}_f,\mathfrak{r}_n,\mathfrak{l}_n,\tilde{x},\hat{\kappa})\ (\nu\ s_m,r_m,t_m)\,(\mathfrak{z}_n \mid \overline{m}_i\langle\mathfrak{z}_m,\mathfrak{z}_f,\mathfrak{r}_m,\mathfrak{l}_m,\tilde{x}\rangle \mid \natural_m.\natural r_m(\tilde{u}).(\mathfrak{r}_n\langle\tilde{u}\rangle \mid \mathfrak{l}_n \mid \natural_m.\hat{\kappa})) \\ &\equiv (\tilde{m},\mathfrak{z}_n,\mathfrak{z}_f,\mathfrak{r}_n,\mathfrak{l}_n,\tilde{x},\hat{\kappa})\ (\nu\ s_m,r_m,t_m)\ \mathfrak{z}_n.\overline{m}_i\langle\mathfrak{z}_m,\mathfrak{z}_f,\mathfrak{r}_m,\mathfrak{l}_m,\tilde{x}\rangle.\natural_m.\natural r_m(\tilde{u}).\mathfrak{r}_n\langle\tilde{u}\rangle.\mathfrak{l}_n.\natural_m.\hat{\kappa}. \end{aligned} \qquad 4\text{−}13$$

Now, if write

$$\begin{aligned} M'_i &\triangleq (\tilde{m},\mathfrak{z}_m,\mathfrak{z}_f,\mathfrak{r}_m,\mathfrak{l}_m,\tilde{x})\,(\mathfrak{z}_m \mid \mathfrak{z}_f \mid P_i\text{«}(\tilde{u})(\mathfrak{r}_m\langle\tilde{u}\rangle \mid \mathfrak{l}_m \mid P'_i\text{«}\hat{\kappa}\text{»})\text{»})\text{»}) \\ &\equiv (\tilde{m},\mathfrak{z}_m,\mathfrak{z}_f,\mathfrak{r}_m,\mathfrak{l}_m,\tilde{x})\,\mathfrak{z}_m.\mathfrak{z}_f.P_i\text{«}(\tilde{u})(\mathfrak{r}_m\langle\tilde{u}\rangle.\mathfrak{l}_m.P'_i\text{«}\hat{\kappa}\text{»})\text{»} \end{aligned} \qquad 4\text{−}14$$

then we have

$$(\nu\ \tilde{m})\,(C\langle\tilde{n},\tilde{m}\rangle \mid F\langle\tilde{m},\tilde{n}\rangle)\ \approx_{\kappa r}\ \rfloor\circ(\overline{\mathscr{G}}\langle\tilde{n}\rangle\text{«}M'_{i\in I}\langle\tilde{n}\rangle\text{»}). \qquad 4\text{−}15$$

The action diagram in Figure A2-1 illustrates the scenario of this G_{Ti} given by Equation 4−13. Different forms of G_{Ti} will result in different forms of M'_i. The action diagrams in from Figure A2-2 to Figure A2-5 shown the scenario of some other examples of G_{Ti}. The code such as `{wV;sV;sE;wE;rel;}` shown in the bottom of these diagrams, is a programmer-friendly syntax of the expression for G_{Ti}, where `wV`, `wE`, `sV`, `sE` and `rel` respectively represent the signals wait return value $\natural r_m(\tilde{u})$, wait end signal $\natural_m$, send return value $\mathfrak{r}_n\langle\tilde{u}\rangle$, send end signal $\mathfrak{l}_n$ and send release signal $\hat{\kappa}$. We will give a brief introduction about them in the Appendices A1, and present more details about their syntax and usage in an extended object-oriented programming languege in another paper.

From all these examples we can see that in this model the unlocking signal channel κ can be completely encapsulated within the control *C*, and the functional object *F* needs no knowledge about κ at all.

With this model, the structure of the control process *C* itself actually has already included the separation of some different aspects of concurrency:

1. The locking list Λ can be viewed as a thread monitor, and may be extended for access controlling or other usage;
2. The canonical higher-ordered GEC choice $\overline{\mathscr{G}}$ describes exclusion relation among the methods;
3. The tail controller $\{G_{Ti\in I}\}$ acts like a scheduler control synchronisation timing.

Corrsponding to the functional object *F* and the control *C* in this object model, a method call which waits the return value then can be modelled as :

$$(\nu\ s_n,s_f,r_n,t_n)(\ \overline{n}\langle\mathfrak{z}_n,\mathfrak{z}_f,\mathfrak{r}_n,\mathfrak{l}_n,\tilde{x}\rangle \mid \natural r_n(\tilde{v}).\natural_n.Q\ ) \qquad 4\text{−}16$$

where $\tilde{v}$ are the values returned from the method call, and *Q* is the continuation process which waiting the return value. When no return value is needed to wait, a proceduel call can be modelled as:

$$(\nu\ s_n,s_f,r_n,t_n)(\ \overline{n}\langle\mathfrak{z}_n,\mathfrak{z}_f,\mathfrak{r}_n,\mathfrak{l}_n,\tilde{x}\rangle \mid \natural_n.Q\ ) \qquad 4\text{−}17$$

The role of the channel t_n in Equation 4−16 and 4−17 is to signalise when the waiting thread *Q* can continue the execution. It is a little bit different from that of the channel t_m for the method body M_i shown in Equation 4−10 and Figure 4-7, where t_m signalise the termination of the method execution. We may say that t_n is for signal of continue.

Since in both Equation 4−16 and 4−17 the process Q has to wait the continue signal $\dagger_n$ before it can continue to execute, now the caller-side concurrent behaviour of a method call can be controlled by the control processes in the called object side. For example, with the version of G_{Ti} shown in Equation 4−13 and Figure A2-1, where the continue signal $\bar{\imath}_n$ is sent before the receiving of $\dagger_m$, the process Q is able to continue as soon as the the return value is produced, and execute concurrently with the method body without waiting for the termination of the latter. However, if we swap the order of $\dagger_m$ and $\bar{\imath}_n$ in G_{Ti}, as shown in Figure A2-6, then Q will have to wait for the termination of the method body before continue.

In both Equations 4−16 and 4−17 , the signal receotors $\mathfrak{F}_n$ and $\mathfrak{F}_f$ are not appreared since the caller is not interested in the signal $\bar{\mathfrak{F}}_n$ and $\bar{\mathfrak{F}}_f$. They may only be useful for some controller processes. Other two signal receotors, $\dagger_{r_n}$ and $\dagger_{\bar{\imath}_n}$, are only used at most once. In fact, we restrict that each polar of *s*, *r* or *t* can be used at most only once.

In all examples of G_{Ti} demonstrated in Equation 4−13 and diagrams Figure A2-1 to Figure A2-5 using only channels *m*, *s*, s_f, *r*, *t* and κ for message passing and synchronisation, no other name involved. We call such a schedule process G_{Ti} a *linear schedule process*. The number of different kinds of linear schedule processes is finite, and their composition effect will also finite, which we will study in a different paper.

Sometimes we need to use more synchronisation signals to perform even more flexible and diverse synchronisation controls. For example, we may define a finite queue behaviour, say $FinQue\langle \dagger inq \rangle$, which puts method call requests into a FIFO queue with limited length, and blocks futher requests when the queue is full. To include this behaviour *FinQue* in a control, we may construct a *nonlinear schedule process* such as

$$G_{Ti} \stackrel{\text{def}}{=} (\tilde{m},\bar{q},\bar{\mathfrak{F}}_n,\bar{\mathfrak{F}}_f,\bar{r}_n,\bar{\imath}_n,\tilde{x},\hat{\kappa})\,(\nu\, s_m,r_m,t_m,in)(\bar{q}\langle in\rangle \mid \dagger in(out).(\bar{\mathfrak{F}}_n \mid \bar{m}_i\langle \bar{\mathfrak{F}}_m,\bar{\mathfrak{F}}_f,\bar{r}_m,\bar{\imath}_m,\tilde{x}\rangle \mid \hat{\kappa} \mid \bar{\mathfrak{F}}_m.\dagger_{r_m}(\tilde{u}).\dagger_{\bar{\imath}_m}.(\bar{r}_n\langle \tilde{u}\rangle \mid \bar{\imath}_n \mid \bar{out})))$$

and then use it in the control process as

$$C \stackrel{\text{def}}{=} (\tilde{n},\tilde{m})(\nu\, q)(\ \rfloor\circ[\bar{\mathscr{G}}\langle \tilde{n}\rangle \ll G_{Ti\in I}\langle \tilde{m},\bar{q}\rangle \gg] \mid FinQue\langle \dagger q\rangle\ ).$$

Detailed studies on nonlinear schedule process will be left to the future works.

When two or more controls are composited, the compositional effect on the schedulers can be reasoned by using those schedulers theirselves only. For example, assume we have

$$(\nu\, \tilde{p})(\ \rfloor\circ[\bar{\mathscr{G}}_1\langle \tilde{n}\rangle \ll G'_{Ti\in I}\langle \tilde{p}\rangle \gg] \mid \rfloor\circ[\bar{\mathscr{G}}_2\langle \tilde{p}\rangle \ll G''_{Ti\in I}\langle \tilde{m}\rangle \gg])\ \approx_{\kappa r}\ \rfloor\circ[(\bar{\mathscr{G}}_1\langle \tilde{n}\rangle \boxplus \bar{\mathscr{G}}_2\langle \tilde{n}\rangle) \ll G_{Ti\in I}\langle \tilde{m}\rangle \gg],$$

where for some $j \in I$

$$G'_{Tj} \stackrel{\text{def}}{=} (\tilde{m},\bar{\mathfrak{F}}_n,\bar{\mathfrak{F}}_f,\bar{r}_n,\bar{\imath}_n,\tilde{x},\hat{\kappa})\,(\nu\, s_m,r_m,t_m)\,\bar{\mathfrak{F}}_n.\bar{m}_i\langle \bar{\mathfrak{F}}_m,\bar{\mathfrak{F}}_f,\bar{r}_m,\bar{\imath}_m,\tilde{x}\rangle.\bar{\mathfrak{F}}_m.\dagger_{r_m}(\tilde{u}).\hat{\kappa}.\dagger_{\bar{\imath}_m}.\bar{r}_n\langle \tilde{u}\rangle.\bar{\imath}_n$$

$$G''_{Tj} \stackrel{\text{def}}{=} (\tilde{m},\bar{\mathfrak{F}}_n,\bar{\mathfrak{F}}_f,\bar{r}_n,\bar{\imath}_n,\tilde{x},\hat{\kappa})\,(\nu\, s_m,r_m,t_m)\,\bar{\mathfrak{F}}_n.\bar{m}_i\langle \bar{\mathfrak{F}}_m,\bar{\mathfrak{F}}_f,\bar{r}_m,\bar{\imath}_m,\tilde{x}\rangle.\bar{\mathfrak{F}}_m.\dagger_{r_m}(\tilde{u})\hat{\kappa}.\bar{r}_n\langle \tilde{u}\rangle.\dagger_{\bar{\imath}_m}.\bar{\imath}_n$$

then it can be proven that $G_{Tj} \equiv G'_{Tj}$.

In the Apendices A3 we give the pseudo-code of automatical reduction of scheduler composition, as well as some example of manually compositon. Detailed studies on scheduler composition will be left to a different paper. The method used in Apendices A3 requires a modification on the κ-calculus to allow a "partial unlock signal" to release a subset of locked methods use the same key. This modification will be left to the future work on the κ-calculus.

The object model discussed in this section clearly demonstrates that the κ-calculus provides a good platform to separate different aspects in modelling concurrent objects. Those different aspects can be separated into different parts in the structure of expression, and can be reasoned about separatedly. These separations we can easily achieved include

1. Separate the functionality *F* (or method bodies) from the concurrent behaviours *C*;
2. Separate the current locking status Λ from the specification of concurrency controls, $\bar{\mathscr{G}}$ and G_{Ti}.
3. Separate the specification of exclusion policy $\bar{\mathscr{G}}$ form the specification of synchonisation timing G_{Ti};
4. Separate several concurrent behaviours into some different control processes, then compose them together.

4.3 Deadlock

In a concurrent object system, a deadlock may be regarded as some lock on some still requested service of an object becomes non-resumable unexpectedly. With our object model, let $O \stackrel{\text{def}}{=} (\tilde{m})\ \rfloor\circ\{\bar{\mathscr{G}}\langle \tilde{m}\rangle \ll \tilde{M} \gg\}$ represent the behaviour of an object in the very initial state, and an actual state of the object can be represented by given an action sequence ℓ for

which $O \xrightarrow{\ell} \Lambda\circ[\overline{\mathcal{G}}\langle\tilde{m}\rangle\,\text{«}\tilde{M}\text{»}]\,|\,P$. Assume there exist some $\kappa\in keys(\Lambda)$ and $u\in\{\tilde{m}\}$ such that $lock(\{u\},\kappa,\Lambda)$, and assume *Env* is the rest of the environment in the whole system, then the atom lock $\check{\kappa}@u\in atoms(\Lambda)$ may become not resumable in one of the followin situations:

1. $\hat{\kappa}\notin fon(P)$.

   This is normally not a deadlock, but expected effect resulted in from the designer's intention to eliminate the availability of some choices. In this case, $\overline{\mathcal{G}}$ can be presented in the form:

   $\overline{\mathcal{G}} \equiv (\tilde{m}_{i\in I})\text{«}\eta_{i\in I}\text{»}\ (!(\nu\,\kappa)\,\tilde{m}_j(\tilde{u})\ \check{\kappa}@J_j.\ \eta_j\langle\tilde{u}\rangle \boxplus \overline{\mathcal{G}}'\langle\tilde{m}_{i\in I-\{j\}}\rangle\text{«}\tilde{\eta}_{i\in I-\{j\}}\text{»}\ )$

   In other words, the key κ is abandoned immediately after the lock is fired. More general, assume

   $\overline{\mathcal{G}} \equiv (\tilde{m}_1,\tilde{m}_2)\text{«}\tilde{\eta}_1,\tilde{\eta}_2\text{»}\ (\overline{\mathcal{G}}_1\langle\tilde{m}_1\rangle\text{«}\tilde{\eta}_1\text{»} \boxplus \overline{\mathcal{G}}_2\langle\tilde{m}_2\rangle\text{«}\tilde{\eta}_2\text{»}\ )$ and $\forall c\in\{\tilde{m}_1\}.(\neg lock(\{c\},\kappa,\Lambda)) \wedge lock(\{\tilde{m}_2\},\kappa,\Lambda)$, then

   $(\tilde{n})\ \Lambda\circ[\overline{\mathcal{G}}\langle\tilde{m}_1,\tilde{m}_2\rangle\ \text{«}\tilde{M}_1,\tilde{M}_2\text{»}] \equiv (\tilde{m})\ \Lambda/@[\tilde{m}_2]\circ[\overline{\mathcal{G}}_1\langle\tilde{m}_1\rangle\ \text{«}\tilde{M}_1\text{»}]$.

