Writing Concurrent Desktop Applications in an Actor-Based Programming Model

Jan Schäfer
University of Kaiserslautern
Computer Science Department
D-67653 Kaiserslautern, Germany
jschaefer@cs.uni-kl.de

Arnd Poetzsch-Heffter
University of Kaiserslautern
Computer Science Department
D-67653 Kaiserslautern, Germany
poetzsch@cs.uni-kl.de

ABSTRACT

GUI frameworks, like Swing, are typically not thread-safe. Desktop applications are thus often written in a purely single-threaded, event-based style. Introducing threads into such applications is not an easy task as potentially all parts of the application may be affected by this change.

Instead of using a thread-based programming model, actor models are regaining attention lately. The actor-based *CoBox model* is based on isolated object-oriented components communicating via asynchronous method calls. The model is implemented in a Java extension, called *JCoBox*, and has been successfully used to implemented several concurrent desktop applications. In this paper we show how a typical desktop application is designed and implemented in JCoBox.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel Programming*; D.3.2 [**Programming Languages**]: Language Classifications—*Concurrent, distributed, and parallel languages*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Concurrent programming structures*

General Terms

Languages, Design

Keywords

concurrency, object-orientation, components, actors, desktop applications, graphical user interfaces

1. INTRODUCTION

The ubiquitous availability of multi-processor desktop computers pushes concurrency into new application domains like standard desktop applications. Typical desktop applications are written in an object-oriented language like Java or C#.

These languages have built-in mechanisms for concurrent programming. However, concurrency in these languages is introduced by dividing the control flow over a number of concurrently running threads working on a shared state. To prevent threads from unwanted interleavings, low-level, basic synchronization concepts, like locks, have to be used. Experience shows that software written in such a way is prone to errors, difficult to debug, hard to maintain and to extend [31, 26].

Regarding desktop applications an additional problem exists, namely that GUI frameworks, like Swing for example, are in general not thread safe. This complicates the design and implementation of concurrent desktop applications as all interactions with the GUI framework must happen by a special event-handling thread.

The actor model [18, 2, 22] has lately regained attention as, in contrast to thread-based concurrency, it encapsulates control flow and data. Prominent examples are Erlang [4] based on a functional language and Scala actors [16] integrated into a modern OOL.

Combining actors with object-orientation comes with some challenging problems. One problem is that actors communicate by message passing instead of invoking methods. Thus, resulting systems are written with two incompatible communications mechanisms. In Scala actors, it is for example unsafe to simply call a method on an actor, as method calls are not protected by the actor. In addition, messages are often only dynamically typed or require additional mailboxes to become statically type safe, further complicating the communication mechanism. Instead of using messages, asynchronous method calls provide a type safe communication mechanism, compatible with standard method calls, which is adopted by many active object approaches [9, 10, 12, 8].

Object-oriented components are often realized by groups of interacting objects. Similar to many other component models, an OSGi component [32], for example, provides a number of so-called *services*, where each service can be realized by a different object. In contrast, typical active object approaches have single objects as a unit of concurrency, which makes it difficult to implement such components, as each service object must be an active object again, which cannot share state with the main component object. Some actor approaches allow to represent the state of an actor by multiple objects, e.g. ASP [10]. These objects, however, cannot be referenced by other actors. This issue is solved in the E programming language [23] and AmbientTalk [33], which allow to reference multiple objects in an actor.

Most actor and active object approaches have in common that a single thread is responsible for executing the code inside an actor. This makes it difficult to have multiple independent control flows within an actor, which is important for two reasons: first, it does not easily allow to combine active with reactive behavior, and second, waiting for certain messages requires the actor to completely block the actor for other activities. Creol proposes a solution, which allows to have multiple cooperatively scheduled tasks within a single active object [12].

The CoBox model and the corresponding JCoBox programming language [29], unifies several actor-based programming models. It is based on the idea of asynchronously communicating object-oriented components, *coboxes*, that run in parallel. Inside a cobox concurrency is realized by cooperative tasks. This paper gives an introduction to the CoBox model and the JCoBox language and shows how they can be used to design and implement a typical object-oriented desktop program.

2. THE COBOX MODEL AND JCOBOX

This Section briefly describes the cobox model and the JCoBox programming language.

2.1 The CoBox Model

The central concept of the CoBox model is the *cobox*. CoBoxes are concurrently running, isolated, object-oriented components. Figure 1 presents a schematic view of the cobox model. A cobox encapsulates both state and behavior and is in this sense similar to active objects [34, 3, 9]. The state of a cobox consists of a heap of objects. The cobox *owns* these objects for their entire lifetime. The behavior of a cobox is represented by a set of *cooperative tasks*, which, again, are owned by the cobox for their entire lifetime.

