Synchronizing Concurrent Objects in the π-Calculus [1]

Jean-Guy Schneider — Markus Lumpe

Software Composition Group
Institut für Informatik (IAM), Universität Bern
Neubrückstrasse 10, CH-3012 Bern, Switzerland
{*lumpe,schneidr*}*@iam.unibe.ch*
http://www.iam.unibe.ch/∼scg

RÉSUMÉ. *Le développement des langages orientés objets concurrents a souffert de l'absence d'un support formel fédérateur dédié à la définition de leur sémantique. C'est une des raisons pour lesquelles nous essayons de trouver une fondation sémantique minimale pour définir les abstractions des langages orientés objets. Nous avons montré précédemment l'intérêt du π-calcul à cet égard en proposant la définition d'un modèle à objets contenant des abstractions communes aux langages orientés objets. Nous nous proposons maintenant de définir un cadre formel de type boîte noire pour la modélisation objet. Nous présentons ici une première extension de notre modèle à objets fondé sur le π-calcul permettant l'intégration des abstractions pour la synchronisation des objets concurrents. Nos résultats montrent d'une part, que les objets sont synchronisés plus aisément si les schémas de synchronisation sont réifiés comme des entités de première classe (des méta-objets) et d'autre part, que le concept de "schéma générique de synchronisation" de McHale forme une base prometteuse pour la définition d'abstractions de synchronisation réutilisables et de plus haut niveau.*

ABSTRACT. *The development of concurrent object-based programming languages has suffered from the lack of any generally accepted formal foundation for defining their semantics. Therefore we are seeking for a minimal semantic foundation for defining features of concurrent object-based languages. Our previous work has shown that the π-calculus is a promising formal foundation for modelling objects, and we have defined an object model integrating common features of object-oriented programming languages. Our goal is to define a black-box framework for modelling objects. As a first extension of our π-calculus based object model, we present in this work the integration of abstractions for synchronizing concurrent objects. Our results show that objects are most easily synchronized when synchronization policies are reified as first class entities (i.e. metaobjects) and that McHale's concept of "generic synchronization policies" forms a promising base for the definition of higher-level, reusable synchronization abstractions.*

1. In *Proceedings of Languages et Modèles à Objets*, Roland Ducournau and Serge Garlatti (Eds.), Hermes, Roscoff, October 1997, pp. 61–76.

MOTS-CLÉS : *π-calcul, modèles á objets, synchronisation, réflexion.*
KEYWORDS: *π-calculus, object models, synchronization, reflection.*

1. Introduction

The development of concurrent object-based programming languages has suffered from the lack of any generally accepted formal foundation for defining their semantics, although several formal models have been presented (see [Men94] for a summary). We are, therefore, seeking for a minimal semantic foundation for defining features of concurrent object-based languages which can also be used for modelling components [NSL96]. It is our goal to define a black-box framework for experimenting with different object and component models.

The π-calculus is a calculus in which the topology of communication can evolve dynamically during evaluation [Mil89]. It has been successfully used to model objects [BS95, Jon93] and simple object-oriented programming languages [Wal95]. Our previous work has shown that the π-calculus is a promising formal foundation for modelling objects [SL96]. It was, therefore, a natural step to extend our existing π-calculus based object model with further abstractions. In our previous work, we mainly concentrated on modelling features of object-oriented programming languages that do not necessarily address concurrency [LSN96]. It is obvious, however, that the presence of concurrent activities within an object requires some degree of synchronization. As a first extension to our object model, we were, therefore, investigating abstractions for synchronizing concurrent objects.

Several synchronization schemes have been proposed to address various levels of concurrency control [Bri96]. Centralized schemes, such as *path expressions* or *bodies*, specify in an abstract way the possible interleavings of method invocations [Ame87, VdBL89]. Decentralized schemes, such as *guards*, are based on boolean activation conditions that may be associated to each method [DLDR$^+$91]. Higher level formalisms are based on the notion of *abstract behaviours* [TV89]. Recent work has tried to integrate these synchronization schemes into a framework for classifying, comparing, customizing, and combining synchronization abstractions for object-oriented concurrent programming [Bri96].

After evaluating several synchronization schemes and mechanisms, we found out that objects are most easily synchronized through metaobjects. This implies that synchronization policies have to be reified as first class entities. Our experiments showed that McHale's concept of "generic synchronization policies" [McH94] could be easily integrated into the metalevel of our existing object model, and that it formed a promising base for the definition of higher-level, reusable synchronization abstractions. Generic synchronization policies do not only allow a complete separation of computational and synchronization abstractions, but enhance the reuse of existing (sequential) components in a

concurrent environment. The extension of our object model also allows us to formalize the concept of generic synchronization policies.

