Active C#

Raphael Güntensperger Jürg Gutknecht

ETH Zürich
Clausiusstrasse 59
CH-8092 Zürich

guentensperger@inf.ethz.ch gutknecht@inf.ethz.ch

ABSTRACT

Active C# is a variant of Microsoft's C# that enhances the basic language with a direct support for concurrency and a new model for object communication. The C# compiler of the Shared Sources Common Language Infrastructure (SSCLI) served as a basis to extend the compiler. Modifications mainly concern the enhancement of C# with an active object concept and a novel communication paradigm based on formal dialogs.

Keywords

Active C#, Programming Languages, Concurrency, Formal Dialogs, Active Objects, AOS, SSCLI

1. BACKGROUND

The roots of Active C# can be found in a ROTOR project partially funded by Microsoft Research [Gu]. The concept of active objects and their synchronization comes from Active Oberon [Gk], a successor of the Oberon Language and from the Active Object System [Mu], an internally developed operating system microkernel. This paper presents a consolidation and enhancement of an experimental language concept introduced in the aforementioned ROTOR project.

2. OVERVIEW

From a historical perspective, we can easily recognize an evolution of the object concept from purely passive *data records* to re-active, *functional entities*. In our language experiment, we evolve the object concept another step further by adding *encapsulated behavior* and *communication capabilities*.

.NET Technologies'2004 *workshop proceedings,*
ISBN 80-903100-4-4

Active C# is an extension of C# which mainly includes two new technologies: *active objects* and *formal dialogs*.

Both technologies support the seamless integration of threading into the programming model, with the aim of increased acceptance and use of concurrency in programs. The idea is that programmers do not need to call the underlying threading framework directly anymore but can still add concurrency to their programs simply by making appropriate use of the programming model.

Active Objects

An active object is an instance of a class with encapsulated behavior, running one or more separate threads.

In Active C#, this idea is supported by *activities*. a new kind of class members. An activity is a method with an empty parameter list and void result, run as a separate thread. Any number of activities are allowed in a class.

Two kinds of activities exist: unnamed and named. An *unnamed activity* automatically starts after object instantiation and is executed only once per instance, where a *named activity* must be started explicitly and can be executed any number of times. The `static` modifier is also allowed for both kinds of activities and, if chosen, the activity is bound to the type of the object rather than to its instance. This implies that a

static unnamed activity is started when the class is loaded and that these activities run in the static context of the class.

Formal Dialogs

Formal dialogs are a vehicle that allows advanced, syntax-controlled communication between objects, notably between remote objects. Thus, a formal dialog serves as a communication interface from the outer world to an object.

Assuming that some object *b* provides a syntactic specification of dialog D and that object *a* wants to communicate with *b*, this is how it works:

- *a* instantiates dialog D with *b*
- *b* creates a separate thread of control, acting as a symbolic channel of communication between *a* and *b*
- *a* and *b* communicate over a symbolic channel according to the syntactical specification of D

In Active C#, dialog interfaces and implementations are represented by keyword enumerations and parser activities respectively. For example:

```
dialog D { u, v, w }
// keywords

class B {
  …
  activity d: D { … }
  // parser
}
```

3. ACTIVITIES

An activity is defined inside a class and follows this syntax:

["static"] "activity" [name] "{" statements "}".

We recall that unnamed activities are launched automatically at instance creation time. In contrast, named activities must be started explicitly by calling an overloading of the `new` operator:

"new" QualIdent.

By concept, activities always run to their end, and there is no explicit option of aborting an activity. Of course, each activity can still decide to finish its work early depending on some state condition.

Activities are inherited from base classes and run in parallel with activities defined in the derived class. Inherited anonymous activities are therefore started automatically at each instance creation of a derived class.

The automatic starting of threads belonging to anonymous activities is handled by our modified compiler at the end of either the instance constructor or the type constructor.

Synchronization

Normally, object activities run in full concurrency. However, sometimes a certain precondition is needed for continuation. In line with our goal of simplifying concurrent programming and avoiding explicit calls to the threading framework, we use a direct

```
await (condition)
```

statement instead of signals for the reactivation of waiting activities. `condition` is an arbitrary Boolean expression representing the condition to be waited for.