2. $\hat{\kappa}\in fon(P)$ but $\hat{\kappa}\notin fon(P')$ for some $P'\equiv P$.

   This is caused by the broken of communication, that is, there exists $\mathcal{P}$ such that

   $\hat{\kappa}\notin fon(\mathcal{P}\text{«}\mathbf{0}\text{»})$ and $P \equiv \mathcal{P}\text{«}\ (\nu\ c)\,c(\tilde{v}).\mathcal{Q}\text{«}\hat{\kappa}\text{»}\ \text{»} \equiv \mathcal{P}\text{«}\mathbf{0}\text{»}$.

   In our object model the scope of $\hat{\kappa}$ is restricted not beyong the control, and the only actions may block $\hat{\kappa}$ are signals s_m, s_f, r_m and t_m. In a programming language where these actions are automatically enforced will eliminate this kind of deadlock possibilily.

3. $P \equiv \mathcal{P}\text{«}\ (\nu\ s,s_f,r,t)\ (a\langle s,s_f,r,t,\tilde{v}\rangle\,|\,h(\tilde{v}).t.\mathcal{Q}\text{«}\ \hat{\kappa}\ \text{»})\ \text{»}$

   This is a true deadlock because the unlock action $\hat{\kappa}$ is blocked by the lock itself. Howevery, similar to the previous case, this kind of deadlock will not happen in our model because the signal $\hat{\kappa}$ has been separated from method body.

4. $P\,|\,Env \equiv \mathcal{P}_1\text{«}\ \mathcal{Q}_1\text{«}\ c(\tilde{v}).\mathcal{R}_1\text{«}\hat{\kappa}\text{»}\ \text{»}\,|\,\mathcal{Q}_2\text{«}\ (\nu\ s,s_f,r,t)\ (a\langle s,s_f,r,t,\tilde{v}\rangle\,|\,h(\tilde{v}).t.\mathcal{R}_2\text{«}\ c\langle\tilde{v}\rangle\ \text{»})\ \text{»}\ \text{»}$

   This is also a self deadlock because it is the same thread, $h(\tilde{v}).t.\mathcal{R}_2\text{«}\ c\langle\tilde{v}\rangle\ \text{»}$, holds the lock and being blocked. This kind of deadlocks usually happen in OOP when self-call, a tread executing a method body of an object calls another method of the same object, or call-back, a tread accessing one object from another then accesses back from the former to the latter without leaving either of them.

   This kind of deadlock can be automatically eliminated in a programming language which lets the lock moniter detecting thread ID automatically such that no lock will locks its owner, in other words, no thread locks itself.

   The principle that "a thread should never lock itself" is acturally always used by sequencial progamming. The "sequencial progamming" is a misleading term since in fact it is equivelant to concurrent programming with single thread restriction.

   The problem of "a thread should never lock itself" is that, when a thread which is accessing some objects splits, more than one child threads may make self-call or call-back. To solve this problem, we propose a "thread ID with historical information" mechanism, that is, when a thread splitting, each of the branches becomes its child thread and is given a new ID which is not only used to distinguish individual threads, but also to remember the ID of the parent thread. When a thread tempts to access a locked object, the thread monitor of the object will check the thread ID against the owners' ID of each effected locks. A thread will not be blocked by a lock owned by itself, or one of its ancestor, but will be blocked by a lock owned by a thread from a different branch of the same family tree.

5. $P\,|\,Env \equiv P_1\,|\,P_2\,|\,\Lambda_n\circ[\overline{\mathcal{G}}_n\langle\tilde{n}\rangle\ \text{«}\tilde{N}\text{»}]$ where

   $P_1 \equiv \mathcal{P}_1\text{«}\mathcal{Q}_1\text{«}\ c_1(\tilde{v}).\mathcal{R}_1\text{«}\hat{\kappa}\text{»}\ \text{»}\,|\,\mathcal{Q}'_1\text{«}\ (\nu\ s,s_f,r,t)\ (a_1\langle s,s_f,r_1,t_1,\tilde{v}\rangle\,|\,h_1\ (\tilde{v}).t_1.\mathcal{R}'_1\text{«}\ c_1\langle\tilde{v}\rangle\ \text{»})\ \text{»}\ \text{»}$
   $P_2 \equiv \mathcal{P}_2\text{«}\mathcal{Q}_2\text{«}\ c_2(\tilde{v}).\mathcal{R}_2\text{«}\hat{\kappa}_2\text{»}\text{»}\,|\,\mathcal{Q}'_2\text{«}\ (\nu\ s,s_f,r,t)\ (a\langle s,s_f,r,t,\tilde{v}\rangle\,|\,h(\tilde{v}).t.\mathcal{R}'_2\text{«}\ c_2\langle\tilde{v}\rangle\ \text{»})\ \text{»}\ \text{»}$
   and either $u_1\in\{\tilde{m}\} \wedge lock(\{u_1\},\kappa_2,\Lambda)$
   or $u_1\in\{\tilde{n}\} \wedge lock(\{u_1\},\kappa_2,\Lambda_n)$.

   This is, two threads, $h_1\ (\tilde{v}).t_1.\mathcal{R}'_1\text{«}\ c_1\langle\tilde{v}\rangle\ \text{»}$ and $h(\tilde{v}).t.\mathcal{R}'_2\text{«}\ c_2\langle\tilde{v}\rangle\ \text{»}$, block each other.

   The continuations of $\mathcal{Q}'_1$ and $\mathcal{Q}'_2$ are describing the pure functionality of some part of methods' body, indicating that how some procedures ($\mathcal{R}'_1$ and $\mathcal{R}'_2$) should be processed depending on the results some servers ($a_1\langle s,s_f,r_1,t_1,\tilde{v}\rangle$ and

$\bar{a}\langle \bar{s}, \bar{s}_f, \bar{r}_1, \bar{t}_1, \tilde{v} \rangle$) provided. While reasonning about the functionality can be independed from the controls, here we have observed that the detecting of deadlock has to have both knowledge about functionality processes (what service is requested) and the control processes (which service to be locked, indicated by $\bar{G}_n$ and $\bar{G}$, and how the lock to be resumed, indicated within Q_1 and Q_2).

5 Theory of Composition

The previous section has presented a way to model compositional concurrent objects, but it did not give a completed model of a single object, and it left many questions unanswered: What is an object in sense of process modelling? What is an object component representing an aspect of the object behaviour? What is a control? And what are the generic properties of these components and their composition? We start our investigation on those questions in this section.

5.1 Object Processes

What is an object? In term of modelling with process algebra, an object can be considered as a process which uses a fixed set of distinguished communication input polars, represent the interface of methods, to receive incoming messages from other processes; then produces corresponging responses to those messages.

For our object model, as described previously, in order to obtain control over synchronisation, we need four control signals $\bar{s}:\bar{\lambda}_s, \bar{s}_f:\bar{\lambda}_f, \bar{r}:\bar{\lambda}_r, \bar{t}:\bar{\lambda}_t$ to be transmitted together with the message in each method call.

Definition 5−42: A link type λ is called *method type* if it can be presented in the form $lnk(\lambda) = (\bar{\lambda}_s, \bar{\lambda}_f, \bar{\lambda}_r, \bar{\lambda}_t, \tilde{\bar{\lambda}}_x)$, where $\bar{\lambda}_s, \bar{\lambda}_f, \bar{\lambda}_r$ and $\bar{\lambda}_t$ are all signal types, that $lnk(\bar{\lambda}_s) = lnk(\bar{\lambda}_f) = lnk(\bar{\lambda}_r) = lnk(\bar{\lambda}_t) = (\ )$.

In an object system, each object is uniquely identified, and method name clash between different objects should be avoided. In other words, a message sent for a given object should never be seized by another object. To enforce this restriction, we required that each process modelling an object is the owner of the input polars of all its method names. This can be guaranteed in a model where method names are always freshly defined for each object when it is created. Even for a purely class based object system where method body may be shared by objects of the same class, this can still be guaranteed, as demonstrated by the class-based object model in [Zhang97] where each object still provides its own method names to receive messages, and then pass the received message to the shared method body. However, in order to simplify our expressions, to concentrate at major issues we currently interested in, and to make our study generic, in this stage we only assume the restriction is guaranteed implicitly rather than present the full details of object/method name declarations. Latter in the paper, we will define an "autonomous context" to show how this restriction can be easily garanteed for objects.

In the practical view of Object-oriented Programming, a concurrent object may have some locked methods even in its initial state after creation. However, we may image that for each object there exists an "*abstracted initial state*", in which all methods of the object are equally accessible, none of them is locked, and any actually state of an object can be considered as derived from that abstracted initial state via a serial actions. For example, a queue object may have its "de-queue" method locked when it is freshly created, and we may consider that is the result of the executing the "constructor" method in its abstracted initial state. This assumption allows us abstract away the complicity of various states from the generic object model. Now we may consider that, in a generic definition, each object can be represented by a "definition process" which describes the behaviour of the object in that abstracted initial state, and an action sequence which indicates the path along which the object evolved from the abstracted initial state to the current state.

Definition 5−43: The process agent abstraction $P:\mathtt{pabs}(\tilde{\bar{\lambda}})$ is an *object template* of type $\mathtt{pabs}(\tilde{\bar{\lambda}})$ interface if the following conditions are all satisfied:

1) $\tilde{\bar{\lambda}}$ is a set of method type;
2) $fin(P)=\varnothing$;
3) $P\langle \tilde{m} \rangle \downarrow m$ for all $\tilde{m}:\tilde{\bar{\lambda}}$ and $m \in \tilde{m}$; and

4) $P\langle\tilde{m}\rangle\not\downarrow\alpha$ for all α which is not an input communication action.

The process $P\langle\tilde{m}\rangle$ is called an *object instantiation process* with receptor set $\tilde{m}$, if P is an object template of type $\mathsf{pabs}(\tilde{\lambda})$ interface and $\tilde{m}$ is a canonical list.

An object template can be viewed as the description, or "definition process", of the structure of an object or an object class. An object instantiation process can be viewed as the description of an object in its abstracted initial state.

Definition 5-44: Process agent abstraction $P{:}\mathsf{pabs}(\tilde{\lambda},\tilde{\lambda})$ is an *object component* of type $\mathsf{pabs}(\tilde{\lambda})$ interface, if $(\tilde{x})P\langle\tilde{x},\tilde{m}\rangle$ is an object template for all $\tilde{x}{:}\tilde{\lambda}$ and $\tilde{m}{:}\tilde{\lambda}$ where $\{\tilde{x}\}\cap\{\tilde{m}\}=\varnothing$. That is, $fn(P)=\varnothing$ and $P\langle\tilde{n},\tilde{m}\rangle\downarrow n$ for all $n\in\tilde{n}$ whenever $\tilde{n}{:}\tilde{\lambda}$, $\tilde{m}{:}\tilde{\lambda}$ and $\{\tilde{n}\}\cap\{\tilde{m}\}=\varnothing$.

The process $P\langle\tilde{n},\tilde{m}\rangle$ is called an *object component process* with *receptor set* $\tilde{n}$ and *sender set* $\tilde{m}$, if $P{:}\mathsf{pabs}(\tilde{\lambda},\tilde{\lambda})$ is an object component for which $\{\tilde{m}\}\cap fn(P)=\varnothing$ and both $\tilde{n}{:}\tilde{\lambda}$ and $\tilde{m}{:}\tilde{\lambda}$ are canonical lists.

An object component process with receptor set $\tilde{n}$ is in fact also an object instantiation process with receptor set $\tilde{n}$: From structural equivalence, the following equations are always hold,

$((\tilde{x})P\langle\tilde{x},\tilde{m}\rangle)\langle\tilde{n}\rangle\equiv P\langle\tilde{n},\tilde{m}\rangle$ and $((\tilde{x},\tilde{y})P\langle\tilde{x}\rangle)\langle\tilde{n},\tilde{m}\rangle\equiv P\langle\tilde{n}\rangle$ where $\tilde{n}{:}\tilde{\lambda}$ where $\{\tilde{x}\}\cap\{\tilde{m}\}=\varnothing$,

and then we can get the above conclusion by applying these equations to the definition of object component process and object instantiation process. The defining of object component process allows us to unify base objects and controlls.

It is easy to verify that the object's functionality F in either Equation 4-1, 4-2 or 4-9, and the control C in either Equation 4-4 are object components, as well as the compositions of them described in the previous section.

Now, we can finially define what is an object:

Definition 5-45: Process P is an *object process* with *method set* $\tilde{m}$ *respect to environment Env*, if P is either an object component process with $\tilde{m}$ as both its receptor set and sender set and $\{\tilde{m}\}\cap fn(Env)=\varnothing$, or a derivative of such an object component process.

There are a couple of important issues has been expressed in this definition: 1. An object is determined by its abstracted initial state and a derivation path (a serial actions); 2. The unification of receptor set and sender set indicates that an object uses its methods for both receiving messages making self calls; 3. an object process is safe for its environment.

Lemma 5-46: Let P be an object process with method set $\tilde{m}$ respect to environment *Env*, then for any Env' which is a derivative of *Env*, P and all its derivatives are also object processes with method set $\tilde{m}$ respect to environment Env'.

Proof: From the definition of object processes, there must exist some object component process Q with $\tilde{m}$ as both receptor set and sender set such that either $P\equiv Q$ or P is a derivative of Q, and Q is also an object process with method set $\tilde{m}$ respect to environment *Env*. By Lemma 3-30, this implies that Q is safe for all derivatives of *Env*, include Env' and all its derivatives. Therefore Q is an object process with method set $\tilde{m}$ respect to environment Env', and also P and all its derivatives are. ■

In the future, whenever discussing an object process without mention the environment, we always assume the safety constraint has been satisfied for the environment where the object process lives, that is, the object process is the owner of its receptor set. Later in this section when defining autonomous context, we will see that this assumption is always guaranted in modelling objects.