2.1.1 CoBox-Local Computations

Inside a cobox, computations are similar to sequential object-oriented programming. All objects of a cobox can be directly accessed by accessing their fields or by invoking methods. Such *direct method calls* are immediately executed by the calling task in the standard stack-based way. To realize concurrency, a cobox supports multiple, possibly interleaved control flows, called *tasks*. Tasks are created when methods are asynchronously called on objects of a cobox and are responsible for executing the called method. Tasks are scheduled *cooperatively*. This means that at most one task can be *active* in a cobox at a time, and that the active task has to give up its control explicitly to allow other tasks to become active.

If a task is not active, it is either *ready* or *suspended*. Initially a task is in the ready state. Ready tasks are organized in a FIFO queue (the *ready queue*). If the active task gives up control the next task from the ready queue becomes active. A task can give up control in three ways: it can terminate, it can *yield*, which immediately adds it to the end of the ready queue, or it can *suspend*, waiting for some condition to be satisfied. When the condition of a suspended task becomes satisfied, the task is *awaked* and added to the ready queue, for eventual execution.

2.1.2 Inter-CoBox Communication

As a cobox is only represented by its objects, inter-cobox communication means that objects of one cobox communicate with objects of another cobox. To do so, an object in one cobox needs to reference objects of other coboxes. Such references are called *far references* (following the naming of E [23]), in contrast to *local references*, which refer to objects of the same cobox. A far reference to an object can only be used to asynchronously invoke methods on these objects. It is not possible to use far references as targets for direct method calls or fields accesses.

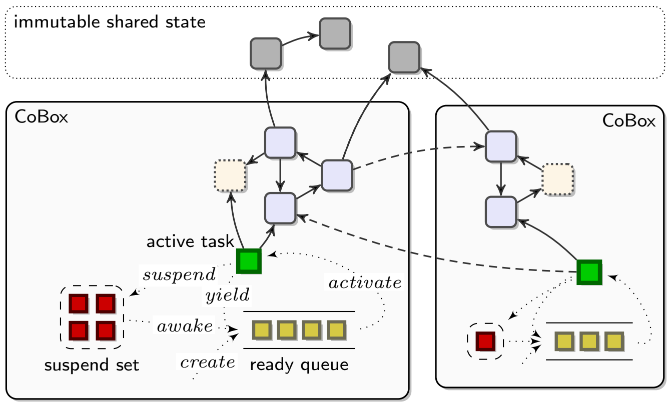


Figure 1: Schematic view of the cobox model. Legend: ▢ standard objects, ⬚ transfer objects, ■ immutable objects, 🟩 active task, 🟨 ready tasks, 🟥 suspended tasks, ⟶ local reference, ⇢ far reference ┈➤ task scheduling.

2.1.3 Data Transfer and Object Sharing

To combine encapsulation, data transfer between coboxes, and sharing of objects among coboxes, the cobox model distinguishes three kinds of objects. *Standard objects* in the CoBox model are passed by reference between coboxes. To transfer data between coboxes, without exposing objects of a cobox by passing it via a far reference, *transfer objects* can be used. Transfer objects are always local to a cobox and can never be referenced by other coboxes using far references. Similar to Serializable objects in RMI, transfer objects are transitively copied, when passed to another cobox. The target cobox then gets a local reference to the object copy instead of a far reference to the original object.

As transferring data using transfer objects can be inefficient due to the copying overhead, it is possible to directly share state between coboxes using *immutable objects* [15, 11, 7]. Immutable objects never change their state, thus it is safe to access this state concurrently. Like standard objects, immutable objects are passed by reference between coboxes. Conceptually, immutable objects are not owned by any cobox. Nevertheless, references to immutable objects are treated as local references. It is thus possible to directly call methods and to directly access the fields of immutable objects. Immutable objects can only reference standard objects or other immutable objects. Transfer objects cannot be referenced by immutable objects.

2.2 The JCoBox Language

A programming model alone cannot be used to write programs. JCoBox is an extension of sequential Java that implements the cobox model to realize concurrency. The syntax of Java is only minimally extended by an additional operator for asynchronous method calls and an extended new expression. All other extensions are done by standard Java annotations and using special purpose classes and interfaces. JCoBox is implemented as a compiler extension for the Polyglot compiler framework [25] and can be download from [19]. We assume that the reader is familiar with sequential Java and thus we only explain the concepts introduced by the JCoBox extension.

2.2.1 *Creating CoBoxes*

Coboxes are not first-class citizens of the language; a cobox is only represented by the objects it contains. A cobox is *implicitly* created when a *cobox class* is instantiated. Instances of cobox classes are standard objects in the cobox model, but with the guarantee that they are always created in a *new* cobox. They are thus the first object of a cobox and can then be used to interact with the cobox. CoBox classes are declared by the **@CoBox** annotation.

2.2.2 *Assigning Objects to CoBoxes*

Objects of cobox classes are always created in a new cobox. Objects of non-cobox classes are created, by default, in the cobox of the creating task. It is possible to create non-cobox objects in other coboxes than the current cobox by using an extended **new** expression that allows to specify a target cobox. For example, **new** A() **in** (b) creates an object of A in the cobox that owns object b.

