# A Language Framework for Multi-Object Coordination

Svend Frølund and Gul Agha

Department of Computer Science
1304 W. Springfield Avenue
University of Illinois at Urbana-Champaign
Urbana, IL 61801, USA

Email: { frolund | agha }@cs.uiuc.edu

**Abstract.** We have developed language support for the expression of multi-object coordination. In our language, coordination patterns can be specified abstractly, independent of the protocols needed to implement them. Coordination patterns are expressed in the form of constraints that restrict invocation of a group of objects. Constraints are defined in terms of the interface of the objects being invoked rather than their internal representation. Invocation constraints enforce properties, such as temporal ordering and atomicity, that hold when invoking objects in a group. A constraint can *permanently* control access to a group of objects, thereby expressing an inherent access restriction associated with the group. Furthermore, a constraint can *temporarily* enforce access restrictions during the activity of individual clients. In that way, constraints can express specialized access schemes required by a group of clients.

## 1 Motivation

Coordination of activities is a fundamental aspect of programming languages for concurrent and distributed systems. In existing languages, multi-object coordination is expressed in terms of explicit communication protocols such as two-phase commit; it is not possible to express coordination patterns independent of their implementation. The inability to abstract over multi-object coordination results in low level specification and reasoning. As a consequence, it is not possible to compose multiple coordination patterns without changing their implementation. Moreover, it is difficult to modify and customize the implementation of coordination patterns since the involved protocols are hard-wired into applications.

Our approach is to express coordination patterns in the form of *multi-object constraints*. A multi-object constraint maintains certain properties such as temporal ordering and atomicity associated with invocations processed by a group of objects. Multi-object constraints are specified abstractly and independent of the protocols required to implement the enforced properties. A multi-object constraint restricts the freedom of objects to process invocations: whether or not an object may process an invocation depends on the current status and invocation history of a group of objects. Enforced restrictions capture a large class of multi-object coordination schemes. Consider the following examples:

- A group of resource administrators must adhere to a common allocation policy. An example of a common policy is a limit on the total number of resources the administrators may collectively allocate. Enforcement of a common policy can be expressed as a constraint on invocations that allocate resources: an allocation request can only be serviced if the total number of allocated resources minus the total number of released resources is below the enforced limit.
- A group of dining philosophers share a number of chopsticks. The number of philosophers is equal to the number of chopsticks and a philosopher needs two chopsticks in order to eat. A coordination scheme should (at least) prevent deadlock when philosophers attempt to pick up chopsticks. Deadlocks can be avoided by a multi-object constraint that enforces atomic invocation of the pick method in two chopstick objects: each philosopher picks up either both or neither chopstick that he needs.

Our constructs build on the observation that multi-object constraints can be specified independent of the representation of the constrained objects. Specifying multi-object constraints in terms of the interface of the constrained objects enables better description, reasoning and modification of multi-object constraints. Only utilizing knowledge of interfaces when describing multi-object constraints separates coordination and object functionality. This separation enables system design with a larger potential for reuse. Objects may be reused independent of how they are coordinated; conversely, multi-object coordination patterns may be reused on different groups of objects. In particular, it is possible to abstract over coordination patterns and factor out generic coordination structures. As noted in [AB92], separation also gives a more natural way of expressing the coordination schemes found in many problem domains.

Multi-object constraints are described in the form of *synchronizers*. Conceptually, a synchronizer is a special object that observes and limits the invocations accepted by a set of ordinary objects that are being constrained. Synchronizers may overlap: multiple, separately specified, synchronizers can constrain the same objects. Operationally, the compiler translates synchronizers into message-passing between the constrained objects. The advantage of synchronizers is that the involved message-passing is transparent to the programmer who specifies multi-object constraints in a high-level and abstract way. The implementation of synchronizers can involve direct communication between the constrained objects, indirect communication with a central “coordinator” or a hybrid. Utilizing a high-level specification of multi-object constraints it is possible to map the same multi-object constraint to different implementations.

In object-based concurrent computing it has become commonplace to describe per object coordination in the form of synchronization constraints [Nie87, TS89, Neu91, AGMB91, MWBD91, Frø92]. Both synchronizers and synchronization constraints are based on conditions which are verified prior to the processing of invocations. The conditions of synchronization constraints are expressed in terms of the encapsulated state of individual objects, e.g. in the form of boolean predicates. The conditions associated with synchronizers involve the status and invocation history of a group of objects.

