Modelling Microsoft COM Using π-Calculus

Loe M.G. Feijs

Philips Research Laboratories and EESI TUE
feijs@natlab.research.philips.com and feijs@win.tue.nl

Abstract. We use the π-calculus to model aspects of Microsoft's COM architecture. The paper introduces certain aspects of COM, first using IDL and C++, and then using a sugared version of the π-calculus (with numbers and lists added). Most of the complexities arise in dynamic interface management. We explore using the reduction rules of the calculus to show that two components (a stack and stack-observer) do indeed connect to each other in the required manner.

1 Introduction

There is considerable experience with using formal techniques for modelling and analysis of classical communication protocols, by which we mean those protocols which deal with such issues as splitting and assembling protocol data units, error control and flow control. Languages like CCS [1], ACP [2], LOTOS [3], PSF [4], SDL [5], MSC [6] etc. have proven to be useful for this. The component technology [7, 8] which is emerging presently, brings with it protocols of a slightly different type: they are concerned with dynamic binding and with negotiating about a component's capabilities. Configurations change dynamically and processes not only exchange data, but they also exchange link-names.

Therefore we consider it worthwhile to experiment with the π-calculus [9], which provides precisely this extra expressive power. We apply the π-calculus to key aspects of one of the most successful component technologies presently available: Microsoft's Component Object Model (COM) [10]. This is the basic technology which makes it possible, among other things, to perform run-time negotiations and establish run-time bindings. COM is the basis of what Microsoft calls Active-X, whose forerunner was called OLE (Object Linking and Embedding) [11]. It is Active-X or OLE which allows to copy-paste a bitmap made by MS-Paint into a MS-Word document, and then find that when the bitmap is embedded in the Word document, it still can be edited in a wysiwyg style.

Survey of the paper: in Sect. 2 we present a brief discussion of the relevance of component technology and introductory remarks on Microsoft COM. In Sect. 3 we present a summary of the π-calculus. In Sect. 4 we present the principles of our approach to modelling COM using π-calculus. Sects. 5 and 6 together form the first part of our case study: the former explaining concrete aspects of COM for a component MyStack by using only IDL and C++ as notations, the latter section describing precisely the same aspects but using π-calculus instead of IDL and C++. Then in Sect. 7 we discuss the key aspect of COM (manipulating interface pointers) and in Sect. 8 we extend the formal π-calculus model in order to properly deal with these key aspects of COM as well. A sample calculation using the rules of π-calculus is given in Sect. 9. Finally Sect. 10 contains some concluding remarks. The paper demands no a-priori knowledge of COM or π-calculus: Sects. 5 and 7 introduce COM and Sect. 3 summarises the π-calculus.

Acknowledgements: the author would like to thank Hans Jonkers for the help and cooperation on the subject of this paper; the author also wants to thank the anonymous referees for their helpful comments.

2 Component Technology

The idea of component technology is that custom programs are composed from reusable parts that serve to perform certain sub-tasks. In the model proposed by Microsoft, components are reused in binary executable form. It includes a binary interfacing mechanism which lets components communicate with each other in an open system. There is a need to add components to an open system and to add new interfaces to a system. The connection and communication mechanisms are standardised and do not depend on the specific interfaces themselves. The set of components in a system can change over time. So there are situations where a newer component that is capable of exploiting a certain new interface encounters an old component that does not know about this new interface. For this purpose there is a negotiation mechanism in COM by which one component or application can find out if a desired interface is supported by another component. It makes sense to think of interfaces as formal legal contracts. Implementers must make sure their implementations meet the contracts. In COM, contracts are identified by interface identifiers (IIDs). When components are enhanced, they get new interfaces, while preserving possibly some of the older interfaces. It is also possible to remove some of the older interfaces. The interfaces themselves are not changed. Each IID identifies a contract which may or may not be supported by a given component (or better, by a given object; in this paper we do not go into the distinction between components and objects). Once the IID is fixed and released, no one is supposed to make even the smallest modification to the signature or the semantics of the interface. The signature is fixed by means of the language IDL. Although most of the COM literature insists that the semantics of the interfaces for a given IID be fixed, there is no dedicated specification language for COM and often the semantic aspects of the contracts are not formally specified. Williams in [12] provides an exposé of the ideas of system evolution underlying COM (this paragraph is based on Williams' text).

The internal operation of a COM component is hidden because COM is a binary standard. A COM component is obtained by compilation of for example, a C++ program, or a Java program. The source code is not released for distribution. Usually only the compiled version, i.e. the machine code is released. It is not possible to read to or write from a component's data structures directly. All access must be done via procedure calls. This approach preserves the freedom for choosing another data structure in a next version of the component. Secondly, it is relatively easy to replace a local function call by a call to a stub, which at its turn executes an RPC (remote procedure call). This kind of replacement, which is easy for procedure calls, would not be so easy to realise for direct access to data structures.

All functions (procedures) are grouped into so-called interfaces. An interface is a set of functions whose semantics is somehow related. This resembles the well-known concept of 'signature' from the theory of algebraic datatypes. An interface however only contains functions, no abstract types. There are auxiliary types such as void, long, etc, and also "struct"s or other interface types; but the main type, the component's type itself remains implicit. Usually a component has several interfaces (which is an important difference with algebraic data types).

3 The π-Calculus

The π-calculus was proposed by Milner, Parrow and Walker in 1992. It is also called 'a calculus of mobile processes', but actually no processes are moved around, only the identities of the ports of the processes can be communicated from one process to another. The π-calculus has a certain simplicity which comes from the fact that all distinction between variables and constants has been removed. The main idea is that the calculus is like CCS, except for the fact that not only values are communicated, but also port identifiers.

