Coordination Control with BCOOPL

Hans de Bruin
Vrije Universiteit, Mathematics and Computer Science Department
De Boelelaan 1081a
1081 HV Amsterdam, The Netherlands
hansdb@cs.vu.nl

Keywords
Component-based development, Design patterns, Language support for coordination control, Software architecture

ABSTRACT
This paper introduces BCOOPL (Basic Concurrent Object-Oriented Programming Language), a language specifically designed to support component-oriented programming. BCOOPL is more than just a programming language. It can also be seen as a design language with which high level architectural elements, like software components and connectors, can be specified. BCOOPL is centered around two concepts: interfaces and patterns. Operations to be implemented in objects are specified in an interface using an augmented regular expression notation, not only detailing when a specific operation may be invoked, but also detailing the parties that are allowed to do so. Object behavior is defined separately with so called patterns. BCOOPL encourages the design of high-quality software components through high level OO abstractions including the built-in support of the Observer and Mediator design patterns. A key characteristic of the Observer design pattern is that it reduces the coupling between objects. An object provides its services by issuing notifications without being aware of the clients that actually use the services. It is up to the clients to link to these notifications in order to receive them. The notifications of various objects can be synchronized with patterns, which are specified using the same regular expression notation as for interfaces. An object in this role can be seen as a mediator controlling the interactions between independently operating objects. As a result, component behavior is decoupled from their interactions with other components, which is a prerequisite for system adaptations and reusability. The expressive and descriptive power of BCOOPL is demonstrated in an extended example in which a solution is given for a real-time, process control problem.

1. INTRODUCTION
In the design of software systems, we search for qualities like adaptability and understandability. These goals are not easily achieved, especially if large and complex applications are concerned. High level building blocks such as software components and connectors give us handles to tame the complexity involved in designing well-balanced systems. The underlying design principle is to separate structural descriptions of component interactions from behavioral descriptions of individual components. If applied correctly, this principle ensures the specification and implementation of flexible systems that are amenable for analysis and adaptation.

In this paper, we show that the Basic Concurrent Object-Oriented Programming Language (BCOOPL) [7] is well-suited for modeling components and connectors with which complex software systems exhibiting intricate interaction patterns can be specified and implemented. BCOOPL is a small language specifically designed to support component-oriented programming. Its roots can be traced back to path expressions [6], and the concurrent object-oriented programming languages Procol [13] and Talktalk [5].

BCOOPL integrates a number of unusual language features, which are particularly useful for applying abstraction principles:

Separation of interface and implementation An interface defines the operations that must be implemented by an object that conforms to that interface. By adhering to the principle of programming to an interface, a certain amount a flexibility is added to a system since new implementations can be provided without breaking existing code. A BCOOPL interface is specified as an augmented regular expression over operations. It not only describes how an operation can be invoked, but also when and by whom.

Weakly-coupled objects BCOOPL offers direct support for the Observer design pattern [8]. An object provides service offerings by means of issuing notification messages, which are specified in its interface. Other objects can subscribe to these notifications and be informed when the publishing object issues them. The use of notifications is the key to realize weakly-coupled objects because the coupling between objects is one-way only. That is, the subscribers have to adhere to the interface of the publisher in order to receive notifications, whereas the publisher does not know its subscribers consciously, it just sends notifications to all subscribers. By using the notification mechanism, stand-alone components can be created that can be subjected to composition to form a system.

SAC 2001, Las Vegas, NV

Component interaction control Stand-alone components must be configured in such a way that the resulting system exhibits the intended functionality. This is achieved in BCOOPL by specifying objects in the role of mediator that control the interactions between objects (see also the Mediator design pattern [8]). As discussed before, BCOOPL interfaces are specified as augmented regular expressions over operations, which have the same expressive power as transition networks. The same regular expression notation is used for object behavior specifications, which are called patterns[1]. Such a specification can actually be seen as a configuration specification, it details how components interact at run-time.

Other language features include concurrency and delegation. BCOOPL supports the homogeneous object model where each object is considered as an active, concurrent entity that may be distributed and moved around over a number of processors. Delegation is used as alternative to implementation (e.g., class) inheritance for sharing code. It is not only used for stimulating black-box composition, but also for maximizing the amount of concurrency in a system. That is, an object may decide to delegate an operation to another object and be ready to accept new requests.

This paper is organized as follows. We start with a brief introduction of BCOOPL's language constructs and computational model. This is followed by an extended example in the realm of real-time, process control systems. The purpose of this example is to show in detail how BCOOPL's language features supports the design of such systems. A high level, software architecture is devised for the process control system, which is elaborated further into an implementation using BCOOPL's interfaces and patterns. We end with a discussion of related work.

2. A BRIEF OVERVIEW OF BCOOPL

A brief introduction of BCOOPL is given in this section. It only covers the basics required to understand the extended example given in a next section. A detailed account of BCOOPL can be found in [7], which also addresses implementation issues. The main principle that guided the design of BCOOPL was, and still is, to keep the language small by supporting only a few but powerful language constructs. As a result, it does not directly support features like real-time processing and exception handling, but instead it offers an object model with which extensions can be defined in terms of the small set of language constructs. Several extension mechanisms may coexist. The common ground that holds them together is the object model, which is comprised of weakly-coupled, concurrent objects whose external behavior is specified in interfaces.

2.1 Core Language Features

BCOOPL is centered around two concepts: patterns and interfaces. The concept of classes and methods specification have been unified in patterns and sub-patterns. The term pattern has been borrowed from the object-oriented programming language Beta [11]. The idea is that objects are instantiated from patterns and behave according to the pattern definition. A pattern describes the allowed sequences of primitives to be executed by an object after a message has been received in a so called *inlet*, which is implicitly defined in a pattern definition. A pattern may contain sub-patterns which also define inlets, and so on. A top-level pattern can be seen as a class definition, whereas sub-patterns can be seen as (sub-)method definitions.