As we explain in the related work section (Section 5), existing approaches to

multi-object coordination do not take local coordination such as synchronization constraints into account. For example, transactions are meant to coordinate access to passive records not active objects. On the other hand, synchronizers are designed to integrate with local synchronization constraints: conceptually, synchronizers supplement synchronization constraints with additional conditions that must be met for invocations to be legal. With synchronizers, the programmer can specify per object coordination and multi-object coordination using similar concepts.

In the following sections we give more details about synchronizers. In Section 2, we introduce a notation for describing synchronizers. The introduced notation is applied in a number of examples in Section 3 and Section 4. Section 5 gives an overview of related work and Section 6 concludes with a discussion of our approach.

## 2 Basic Concepts and Notation

For simplicity we assume that synchronizers are integrated with an object-oriented concurrent language that adheres to the Actor [Agh86] model of computation. Specifically this implies that message-passing is asynchronous so that sending objects are not blocked by enforced synchronizers. Although our development is based on Actors, we believe the concepts behind synchronizers can be integrated with most concurrent object-oriented languages.

It is important to note that our notation only constitutes the "core" functionality of a framework for describing multi-object constraints. The purpose of this paper is to promote the *concept* of synchronizers, their separate specification and high-level expression of multi-object coordination. The main objective of providing a notation for multi-object constraints is to give the principles a concrete form and allow the description of example constraints.

Figure 1 gives an abstract syntax for synchronizers. In the following we will give an informal description of the semantics of synchronizers. The structure of a synchronizer is specified using the $\{\ldots\}$ constructor. Such constructors are named, allowing instantiation of synchronizers with similar structure. A constructor has a list of formal parameters that are bound to actual values when synchronizers are instantiated. The init part of a synchronizer declares a list of local names that hold the state of a synchronizer.

Synchronizers are instantiated by ordinary objects. In that way, objects can enforce constraints on other objects. In our framework, enforcing a constraint on an object requires a reference to that object. However, it is possible to employ other models of locality for constraint enforcement. For example, objects could be required to have certain capabilities in order to enforce constraints on other objects. Furthermore, constraints could be specified for groups of objects that are defined using patterns rather than identities [AC93].

A synchronizer is a special kind of object that observes and limits invocation of other objects: a synchronizer cannot be accessed directly using message-based communication.[1] The invocations observed and limited by a synchronizer are iden-

[1] There is nothing inherent in the synchronizer concept that prevents direct communication. Our convention merely reflects a design choice that, for pedagogical reasons, makes a clear distinction between synchronizers and objects.

```
binding      ::= name := exp                                  |
                 binding_1 ; binding_2
pattern      ::= object.name                                  |
                 object.name(name_1 , ... , name_n)           |
                 pattern_1 or pattern_2                       |
                 pattern where exp
relation     ::= pattern updates binding                      |
                 exp disables pattern                         |
                 atomic(pattern_1 , ... , pattern_i)          |
                 pattern stops                                |
                 relation_1 , relation_2
synchronizer ::= name(name_1 , ... , name_n)
                 { [init binding]
                   relation
                 }
```

**Fig. 1.** Abstract syntax for synchronizers.

tified using pattern matching. Pattern matching is defined by the following rules:

- The pattern $o.n$ matches all messages invoking method $n$ in object $o$.
- The pattern $o.n(x_1 , \ldots , x_n)$ matches all messages matched by the pattern $o.n$ and binds the actual values of a matching message to the names $x_1 , \ldots , x_n$.
- The pattern $p_1$ or $p_2$ matches messages that match either $p_1$ or $p_2$.
- A message matches the pattern $p$ where $exp$ if the message matches the pattern $p$ and the boolean expression $exp$ evaluates to true. The expression is side-effect free and evaluated in a context including the names bound in $p$.

Since we assume an object-oriented invocation model, each message invokes a specific method in a specific object. The or combinator allows the definition of patterns that cover multiple methods in multiple objects. In that way, messages sent to different objects may match the same pattern. Notice that a corresponding and combinator can be expressed by a boolean predicate in a where clause.

Having explained patterns, we are now in a position to describe the semantics of the relation part of a synchronizer. A relation that contains the updates operator can be used to observe specific invocations of specific objects. In particular, relations of the form *pattern* updates *binding* changes the state of the enclosing synchronizer according to *binding* each time an object is invoked by a message that matches *pattern*. In order to maintain consistency of synchronizers, bindings are established as atomic actions. If an invocation matches the pattern of multiple updates relations, the corresponding bindings are made in some indeterminate order after the pattern matching is complete.

