A Compositional Approach to Active and Passive Components

Kung-Kiu Lau and Ioannis Ntalamagkas
School of Computer Science, The University of Manchester
Manchester M13 9PL, UK
{kung-kiu,i.ntalamagkas}@cs.manchester.ac.uk

Abstract

Current software component models lack compositionality. Most of them also do not have both active and passive components. In this paper, we show how we can define a compositional approach to active and passive components. We define these components in such a way that their composition can be defined by explicit composition operators. Our approach not only achieves compositionality, but also enables systematic or hierarchical composition.

1 Introduction

Typically a software system contains a mixture of *active* components, and *passive* components. Active components have their own thread of control and execute autonomously, whereas passive ones only execute when invoked or triggered by an external execution thread. Combining active and passive components in a system raises synchronisation issues, and therefore it is perhaps not surprising that not all software component models [8] include both kinds of components.

Current software component models fall into two main categories [8]: (i) models where components are (collections of) *objects*, as in object-oriented programming; (ii) models where components are *architectural units*, as in software architectures [4]. Exemplars of these categories are Enterprise JavaBeans (EJB) [12] and architecture description languages (ADLs) [11] respectively.

All the component models with objects as components have only passive components, except for COM+ [13], which has both active and passive components. Some ADLs, namely C2 [16], Darwin [10], and Wright [3], have only active components. The only ADL with both active and passive components is PECOS [14].

All current component models lack compositionality, in the sense of algebraic composition with a proper composition theory. This means that current component models do not offer explicit composition operators that compose components (as operands) into new components. In other words, compositionality means that composing two components C_1 and C_2 by a composition operator results in another component C_3 of the same type as C_1 and C_2.

In models with objects as components, objects are assembled by method calls. This is not algebraic composition since an object O_1 calling a method in another object O_2 does not result in a single object, but rather two objects calling each other. In models with architectural units as components, i.e. ADLs, components are connected by connectors via their ports. This kind of connection defines composition at the level of ports; at the level of whole components, it defines an *ad hoc* kind of composition, since the result of the composition depends on which ports of the sub-components are forwarded or exported to the composite.

In this paper we show how we can take a compositional approach to active and passive components. We define these components in such a way that their composition can be achieved by pre-defined explicit composition operators. These operators not only achieve compositionality, but also enable systematic or hierarchical composition.

2 Active and Passive Components

Active components can generate control signals, whereas passive components can only receive them. This is illustrated in Figure 1. In the figure, A is active and B

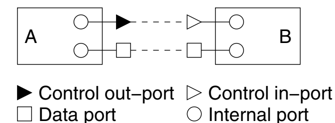

Figure 1. Active and passive components.

is passive; A can autonomously generate signals, whilst B only performs its function when it receives a control signal from A. In the figure, data ports are used for data input/ output.

Among current component models [8], only PECOS [14] and COM+ [13] have both active and passive components. PECOS explicitly distinguishes between active and passive

components, whilst COM+ does not. Therefore, in this paper, we will use PECOS as a primary point of reference.

In PECOS, components communicate using a variation of the description presented in Figure 1. Active components work autonomously and update their internal data on their own thread (hence they have internal and external data ports, see Figure 2). In contrast, passive components do not have internal data, and their data are accessed directly by the composite's thread (hence they have single data ports, see Figure 2). In PECOS, there are also event components; event components are treated as active components as far as composition and synchronisation are concerned.

PECOS is used to model field devices, which are reactive systems. Figure 2 shows a PECOS device that displays time in analogue and digital format. In this device, EventLoop

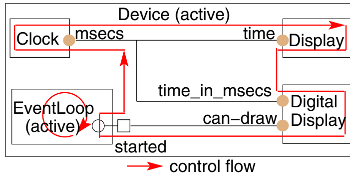

Figure 2. An example of a PECOS device.

is an active component that handles graphical events such as mouse clicks. Display and DigitalDisplay are passive components that display the time (which is read from the Clock component) in analogue and in digital format respectively. DigitalDisplay can only start drawing if the EventLoop component has started, hence it is connected to the `started` port of the EventLoop.

