SPECIFYING REUSABLE CONTROLLERS FOR SOFTWARE COMPONENTS

José M. Troya
and Antonio Vallecillo

Dpto. Lenguajes y Ciencias de la Computación
Universidad de Málaga. Campus de Teatinos. 29071 Málaga. Spain
{troya,av}@lcc.uma.es

Abstract: The design of components for open and distributed systems is challenging the software community with its specific problems. One of the current approaches is based on a reflective model that uses standard, independent, composible meta-components (*controllers*) to coordinate components and modify their behavior according to the user requirements. However, this approach still has some pending issues, like the definition of design methodologies that lead to reusable and composible components and controllers, and the use of formal tools to reason about the correctness of the composed applications. This paper presents a formal framework in Object-Z for specifying reusable controllers, based on a component model for this kind of systems. The basic mechanisms of the model are formalized, together with the concepts and methods that allow developers of the controllers to prove their correctness, specify their behavior, and characterize the effect of adding them to components.

INTRODUCTION

Current architectural approaches for developing software applications rely on components and connectors. Components encapsulate computation, while connectors describe how components are integrated into the architecture. This separation of concerns has clear advantages for system design, verification and reuse, and provides a compositional methodology for specifying the architecture of applications, specially important for open and distributed systems (ODS).

However, this approach also presents some limitations, since connectors are good for defining and managing the interconnections between components, but they still lack the

ability to abstract other important *properties*, like resource discovery and management, placement policies, reliability features, and other context-specific requirements [4].

To address these concerns, system developers are working on better and more sophisticated systems that incorporate new features, while component developers have to undertake those requirements not covered by the systems themselves. However, this is definitely not a good approach in the long term, since it leads to unnecessarily complicated systems and components, hindering their reusability, portability and openness. A more preferable approach would be to design generalized components which may be customized to particular architectural contexts. Connectors would encapsulate these customizations, keeping both components and systems as simple as possible, and free of orthogonal and context-specific concerns and requirements.

Different authors follow a reflective approach [2, 5, 6, 10] that considers components as black boxes that transparently modify their behavior through *controllers* —layers, meta-objects or wrappers—, first-class entities that wrap them. Our proposal is also based on this model, offering a three-layered structure: "Systems-Controllers-Components". Systems can be simplified to the minimum, offering just the infrastructure for the creation and communication of components. Components encapsulate computations, and the standard add-on controllers provide components with the required behavior. In our software market there is not only room for systems and components manufacturers, but also for developers of reusable controllers. The idea is to ease the task of building applications by using off-the-shelf components and controllers.

In order to do so, some goals must be achieved. First, components and controllers should be defined in such way that controllers can be added to components in a compatible, modular and independent manner, and composed to apply multiple properties simultaneously to a component. And second, formal methods and models are needed for specifying the behavior of the components, the controllers and the aggregates, for reasoning about them, and for proving that the application requirements can be met when putting all the pieces together.

Our work tries to achieve those goals. We have defined a reflective model for open systems that includes the concepts of components and controllers, allows their modular composition and aggregation to build up applications, and also targets other ODS-related issues, like component evolution, environment-awareness, and dynamic configuration [17]. It has two different parts: a communication model based on asynchronous messages with local broadcasting capabilities, and a reflective architecture on top of it to wrap components with controllers that modify their behavior according to the user requirements. Controllers are not just mere computational filters, since they not only capture and modify messages, but they can also split, reorder or join them, reply to messages, or even interrogate the system and reconfigure themselves accordingly.

However, providing component and application developers with (yet) another compositional model that aims for a global component marketplace is not enough, unless it is supported by a *methodology* that guide and simplify their work and a *formal framework* to reason about the component and the application properties.

In this paper we present the formal framework that supports the model, showing how the model mechanisms and concepts can be formally specified, and the sort of

formal results that can be obtained. In particular, we study (*a*) the specification of the component interfaces, (*b*) the characterization of the way in which controllers modify the interface of components, and (*c*) the degree of replaceability of the modified components with regard to the original ones.

The paper is structured as follows. Next section briefly describes the component model used, and the following two sections define the concepts that allow the reasoning about equivalence, compatibility and replaceability of components. After that the sort of outcomes that our formal framework produces are shown, from both the theoretical and practical points of view. In particular, we will show a method for producing specifications of controllers and how to derive from them the desired formal results. Finally, we relate our work to the contributions of other authors and draw some conclusions.