Observations made by a synchronizer are consistent with the temporal and causal ordering between the observed invocations. For example, if an object is invoked by

the message $m_2$ after the message $m_1$, synchronizers will observe $m_2$ after $m_1$. Using the updates operator, a synchronizer can record the invocation history of objects.

The disables operator is used to prevent invocations. The relation *exp* disables *pattern* prevents acceptance of messages that match *pattern* if the expression *exp* evaluates to true in the current state of the enclosing synchronizer. Relations that contain the disables operator define conditions that must be met before an object can be invoked by certain messages. Prevented invocations are delayed at the receiver if the enforced multi-object constraints are not satisfied.

Synchronizers that contain both updates and disables relations can enforce temporal ordering where the legality of invocations is determined by the past invocation history. A given message can match the pattern of both updates and disables relations in the same synchronizer. For such messages, there is exclusive access to the state of the enclosing synchronizer during evaluation of the expression associated with disables relations and a possible subsequent binding caused by an updates relation. Hence, a combination of updates and disables operators can express mutual exclusion and exclusive choice involving sets of invocations.

Certain coordination schemes require the expression of atomicity constraints, i.e. multiple invocations that occur without any observable temporal ordering. The relation atomic($pattern_1, \ldots, pattern_i$) involves $i$ kinds of messages which match the patterns $pattern_1, \ldots, pattern_i$, respectively. The relation ensures that acceptance of a message from either kind occurs along with acceptance of messages from the other $i-1$ kinds without any observable middle states where only part of the messages have been accepted. Regardless of the observing context, either all or none of the patterns are matched and there is no temporal ordering between the matching invocations.

The atomic operator gives rise to indivisible scheduling of multiple invocations at multiple objects. In our framework, a set of invocations scheduled atomically cannot contain multiple invocations of the same object. Consequently, atomicity of a set of invocations can be ensured prior to the processing of the invocations. This is in contrast to transactions that provide atomicity relative to the transitive effect of invocations. Our notion of atomicity is cheaper to implement and can be combined with temporal constraints. For example, it is possible to describe temporal ordering as well as exclusive choice between sets of invocations that are scheduled atomically.

Once instantiated, a synchronizer remains in effect until observing an invocation that matches the pattern of a stops relation. The relation *pattern* stops implies that acceptance of a message that matches *pattern* terminates the enclosing synchronizer. A synchronizer without a stops operator remains in effect permanently. In practice, more sophisticated control over the duration of synchronizers may be needed. We simply want to make the case for dynamic enforcement of synchronizers and we incorporate the stops relation as a simple way of achieving dynamic constraint enforcement.

In summary, a synchronizer coordinates a group of objects by enforcing temporal and atomicity properties on certain patterns of invocations processed by the group. The groups being coordinated can overlap and the coordination may be initiated and terminated dynamically. Figure 2 illustrates the functionality of synchronizers. In that figure, pattern matching is depicted as matching shapes between patterns and messages. Synchronizers may disable certain patterns: disabled patterns are black and enabled patterns are white. Arrows illustrate the status of messages: an arrow