This report is organized as follows: in section 2 we introduce or π-calculus object model followed by a description of McHale's "generic synchronization policies" in section 3. In section 4, we present the integration of the "generic synchronization policies" into our π-calculus based object model. Section 5 summarizes the main observations and results of our integration. We conclude with some remarks about future work and directions.

2. Objects in the π-calculus

In this section, we will briefly introduce our π-calculus based object model. For further details, refer either to [LSN96] or [SL96].

We have used PICT [PT97], an experimental programming language based on the polyadic mini π-calculus [San95], as an executable specification language for our modellings. We will first informally present the polyadic mini π-calculus, which is a simplified polyadic π-calculus [Mil91]. The polyadic mini π-calculus is built from the operators of inaction, input prefix, output, parallel composition, restriction, and replication. Small letters $a, b, ..., x, y, ...$ range over the infinite set of names, and $P, Q, R, ...$ over the set of processes:

$$P ::= \mathbf{0} \mid a(\tilde{x}).P \mid \overline{a}\langle\tilde{x}\rangle \mid P_1|P_2 \mid (\upsilon\ a)P \mid !P$$

$\mathbf{0}$ is the inactive process. An input-prefixed process $a(\tilde{x}).P$, where $\tilde{x}$ has pairwise distinct components, waits for a tuple of names $\tilde{y}$ to be sent along a and then behaves like $P\{_{\tilde{x}}\backslash^{\tilde{y}}\}$, where $\{_{\tilde{x}}\backslash^{\tilde{y}}\}$ is the simultaneous substitution of names $\tilde{x}$ with names $\tilde{y}$. An output $\overline{a}\langle\tilde{x}\rangle$ emits names $\tilde{x}$ at a. Parallel composition runs two processes in parallel. The restriction $(\upsilon\ a)P$ makes name a local to P[2]. A replication $!P$ stands for a countable infinite number of copies of P in parallel. We use the special name _ as wildcard symbol. Values bound to this name are unimportant for the following process and will be ignored.

Throughout the rest of this report, we will use the following notation to denote process abstractions:

$$ProcessAbstraction(global\text{-}channel\text{-}arguments) \equiv P$$

Our object model is based on the basic object model introduced by Pierce and Turner [PT95]. Using a π-calculus notation, the modelling of a *reference cell* object can be illustrated as follows[3]:

$$\begin{aligned} RefCell\ (init) \ \equiv\ & (\upsilon\ contents)(\ \overline{contents}\ \langle init\rangle \\ & \mid\ get(res).contents(val).(\overline{contents}\ \langle val\rangle \mid \overline{res}\ \langle val\rangle) \\ & \mid\ set(val).contents(\_).\overline{contents}\ \langle val\rangle\ ) \end{aligned}$$

2. A local channel name can be communicated to other processes. This mechanism is called scope extrusion.

3. A reference cell is an updatable data structure. In order to simplify the notation, we have used a channel based instead of an record based reference cell.

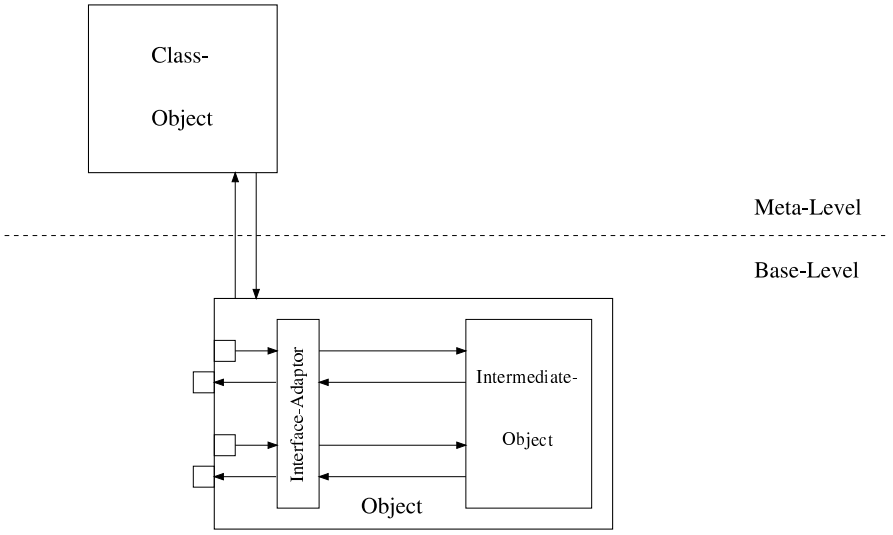

Figure 1. *Object model in the π-calculus.*

This object model integrates the essentials of concurrent objects (e.g., instance variables and methods), but does not capture other common features of object-oriented programming languages such as self-references, dynamic binding, class variables, and inheritance. In order to integrate those features, we extended the object model by introducing a metalevel (see figure 1).