THE COMPONENT MODEL

This section begins with a brief explanation of the model, and then shows how its concepts and mechanisms are specified in our formal framework. Instead of choosing any of the existing component models, we have defined a neutral one with the minimum set of features required for components to interoperate in ODS. The model is devised to serve as a common component platform that abstracts all these features and implements them in a natural way. None of the most commonly used component models fulfills all ODS requirements, and using a simple and neutral model greatly simplifies the reasoning processes about its components, and eases its implementation in any other model. In particular, our aim is to be able to transfer our theoretical results to existing commercial component platforms like CORBA, DCOM or JavaBeans.

Generally speaking, any computational entity can be modeled as an object (even if internally implemented by many), with a state (given by its attributes) and some access operations (its methods). We define *component* as an object encapsulated with an interface compatible with the communication mechanisms offered by the system. The capsule abstracts its properties, hides its implementation and allows it to interact with other components.

In our model, components interoperate using mailboxes and asynchronous messages. Each component has a mailbox with a unique global identifier, through which the component sends messages to other mailboxes and receive messages from other components. Messages are information entities with a header and a body. The header is a set of fields with the delivery information. The body is just another field with no predefined structure, used to store the data being delivered.

An important field of every message is its *selector*, that determines the operation to be executed by the target component. For every method f implemented by a component, we define four different selectors: $!f$, $?f$, Re:$!f$ and Re:$?f$. The first one invokes the method, and Re:$!f$ is used for replying to it. Selector $?f$ asks the destination component whether it implements method f or not, and Re:$?f$ answers this question. Besides, the special selector '??' requests a component for the list of its methods.

Components being black boxes, their behavior is defined by their interfaces. We define the *interface* of a component as the set of message selectors that it sends out (outputs) plus the received ones that it understands and treats (inputs), supposing that

received messages not understood by a component are discarded. From these sets the concepts of (syntactic) compatibility and replaceability of components will be defined.

On the other hand, mailbox identifiers have two parts: a local name and a domain address. In our context, a *domain* is a set of interconnected machines, and defines the 'environment' of a component. The sender of a message can specify just a destination name, meaning that the mailbox belongs to its local domain, or a whole mailbox identifier. But it can also specify a special name (*BCST*) so that the message gets sent to all mailboxes currently in the destination domain.

Our model tries to minimize both systems and components requirements, dealing with every context-specific requirement in a modular and independent way through the use of *controllers*. Controllers are first-class entities that can be attached to mailboxes, capturing their incoming and outgoing messages and modifying them according to their purpose. Multiple controllers can be attached to the mailbox of a component, getting chained in such way that outgoing messages from a controller become incoming messages to its successor.

Each controller implements a *property* that deals with an ODS specific requirement, like dynamic re-configuration, error detection and recovery, maintainability or adaptability. We have initially identified a set of properties that we think of particular interest for ODS, and that can be achieved through the use of controllers (many behavioral properties can be implemented that way, although not all, as can be learned from Aspect-Oriented Programming [11]). We will not go in detail about them, just mention three that provide components with autonomy in an open environment:

Independence Components should be self-governed, able to discover the services they need and free to decide the provider to use. A controller implementing this property maintains a list of the services used by its component, dynamically updated with the information from the received messages. When delivering a message, the controller checks whether the target component is working or not, sending always the message to a known active component. The controller is also able to interrogate its environment for valid service providers, therefore making the component 'environment-aware'.

Self-Protection Components should protect themselves against external failures and avoid never-ending waits. Controllers implementing this property use a time-out table for outgoing messages, together with the component instructions for handling timeout conditions. They produce replies to the component when the target component of a method does not reply within a given deadline.

Adaptability Components should be extensible and able to accommodate to different interfaces and protocols. Regarding extensibility, controllers of this property try to find available service providers and re-divert to them the incoming service requests not implemented by the component. Regarding interoperability and interface adaptation, they try to find *translators* for the outgoing messages they handle, entities similar to Wiederhold's mediators [18] or to Yellin and Storm's adaptors [20]. Mediation is a smart way to achieve evolutionary adaptation of interfaces, since it separates the controllers from the adapters themselves,

improving the reusability of the controllers and the independent evolution of the adapters.

The remainder of this section is devoted to the specification of the basic mechanisms of the model. Object-Z [8], an object-oriented variant of Z [15], has been chosen as formal notation. Z is extremely powerful for dealing with sets and functions, the basic terms in which our model can be formally expressed, and Z also allows the incremental development of specifications and the refinement and derivation of code, very important for easing the implementation of the specified systems. On top of that, Object-Z provides a more structured specification design, the possibility of associating operations with states, and the use of inheritance for specializing controllers. We will also make use of some temporal logic operators like *always* ($\Box$), *eventually* ($\Diamond$) or *previously* ($\ominus$) in addition to Z's first order logic. These operators are commonly used in Object-Z for specifying class history invariants, and their semantic description can be found in [13]. Using temporal logic has greatly simplified our definitions and alleviated many of the formal proofs.