Now, the generic form of control processes can be defined.

Definition 5−47: An object component C, of type $\mathsf{pabs}(\tilde{\lambda})$ interface, is a *control abstraction* (or *control* for short) if for all canonical lists $\tilde{n}{:}\tilde{\lambda}$ and $\tilde{p}{:}\tilde{\lambda}$, to each pair of $\natural_i \in \tilde{n}$ and $p_i \in \tilde{p}$, and an input action $\alpha_1 = \natural_i(\mathfrak{s}_1, \mathfrak{s}_f, \mathfrak{r}_1, \mathfrak{t}_1, \tilde{v})$, there exist some processes C', C'', action sequence ℓ satisfying $\{\tilde{n}, \tilde{p}\} \cap fn(\ell) = \varnothing$, and an output action either $\alpha_2 = p_i\langle \mathfrak{s}_1, \mathfrak{s}_f, \mathfrak{r}_1, \mathfrak{t}_1, \tilde{v}\rangle$ or $\alpha_2 = (\nu\, s_2, r_2, t_2) p_i\langle \mathfrak{s}_2, \mathfrak{s}_f, \mathfrak{r}_2, \mathfrak{t}_2, \tilde{v}\rangle$, such that $C\langle \tilde{n}, \tilde{p}\rangle \xrightarrow{\alpha_i} C'$, $C' \xrightarrow{\ell} C''$ and $C'' \downarrow \alpha_2$.

Here, whenever the canonical lists $\tilde{n}{:}\tilde{\lambda}$ and $\tilde{p}{:}\tilde{\lambda}$ satisfy $\{\tilde{n}\} \cap \{\tilde{p}\} = \varnothing$, $C\langle \tilde{n}, \tilde{p}\rangle$ is called a *control process* with *socket* $\tilde{n}$ and *plug* $\tilde{p}$, that is, the input polar of its receptor set and the output polars of its sender set repectively.

Through socket and plug of matching types, control processes can be composed in a "plug and play" style.

A control process C acts as a message deliver, and initially is able to pass any incoming message. It is close to the "forwarder" in [Honda95] (or "wire" in [Sangiorgi96b] and "link" in [Merro98] or [Amadio96]), except the forwarding in the former can be delayed or blocked (in other word, can be controlled), and the latter, a "forwarder", can be regarded as a special case of control processes, the empty control process.

Definition 5−48: A control abstraction $E{:}\mathsf{pabs}(\tilde{\lambda}, \tilde{\lambda})$ is said to be an *empty* control, if

1. the relation $(\nu\, \tilde{m})(E\langle \tilde{n}, \tilde{m}\rangle \mid P\langle \tilde{m}\rangle) \approx_{Kr} P\langle \tilde{n}\rangle$ is always held for all object template $P{:}\mathsf{pabs}(\tilde{\lambda})$ and channel sets $\tilde{m}{:}\tilde{\lambda}$ and $\tilde{n}{:}\tilde{\lambda}$ where $\{\tilde{m}\} \cap \{\tilde{n}\} = \varnothing$, $\{\tilde{m}, \tilde{n}\} \cap fn(E) = \varnothing$ and $\{\tilde{m}\} \cap fn(P) = \varnothing$; and
2. the relations $(\nu\, \tilde{p})(E\langle \tilde{n}, \tilde{p}\rangle \mid C\langle \tilde{p}, \tilde{m}\rangle) \approx_{Kr} C\langle \tilde{n}, \tilde{m}\rangle$ and $(\nu\, \tilde{p})(C\langle \tilde{n}, \tilde{p}\rangle \mid E\langle \tilde{p}, \tilde{m}\rangle) \approx_{Kr} C\langle \tilde{n}, \tilde{m}\rangle$ are always held for all control abstraction $C{:}\mathsf{pabs}(\tilde{\lambda}, \tilde{\lambda})$ and channel sets $\tilde{m}{:}\tilde{\lambda}$, $\tilde{n}{:}\tilde{\lambda}$ and $\tilde{p}{:}\tilde{\lambda}$ where $\{\tilde{m}\} \cap \{\tilde{n}\} \cap \{\tilde{p}\} = \varnothing$ and $\{\tilde{m}, \tilde{n}, \tilde{p}\} \cap (fn(E) \cup fn(C)) = \varnothing$.

That is, an empty control will provide no behaviour change, no matter composed for left or right. An empty control may be constructed from an empty exclusion and a set of simple message resenders.

Corollary 5−49: Let $\overline{\overline{E}} \triangleq^{\text{def}} (\tilde{n}) \bigotimes_{i \in I} !(\nu\, \kappa) \natural_i(\tilde{y})\, \kappa @ \varnothing.\eta_i\langle \tilde{y}, \hat{\kappa}\rangle$ and $C_{Ei \in I} \triangleq^{\text{def}} (\tilde{m})(\tilde{y}, \hat{\kappa}) m_i\langle \tilde{y}\rangle$, where $\tilde{m}{:}\tilde{\lambda}$ and $\tilde{n}{:}\tilde{\lambda}$ are canonical channel name lists with the same set of method types, then the control

$$E \triangleq^{\text{def}} (\tilde{n}, \tilde{m})\ \rfloor \circ [\overline{\overline{E}}\langle \tilde{n}\rangle \mathbin{\ll} \tilde{C}_E\langle \tilde{m}\rangle \gg] \qquad 5\text{−}1$$

is an empty control.

Proof: Assume some canonical channel name lists $\tilde{m}{:}\tilde{\lambda}$, $\tilde{n}{:}\tilde{\lambda}$ and $\tilde{p}{:}\tilde{\lambda}$ satisfying $\{\tilde{m}\} \cap \{\tilde{n}\} \cap \{\tilde{p}\} = \varnothing$, and assume some object template $P{:}\mathsf{pabs}(\tilde{\lambda})$ and control $C{:}\mathsf{pabs}(\tilde{\lambda}, \tilde{\lambda})$ satisfying $\{\tilde{m}\} \cap fn(P) = \varnothing$ and $\{\tilde{m}, \tilde{n}, \tilde{p}\} \cap fn(C) = \varnothing$ respectively. For the expression of E, we have $\{\tilde{m}, \tilde{n}\} \cap fn(E) = \varnothing$.

Since $E\langle \tilde{n}, \tilde{p}\rangle \xrightarrow{\natural_i(\tilde{u})} E\langle \tilde{n}, \tilde{p}\rangle \mid p_i\langle \tilde{u}\rangle$ for all $i \in I$, and $(\nu\, \tilde{p})(E\langle \tilde{n}, \tilde{p}\rangle \mid p_i\langle \tilde{u}\rangle \mid Q) \equiv (\nu\, \tilde{p})[p_i\langle \tilde{u}\rangle](E\langle \tilde{n}, \tilde{p}\rangle \mid Q)$ for all Q, it is trivial to prove $(\nu\, \tilde{p})(E\langle \tilde{n}, \tilde{p}\rangle \mid P\langle \tilde{p}\rangle) \approx_{Kr} P\langle \tilde{n}\rangle$ and $(\nu\, \tilde{p})(E\langle \tilde{n}, \tilde{p}\rangle \mid C\langle \tilde{p}, \tilde{m}\rangle) \approx_{Kr} C\langle \tilde{n}, \tilde{m}\rangle$;

And since $E\langle \tilde{p}, \tilde{m}\rangle \xrightarrow{p_i(\tilde{u})} E\langle \tilde{p}, \tilde{m}\rangle \mid m_i\langle \tilde{u}\rangle$, it is even more trivial to prove $(\nu\, \tilde{p})(C\langle \tilde{n}, \tilde{p}\rangle \mid E\langle \tilde{p}, \tilde{m}\rangle) \approx_{Kr} C\langle \tilde{n}, \tilde{m}\rangle$. ■

It is easy to see, the empty control E of Equation 5−1 satisfies

$$E \equiv (\tilde{n}, \tilde{m}) \prod_{i \in I} !\natural_i(\tilde{y}).m_i\langle \tilde{y}\rangle).$$

We use Equation 5−1 because this form of presentation has the same structure as the controls in Equation 4−4, that makes it a merely sepcial case of the latter. If in the empty control expression of Equation 5−1, the data $\tilde{y}$ is unfolded to show control signals, then effect of the following two controls can be considered as the same

$$(\tilde{n}, \tilde{m})\ \rfloor \circ [\overline{\overline{E}}\langle \tilde{n}\rangle \mathbin{\ll} \tilde{C}_E\langle \tilde{m}\rangle \gg] \quad \text{and} \quad (\tilde{n}, \tilde{m})\ \rfloor \circ [\overline{\overline{E}}\langle \tilde{n}\rangle \mathbin{\ll} \tilde{C}'_E\langle \tilde{m}\rangle \gg] \qquad 5\text{−}2$$

where $\overline{\overline{E}} \triangleq^{\text{def}} (\tilde{n}) \bigotimes_{i \in I} !(\nu\, \kappa) \natural_i(\mathfrak{s}_n, \mathfrak{s}_f, \mathfrak{r}_n, \mathfrak{t}_n, \tilde{u})\, \kappa @ \varnothing.\eta_i\langle \mathfrak{s}_n, \mathfrak{s}_f, \mathfrak{r}_n, \mathfrak{t}_n, \tilde{u}, \hat{\kappa}\rangle$,

$C_{Ei} \triangleq^{\text{def}} (\tilde{m})(\mathfrak{s}_n, \mathfrak{s}_f, \mathfrak{r}_n, \mathfrak{t}_n, \tilde{x}, \hat{\kappa}) m_i\langle \mathfrak{s}_n, \mathfrak{s}_f, \mathfrak{r}_n, \mathfrak{t}_n, \tilde{x}\rangle$

$C'_{Ei} \triangleq^{\text{def}} (\tilde{m})(\mathfrak{s}_n, \mathfrak{s}_f, \mathfrak{r}_n, \mathfrak{t}_n, \tilde{x}, \hat{\kappa})(\nu\, s_m, r_m, t_m)(m_i\langle \mathfrak{s}_m, \mathfrak{s}_f, \mathfrak{r}_m, \mathfrak{t}_m, \tilde{x}\rangle \mid !\mathfrak{s}_m.\mathfrak{s}_n \mid !\mathfrak{r}_n(\tilde{y}).\mathfrak{r}_n\langle \tilde{y}\rangle \mid !\mathfrak{t}_m.\mathfrak{t}_n \mid \hat{\kappa})$.

Furthermore, notice that in a method call of our object model described by equation 4−16 or 4−17, the input polar of each signal channals s_f, s_n, r_n or t_n is used at most once in their definition scope, then the third version of empty control

$$(\tilde{n}, \tilde{m})\ \rfloor \circ [\overline{\overline{E}}\langle \tilde{n}\rangle \mathbin{\ll} \tilde{C}''_E\langle \tilde{m}\rangle \gg]$$

where $G'_{Ei} \stackrel{\text{def}}{=} (\tilde{m})(\mathfrak{z}_n, \mathfrak{z}_f, \mathfrak{r}_n, \mathfrak{t}_n, \tilde{x}, \hat{\kappa})(\nu\, s_m, r_m, t_m)(\mathfrak{m}_i\langle \mathfrak{z}_m, \mathfrak{z}_f, \mathfrak{r}_m, \mathfrak{t}_m, \tilde{x}\rangle \,|\, \mathfrak{z}_m.\mathfrak{z}_n \,|\, \mathfrak{r}_n(\tilde{y}).\mathfrak{r}_n\langle \tilde{y}\rangle \,|\, \mathfrak{t}_m.\mathfrak{t}_n \,|\, \hat{\kappa})$.

can also considered as of the same effect as the above two, while the format of G'_{Ei} allows itself to be classified as a special case of the linear schedule processes.

5.2 Composition and decomposition of object components

Sometimes we need to group two or more object components (or object templates) to form a larger object component (or object template, respectively). These including the following situations: 1. The functionality of the object can be considered as the grouping of a set of relatively independent sub-functionality's; 2. The object is a composition of two of more component objects, each of them performs a portion of the function of the whole object; 3. New components are added into a child object (or class) through the inheritance; etc.

Definition 5−50: The two object instantiation processes, P_1 with receptor set $\tilde{m}$ and P_2 with receptor set $\tilde{n}$, are *groupable* to form a new object instantiation process of receptor set $\{\tilde{m}\}\cup\{\tilde{n}\}$, iff $\{\tilde{m}\}\cap\{\tilde{n}\}=\varnothing$. We may abbreviate this composition in two ways:

$P_1 \curlywedge P_2$ indicates that the object instantiation processes P_1 and P_2 are groupable, and is also used as a synonym of the composition $P_1 \,|\, P_2$ when P_1 and P_2 are groupable.

$F_1 \curlywedge F_2$ represents the composed object template $(\tilde{m},\tilde{n})(F_1\langle\tilde{m}\rangle \curlywedge F_2\langle\tilde{n}\rangle)$ for object templates $F_1 \stackrel{\text{def}}{=} (\tilde{m})P_1$ and $F_2 \stackrel{\text{def}}{=} (\tilde{n})P_2$.

The two object component processes, D_1 with receptor set $\tilde{m}$ and sender set $\tilde{p}$ and P_2 with receptor set $\tilde{n}$ and sender set $\tilde{q}$, are groupable to form a new object component process of receptor set $\{\tilde{m}\}\cup\{\tilde{n}\}$ and sender set $\{\tilde{m}\}\cup\{\tilde{n}\}$, iff $\{\tilde{m}\}\cap\{\tilde{n}\}=\varnothing$ and $\{\tilde{p}\}\cap\{\tilde{q}\}=\varnothing$. We may abbreviate this composition in two ways:

$D_1 \curlywedge D_2$ indicates that the object component processes D_1 and D_2 are groupable, and is also used as a synonym of the composition $D_1 \,|\, D_2$ when D_1 and D_2 are groupable;

$C_1 \curlywedge C_2$ represents the composed object component $(\tilde{m},\tilde{n},\tilde{p},\tilde{q})(C_1\langle\tilde{m},\tilde{p}\rangle \curlywedge C_2\langle\tilde{n},\tilde{q}\rangle)$ for object components $C_1 \stackrel{\text{def}}{=} (\tilde{m},\tilde{p})D_1$ and $C_2 \stackrel{\text{def}}{=} (\tilde{n},\tilde{q})D_2$.