A base-level object consists of three parts: an *intermediate object*, an *interface adaptor*, and a set of *request and reply channels*. All request and reply channels represent the service interface of an object. In order to call a particular method, a message representing the actual input parameters has to be sent along the corresponding request channel. The result of this method invocation can be obtained by reading the appropriate reply channel. In figure 1, request channels are represented by a small square inside and reply channels by a square outside an object, respectively. The solid lines with arrowheads denote channels used for one-way communication within an object and between an object and its class object, and are not accessible by external clients. The intermediate object consists of a set of local channels representing instance variables and a number of process abstractions representing all method implementations. The interface adaptor maps each request and reply channel to the process abstraction representing the corresponding method implementation. One may note that there is no limit on the number of concurrently executing methods in an object and that methods are not explicitly synchronized.

In order to reuse (inherit) method implementations, but having a correct binding of self-references, intermediate objects still have an unbound self-

reference[4]. The actual binding of self is achieved in the interface adaptor by passing the value of self as a first parameter to each method invocation.

A commonly used mechanism in various object-oriented programming languages is to represent classes as objects in a metalevel [KdRB91]. We have used this technique not only to model class features as features of a metaobject, but also to obtain a disciplined way to create objects and to model inheritance. In order to create an object, the class object instantiates an intermediate object, defines a new set of request and reply channels, and binds them all using an appropriate interface adaptor. Inheritance can be achieved by combining and extending intermediate object templates of already existing classes [SL96].

3. Generic Synchronization Policies

This section briefly describes the concept of Generic Synchronization Policies (GSPs), the synchronization mechanism we have integrated into our π-calculus based object model. For further information, refer to the thesis of McHale [McH94].

GSPs provides a mechanism to synchronize objects at the granularity of method invocations and are based on a paradigm called "Service-object Synchronization" (Sos). The paradigm consists of the following four concepts: 1) events (and code executed at them), 2) delaying and starting method invocations, 3) accessing information about method invocations, and 4) a strict separation between synchronization code/data and code/data of the object itself.

In order to show how this paradigm works, we first need to describe the sequence of events that take place when an object invokes an operation upon another object.

From the service object's perspective (which is the only one we will consider), there are three events of interest: *arrival*, *start* and *term* (short for termination) of a method invocation. When an invocation arrives, it may be delayed due to some synchronization constraints. Some time later, it will start execution, and finally it will terminate execution. We assume that events do not overlap. For example, if two invocations arrive at the same time, we assume that their arrival events will be ordered. The sequence of events is summarized in figure 2.

GSPs permit an action (user code) to be associated with each possible event. The execution of an action will always complete before another event can occur. Synchronization constraints between methods are expressed using the concept of a guards (i.e. a boolean expression). Each invocation will be delayed until the corresponding guard evaluates to true.

In GSPs the genericity lies in the fact that actions and guards are not associated directly with a particular method of a given object. Conceptually,

4. They can be compared to generator objects used by Abadi and Cardelli [AC96].

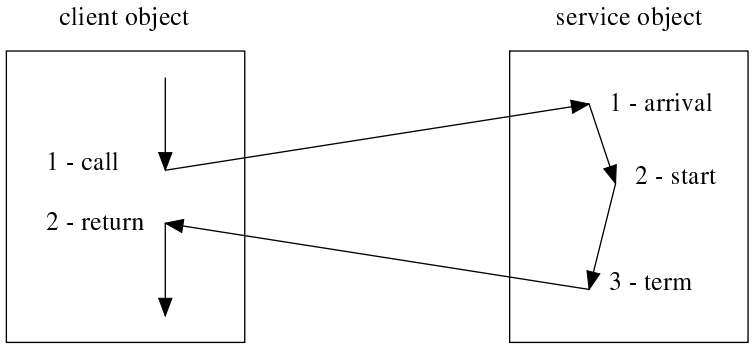

Figure 2. *The events in the lifespan of a typical method invocation.*

methods are grouped into *categories* for which actions and guards are specified. At instantiation time, all methods will "inherit" all actions and guards according to their associated category.