3 Our Approach

We define active and passive components differently from PECOS. We want to be able to use explicit composition operators instead of port connections for composing components. To this end, we define active and passive components as shown in Figure 3. In this presentation we use

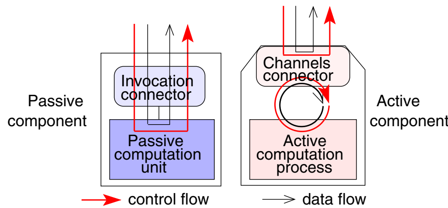

Figure 3. Atomic components.

different shapes for visually differentiating between active and passive components. In the figure, both components are *atomic* components, as opposed to *composite* components, which we will discuss in Section 5.

A passive component consists of a *passive computation unit* and an *invocation connector*. The passive computation unit contains a set of methods or operations, and performs them when invoked by the invocation connector. The invocation connector provides the interface of the component and is activated by control signals from outside the component, i.e. from other components (via composition operators, see later). It is the external control that accesses the computation unit (through the invocation connector) that forces the computation to take place. Without this external control, the passive component does nothing. As can be seen in the figure, data flow follows control flow.

An active component consists of an *active computation process* and a *channels connector*. The active computation process performs computation on its own thread of control continuously, and periodically interacts with the channels connector to pass control signals and perform data i/o. The channels connector provides the interface of the component and can communicate with other components (via composition operators). Thus an active component has an internal control thread, i.e. control inside the active computation process, and can receive an external control thread. The external thread represents the client of the component that wishes to interact with the active computation process. The two control threads run independently and interact only inside the channels connector where data communication also takes place.

We define the active computation process in an active component as a non-empty set of Communicating Sequential Processes (CSP) executing in parallel. CSP [6] is a process algebra used to model concurrent systems. A system modelled using CSP consists of a set of sequential processes that execute in parallel, and communicate via synchronous shared channels.

Example. An example of an active component is a cruise controller component (Figure 4), which is based on the cruise controller system presented in [9]. The sys-

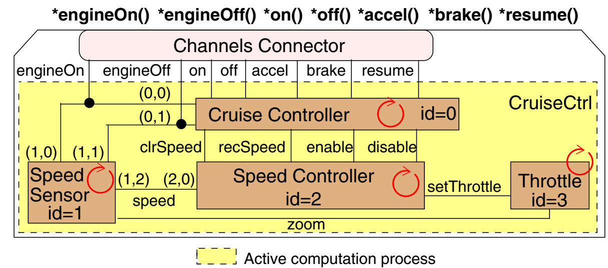

Figure 4. Cruise controller component.

tem in [9] is defined as a set of Finite State Processes (FSPs). The purpose of the cruise controller is to maintain the car speed at a desired level. Its active computation process (*CruiseCtrl*) consists of four (interacting) CSP processes: the *CruiseController*, the *SpeedSensor*, the *SpeedController* and the *Throttle*.

The user of this active component is the car driver: the driver decides when to turn the engine on (*engineOn*) or off (*engineOff*), accelerate (*accel*) or brake (*brake*). The driver

can also control the cruise controller through three buttons: *on*, *off* and *resume*. Process ids and channel numbers are discussed in the next section.

The channels connector provides a suitable interface for the component. It consists of the channels (*engineOn() for *engineOn*, *engineOff() for *engineOff*, etc.) connected to the active computation process. The asterisks (*) before the interface names denote that this component is active. This is further discussed in Section 5, where we compose active with passive components to form the whole system.

Users of the active component may interact with the active computation process by calling any of the interface methods, i.e. channels. We will see in Section 4 how exactly the cruise controller component is defined and implemented.

4 Implementing Active Components

We have implemented active and passive components in Java and we have introduced a Java API for this purpose. The implementation of passive components has been explained elsewhere [7]. In this section we explain how we have implemented active components.

First we need to explain how any component, passive or active, can be used via its interface. A passive component provides a set of methods that can be invoked through (its interface) the invocation connector. Each method is defined by a name, the types of its input parameters and the types of its output parameters: ⟨`name, input parameter types, output parameter types`⟩. Each time the component is used, one method is invoked and the results are returned. Thus the input data to the invocation connector are ⟨`(method) name, input parameters, placeholders for output parameters`⟩. This is achieved by providing the invocation connector with an `execute` method with the signature: `void execute(String, Vector<Object>, Vector<Object>)`.