Corollary 5−51: If both C_1 and C_2 are control abstractions then $C_1 \curlywedge C_2$ is also a control abstraction.

Syntactically, $F_1 \curlywedge F_2 \not\equiv F_1 \curlywedge F_2$ and $C_1 \curlywedge C_2 \not\equiv C_2 \curlywedge C_1$, even though

$(F_1 \curlywedge F_2)\langle\tilde{n}',\tilde{n}''\rangle \equiv (F_2 \curlywedge F_1)\langle\tilde{n}'',\tilde{n}'\rangle$ and $(C_1 \curlywedge C_2)\langle\tilde{n}',\tilde{n}'',\tilde{m}',\tilde{m}''\rangle \equiv (C_2 \curlywedge C_1)\langle\tilde{n}'',\tilde{n}',\tilde{m}'',\tilde{m}'\rangle$.

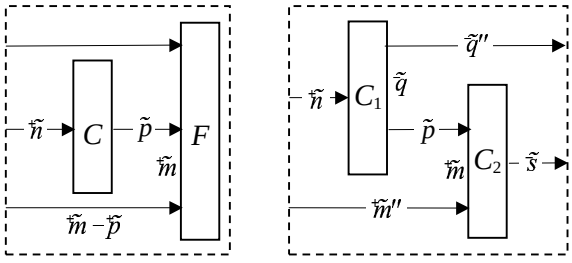
But if we assume the labelled tuples implicitly, then $F_1 \curlywedge F_2 \equiv F_1 \curlywedge F_2$ and $C_1 \curlywedge C_2 \equiv C_2 \curlywedge C_1$ can also be assumed.

A control can be composed to an object template to form a new object template (Figure 5-1a), or to another object component to form a new object component (Figure 5-1b). These can be described as.

Definition 5−52 (**Composibility**): A control process D with receptor set $\tilde{n}$ and sender set $\tilde{p}$ is *composible* to an object instantiation process P with receptor set $\tilde{m}$ to form a new object instantiation process of receptor set $\{\tilde{n}\}\cup\{\tilde{m}-\tilde{p}\}$, iff $\tilde{p}\subseteq\tilde{m}$ and $\{\tilde{m}\}\cap\{\tilde{n}\}=\varnothing$. We may abbreviate this composition in two ways:

$D \succ P$ indicates that the control process D is composible to the object instantiation process P, and is also used as a synonym of the composition $(\nu\,\tilde{p})(D \,|\, P)$ when D is composible to P.

$C \succ F$ represents some composed object template $(\tilde{m}')(C\langle\tilde{n},\tilde{p}\rangle \succ F\langle\tilde{m}\rangle)$ for control abstraction



(a) (b)

Figure 5-1

$C \stackrel{\text{def}}{\equiv} (\tilde{n},\tilde{p})D$ and object template $F \stackrel{\text{def}}{\equiv} (\tilde{m})P$, where $\tilde{m}' \stackrel{\text{def}}{\equiv} \tilde{m}\sigma$ and the input polar substitution $\sigma \stackrel{\text{def}}{\equiv} \{\tilde{n}/\tilde{p}\}$. (Figure 5-1a)

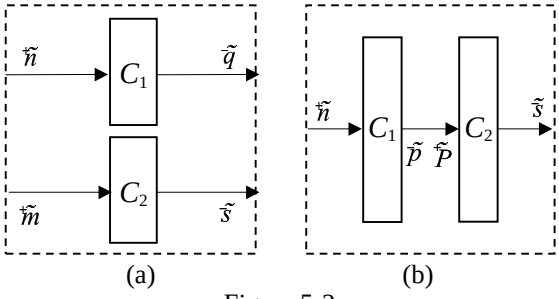
A control process D_1 with receptor set $\tilde{n}$ and sender set $\tilde{q}$ is *composible* to another object component process D_2 with receptor set $\tilde{m}$ and sender set $\tilde{s}$ to form a new object component process of receptor set $\{\tilde{n}\}\cup\{\tilde{m}-\tilde{q}\}$ and sender set $\{\tilde{q}-\tilde{m}\}\cup\{\tilde{s}\}$, iff $\{\tilde{n}\}\cap\{\tilde{m}\}=\varnothing$.We may also abbreviate this composition in two ways:

$D_1 \succ D_2$ indicates that the control process D_1 is composible to the object component process D_2, and is also used as a synonym of the composition $(\nu\,\tilde{p})(D_1 \mid D_2)$, where $\tilde{p}=\tilde{m}\cap\tilde{q}$, when D_1 is composible to D_2;

$C_1 \succ C_2$ represents some composed object component $(\tilde{n},\tilde{m}'',\tilde{q}'',\tilde{s})(C_1\langle\tilde{n},\tilde{q}\rangle \succ C_2\langle\tilde{m},\tilde{s}\rangle)$ for control abstraction $C_1 \stackrel{\text{def}}{\equiv} (\tilde{n},\tilde{q})D_1$ and object component $C_2 \stackrel{\text{def}}{\equiv} (\tilde{m},\tilde{s})D_2$, where $\tilde{m}'' \stackrel{\text{def}}{\equiv} \tilde{m}-\tilde{q}$ and $\tilde{q}' \stackrel{\text{def}}{\equiv} \tilde{q}-\tilde{m}$. (Figure 5-1b)

The composition described in Proposition-4-35 is a special case of the composition described above (and Figure 4-4 is a synonym of Figure 5-1b).

Corollary 5-53: In the above definition, $C \succ F$ is an object template; $C_1 \succ C_2$ is an object component; and if both C_1 and C_2 are control abstractions, and $\{\tilde{n}\}\cap(\{\tilde{q}\}\cup\{\tilde{s}\})=\varnothing$ and $\{\tilde{s}\}\cap(\{\tilde{m}\}\cup\{\tilde{q}\})=\varnothing$, then $C_1 \succ C_2$ is also a control abstraction.



(a) (b)

Figure 5-2

Definition 5-54: In the two special cases of the composition illustrated by Figure 5-1b, $C_1 \succ C_2$ is respectively called:

1. A *paralleled control composition*, if $\{\tilde{q}\}\cap\{\tilde{m}\}=\varnothing$ (Figure 5-2a), in this case we have $C_1 \succ C_2 \equiv C_1 \curlywedge C_2$;
2. A *serial control composition*, if $\{\tilde{q}\}=\{\tilde{m}\}$ (Figure 5-2b).

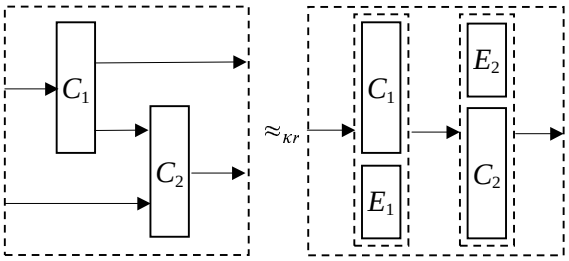


Figure 5-3

For all cases where $\{\tilde{q}\}\neq\{\tilde{m}\}$, we can always first extend C_1 and C_2 to be of the same type interface by composing them with some empty controls E_1 and E_2 respectively, and then serial compose the two result processes (Figure 5-3). In the following lemma we will prove that such a composition has the same effect as the original composition $C_1 \succ C_2$. Similarly, for the composition $C \succ F$ illustrated in Figure 5-1a, we can always first extend C to be of the same type interface as F by composing it with some empty control, and then serially composes to F. That is, we need only study the compositions between a control and an object component (or object template) of the same type interface.

Definition 5-55 : Assume an object component C of type $\mathsf{pabs}(\tilde{\lambda})$ interface and an empty control E of type $\mathsf{pabs}(\tilde{\lambda}')$ interface, then we say that:

$C_1 \stackrel{\text{def}}{\equiv} C \curlywedge E$ is an *extended object component* of C from interface of type $\mathsf{pabs}(\tilde{\lambda})$ to type $\mathsf{pabs}(\tilde{\lambda},\tilde{\lambda}')$;
$C_2 \stackrel{\text{def}}{\equiv} E \curlywedge C$ is an *extended object component* of C from interface of type $\mathsf{pabs}(\tilde{\lambda})$ to type $\mathsf{pabs}(\tilde{\lambda}',\tilde{\lambda})$.

Lemma 5-56: The composition of a control to another object component is equivalent to the composition of extended object components of them, as shown in (Figure 5-3). In other words:

Assume control C_1 of type $\mathsf{pabs}(\tilde{\lambda}')$ interface and object component C_2 of type $\mathsf{pabs}(\tilde{\lambda}'')$ interface, and the composed object component $C_1 \succ C_2$ has the type $\mathsf{pabs}(\tilde{\lambda})$ interface. Let C_1' be the extended object component of C_1 from interface of type $\mathsf{pabs}(\tilde{\lambda}')$ to type $\mathsf{pabs}(\tilde{\lambda})$, C_2' be the extended object component of C_2 from interface of

type $\mathsf{pabs}(\tilde{\lambda}'')$ to type $\mathsf{pabs}(\tilde{\lambda})$, and the serial control composition $C_1' \succ C_2'$, then for all disjoint canonical lists $\tilde{m}:\tilde{\lambda}$, $\tilde{n}:\tilde{\lambda}$, we have $(C_1 \succ C_2)\langle \tilde{n}, \tilde{m} \rangle \approx_{\kappa r} (C_1' \succ C_2')\langle \tilde{n}, \tilde{m} \rangle$.

Proof: By the definition of extended object component, there exists some empty controls E_1 and E_2 such that C_1' and C_2' are constructed by grouping them with C_1 and C_2 respectively. Without lose the generality, assume $C_1' \triangleq C_1 \curlywedge E_1$ and $C_2' \triangleq C_2 \curlywedge E_2$, let $\tilde{m}':(\tilde{\lambda}-\tilde{\lambda}'')$, $\tilde{m}'':\tilde{\lambda}''$, $\tilde{n}':\tilde{\lambda}'$, $\tilde{n}'':(\tilde{\lambda}-\tilde{\lambda}')$, $\tilde{p}':(\tilde{\lambda}-\tilde{\lambda}'')$, $\tilde{p}'':(\tilde{\lambda}' \cap \tilde{\lambda}'')$ and $\tilde{p}''':(\tilde{\lambda}-\tilde{\lambda}')$ be disjoint canonical lists satisfying $\tilde{m} \triangleq \tilde{m}' \cup \tilde{m}''$ and $\tilde{n} \triangleq \tilde{n}' \cup \tilde{n}'$, and write $\tilde{p} \triangleq \tilde{p}' \cup \tilde{p}'' \cup \tilde{p}'''$, then we have

$$((\tilde{n}',\tilde{n}'',\tilde{m}',\tilde{m}'')(\nu\, \tilde{p}'')(C_1\langle \tilde{n}',\tilde{m}',\tilde{p}' \rangle \,|\, C_2\langle \tilde{p}'',\tilde{n}'',\tilde{m}'' \rangle))\langle \tilde{n}, \tilde{m} \rangle \approx_{\kappa r}$$
$$((\tilde{n}',\tilde{n}'',\tilde{m}',\tilde{m}'')(\nu\, \tilde{p})(C_1\langle \tilde{n}',\tilde{p}',\tilde{p}'' \rangle \,|\, E_1\langle \tilde{n}'',\tilde{p}''' \rangle \,|\, E_2\langle \tilde{p}',\tilde{m}' \rangle \,|\, C_2\langle \tilde{p}'',\tilde{p}''',\tilde{m}'' \rangle))\langle \tilde{n}, \tilde{m} \rangle \quad \blacksquare$$

For for the reason of simplicity, from now on we can always assume the two parties of a composition have the same type of interface or have been extended to the same type of interface, unless expressed explicitly.

For our purpose of study, whenever given a process which models some behaviours of an object, we want to see if it can be equally represented by a composition of two or more processes, each of which separately describes a portion of these behaviours. This leads to the concept of the decomposibility of a process.

Definition 5-57 (**Decomposibility**): An object instantiation process $P\langle \tilde{n} \rangle$ with receptor set $\tilde{n}:\tilde{\lambda}$ is said to be *decomposible* if there exist an object template $F:\mathsf{pabs}(\tilde{\lambda})$, where $F \not\equiv P$, and some non-empty control $C:\mathsf{pabs}(\tilde{\lambda},\tilde{\lambda})$ such that $(C \succ F)\langle \tilde{n} \rangle \approx_r P\langle \tilde{n} \rangle$, that is, $(\nu\, \tilde{m})(C\langle \tilde{n},\tilde{m} \rangle \,|\, F\langle \tilde{m} \rangle) \approx_r P\langle \tilde{n} \rangle$.

An object component process $C\langle \tilde{n},\tilde{m} \rangle$ with receptor set $\tilde{n}:\tilde{\lambda}$ and sender set $\tilde{m}:\tilde{\lambda}$, is said to be decomposible if there exist some non-empty controls $C_1:\mathsf{pabs}(\tilde{\lambda},\tilde{\lambda})$ and object component $C':\mathsf{pabs}(\tilde{\lambda},\tilde{\lambda})$ where $C' \not\equiv C$, such that $(C_1 \succ C')\langle \tilde{n},\tilde{m} \rangle \approx_r C\langle \tilde{n},\tilde{m} \rangle$, that is, $(\nu\, \tilde{p})(C_1\langle \tilde{n},\tilde{p} \rangle \,|\, C'\langle \tilde{p},\tilde{m} \rangle) \approx_r C\langle \tilde{n},\tilde{m} \rangle$.

An object template $P:\mathsf{pabs}(\tilde{\lambda})$, is decomposible if $P\langle \tilde{n} \rangle$ is decomposible for all $\tilde{n}:\tilde{\lambda}$.
An object component $C:\mathsf{pabs}(\tilde{\lambda},\tilde{\lambda})$ is decomposible if $C\langle \tilde{n},\tilde{m} \rangle$ is decomposible for all $\tilde{n}:\tilde{\lambda}$ and $\tilde{m}:\tilde{\lambda}$.