The second concept of the Sos paradigm is needed in order to delay a method invocation due to some synchronization constraints and start its execution when all synchronization constraints are fulfilled. The Sos paradigm, however, does not specify how the mechanism for delaying and starting invocation has to be implemented.

In order to express complex synchronization schemes, it is necessary to access information about the method invocations upon an object. In code for actions and guards, the following information needs to be made available:

- the arrival time of the current invocation (for which the action or guard is executed),
- the number of waiting invocations from a given category,
- the number of executing invocations from a given category,
- a list of all waiting invocations from a given category, and
- the method's parameters.

Other information could be added, like, for example, the number of terminated invocations or the list of all executing invocations from a given category. In our modelling, however, we restricted ourselves to the points mentioned above.

Finally, the Sos paradigm requires a strict separation between synchronization code/data and code/data of the object itself. As an example, synchronization code (guards and actions) cannot access instance variables of the object and vice versa, ensuring that concurrent execution of synchronization and sequential code cannot interfere (e.g., synchronization code cannot access information currently being updated by sequential code). This requirement also

ensures that it is not only possible to completely separate the specification and implementation of synchronization code from sequential code, but also allows to reuse synchronization policies in a different setting.

As an example, we present the (well-known) Readers-Writers policy. The policy has two categories of methods: one category representing methods that only read instance variables and another category representing methods that change the value of some instance variables of an object. Using GSPs, the Readers-Writers policy could be specified as follows (we use the same syntax as in [McH94]):

```
policy ReadersWriters [ReadOps, WriteOps] {

        function ReaderAllowed (t: Invocation): Bool
        begin
                return exec (WriteOps) = 0;
        end

        function WriterAllowed (t: Invocation): Bool
        begin
                return exec (ReadOps) + exec (WriteOps) = 0;
        end

        map     guard(ReadOps) → ReaderAllowed
                guard(WriteOps) → WriterAllowed
}
```

This policy specifies that a read operation can only take place when there is no executing write operation (i.e. exec (WriteOps) = 0) and a write operation can only take place when no other operation is executing (exec (ReadOps) + exec (WriteOps) = 0). The construct

```
        map     guard(ReadOps) → ReaderAllowed
                guard(WriteOps) → WriterAllowed
```

binds the guards for the different method categories.

The Readers-Writers synchronization policy defined above can be used in a class-based way to synchronize objects where all methods are either reader or writer methods, like in the example below:

```
class IntStack {
        ...             /* defines empty, push, pop, and top */
        synchronization
                ReadersWriters [ <push pop>, <empty top> ]
}
```

4. Modelling GSPs in the π-calculus

The concept of GSPs is not tied to a particular language and can be easily adapted to most object models. In the following, we are illustrating the integration of GSPs into our π-calculus based object model. This integration not only

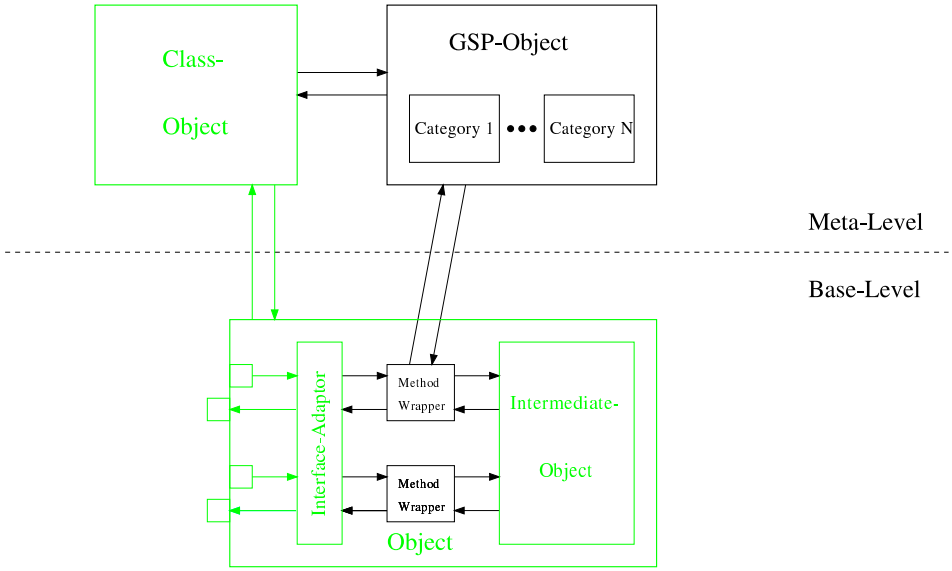

Figure 3. *Integration of GSPs into the object model.*

shows that our object model is open enough to be extended with additional features like GSPs, but also demonstrates that the GSP paradigm can be easily adapted to be used in any object model.