A notification pattern is part of a pattern definition. It specifies the output behavior of a pattern in terms of notifications. An object interested in a particular notification of a publishing object can subscribe to that notification. The subscription information is comprised of, amongst others, the name of the notification, the identity of the subscriber and the pattern to be invoked in the subscriber. Notifications are issued through an *outlet* by means of a *bang-bang* (!!) primitive. As a matter of fact, notifications are not only used for implementing the Observer design pattern, but they are also used for getting a reply value as a result of sending a request to some object. The basic idea is to send a message to an object and then wait for a notification to be received in an inlet following the send primitive. The concept of notification patterns has been explored in Talktalk [5].

The type or types of a pattern are provided by interfaces. A pattern that implements an interface has the type of that interface. As in Java, multiple interface inheritance is supported in BCOOPL. That is, an interface may extend one or more sub-interfaces. In contrast to Java, interfaces contain sequence information specifying when a pattern may be invoked and by whom.

2.2 Computational Model

The computational model of BCOOPL is based on message passing and concurrent objects. Objects are instantiated from patterns by executing the *new* primitive. A pattern may contain sub-patterns and their corresponding objects are instantiated implicitly whenever a super-pattern is instantiated. A conceptual model of an object and its sub-objects is shown in Figure 1. Each object has an unique I.D., which is used as an address for message exchange. Objects communicate with other objects by means of a restricted form of asynchronous message passing in the sense that the partial ordering of messages sent from one object to another is preserved. An object receiving a message does not process the message right away, instead the message is placed in an unbounded message buffer. The dispatcher searches the message buffer on a *first-come-first-served* basis of acceptable messages. The acceptability of a message is determined by the state of patterns in execution. If an acceptable message is found, the dispatcher passes the message on to the corresponding (sub-)object's inlet, otherwise it waits for the arrival of new messages. Thus, the communications between objects can be summarized as synchronous message buffering, but asynchronous message processing.

A computation in BCOOPL is achieved by sending messages. This includes control flow structures like selection (*if-then-else*) and repetition (*while-do*). A computation proceeds as follows. After a message has been received in an inlet, the sequence of primitives following the inlet are executed until one or more inlets (sub-patterns) are encountered. Regular expression operators, such as the selection (+) and the repetition (*), imply choices. Each branch resulting from such a choice must be guarded with an inlet. That is, the choice to follow a particular branch is made by sending an appropriate message. There is no such concept

[1]A pattern in BCOOPL should not be confused with a design pattern. The latter provides a solution for a design problem within a given context.

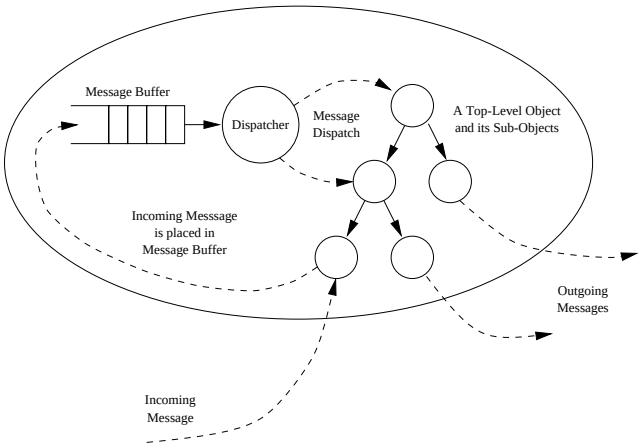

Figure 1: Object model.

as non-deterministic choices.

Within an object and its sub-objects the *one-at-a-time* principle of executing primitives applies. Multiple execution threads may occur within a tree of (sub-)objects, which are introduced with the interleave operator (||). At most one thread, however, is active at any one time. Thread switching occurs at the time the active thread runs into one or more sub-patterns. Because almost every computation step is expressed in terms of message passing, thread switches occur frequently, which amounts to a semi-parallel object model. The next active thread is selected on the basis of the acceptability of the pending messages in the message buffer. This is a fair scheduling mechanism since the active thread cannot interfere with the scheduling of other threads, unless it claims explicitly the exclusive ownership of the object by means of the synchronize operator ($\ll expr \gg$). In contrast to intra-object concurrency, top-level objects (and their sub-objects) operate on a truly concurrent basis.

2.3 Interface and Pattern Specification

An interface is identified by a name and may extend one or more base interfaces. It is defined by means of an *interface interaction term*.

interface *Interface Name*
  extends $[interfaces]_{opt}$
defines [
  interface interaction term
]$_{opt}$

An interface interaction term is specified using the following syntax:

client specifications $\mapsto$
  Pattern Name (*input arguments*) $\Rightarrow$ (*notification pattern*) [
  regular expression over interface interaction terms
]$_{opt}$

An interface interaction term corresponds with a (sub-)pattern definition that implements the interface. It defines the pattern name, the formal input arguments, a notification pattern that specifies sequences of notification messages, and client specifications. An interface interaction term is recursively defined as a regular expression over interface interaction terms leading to a hierarchical interface specification. The regular expression operators used for constructing an interface and their meaning are summarized in Table 1.

Expression	Operator	Meaning
$\ll$ E $\gg$	synchronize	E is executed uninterupted
E $\|\|$ F	interleave	E and F may occur interleaved
E + F	selection	E or F can be selected
E ; F	sequence	E is followed by F
E $*$	repetition	Zero or more times E
E $[m, n]$	bounded rep.	i times E with $m \leq i \leq n$

Table 1: Semantics of regular expression operators.