5.3 Properties of composition

As pointed out by [Zhang02B] and [Zhang02D], responsive bisimulation is congruence for autonomous processes. Let process P be an object component process or object instantiation process with receptor set $\tilde{n}$, it can be guaranteed to be a safe process in the environment where it resides in, but is not an autonomous process. However, what it is used for is to model the initial behaviour of object component, and follow the modelling method in [Walker95] and [Zhang97], the creation of an object with P as the initial behaviour can be modeled as:

$$!new(l_0).(\nu\, a)(l_0\langle a \rangle \,|\, (\nu\, \tilde{n})(!a(l).l\langle a \rangle \,|\, P)$$

and each created object $(\nu\, a)(l_0\langle a \rangle \,|\, (\nu\, \tilde{n})(!a(l).l\langle \tilde{n} \rangle \,|\, P)$ is clearly an autonomous process.

Definition 5-59 : A process context of the form $\mathcal{C}[.] \triangleq (\nu\, a)(l_0\langle a \rangle \,|\, (\nu\, \tilde{n})(!a(l).l\langle \tilde{n} \rangle \,|\, [.]))$ is called an *autonomous context* for processes with receptor set $\tilde{n}$. And since for processes and satisfying

$\mathcal{C}[P_1] \approx_r \mathcal{C}[P_2]$ (or $\mathcal{C}[P_1] \approx_{\kappa r} \mathcal{C}[P_2]$) iff $a \notin (fin(P_1) \cup fin(P_2))$ and $P_1 \approx_r P_2$ (or $P_1 \approx_{\kappa r} P_2$, respectively);
both $\mathcal{C}[P_1]$ and $\mathcal{C}[P_1]$ are an autonomous processes if $(fin(P_1) \cup fin(P_2)) \subseteq \tilde{n}$;
$\approx_r$ and $\approx_{\kappa r}$ are congruencces for autonomous processes,

we say that:

1. $\approx_r$ and $\approx_{\kappa r}$ are congruences under autonomous context;
2. the processes P_1 and P_2 are congruent under autonomous context, denoted as $P_1 \equiv_r P_2$ (or $P_1 \equiv_{\kappa r} P_2$) if $P_1 \approx_r P_2$ (or $P_1 \approx_{\kappa r} P_2$, respectively);
3. the process abstractions $A_1 \triangleq (\tilde{n},\tilde{m})P_1$ and $A_2 \triangleq (\tilde{n},\tilde{m})P_2$ are congruent under autonomous context, denoted as $A_1 \equiv_r A_2$ (or $A_1 \equiv_{\kappa r} A_2$) if $P_1 \approx_r P_2$ (or $P_1 \approx_{\kappa r} P_2$, respectively) and $(fin(P_1) \cup fin(P_2)) \subseteq \tilde{n}$.

Now lets give some properties of composition.

Propersition 5−60 (**Identity Law**): For each object template F of type $\mathsf{pabs}(\tilde{\tilde{\lambda}})$ interface, or object component C of type $\mathsf{pabs}(\tilde{\tilde{\lambda}})$ interface, there exists some empty control E of the same type interface such that for all disjoint canonical lists $\tilde{m}{:}\tilde{\tilde{\lambda}}$, $\tilde{n}{:}\tilde{\tilde{\lambda}}$ and $\tilde{p}{:}\tilde{\tilde{\lambda}}$ the following relation are held:

$(E\langle\tilde{n},\tilde{p}\rangle \succ F\langle\tilde{p}\rangle) \approx_{Kr} F\langle\tilde{n}\rangle$, $(E\langle\tilde{n},\tilde{p}\rangle \succ C\langle\tilde{p},\tilde{m}\rangle) \approx_{Kr} C\langle\tilde{n},\tilde{m}\rangle$ and $(C\langle\tilde{n},\tilde{p}\rangle \succ E\langle\tilde{p},\tilde{m}\rangle) \approx_{Kr} C\langle\tilde{n},\tilde{m}\rangle$,

or simply written as $E \succ F \equiv_{Kr} F$ and $E \succ C \equiv_{Kr} C \succ E \equiv_{Kr} C$.

Proof: An empty control of type $\mathsf{pabs}(\tilde{\tilde{\lambda}})$ interface can always be constructed in the form of quation 5−1. ■

With the identity law, we may describe the concurrency behaviour composition in a more portable style, or "*plug-and-play*" style: We need not require the sender set of a control process D to be directly the same name set of the receptor set of a object component process A, but only require them have the same type $\tilde{\tilde{\lambda}}$, and use an empty control of type $\mathsf{pabs}(\tilde{\tilde{\lambda}})$ interface to establish a connection between the two sets of names (see Figure 5-4).

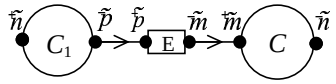


Figure 5-4

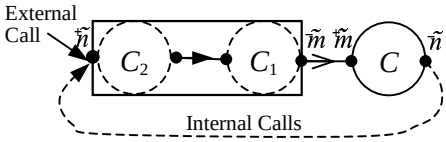


Figure 5-5

Propersition 5−61 (**Association Law**): Assume control abstractions C_1 and C_2, and object template F or object component C are all of type $\mathsf{pabs}(\tilde{\tilde{\lambda}})$ interface, then for all disjoint canonical lists $\tilde{m}{:}\tilde{\tilde{\lambda}}$, $\tilde{n}{:}\tilde{\tilde{\lambda}}$, $\tilde{p}{:}\tilde{\tilde{\lambda}}$ and $\tilde{q}{:}\tilde{\tilde{\lambda}}$ the following relations are held:

$C_2\langle\tilde{n},\tilde{p}\rangle \succ (C_1\langle\tilde{p},\tilde{m}\rangle \succ F\langle\tilde{m}\rangle) \approx_{Kr} (C_2\langle\tilde{n},\tilde{p}\rangle \succ C_1\langle\tilde{p},\tilde{m}\rangle) \succ F\langle\tilde{m}\rangle$ and

$C_2\langle\tilde{n},\tilde{p}\rangle \succ (C_1\langle\tilde{p},\tilde{q}\rangle \succ C\langle\tilde{q},\tilde{m}\rangle) \approx_{Kr} (C_2\langle\tilde{n},\tilde{p}\rangle \succ C_1\langle\tilde{p},\tilde{q}\rangle) \succ C\langle\tilde{q},\tilde{m}\rangle$,

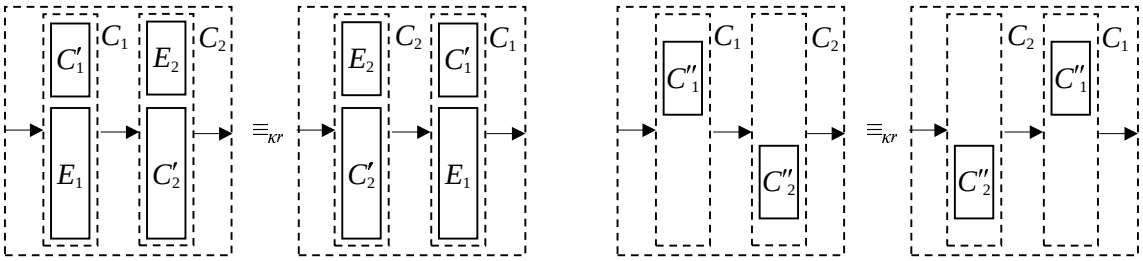
which may simply be written as: $C_2 \succ (C_1 \succ F) \equiv_{Kr} (C_2 \succ C_1) \succ F$ and $C_2 \succ (C_1 \succ C) \equiv_{Kr} (C_2 \succ C_1) \succ C$ respectively.

Proof: Simply by the structural equivalence, we have

$(\nu\ \tilde{p})(C_2\langle\tilde{n},\tilde{p}\rangle \mid (\nu\ \tilde{m})(C_1\langle\tilde{p},\tilde{m}\rangle \mid F\langle\tilde{m}\rangle)) \equiv (\nu\ \tilde{m})((\nu\ \tilde{p})(C_2\langle\tilde{n},\tilde{p}\rangle \mid C_1\langle\tilde{p},\tilde{m}\rangle) \mid F\langle\tilde{m}\rangle)$, and

$(\nu\ \tilde{p})(C_2\langle\tilde{n},\tilde{p}\rangle \mid (\nu\ \tilde{q})(C_1\langle\tilde{p},\tilde{q}\rangle \mid C\langle\tilde{q},\tilde{m}\rangle)) \equiv (\nu\ \tilde{q})((\nu\ \tilde{p})(C_2\langle\tilde{n},\tilde{p}\rangle \mid C_1\langle\tilde{p},\tilde{q}\rangle) \mid C\langle\tilde{q},\tilde{m}\rangle)$. ■

The existence of association law enables us to combine the concurrence constraints with each other first, then add the combined constraint to the functional behaviour (see Figure 5-5).



(a) (b)

Figure 5-6

The *commutativity*, $(C_1 \succ C_2) \equiv_{Kr} (C_2 \succ C_1)$ or $(C_1 \succ C_2) \equiv_r (C_2 \succ C_1)$, does not hold in general for the composition between two control abstractions C_1 and C_2. In other words, in general the order of adding concurrence constraints to an object *does* matter. However, the commutativity between $C_1 \succ C_2$ and $C_2 \succ C_1$ is certainly held in cases (Figure 5-6a) where there exist some control abstractions, C_1' of type $\mathsf{pabs}(\tilde{\tilde{\lambda}}')$ interface and C_2' of type $\mathsf{pabs}(\tilde{\tilde{\lambda}}'')$ interface, such that C_1 is an extended object component of C_1' from interface of type $\mathsf{pabs}(\tilde{\tilde{\lambda}}')$ to type $\mathsf{pabs}(\tilde{\tilde{\lambda}}', \tilde{\tilde{\lambda}}'')$, and C_2 is an extended object component of C_2' from interface of type $\mathsf{pabs}(\tilde{\tilde{\lambda}}'')$ to type $\mathsf{pabs}(\tilde{\tilde{\lambda}}', \tilde{\tilde{\lambda}}'')$. Since it is possible that C_1' and/or C_2' themselves is an extended object components of some other control abstraction, commutativity in a little bit more generical situation as Figure 5-6b can be in sight. What are the other situations the commutativity exists, and what are rules to determine that, is an interesting topic in the future works.

The *distribution* property also exists in some restricted cases, as described in the following corollary:

Corollary 5−62: Let C_1 and C_2 be some control abstractions, F_1 and F_2 be some object templates and C_1' and C_2' be some object components such that C_1, F_1 and C_1' have the same interface type $\mathsf{pabs}(\tilde{\lambda}')$, and C_2, F_2 and C_2' have the same interface type $\mathsf{pabs}(\tilde{\lambda}'')$, if $C_1 \succ C_2$ is a paralleled control composition, then:

1. $(C_1 \succ F_1) \curlywedge (C_2 \succ F_2) \equiv_r (C_1 \curlywedge C_2) \succ (F_1 \curlywedge F_2)$ whenever both $C_1 \succ F_1$ and $C_2 \succ F_2$ exist;
2. $(C_1 \succ C_1') \curlywedge (C_2 \succ C_2') \equiv_r (C_1 \curlywedge C_2) \succ (C_1' \curlywedge C_2')$ whenever both $C_1 \succ C_1'$ and $C_2 \succ C_2'$ are serial control compositions.

Proof: Assume disjoint canonical lists let $\tilde{m}':\tilde{\lambda}'$, $\tilde{m}'':\tilde{\lambda}''$, $\tilde{n}':\tilde{\lambda}'$, $\tilde{n}'':\tilde{\lambda}''$, $\tilde{p}':\tilde{\lambda}'$ and $\tilde{p}'':\tilde{\lambda}''$,

$$(\nu\ \tilde{m}')(C_1\langle \tilde{n}', \tilde{m}'\rangle \mid F_1\langle \tilde{m}'\rangle) \mid (\nu\ \tilde{m}'')(C_2\langle \tilde{n}'', \tilde{m}''\rangle \mid F_2\langle \tilde{m}''\rangle) \equiv (\nu\ \tilde{m}', \tilde{m}'')(C_1\langle \tilde{n}', \tilde{m}'\rangle \mid C_2\langle \tilde{n}'', \tilde{m}''\rangle \mid F_1\langle \tilde{m}'\rangle \mid F_2\langle \tilde{m}''\rangle),$$

$$(\nu\ \tilde{p}')(C_1\langle \tilde{n}', \tilde{p}'\rangle \mid C_1'\langle \tilde{p}', \tilde{m}'\rangle) \mid (\nu\ \tilde{p}'')(C_2\langle \tilde{n}'', \tilde{p}''\rangle \mid C_2'\langle \tilde{p}'', \tilde{m}''\rangle) \equiv (\nu\ \tilde{p}', \tilde{p}'')(C_1\langle \tilde{n}', \tilde{p}'\rangle \mid C_2\langle \tilde{n}'', \tilde{p}''\rangle \mid C_1'\langle \tilde{p}', \tilde{m}'\rangle \mid C_2'\langle \tilde{p}'', \tilde{m}''\rangle). \blacksquare$$

The significant of the distribution property of composition is, for a composed object which has some component objects to perform a portion of the functions of the whole object, the control constraints can be either put on each component objects, or put the container object (Figure 5-7).

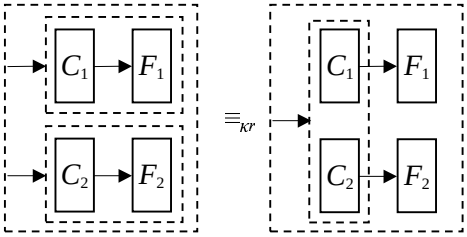


Figure 5-7

6 Conclusion

In this paper presents the theory of composition for concurrenct object systems, based on the object modelling in the κ-calculus. These include a compositional concurrent object model, and a generic theory of object behaviour composition. We have shown that with the κ-calculus, the compositional concurrent object model we proposed allows a natural separation of different aspects in currency, such as locking states, exclusion, synchronisation scheduling, etc., and allows those aspects to be reasoned about separately in certain degree. Based on this model, an extension to existing Object-Oriented programming language will be developed, which will provide a better description on behaviour composition, and also allow certain degree of automatical or manual reasoning on composition. In the study of theory of composition we have identified what is an object, a component or composible behaviour of an object, while abstracted away the details of abject model. Based on that, we have studied the properties of object behaviour composition, such as the proving of the identity law and associative law, and the analysing of commutativity and distribution of the composition. This theory will allow us study the compositional concurrent object in deep in the future, including to characterise when a behaviour can be further decomposed into atomic behaviours.