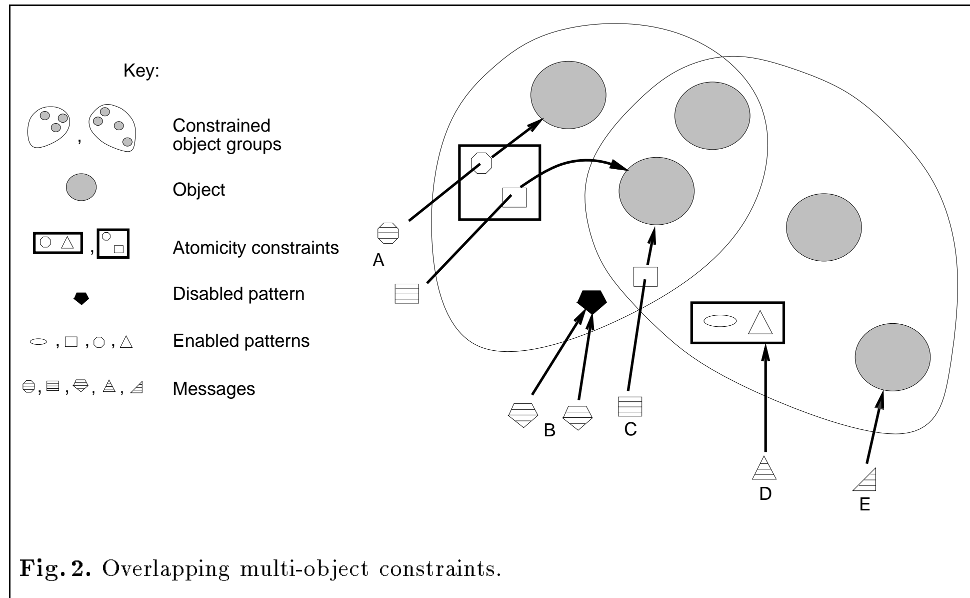


**Fig. 2.** Overlapping multi-object constraints.

that ends at an object starts at an enabled message and an arrow that ends at a pattern starts at a disabled message. Objects that are not pointed to by an arrow have no messages to process in the current system state. The messages at `B` are disabled whereas the message at `C` is enabled. The `E` message is unconstrained since it does not match any pattern. Patterns may be grouped together into atomicity constraints that ensure that multiple invocations are scheduled as an atomic action. Grouping of patterns into atomicity constraints is depicted by boxes around the groups. The messages at `A` satisfy the atomicity constraint whereas the message at `D` is blocked waiting for another message before both can be scheduled atomically.

Having introduced the core principles of synchronizers, the following two sections illustrate the use of synchronizers. We believe that synchronizers can be used to express many different aspects of multi-object coordination within the same basic framework. Much of the generality of synchronizers stem from the flexible binding scheme between constraints and constrained invocations. Section 3 provides examples of constraints that express logical grouping and interdependence between objects. The constraints of Section 4 capture specialized access schemes defined and enforced by clients.

## 3 Coordinating Groups of Objects

In this section we demonstrate how certain kinds of interdependence between objects can be represented by means of synchronizers that permanently constrain a group of objects. Synchronizers can express interdependence relations that take the form of properties that must hold for invocations processed by a group of objects.

The cooperating resource administrators mentioned in the introduction is a prototypical example of interdependent objects. In Example 1 below, we illustrate how

our notation can capture the dependence between the cooperating administrators.

**Example 1** (*Cooperating Resource Administrators*). Consider two resource administrators that allocate different kinds of resources, e.g. printers and disks. Each administrator has a local synchronization constraint that prevents allocation of more resources than the system provides. In addition to the local constraint the administrators may be subject to a collective constraint on the number of resources they allocate collectively. For example, enforcing a collective constraint may be necessary if the allocated resources are external devices and the bus that interconnects the devices has a maximum capacity.

```
AllocationPolicy(adm1,adm2,max)
{  init prev := 0

   prev >= max  disables (adm1.request  or adm2.request),
   (adm1.request  or adm2.request)  updates prev := prev + 1,
   (adm1.release  or adm2.release)  updates prev := prev - 1
}
```

**Fig. 3.** A Synchronizer that enforces collective bound on allocated resources.

Maintenance of a collective bound on allocated resources can be described external to the administrators as a synchronizer. Figure 3 contains a synchronizer that prevents collective allocation of more than `max` resources by the two administrators. The names `adm1` and `adm2` are references to the two constrained administrators and the variable `prev` holds the total number of resources that have previously been allocated and not yet released. □

Synchronizers give flexible ways of describing coordination relations between shared servers. With synchronizers, coordination is expressed independent of the representation of the involved servers. The resulting modularity makes it possible to modify the coordination scheme without changing the servers and vice versa. In particular, it is possible to dynamically add new servers to an existing coordinated group by instantiating a new synchronizer. Furthermore, synchronizers allow the description of overlapping groups of interdependent servers.

A part-whole hierarchy is another structure that gives rise to permanent enforcement of constraints on a group of objects. A part-whole hierarchy contains a set of objects that are related by an *is-component-of* relation. A part-whole hierarchy induces consistency constraints on its members: the independence of individual parts is limited by the fact that they are parts of a containing entity. As we illustrate in the following example, synchronizers can be used to capture consistency constraints of concurrent part-whole hierarchies.