Client specifications denote the parties that are allowed to invoke the corresponding pattern. They are defined by any combination of the following: by interface name, by interface name set (specified with the @ modifier), or by object reference set (specified with the $ modifier). The sets are used to dynamically specify the clients that are allowed to interact. A pattern implementing such an interface is responsible for the contents of a particular set.

Notifications issued by a pattern are guaranteed to be emitted according to the defined sequences specified in its notification pattern. A notification pattern is defined as a regular expression over notification terms that are specified as follows:

Notification Name (output args)

The co- and contra-variance rules apply for specifying interfaces. An interface interaction term may be redefined in a derived interface. The types of the input arguments must be the same as or generalized from the argument types of the base interface (i.e., contra-variance rule). In contrast, a notification pattern may be extended in a derived interface, both in terms of notification output arguments having derived interfaces (i.e., co-variance rule) and additional notifications.

The interface *Any* acts as a base type for every other interface. That is, every interface extends *Any* implicitly. *Any* is defined as:

interface Any

As an example of interface specification, consider the interface for a bounded buffer.

```
interface BoundedBuffer defines [
    Any ↦ (maxSize : Integer) ⇒ () [
        (
            NotFull : @Any ↦ put (anObject : Any) ⇒ (done()) +
            NotEmpty : @Any ↦ get () ⇒ (value(anObject : Any))
        ) *
    ]
]
```

It is assumed that in the implementation of a bounded buffer the interface set *NotFull* contains the type *Any* when the buffer is not full, otherwise the set is empty. Therefore, the *put* pattern can only be invoked in the case that the buffer is not full. Likewise, the interface set *NotEmpty* is supposed to contain *Any* if the buffer is not empty. The *done* notification of the *put* pattern is issued to indicate that an object has been stored in the buffer. By the same token, the *get* pattern issues a *value* notification with which an object from the buffer can be obtained.

Patterns and sub-patterns are defined identically. The overall structure of a pattern is the following:

```
client/server specifications ↦ pattern Pattern Name
                (input arguments) ⇒ (notification pattern)
    implements [interfaces]opt
    declares [local variables]opt
does [
    pattern implementation;
    a regular expression over primitives and sub-patterns
]
```

Client/server specifications are defined in patterns similar to client specifications in interfaces. Client specifications identify the kind of objects that may communicate with a pattern, whereas server specifications denote declaratively specified linkages to notifications. The syntax for server specifications is as follows.

```
Local Variable.Pattern Name1.Pattern Name2.· · · .
            Pattern Namen.Notification Name(formal arguments)
Object Set.Pattern Name1.Pattern Name2.· · · .
            Pattern Namen.Notification Name(formal arguments)
```

Notification linkages are established at run-time. An assignment to a variable involved in notification linkage results in first abolishing the current link, provided the variable is not *nil*, then assigning to the variable, and finally establishing a new link if the variable does not equal *nil*. In the case of an object set, a notification link is established when an object is added to the set, likewise it is abolished when the object is removed from the set.

The behavior of a pattern is defined in the *does* section. It is defined as a regular expression over primitives and sub-patterns. The supported primitives are summarized in Table 2. For example, a rudimentary implementation of the *BoundedBuffer* is given below. It uses pseudo-code (shown in slanted font) for control flow, arithmetic expressions, and for storing and retrieving objects in and from a container.

```
Any ↦ pattern boundedBuffer (maxSize : Integer) ⇒ ()
    implements [ BoundedBuffer ]
    declares [
        size : Integer ;
        NotFull, NotEmpty : @Any ;
    ]
does [
    // initialization code
    size := 0 ;
    @NotFull.add(Any) ;

    // put and get operations
    (
        @NotFull ↦ pattern put (anObject : Any) ⇒ (done())
        does [
            store anObject in some container
            @NotEmpty.add(Any) ;
            size := size + 1 ;
            if size == maxSize then @NotFull.remove(Any) ;
            !! done()
        ] +
        @NotEmpty ↦ pattern get () ⇒ (value(anObject : Any))
        does [
            remove anObject from some container
            @NotFull.add(Any) ;
            size := size - 1 ;
            if size == 0 then @NotEmpty.remove(Any) ;
            !! value(anObject)
        ]
    ) *
]
```

Notice that by protecting the *put* and the *get* patterns with guards (i.e., interface name sets) these two patterns can be executed only if they are guaranteed to succeed. This should be contrasted with monitors where conditions are tested and set inside an operation, typically by means of a *wait* and a *signal* primitive.

A pattern returns results by means of issuing notifications. For instance, in the *get* pattern an object is returned by issuing the *value* notification with *anObject* as argument. In principle, the notification is sent to all objects that have expressed their interest in this particular notification, which might include the object that sent the *get* message in the first place. This is probably not the required behavior in this situation. By prefixing the send with the keyword *request*, the notification is only sent to sub-objects of the object that invoked the request, provided that sub-objects have expressed their interest in the notification. Thus, to get a value from a bounded buffer, one would write:

```
    .
    .
request myBuffer.get() ;
myBuffer.get.value(Any) ↦ pattern getIt (myObject : Any) ;
    .
    .
```

Observe the decoupling from the request and the reply in the form of a notification. An object is free to perform other actions in between a request and a reply, which might increase the amount of concurrency in a system.

3. EXTENDED EXAMPLE: REAL-TIME, PROCESS CONTROL SYSTEM

The example presented in this section demonstrates BCOOPL's fitness for tackling real-time, process control problems. The purpose of this example is to show how a high level design, in the form of software architecture, can be systematically translated into a detailed design and an implementation. In particular, we show how software architectural elements like components and connectors can be modeled and realized with BCOOPL's interfaces and patterns. Moreover, we demonstrate how a system comprised of BCOOPL components can be refined to meet fault-tolerance requirements without actually changing the basic functionality that assumes an ideal, error-free environment.