References:

[Aksit92] Mehmet Aksit and Lodewijk Bergmans "Obstacles in Object-oriented Software Development", *OOPSLA '92 Conference Proceedings*, volume 27 of ACM SIGPLAN Notices, pages 341-358, New York, October 1992

[Amadio96] Roberto M. Amadio, Ilaria Castellani and Dacide Sangiorgi, "On Bisimulations for the Asynchronous π-calculus", in *Proceedings of CONCUR'96*, LNCS volume 1119, Springer Verlag, 1996

[Amadio97] Roberto M. Amadio, "An Asynchronous Model of Locality, Failure, and Process Mobility", In D. Garlan and D. Le Metayer, editor, *Proceedings of The Second International. Conference on Coordination Models and Languages (COORDINATION'97)*, LNCS 1282, Springer, 1997

[Bos89] J. van den Bos and C. Laffra, "PROCOL A Parallel Object Language with Protocols, " in Proceedings of the 1989 OOPSLA Conference, New Orleans, Louisiana, September 1989.

[Bos91] Jan van den Bos, Chris Laffra: "PROCOL: A Concurrent Object-Oriented Language with Protocols Delegation and Constraints. " Acta Informatica 28(6): 511-538 (1991)

[Busi95] Nadia Busi and Roberto Gorrieri, "Distributed Conflicts in Communicating Systems", in Christine Mingins, Roger Duke and Bertrand Meyer, editors, *Object-Based Models and Languages for Concurrent Systems*, LNCS vol 924, pages 49-65, Springer-Verlag, 1995. URL: ftp://ftp.cs.unibo.it/pub/techreports/94-08.ps.gz

[Crno98] Lobel Crnogorac, Anand S. Rao, and Kotagiri Ramamohanarao, "Classifying Inheritance Mechanisms I concurrent Object-Oriented Programming," in Eric Jul, editor, *Proceedings of ECOOP'98*, volume 1445 of *Lecture Notes in computer Science*, pages 571-600. Springer Verlag, 1998.

[Holmes97] David Holmes, James Noble, John Potter, "Aspects of Synchronisation", in Christine Mingins, Roger Duke and Bertrand Meyer, editors, *Technology of Object-Oriented Languages and Systems TOOLS 25 - Proceedings of The 25th International Conference TOOLS* (TOOLS Pacific'97), pages 7-18, Melbourne, Australia, November 1997

[Honda91] Kohei Honda and Mario Tokoro, "An Object Calculus for Asynchronous Communication", in P. America, editor, *ECOOP'91*, LNCS vol 512, pages 133-147, Springer-Verlag, 1991.

[Honda92] Kohei Honda and Mario Tokoro, "On Asynchronous Communication Semantics", in M. Tokoro, O. Nierstrasz, and P. Wegner, editors, *Object-Based Concurrent Computing 1991*, LNCS vol 612, pages 21-51, Springer-Verlag, 1992.

[Honda95] Kohei Honda and Mario Tokoro, "On Reduction-based Process Semantics", *Theoretical Computer Science*, 152(2):437-486, 1995.

[Hüttel96] Hans Hüttel and Josva Kleist, "*Objects as mobile processes*", Aalborg University, August 1996. URL: http://www.cs.auc.dk/~kleist/ObjMobile

[Jalloul94] Ghinwa Jalloul, "*Concurrent Object-Oriented Systems: A Disciplined Approach*", PhD Dissertation, University of Technology, Sydney, Australia, June 1994

[Jones93] Cliff B. Jones, "A π-calculus Semantics for an Object-based Design Notation", in E. Best, editor, *Proceedings of CONCUR'93*, volume 715 of *Lecture Notes in computer Science*, pages 158-172. Springer Verlag, 1993

[Laff92] C. Laffra, "PROCOL: a Concurrent Object Language with Protocols, Delegation, Persistence and Constraints, " Ph.D. thesis, Erasmus Universiteit, Rotterdam, the Netherlands, May 1992.

[Liu97] Xinxin Liu and David Walker, "Concurrent Objects as Mobile Processes", to be appeared in G. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*, MIT Press.

[Matsuoka93] Satoshi Matsuoka, "Language Features for Reuse and Extensibility in Concurrent Object-Oriented Programming", PhD thesis, Department of Information Science, University of Tokyo, Japan, April 1993

[McHale94] Ciaran McHale, "Synchronisation in Concurrent, Object-oriented Languages: Expressive Power, Genericity and Inheritance", PhD. Thesis, Department of Computer Science, Trinity college, University of Dublin, Ireland, October 1994. URL: ftp://ftp.dsg.cs.tcd.ie/pub/doc/dsg_86.ps.gz

[Milner92] Robin Milner, Joachim Parrow, David Walker, "A Calculus of Mobile Process" (Parts I and II), *Journal of Information and Computation*, 100:1-77, September 1992. URL: http://www.dcs.ed.ac.uk/lfcsreps/EXPORT/89

[Milner92b] Robin Milner and Davide Sangiorgi, "Barbed Bisimulation", in W. Kuich, editor, *Proceeding of 19th ICALP*, volune 623 of *Lecture Notes in computer Science*, Springer Verlag, 1992

[Milner96] Robin Milner, "*The π-calculus*", hand-written tutorial. Computer Science Tripos, Cambridge University 1996

[Merro98] Massimo Merro and Davide Sangiorgi, "On Asynchrony in Name-passing calculi", In 25th ICALP, volune 1442 of *Lecture Notes in computer Science*, pages ??. Springer Verlag, 1998

[Merro00] Massimo Merro, "Locality and Polyadicity in Asynchronous Name-passing Calculi", In *Proceedings of FOSSACS 2000*, Berlin, Germany, volume 1784, pages 238-251, Lecture Notes in Computer Science, Springer Verlag, 2000

[Nestmann96] Uwe Nestmann and Benjamin C. Pierce, "Decoding Choice Encodings", *Journal of Information & Computation*, 163: 1-59, November 2000. URL: http://www.brics.dk/RS/99/42

[Noble00] James Noble and John Potter, "Exclusion for Composite Objects", In *Proceedings of OOPSLA 2000*, Minneapolis, Minnesota USA, ACM press, 2000

[Odersky95a] Martin Odersky, "Polarized Name Passing", in Proceedings of 15th Foundations of Software Technology and Theoretical Computer Science (FST&TCS'95), Bangalore, India, December 18-20, 1995. URL: http://lampwww.epfl.ch/~odersky/papers

[Odersky95c] Odersky, M. "Polarized bisimulation", In *Proceedings of Workshop on Logic, Domains, and Programming Languages*, Darmstadt, Germany, 1995

[Philippou96] Anna Philippou and David Walker, "On Transformations of Concurrent-Object Programs", *Theoretical Computer Sciences*, to appear. Extended abstract in Proceedings of CONCUR'96, papers 131-146, Springer 1996

[Philippou97] Anna Philippou and David Walker, "*A Process-Calculus Analysis of Concurrent Operations on B-Trees*", Technical report, University of Warwick, UK, 1997

[Pierce93] Benjamin C. Pierce and Davide Sangiorgi, "Typing and Subtyping for Mobile Processes", In *Proceedings of 8th Symposium on Logic in Computer Science*, pages 409-454, IEEE Computer society Press, 1993. URL: http://www.inria.fr/meije/personnel /Davide.Sangiorgi/mypapers.html

[Pierce95] Benjamin C. Pierce, David N. Turner, "Concurrent Objects in a Process Calculus", In Takayasy Ito and Akinori Yonezawa, editors, *Theory and Practice of Parallel Programming (TPPP)*, LNCS 907, pages 187-215. Springer, April 1995. URL: http://www.cis.upenn.edu/~bcpierce/papers

[Pierce96] Benjamin C. Pierce, David N. Turner, "PICT: A Programming Language Based on the π-calculus". URL: http://www.cis.upenn.edu/~bcpierce/papers

[Ravara97] António Ravara and Vasco T. Vasconcelos, "Behavioural types for a calculus of concurrent objects". In C. Lengauer, M. Griebl, and S. Gorlatch, editors, *Proceddings of 3rd International Euro-Par Conference*, LNCS 1300, pages 554--561. Springer-Verlag, 1997

[Sangiorgi92a] David Sangiorgi, "From π-calculus to Higher-Order π-calculus, and Back", In *Proceedings of TAPSOFT'93*., LNCS 668, Springer Verlag, 1992. URL: http://www-sop.inria.fr/mimosa/personnel/Davide.Sangiorgi/mypapers.html

[Sangiorgi92b] David Sangiorgi, "*Expressing Mobility in Process Algebras: First-Oreder and Higher-Order paradigms*", PhD thesis, Computer Science Department, University of Edinburgh, UK, 1992. Available from URL: http://www-sop.inria.fr/mimosa/personnel/Davide.Sangiorgi/mypapers.html

[Sangiorgi95] David Sangiorgi, "*Lazy functions and mobile processes*", INRIA Technical Report RR-2515, August 1996. URL: http://www-sop.inria.fr/mimosa/personnel/Davide.Sangiorgi/mypapers.html

[Sangiorgi96] David Sangiorgi, "*An Interpretation of Typed Objects into Typed π-calculus*", INRIA Technical Report RR-3000, August 1996. URL: http://www-sop.inria.fr/mimosa/personnel/Davide.Sangiorgi/mypapers.html

[Sangiorgi96b] David Sangiorgi, "Locality and Non-interleaving Semanitics in Calculi for Mobiule Processes", *Theoretical Computer Science*, 155:39-83, 1996

[Sangiorgi97] David Sangiorgi, "The Name Discipline of Uniform Receptiveness", In 24th ICALP, volune 1256 of *Lecture Notes in computer Science*, pages ??. Springer Verlag, 1997

[Schneider97] Jean-guy Schneider and Markus Lumpe, "Synchronizing Concurrent Objects in the ", Proceedings of Langages et Modèles à Objets '97, Roland Ducournau and Serge Garlatti (Ed.), Hermes, Roscoff, October 1997, pp. 61-76. URL: ftp://ftp.iam.unibe.ch /pub/scg/Papers/lmo97.ps.gz

[Walker95] David Walker, "Objects in the π-Calculus", *Information and Computation*, 116(2): 253-271 (1995)

[Zhang97] Xiaogang Zhang and John Potter, "Class-based models in π-calculus", in Christine Mingins, Roger Duke and Bertrand Meyer, editors, *Technology of Object-Oriented Languages and Systems, TOOLS 25* (TOOLS Pacific'97), Melbourne, Australia, 24th-27th November 1997, pages 238-251, IEEE Computing Society Press, 1998. URL: ftp://ftp.mpce.mq.edu.au/pub/mri/people/xzhang/papers/class97.ps.gz

[Zhang98A] Xiaogang Zhang and John Potter, "*Compositional Concurrency Constraints for Object Models in π-calculus*", Technical Report C/TR-9804, Macquarie University, Sydney, Australia, 1998. URL: ftp://ftp.mpce.mq.edu.au/pub/mri/people/xzhang/papers/TR98-04.doc

[Zhang98B] Xiaogang Zhang and John Potter, "A Compostion Approach to Concurrent Objects", in Jian Chen, Mingshu Li, Christine Mingins and Bertrand Meyer, editors, *Technology of Object-Oriented Languages and Systems, TOOLS 27* (TOOLS Asia'98), Beijing, China, 22nd-25th September 1998, pages 116-126, IEEE Computing Society Press, 1998. URL: ftp://ftp.mpce.mq.edu.au/pub/mri/people/xzhang/papers/tools27.ps.gz

[Zhang02A] Xiaogang Zhang and John Potter, "Responsive Bisimulation", in Ricardo Baeza-Yates, Ugo Montanari and Nicola Santoro, editors, *Foundations of Information Technology in the Era of Network and Mobile Computing* (IFIP-TCS 2002), Montréal, Québec, Canada, August 25th -30th 2002, page 6001-612, Kluwer Academic Publishers.

[Zhang02B] Xiaogang Zhang and John Potter, "*The Responsive Bisimulations in the polar π-calculus*", Technical report UNSW-CSE-TR-0203.

[Zhang02C] Xiaogang Zhang and John Potter, "*A Constraint Description Calculus for Compositional Concurrent Objects*", Technical report UNSW-CSE-TR-0204.

[Zhang02D] Xiaogang Zhang and John Potter, "*On Responsive Bisimulations in the κ-calculus*", Technical report UNSW-CSE-TR-0205.

APPENDICES

A1 The syntax of control actions

To make the description of a scheduler control to be understandable to ordinary programmers, we add new keywords

```
<actnKWd>  ::= wV| sV | wA | sA | wS| sS|  wE | sE | rel
```

to an extended Object-Oriented programming language. They are corresponding to the symbols $\overline{r}_m(\tilde{u})$, $r_n\langle\tilde{u}\rangle$, $\overline{s}_m$, $\overline{s}_n$, $\overline{s}_f$, $\overline{s}_f$, $\overline{t}_m$, $\overline{t}_n$ and $\hat{\kappa}$, respectively, in the Equation 4−13. The syntax can described as

```
<ActBlock> ::= {<Acts>}
<Acts>     ::= <Act>; [<Acts>]                                  - (a list of actions)
             | <ActBlock>|| <ActBlock> [|| <ActBlock>];         - (forced parallel actions)
<Act>      ::= <BAct>| <NBAct>                                  - (blocking actions, no-blocking actions)
<BAct>     ::= wV;<Acts>| wA|wS| wE                             - (wait value, arrive, start and end signals)
<NBAct>    ::= <SAct>| rel | rel(<MLst>)                       - (sending actions, releaseing signal)
<SAct>     ::= sV| sA|sS| sE                                    - (send value, arrive, start and end signals)
<MLst>     ::= MethodName;[<MLst>]                              - (a list of method names)
```

With restriction:

Each of **wV**, **wA**, **wS**, **wE**, **sV**, **sA**, **sS**, **sE**, **rel** can only appear once within any <Act*Block*>

For the statement **wV;**<Acts>, the signal **sV** must appear in the <Acts> which follows **wV**.

A2 Action diagram examples for scheduler

The effect of a scheduler control can also be explainted with interaction diagram.

In an interaction diagram, we use verticle lines to represent threads. A deshed section of a line represents a blocked duration of the thread, and the solid sections of the line represents the non-blocked durations. A verticle rectangle, which can be regarded as a very thick line section, represents that the thread is doing some processing. A triangle on a thread is a small duration on the thread of scheduler, in which a couple of control signals being communicated. A harizontal arrow repesent a control signal being sent from one thread to another.