Basic Types

The following definitions specify some of the basic types used in our model:

$$[\mathit{NAME}, \mathit{DOMAIN}, \mathit{SERVICE}] \qquad \mathit{DECORATOR} ::= \; ! \mid ? \mid \mathrm{Re}{:}! \mid \mathrm{Re}{:}?$$
$$\mathit{ADDR} == \mathit{NAME} \times \mathit{DOMAIN} \qquad \mathit{MsgID} == \mathit{DECORATOR} \times \mathit{SERVICE}$$

Mailbox identifiers are of type *ADDR*, which has two parts: the name of the mailbox and the domain where it 'lives' (e.g. `av@lcc.uma.es`). Service identifiers (i.e. method names) belong to type *SERVICE*, and message selectors are of type *MsgID*. Types *NAME*, *DOMAIN* and *SERVICE* are considered as basic types, since their internal structure is not relevant at this level of the specification. The only restriction is that there should be a constant of type *NAME* indicating that a message has to be sent to all mailboxes in a domain:

$$\mathit{BCST} : \mathit{NAME}$$

The following functions are used to 'project' the parts of the mailbox identifiers and message selectors:

$$\mathit{Name} == \mathit{first}[\mathit{ADDR}] \qquad \mathit{Decorator} == \mathit{first}[\mathit{MsgID}]$$
$$\mathit{Dom} == \mathit{second}[\mathit{ADDR}] \qquad \mathit{Service} == \mathit{second}[\mathit{MsgID}]$$

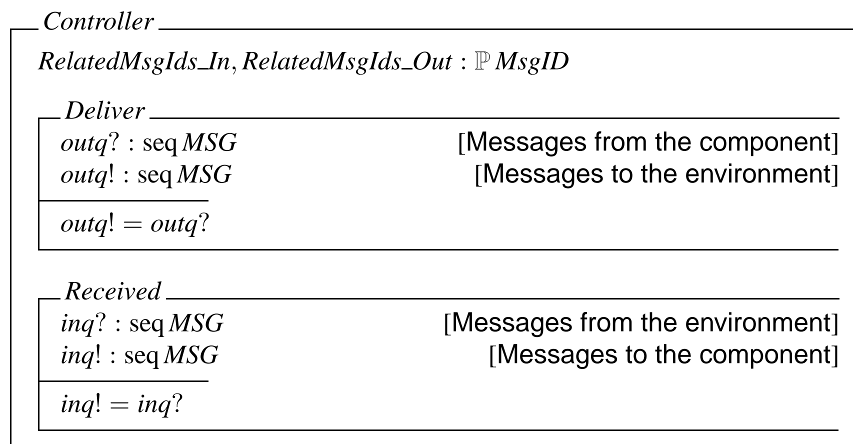
Type *MSG* describes the messages sent and received by components through mailboxes (we have only included here the fields relevant to this specification):

MSG

$\mathit{To} : \mathit{ADDR}$	[Destination mailbox]
$\mathit{From} : \mathit{ADDR}$	[Originator mailbox]
$\mathit{Selector} : \mathit{MsgID}$	[Msg Selector]
$\mathit{Info} : \mathit{STRING}$	[Msg Data]

Controllers

Controllers can be considered as active wrappers that modify the behavior of components. They all have the same structure, with two basic operations (*Received* and *Deliver*) that determine the way in which they operate with the component incoming and outgoing messages. The following class specifies a general controller. This is the most basic controller, that simply let messages go through it unmodified:



Class *Controller* has two constants (in the Object-Z style) that determine the set of (incoming and outgoing) message selectors that will be considered as relevant for a particular controller. Both values are used when configuring a controller to be plugged to a particular component, and define the component preferences. Only messages with selectors belonging to those sets will be treated by the controller, while the rest of messages will pass transparently through it. Operations *Deliver* and *Received* process outgoing and incoming messages, respectively.

All controllers will be defined from this class by inheritance, renaming its operations according to their purposes.

Mailboxes

Mailboxes have an identifier (*Addr*, of type *ADDR*), two message queues (*inq* and *outq*) to hold incoming and outgoing messages, and a list of attached controllers (*controllers*). Components send and receive messages to and from their environment using operations *Send* and *Receive*, that make also use of the operations of the controllers attached to the mailbox, forcing messages to go sequentially through them. The system transfers messages among components using their mailbox operations *ExternalPut* and *ExternalGet*, that allow the access to the message queues from the environment side (see next heading).