To ensure proper synchronization, the await statement must occur in a context that is locked with respect to the enclosing object and it must refer to a purely object-local condition. In Active C#, we use an overloading of the `lock` statement for this purpose:

```
lock { }
```

whose semantics is given in Figure 1, where *context* refers either to the current object (`this`) or its type descriptor (`typeof(Class)`) and `Monitor` refers to the corresponding type of the .NET Framework's threading library.

```
Monitor.Enter(context);
try {
        // statements
}
finally {
        Monitor.PulseAll(context);
        Monitor.Exit(context);
}
```

Figure 1. The `lock` construct decomposed

The `lock` statement simplifies both the specification of context-locking actions and the implementation of the Active C# compiler. Assuming that object state-changes occur within context-locked sections only, it is reasonable to

- map `await (condition)` to
  `while (!condition)`
  `  Monitor.Wait()`
- generate a
  `Monitor.PulseAll (context)`
  at the end of each lock block

However, interestingly, this is not sufficient. Another `PulseAll (context)` is necessary right before

an unsatisfied await statement suspends its thread for the first time[1]
In summary, all this leads to the decomposition of the await statement shown in Figure 2.

In principle, time-oriented conditions could be handled by await statements of the form

```
await (t >= T)
```

within some *Timer* object. However, for convenience, a special *passivate* statement is provided for this purpose. This is its form and semantics:

```
passivate (duration);
```

where the `duration` parameter specifies the number of milliseconds the current thread is to be suspended. The passivate statement can occur at any places in the code and takes any integer expression as argument.

4. DIALOGS

Our dialog model is based on *formal grammars* that constitute some kind of contract between caller and callee. An element of such a grammar is called a *token*. Each token basically specifies a data type and a direction. Our implementation of formal dialogs associates two buffers with each communication. Tokens sent by the caller are stored in the *input buffer* to be processed by the callee. Conversely, tokens sent by the callee are stored in the output buffer to be processed by the caller. Technically, both buffers are instances of `System.Dialog.DialogBuffer` and implemented as self-expanding ring-buffers.

Encoding and decoding

Because dialogs are designed to be used in remote environments as well, an encoding must be specified and agreed upon for each token type, and a *codec* must be plugged into the sender and receiver program respectively. This system works because the token buffers act as FIFO-queues and therefore allow their contents to be treated as a byte stream.

The current codec supports the C# built-in types `int`, `long`, `float`, `double`, `bool`, `char`, `string`, `byte`, `byte[]`, the new Active C# type `keyword` and an `escape` type used in some formal grammars.

[1] Before suspending a thread after checking the condition of its await statement at all later times, no signal is necessary, because this thread had no possibility to change any condition in the meantime.

```
bool waitingAlready = false;
while(!condition)
{
        if(!waitingAlready)
        {
                Monitor.PulseAll(ref);
                waitingAlready = true;
        }
        Monitor.Wait(ref);
}
```

Figure 2. Decomposition of the await statement

Dialog specification and implementation

A *dialog specification* is an element of a namespace (on the same level as classes and interfaces) and has the following syntax:

> [accmod] ”dialog” DialogTypeName keywords.
> accmod = private | internal | protected | public.
> keywords = ”{” [{ keyword ”,” } keyword] ”}”.

This declaration defines the *dialog type*, including the list of keywords of the underlying grammar. User defined dialog types are always implicitly derived from `System.Dialog.Dialog`, a predefined type that specifies the dialog accessors (see next section) and some references to internal ingredients of a running dialog, such as its buffers.
All keywords are of the new built-in type `keyword`, mapped to the enumeration type `System.Dialog.Keyword`. Their values are used by the sender and receiver, which guarantees an efficient transfer of keyword tokens.
Note that dialog types have a comprehensive character and provide the following infrastructure:

- An enumeration type for keywords
- An interface for a dialog implementation
- The data structure to control a running dialog

A *dialog implementation* is a named activity that implements the corresponding dialog specification. The syntax is familiar from interface implementation:

> [“static”] “activity” ActivityName
> “:” DialogType “{“ statements “}”.

Note that, in the case of activities, a formal syntax consistently replaces the argument list occurring in method declarations. We will use the C# attribute concept to bind a formal syntax to a dialog declaration. An automatic parser generator, which we are implementing in a related project, may read this syntax to produce an appropriate parser.

Dialog operators

In Active C#, four dialog-related operators exist: `new, ~, !, ?` and `??`. In turn, their meaning is create a new dialog instance, close a dialog instance, send a token and receive a token in blocking and unblocking mode respectively.