A very brief summary of the calculus is given here. If p is a port, then $\overline{p}v \,.\, P$ is the process which sends value v along port p and then proceeds as P. Conversely, $p(x) \,.\, Q$ is the process which receives a value over port p, binds the value thus received to x, and proceeds as Q. In the body Q, this x may be used. The special thing about π-calculus is that port identifiers may be sent and received as well. For example $q(p) \,.\, \overline{p}v \,.\, R$ is the process which receives port identifier p via port q, and then uses this p for sending something else, viz. v (and proceeds as R).

Further operators of the π-calculus include $+$ for alternative composition, $|$ for parallel composition, recursive definition, inaction $\mathbf{0}$, silent step τ, matching prefix $[x = y]$ and binding prefix (x). The main rule of computation is that $(... + \overline{y}x.P + ...) \,|\, (... + y(z).Q + ...) \xrightarrow{\tau} P|Q\{x/z\}$.

4 Modelling Approach

The main modelling techniques that we propose and that we shall put into action in Sects. 6 and 8 are the following:

- invocation of a procedure with n input and m output parameters is modelled by $n + 1$ send actions followed by m receive actions. `HRESULT` (handle to a result) and call-by-reference parameters are treated in the same way.
- interface pointers are modelled as π-calculus ports, for example if the one-argument procedure p belongs to interface i, then invoking p is modeled as $\overline{i}p \,.\, \overline{i}a \,.\, i(h) \,.\, \cdots$. So p is the procedure's name, a is its input argument and h is the result (of type `HRESULT`).

– the state-based behaviour of the component is modelled by recursive equations where the various parts of the state are parameters.

5 The 'Interface' Concept of COM

Our example is about a stack. We begin with its main interface (the push and pop behaviour). Each interface has a name. In COM this name always starts with a capital I. So we assume that the usual push and pop functions are inside IManipulate. There is a language for describing interfaces called IDL (Interface Definition Language). For the first interface, we assume a few auxiliary types:

– HRESULT, whose value set has 2^{32} elements, among which are S_OK, S_FALSE, E_NOINTERFACE, E_NOTIMPL and E_FAIL.
– The set ITEM (the values that will be pushed onto the stack), with a value set which is considered not being interesting now (32 bits integers).

For the purpose of classifying the obtained HRESULT values there are two auxiliary functions, FAILED(...) and SUCCEEDED(...), which we fix for the time being by means of equations: FAILED(S_OK) = FALSE, FAILED(S_FALSE) = FALSE, FAILED(E_NOINTERFACE) = TRUE, etcetera. Generally FAILED(S_...) = FALSE and SUCCEEDED(S_...) = TRUE. Conversely FAILED(E_...) = TRUE and SUCCEEDED(E_ ...) = FALSE.

Now we are ready to present our first interface definition in IDL (we have left out a few things, viz. [in], [out] which serve for classifying parameters, and :IUnknown, which indicates inheritance on interfaces).

```
interface IManipulate
{
    HRESULT clear();
    HRESULT is_empty();
    HRESULT push(ITEM i);
    HRESULT pop(ITEM *retval);
}
```

First we discuss the syntax of this specification. The first word, "interface" is a key-word. The second word, "IManipulate" is the name that is given to the newly defined interface. Thereafter, between "{" and "}" there is a set of four function headers. These function headers are denoted with a syntax which resembles C or C++. Recall that in C and C++ declarations always mention the type first, followed by the name of the variable or parameter of that type. So for example HRESULT push(ITEM i); means that push is a function that takes an ITEM value and that yields a HRESULT value. Please note that push is a function in the sense of the programming languages C and C++, that is, a procedure with side-effect. The IDL description only contains signature information; in particular, it does not say which variables are assigned to. For is_empty we can use the distinction between S_OK and S_FALSE to indicate whether the stack is empty (S_OK) or not empty (S_FALSE). Usually this is not recommended, but it is possible. Also note the asterisk in `(ITEM *retval)`: the C or C++ conventions apply. So this function has to be called having as its argument a pointer to a variable in which an `ITEM` will fit. This variable could be called `retval`. In that case `&retval` is a pointer to this variable. Therefore the call `pop(&retval)` has the effect that upon return we find that `retval` contains the `ITEM` value which was first on top of the stack (assuming that `HRESULT` delivered the value `S_OK`).

If we have a stack object, we can perform push and pop operations, etc. But we shall never have objects as such; we will only have direct access to pointers which refer to interfaces. These interface pointers can be dereferenced (which gives us interfaces), and by using an interface we can call `clear`, `is_empty`, `push` and `pop`. Suppose for example that `ps` is a pointer to an `IManipulate` interface of a component with stack behaviour, then we can run the following fragment of C++ (if we prefer Java we have to write "." instead of "`->`"). So this is code of some component or application which has to use a stack.

```
HRESULT hr;
ITEM i,retval;
BOOL test = FALSE;

hr = ps->clear();
if (SUCCEEDED(hr))
{
    hr = ps->push(i);
    if (SUCCEEDED(hr))
    {
       hr = ps->pop(&retval);
       if (SUCCEEDED(hr))
       {
          test = (retval == i);
       }  else // failed to pop
    }  else // failed to push
}   else // failed to clear
```

6 Modelling Interface Behaviour in π-Calculus

We show a recursive definition of a process `MyStack` which models the interface behaviour of the `IManipulate` interface. It is based on the principles of Sect. 4.

The state-based behaviour of the various components is modelled again by recursive process equations where the various parts of the state are carried along as parameters of the processes. Although the pure π-calculus does not provide for built-in data types and process parameters, we assume that these can be simulated thanks to the power of the π-calculus (which is known to simulate full λ-calculus). We assume additional operators `<>`, `<.>` and `++` for lists of items. Here `<>` denotes the empty stack, `<.>` is the operator which makes a one-element list, and `++` denotes concatenation.