**Example 2** (*Vending Machine*). A vending machine is an example of a concurrent part-whole hierarchy with a consistency constraint. A vending machine has a number

of parts: a coin accepter into which coins can be inserted and a number of slots that each contain a piece of fruit. The parts of a vending machine are subject to a consistency requirement in order for the vending machine to have the desired functionality: insert enough money and get back a piece of fruit from one of the slots. When a sufficient amount of money has been inserted into the coin accepter, one or more of the slots are available for opening (each slot may be priced differently). Opening one of the slots will remove the inserted money from the coin accepter and prevent other slots from being opened. Pushing a special button on the coin accepter, it is possible to get a refund.

```
VendingMachine(accepter,apples,bananas,apple_price,banana_price)
{   init amount := 0

    amount < apple_price   disables apples.open,
    amount < banana_price  disables bananas.open,
    accepter.insert(v)  updates amount := amount + v,
    (accepter.refund  or apples.open  or bananas.open)
        updates amount := 0
}
```

**Fig. 4.** A Synchronizer that coordinates access to the parts of a vending machine.

Assume that each part of the vending machine is represented as a separate object. The constraint of a vending machine hierarchy can be modeled as a synchronizer that enforces a constraint on the parts of the vending machine. Figure 4 contains a synchronizer which reflects the functionality of a vending machine constraint. The vending machine in Figure 4 has two slots: one for apples and one for bananas. The name `apples` refers to the apple slot and the name `apple_price` is the price of apples and vice versa for bananas.[2] The name `accepter` refers to an object representing the coin accepter. The methods of a coin accepter are `insert()` and `refund()`. The apple and banana slots each have an `open` operation that can be invoked if the `accepter` contains enough money.

In Figure 4, the variable `amount` holds the amount of money that is available in the coin accepter. Insertion of money into the coin accepter increments the variable `amount`. Opening any slot or requesting a refund clears the coin accepter and binds the value 0 to the `amount` variable; for simplicity we have ignored the issue of giving back change. Since constraint evaluation and state mutation associated with the same message is done as an atomic action, only one slot can be opened before the coin accepter is cleared.

[2] In practice, the constraint in Figure 4 would be parameterized with an arbitrary set of slot objects that has an associated price. We have ignored such parameterization issues in order to focus on the essential aspects of multi-object constraints. Note that price changes could easily be incorporated by having the synchronizer observe `change_price` requests sent to such slot objects.

Without synchronizers it would be necessary to explicitly implement the consensus protocol between the slot objects so that only one slot can be opened. Similar to the cooperating resource administrators, the consensus protocol could be implemented centrally with a coordinator, distributed by explicit message-passing, or some hybrid scheme could be used. In all cases, explicit implementation would require construction of complicated and inflexible code. □

The vending machine example illustrates an important real-world property: the behaviors of objects depend on the context in which they exist. Synchronizers allow us to express the contextual constraints of a part-whole hierarchy as an aspect of the *hierarchy*, not an aspect of the parts. With synchronizers, it is easier to reason about context constraints since they are expressed more directly. Furthermore, it is possible to exchange parts without affecting the consistency constraint of the containing hierarchy.

## 4 Customized Access Schemes

In this section we present constraints that are defined and enforced by clients that access a number of shared objects. Such client based constraints express customized access schemes utilized by individual clients; they are temporary in nature, enforced for a specific set of invocations issued by one or more clients. Synchronizers give a flexible and expressive model for describing access restrictions such as atomicity, mutual exclusion and temporal ordering. We use the dining philosophers problem to illustrate constraints used as access restrictions.

**Example 3** (*Dining Philosophers*). The dining philosophers is a classical example from the literature on concurrent systems. A group of philosophers (processes) need to coordinate their access to a number of chopsticks (resources). The number of chopsticks and the number of philosophers is the same and a philosopher needs two chopsticks in order to eat. Access to the chopsticks needs to be regulated in order to avoid deadlock and starvation. The constraint that each philosopher picks up two chopsticks atomically ensures absence of deadlock. Note that this multi-object constraint does not explicitly guarantee freedom from starvation; that condition may be ensured by fairness in the implementation of synchronizers.

```
PickUpConstraint(c1,c2,phil)
{
   atomic( (c1.pick(sender) where sender = phil),
           (c2.pick(sender) where sender = phil) ),
   (c1.pick  where sender = phil) stops
}
```

**Fig. 5.** A synchronizer giving atomic access of chopsticks.

Each philosopher has an `eat` method that is called when the philosopher is hungry. The `eat` method attempts to pick up two chopsticks by sending a `pick` message to the two chopsticks. The `eat` method is parameterized by the two chopsticks that must be picked up in order to eat. The `eat` method instantiates a synchronizer with a structure similar to the synchronizer depicted in Figure 5. The synchronizer is parameterized with the two chopsticks (`c1` and `c2`) and the philosopher who accesses the chopsticks (`phil`). The synchronizer only applies to `pick` messages sent by the philosopher referred by the name `phil`. We assume that the `eat` method concurrently invokes `pick` on each of the needed chopsticks. The instantiated synchronizer guarantees that the access to the chopsticks occurs atomically. When both chopsticks have been acquired, the eat method starts using the chopsticks and releases them when done eating. When the philosopher has successfully acquired both chopsticks, the constraint is terminated.