Not surprisingly, the Active C# compiler and runtime depend on powerful library support for the implementation of dialogs, especially for remote dialogs (see the corresponding section below). We already mentioned the types `System.Dialog.Dialog` and `System.Dialog.DialogBuffer`.

These are the library methods that correspond to the Active C# operators:

- **`constructor`** instantiates a dialog and returns a reference to the instance
- **`close`** explicitly discards a dialog and stops its associated thread
- **`send`** takes an object, encodes it and passes the encoded data to the input buffer
- **`receive`** tries to decode the output buffer and returns an object

The `receive` accessor can be called in two modes. In *blocking* mode, control is given back to the caller only after a complete object has been received, where in the *non-blocking* mode the accessor immediately returns control, however with a possible `null` return value if not enough bytes were available to decode a complete object at the time of invocation.

Two variants `put` and `get` of `send` and `receive` are used within the callee class. They take the dialog reference directly from the thread context and are in-lined by the compiler directly into the parser code. While the accessor methods work with the general `object` type, the compiler automatically casts the received object to the type of the target variable.

Dialog lifecycle control

Activities are launched in Active C# simply by calling their name, qualified by a reference to the object instance or class name (in the case of static activities). In the special case of *dialog* activities, a reference to the launched activity is needed in sending and receiving operations. For this reason, an overloading of the new operator is provided:

ref = ”new” TypeOrRef ”.” ActivityName.

where `TypeOrRef` is the name of the type for a static dialog or a reference to the callee respectively. Note that the reference returned by `new` refers to one specific instance of a dialog and is necessary to specify the context of the communication. Internally (that is, on the callee side) it is registered relative to the activity thread descriptor and loaded in a local variable at the beginning of each method which might potentially make use of it[2], thus the programmer does not have to refer to it explicitly. In this way, the reference to the current dialog instance is available even across method calls. The caller can discard the current instance of a dialog explicitly by calling its destructor:

”~” ref.

Any further access to this dialog would raise an exception.

When the dialog activity terminates regularly, the corresponding thread is discarded and no further communication is possible, although the reference to the dialog instance remains valid.

Communication

The *send* and *receive* operators are designed to take generic arguments of type `object`. Received objects are type-checked and cast back to their actual type. Table 1 shows the communication syntax. *d* denotes a reference to the current dialog and *obj* is the token to be exchanged. On the callee side, the reference to the dialog is implicit.

The use of separate buffers for input and output allows a full-duplex data-flow. The buffer size is increased automatically on demand but can be limited on desire. If the input buffer is full, the next send operation blocks.

Action	By client	In parser context
Send	d!obj;	!obj;
Receive (blocking)	d?obj;	?obj;
Receive (non-blocking)	d??obj;	??obj;

Table 1: Active C# communication syntax

An example

The communication mechanism supported by Active C# really shines when it comes to “stateful” dialogs such as, for example, negotiations. An upgraded version of John Trono’s Santa Claus concurrency exercise [Tr] may illustrate this.

The original version goes like this: Santa Claus sleeps at the North Pole until awakened by either all of the nine reindeer, or by a group of three out of ten

[2] Each method which contains at least one send or receive statement is marked appropriately

```
c = new Coordinator.CoordElves;
while (true) {
   passivate(Christmas.Rnd());
   c!CoordElvesDialog.join;
   c?msg;
   if (msg == CoordElvesDialog.wait)
      if ((Christmas.Rnd() % 3) == 0)
         c!CoordElvesDialog.release;
      else c!CoordElvesDialog.join;
   }
 }
```

Figure 3. Behavior of an elf

elves. He performs one of two indivisible actions: If awakened by the group of reindeer, Santa harnesses them to a sleigh, delivers toys, and finally unharnesses the reindeer who then go on vacation. If awakened by a group of elves, Santa shows them into his office, consults with them on toy R&D, and finally shows them out so they can return to work constructing toys. A waiting group of reindeer must be served by Santa before a waiting group of elves. Since Santa's time is extremely valuable, marshalling the reindeer or elves into a group must not be done by Santa.

The following complication now adds an element of negotiation: If complete groups are waiting for Santa when an elf desires to join, she should be given the option of withdrawing and walking away. Also, if one and the same elf desires to join excessively often, the coordinator should reject her.