In the design of the system, we make a sharp distinction between stand-alone components that provide reusable functionality and control components that mediate interactions between the former components. As will turn out in the example, the augmented regular expression notation provides an expressive and concise notation for expressing explicit interaction sequences amongst components at the interface level as well as the implementation level.

The example has been taken from one of Booch's books on Ada [4, chapter 18]:

> There exists a collection of ten independent sensors that continually monitor temperatures. We may explicitly enable or disable a particular sensor, and we may also force its status to be recorded. Furthermore, we may set the lower and upper limits of a given sensor. In the event that any of the enabled sensors register an out-of-limit value, the system must immediately post an alarm condition. Additionally, it must request and record the status of all the sensors every fifteen minutes (set by a timer hardware

Primitive	Abstract Syntax	Remarks
Assignment	*variable* := FQN*expression*	A FQN (Fully Qualified Name) denotes an object. It is specified as: *(Pseudo-)Variable.Pattern Name*$_1$.$\cdots$.*Pattern Name*$_n$
New	**new***Pattern Name*	The designated pattern is instantiated along with its sub-patterns resulting in an object tree. Unreferenced objects are reclaimed by a garbage collector.
Send	FQN (*message args*)	A message is sent to the object designated with the FQN
Request	**request** FQN (*message args*)	Identical to a send with the exception that a reply (i.e., a notification) is sent only to a sub-object of the object that issued the request.
Inlet (Pattern)	*CS specs* $\mapsto$ **pattern** *Name* (*in args*) $\Rightarrow$ (*notifications*) **does** [$\cdots$]	A message is received in an inlet which is implicitly defined in a pattern.
Lightweight Inlet (Pattern)	*CS specs* $\mapsto$ **pattern** *Name* (*vars*)	In contrast with an ordinary inlet, a lightweight inlet does not introduce a local scope. The received message arguments are stored in the designated variables.
Outlet	!!*Notification Name* (*message args*)	A notification is issued.
Client/Server Set Operations	**add**(*element*) and **remove**(*element*)	The add and the remove are currently the only supported operations.
Delegate	*beyond the scope of this paper*	

Table 2: Primitives.

interrupt). If we do not get a response from any sensor within five seconds after this time, we must assume that the sensor is broken and immediately post another alarm. Asynchronously, we may get a user command to enable or disable a specific sensor, set the temperature limits, or force the status of a given sensor to be recorded. In any case, failure of the user interface must not affect the monitoring of any currently enabled sensors.

3.1 Software Architecture

As a first step in finding a solution for the process control problem we devise a software architecture. The key components are readily found from the problem statement: sensor manager, sensor, recording device, alarm, timer and user interface. Next we decide how these components interact and to choose appropriate connectors that can realize the intended interactions. A high level conceptual architecture, which is comprised of components and connectors, is shown in Figure 2. The notation is borrowed from Hofmeister et al. [10], which is a sugared version of UML and extends the real-time, object-oriented modeling technique ROOM [12]. Components communicate with each other as agreed upon in a protocol. Ports are provided by components as hooks to which connectors can be attached for inter-component message exchange. The connectors are responsible for abstracting away communication peculiarities between components.

Connection details are given in Table 3. The connections to the recording device, alarm, timer, and user interface are assumed to be reliable. This in contrast to the connections to the sensors, which are placed remote from the computer system on which the sensor manager is running. Not only the sensors can fail due to hostile environmental conditions, but also the physical connections (e.g., cables) between the sensor manager and a sensor. For this reason, a timeout facility will be built in the connector to detect the unresponsiveness of sensors. Obviously, the timeout facility is located at the sensor manager site to detect not only sensor failures, but physical connection failures as well.

3.2 BCOOPL Interfaces and Patterns

We are now in the position to translate the conceptual architecture into BCOOPL interfaces and patterns. A couple of interfaces are used whose specifications are postponed until the detailed design phase. In particular, sensor status information is encapsulated in objects that conform to the *Status* interface.

```
interface Status defines [
    .
    . // interface for setting and getting Sensor status information
]
```

The interface for a *Sensor* is defined below. A sensor's status can be obtained by invoking the *getStatus* pattern. Alternatively, a sensor emits a *periodicUpdate* notification after a specified elapsed time interval or it emits a *changedUpdate* notification indicating that the status has been changed. When an enabled sensor senses an out-of-limits value, an *outOfLimits* notification is issued. These notifications should be seen as service offerings. It is up to the client of a sensor to link to them.

```
interface Sensor defines [
    Any ↦ () ⇒
          ((periodicUpdate(status : Status) +
              changedUpdate(status : Status) +
              outOfLimits(status : Status)) *) [
        (
            Any ↦ setLimits (lower : Float, upper : Float) ⇒ () +
            Any ↦ setTimeInterval (seconds : Float) ⇒ () +
            Any ↦ enable () ⇒ () +
            Any ↦ disable () ⇒ () +
            Any ↦ getStatus () ⇒ (value(status : Status) + failure())
        ) *
    ]
]
```

The task of a recording device is to record status information that has been obtained from the sensors. The interface specification is quite simple. It basically states that status information can be recorded periodically. This is a good example of one of BCOOPL's key concept to separate the interface from an implementation. For instance, one implementation can write status information to a printer, while another one can employ a tape drive. The actual recording device that is being used is unimportant because its use has been abstracted away in the *RecordingDevice* interface.

```
interface RecordingDevice defines [
    Any ↦  () ⇒ () [
        Any ↦ record (status : Status) ⇒ () *
    ]
]
```

The interfaces for the alarm and timer are specified as shown below. Again, many implementations can be given that conform to these interfaces.