An active component, on the other hand, provides a set of channels connected to (a set of) CSP processes. The user can input data values to the channels via the channels connector, and get back the results of executing the processes with these input values, also via the channels connector. Similar to the methods of passive components, each channel is defined by a name, the types of its input parameters and the types of its output parameters: ⟨`name, input parameter types, output parameter types`⟩. Thus the input data to the channels connector contain the same data as that to an invocation connector, i.e. ⟨`(channel) name, input parameters, placeholders for output parameters`⟩. Not surprisingly, the channels connector is also provided with an `execute` method with identical signature to that of the invocation connector.

The main implementation issue for active components is how to define and implement the interaction between the channels connector and the active computation process. Our approach is shown in Figure 5. We define this interaction

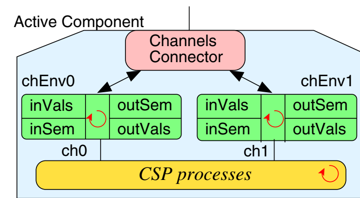

Figure 5. Structure of an active component.

by special active agents that we call *channel environments*. These environments (`chEnv0` and `chEnv1` in Figure 5) are predetermined interaction points between the channels connector and the CSP processes. We define one channel environment for each channel connected to the CSP processes.

The channels connector is passive; it waits for external control threads and it then forwards the control signal along with any data to the chosen channel environment. CSP processes are active, and communicate via channels, and every communication on a channel is considered to be an event. Therefore, in order to communicate with a CSP channel, a channel environment must be able to generate and accept events. This implies that, channel environments need to combine both event-based and data-based synchronisation mechanisms, i.e. *channels* and *semaphores*, in order to communicate with the CSP processes and the channels connector respectively.

We implement the channels connector in Java, and CSP processes in JCSP [1] (a Java implementation of CSP). For a channel environment, the channels part is implemented in JCSP, whereas the semaphores part is implemented in Java, using standard JDK semaphores. Our implementation allows the user to choose any channel from the CSP processes to interact with (through channel environments).

As already discussed, a channels connector takes as input a triple ⟨`name, input parameters, placeholders for output parameters`⟩via its `execute` method. This is the method that performs the run-time functionalities of the channels connector and is outlined below:

```
public void execute(String name,
 Vector<Object>inParams,Vector<Object>outParams){
  ChannelEnvironment env=
           ((ChannelEnvironment) envs.get(name));
  //Set input values
  env.setInputParams(inParams);
  //Signal the ch.env.
  env.inSem.release();
  // Wait for signal that output is ready
  env.outSem.acquireUninterruptibly();
  //get output values
  outParams=env.getOutParams();        }
```

When this `execute` method is called, the channels connector interacts with the chosen channel environment. The

incoming thread of the channels connector is used to set the input values `inVals` (see Figure 5) of the channel environment to the input parameters of the call to execute. The channel environment is then signalled using semaphore `inSem` to communicate with the chosen (connected) channel. The `execute` method then waits on semaphore `outSem` for the signal from the channel environment that indicates that channel communication has finished. Output values `outVals` are then returned to the caller.

Channel environments are active and they operate inside a non-terminating loop, implemented as the method `run`:

```
public void run(){
 while(true){
  //Wait for input value(s) before communicating
  inSem.acquireUninterruptibly();
  //Communicate on the given channel and set
  //output values
  communicate();
  //Signal that communication has finished
  outSem.release();      }}
```

Their runtime behaviour is complementary to the `execute` method of the channels connector. They actively wait to be signalled by the channels connector that the input values have been set. Then these values are communicated to the connected CSP channel, and finally they signal the channels connector that communication has finished. The actual implementation of the `communicate` method depends on the type of the connected channel, for example an integer channel, a channel with no input/output values, and so on.

Our implementation ensures that CSP semantics are preserved and this is reflected in the behaviour of the active component. The `communicate` call in the `run` method is not guaranteed to return immediately. For example, it is possible that the caller gets blocked until some values are input in another channel.