The process `MyStack` has two parameters. The first of these, `pIman`, models the interface pointer along which all communication takes place. The second parameter is the contents of the stack.

```
MyStack(pIman,<>) =
( pIman (f) .
  ( [f = clear] .
     pIman S_OK .
     MyStack(pIman,<>)
  + [f = is_empty] .
     pIman S_OK .
     MyStack(pIman,<>)
  + [f = push] .
     pIman (j) .
     pIman S_OK .
     MyStack(pIman,<j>)
  + [f = pop] .
     pIman E_FAIL .
     MyStack(pIman,<>)
)  )

MyStack(pIman,<i>++s) =
( pIman (f) .
 ( [f = clear] .
  pIman S_OK .
  MyStack(pIman,<>)
 + [f = is_empty] .
  pIman S_FALSE .
  MyStack(pIman,<i>++s)
 + [f = push] .
  pIman (j) .
  pIman S_OK .
  MyStack(pIman,<j>++<i>++s)
 + [f = pop] .
  pIman S_OK .
  pIman i .
  MyStack(pIman,s)
) )
```

There is nothing special about this model yet. It could be written in CCS or any process algebraic formalism. But in the next section we present other aspects and other examples in COM, the modelling of which becomes more interesting.

7 Manipulating COM Interface Pointers

An important question to be addressed now is: "how do we get an interface pointer of a stack"? There are two answers: (1) get it from somebody else, (2) create it yourself.

For the first case (got it from somebody else) it is best to first perform some querying in order to make sure that we have a valid interface pointer of the desired interface identifier (IID); there is a general mechanism for that. For a given interface pointer it is possible to ask in a dynamic way whether its component has an interface with `IManipulate` behaviour (that is, whether the component implements stack behaviour). This asking in a dynamic way is important because objects are created in a dynamic way, on different machines, possibly from different or even incompatible component versions. So it is very essential first to find out more about the component behind a given pointer. This asking/testing mechanism makes it possible to obtain other interface pointers once we have the first one that belongs to a certain component. The mechanism is implemented as a procedure which is called `QueryInterface`.

For the second case it is possible to get hold of an interface pointer via a so-called "factory". Once we have the first, we can use `QueryIterface` to get the others. First we discuss the `QueryInterface` mechanism. Each component supports one interface that is obligatory: `IUnknown`. Later we shall show more of our component `MyStack` which happens to have three interfaces. These will be called `IUnknown`, `IManipulate` and `IOverflow` (see figure). The interface `IUnknown` has to be supported by every component. There is no language construct to express that `MyStack` has these three interfaces.

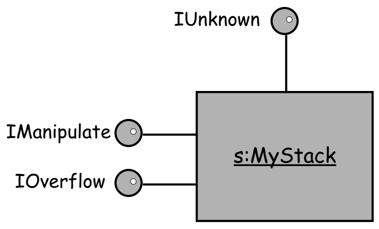

Fig. 1. MyStack object having three interfaces.

`MyStack` is a concrete component which contains a 'class', of which instances can be created. In Figure 1 we show an instance (which is why we underlined s:MyStack). The asking/testing mechanism is provided by `IUnknown`, which has the following IDL:

```
interface IUnknown
{
    HRESULT QueryInterface(REFIID iid, void** ppv);
    HRESULT AddRef();
    HRESULT Release();
}
```

`REFIID` is the type of pointers to `IID`, where `IID` is the type of interface identifiers. Interface identifiers are statically determined identifications. Note: this `iid` must be viewed as an input parameter; the fact that `REFIID` is a pointer itself is only an efficiency trick which amounts to a call-by-reference mechanism. The second parameter yields an untyped pointer (an interface pointer). This `QueryInterface` function embodies the mentioned asking/testing mechanism.

What comes next is an intermezzo about interface identifiers and interface pointers. Interface pointers indicate specific instances of interfaces, associated with corresponding object instances, whereas interface identifiers effectively give the "type" of the interface. An interface identifier is obtained statically using a special number generator (uuidgen, to be run by the interface designer, no central registering). This could yield 6A92D9A0-C04D-11D3-A11B-00A024674DFA for `IManipulate`. In the same way `IUnknown` has its own interface identifier, but this is always the same, on all machines, viz. 00000000-0000-0000-C000000000000046. This takes care of all numbers being unique. Using the number generator at another point in time or at another machine yields a different number. This number 00000077-0000-0000-C000-000000000048 could be the `IID` that belongs to all `IManipulate` interfaces. If we have ten stacks then we have 30 interfaces: ten of the first `IID` (00000000-0000-0000-C000-000000000046), and ten of the `IID` of `IManipulate,` and yet another 10 of that of `IOverflow.` But all 30 of them have another interface, and hence another interface pointer.

Next we put the asking/testing mechanism into operation. Let us presuppose constants for the interface identifiers, typically fixed by a `#define IID_IUNKNOWN 00000000-0000- 0000-C000-000000000046`, etc. Now assume that the factory has given us a pointer, `pStack` say, as in the following program fragment:

```
void* pStack;
pStack = ... // from the factory
```

Then we can test `pStack` by asking if indeed the `IID` of `IUnknown` is known.

```
void* pStack_;
HRESULT hr;
hr = pStack->QueryInterface(IID_IUNKNOWN, &pStack_);
```

If `hr` equals `S_OK`, or if it is one of the other `S_` values, then we know that we have got an `IUnknown` interface pointer. Besides that, `QueryInterface` also provides a result, in this case in `pStack_`, and if all is right, this is again a pointer to the same interface as `pStack`. Although this was a nice test, it does not seem to advance us much. So next we shall use `QueryInterface` to obtain a pointer to another interface, `IManipulate`. This works the same as just before: call `QueryInterface`, but now giving it the `IID` (obtained once from the special number generator) of `IManipulate`

```
void* pIman;
HRESULT hr;
hr = pStack->QueryInterface(IID_IMANIPULATE, &pIman);
```

If `hr` equals `S_OK`, or one of the other `S_` values, then we know that in `pIman` we have got an `IManipulate` interface pointer. We may assume that a stack has been created (or at least something else that implements `IManipulate` behaviour). Now we are ready to use this object.

```
hr = pIman->clear();
hr = pIman->push(i);
hr = pIman->pop(&retval);
```

This was quite involved, but the advantage is that, starting from a suspect pointer, which may or may not come from the appropriate "factory", we have verified that it belongs to an object with stack behaviour. And in this way we arrived at functions for which there is no more reason to doubt that they will meet our expectations.