MAILBOX

$\upharpoonright(\text{INIT}, Addr, ExternalPut, ExternalGet, Send, Receive)$

$Addr : ADDR$ [Mailbox Address]

$inq, outq : \text{seq}\, MSG$ [Incoming and outgoing msg queues]
$controllers : \text{seq}\, \downarrow Controller$ [Controllers attached]
Δ
$nc : \mathbb{N}$

$nc = \# controllers$

INIT

$inq = outq = controllers = \langle\,\rangle$

AddController

$\Delta(controllers)$
$L? : \downarrow Controller$

$controllers' = controllers \frown \langle L? \rangle$

ExternalGet

$\Delta(outq)$
$m! : MSG$

$outq \neq \langle\,\rangle$
$m! = head\ outq$
$outq' = tail\ outq$

ExternalPut

$\Delta(inq)$
$m? : MSG$

$m?.To = Addr \vee (Name(m?.To) = BCST \wedge Dom(m?.To) = Dom(Addr))$
$inq' = inq \frown \langle m? \rangle$

BasicSend

$\Delta(outq)$
$outq? : \text{seq}\, MSG$

$outq' = outq \frown outq?$

BasicReceive

$\Delta(inq)$
$inq! : \text{seq}\, MSG$

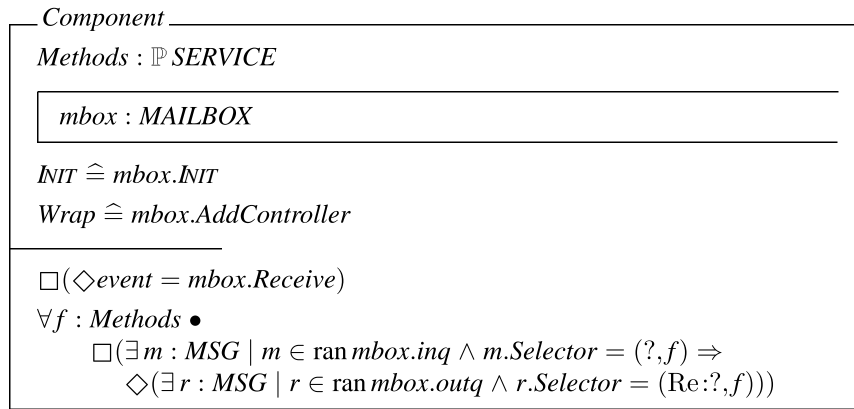
$inq \neq \langle\,\rangle$
$inq! = inq$
$inq' = \langle\,\rangle$

$$Send \mathrel{\widehat=} [\, nc > 0 \,] \wedge ((\fatsemi i = 1 \mathinner{..} nc \bullet controllers(i).Deliver) \fatsemi BasicSend) \;[\!]\; [\, nc = 0 \,] \wedge BasicSend$$

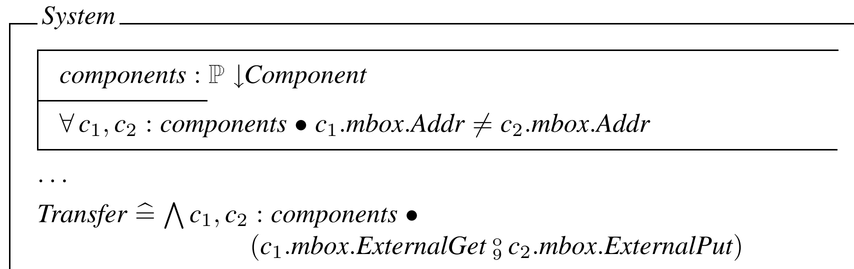
$$Receive \mathrel{\widehat=} [\, nc = 0 \,] \wedge BasicReceive \;[\!]\; [\, nc > 0 \,] \wedge (BasicReceive \fatsemi (\fatsemi i = 1 \mathinner{..} nc \bullet controllers(nc - i + 1).Received))$$

Components

In this section we will give a very simple specification of the components, just considering their relevant aspects to our model. A component has a set with the names of methods it implements, specified by a constant (*Methods*), and a mailbox to communicate with other components. Operation *Wrap* is used to attach a controller to the mailbox of a component. The two history invariants force the component to read its mailbox from time to time, and to reply messages requesting information about its methods.



Finally, communication among components is achieved at the system level, using their mailboxes:



INTERFACES

Components being black boxes, the way to describe their behavior is through their interfaces. Traditional object-oriented interfaces contain only information about the incoming messages. However, in component-based models it is important to consider outgoing messages too; without them it is not possible to check the compatibility between two components, since we do not only need to know the services the component implements, but also the services it requests to other components.