According to Equation 4−16, a method call with continuation, $v = n(u); Q$, can be modelled as a "send and wait" expression $(\nu\, s_n, s_f, r_n, t_n)(\, \overline{n}\langle \overline{s}_n, \overline{s}_f, r_n, t_n, \tilde{x}\rangle \,|\, \overline{r}_n(\tilde{v}).\overline{t}_n.Q\,)$. That is, the continuation Q can continue only after the scheduler has sent both $\overline{r}_n(\tilde{v})$ and $\overline{t}_n$ singals. In the interaction diagram, the caller thread always has a deshed section between the event of sending method call message , and the point where both $\overline{r}_n(\tilde{v})$ and $\overline{t}_n$ signals have received.

To explaint the scenario of the control process, lets look at Figure A2-1, where we have a linear schedule process $\overline{r}_m(\overline{v}).(r_n\langle\overline{v}\rangle \,|\, \overline{t}_n \,|\, \overline{t}_m.\hat{\kappa})$. Because the roles of the arriving signal s_n and the start signal s_f are not significant in a linear schedule process, we do not show them.

In the Figure A2-1, the method call signal $\overline{n}\langle\tilde{u}\rangle$ is received by the method *n*, which is not locked in this moment, a control thread is invoked for method *n*, and the lock *L* is triggered to lock specified methods, which may (and may not) included the *n* method itself. In the same time, the method call message is forwarded to method body *m*, and invoked a new execution thread for *m*.

Once the return value $\overline{r}_m(\overline{v})$, produced by the method body *m*, arrives the scheduler thread, the scheduler forward it to the caller, and also sends a "continue" signal back to the caller to allow the later to continue. Once the method body is terminated in execution and sends the termination signal $\overline{t}_m$ to the scheduler, the scheduler release the lock to allow the locked methods to be accessable.