2.2.3 *Transfer and Immutable Objects*

Standard objects in JCoBox are instances of cobox classes or plain classes. Transfer and immutable objects are instances of *transfer* and *immutable classes*, respectively, which are declared by **@Transfer** and **@Immutable** annotations. To guarantee immutability of immutable objects, JCoBox currently adopts a simple mechanism, namely to make all fields of immutable classes implicitly final and restrict the type of fields to not be transfer classes. This is a rather restrictive definition, but is, nevertheless, sufficient in many cases. A more flexible notion of immutability based on existing work [14, 11, 15] is planned for the future.

2.2.4 *Asynchronous Method Calls*

Asynchronous method calls are expressed by using the **!** operator instead of a dot. For example, a **!** m() asynchronously invokes the method m() on object a. Any method can be invoked asynchronously in JCoBox. Asynchronous method calls are partially ordered. Partially means that two subsequent calls, from the same cobox, targeting objects of the same cobox, are executed in the given order. For example, the bodies of the method calls x **!** m(); y **!** n() are executed in the given order if x and y refer to objects owned by the same cobox.

2.2.5 *Futures*

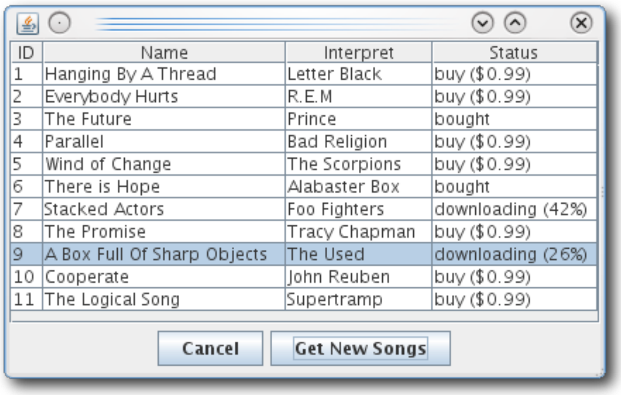
Asynchonous method calls return *futures* [6, 17]. Futures are place holders for the results of asynchronous method calls. Futures in JCoBox are instances of the special interface Fut<V>, where V is the type of the future value. In order to get the value of a future, it has to be explicitly *claimed* [1]. Similar to the Creol approach [12], claiming a future in JCoBox can be done in two different ways: a blocking and a cooperative one. Claiming a future blockingly is done by using the get() method. Blockingly means that the active task waits for the future without giving up control, thus preventing other tasks from being activated. A task can also wait cooperatively for a future using the await() method. Cooperatively means that when the future is not ready yet, the waiting task gives up control, allowing other tasks to be executed. Synchronous communication can be simulated by asynchronous method calls with an immediate get() or await(). In fact, a synchronous call x.m() is treated as x!m().**get**() if x refers to a far reference.

2.2.6 *Task Cooperation*

Beside using await() to give up control to another task, it is also possible to directly yield control to the next ready task by using the JCoBox.**yield**() method.

3. EXAMPLE: CONCURRENT MUSIC MANAGER

In this Section we show how the CoBox model and JCoBox can be used to develop and implement a typical desktop application. Object-oriented desktop applications are in general built by the model-view-controller pattern [13]. The pattern divides the application into at least three components, namely a model, a view, and a controller component. In a sequential program these components are tightly coupled, because they communicate by synchronous method calls. As the view is forced to wait for the other components, an application can quickly become unresponsive if one of its components cannot return from a method call in a short amount of time. If I/O is involved in one of the tasks, the response time can, in general, not be predicted. Another problem of synchronous communication are callbacks [23]. For example, if the model notifies its views about a state change and one view calls back to the model, the state of the model might arbitrary change during the notification process.



ID	Name	Interpret	Status
1	Hanging By A Thread	Letter Black	buy ($0.99)
2	Everybody Hurts	R.E.M	buy ($0.99)
3	The Future	Prince	bought
4	Parallel	Bad Religion	buy ($0.99)
5	Wind of Change	The Scorpions	buy ($0.99)
6	There is Hope	Alabaster Box	bought
7	Stacked Actors	Foo Fighters	downloading (42%)
8	The Promise	Tracy Chapman	buy ($0.99)
9	A Box Full Of Sharp Objects	The Used	downloading (26%)
10	Cooperate	John Reuben	buy ($0.99)
11	The Logical Song	Supertramp	buy ($0.99)

Figure 2: Screen shot of the CoMusic application

3.1 CoMusic: Requirements

As a concrete example, we use a very simple imaginary music management application, called *CoMusic*. Figure 2 shows a screen shot of how it could look like. The application should manage a list of songs. Songs can be bought from a shop, and bought songs can be played. It should be possible to download multiple songs, to update the list of available songs, and to play a bought song. All these tasks should run concurrently. At all times the application should be able to react to user input.