Now let us have a look at the third interface of `MyStack`, the `IOverflow`. The idea is that it is employed for connecting "callbacks". We imagine that heavy usage of the push operation (much more push'es than pop's) could lead to an overflow of the stack. In fact, each stack has only a limited memory capacity (1000 items say) and if this is exceeded, the normal stack behaviour can no longer be guaranteed. It should be tried to prevent this, which is better than trying to restore a stack where the damage has already occurred. Therefore we assume that there is another component, for example called `MyStackObserver`, which has to be warned whenever the threat for overflow occurs (`MyStackObserver` is only a concrete example, in fact we are concerned with the general idea of a component which observes a stack). Of course it would be possible to have the calls of the observer's procedures "hard-coded" in `MyStack`. But assume that we refrain from doing so, and instead of that let us demand that objects with stack behaviour work for arbitrary observers, not just this specific `MyStackObserver`. So an arbitrary component must be able to subscribe to warnings concerning stack overflow. Therefore such an object (e.g. `MyStackObserver`) must tell the object with stack behaviour which procedure must be called if an overflow threat occurs and which procedure must be called if an overflow happened nevertheless. In this context we call these procedures of the observer "callback procedures". In general there may be several callbacks procedures; it is COM-style to group them into an interface. In our running example we choose for naming this interface `IStackObserver`; it has to be implemented by `MyStackObserver`.

In this example we want two call back procedures, `onStackHalfFull()` for when the stack is about half full and `onStackOverflow()` for handling a real overflow. We give the IDL description of this interface:

```
interface IObserver
{
    HRESULT onStackHalfFull();
    HRESULT onStackOverflow();
}
```

`MyStackObserver` has to "inform" the stack object of these two procedures, but it will not do so for each procedure separately. It does so in a single step, namely by sending its `IStackObserver` interface pointer to the object with stack behaviour. Now the problem of "informing" has been reduced to transferring an interface pointer. That is easy if we choose the `IOverflow` interface as follows:

```
interface IOverflow
{
    HRESULT subscribe(int p, IStackObserver* obs);
    HRESULT unsubscribe(IStackObserver* obs);
}
```

The parameter p of subscribe indicates at which percentage of the stack space the warning is generated. For example if p equals 50 then the warning will come when precisely half of the available stack space has been used up. Now the intention of all this is that two component instances will get connected as shown in Figure 2 below.

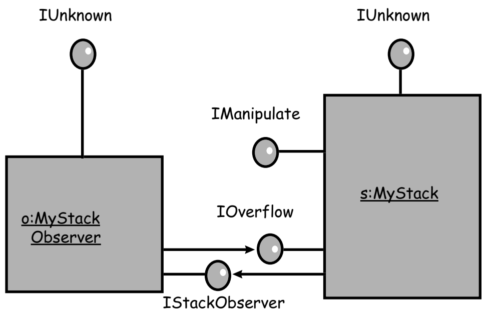

Fig. 2. MyStackObserver and MyStack coupled.

The arrow from MyStack to the lollipop of IStackObserver indicates that object s, being an instantiation of component MyStack can perform calls to procedures of the object IStackObserver. Whereas IOverflow is an *incoming interface* of the object with stack behaviour, we say that IStackObserver is an *outgoing interface* of it. Somewhere in the initialisation of MyStackObserver there is a call of subscribe(...) as we shall show in the corresponding program fragment given below. Let us assume that pStack is pointing to the IUnknown interface of an object with stack behaviour. Also assume that IID_IOVERFLOW is defined by means of a #define.

```
IUnknown * pStack;     // IUnknown pointer of e.g. MyStack      (given)
IStackObserver* pIobs; // interface pointer of observator self  (given)
IOverflow* pIovr;      // interface pointer           (to be filled in)
HRESULT hr;

hr = pStack->QueryInterface(IID_IOVERFLOW, &pIovr);
if SUCCEEDED(hr) {
   hr = pIovr->subscribe(50, pIobs);
   if SUCCEEDED(hr) {
      // coupling made
   } else ...
} else ...
```

We assume that somewhere inside the object with stack behaviour this value 50 is stored, for example in a variable called warninglevel. We also assume that this object with stack behaviour can only deal with one subscriber, whose IStackObserver interface pointer is kept in the variable pIobs (internally in the object with stack behaviour). So the implementation of subscribe, possibly being a part of the implementation of MyStack, could look as follows:

```
int warninglevel;           //            (to be filled in)
IStackObserver* pIobs;      // subscriber (to be filled in)

HRESULT subscribe(int p, IStackObserver* obs)
{   warninglevel = p
    pIobs = obs;
    return S_OK;
}
```

Figure 3 below illustrates the entire structure of pointers built-up in this way. In this state we find that the system consisting of MyStackObserver and MyStack is sufficiently coupled in order that the operational behaviour of the component with stack behaviour can begin.

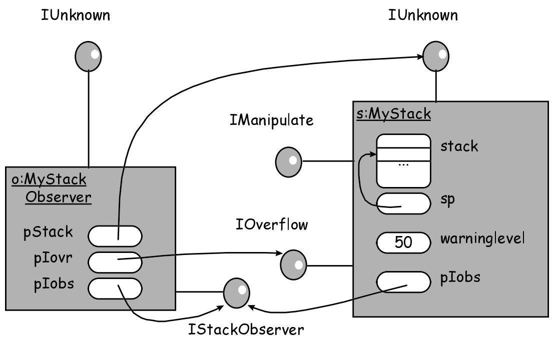

Fig. 3. Implementation of MyStackObserver and MyStack coupling.