While the translation of the original Santa scenario into an elegant C# program is easy, the negotiation added provides a bigger challenge, mainly because no appropriate language construct is readily available. However, using the dialog construct of Active C#, the following solution of uncompromised elegance is straightforward.

Figure 3 shows the behaviour of an elf while Figure 4 depicts the coordinator activity which is the dialog partner of the elf. Note the negotiation which takes place between the two participants.

It is perhaps interesting to compare our full C# program in the Appendix with Ben-Ari's carefully crafted solution [Be] in Ada95 [Ad], albeit without the complication of negotiation.

Remote dialogs

Up to this point, we have concentrated our discussion on dialogs in local contexts, which allows us to refer to callee objects and dialog instances directly via memory references. However, the communication concept is by no means limited to local environments. The two basic upgrades needed to enable remote dialogs are:

```
while (true) {
  ?msg;
  if (eBuild <= groupNo + 2)
    !CoordElvesDialog.reject;
  else {
    if (eGo < eBuild) {
       !CoordElvesDialog.wait;
       ?msg; }
    if (msg ==
       CoordElvesDialog.join) {
       lock { groupNo = eBuild;
          eSize++;
          if (eSize ==
             Christmas.reqElves)
           {  eSize = 0; eBuild++;}
          await (eGo > groupNo);
       }
       !CoordElvesDialog.release;
    }
  }
}
```

Figure 4. Behavior of the elf coordinator

- Use GUIDs instead of memory references for the identification of both the callee object and the current dialog
- Adjust the supporting dialog libraries to make them work on top of some suitable transport layer

See [Gu] for more details.

Summary

We have presented an enhanced variant of C# called Active C#, featuring a new kind of class members called *activity*. Activities provide a uniform tool for two different purposes: specification of active behavior of objects and implementation of dialogs. The rationale behind is a new object model centered around interoperating active objects, in contrast to passive objects that are remote-controlled by threads. Important advantages of the new model are integrated threading and compatibility with remote object scenarios.

While our first experiments with active objects were based on our proprietary language Active Oberon (one activity per object, no dialogs), the ROTOR Shared Source initiative and the availability of the C# compiler in source form (written in C++) allowed us to go a significant step further. The resulting Active C# compiler is fully functional and available [Ac].

Acknowledgement

We gratefully acknowledge the opportunity of developing and implementing our ideas, given to us by Microsoft Research in the context of the ROTOR project.

REFERENCES

[Ac] Active C# Compiler, http://www.cs.inf.ethz.ch/~raphaelg/ACSharp/

[Ad] Intermetrics, Inc., Ada 95 Reference Manual, ISO/IEC 8652:1995.

[Be] M. Ben-Ari, How to solve the santa claus problem. Wiley & Sons, 1997.

[Gk] J. Gutknecht, Do the Fish Really Need Remote Control? A Proposal for Self-Active Objects in Oberon, JMLC 97, p. 207-220.

[Gu] Güntensperger Raphael, Jürg Gutknecht: Activities & Channels, IEE Proceedings, Volume 150, October 2003.

[Ho] C. A. R. Hoare, Communicating Sequential Processes, Prentice Hall, 1985.

[Mu] Muller Pieter Johannes: The Active Object System – Design and Multiprocessor Implementation, Diss. ETH No. 14755, 2002.

[Re] P. Reali, Structuring a Compiler with Active Objects, JMLC 2000, p. 250-262.

[Tr] John A. Trono: A New Exercise in Concurrency, ACM SIGCSE Bulletin, Volume 26, #3, September 1994.

Appendix

Sample Active C# Program: Santa Claus++ (Original by John Trono [Tr])

Santa Claus sleeps at the North Pole until awakened by either all of the nine reindeer, or by a group of three out of ten elves. He performs one of two indivisible actions: If awakened by the group of reindeer, Santa harnesses them to a sleigh, delivers toys, and finally unharnesses the reindeer who then go on vacation. If awakened by a group of elves, Santa shows them into his office, consults with them on toy R&D, and finally shows them out so they can return to work constructing toys. A waiting group of reindeer must be served by Santa before a waiting group of elves. Since Santa's time is extremely valuable, marshalling the reindeer or elves into a group must not be done by Santa.