We have implemented a tool for generating active components from their definition, as illustrated in the following example.

Example. The implementation of cruise controller in Figure 4 is outlined as follows. First, we specify the active computation process *CruiseCtrl* in CSP as:

$CruiseController \| SpeedSensor \| SpeedController \| Throttle$, where $\|$ is the parallel composition operator in CSP. Then we specify the sub-processes. For example, part of the specification of the *Cruise Controller* process is:

$$\begin{array}{rcl} CruiseController & = & engineOn \rightarrow clrSpeed \rightarrow Active \\ & & \Box\ off \rightarrow CruiseController \\ & & \Box\ brake \rightarrow CruiseController \\ & & \Box\ accel \rightarrow CruiseController \\ Active & = & \ldots \end{array}$$

The symbol $\Box$ stands for the external choice operator: the environment of the process decides which event occurs next. This specifies that the cruise controller in its initial state can accept any of the events *engineOn*, *off*, *brake*, *accel*. After an *engineOn* event is accepted, it clears the speed of the speed controller, and then it becomes active. The other three events simply return the cruise controller to its initial state. These events describe no i/o behaviour: they are simple synchronisation events.

Secondly, the CSP processes are implemented manually in Java using JCSP. This is because at present there is no tool for automatically translating machine-readable CSP into Java [15]. For example, the Java class implementing the *CruiseController* process is declared as:

```
package cruiseControlller;
public class CruiseController implements
                     jcsp.lang.CSProcess {...}
```

Thirdly, inter-process communication is defined. The above Java implementation only defines single processes and how they behave in isolation; it does not (and cannot) define how processes interact with each other. The latter has to be defined according to the CSP specification defined in the first step. This information (that also includes which channels are to be exposed to the interface of the composite), is passed in the form of a script to our tool that generates the active component. For the automobile cruise controller, part of the script is shown below and an explanation follows (lines beginning with `--` are comment lines):

```
--Declare the processes, first process gets id 0
cruiseController.CruiseController
--then process with id 1
cruiseController.SpeedSensor
...
--Define the connections
--Enter channels, then connection type
--connection for channel engineOn
(0,0)~(1,0) ALTING_BARRIER_HUB_IN
--connection for channel speed
(1,2)~(2,0) ALTING_BARRIER_HUB
...
```

After declaring the processes that constitute the CSP component, and as part of the implementation directives imposed by JCSP, the developer specifies how individual process channels connect with each other. These channel connections reflect the ones shown in Figure 4. For example channel *(0,0)* connects with channel *(1,0)*. This means connect from the process with id 0 (CruiseController) the first declared channel inside its class definition, to process with id 1 (SpeedSensor) to channel with id 0. Process and channel ids are given automatically by our tool. Immediately after the connecting channels are defined, the channel types are specified. For example, `ALTING_BARRIER_HUB_IN` defines that a JCSP `AltingBarrier` will be used,[1] and that this channel should be exposed by the component. The appropriate channel environment is then created for this channel automatically. In contrast, channels connected

[1] `AltingBarrier` is the JCSP class representing a channel with no i/o behaviour, where the participating processes may alternate.

through ALTING_BARRIER_HUBs remain hidden and for example channel *speed* – connection of *(1,2)* with *(2,0)*, is not exposed and a channel environment is not created.

5 Composite Components

In PECOS, active components and passive components are composed to form composites. The composition is done by connecting the components' data ports and defining a schedule for execution and synchronisation. The schedule defines in what order the passive sub-components execute. For active sub-components, the schedule defines when the data values in their internal data ports will be copied into their respective external data ports, or vice versa. The schedule is *ad hoc*, and only defines sequential control, i.e. one passive component executes or one data synchronisation occurs at a time. For example, the Device in Figure 2 is a composite. Its structure is as follows:

```
active component Device{
  Clock clock;
  Display display;
  DigitalDisplay digitalDisplay;
  EventLoop eventLoop;
  connector time(clock.msecs,display.time,
      digitalDisplay.time_in_msecs);
  connector eventLoop_started(eventLoop.started,
      digitalDisplay.can_draw);     }
```