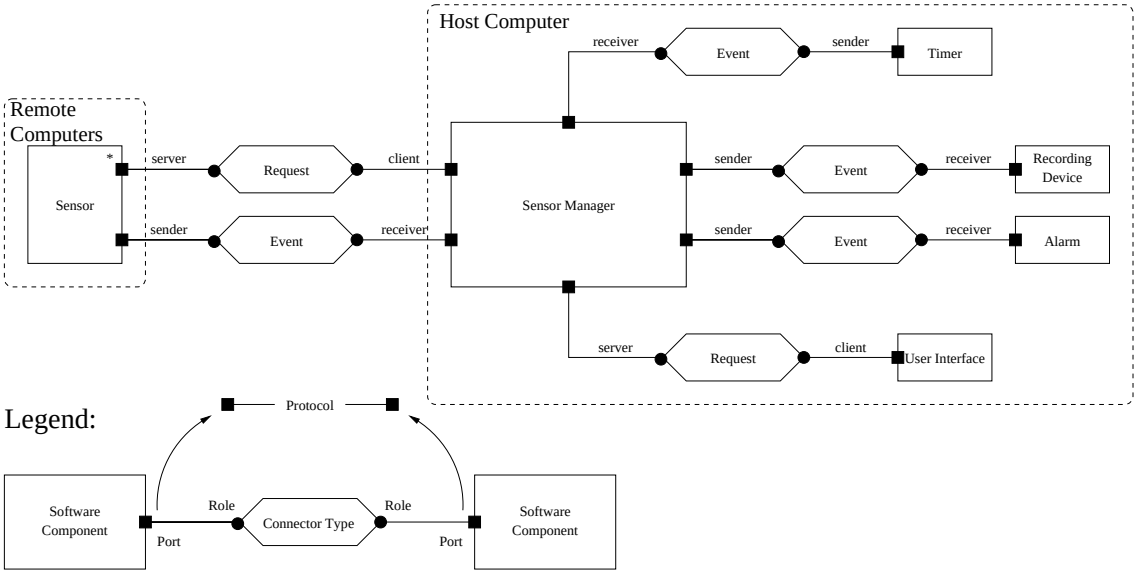

Figure 2: A conceptual architecture for the process control system.

Sensor Manager connected to	Connection Type	Characteristics	Description
Sensor	Synchronous request initiated by the Sensor Manager with failure detection by means of time-outs. This connection must be programmed explicitly in BCOOPL.	Unreliable	Since the sensors are located in a potentially hostile environment, there is a high risk that a sensor or a physical connection fails.
Sensor	Asynchronous multicast (BCOOPL notification mechanism) issued by a Sensor used for periodic update and out-of-limits events.	Unreliable	Even though the connection is unreliable, no special error recovery measures are taken. This is already taken care of by the aforementioned time-out facility.
Recording Device	Asynchronous multicast (BCOOPL notification mechanism) issued by the Sensor Manager.	Reliable	It is assumed that the connection operates reliable, so no special error recovery measures are taken.
Alarm	Asynchronous multicast (BCOOPL notification mechanism) issued by Sensor Manager.	Reliable	No special measures required (see remarks above).
Timer	Asynchronous multicast (BCOOPL notification mechanism) issued by Timer.	Reliable	No special measures required (see remarks above).
User Interface	Asynchronous request issued by the User Interface. A reply, if any, issued with BCOOPL's notification mechanism.	Reliable	No special measures required (see remarks above).

Table 3: Connection characteristics.

```
interface Alarm defines [
    Any ↦ () ⇒ () [
        (
            Any ↦ soundOutOfLimitsAlarm () ⇒ () +
            Any ↦ soundBrokenSensorAlarm () ⇒ ()
        ) *
    ]
]
```

```
interface Timer defines [
    Any ↦ (seconds : Float) ⇒ (ping() *)
]
```

The sensor manager forms the heart of the system. It acts as a mediator controlling the interactions with all other components. The Mediator design pattern is described in [8, page 273–282] as follows:

> Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.

BCOOPL supports the Mediator pattern in two crucial ways. Firstly, the notification mechanism allows objects to be defined that provide their services through notifications. As remarked before, the one-way coupling implied by using notifications is the key to achieving low coupling (see for a detailed discussion of the Observer design pattern [8, page 293–299]). The Mediator knows about the objects involved and links to the appropriate notifications, the objects simply provide their services without being aware of the Mediator or the other objects. In other words, the objects do not even know that they are collaborating. Secondly,