In order to model GSPs, we use a similar architecture as for modelling normal objects. Synchronization policies are represented as objects in the metalevel (and, therefore, they behave as metaobjects) while so-called method wrapper objects providing an interface for synchronization are part of the base level. The main difference of modelling GSPs compared with our modelling of plain objects is that the policy metaobjects are not linked to method wrapper objects a priori. Method wrapper objects are fully generic: they can wrap any method of any type and they can be bound to any policy. Furthermore, the binding to a policy can be changed at runtime. The overall structure of the GSP integration is shown in figure 3.

4.1. *Synchronization wrappers*

Like McHale we have also opted for the technique of placing a synchronization wrapper around each method to be synchronized. This technique is commonly employed in the implementation of synchronization mechanisms. The main idea of the synchronization wrappers is to take a pure unsynchron-

ized object and to wrap its methods in a pre- and post-synchronization code. For example, a method like

```
method m1()
begin
        body;
end
```

will be transformed (wrapped) into

```
method m1()
begin
        pre-synchronization code;
        body;
        post-synchronization code;
end
```

In terms of the π-calculus this means, that if a method is represented as the process

$$M \equiv a(x_n, r).(\upsilon\ y_m)(P \mid \overline{r}\langle y_m \rangle)$$

the wrapping operation will lead to the following process:

$$\begin{aligned} WM \equiv (\upsilon\ a_w, r_w)\ (\ & a(x_n, r).(PreWrapper \mid \overline{a}_w\langle x_n, r_w \rangle) \\ & \mid a_w(x_n, r_w).(\upsilon\ y_m)(P \mid \overline{r}_w\langle y_m \rangle) \\ & \mid r_w(y_m).(PostWrapper \mid \overline{r}\langle y_m \rangle)\ ) \end{aligned}$$

Fortunately, the encoding of this wrapping operation can be greatly simplified by the use of records [Var96]. The reader should note that the term $(\upsilon\ y_m)(P \mid \overline{r}\langle y_m \rangle)$ is an asynchronous encoding of a method.

The wrapping process itself is modelled by an object which has at least two methods. One method is used to execute the synchronized method (compare with the given wrapped process WM) and the other method is used to set the policy specific method wrappers. The following π-process illustrates the wrapping:

$$\begin{aligned} & (\upsilon\ PreWrapper, PostWrapper) \\ & \quad (\ SetWrapper(pre, post).(\ PreWrapper(\_).\overline{PreWrapper}\ \langle pre \rangle \\ & \qquad\qquad\qquad\qquad \mid PostWrapper(\_).\overline{PostWrapper}\ \langle post \rangle) \\ & \quad \mid WM\ ) \end{aligned}$$

The wrapper objects behave as two-way generic communication interceptors. The reason why we had to use such objects is that the language PICT does not support message interception. Such a facility could be added to the language through an extension of the semantics of channel creation. The creation of a new channel would be mapped internally to a process which can be parameterized with an input and output related process which is activated when an input or output takes place. After the interceptor process has finished, the initiated communication proceeds as usual.

4.2. *Binding a GSP to an object*

A Readers-Writers policy described in section 3 can roughly be represented by the following process:

$$\begin{aligned} ReadersWritersPolicy(Readers, Writers) \equiv (\ & ReaderAllowed(Invocation) \\ & \mid WriterAllowed(Invocation) \\ & \mid Map(Readers, ReaderAllowed) \\ & \mid Map(Writers, WriterAllowed)\ ) \end{aligned}$$

This process takes as arguments the methods belonging to the appropriate categories. Then the *ReadersWritersPolicy* process starts four processes in parallel: *ReaderAllowed* and $WriterAllowed$ which are the policy specific synchronization guards and two *Map* processes which immediately end after binding the methods of a category to its synchronization guards (see section 3 for details).