In the following definitions we will just consider the message selectors for specifying the interfaces. This provides the flexibility required to express the dynamic changes and the evolution of components in open and independently extensible systems.

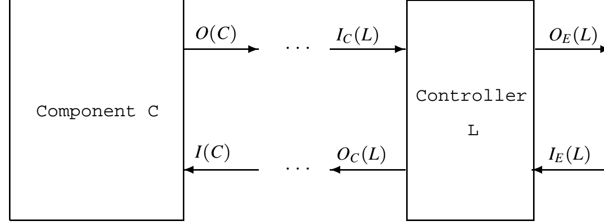


Figure 1 The interfaces of a component and a controller.

Mailbox Interface

The interface of a mailbox is determined by two set functions, *Inputs* and *Outputs*, that contain the messages that have *ever* been received or transmitted through the mailbox:

$$
\begin{array}{l}
\mathit{Inputs}, \mathit{Outputs} : \mathit{MAILBOX} \rightarrow \mathbb{P}\, \mathit{MSG} \\
\hline
\forall\, mb : \mathit{MAILBOX} \bullet \\
\quad \mathit{Inputs}(mb) = \{m : \mathit{MSG} \mid \Diamond(m \in \operatorname{ran} mb.inq)\} \\
\quad \mathit{Outputs}(mb) = \{m : \mathit{MSG} \mid \Diamond(m \in \operatorname{ran} mb.outq)\}
\end{array}
$$

With them, the interface of a mailbox *mb* is defined by a pair of functions (I and O), obtained from the previous ones by simply 'projecting' the selectors of the messages:

$$
\begin{array}{l}
I, O : \mathit{MAILBOX} \rightarrow \mathbb{P}\, \mathit{MsgID} \\
\hline
\forall\, mb : \mathit{MAILBOX} \bullet \\
\quad I(mb) = \{m : \mathit{MSG} \mid m \in \mathit{Inputs}(mb) \bullet m.\mathit{Selector}\} \\
\quad O(mb) = \{m : \mathit{MSG} \mid m \in \mathit{Outputs}(mb) \bullet m.\mathit{Selector}\}
\end{array}
$$

Controller Interfaces

Controllers offer two interfaces (see figure 1): one to the component (given by I_C and O_C) and one to the environment (given by I_E and O_E). They are defined similarly to mailbox interfaces:

$$
\begin{array}{l}
I_E, O_E, I_C, O_C : \downarrow \mathit{Controller} \rightarrow \mathbb{P}\, \mathit{MsgID} \\
\hline
\forall\, L : \downarrow \mathit{Controller} \bullet \\
\quad I_E(L) = \{m : \mathit{MSG} \mid \Diamond(m \in \operatorname{ran} L.\mathit{Received}.inq?) \bullet m.\mathit{Selector}\} \\
\quad O_E(L) = \{m : \mathit{MSG} \mid \Diamond(m \in \operatorname{ran} L.\mathit{Deliver}.outq!) \bullet m.\mathit{Selector}\} \\
\quad I_C(L) = \{m : \mathit{MSG} \mid \Diamond(m \in \operatorname{ran} L.\mathit{Deliver}.outq?) \bullet m.\mathit{Selector}\} \\
\quad O_C(L) = \{m : \mathit{MSG} \mid \Diamond(m \in \operatorname{ran} L.\mathit{Received}.inq!) \bullet m.\mathit{Selector}\}
\end{array}
$$

Component Interface

With the previous definitions, the interface of a component can be specified as the set of message selectors that will *ever* be sent or received through its mailbox (figure 1):

$$I, O : \downarrow Component \rightarrow \mathbb{P}\, MsgID$$

$$\forall\, C : \downarrow Component \bullet$$
$$\textbf{let}\ related == \cup_{i=1}^{C.mbox.nc} C.mbox.controllers(i).RelatedMsgIds_In \bullet$$
$$O(C) = \{s : MsgID \mid \Diamond(s \in O(C.mbox))\}$$
$$I(C) = \{s : MsgID \mid \Diamond\,(s \in I(C.mbox) \cap ((\{?, !\} \times C.Methods) \cup related))\}$$

These functions are monotonic, in the sense that if a message selector ever belongs to an interface set, then it belongs to it from that moment on. Using temporal logic provides a nice way of reasoning about asynchronous message interfaces, since it does not only allow to account for the messages that have gone through a mailbox (which constitute the usual semantics for message passing systems [1]), but also to consider the messages that will *eventually* go in or out through it.