CoMusic has to be implemented in a concurrent way, as the process of downloading a song should not prevent another song from being played, for example. In the typical multi-threaded model of object-oriented languages like Java, these independent control flows would be modeled as threads. As these threads may potentially interact with any component, these components have to be written in a thread-safe way to avoid data races. As communication is in general synchronous in the standard thread-model, deadlocks must be avoided, which is difficult in typical MVC-applications because of the circular dependencies of the participating components. Finally, as threads are conceptually different to components, it is difficult to understand such an application, as active threads are intermixed with passive components.

3.2 CoBox Design

A JCoBox program is realized as a set of interacting, parallel running coboxes. In the design phase of JCoBox programs, it is thus decided which coboxes should exist at runtime. Figure 3 shows how the *CoMusic* example can be divided into coboxes.

The model is realized by the SongModel cobox. It holds the information about all songs as well as their download status. To keep the GUI responsive all the time, all GUI-related code is realized as a separate GUICtrl cobox, which consists of two parts: a controller object of type GUICtrl, which interacts with other coboxes and a set of Swing objects, which realize the graphical user interface. As the Swing objects are implemented by standard Java classes and not by JCoBox classes, they can only interact with objects of the same cobox. This is why in the diagram they are drawn with a dashed line and a red background color. Finally, download requests are handled by the DLCtrl cobox. It uses additional DLProcess coboxes to handle separate downloads concurrently. For simplicity, we left out the PlayCtrl cobox, which handles the playing of songs.

Songs are represented by immutable Song objects. This allows all components to efficiently access and share Song objects. In addition, SongStatus objects are used to represent the download status of songs. These objects are realized as mutable transfer objects and are thus copied when passed between coboxes.

From the runtime view of the *CoMusic* application is it immediately clear which parts of the application can run in parallel. Each cobox represents a potential parallel execution. Inside a cobox there may only be cooperative multi-tasking. In a standard thread-based implementation it would be much more difficult to describe concurrency, as threads and components are orthogonal.

3.3 JCoBox Implementation

After having defined how the architecture of *CoMusic* should look like, we can start with the implementation. Each cobox appearing at runtime is created by instantiating a corresponding *cobox class*.

3.3.1 The Main CoBox

Every JCoBox program starts by executing the standard main method in the initial *Main* cobox. In the *CoMusic* example, the main method creates and wires the required coboxes (Listing 1). A JCoBox program normally terminates if there are no tasks in any cobox anymore. As in Swing applications it might be the case that there are no tasks at all, because the application waits for user input, the default behavior has to be disabled by invoking JCoBox.setAutomaticShutdown(false). Shutting down the application must then be done manually by invoking JCoBox.shutdown().

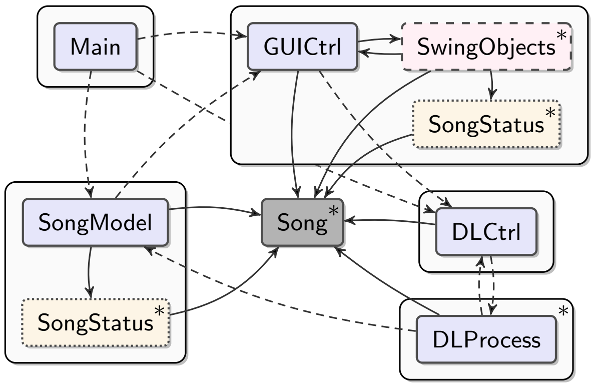


Figure 3: Runtime view of the CoMusic application.

```
class CoMusic {
  public static void main(String[] args) {
     JCoBox.setAutomaticShutdown(false);
     GUICtrl gui = new GUICtrl();
     gui ! init().await();
     DLCtrl dl = new DLCtrl(model);
     gui ! registerBuyListener(dl);
     SongModel model = new SongModel();
     model ! registerListener(gui);
  }
}
```

Listing 1: The main method of *CoMusic*

3.3.2 Data

The main data of the music application are Song objects. To allow all components to efficiently access and transfer these objects, they are made immutable. The implementation of the Song class is as follows.

```
@Immutable class Song {
   int ID;
   String name;
   String interpret;
   Song(int id, String n, String i) {
      ID = id; name = n; interpret = i;
   }
}
```

Note that all fields of @Immutable classes are implicitly final.