An initial binding of a GSP is done at object creation time. In order to add GSPs to our object model, the method *Create*[5] of the class object had to be modified. First, this method takes an additional parameter *Policy* which holds the actual synchronization policy and second, two additional processes ($CreateWrapped$ and *BindPolicy*) had to be added to the method *Create*. The following process illustrates the modified method *Create*:

$$\begin{aligned} Create(Policy, Object) \equiv (\upsilon\ & Intermediate, WrappedIntermediate, NewInstance, Self\ ) \\ & (\ CreateIntermediate(Intermediate) \\ & \mid CreateWrapped(Intermediate, WrappedIntermediate) \\ & \mid CreateInstance(WrappedIntermediate, NewInstance) \\ & \mid (BindPolicy(NewInstance, Policy, Self\ ) \mid \overline{Object}\ \langle Self\rangle)\ ) \end{aligned}$$

To create a new object, the method *Create* receives a synchronization policy in *Policy* and a reply channel *Object* which is used to return the new created object to the caller. The method *Create* starts four processes in parallel which are synchronized through its communicated channels. *CreateIntermediate* sends an intermediate object along the channel *Intermediate*. *CreateWrapped* takes this object and sends an intermediate object with empty synchronization wrappers along *WrappedIntermediate*. *CreateInstance* creates the interface adapter and sends the resulting object along *NewInstance*. Finally, *BindPolicy* takes this instance, binds it to the given policy, and establishes the correct binding of self.

With this implementation, we have an object based approach of the GSP mechanism because the object creation is parameterized with the actual synchronization policy. McHale proposed that the synchronization code is class based and that the synchronization policy is part of the class definition. We use an object based approach in order to allow different synchronization policies to be assigned to objects of the same class and that policies can also be changed at runtime. As mentioned earlier, a policy is represented as an object in the metalevel (see figure 3). This policy object is a metaobject for the

5. *Create* is a method of the class object used to create a new objects (see [LSN96]).

method wrapper objects as well as for the synchronized object. The method wrapper objects themselves do not have any instance variables: all method wrapper objects use exclusively the synchronization variables provided by the policy object. Therefore, the synchronization variables can been viewed as class variables, and the policy methods as class methods.

It is important to note that the binding of a GSP only affects the creation of an object. The interface of an object for use remains the same.

4.3. *GSPs and Inheritance*

In our model, synchronization code is not inherited to subclasses. This means that when a subclass is defined, we have to assign synchronization code also to the inherited methods. This sounds like a drawback, but this decision has a serious reason.

PICT is a strongly typed language [PT97]. The type system of PICT allows to define polymorphic data structures which we heavily use in our object encodings. Unfortunately, our encodings are also restricted by the PICT type system. When we inherit wrapped methods, the types of the methods of the superclass and the subclass have to be invariant. This invariance is introduced by the self parameter in the method interfaces[6] and the method wrapper objects, especially by the $PreWrapper$ process (see the process encoding of WM). When the type of the arguments (x_n, r) is T, it is type safe to pass arguments of type S to $PreWrapper$ along channel a when S is a subtype of T (written $S <: T$). The prewrapper process ends sending the received value $\langle x_n, r\rangle$ along channel a_w. When the type of $\langle x_n, r\rangle$ is T, it is type safe to send a value of type S along a_w when T is a subtype of S (written $T <: S$). From these two conditions, it follows that T and S must be invariant (written $T \equiv S$). But this condition is too restrictive for practical use because it implies that the type of self must be the same in the super- and subclass which means that it is not possible to extend a subclass with additional features. Therefore, synchronization code cannot be inherited.

Furthermore, inheritance and synchronization constraints in concurrent object systems can often conflict with each other, resulting in inheritance anomalies where reprogramming of inherited code is necessary [MY93]. Our approach that wrapped methods cannot be inherited, does not intent to overcome inheritance anomalies; it is rather driven by the underlying type system. A more detailed analysis of this problem is part of future research.

6. The parameter self is part of the implementation interface of methods used in the intermediate object [LSN96].

5. Discussion

In the following, we will discuss our main observations of the integration of GSPs into our π-calculus based object model.

5.1. *Evaluation of Generic Synchronization Policies*

Our experiments have shown that the concept of generic synchronization policies is a promising base for the definition of higher-level, reusable synchronization abstractions. The original specification described in [McH94], however, has a few drawbacks which we will discuss in the following.

First of all, GSPs are used to synchronize concurrent *objects*, but their instantiation is restricted to *classes*: each instance of a class has the same synchronization code. Due to the fact that the concept of GSPs is independent of classes, we have extended it in our modelling in order to decide at object creation time which synchronization policy is bound to an object. Therefore, it is possible that not all instances of a class have the same synchronization policy.

McHale claims that both, external and Self calls of methods, should be synchronized by the same synchronization mechanism [McH94]. However, we argue that there are good reasons why a single layer/scheme of synchronization is not enough. On a prototype implementation of connector types [Duc97] in a concurrent setting, we discovered that it is necessary to call (internal) methods of an object without passing through the external interface and, therefore, bypassing the synchronization layer for external clients. In order to guarantee consistency of an objects internal state, there is a need for a second layer of synchronization which is a topic for future research.