Wrapping

So far we have talked about *wrapping* components with controllers when attaching a controller to their mailboxes. We can now formally state it:

Theorem 1 *Let $C{:}\downarrow Component$ be a component, and $L{:}\downarrow Controller$ a controller. Then*

$$I(C.mbox) \subseteq O_C(L), O(C.mbox) \subseteq I_C(L), I(C.mbox') = I_E(L), O(C.mbox') = O_E(L)$$

where $C.mbox$ and $C.mbox'$ represent C's mailbox before and after attaching L to it, respectively.

Proofs have not been included for space reasons. However, they are all based on standard Z reasoning mechanisms, and most of them are straightforward consequences of the previous definitions and Object-Z operators laws. In particular, theorem 1 can be easily proved using induction over the number of controllers attached to the component.

In the following, C^L will denote the component obtained by wrapping component C with controller L, and the composition of two controllers will be denoted by $C^{L_1 \circledcirc L_2} == \left(C^{L_1}\right)^{L_2}$.

Operation $\circledcirc$ is not commutative or associative in general, although we will say that two properties $\mathcal{L}_1$ and $\mathcal{L}_2$ are *independent* if the composition of their controllers L_1 and L_2 is commutative, i.e. $\forall\, C : \downarrow Component \bullet (C)^{L_1 \circledcirc L_2} = (C)^{L_2 \circledcirc L_1}$. We can also define the composition of more than two controllers:

$$(C)^{L_1 \circledcirc \ldots \circledcirc L_n} == \left((C)^{L_1 \circledcirc \ldots \circledcirc L_{n-1}}\right)^{L_n}.$$

Internal message passing inside a controller

The following four functions determine the internal message passing inside a controller (fig. 2):

$$In2Component, In2Environment : MsgID \nrightarrow \mathbb{P}\, MsgID$$
$$Out2Component, Out2Environment : MsgID \nrightarrow \mathbb{P}\, MsgID$$

$$\mathrm{dom}\, In2Component = \mathrm{dom}\, In2Environment = RelatedMsgIds_In$$
$$\mathrm{dom}\, Out2Component = \mathrm{dom}\, Out2Environment = RelatedMsgIds_Out$$

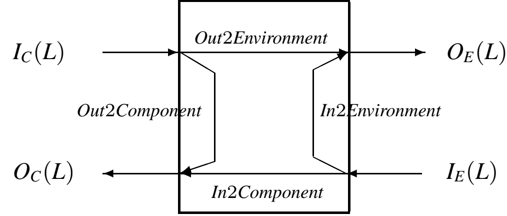


Figure 2 Internal message passing inside a controller.

With them it is possible to relate the set of outputs of a controller L to the set of its inputs, expressing the former in terms of the latter:

$$\begin{aligned} O_C(L) = {} & \big(I_E(L) \setminus L.RelatedMsgIds_In\big) \cup \\ & In2Component(\!|\; I_E(L) \cap L.RelatedMsgIds_In \;|\!) \cup \\ & Out2Component(\!|\; I_C(L) \cap L.RelatedMsgIds_Out \;|\!) \end{aligned} \qquad \text{[O1]}$$

$$\begin{aligned} O_E(L) = {} & \big(I_C(L) \setminus L.RelatedMsgIds_Out\big) \cup \\ & Out2Environment(\!|\; I_C(L) \cap L.RelatedMsgIds_Out \;|\!) \cup \\ & In2Environment(\!|\; I_E(L) \cap L.RelatedMsgIds_In \;|\!) \end{aligned} \qquad \text{[O2]}$$

COMPATIBILITY AND REPLACEABILITY

There are two important concepts when building up applications from components and managing their evolution: compatibility and replaceability. Compatibility ensures interface matching between components, that is, that exchanged methods between them can be understood by each other. Replaceability deals with the ability of a component to substitute another without forcing the application to evolve.