It can be seen how the outgoing arrows of Figure 2 are nothing but abstractions of the implementation-level arrows (that is, pointers). Of course Figure 3 is not suited for specification purposes because it reveals too much implementation detail. It is an illustration of one of the possible ways the implementation could work; but variables such as pStack, pIovr, pIobs, stack, sp, warninglevel and (the other) pIobs are not visible from the ouside of the component.

Once the coupling has been established, the object with stack behaviour can perform calls of onStackHalfFull() and onStackOverflow(). Let us have a look at a possible implementation of push(...) inside MyStack. We repeat the declarations of warninglevel and pIobs.

```
#define MAX 1000

int warninglevel;      //              (given)
IStackObserver* pIobs; // subscriber  (given)
```

```
int sp;                 // stack pointer
ITEM stack[MAX];        // contents of the stack
HRESULT hr;

HRESULT push(Item i)
{   if (sp >= MAX) {
       hr = pIobs->onStackOverflow();
       return E_FAIL;
    }
    else {
       if (sp >= warninglevel*(MAX / 100)) {
          hr = pIobs->onStackHalfFull();
       }
       stack[sp++] = i;
       return S_OK;
}   }
```

By now it should be clear that Figure 1 (stack behaviour with three interfaces) is somehow incomplete: the view of an object with stack behaviour is only complete if we include its outgoing interface as well. Whenever we want to fix a contract concerning stack behaviour we have to describe `IUnknown`, `IManipulate`, `IOverflow` and `IStackObserver`. Then we have an interface suite of stack behaviour which is independent of the context. Only in this way it may become possible to have a complete specification of the suite (and hence of a component that supports that suite).

Although everybody is free to invent new interfaces and make agreements on their usage, there are a number of standard interfaces which are themselves part of the COM framework. Next to `IUnknown` which was discussed above, the following four interfaces are frequently used; these belong together, providing a general mechanism for binding all kinds of subscribers to components that will perform callbacks: `IConnectionPoint`, `IConnectionPointContainer`, `IEnumConnectionPoints` and `IEnumConnections`. They resemble `IOverflow`, but are much more general. The basic idea is that for each outgoing interface (such as `IStackObserver`) there is an extra incoming interface that offers the possibility of subscribing to certain events (coupling them to callback functions). This extra incoming interface is called `IConnectionPoint`. It makes it possible to have more than one subscriber. Moreover, `IConnectionPoint` is standardised: there is no need to invent from scratch what the interface will look like. The interface can always be the same, quite independently of the precise nature of the outgoing interface itself.

8 Modelling COM Interface Manipulation in π-Calculus

In this section we present a formal model of `MyStack` which support the interfaces `IUnknown`, `IManipulate` and `IOverflow`. We also show parts of `MyStackObserver` which supports the interfaces `IUnknown` and `IStackObserver`. Finally we show a part of `StackUser`, which supports no interfaces but which does have a certain active behaviour. As before, we assume operators `<>`, `<.>` and `++` for lists of items. Moreover, if s is a list of items, we let $|s|$ be the length of the list. We assume 0,1,... and + for natural numbers. We assume `IID_IUNKNOWN`, `IID_IMANIPULATE`, `IID_IOVERFLOW` and `IID_ISTACKOBSERVER` for interface identifiers. And we assume i_0 to be some value of type `ITEM`. We adopted a simplification, viz. to have only one reference counter keeping the total number of references to any of the interfaces of the object (this is done often although conceptually there is one counter per interface). The present model does not build further on the model of Sect. 6, we just start from scratch again. There is one COM feature which we have left out in order to simplify the presentation; this is the fact that all interfaces 'inherit' from `IUnknown`.

```
MyStack(pIunk,pIman,pIovr,pIobs,refs,stack,wl) =
  ( IUnknown(pIunk,pIman,pIovr,pIobs,refs,stack,wl)
  + IManipulate(pIunk,pIman,pIovr,pIobs,refs,stack,wl)
  + IOverflow(pIunk,pIman,pIovr,pIobs,refs,stack,wl)
  )
```