Its schedule is shown below:

```
schedule sched of Device every 1000 at 10{
 { sync eventLoop;
   exec clock;
   exec display;
   exec digitalDisplay;
 } at 0;                    }
```

The schedule defines a period of execution of 1000 milliseconds (`every 1000`), that executes at the specified priority (`at 10`). Inside the schedule, a set of jobs are defined that execute at the beginning of the period of the Device (`at 0`). For active sub-components, `sync` means synchronise the data of its internal data ports with the data of the external data ports, and at runtime the `synchronise()` of the component is called. For passive components, `exec` means execute the component, and at runtime its `execute()` method invoked. In Figure 2 we have drawn the control flow as defined by the above schedule.

A disadvantage of this approach is that the synchronisation mechanism used for transferring data between active and passive components is error prone. For example, the developers of active components must explicitly place locks inside their `synchronise()` and `execute()` methods. Another disadvantage of PECOS composition is that the scheduling is strictly sequential, while for instance the two display components should be scheduled to execute in parallel. In this section, we show how we can use explicitly defined composition operators to compose active and passive components as these were defined in Section 3.

5.1 Passive and Active Composites

The atomic components defined in Figure 3 can be composed via their interfaces by using explicitly defined composition operators. Depending on the nature of these operators, composition can yield either a passive composite or an active composite (Figure 6).

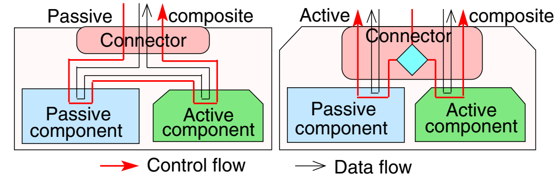

Figure 6. Passive and active composites.

In [7] we defined explicit composition operators as connectors that coordinate control (and data) flow between the sub-components. A composition connector encapsulates *control*. It is used to define and coordinate the control for a set of components. For example, for sequencing we use the *pipe* and *sequencer* connectors, and for branching, we use the *selector* connector. A *pipe* connector that composes components $C_1, \ldots, C_n$ can call methods in $C_1, \ldots, C_n$ in that order, and pass the results of calls to methods in C_i to those in C_k, $k > i$. A *sequencer* connector is the same as a pipe but does not pass the results of C_i to C_k. Whenever composition occurs, the result is a composite component with an interface based on the interfaces of the connected components and on the composition operator used.

Using the composition connectors that we previously defined, we get a passive composite whenever we compose a passive component and an active one. For example, the passive composite in Figure 6 is composed by a sequencer connector. This composite is passive because it behaves like the passive sub-component, i.e. it can take and respond to a single method invocation at a time and it needs an external control thread in order to execute. The active sub-component on the other hand executes on its own control thread, and only needs sufficient input data in order to execute continuously. The role of the external control thread is to provide these data to the component. As a result, each call to the channels connector may provide more than a single channel name to execute, for example a sequence of channel names, along with the appropriate input parameters. These data are then consumed by the active computation process inside its own control loop.

However, in the passive composite in Figure 6, control reaches the active sub-component via the passive sub-component, and the composition operator treats similarly both sub-components. Therefore, there can be exactly one input to the active sub-component for each call to the passive sub-component. To achieve an active composite component, we need to use a different kind of composition connector, namely a *stateful facade*, as we now explain.

In object-oriented programming, a facade for a set of objects is a unified interface to all the methods of the objects, and the main method of the facade invokes methods in the objects. In a stateful facade [5], the behaviour of the main method depends on the current state of the facade.

By using a composition connector defined as a stateful facade, we can produce an active composite (from active and passive sub-components), since such a facade provides access to the methods of all the sub-components. The state of the facade determines which sub-component's methods can be called. Thus if a sub-component is active, then the composite will also be active, i.e. it can service an endless sequence of inputs and the role of the external thread is to provide these inputs to the component. An example of a generic active composite is shown in Figure 7, that

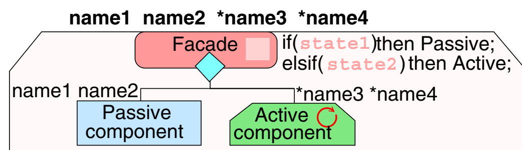

Figure 7. A generic active composite.