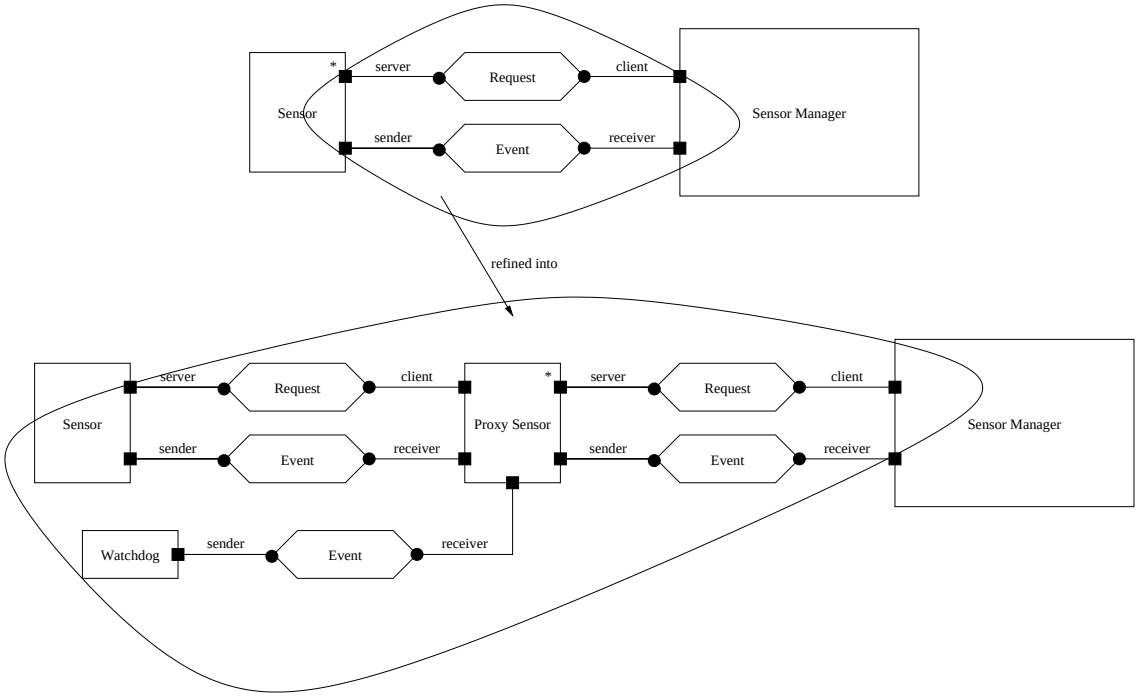

Figure 3: A proxy sensor copes with sensor failures.

BCOOPL's patterns can be used for synchronization purposes. The Mediator centralizes control, which may result in complex synchronization schemes, especially if many objects are involved. Care must be taken that the Mediator itself does not become too complex. BCOOPL helps in reducing the complexity because the allowed interactions can be specified conveniently by means of pattern expressions (i.e., regular expressions over sub-patterns).

The interface and implementation of the sensor manager is given in Figure 4. The interface defines the hooks for attaching the user interface, the alarm and the recording device. The implementation of the sensor manager boils down to reacting to timer signals indicating that the enabled sensors must be polled, handling out-of-limits events received from sensors, and controlling the interactions with the user interface. Notice the event driven way of interacting with the alarm and recording device. These devices can be attached to the sensor manager by linking to the *record*, *outOfLimits*, and *broken* notifications. In this way, a certain amount of flexibility has been added in the sense that the actual configuration can be postponed until run-time. Devices can be attached or detached dynamically, even multiple devices sharing the same interface may be connected simultaneously to the sensor manager.

3.3 Fault-Tolerance

It is interesting to observe that in the specification of the sensor manager we have abstracted away from unreliable sensors. We assume that a sensor will emit a *failure* notification when something bad happens. This is, of course, not a realistic assumption because if a sensor fails for some reason, the chance that a *failure* notification will be actually received by the sensor manager is slim. Nevertheless, the interfaces of *Sensor* and *SensorManager* do support the required hooks in order to handle failures gracefully. To this end, we introduce a proxy sensor, which is responsible for setting up a reliable sensor as far as the sensor manager is concerned. That is, a proxy sensor does issue a *failure* notification when a sensor is apparently malfunctioning. It does so by means of a watchdog that detects the unresponsiveness of a sensor due to a broken sensor or a broken connection. The role of the proxy sensor is exemplified in the refined conceptual architecture shown in Figure 3.

The interface for the *WatchDog* is identical to the *Timer* with the exception that a watchdog barks only once.

```
interface WatchDog defines [
    Any ↦ (seconds : Float) ⇒ (bark())
]
```

The interface and implementation of the proxy sensor is shown in Figure 5. The *ProxySensor* interface extends the *Sensor*. It supports a *setSensor* pattern to associate the proxy sensor with the real sensor in addition to the functionality provided by the real *Sensor*. The proxy sensor passes requests on to the real sensor, but in addition it sets up a watchdog to detect failures. Only the code for the *setSensor* and the *getStatus* pattern is given, the other patterns are implemented similarly.

3.4 Configuration

The *main* pattern given in Figure 6 shows how the process control system is configured. First the components are created before an endless loop is entered. The loop is responsible for handling the *record*, *outOfLimits* and *broken* notifications issued by the sensor manager. The notifications are forwarded to a recording device and an alarm. As in the case of the sensor manager, this loop acts as a mediator establishing loose coupling between the components involved.

```
interface SensorManager defines [
    Any ↦  (sensorSet : Set) ⇒ ((record(status : Status) + outOfLimits(status : Status) + broken() )*) [
        (
            Any ↦ enable (name : String) ⇒ () +
            Any ↦ disable (name : String) ⇒ () +
                .
                . other user interface functionality
        ) *
    ]
]

Any ↦ pattern sensorManager (sensorSet : Set) ⇒ ((record(status : Status) + outOfLimits(status : Status) + broken() )*)
    implements [ SensorManager ]
    declares [
        EnabledSensorsSet : @Sensor ;
        timer : Timer ;
    ]
does [
    // initialize the EnabledSensorsSet
    for all sensors in sensorSet do @EnabledSensorsSet.add(sensor) od ;

    // set up the timer
    timer := new Timer ; timer(900.0) ; // initialize it with a 15 minutes time interval

    // enter the process control section, that is, react to events continuously
    (
        // poll all enabled sensors after a 15 minutes interrupt (timer.ping()) has been received
        timer.ping() ↦ pattern pollSensors () ⇒ () declares [ sensor : Sensor ] does [
            for all sensors in sensorsSet do
                request sensor.getStatus() ; (
                    sensor.getStatus.value(Status) ↦ pattern sensorStatus (status : Status) ⇒ () does [ !! record(status) ] +
                    sensor.getStatus.failure() ↦ pattern sensorFailure () ⇒ () does [ !! broken() ]
                )
            od
        ] *
    ||
        // handle sensor out-of-limits events
        @EnabledSensorsSet.outOfLimits(Status) ↦ pattern sensorOutOfLimits (status : Status) ⇒ () does [
            !! outOfLimits(status)
        ] *
    ||
        // handle the user interface (not shown here)
    )
]
```