The other data objects are SongStatus objects, which hold the current download status of a song. As the download status changes over time, the class is defined as a transfer class. SongStatus objects are thus copied when they are transferred to other coboxes.

```
@Transfer class SongStatus {
   int progress;
```

```
    Song song;
    SongStatus(Song s) {
      song = s;
    }
    ... // setters and getters
}
```

3.3.3 The Model CoBox

The model cobox is implemented by the SongModel cobox class (Listing 2). Its state consists of a map, which maps song IDs to corresponding SongStatus objects. In addition, it has a list of SongModelListener objects, which are notified about state changes. Listeners are notified asynchronously. This has a couple of advantages compared to a synchronous notification [23]. The first advantage is that during the notification process no listener can callback the model, and in particular the list of listeners cannot change. The second advantage is that the model has not to wait until each listener has handled the notification, which allows the model to be quickly available again. Finally, the model is unaffected by any exceptions thrown in listeners. Noteworthy is the fact that the SongModel implements the *push*-based subject-observer pattern [13], i.e., the changed data is directly passed to the observers to avoid that observers have to callback to the model to obtain the new data. We made the experience that the push-based subject-observer pattern is much better suited for concurrent loosely-coupled systems as it simplifies the communication structure and increases parallelism. Note also that the SongStatus objects are transfer objects and are copied when passed to the listeners.

```
@CoBox class SongModel {
    private Map<Integer,SongStatus> songs = ...
    private List<SongModelListener> listeners = ...
    ...
    public void updProgress(Song s, int p) {
        SongStatus stat = songs.get(s.ID);
        stat.setProgress(p);
        for (SongModelListener lis : listeners) {
            lis ! statusChanged(stat); }
} }
```

Listing 2: The SongModel cobox class

3.3.4 The GUI Component

The graphical user interface is implemented by the GUICtrl class (Listing 3). It acts as a controller class, which handles all communication to and from Swing objects that realize the GUI. This communication happens in a standard sequential event-based style. By annotating the GUICtrl class with @Swing, JCoBox ensures that all code of that cobox is executed by the event-dispatching thread of Swing. It is thus always safe to interact with Swing objects inside such a cobox. But it also means that all code executions inside such coboxes block the GUI. Such blocking, however, can be kept to a minimum by delegating requests to other coboxes as soon as possible. For example, the action handling code for the buy button first gets the selected song from the SongTable and then immediatly delegates the buy request to the DLCtrl cobox using an asynchronous method call. The SongTable class is annotated with @PlainJava as it inherits from a standard Java class. The GUICtrl class communicates in a standard synchronous way with SongTable as the SongTable object lives in the same cobox as the GUICtrl object.

```
@CoBox @Swing class GUICtrl
    implements SongModelListener {
    private DLCtrl dlctrl;
    private SongTable table = new SongTable();
    private JButton buyBtn;

    public void init() {
      ...
      buyBtn = new JButton("Buy");
      buyBtn.addActionListener(new ActionListener() {
          public void actionPerformed(ActionEvent a) {
              Song s = table.getSelectedSong()
              dlctrl ! buy(s); }});
      ...
    }
    public void statusChanged(SongStatus s) {
       table.updateStatus(s);
    }
    ... // further code omitted
}

@PlainJava
class SongTable extends AbstractTableModel {
   ... // implementation omitted
}
```

Listing 3: The GUICtrl class and the SongTable class

3.3.5 The Download Component

As the downloading of songs and song information requires I/O operations, it is put into a separate cobox. The DLCtrl cobox is responsible for managing different downloads. Each single download is then handled by additional DLProcess coboxes (Listing 4). The DLProcess has a long running task that downloads a single song. As it should be possible to cancel the download process, the task constantly calls JCoBox.yield() to allow the execution of possible calls to the cancel() method. As the scheduling of tasks is always fair in JCoBox, it is guaranteed that all other tasks are executed before a yielded task is executed again.

```
@CoBox class DLProcess {
    DLCtrl ctrl;
    boolean canceled;
    DLProcess(DLCtrl c) {
       ctrl = c;
    }
    void cancel() {
       canceled = true;
    }
    void start(Song s) {
       while (!canceled) {
          ... // download next piece
          JCoBox.yield(); // process cancel calls
       }
       ctrl ! finished(s);
    }
    ... // further code omitted
}
```