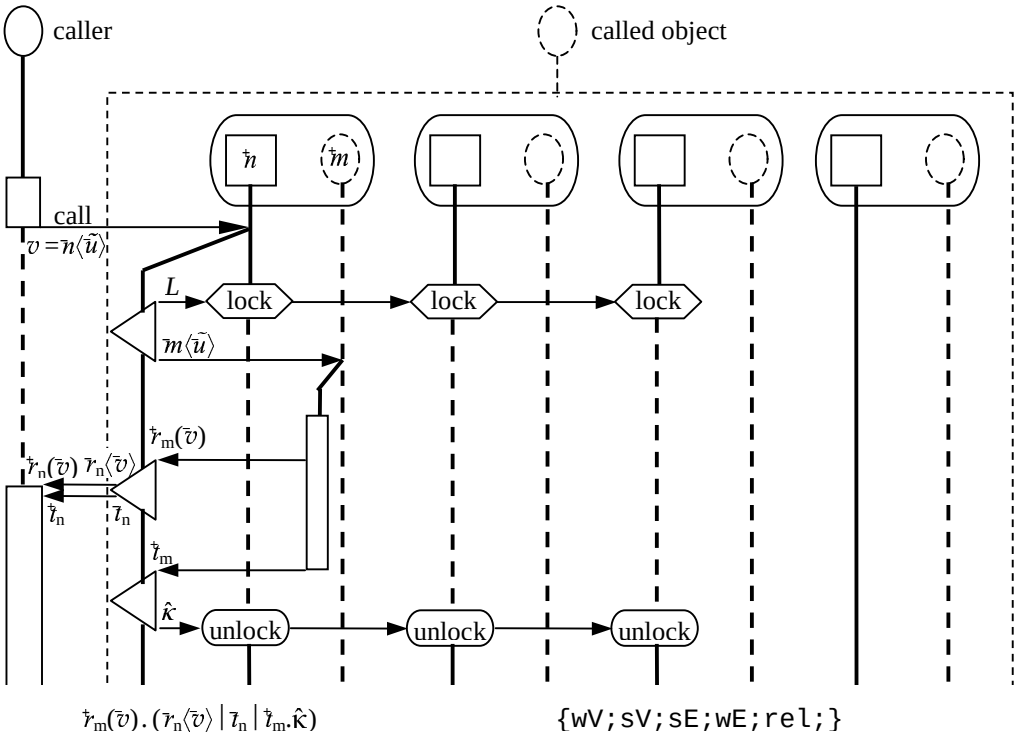


Figure A2-1 Scenario of Equation 4−13: Early return late unlock

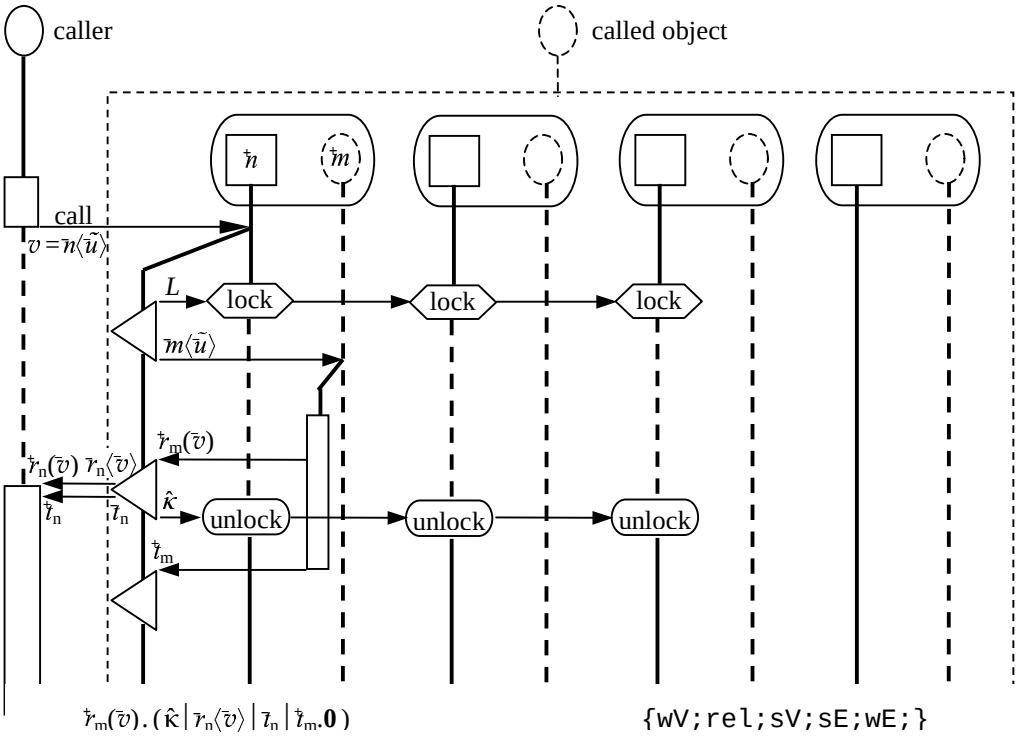


Figure A2-2 Early return early unlock

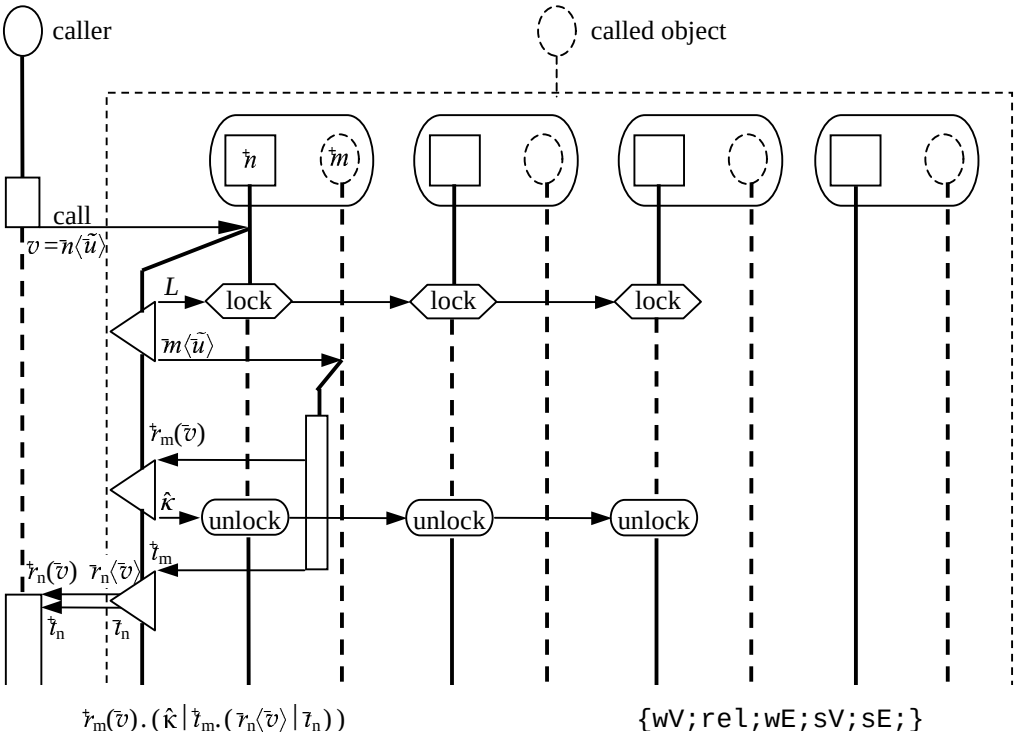


Figure A2-3 Late return early unlock

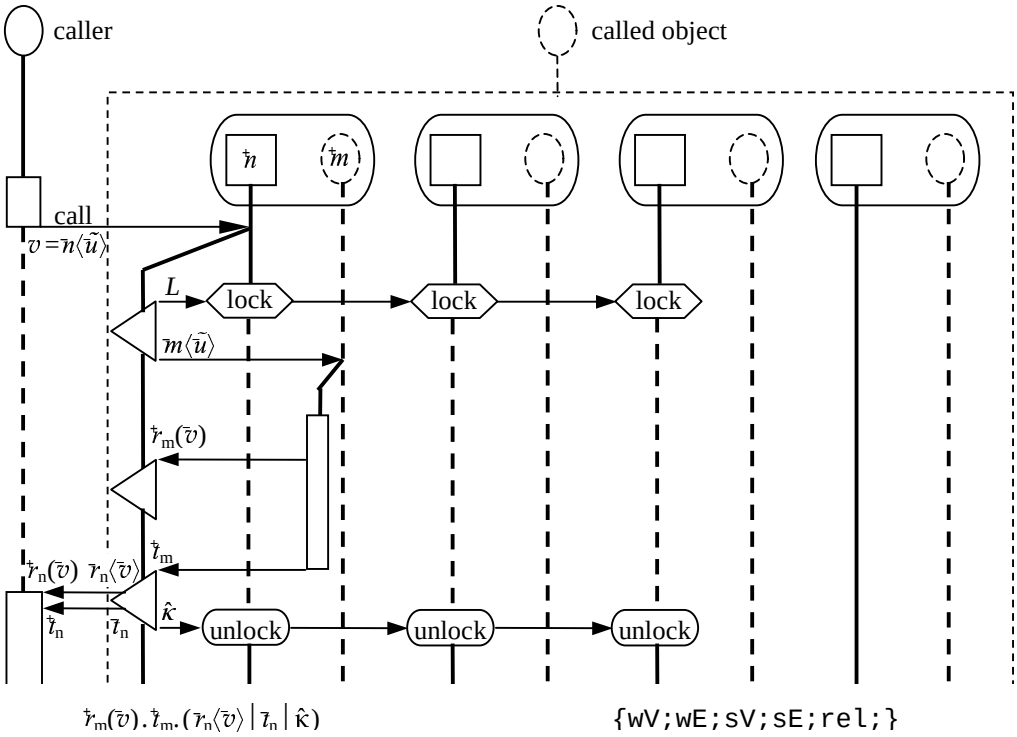


Figure A2-4 Late return late unlock

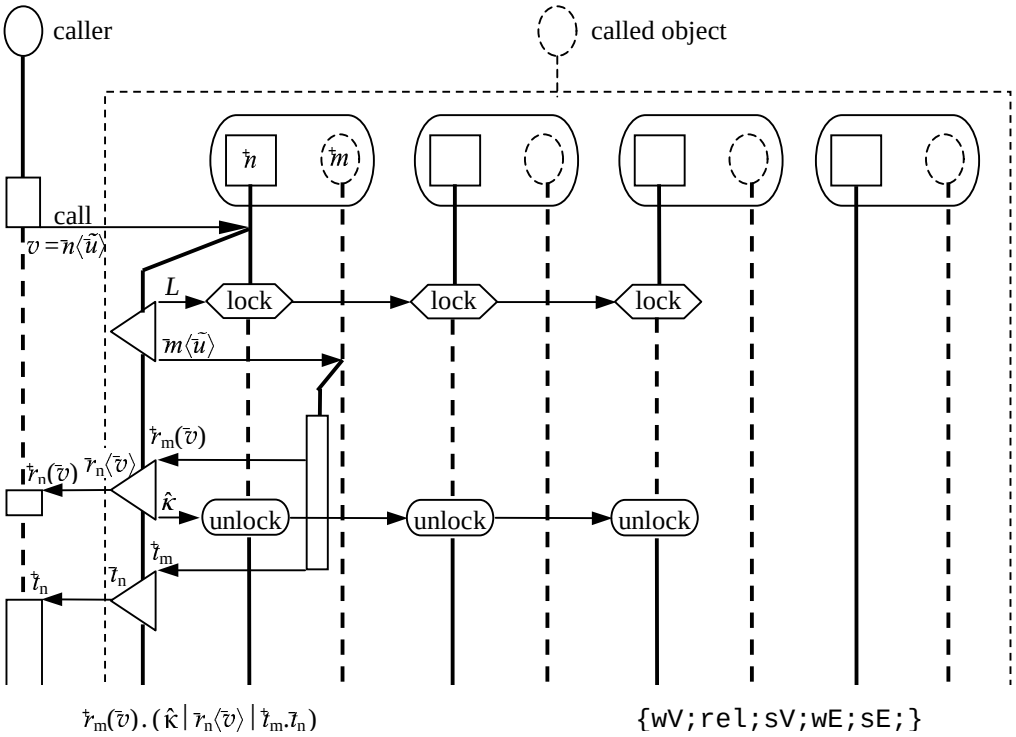


Figure A2-5 Another version of late return early unlock as Figure A2-3

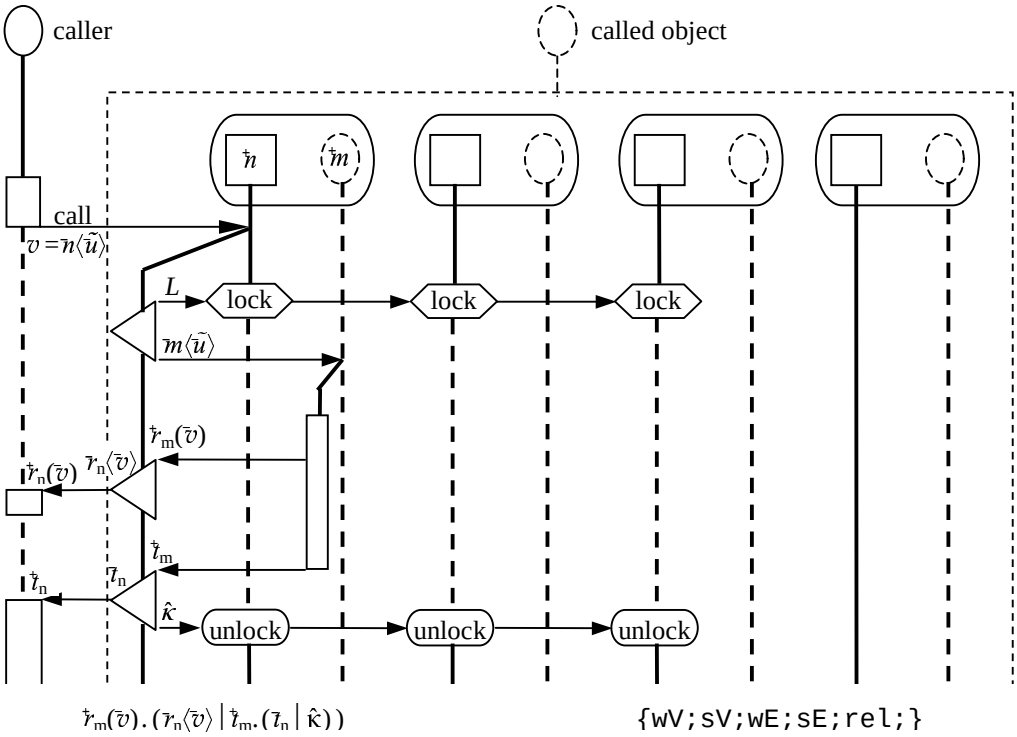


Figure A2-6 Another version of late return late unlock as Figure A2-4

A3 Scheduler composition

When two or more controls composed together, their effect may be equally represented by a single control. It can be necessary for software verification and system optimisation to find out the equivalent single control. Since in our object model the scheduler control is described separately, the composition effect on them can also be reasoned about separately. There are procedures for such an inference, both for inference automatically and manually.

The following pseudo-code is the procedure for automatically inference on scheduler composition, and the call

```
Outer_process( OuterScheduler, InnerScheduler, ResultScheduler );
```

will generate `ResultScheduler`, the equivalent single scheduler of the composition of the two schedulers, `OuterScheduler` and `InnerScheduler`.

```
Outer_process( CurrentOuterActions, CurrentInnerActions, Result ) {
   if( CurrentOuterActions is the EndOfBlock ) {
      terminate;
   } else if( CurrentOuterActions is a NonParalBlocks ) {
      for ( each sub_block of CurrentOuterActions ) {
         Outer_process( firstAct_of_the_sub_block, CurrentInnerActions, Result );
      }
      if( CurrentOuterActions becomes an empty_block ) {
         Outer_process( NextOuterAction, CurrentInnerActions, Result );
      } else {
         Inner_process( CurrentInnerActions, CurrentOuterActions, Result );
   } else if( CurrentInnerActions is a NBAct ) {
      move the CurrentOuterActions to Result;
      advance the Result point;
      Outer_process(NextOuterAction, CurrentInnerActions, Result);
   } else {      // CurrentOuterActions is a Bact //
      Inner_process( CurrentInnerActions, CurrentOuterActions, Result );
}


Inner_process ( CurrentInnerActions, CurrentOuterActions, Result ) {
   if( CurrentInnerActions is the EndOfBlock ) {
      terminate;
   } else if( CurrentInnerActions is a NonParalBlock ) {
      create a NonParalBlock in the Result;
      for( each sub_inner_block of CurrentInnerActions ) {
         create a SubResulBlock in the Result;
         Inner_process( FirstActionOfTheSubInnerBlock, CurrentInnerActions, SubResulBlock );
      }
      if( CurrentOuterActions becomes an empty_block ) {
         Outer_process( NextOuterAction, CurrentInnerActions; Result );
      } else {
         Inner_process( CurrentInnerActions, CurrentOuterActions, Result );
      }
   }  else if( CurrentInnerActions is a Sact ) {
      erase the corresponding Bact in CurrentOuterActions;
      Inner_process( NextInnerAction, CurrentOuterActions, Result );
   } else {     // if(CurrentInnerActions is not a SAct) //
      move the CurrentInnerActions to Result;
      advance the Result point;
      Inner_process (NextInnerAction, CurrentOuterActions, Result);
   }
}
```

The following table is an example of scheduler composition, by both automatically inference and manually inference.

Inner layer	Outter layer		Result	
14’ $\bar{r}_p(v).(r_m\langle v\rangle \mid \bar{l}_p.\hat{\kappa}_m \mid \bar{l}_m)$ {wV;sV;sE;wE;rel;}	1	$\hat{\kappa}_n \mid \bar{r}_m(v).\bar{l}_m.(r_n\langle v\rangle \mid \bar{l}_n)$ {wV;wE;sV;sE;}	14’	$\hat{\kappa}_n \mid \bar{r}_p(v).(r_n\langle v\rangle \mid \bar{l}_p.\hat{\kappa}_m \mid \bar{l}_n)$ {wV;sV;sE;wE;rel(mlst$_1$);}
	2	$\hat{\kappa}_n \mid \bar{r}_m(v).\bar{l}_m.r_n\langle v\rangle \mid \bar{l}_n$ {sE;wV;wE;sV;}	17’	$\hat{\kappa}_n \mid \bar{r}_p(v).(r_n\langle v\rangle \mid \bar{l}_p.\hat{\kappa}_m) \mid \bar{l}_n$ {sE;wV;sV;wE;rel(mlst$_1$);}
	3	$\hat{\kappa}_n \mid \bar{r}_m(v).r_n\langle v\rangle \mid \bar{l}_m.\bar{l}_n$ {{wV;sV;}||{wE;sE;};}	14’	{wV;sV;sE;wE;rel(mlst$_1$);}
	4	$\hat{\kappa}_n \mid \bar{r}_m(v).(r_n\langle v\rangle \mid \bar{l}_m.\bar{l}_n)$ {wV;sV;wE;sE;}	14’	{wV;sV;sE;wE;rel(mlst$_1$);}
	5	$\hat{\kappa}_n \mid \bar{r}_m(v).r_n\langle v\rangle \mid \bar{l}_m.\mathbf{0} \mid \bar{l}_n$ {sE;wV;sV;}	17’	{sE;wV;sV;wE;rel(mlst$_1$);}
	5’	$\hat{\kappa}_n \mid \bar{r}_m(v).(r_n\langle v\rangle \mid \bar{l}_m.\mathbf{0}) \mid \bar{l}_n$ {sE;wV;sV;}	17’	{sE;wV;sV;wE;rel(mlst$_1$);}
	6	$\hat{\kappa}_n \mid \bar{r}_m(v).(r_n\langle v\rangle \mid \bar{l}_n) \mid \bar{l}_m.\mathbf{0}$ {wV;sV;sE;}	14’	{wV;sV;sE;wE;rel(mlst$_1$);}
	6’	$\hat{\kappa}_n \mid \bar{r}_m(v).(r_n\langle v\rangle \mid \bar{l}_m.\mathbf{0} \mid \bar{l}_n)$ {wV;sV;sE;}	14’	{wV;sV;sE;wE;rel(mlst$_1$);}
	7	$\hat{\kappa}_n \mid \bar{r}_m(v).(\bar{l}_n \mid \bar{l}_m.r_n\langle v\rangle)$ {wV;sE;wE;sV;}	14’	{wV;sV;sE;wE;rel(mlst$_1$);}
	8	$\bar{r}_m(v).(\hat{\kappa}_n \mid \bar{l}_m.(r_n\langle v\rangle \mid \bar{l}_n))$ {wV;rel;wE;sV;sE;}	11’/14’	$\bar{r}_p(v).(\hat{\kappa}_n \mid r_n\langle v\rangle \mid \bar{l}_n \mid \bar{l}_p.\hat{\kappa}_m)$ {wV;rel(mlst$_2$);sV;sE;wE;rel(mlst$_1$);}
	8’	$\bar{r}_m(v).(\hat{\kappa}_n \mid \bar{l}_m.r_n\langle v\rangle) \mid \bar{l}_n$ {sE;wV;rel;wE;sV;}	19’/17’	$\bar{r}_p(v).(\hat{\kappa}_n \mid r_n\langle v\rangle \mid \bar{l}_p.\hat{\kappa}_m) \mid \bar{l}_n$ {sE;wV;rel(mlst$_2$);sV;wE;rel(mlst$_1$);}
	9	$\bar{r}_m(v).(\hat{\kappa}_n \mid \bar{l}_m.r_n\langle v\rangle \mid \bar{l}_n)$ {wV;rel;sE;wE;sV;}	11’/14’	{wV;rel(mlst$_2$);sV;sE;wE;rel(mlst$_1$);}
	10	$\bar{r}_m(v).(\hat{\kappa}_n \mid r_n\langle v\rangle \mid \bar{l}_m.\bar{l}_n)$ {wV;rel;sV;wE;sE;}	11’/14’	{wV;rel(mlst$_2$);sV;sE;wE;rel(mlst$_1$);}
	11	$\bar{r}_m(v).(\hat{\kappa}_n \mid r_n\langle v\rangle \mid \bar{l}_n) \mid \bar{l}_m.\mathbf{0}$ {wV;rel;sV;sE;}	11’/14’	{wV;rel(mlst$_2$);sV;sE;wE;rel(mlst$_1$);}
	11’	$\bar{r}_m(v).(\hat{\kappa}_n \mid r_n\langle v\rangle \mid \bar{l}_m.\mathbf{0} \mid \bar{l}_n)$ {wV;rel;sV;sE;}	11’/14’	{wV;rel(mlst$_2$);sV;sE;wE;rel(mlst$_1$);}
	12	$\bar{r}_m(v).\bar{l}_m.(\hat{\kappa}_n \mid r_n\langle v\rangle \mid \bar{l}_n)$ {wV;wE;rel;sV;sE;}	11’/14’	{wV;rel(mlst$_2$);sV;sE;wE;rel(mlst$_1$);}
	12’	$\bar{r}_m(v).\bar{l}_m.(\hat{\kappa}_n \mid r_n\langle v\rangle) \mid \bar{l}_n$ {sE;wV;wE;rel;sV;}	19’/17’	{sE;wV;rel(mlst$_2$);sV;wE;rel(mlst$_1$);}
	13	$\bar{r}_m(v).(r_n\langle v\rangle \mid \bar{l}_m.(\hat{\kappa}_n \mid \bar{l}_n))$ {wV;sV;wE;rel;sE;}	11’/14’	{wV;rel(mlst$_2$);sV;sE;wE;rel(mlst$_1$);}
	14	$\bar{r}_m(v).(r_n\langle v\rangle \mid \bar{l}_n) \mid \bar{l}_m.\hat{\kappa}_n$ {{wV;sV;sE;}||{wE;rel;};}	11’/14’	{wV;rel(mlst$_2$);sV;sE;wE;rel(mlst$_1$);}
	14’	$\bar{r}_m(v).(r_n\langle v\rangle \mid \bar{l}_m.\hat{\kappa}_n \mid \bar{l}_n)$ {wV;sV;sE;wE;rel;}	11’/14’	{wV;rel(mlst$_2$);sV;sE;wE;rel(mlst$_1$);}
	15	$\bar{r}_m(v).(\bar{l}_n \mid \bar{l}_m.(\hat{\kappa}_n \mid r_n\langle v\rangle))$ {wV;sE;wE;rel;sV;}	11’/14’	{wV;rel(mlst$_2$);sV;sE;wE;rel(mlst$_1$);}
	16	$\bar{r}_m(v).r_n\langle v\rangle \mid \bar{l}_m.(\hat{\kappa}_n \mid \bar{l}_n)$ {{wV;sV;}||{wE;rel;sE;};}	11’/14’	{wV;rel(mlst$_2$);sV;sE;wE;rel(mlst$_1$);}
	17	$\bar{r}_m(v).r_n\langle v\rangle \mid \bar{l}_m.\hat{\kappa}_n \mid \bar{l}_n$ {sE;{wV;sV;}||{wE;rel};}	19’/17’	{sE;wV;rel(mlst$_2$);sV;wE;rel(mlst$_1$);}
	17’	$\bar{r}_m(v).(r_n\langle v\rangle \mid \bar{l}_m.\hat{\kappa}_n) \mid \bar{l}_n$ {sE;wV;sV;wE;rel;}	19’/17’	{sE;wV;rel(mlst$_2$);sV;wE;rel(mlst$_1$);}
	18	$\bar{r}_m(v).(\hat{\kappa}_n \mid r_n\langle v\rangle) \mid \bar{l}_m.\bar{l}_n$ {{wV;rel;sV;}||{wE;sE;};}	11’/14’	{wV;rel(mlst$_2$);sV;sE;wE;rel(mlst$_1$);}
	19	$\bar{r}_m(v).(\hat{\kappa}_n \mid r_n\langle v\rangle) \mid \bar{l}_m.\mathbf{0} \mid \bar{l}_n$ {sE;wV;rel;sV;}	19’/17’	{sE;wV;rel(mlst$_2$);sV;wE;rel(mlst$_1$);}
	19’	$\bar{r}_m(v).(\hat{\kappa}_n \mid r_n\langle v\rangle \mid \bar{l}_m.\mathbf{0}) \mid \bar{l}_n$ {sE;wV;rel;sV;}	19’/17’	{sE;wV;rel(mlst$_2$);sV;wE;rel(mlst$_1$);}

Note: $mlst_1$ is the list methods locked by the Inner control, and $mlst_2$ is that locked by the Outter control.