Figure 4: Sensor manager interface and implementation.

```
interface ProxySensor extends [ Sensor ] defines [
    Any ↦  () ⇒ ((periodicUpdate(status : Status) + changedUpdate(status : Status) + outOfLimits(status : Status)) *) [
        Any ↦ setSensor (sensor : Sensor) ⇒ () ;
        (
            Any ↦ setLimits (lower : Float, upper : Float) ⇒ () +
            Any ↦ setTimeInterval (seconds : Float) ⇒ () +
            Any ↦ enable () ⇒ () +
            Any ↦ disable () ⇒ () +
            Any ↦ getStatus () ⇒ (value(status : Status) + failure())
        ) *
    ]
]

Any ↦ pattern proxySensor () ⇒ ((periodicUpdate(status : Status) + changedUpdate(status : Status) + outOfLimits(status : Status)) *)
    implements [ ProxySensor ]
    declares [ sensor : Sensor ; ]
does [
    Any ↦ pattern setSensor (realSensor : Sensor) ⇒ () does [ sensor := realSensor ] ;
    (
            .
            .
        Any ↦ pattern getStatus () ⇒ (value(status : Status) + failure())
            declares [ watchDog : WatchDog ; ]
        does [
            watchDog := new WatchDog ; watchDog(5.0) ;
            request sensor.getStatus() ; (
                sensor.getStatus.value(Status) ↦ pattern value (status : Status) ⇒ () does [ !! value(status) ] +
                sensor.getStatus.failure() ↦ pattern failure () ⇒ () does [ !! failure() ] +
                watchDog.bark() ↦ pattern timeout () ⇒ () does [ !! failure() ]
            )
        ]
    ) *
]
```

Figure 5: Proxy sensor interface and implementation.

```
Any ↦ pattern main () ⇒ ()
    declares [
        sensorSet : Set ;
        sensor1, . . . , sensor10 : Sensor ;
        proxySensor1, . . . , proxySensor10 : ProxySensor ;
        recordingDevice : RecordingDevice ;
        alarm : Alarm ;
        sensorManager : SensorManager ;
    ]
does [
    // create and initialize the components
    sensorSet := new Set ; sensorSet() ;

    sensor1 := new Sensor ; sensor1() ;
    proxySensor1 := new ProxySensor ; proxySensor1() ;
    proxySensor1.setSensor(sensor1) ;
    sensorSet.add(proxySensor1) ;

        .
        .
    sensor10 := new Sensor ; sensor10() ;
    proxySensor10 := new ProxySensor ; proxySensor10() ;
    proxySensor10.setSensor(sensor10) ;
    sensorSet.add(proxySensor10) ;

    recordingDevice := new RecordingDevice ; recordingDevice() ;
    alarm := new Alarm ; alarm() ;

    sensorManager := new SensorManager ; sensorManager(sensorSet) ;

    // continuously react to sensor manager events
    (
        sensorManager.record(Status) ↦ pattern record (status : Status) ⇒ () does [
            recordingDevice.record(status)
        ] +
        sensorManager.outOfLimits(Status) ↦ pattern outOfLimits (status : Status) ⇒ () does [
            alarm.soundOutOfLimitsAlarm()
        ] +
        sensorManager.broken() ↦ pattern broken () ⇒ () does [
            alarm.soundBrokenSensorAlarm()
        ]
    ) *
]
```

Figure 6: System configuration.

A pattern used in this way can be seen as a configuration specification. It describes how components are tied together to form a system.

4. DISCUSSION

The need to separate individual component behavior from component interactions can be justified on the grounds of flexibility (e.g., adaptability, portability, and heterogeneity) as has been argued by, amongst others, Gelernter et al. [9]. The question now is: what kind of language features are required to express coordination control? To begin with, most general-purpose programming languages and theoretical models are not well-equipped for coordination control because they are based on the Target-Send/Receive (TSR) model. The TSR model is asymmetric in the sense that the sender knows the target or targets of a message, whereas the receiver accepts a message unconditionally. As a consequence, the sender has to name the target(s), which increases the coupling between components. In the IWIM (Idealized Worker Idealized Manager) model, as manifested in Manifold [1], this unbalance between sender and receiver is repaired by adopting a model based on processes, events, ports and channels. Processes have input and/or output ports that can be connected by channels for message exchange. The connection between processes can be setup as follows:

```
p.o → q.i
```

This denotes that process p is connected via its output port o with the input port i of process q. Notice the similarity with a notification linkage specification in BCOOPL.

```
server.notification(formals) ↦ pattern
        Pattern Name (input arguments) ⇒ (notification pattern)
does [
    implementation
]
```

The construct $p.o \rightarrow q.i$ translates into:

```
p.o() ↦ pattern myPattern () ⇒ () does [ q.o() ]
```

In other words, BCOOPL offers a similar language construct with which connections between independently operating components can be specified. A small amount of syntatic sugar can make this as expressive as the corresponding Manifold construct.