Listing 4: The DLProcess class

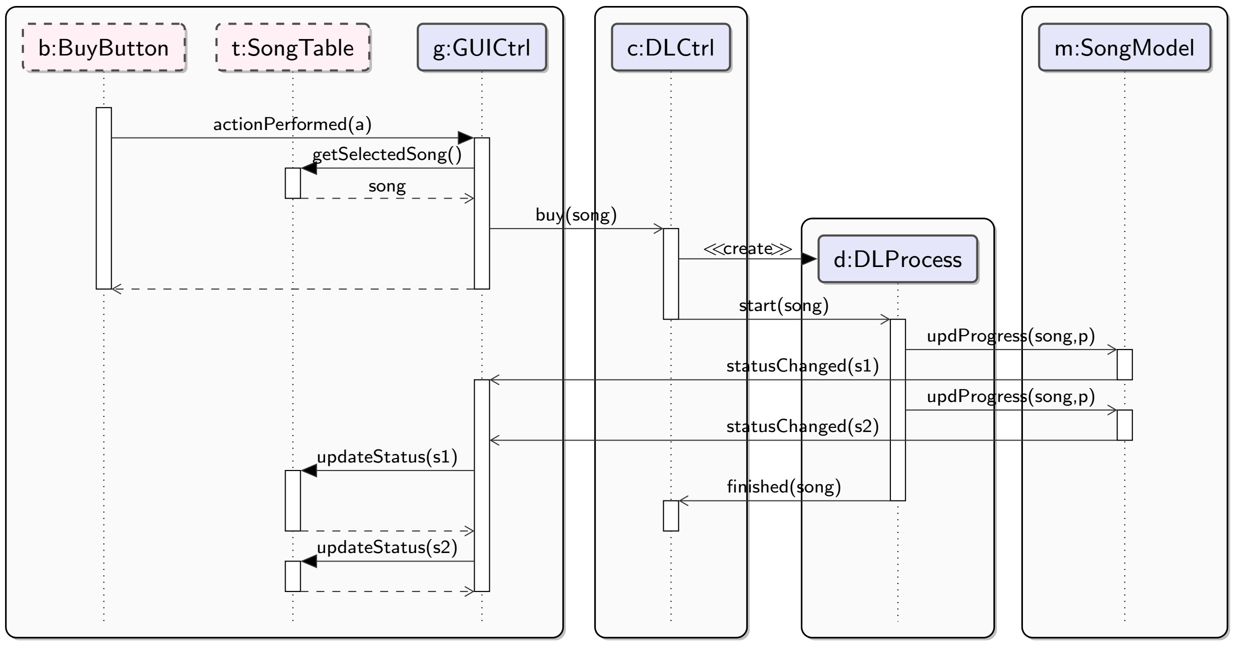


Figure 4: Sequence diagram showing the behavior, when the buy-button is pressed. The gray rounded rectangles represent coboxes. Activities of different coboxes run concurrently. Inside a cobox, standard sequential programming is used. Communication between coboxes only happens by asynchronous method calls.

3.3.6 *Runtime Behavior*

To illustrate a typical message flow in the *CoMusic* application, we show the process of buying a song in a sequence diagram (Figure 4). To make clear, which activities run in the same cobox, we draw a cobox boundary around them. Activities, which run in different coboxes, can run concurrently, activities in a single cobox are executed sequentially. An important difference to standard sequence diagrams is that asynchronous messages that have the same source and target cobox are ordered. For example, the two statusChanged() calls from the SongModel to the GUICtrl are guaranteed to be executed in the order in which they have been sent. This is important as otherwise an outdated status may be shown.

3.4 Discussion

The implementation of the example application shows that the cobox model fits naturally into typical object-oriented programming. The general structure of an application is similar to a typical sequential implementation, but components become loosely-coupled and run concurrently. The JCoBox implementation has several advantages compared to a thread-based implementation. The first is that data races are in principle not possible in JCoBox. However, for pragmatic reasons, legacy Java classes can be used in JCoBox, which opens the possibility of data races. The programmer has to ensure that objects of legacy Java classes are either thread-safe, or are never shared by two coboxes at the same time. If no thread-unsafe legacy Java classes are used, a JCoBox program is guaranteed to be free of data races. The second advantage is that the possibility of deadlocks is greatly reduced, due to the use of asynchronous communication. Deadlocks are still possible when blockingly claiming a future, but are, according to our experience, much easier to find and are often even deterministic. Last but not least, applications written in JCoBox are easier to understand because concurrency is structured by components. The components themselves are the concurrent entities, not orthogonal threads. The behavior of components can be understood in isolation, without taking concurrency of the environment into account. Due to cooperative multi-tasking inside components, the number of possible task interleavings is greatly reduced, which makes it much easier to understand and test JCoBox components.

The chosen design for the *CoMusic* application is only one possibility. If the application has already existed in a purely sequential implementation, one might have started with a single Swing cobox, which runs all existing code. This implementation would then be semantically identical to the purely sequential one. Concurrency can then be introduced step-by-step, by introducing new coboxes. In contrast to adding threads to a sequential program, adding coboxes is completely safe in the sense that the existing sequential code is not affected by this change. This allows for a smooth migration from sequential to concurrent programs.

4. RELATED AND FUTURE WORK

JCoBox does not introduce radically new ideas or features. It rather unifies existing ideas into a single programming model and provides a practical implementation on top of Java.

The principle idea of a data-centric concurrency model, i.e. a model where concurrency is structured by the data, stems from the actor model [18, 2, 22]. The pure actor model is defined in a functional setting, without mutable state, with Erlang [4] being the most prominent implementation of the recent past. The combination of stateful objects and actors as *active objects* was first done in ABCL/1 [34], POOL2 [3], and Eiffel// [9]. These approaches have single objects as the unit of concurrency. ASP [10] generalized this to *groups* of objects called *activities*. Activities, however, have only one distinguished object, the *active object*, which can be referenced by other activities. Multiple service objects for one activity are thus not possible. ASP also introduced the notion of *passive objects*, which can only be referenced inside a single activity and are deep-copied when passed to other activities. Passive objects correspond to transfer objects in JCoBox. The ASP concurrency model is implemented in ProActive [5].