A chopstick can only be picked up by one client (philosopher) at a time. The local synchronization constraints of a chopstick will disable the `pick` method after a successful pick up. When the `put` method is invoked, `pick` is enabled again. Note that a chopstick may be subject to additional local synchronization constraints. One local constraint could be that a chopstick is washed after being put down. While being washed, a chopstick cannot be picked up. Conventional methods for solving the dining philosophers problem cannot be combined with such local constraints in a modular fashion. □

Synchronizers support a wide range of access restrictions. Example 3 focussed on atomicity. Other examples include exclusive access and temporal ordering imposed on sets of invocations. For example, it is possible to describe exclusive choice where clients issue a number of requests under the constraint that only one of the requests are processed by the receiver.

As the above example shows, synchronizers allow access restrictions to be specified *separate* from the accessed objects. Separation makes it possible to access groups of objects in ways that were not anticipated when the group was instantiated. Thus, synchronizers provide an extensible way of describing access constraints.

## 5 Related Work

In existing languages and systems, client enforced access restrictions can be specified using synchronous communication, transactions or atomic broadcasts. The temporal constraints of synchronizers are more abstract and expressive than the temporal constraints expressible by synchronous communication. Transactions provide atomicity in a number of systems [LS82, WL88, DHW88, KHPW90, GCLR92, WY91]. As we have already mentioned, transactions cannot be combined with temporal constraints. Furthermore, the atomicity provided by the atomic operator is cheaper to implement. It should be emphasized that we do not perceive synchronizers as a substitute for transaction systems: we think of synchronizers as a structuring tool that can supplement transactions in an orthogonal and non-interfering way.

A number of systems provide broadcasts with different delivery guarantees. Examples are ISIS [BJ87], Consul [MPS91] and the notion of Interface Groups proposed

in [OOW91]. In such systems, it is possible to ensure certain properties for the reception of broadcast messages. None of the broadcast systems deal with objects that have local synchronization constraints: the guarantees provided are relative to delivery, not acceptance.

Neither synchronous communication, transactions or broadcast protocols can be used to describe constraints permanently associated with object groups. In all three cases, the constraints expressible are associated with individual clients.

The constraints of Kaleidoscope [FBB92] capture numerical relations between instance variables of multiple objects. Kaleidoscope emphasizes state-consistency by propagating updates. In contrast, synchronizers express invocation consistency by request scheduling. Synchronizers and the constraints of kaleidoscope capture fundamentally different aspects of multi-object consistency.

The constraints of Procol [vdBL91] and RAPIDE [LVB+92] provide triggering of activities based on observation of events. The constraints of Procol and RAPIDE can only observe, not limit invocations.

Several attempts have been made at documenting multi-object coordination at an abstract level. In [Ara91], a temporal logic framework is used for assertions about the message passing behavior of objects. The notion of a *contract* is proposed in [HHG90, Hol92]. Contracts extend the notion of type to capture the message passing obligations fulfilled by objects. In [CI92] a mix of path-expressions, data-flow and control-flow is used to capture coordination between objects comprising a sub system. The above approaches to multi-object interactions *describe* rather than *prescribe* multi-object coordination. The expression of multi-object coordination serves to document and verify the coordination behavior of a set of interacting objects. The coordination behavior is realized by traditional means such as explicit message passing between the objects. In contrast to the above approaches, synchronizers are executable specifications of multi-object coordination.

In the Dragoon language [RdPG91, AGMB91], synchronization constraints are specified external to the constrained objects. Constraints are general specifications that can be mixed-in with class definitions yielding a class with synchronization constraints. The common factor between Dragoon and synchronizers is that coordination and computation are two distinct aspects that are combined in a flexible way. A major difference is that the Dragoon specifications of coordination only cover single objects.

## 6 Discussion

We have introduced synchronizers as tools for coordinating access to multiple objects. Compared to existing schemes where multi-object constraints are implemented by explicit communication, synchronizers provide abstraction, modularity and integration with synchronization constraints. Modularity enables a compositional approach to multi-object coordination: multiple synchronizers can be imposed on the same set of constrained objects, composing multiple constraint specifications. Consequently, synchronizers support system design that emphasizes physical separation of logically distinct aspects.