The state-based behaviour of the various components is modelled again by recursive process equations where the various parts of the state are parameters of the processes. The remarks of Sect. 6 apply here too.

```
IUnknown(pIunk,pIman,pIovr,pIobs,refs,stack,wl) =
( pIunk (f) .
  ( [f = QueryInterface] .
    pIunk (iid) .
    ( [iid = IID_IUNKNOWN]
      pIunk S_OK .
      pIunk pIunk .
      MyStack(pIunk,pIman,pIovr,pIobs,refs + 1,stack,wl)
    + [iid = IID_IMANIPULATE]
      pIunk S_OK .
      pIunk pIman .
      MyStack(pIunk,pIman,pIovr,pIobs,refs + 1,stack,wl)
    + [iid = IID_IOVERFLOW]
      pIunk S_OK .
      pIunk pIovr .
      MyStack(pIunk,pIman,pIovr,pIobs,refs + 1,stack,wl)
    + ["otherwise"]
      pIunk E_NOINTERFACE .
      pIunk NULL .
      MyStack(pIunk,pIman,pIovr,pIobs,refs,stack,wl)
    )
  + [f = AddRef] .
    pIunk S_OK .
    MyStack(pIunk,pIman,pIovr,pIobs,refs + 1,stack,wl)
  + [f = Release] .
    ( [refs = 1]
      pIunk S_OK .
      0
    + [refs > 1]
      pIunk S_OK .
      MyStack(pIunk,pIman,pIovr,pIobs,refs - 1,stack,wl)
) ) )

IManipulate(pIunk,pIman,pIovr,pIobs,refs,<>,wl) =
( pIman (f) .
    ( [f = clear] .
      pIman S_OK .
      MyStack(pIunk,pIman,pIovr,pIobs,refs,<>,wl)
    + [f = is_empty] .
      pIman S_OK .
      MyStack(pIunk,pIman,pIovr,pIobs,refs,<>,wl)
    + [f = push] .
      pIman (j) .
      pIman S_OK .
      MyStack(pIunk,pIman,pIovr,pIobs,refs,<j>,wl)
    + [f = pop] .
      pIman E_FAIL .
      MyStack(pIunk,pIman,pIovr,pIobs,refs,<>,wl)
)   )

IManipulate(pIunk,pIman,pIovr,pIobs,refs,<i>++s,wl) =
( pIman (f) .
  ( [f = clear] .
    pIman S_OK .
    MyStack(pIunk,pIman,pIovr,pIobs,refs,<>,wl)
  + [f = is_empty] .
    pIman S_FALSE .
    MyStack(pIunk,pIman,pIovr,pIobs,refs,<i>++s,wl)
  + [f = push] .
    pIman (j) .
    ( [|<i>++s| ≥ MAX]
      pIobs onStackOverflow .
      pIobs (h) .
      pIman E_FAIL .
      MyStack(pIunk,pIman,pIovr,pIobs,refs,<i>++s,wl)
    + [|<i>++s| < MAX]
      ( [|<i>++s| ≥ wl*(MAX/100)]
        pIobs onStackHalfFull .
        pIobs (h) .
        pIman S_OK .
        MyStack(pIunk,pIman,pIovr,pIobs,refs,<j>++<i>++s,wl)
      + [|<i>++s| < wl*(MAX/100)]
        pIman S_OK .
        MyStack(pIunk,pIman,pIovr,pIobs,refs,<j>++<i>++s,wl)
      )
    )
  + [f = pop] .
    pIman S_OK .
    $\overline{\text{pIman}}$ i .
    MyStack(pIunk,pIman,pIovr,pIobs,refs,s,wl)
) )
```

For `IOverflow` we only show the `subscribe` procedure; because of space limitations we leave out our earlier `unsubscribe` (which poses no special problems).

```
IOverflow(pStack,pIman,pIovr,pIobs,refs,stack,wl) =
( pIovr (f) .
  [f = subscribe] .
  pIovr (w) .
  pIovr (b) .
  $\overline{\text{pIovr}}$ S_OK .
  MyStack(pStack,pIman,pIovr,b,refs,stack,w)
)
```

Next we present `MyStackObserver`, which is described by a few initialisation steps where the subscription takes place, followed by `MyStackObserverCont` (for continuation) which is described by recursion. Note that `MyStackObserver` supports two interfaces.

```
MyStackObserver(pIunk,pIobs,pStack,refs) =
  $\overline{\text{pStack}}$ QueryInterface .
  $\overline{\text{pStack}}$ IID_IOVERFLOW .
  pStack (h) .
  pStack (pIovr) .
  $\overline{\text{pIovr}}$ subscribe .
  $\overline{\text{pIovr}}$ 50 .
  $\overline{\text{pIovr}}$ pIobs .
  pIovr (h) .
  MyStackObserverCont(pIunk,pIobs,pStack,pIovr,refs,0,0)

MyStackObserverCont(pIunk,pIobs,pStack,pIovr,refs,x,y) =
  ( IUnknown'(pIunk,pIobs,pStack,pIovr,refs,x,y)
  + IStackObserver(pIunk,pIobs,pStack,pIovr,refs,x,y)
  )
```

Next we present `IUnknown'`, which is the implementation of COM's `IUnknown` interface for the stack observer. Note that although it is said that each component has to implement COM's `IUnknown`, we see that the implementation of this `IUnknown'` is slightly different from the `IUnknown` given before, just because `MyStackObserver` has different interfaces than `MyStack`.

```
IUnknown'(pIunk,pIobs,pStack,pIovr,refs,x,y) =
( pIunk (f) .
  ( [f = QueryInterface] .
    pIunk (iid) .
    ( [iid = IID_IUNKNOWN]
      $\overline{\text{pIunk}}$ S_OK .
      $\overline{\text{pIunk}}$ pIunk .
      MyStackObserverCont(pIunk,pIobs,pStack,pIovr,refs + 1,x,y) =
    + [iid = IID_ISTACKOBSERVER]
      pIunk S_OK .
      pIunk pIobs .
      MyStackObserverCont(pIunk,pIobs,pStack,pIovr,refs + 1,x,y) =
    + ["otherwise"]
      pIunk E_FAIL .
      MyStackObserverCont(pIunk,pIobs,pStack,pIovr,refs,x,y) =
    )
  + [f = AddRef] .
    pIunk S_OK .
    MyStackObserverCont(pIunk,pIobs,pStack,pIovr,refs + 1,x,y) =
  + [f = Release] .
    ( [refs = 1]
      pIunk S_OK .
      0
    + [refs > 1]
      pIunk S_OK .
      MyStackObserverCont(pIunk,pIobs,pStack,pIovr,refs - 1,x,y) =
) ) )

IStackObserver(pIunk,pIobs,pStack,pIovr,refs,x,y) =
( pIobs (f) .
  ( [f = onStackHalfFull] .
    pIobs S_OK .
    IStackObserver(pIunk,pIobs,pStack,pIovr,refs,x + 1,y)
  + [f = onStackOverflow] .
    pIobs S_OK .
    IStackObserver(pIunk,pIobs,pStack,pIovr,refs,x,y + 1)
) )
```