consists of an active and a passive sub-component. The facade connector contains its own data that define its state. Based on this state, the facade defines a conditional statement that declares the conditions that when true will enable each sub-component to execute. For example, if a call to `name1` is made while the facade is in state2, state2≠state1, the call does not succeed and an error message is returned to the caller.[2] In the figure, the active sub-component may accept any sequence of (channel) names. This property is preserved when composing via a stateful facade, and therefore a single call to the composite may provide any sequence (of finite length) of the names `name3`, `name4`, but only a single name when invoking one of `name1` or `name2`.

Finally, an active composite with a stateful facade as its interface can be further composed, with passive or active components. The result of each such composition is another composite with a stateful facade as its interface. In contrast, using the composition operators we defined previously, in each such composition, an active composite is treated like a passive component, i.e. a single name of its interface is called.

5.2 Implementing Active Composites

In [7] we have presented the implementation of passive composite components. In this section we present how active composites are implemented by using as example an authenticated cruise controller.

[2]Clearly, facade's state affects the way the composite component is used. We are currently investigating ways of exposing this information in the composite's interface.

An authenticated cruise controller is a composite component consisting of an authentication component and the cruise controller system, as shown in Figure 8. The au-

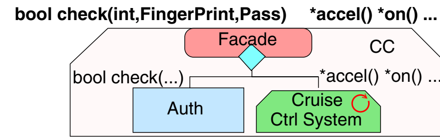

Figure 8. Authenticated cruise controller.

thentication component is a passive component that authenticates a user, based on the user id, a fingerprint, and a password. The active component is the cruise controller component shown in Figure 4. The requirements for the system describe that before users can access the cruise controller, they need to be authenticated first. After authentication, users may access the cruise controller directly, until the engine is turned off, at which point they need to authenticate again prior to using the cruise controller. We use a stateful facade for composing the two components into composite CC, where the state of the facade reflects whether a user has been authenticated or not.

Constructing the active composite comprises of several steps. Firstly, the system developer must define the state (data) of the facade. We define these data as a set of triples of the form $\langle name, type, value \rangle$. For the authenticated cruise controller of Figure 8, the only data needed is a single boolean variable: $\langle hasAuthenticated, Boolean, Boolean.FALSE \rangle$. We have implemented component data in Java as a class named *Data* that provides read/ write access to individual data elements.

Secondly, the system developer must define the conditions that when true will enable each subcomponent to execute. Each subcomponent gets therefore associated with a boolean expression: if the expression is true, then the component executes. For the authenticated car controller, the boolean expressions associated with each component are written in reverse polish notation:

```
Auth: hasAuthenticated FALSE EQ
CruiseCtrl: hasAuthenticated TRUE EQ
```

Thirdly, the system developer defines how the composite's state gets updated *after* a call to a subcomponent returns. This is because the facade's data reflect the facade's definition that the result of an invocation to a subcomponent affects the next invocation. In order to update its own state, every facade connector contains an Updater object that has an `update` method. That method is called when the execution of one of the sub-components terminates, and it may change the data variables of the composite component. The system developer must provide an Updater object to the facade's constructor. For the authenticated cruise controller the overridden method will be:

```
public void update(String name,Vector<Object>
```

```
            inParams, Vector<Object> outParams){
 if(name.equals("check")) {
  //method "check" has a single output parameter
  Boolean x = (Boolean) outputParams.get(0);
  data.write("hasAuthenticated", x);
 } else if(name.contains("engineOff")) {
  data.write("hasAuthenticated", Boolean.FALSE);
 }      }
```

Below we outline the code for the composite's `execute` method, i.e. the method that executes when we call the composite component:

```
public void execute(String name, Vector<Object>
            inParams, Vector<Object> outParams)
                                  throws Exception{
 checkIfValid(name, inParams);
 //Invoke the subcomponent
 invoke(name, inParams, outParams);
 //Update the composite's state
 this.updater.update(names,inParams,outParams); }
```

The input `name` parameter is a string, and therefore may consist of a sequence of space separated names. It is initially necessary to check that the input `name` and parameters are valid. A validity check defines that for a sequence of names, all names must belong to the same component. More importantly, it is checked whether the state of the facade allows the execution of the component to which these names belong. The boolean expressions specified in the second step are utilised. If the invocation is possible, then the subcomponent gets invoked, and the state of the facade gets updated according to the update method presented in the third step above. If not, an exception is thrown.