Complications: If complete groups are waiting for Santa when an elf desires to join, she should be given the option of withdrawing and walking away. Also, if one and the same elf desires to join excessively often, the coordinator should reject her.

```
using System;
using System.Dialog;

namespace SantaClaus
{

    // dialog declarations
    dialog CoordReindeerDialog { join, release }
    dialog CoordElvesDialog { join, reject, wait, release }
    dialog ActivSantaDialog { deliver, consult, done }

    class Reindeer {
        // an unnamed instance activity -> starts when object is instantiated
        activity {
           object msg;
           // create a new dialog instance
           CoordReindeerDialog c = new Coordinator.CoordReindeer;
           while (true) {
               passivate(Christmas.Rnd()); // wait for a random time
               c!CoordReindeerDialog.join; // send the keyword 'join'
               c?msg; // receive whatever is sent
           }
        }
    }

    class Elf {
        activity {
            keyword msg; // a variable of the special type 'keyword'
            CoordElvesDialog c = new Coordinator.CoordElves;
            while (true) {
```

```
            passivate(Christmas.Rnd());
            c!CoordElvesDialog.join;
            c?msg;
            // Note: automatic casting to the target type is done
            // by the compiler
            if (msg == CoordElvesDialog.wait)
              // the elf has to decide by her own what she wants to do now…
              if ((Christmas.Rnd() % 3) == 0) c!CoordElvesDialog.release;
              else c!CoordElvesDialog.join;
        }
    }
}

class Santa {
    const int consultTime = 10, deliverTime = 20;

    static activity ActivSanta : ActivSantaDialog {
        keyword msg;
        while (true) {
            ?msg;
            if (msg == ActivSantaDialog.deliver) {
               Console.WriteLine("Santa delivering toys");
               passivate(deliverTime);
            }
            else {
               // if it is not 'deliver' it must be 'consult'
               Console.WriteLine("Santa consulting");
               passivate(consultTime);
            }
            !ActivSantaDialog.done;  // send the keyword 'done'
        }
    }
}

class Coordinator {
    static int rGo = 0, rBuild = 0, rSize = 0;
    static int eGo = 0, eBuild = 0, eSize = 0;

    static activity CoordReindeer : CoordReindeerDialog {
        object msg;
        int groupNo;
        while (true)
        {
            ?msg;
            // sections with state changes and await statements must be locked
            lock {
                groupNo = rBuild; rSize++;
                if (rSize == Christmas.reqReindeer) {
                    // this group is full, prepare to build a new one
                    rSize = 0; rBuild++; }
                // wait until this group of reindeers comes back
                // from delivering
                await (rGo > groupNo);
            }
            !CoordReindeerDialog.release;
        }
    }

    static activity CoordElves : CoordElvesDialog {
        keyword msg;
        int groupNo = -9999;
        while (true)
        {
            ?msg;
            // an elf is not allowed to join too often
```

```
                if (eBuild <= groupNo + 2) !CoordElvesDialog.reject;
                else {
                    if (eGo < eBuild) {
                        // complete groups are already waiting for santa
                        // let the elf decide to join or to leave
                        !CoordElvesDialog.wait; ?msg;
                    }
                    if (msg == CoordElvesDialog.join) {
                        lock {
                            groupNo = eBuild; eSize++;
                            if (eSize == Christmas.reqElves) {
                                // this group is full, prepare to build a new one
                                eSize = 0; eBuild++; }
                            await (eGo > groupNo);
                        }
                        !CoordElvesDialog.release;
                    }
                }
            }
        }


        static activity {
            object msg;
            ActivSantaDialog c = new Santa.ActivSanta;
            while (true)
            {
                lock {
                    await ((rBuild > rGo) || (eBuild > eGo));
                }
                if (rBuild > rGo) {
                    c!ActivSantaDialog.deliver;
                    c?msg;
                    // the state change of this variable has to appear in a locked
                    // section in order to be recognized by an await statement
                    lock { rGo++; }
                }
                else {
                    c!ActivSantaDialog.consult;
                    c?msg;
                    lock {eGo++; }
                }
            }
        }
    }

    public class Christmas {
        public const int nofReindeer = 9, reqReindeer = 9;
        public const int nofElves = 10, reqElves = 3;
        static Random rnd = new Random();

        public static int Rnd () { return rnd.Next(1000); }

        static void Main() {
           for (int i = 0; i < nofReindeer; i++) new Reindeer ();
           for (int i = 0; i < nofElves; i++) new Elf ();
           new Santa ();
        }
    }
}
```