Now we may compose a system out of various instances of these components. We show the obvious combination having one instance of each. So we assume three initial interface pointers to the three interfaces of MyStack. We also assume two interface pointers to the two interfaces of MyStackObserver. Finally we assume one interface pointer to the IUnknown interface of StackUser. Of course all these six interface pointers are different. Let these initial interface pointers be called PSTACK, PIMAN, PIOVR, PIUNK, PIOBS and PUSER, respectively. Upon initialisation, the MyStack instance only knows its own interfaces, whereas MyStackObserver and StackUser know, next to their own interfaces, also the IUnknown interface pointer of the instance of MyStack.

```
System = (  MyStack(PSTACK,PIMAN,PIOVR,NULL,1,<>,100)
         |  MyStackObserver(PIUNK,PIOBS,PSTACK,1)
         |  StackUser(PUSER,PSTACK)
         )
```

9 Calculations

In this section we show an example of a calculation. This shows one way of using the formal model. Let us consider only the first two parallel components of System, leaving out the stack user. Now we are ready to do some calculation work.

```
  (  MyStack(PSTACK,PIMAN,PIOVR,NULL,1,<>,100)
  |  MyStackObserver(PIUNK,PIOBS,PSTACK,1)
  )
```

$=$

```
  ( ( PSTACK (f) .
      ( [f = QueryInterface]
        PSTACK (iid) .
        ( [iid = IID_IOVERFLOW]
          P̅S̅T̅A̅C̅K̅ S_OK .
          P̅S̅T̅A̅C̅K̅ PIOVR .
          MyStack(PSTACK,PIMAN,PIOVR,NULL,2,<>,100)
        + [‘‘other iid values’’] ...
        )
      + [‘‘other f values’’] . ...
      )
    + PIMAN (f) . ...
    + PIOVR (f) . ...
    )
  | ( P̅S̅T̅A̅C̅K̅ QueryInterface .
      P̅S̅T̅A̅C̅K̅ IID_IOVERFLOW .
      PSTACK (h) .
      PSTACK (pIovr) .
      p̅I̅o̅v̅r̅ subscribe .
      p̅I̅o̅v̅r̅ 50 .
      p̅I̅o̅v̅r̅ PIOBS .
      pIovr (h) .
      MyStackObserverCont(PIUNK,PIOBS,PSTACK,pIovr,1,0,0)
  ) )
```

$\xrightarrow{\tau}$

```
  ( PSTACK (iid) .
    ( [iid = IID_IOVERFLOW]
      P̅S̅T̅A̅C̅K̅ S_OK .
      P̅S̅T̅A̅C̅K̅ PIOVR .
      MyStack(PSTACK,PIMAN,PIOVR,NULL,2,<>,100)
    + [‘‘other iid values’’] ...
    )
  | ( P̅S̅T̅A̅C̅K̅ IID_IOVERFLOW .
      PSTACK (h) .
      PSTACK (pIovr) .
      p̅I̅o̅v̅r̅ subscribe .
      p̅I̅o̅v̅r̅ 50 .
      p̅I̅o̅v̅r̅ PIOBS .
      pIovr (h) .
      MyStackObserverCont(PIUNK,PIOBS,PSTACK,pIovr,1,0,0)
  ) )
```

$\xrightarrow{\tau}$
$\xrightarrow{\tau}$

$$\begin{array}{l}
\texttt{((}\overline{\texttt{PSTACK}}\texttt{ PIOVR .}\\
\qquad\texttt{MyStack(PSTACK,PIMAN,PIOVR,NULL,2,<>,100)}\\
\quad\texttt{)}\\
\texttt{| (PSTACK (pIovr) .}\\
\qquad\overline{\texttt{pIovr}}\texttt{ subscribe .}\\
\qquad\overline{\texttt{pIovr}}\texttt{ 50 .}\\
\qquad\overline{\texttt{pIovr}}\texttt{ PIOBS .}\\
\qquad\texttt{pIovr (h) .}\\
\qquad\texttt{MyStackObserverCont(PIUNK,PIOBS,PSTACK,pIovr,1,0,0)}\\
\texttt{))}
\end{array}$$

$\xrightarrow{\tau}$

$$\begin{array}{l}
\texttt{(MyStack(PSTACK,PIMAN,PIOVR,NULL,2,<>,100)}\\
\texttt{| (}\overline{\texttt{PIOVR}}\texttt{ subscribe .}\\
\qquad\overline{\texttt{PIOVR}}\texttt{ 50 .}\\
\qquad\overline{\texttt{PIOVR}}\texttt{ PIOBS .}\\
\qquad\texttt{pIovr (h) .}\\
\qquad\texttt{MyStackObserverCont(PIUNK,PIOBS,PSTACK,PIOVR,1,0,0)}\\
\texttt{))}
\end{array}$$

This can be interpreted as: the composition of `MyStack(PSTACK,PIMAN, PIOVR, NULL, 1,<>,100)` and `MyStackObserver(PIUNK,PIOBS,PSTACK,1)` can evolve to the situation of Fig. 4. The calculation result represents the state where the link indicated by the arrow from the `pIovr` variable of `o:MyStackObserver` to the `IOverflow` lollipop of `s:MyStack` has been established. This means that we have the situation of Figure 4.

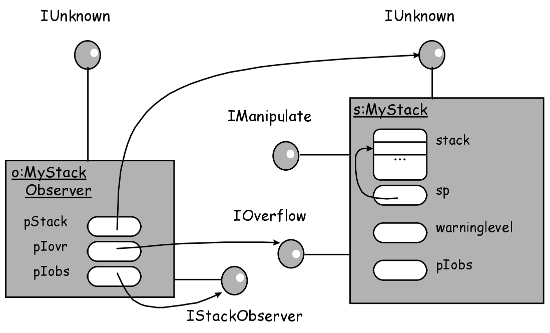

Fig. 4. MyStackObserver and MyStack partially coupled.

The link from the `pIobs` variable in `s:MyStack` to the lollipop of `IStackObserver` has not been established yet (that is why the `MyStack` term still has one `NULL` argument), but of course this is what will happen next if we would continue our calculation. The arrow from `pStack`, and `pIobs` of `o:MyStackObserver` were assumed to be available from the very beginning; these are `PSTACK` and `PIOBS`. continuing the calculation, the situation of Figure 3 will be reached in a finite number of steps. It will be reached necessarily because there are no alternative receive constructs that match the names sent.