Finally, the composite's interface as shown in Figures 7 and 8, is generated automatically during the composite's construction, and it consists of the union of the (channel or method) names appearing in the connected subcomponents. If the same name appears in both components, then it is ignored. If more than one subcomponents are active, then two stars will be placed before each name belonging to the second active component (three to the third, and so on), in order to able to distinguish at the interface level which names correspond to which active component.

We have implemented and run the authenticated cruise controller system using our tool, as shown in Figure 9. The figure shows the composite `CC`, and its interface, that was created via composition of `Auth` with `CruiseCtrl`. We have inserted printout statements in the the facade connector and in the two subcomponents, in order to produce the sample execution trace in the figure. The trace shows that it is not possible to execute the `CruiseCtrl` unless the user has authenticated. The composite then executes a sequence of (channel) names within a single call, because the active subcomponent allows this. Because the user has authenticated, it is not possible to authenticate again. Finally, in the next call, the external user will turn the cruise controller and then the engine off.

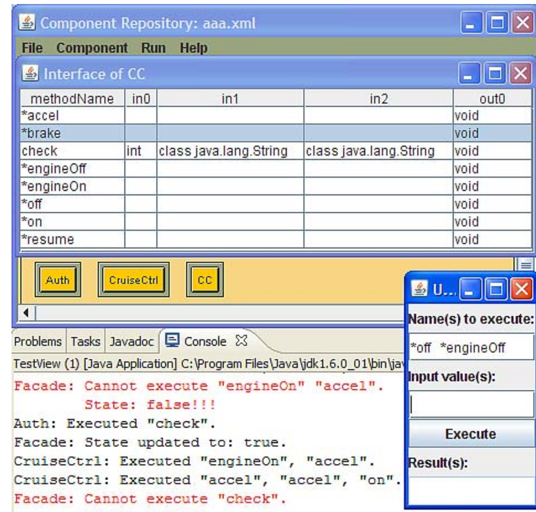

Figure 9. Executing the cruise controller.

It is worth noting that using our previously defined composition operators instead of the stateful facade would produce not only a passive composite, but also a wrong one. For instance, the composite could be composed using a pipe and a guard (Figure 10). The result of the authentication is passed into the guard connector. The guard is an adaptation connector, that allows control to pass only if an internal condition is satisfied. Only if the authentication result is true

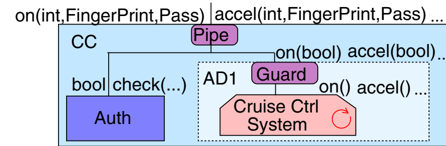

Figure 10. Authenticated cruise controller.

may the user access the cruise controller. The obvious disadvantage of this solution, is that the user needs to authenticate every time before accessing the cruise controller. This is not the right behaviour that we want.

6 Discussion

Our definitions of passive and active components are different from those in PECOS and in ADLs, e.g. Wright, which also use CSP to define active components (Figure 11). The main difference is that our component, be it active or passive, has an interface for composing whole components, whereas the ports in a PECOS or Wright component only provide connection points for linking to other ports. Our operators compose whole components, and not just ports. Composition via stateful facade is different from port forwarding, because the stateful facade adds behaviour to the composite component (in terms of its state), whereas port forwarding does not. Therefore, our approach offers

	PECOS	Wright	Our approach
passive atomic component	computation + ports	----	computation + interface
active atomic component	schedule + computation + ports	CSP process + ports	CSP process + interface
composite component	sub-components + schedule + connections [+ forwarded ports] (active)	sub-components + connectors + connections [+ forwarded ports] (active)	sub-components + composition operator (interface) (active or passive)

Figure 11. Related definitions.

the advantage of a proper composition theory, which supports hierarchical composition. Every composition produces a composite that can be further composed with other components, and so on. In other words, our approach has *compositionality*, unlike current component models.