The E programming language [23] introduces a concurrency model called *communicating event loops*. The unit of concurrency in E is a *vat*, which hosts a group of objects. All objects of a vat can be referenced by other vats, allowing multiple service objects, equally to coboxes. Computations inside a vat, however, are only executed by a single thread of control. This leads to an event-based programming model, where the control flows has to be spread over event handlers. The E programming model can be simulated in JCoBox by never suspending or yielding a task. Ordering of messages in E is only with respect to single objects, where in JCoBox it is with respect to coboxes, which we believe is what a programmer in general expects. Like JCoBox, E allows to execute a vat by the event handling thread of Swing, to allow for seamless GUI interaction. The E programming model has been lately adopted by AmbientTalk [33]. AmbientTalk also integrates a notion of transfer objects called *isolates*.

The idea of using cooperative multitasking for concurrency inside of active objects stems from Creol [20, 12, 21]. In addition, combining cooperative multitasking with futures, was also pioneered by Creol. Creol, however, has single objects as the unit of concurrency, which does not allow for multiple service objects. The CoBox model can be seen as a generalization of the Creol model, where every object is owned by a unique cobox. In Symbian OS [24] active objects are scheduled cooperatively within the same active scheduler, which are thread-local. Each active object only has a single thread of control. Symbian OS also shows how to combine active objects with GUI programming. Kilim [30] allows to schedule tasks cooperatively if the assigned scheduler is configured to be single-threaded. The Lua language [28] combines cooperative multitasking with asynchronous RMI.

Thorn [8] is an actor approach, which combines message sending and asynchronous method calls. Unlike JCoBox, Thorn does not support multiple tasks within a process. However, Thorn has a special splitSync construct, which can be used to solve the problems of the single-threading of Thorn components in some cases. Only methods declared to be asynchronous can be invoked in an asynchronous way in Thorn, where in JCoBox, any method can be invoked asynchronously.

5. CONCLUSIONS AND FUTURE WORK

Concurrency is reaching the main stream. Current programming models make it difficult to write concurrent desktop applications. This paper shows how JCoBox can be used to write such applications in a safe way. JCoBox realizes concurrency by so-called *CoBoxes*. CoBoxes are parallel running, isolated, object-oriented components, which only interact by asynchronous method calls. Concurrency inside coboxes is realized by cooperative multi-tasking, allowing state-sharing independent control flows inside a cobox. The model is well-suited for writing typical desktop applications, which are in general based on the model-view-controller pattern. Each component can be realized as a separate cobox, resulting in a loosely-coupled system. Existing single-threaded code, like GUI framework code, can be wrapped into a single cobox. This allows writing GUI-related code in the standard sequential way, which can safely interact with concurrent parts of the application by using asynchronous method calls.

In JCoBox each object belongs to a certain cobox. In our experience, it is in general clear to the programmer to which cobox an object belongs to. However, we plan to integrate a pluggable type system extension [27] to make this relation explicit in the source code. This would also allow statically show the absence of object sharing, which would allow the safe usage of unsafe Java legacy classes. Immutable classes in JCoBox are currently very restricted. We are working on a more flexible definition based on the ideas of existing work [15, 11, 14]. JCoBox programs are written in a Java extension and are compiled to Java code. This has the disadvantage that standard Java tools that work on the source code cannot be used for JCoBox source code. We are currently working on a byte-code rewriting tool, which post-processes byte-code compiled from standard Java code.

6. ACKNOWLEDGMENTS

We thank the anonymous reviewers of IWMSE for their helpful comments. This work has been partially supported by the EU-project FP7-231620 HATS: Highly Adaptable and Trustworthy Software using Formal Methods.

7. REFERENCES

[1] E. Ábrahám, I. Grabe, A. Grüner, and M. Steffen. Behavioral interface description of an object-oriented language with futures and promises. *Journal of Logic and Algebraic Programming*, 78(7):491–518, Aug. 2009.

[2] G. Agha. *Actors: a model of concurrent computation in distributed systems*. MIT Press, 1986.

[3] P. America. Issues in the design of a parallel object-oriented language. *Form. Asp. Comput.*, 1(4):366–411, 1989.

[4] J. Armstrong. *Making reliable distributed systems in the presence of software errors*. PhD thesis, The Royal Institute of Technology, Stockholm, Sweden, 2003.

[5] L. Baduel, F. Baude, D. Caromel, A. Contes, F. Huet, M. Morel, and R. Quilici. Programming, deploying, composing, for the Grid. In J. C. Cunha and O. F. Rana, editors, *Grid Computing: Software Environments and Tools*. Springer, Jan. 2006.

[6] H. G. Baker, Jr. and C. Hewitt. The incremental garbage collection of processes. In *Proceedings of the 1977 symposium on Artificial intelligence and programming languages*, pages 55–59. ACM, Aug. 1977.