In this section those concepts will be defined more precisely but, before we start, we need to restrict the functions that determine the interface of a component with regard to the targets and originators of the messages:

$$\begin{array}{|l} I_From, O_To : \downarrow Component \times \mathbb{P}\,ADDR \rightarrow \mathbb{P}\,MsgID \\ \hline \forall\, C : \downarrow Component, X : \mathbb{P}\,ADDR \bullet \\ \quad I_From(C, X) = I(C) \cap \{m : MSG \mid m.From \in X \bullet m.Selector\} \\ \quad O_To(C, X) = O(C) \cap \{m : MSG \mid m.To \in X \bullet m.Selector\} \end{array}$$

We also need to select the output messages that are produced for a given input message. For each component C and a set of (input) message selectors, the following function *O_For* returns the set of selectors that the component outputs to implement them:

$$\begin{array}{|l} O_For : \downarrow Component \times \mathbb{P}\,MsgID \rightarrow \mathbb{P}\,MsgID \\ \hline \forall\, C : \downarrow Component, M, N : \mathbb{P}\,MsgID \bullet \\ \quad O_For(C, M) \subseteq O(C) \\ \quad O_For(C, I(C)) = O(C) \\ \quad M \subseteq N \Rightarrow O_For(C, M) \subseteq O_For(C, N) \end{array}$$

In particular, $O_For(C, \varnothing)$ is the set of message selectors that a component sends by its own initiative. Finally, we can also combine both interface restrictions and define:

$$O_For_To : \downarrow Component \times \mathbb{P}\, MsgID \times \mathbb{P}\, ADDR \rightarrow \mathbb{P}\, MsgID$$

$$\forall\, C : \downarrow Component, M : \mathbb{P}\, MsgID, X : \mathbb{P}\, ADDR \bullet$$
$$O_For_To(C, M, X) = O_For(C, M) \cap O_To(C, X)$$

Another important interface restriction has to do with the messages that request information about the methods implemented by a component. We knew from the second history invariant of class *Component* that components should reply to those messages. Having this into account we can concentrate only in the messages that request method invocations, and therefore define:

$$Invocations == \{s : MsgID \mid Decorator(s) \in \{!, \mathrm{Re}:!\}\}$$

Based on this set, the following function restricts the message selectors:

$$_^{\star} : \mathbb{P}\, MsgID \rightarrow \mathbb{P}\, MsgID$$

$$\forall\, S : \mathbb{P}\, MsgID \bullet S^{\star} = S \cap Invocations$$

Equivalence and Compatibility

We are now in a position to define the concepts of equivalence ($\cong$) and compatibility ($\leftrightarrows$) of components:

$$_ \cong _ : \downarrow Component \leftrightarrow \downarrow Component$$

$$\forall\, c_1, c_2 : \downarrow Component \bullet c_1 \cong c_2 \Leftrightarrow I(c_1)^{\star} = I(c_2)^{\star} \wedge O(c_1)^{\star} = O(c_2)^{\star}$$

$$_ \leftrightarrows _ : \downarrow Component \leftrightarrow \downarrow Component$$

$$\forall\, c_1, c_2 : \downarrow Component \bullet c_1 \leftrightarrows c_2 \Leftrightarrow$$
$$O_To(c_1, c_2.mbox.Addr)^{\star} \subseteq I_From(c_2, c_1.mbox.Addr)^{\star}$$
$$O_To(c_2, c_1.mbox.Addr)^{\star} \subseteq I_From(c_1, c_2.mbox.Addr)^{\star}$$

Informally speaking, two components are said to be equivalent if they have the same interface, and compatible if their interfaces match, i.e. all the messages they send to each other are understood by the receiver.

Relation $\cong$ is an equivalence relation and $\leftrightarrows$ is commutative.

With this definitions it is obvious to prove that attaching to a component a controller that does not modify any message produces an equivalent component to the original one:

Theorem 2 *Let* $C{:}\downarrow Component$ *be a component,* $L : Controller$ *a controller of class Controller (that does not modify messages), and* C' *the resulting component of wrapping C with L. Then C and* C' *are equivalent, i.e.* $C' \cong C$.

The previous definitions also allow us to get rid of the particular implementations of the components when specifying applications. In our model, an *application* is defined as

parallel composition of equivalent classes of components (modulo $\cong$), in which their components are pairwise compatible.

For simplicity, in the rest of the paper we will identify applications by the addresses of the mailboxes of their constituent components, and therefore define: $\mathit{Application} == \mathbb{P}\,\mathit{ADDR}$.

Replaceability

From the point of view of the developer of a controller L, it is very interesting to be able to prove properties about the behavior of the component obtained by wrapping a given component C with L. In this sense we talk about *replaceability* of components, that deals with the level of backwards-compatibility between two versions of a component. More precisely,

$$
\begin{array}{l}
_ \sqsubseteq _ : \downarrow\!\mathit{Component} \leftrightarrow \downarrow\!\mathit{Component} \\
\hline
\forall\, c_1, c_2 : \downarrow\!\mathit{Component} \bullet c_1 \sqsubseteq c_2 \Leftrightarrow \\
\qquad I(c_1)^{\star} \subseteq I(c_2)^{\star} \wedge \mathit{O_For}(c_2, I(c_1))^{\star} = O(c_1)^{\star}
\end{array}
$$

If $c_1 \sqsubseteq c_2$ we say that component c_1 is *replaceable* by c_2, or that c_2 is *backwards compatible* with c_1. In other words, we are asking c_2 to understand and implement all methods that c_1 implements, and that c_2's outputs when implementing them are the same as c_1's. Relation $\sqsubseteq$ defines a partial order, modulo $\cong$.