The concept of GSPs does not take care of method invocations that recursively call other methods of the same object by external clients (e.g., method m_1 of object A calls a method of object B which calls method m_2 of A). There is no possibility to assign an information to the invocation of m_2 that this invocation is a direct cause of the execution of m_1 and, therefore, should be synchronized differently.

5.2. *π-calculus based object model*

As we have described in section 4, the integration of GSPs into our object model is not straightforward. The integration would be much easier if we could observe and control method invocations directly from a metalevel without the need to explicitly instantiate method wrapper objects. What we need to add to the language is a metalevel abstraction which allows to control the message sending over channels. One could think of a kind of channel interceptors which observe, delay, and eventually modify all messages sent over the channel to be

controlled. We think that such an abstraction will not have any implications on the underling calculus, but only changes the semantics of message passing in the language itself. It is a topic of ongoing research to investigate the expressive power of message interception in general, and in our π-calculus based object model in particular.

On the other hand, our π-calculus based object model is open enough to be easily extended with additional abstractions. It has proven to be a promising formal base for the definition of a black-box framework for modelling objects and compositional abstractions. Furthermore, our work illustrates that the use of a metalevel allows for an easy and flexible integration of additional abstractions without the need to change the underlying existing model.

5.3. *Evaluation of* PICT

Both the integration of GSPs into our object model and first experiments in integrating connector types [Duc97] have shown that it is necessary to extend the existing metalevel and metaobject protocol (MOP). The current version of PICT, however, did not allow a straightforward integration of these extensions. We are, therefore, looking for a set of small extensions to the current PICT version which allow us to express the required abstractions in a more natural way.

The type system of PICT integrates a number of features found in recent work on theoretical foundations for typed object-oriented languages [Tur96] and allows the definition of polymorphic data structures and processes what we heavily use in our encodings. However, our results show that the current type system is too restrictive for an efficient implementation of metaobject protocols and lacks of a support for runtime type information. One of our next steps will be to investigate how the type system could be extended in order to fulfill our needs.

Implementing GSPs, we have discovered an interesting property of the type system of PICT: it is not only possible to define generic *classes*, but also generic *methods*. We have used this property for the *SetWrapper* process of a method wrapper object, in order to attach generic synchronization code. To our knowledge, no strongly typed object-oriented programming language supports such a feature.

6. Conclusions and future work

Our experiments show that the π-calculus is a promising formal foundation for experimenting with different object and component models, that objects are most easily synchronized when synchronization policies are reified as first class entities (i.e. metaobjects), and that McHale's concept of "generic syn-

chronization policies" forms a base for the definition of higher-level, reusable synchronization abstractions.

Ultimately, we are targeting the development of open, hence distributed systems [NSL96]. Given the ad hoc way in which the development of open systems is supported in existing languages, we identify the need for composing software from predefined, plug-compatible software components. The overall goal of our work is the development of a formal model for software composition, integrating a black-box framework for modelling objects and components, and an executable composition language for specifying components and applications as compositions of software components.

A composition language for open systems should not only have its formal semantics specified in terms of communicating processes, but should really support concurrent and distributed behaviour. Since the present runtime system of PICT only works for a single processor system, and the development of the language itself focuses more on a functional programming style, we have decided to implement an environment for a PICT-like language on top of Java. This will allow us to further experiment with abstractions needed for distribution and the implementation of a sophisticated metalevel.

Acknowledgements

We thank all members of the Software Composition Group for their support of this work, especially Oscar Nierstrasz, Patrick Varone, Tamar Richner, and Franz Achermann.

References

[AC96] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer, 1996.

[Ame87] Pierre America. POOL-T: A Parallel Object-Oriented Language. In Akinori Yonezawa and Mario Tokoro, editors, *Object-Oriented Concurrent Programming*, pages 199–220. MIT Press, 1987.

[Blo79] Toby Bloom. Evaluating Synchronization Mechanisms. In *Proceedings of the Seventh Symposium on Operating Systems Principles*, pages 24–32, December 1979.

[Bri96] Jean-Pierre Briot. An Experiment in Classification and Specialization of Synchronization Schemes. In Kokichi Futatsugi and Satoshi Matsuoka, editors, *Object Technologies for Advanced Software*, LNCS 1049, pages 227–249. Springer, March 1996. Proceedings ISOTAS '96.