Additionally, when compared to PECOS, our active component has a more expressive language, i.e. CSP, for defining computation. Simple and elegant CSP designs inside our active components can become overly complicated in PECOS. Additionally, CSP allows us to model check the CSP processes for certain temporal properties like deadlock and livelock freedom using automated tools, e.g. FDR [2]. This is not possible in PECOS.

More importantly, in a composite component in PECOS, active components are treated as passive, i.e. whenever the control of the composite reaches an active subcomponent, a single data communication is performed per port, whereas it should be possible to provide multiple data values to the port. In contrast, our facade composition operator allows an active subcomponent to retain its active nature, by letting users of the composite perform multiple data communications within a single call to it.

In Wright, an ADL with active components and explicit active connectors, composition is similar to PECOS, i.e. it is *ad hoc*, and composites are defined through explicit port forwarding; on the other hand, because Wright uses CSP for defining the behaviour of components and connectors, no schedule is necessary. However, passive components are not defined in Wright. This introduces unnecessary complexities especially when designing systems where passive components are the natural choice. For example, server-side systems, which many component models, e.g. EJB [12], focus on, are usually modelled and implemented using passive components.

7 Conclusion

In this paper we have proposed a compositional, hierarchical approach for building concurrent systems. We have introduced (atomic) active components as part of a system and explained how they can be composed with other passive or active components to form composite components. Composite components can be either active or passive depending on the composition operator used. We have provided the implementation of both kinds of active components, as well as a suitable composition operator. Our approach contrasts with the approaches followed by other component models, which do not usually model both active and passive components. In addition, composition in other models is *ad hoc* and takes the form of port connections and port forwarding, in contrast to our compositional approach.

We are currently working on adding concurrency support to all the connectors of our model by allowing multiple users to access them concurrently. This will introduce the need for additional synchronisation mechanisms, and may have implications on the way composition is performed.

References

[1] Communicating Sequential Processes for Java (JCSP) Home Page. Web. `http://www.cs.kent.ac.uk/projects/ofa/jcsp/`.

[2] FDR2 Home Page. Web. `http://www.fsel.com/software.html/`.

[3] R. J. Allen. *A Formal Approach to Software Architecture*. PhD thesis, School of Computer Science, Carnegie Mellon, May 1997.

[4] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley, second edition, 2003.

[5] W. Crawford and J. Kaplan. *J2EE Design Patterns*. O'Reilly Media, Sep 2003.

[6] C. Hoare. *Communicating Sequential Processes*. Prentice Hall, April 1985.

[7] K.-K. Lau, P. Velasco Elizondo, and Z. Wang. Exogenous connectors for software components. In *Proc. CBSE, LNCS 3489*, pages 90–106. Springer-Verlag, 2005.

[8] K.-K. Lau and Z. Wang. Software component models. *IEEE Trans. Soft. Eng.*, 33(10):709–724, 2007.

[9] J. Magee and J. Kramer. *Concurrency: State Models & Java Programs*. John Wiley & Sons, March 1999.

[10] J. Magee, J. Kramer, and D. Giannakopoulou. Behaviour analysis of software architectures. In *WICSA*, volume 140 of *IFIP Conference Proceedings*, pages 35–50. Kluwer, 1999.

[11] N. Medvidovic and R. N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Trans. Soft. Eng.*, 26(1):70–93, 2000.

[12] R. Monson-Haefel. *Enterprise JavaBeans 3.0*. O'Reilly & Associates, fifth edition, 2006.

[13] T. Pattison. *Programming Distributed Applications with COM+ and Microsoft Visual Basic 6.0*. Microsoft Press, 2000.

[14] Pecos: Pervasive component systems. `http://www.iam.unibe.ch/~pecos/`.

[15] V. Raju, L. Rong, and G. S. Stiles. Automatic Conversion of CSP to CTJ, JCSP, and CCSP. In *Communicating Process Architectures 2003*, pages 63–81, 2003.

[16] R. N. Taylor, *et al*. A component- and message-based architectural style for GUI software. *IEEE Trans. Soft. Eng.*, 22(6):390–406, 1996.