It should be emphasized that we do not consider synchronizers the complete answer to the challenge of describing multi-object coordination. The primary purpose

of this paper is to promote constraints as a means of describing multi-object coordination. Furthermore, the principles presented in this paper only constitute the core of a language for describing multi-object constraints. Due to space limitations we have focused on the essential principles. For example, in practice it might be desirable for synchronizers to be able to receive messages as well as trigger new activities. The ability to trigger activities would provide an elegant way of extending the vending machine in Example 2 to give back change: successful opening of a slot triggers a refund of the money remaining in the accepter.

In order to gain more experience with the proposed concepts, we are currently developing a prototype implementation and semantic foundation for synchronizers. We are experimenting with different protocols for constraint evaluation and enforcement. An interesting aspect that we are investigating is the price paid for composition and fairness. Synchronizers can be implemented in numerous ways, e.g. purely distributed, with a central coordinator or some hybrid. Which implementation to choose depends on application specific characteristics. Therefore, it may be necessary to extend the language so that the choice of implementation is guided to a certain extent by the application programmer. One possible model for customization of implementation is to use reflection along the lines pointed out in [YMFT91, AFPS93]. Synchronizers make it feasible to customize the implementation of multi-object coordination since the coordination protocols are no longer hard-wired into applications but specified separately.

## Acknowledgments

The first author is supported in part by the Danish Research Academy and a research fellowship from the Natural Science Faculty of Århus University in Denmark.

The research described in this paper was carried out at the University of Illinois Open Systems Laboratory (OSL). The work at OSL is supported by grants from the Office of Naval Research (ONR contract number N00014-90-J-1899), Digital Equipment Corporation, and by joint support from the Defense Advanced Research Projects Agency and the National Science Foundation (NSF CCR 90-07195).

The authors wish to thank Carolyn Talcott and Shingo Fukui as well as past and present members of OSL for reading and commenting on manuscript versions of this paper.

## References

[AB92] M. Aksit and L. Bergmans. Obstacles in Object-Oriented Software Development. In *Proceedings OOPSLA 92*. ACM, October 1992.

[AC93] G. Agha and C.J. Callsen. ActorSpace: An Open Distributed Programming Paradigm. In *1993 ACM Conference on Principles and Practice of Parallel Programming (PPOPP)*, 1993. (To be published).

[AFPS93] G. Agha, S. Frølund, R. Panwar, and D. Sturman. A Linguistic Framework for Dynamic Composition of Dependability Protocols. In *Dependable Computing for Critical Applications III*. International Federation of Information Processing Societies (IFIP), Elsevier Scienc Publisher, 1993. (To be published).

[Agh86] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, 1986.

[AGMB91] C. Atkinson, S. Goldsack, A. D. Maio, and R. Bayan. Object-Oriented Concurrency and Distribution in DRAGOON. *Journal of Object-Oriented Programming*, March/April 1991.

[Ara91] C. Arapis. Specifying Object Interactions. In D. Tsichritzis, editor, *Object Composition*. University of Geneva, 1991.

[BJ87] K. P. Birman and T. A. Joseph. Communication Support for Reliable Distributed Computing. In *Fault-tolerant Distributed Computing*. Springer-Verlag, 1987. LNCS.

[CI92] R. H. Campbell and N. Islam. A Technique for Documenting the Framework of an Object-Oriented System. In *Proceedings of the Second International Workshop on Object Orientation in Operating Systems*, September 1992.

[DHW88] D. L. Detlefs, M. P. Herlihy, and J. M. Wing. Inheritance of Synchronization and Recovery Properties in Avalon/C++. *IEEE Computer*, 21(12):57–69, December 1988.

[FBB92] Bjorn N. Freeman-Benson and Alan Borning. Integrating Constraints with an Object-Oriented Language. In O. Lehrmann Madsen, editor, *Proceedings ECOOP '92*, LNCS 615, pages 268–286, Utrecht, The Netherlands, July 1992. Springer-Verlag.

[Frø92] Svend Frølund. Inheritance of Synchronization Constraints in Concurrent Object-Oriented Programming Languages. In O. Lehrmann Madsen, editor, *Proceedings ECOOP '92*, LNCS 615, pages 185–196, Utrecht, The Netherlands, July 1992. Springer-Verlag.