The IWIM model supports five kinds of channels, ranging from synchronous communication channels to asynchronous communication channels with several variations of automatic disconnection. A channel is a treated as a first class citizen, it has an I.D. that can be communicated. In contrast, communication in BCOOPL is based on asynchronous communication (point-to-point and multicast) assuming reliable connections. However, we have shown in the process control example how connectors can be specified explicitly to abstract away from communication peculiarities between components. Thus, if the need arises, connectors can be treated as first class citizens in BCOOPL. Moreover, with BCOOPL we have the full power of a programming language to specify sophisticated connectors. A connector with

a time-out facility as used in the process control example is a good example of such a connector.

Setting up connections between components is not the end of the story. Communications between components must be synchronized, since not all components are prepared to handle all sorts of messages at anyone time. There must be ways to specify the legality of a message. In BCOOPL this is done in interface specifications by means of regular expressions over operations. A similar approach is used in ToolBus [3]. ToolBus can be seen as a hardware communication bus that controls the interactions between software components (tools), which are described by T-scripts. A T-script is a process-oriented description featuring process composition (sequences, choices and iterations of processes), synchronous and asynchronous (notification style) communications, dynamic process creation. BCOOPL has a lot in common with, for instance, composition operators, notifications and dynamics. One difference can be found in the fact that component behavior as well as component interactions are both specified in BCOOPL. This allows for choosing multiple levels of granularity because basic components (i.e., tools) can be composed with patterns to form a new component. The external behavior of this composition can be abstracted away in an interface to facilitate composition at higher levels of abstraction, and so on. Another difference can be found in the object-orientedness of BCOOPL, which encourages reuse by means of interface inheritance and black-box composition through delegation.

The underlying design principle of BCOOPL is to provide a small but powerful set of language features with which a wide range of application domains can be addressed. A number of language features commonly found in general purpose languages are not part of BCOOPL, including real-time processing and exception handling. This was a deliberate design decision. In the design of BCOOPL, we have focussed on run-time flexibility to dynamically extend systems in terms of adaptable object configurations. The key language features for obtaining this required level of flexibility are delegation and the built-in support for the Observer and Mediator design patterns. This approach can be contrasted with the design philosophy of Ada 83 and its successor Ada 95, which can be charaterized as feature-loaded languages. The principle synchronization mechanism in Ada is the rendez-vous, which is inflexible in the sense that it introduces strong coupling between the parties involved in message exchange. The concept of protected objects has been added to Ada 95 to have a lightweight synchronization mechanism in line with the OO paradigm [2]. The basic idea of a protected object is to synchronize the access to shared data. This concept is similar to a BCOOPL pattern, although a pattern is more powerful by offering nested patterns and explicit sequencing in the form of regular expressions.

5. CONCLUDING REMARKS

We have shown how BCOOPL can be used as a design and implementation language for process control problems complementing architectural design descriptions. Its main application area is to specify weakly-coupled components and the interaction sequences between those components. The key concepts can be summarized as follows. The built-in notification mechanism allows to specify stand-alone components that provide server offerings for other components. These service offerings are abstracted away in interfaces (see, for example, the interface for a sensor). Components are configured in patterns in the role of mediator controlling the interactions between independently operating components (see, for example, the sensor manager pattern). As a result, individual component behavior is separated strictly from component interactions, which facilitates the design of flexible systems.

6. REFERENCES

[1] F. Arbab. The IWIM model for coordination of concurrent activities. In P. Ciancarini and C. Hankin, editors, *Proceedings of the First Conference on Coordination Languages and Models (Coordination'96), Cesena, Italy*, volume 1061 of *Lecture Notes in Computer Science (LNCS)*, pages 34–56, Berlin, Germany, Apr. 1996. Springer-Verlag.

[2] J. E. Barnes. *Ada 95: The Language, The Standard Libraries*. Number 1247 in Lecture Notes in Computer Science (LNCS). Springer-Verlag, Berlin, Germany, 1997.

[3] J. Bergstra and P. Klint. The ToolBus coordination architecture. In P. Ciancarini and C. Hankin, editors, *Proceedings of the First Conference on Coordination Languages and Models (Coordination'96), Cesena, Italy*, volume 1061 of *Lecture Notes in Computer Science (LNCS)*, pages 75–88, Berlin, Germany, Apr. 1996. Springer-Verlag.

[4] G. Booch. *Software Engineering with Ada*. Bejamin/Cummings, Menlo Park, California, 1983.

[5] P. Bouwman and H. de Bruin. Talktalk. In P. Wisskirchen, editor, *Object-Oriented and Mixed Programming Paradigms*, Eurographics Focus on Computer Graphics Series, chapter 9, pages 125–141. Springer-Verlag, Berlin, Germany, 1996.

[6] R. Campbell and A. Habermann. The specification of process synchronization by path expressions. In *Lecture Notes in Computer Science 16*, pages 89–102. Springer-Verlag, Berlin, Germany, 1974.

[7] H. de Bruin. BCOOPL: Basic concurrent object-oriented programming language. *Software Practice & Experience*, 30(8):849–894, July 2000.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Professional Computing Series. Addison-Wesley, Reading, Massachusetts, 1995.

[9] D. Gelernter and N. Carriero. Coordination languages and their significance. *Communications of the ACM*, 35(2):96–107, Feb. 1992.

[10] C. Hofmeister, R. Nord, and D. Soni. *Applied Software Architecture*. Object Technology Series. Addison-Wesley, Reading, Massachusetts, 1999.

[11] O. L. Madsen, B. Møller-Pedersen, and K. Nygaard. *Object-Oriented Programming in the Beta Programming Language*. Addison-Wesley, Reading, Massachusetts, 1993.

[12] B. Selic, G. Gullekson, and P. T. Ward. *Real-Time Object-Oriented Modelling*. John Wiley and Sons, New York, 1994.

[13] J. van den Bos and C. Laffra. Procol: a concurrent object language with protocols, delegation and persistence. *Acta Informatica*, 28:511–538, Sept. 1991.