[7] J. Bloch. *Effective Java*. Addison-Wesley, second edition, 2008.

[8] B. Bloom, J. Field, N. Nystrom, J. Östlund, G. Richards, R. Strniša, J. Vitek, and T. Wrigstad.

Thorn: robust, concurrent, extensible scripting on the JVM. *SIGPLAN Not.*, 44(10):117–136, 2009.

[9] D. Caromel. Towards a method of object-oriented concurrent programming. *Communications of the ACM*, 36(9):90–102, 1993.

[10] D. Caromel, L. Henrio, and B. P. Serpette. Asynchronous and deterministic objects. In *Proc. POPL '04*, pages 123–134. ACM, Jan. 2004.

[11] D. Clarke, T. Wrigstad, J. Östlund, and E. B. Johnsen. Minimal ownership for active objects. In *Proc. APLAS '08*, pages 139–154. Springer-Verlag, 2008.

[12] F. S. de Boer, D. Clarke, and E. B. Johnsen. A complete guide to the future. In *ESOP 2007*, volume 4421 of *LNCS*, pages 316–330, Mar. 2007.

[13] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[14] C. Haack and E. Poll. Type-based object immutability with flexible initialization. In S. Drossopoulou, editor, *ECOOP*, volume 5653 of *LNCS*, pages 520–545. Springer, 2009.

[15] C. Haack, E. Poll, J. Schäfer, and A. Schubert. Immutable objects for Java-like languages. In *European Symposium on Programming (ESOP'07)*. Springer, March 2007.

[16] P. Haller and M. Odersky. Scala actors: Unifying thread-based and event-based programming. *Theoretical Computer Science*, 410(2-3):202–220, 2009.

[17] R. H. Halstead, Jr. Multilisp: a language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.*, 7(4):501–538, 1985.

[18] C. Hewitt, P. Bishop, and R. Steiger. A universal modular ACTOR formalism for artificial intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, pages 235–245. William Kaufmann, Aug. 1973.

[19] JCoBox website. `http://softech.cs.uni-kl.de/~jcobox`, 2010.

[20] E. B. Johnsen, J. C. Blanchette, M. Kyas, and O. Owe. Intra-object versus inter-object: Concurrency and reasoning in creol. *Electron. Notes Theor. Comput. Sci.*, 243:89–103, 2009.

[21] E. B. Johnsen and O. Owe. An asynchronous communication model for distributed concurrent objects. *Software and Systems Modeling*, 6(1):35–58, Mar. 2007.

[22] H. Lieberman. Concurrent object-oriented programming in Act 1. In A. Yonezawa and M. Tokoro, editors, *Object-Oriented Concurrent Programming*, pages 9–36. MIT Press, 1987.

[23] M. S. Miller, E. D. Tribble, and J. S. Shapiro. Concurrency among strangers. In R. D. Nicola and D. Sangiorgi, editors, *Proc. TGC'05*, volume 3705 of *LNCS*, pages 195–229. Springer, 2005.

[24] B. Morris. Cactive and friends. *Symbian Developer Network*, June 2008.

[25] N. Nystrom, M. R. Clarkson, and A. C. Myers. Polyglot: An extensible compiler framework for Java. In G. Hedin, editor, *CC*, volume 2622 of *LNCS*, pages 138–152. Springer, 2003.

[26] J. Ousterhout. Why threads are a bad idea (for most purposes), 1996. Invited talk at the 1996 USENIX Conference.

[27] M. M. Papi, M. Ali, T. L. Correa, Jr., J. H. Perkins, and M. D. Ernst. Practical pluggable types for Java. In *ISSTA '08: Proceedings of the 2008 international symposium on Software testing and analysis*, pages 201–212. ACM, 2008.

[28] N. Rodriguez and S. Rossetto. Integrating remote invocations with asynchronism and cooperative multitasking. *Parallel Processing Letters*, 18(1):71–85, 2008.

[29] J. Schäfer and A. Poetzsch-Heffter. JCoBox: Generalizing active objects to concurrent components. to appear, 2010.

[30] S. Srinivasan and A. Mycroft. Kilim: Isolation-typed actors for java. In J. Vitek, editor, *ECOOP*, volume 5142 of *LNCS*, pages 104–128. Springer, July 2008.

[31] H. Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3), Mar. 2005.

[32] The OSGi specification release 4 version 4.2. `http://www.osgi.org`, 2009.

[33] T. Van Cutsem, S. Mostinckx, E. G. Boix, J. Dedecker, and W. D. Meuter. Ambienttalk: Object-oriented event-driven programming in mobile ad hoc networks. In *Proc. SCCC '07*, pages 3–12. IEEE Computer Society, 2007.

[34] A. Yonezawa, J.-P. Briot, and E. Shibayama. Object-oriented concurrent programming ABCL/1. In N. Meyrowitz, editor, *Proc. OOPSLA '86*, pages 258–268. ACM Press, 1986.