[GCLR92] Rachid Guerraoui, Riccardo Capobianchi, Agnes Lanusse, and Pierre Roux. Nesting Actions through Asynchronous Message Passing: the ACS Protocol. In O. Lehrmann Madsen, editor, *Proceedings ECOOP '92*, LNCS 615, pages 170–184, Utrecht, The Netherlands, July 1992. Springer-Verlag.

[HHG90] Richard Helm, Ian M. Holland, and Dipayan Gangopadhyay. Contracts: Specifying Behavioural Compositions in Object-Oriented Systems. In *Proceedings OOPSLA/ECOOP '90*, pages 169–180, October 1990. Published as ACM SIGPLAN Notices, volume 25, number 10.

[Hol92] Ian M. Holland. Specifying Reusable Components Using Contracts. In O. Lehrmann Madsen, editor, *Proceedings ECOOP '92*, LNCS 615, pages 287–308, Utrecht, The Netherlands, July 1992. Springer-Verlag.

[KHPW90] G. E. Kaiser, W. Hseush, S. S. Popovich, and S. F. Wu. Multiple Concurrency Control Policies in an Object-Oriented Programming System. In *Proceedings of the Second Symposium on Parallel and Distributed Processing, Dallas Texas.* IEEE, December 1990.

[LS82] Barbara Liskov and Robert Scheifler. Guardians and Actions: Linguistic Support for Robust, Distributed Programs. In *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages*, pages 7–19, Albuquerque, New Mexico, January 1982. ACM.

[LVB+92] D. C. Luckham, J. Vera, D. Bryan, L. Augustin, and F. Belz. Partial Orderings of Event Sets and Their Application to Prototyping Concurrent Timed Systems. In *Proceedings of the 1992 DARPA Software Technology Conference*, April 1992.

[MPS91] S. Mishra, L. L. Peterson, and R. D. Schlichting. Consul: A Communication Substrate for Fault-Tolerant Distributed Programs. Technical report, University of Arizona, Tucson, 1991.

[MWBD91] C. McHale, B. Walsh, S. Baker, and A. Donnelly. Scheduling Predicates. In M. Tokoro, O. Nierstrasz, and P. Wegner, editors, *Object-Based Concurrent Computing*, pages 177 – 193. Springer-Verlag, July 1991. LNCS 612.

[Neu91] Christian Neusius. Synchronizing Actions. In P. America, editor, *Proceedings ECOOP '91*, LNCS 512, pages 118–132, Geneva, Switzerland, July 1991. Springer-Verlag.

[Nie87] Oscar Nierstrasz. Active Objects in Hybrid. In *Proceedings OOPSLA '87*, pages 243–253, December 1987. Published as ACM SIGPLAN Notices, volume 22, number 12.

[OOW91] M. H. Olsen, E. Oskiewicz, and J. P. Warne. A Model for Interface Groups. In *Tenth Symposium on Reliable Distributed Systems*, Pisa, Italy, 1991. IEEE.

[RdPG91] Stefano Crespi Reghizzi, Guido Galli de Paratesi, and Stefano Genolini. Definition of Reusable Concurrent Software Components. In P. America, editor, *Proceedings ECOOP '91*, LNCS 512, pages 148–166, Geneva, Switzerland, July 1991. Springer-Verlag.

[TS89] Chris Tomlinson and Vineet Singh. Inheritance and Synchronization with Enabled Sets. In *Proceedings OOPSLA '89*, pages 103–112, October 1989. Published as ACM SIGPLAN Notices, volume 24, number 10.

[vdBL91] J. van den Bos and C. Laffra. PROCOL, a Concurrent Object-Oriented Language with Protocols Delegation and Constraints. *Acta Informatica*, 28:511 – 538, 1991.

[WL88] C. T. Wilkes and R. J. LeBlanc. Distributed Locking: A Mechanism for Constructing Highly Available Objects. In *Seventh Symposium on Reliable Distributed Systems*, Ohio State University, Columbus, Ohio, 1988. IEEE.

[WY91] K. Wakita and A. Yonezawa. Linguistic Supports for Development of Distributed Organizational Information Systems in Object-Oriented Concurrent Computation Frameworks. In *Proceedings of the First Conference on Organizational Computing Systems, Atlanta Georgia*. ACM, September 1991.

[YMFT91] Y. Yokote, A. Mitsuzawa, N. Fujinami, and M. Tokoro. Reflective Object Management in the Muse Operating System. In *Proceedings of the 1991 International Workshop on Object Orientation in Operating Systems*, Palo Alto, California, October 1991.

This article was processed using the LaTeX macro package with LLNCS style