[BS95] Manuel Barrio Solorzano. *Estudio de Aspectos Dinamicos en Sistemas Orientados al Objecto*. PhD thesis, Universidad de Valladolid, September 1995.

[DLDR+91] D. Decouchant, P. Le Dot, M. Riveill, C. Roisin, and X. Rousset de Pina. A Synchronization Mechanism for an Object Oriented Distributed

System. In *Proceedings of 11th International Conference on Distributed Computing Systems*, pages 152–159. IEEE, May 1991.

[Duc97] Stéphane Ducasse. Réification des schémas de conception: une expérience. In Roland Ducournau and Serge Garlatti, editors, *Proceedings of Langages et Modèles à Objets '97*, pages 95–110, Roscoff, October 1997. Hermes.

[Jon93] Cliff B. Jones. A Pi-Calculus Semantics for an Object-Based Design Notation. In Eike Best, editor, *Proceedings CONCUR '93*, LNCS 715, pages 158–172. Springer, 1993.

[KdRB91] Grégor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[LSN96] Markus Lumpe, Jean-Guy Schneider, and Oscar Nierstrasz. Using Metaobjects to Model Concurrent Objects with PICT. In *Proceedings of Langages et Modèles à Objets '96*, pages 1–12, Leysin, October 1996.

[McH94] Ciaran McHale. *Synchronization in Concurrent, Object-oriented Languages: Expressive Power, Genericity and Inheritance*. PhD thesis, Department of Computer Science, Trinity Collegue, Dublin, Ireland, October 1994.

[Men94] Tom Mens. A survey on formal models for OO. Technical Report vub-tinf-tr-94-03, Department of Computer Science, Vrije Universiteit Brussel, Belgium, 1994.

[Mil89] Robin Milner. A Calculus of Mobile Processes, Part I+II. Technical Report ECS-LFCS-89-85, Computer Science Department, University of Edinburgh, UK, 1989.

[Mil91] Robin Milner. The Polyadic Pi-Calculus: a Tutorial. Technical Report ECS-LFCS-91-180, Computer Science Department, University of Edinburgh, UK, October 1991.

[MY93] Satoshi Matsuoka and Akinori Yonezawa. Analysis of inheritance anomaly in object-oriented concurrent programming languages. In Gul Agha, Peter Wegner, and Akinori Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 107–150. MIT Press, 1993.

[NSL96] Oscar Nierstrasz, Jean-Guy Schneider, and Markus Lumpe. Formalizing Composable Software Systems – A Research Agenda. In *Proceedings 1st IFIP Workshop on Formal Methods for Open Object-based Distributed Systems*, pages 271–282. Chapmann & Hall, 1996.

[PT95] Benjamin C. Pierce and David N. Turner. Concurrent Objects in a Process Calculus. In Takayasu Ito and Akinori Yonezawa, editors, *Theory and Practice of Parallel Programming (TPPP)*, LNCS 907, pages 187–215. Springer, April 1995.

[PT97] Benjamin C. Pierce and David N. Turner. Pict: A Programming Language based on the Pi-Calculus. Technical Report CSCI 476, Computer Science Department, Indiana University, March 1997.

[San95] Davide Sangiorgi. Lazy functions and mobile processes. Technical Report RR-2515, INRIA Sophia-Antipolis, April 1995.

[SL96] Jean-Guy Schneider and Markus Lumpe. Modelling Objects in PICT. Technical Report IAM-96-004, University of Bern, Institute of Computer Science and Applied Mathematics, January 1996.

[Tur96] David N. Turner. *The Polymorphic Pi-Calculus: Theory and Implementation*. PhD thesis, Department of Computer Science, University of Edinburgh, UK, 1996.

[TV89] Chris Tomlinson and Singh Vineet. Inheritance and Synchronization with Enabled-Sets. In Norman Meyrowitz, editor, *Proceedings OOPSLA '89*, volume 24 of *ACM SIGPLAN Notices*, pages 103–112, October 1989.

[Var96] Patrick Varone. Implementation of "Generic Synchronization Policies" in PICT. Technical Report IAM-96-005, University of Bern, Institute of Computer Science and Applied Mathematics, April 1996.

[VdBL89] Jan Van den Bos and Jan Laffra. PROCOL – A Parallel Object Language with Protocols. In Norman Meyrowitz, editor, *Proceedings OOPSLA '89*, volume 24 of *ACM SIGPLAN Notices*, pages 95–102, October 1989.

[Wal95] David J. Walker. Objects in the Pi-Calculus. *Information and Computation*, 116(2):253–271, 1995.