So the above calculation shows one way of exploiting the model. In general, the exploitation can be done in various ways analogous to the exploitation of classical communication protocol models in CCS and other process algebraic formalism such as ACP, μCRL, PSF, LOTOS:

– equational reasoning to show behavioural congruence of specifications and implementations (there is a theory of bisimulation for the π-calculus),
– simulation to demonstrate, visualise or test the operational behaviour of a given model in a given context (as demonstrated above).

10 Concluding Remarks

The modelling of COM mechanisms turned out easy and natural (interface pointer manipulations and π-calculcus have good semantic match). The case study was about modelling a component with stack-manipulation behaviour, the obligatory `QueryInterface` behaviour, and an 'observer' which is more or less similar to the well-known observer pattern [13].

Related work: Kramer and Magee defined the ADL called Darwin [14]. It is a combination of a Module Interconnection Language (MIL) and a behavioural specification language. A key ingredients of the MIL part of Darwin is the 'bind' construct: r `--` p means that a required service r is bound to a provided service p. It gets its semantics via π-calculus as follows: to the semantic models of r and p a special agent is added (as a component in a parallel composition); the task of the agent is to send the name of p to r. See [15]. A special elaboration algorithm guarantees that all the bindings specified in Darwin lead to the desired exchange of names. A difference with our work is that we use no intermediate ADL with built-in solutions for the exchange of names.

Sullivan et al. [16] model aspects of COM using Z. Components are modelled as consisting of a finite set of interfaces, a corresponding set of IIDs, and an `iunknown` interface (which is an element of this former finite set). Every interface supports the `QueryInterface` operation, which is modelled as a partial function QI that maps the interface and a given IID to another type. They show how formal specification techniques help in explaining and analysing the complexities of COM. Neither COM interfaces nor COM function calls are mapped directly to Z schemas (indirections are modelled as functions, e.g. QI).

Other interesting references include [17] (OO notation $\pi o\beta\lambda$ based on π-calculus), [18] (a research program for component frameworks, including a discussion on use of π-calculus for open systems components) and [19] (components are interactive systems communicating asynchronously through channels).

There are several issues not addressed but worth further investigation: adding features to π-calculus, concurrency aspects (see the notes on molecular actions and private names in [9]), and re-entrant procedures. The present paper is an exercise in trying to understand component-technology. We do not yet advocate the direct usage of π-calculus. Most of the semantic aspects of interfaces can be described well by languages in the tradition of VDM [20], Z [21] and COLD [22], but there may be a need for special syntactic sugar and special methodological and tool-based support.

References

[1] Milner, R.: Communication and concurrency, Prentice Hall (1989)

[2] Bergstra, J.A., Klop, J.W.: Process algebra for synchronous communication. Information and Computation, **60**(1/3):109-137 (1984)

[3] Bolognesi, T., Brinksma, E.: Introduction to the ISO specification language LOTOS, Computer Networks and ISDN Systems, **14**, (1987) 25–59

[4] Mauw, S., Veltink, G.J. (Eds.): Algebraic specification of communication protocols, Cambridge Tracts in Theoretical Comp. Sc. **36**, CUP (1993)

[5] CCITT. Specification and Description Language (SDL), Rec. Z.100

[6] CCITT. Message Sequence Chart (MSC), Rec. Z.120, Study Group X (1996)

[7] Szyperski, C.: Component Software, Beyond Object-oriented Programming, Addisson Wesley, ISBN 0-201-17888-5

[8] Orfali, R., Harkey, D., Edwards, J.: The essential distributed objects survival guide, John Wiley & Sons, Inc. (1996)

[9] Milner, R., Parrow, J., Walker, D.: A Calculus of Mobile Processes Pt.1 Information and Computation **100**(1) (1992) 1–40

[10] Microsoft Corporation. The Component Object Model Specification, Version 0.9, Microsoft (1995)

[11] Brockschmidt, K.: How OLE and COM solve the problems of component software design, Microsoft Systems Journal, (1996) 63–80

[12] Williams, T.: Reusable Components for Evolving Systems, IEEE 1998 Software Reuse Conference (pp. 12–16)

[13] Gamma, E.,, Helm, R., Johnson, R., Vlissides, J.: Design patterns: elements of reusable object-oriented software, Addison-Wesley (1994)

[14] Magee, J., Kramer, J.: Dynamic Structure in Software Architectures, in: Proc. 4th ACM SIGSOFT Symp. on the Foundations of Software Engineering

[15] Eisenbach, S., Paterson, R.: pi-Calculus semantics for the concurrent configuration language Darwin, Hawaii Int. Conf. on System Sciences (1993)

[16] Sullivan, K.J., Socha, J., Marchukov, M.: Using formal methods to reason about architectural standards, International conference on software engineering ICSE '97, (1997) 503–512

[17] Jones, C.B.: A π-calculus semantics for an object-based design notation, in: E. Best (Ed.), Proceedings of CONCUR'93, Springer-Verlag LNCS 715, (1993) 158–172

[18] Nierstrasz, O.: Infrastructure forsoftware component frameworks, Internet `http://www.iam.unibe.ch/~scg/Archive/NFS/iscf.html` (1996)

[19] Broy, M.: Towards a mathematical concept of a component and its use, Software – concepts and tools **18**, (1997) 137–148

[20] Jones, C.B.: Systematic software development using VDM, Prentice Hall (1986)

[21] Spivey, J.M.: Understanding Z: a specification language and its formal semantics, Volume 3 of Cambridge Tracts in Theoretical Computer Science. Cambridge University Press (1988)
[22] Feijs, L.M.G., Jonkers, H.B.M., Middelburg, C.A.: Notations for Software Design, FACIT Series, Springer-Verlag (1994)