Analogously, we can define *relative replaceability* with regard to the components of a given application A as:

$$
\begin{array}{l}
_ \sqsubseteq_{_} _ : \mathit{Application} \rightarrow (\downarrow\!\mathit{Component} \leftrightarrow \downarrow\!\mathit{Component}) \\
\hline
\forall\, c_1, c_2 : \downarrow\!\mathit{Component}, A : \mathit{Application} \bullet c_1 \sqsubseteq_A c_2 \Leftrightarrow \\
\qquad \mathit{I_From}(c_1, A)^{\star} \subseteq \mathit{I_From}(c_2, A)^{\star} \\
\qquad \mathit{O_For}(c_2, \mathit{I_From}(c_1, A))^{\star} = \mathit{O_For}(c_1, \mathit{I_From}(c_1, A))^{\star}
\end{array}
$$

In this case we are asking c_2 to implement all methods that c_1 admits from the components of application A, and that the outputs of both c_1 and c_2 are the same when implementing them. We also can weaken this definition a little, and talk about *relative weak replaceability*:

$$
\begin{array}{l}
_ \sqsubset_{_} _ : \mathit{Application} \rightarrow (\downarrow\!\mathit{Component} \leftrightarrow \downarrow\!\mathit{Component}) \\
\hline
\forall\, c_1, c_2 : \downarrow\!\mathit{Component}, A : \mathit{Application} \bullet c_1 \sqsubset_A c_2 \Leftrightarrow \\
\qquad \mathit{I_From}(c_1, A)^{\star} \subseteq \mathit{I_From}(c_2, A)^{\star} \\
\qquad \mathit{O_For_To}(c_2, \mathit{I_From}(c_1, A), A)^{\star} = \mathit{O_For_To}(c_1, \mathit{I_From}(c_1, A), A)^{\star}
\end{array}
$$

We are allowing here c_2 to use methods from other components outside application A when implementing the methods requested by A's components, although the responses to them should be the same as c_1's. The difference we have introduced is that new components may have to 'join' application A when weakly replacing one of its components, since c_2 may requests services not implemented by any of A's components (for instance, this happens quite often in the PC world: replacing a product may force us to either upgrade others or to install new ones).

The four relations can be ordered in the following way:

$$\forall\, c_1, c_2 : \downarrow Component, A : Application \bullet$$
$$(c_1 \cong c_2) \Rightarrow (c_1 \sqsubseteq c_2) \Rightarrow (c_1 \sqsubseteq_A c_2) \Rightarrow (c_1 \sqsubset_A c_2)$$

BUILDING CONTROLLERS

One of the most important goals of any engineering discipline is the production of methods and processes for the reliable and automated construction of products, let they be bridges, buildings, cars, or whichever entities the discipline deals with. Accordingly, we cannot stop at defining the controllers; we would like to have a methodology for their systematic construction too. This methodology should allow their guided specification, together with the derivation from their formal specifications of a set of interesting outcomes:

(*a*) In the first place, the possibility of reasoning about the behavior of the modified components, in terms of their interfaces.

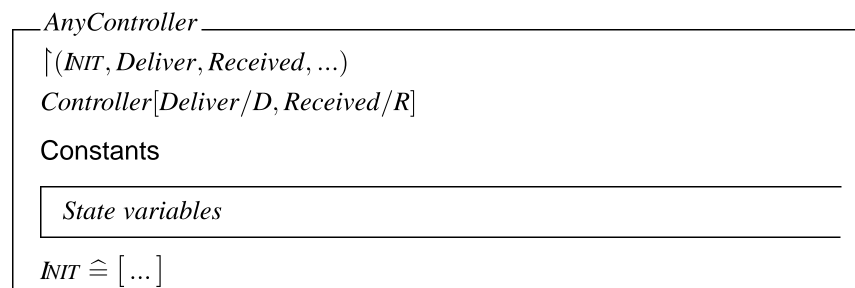
(*b*) We also need a description language (CDL) in which controllers can be defined, selected and configured by developers to build up their applications by attaching them to existing components. There should be a method in place to derive the expression in CDL of a controller right from its formal specification.

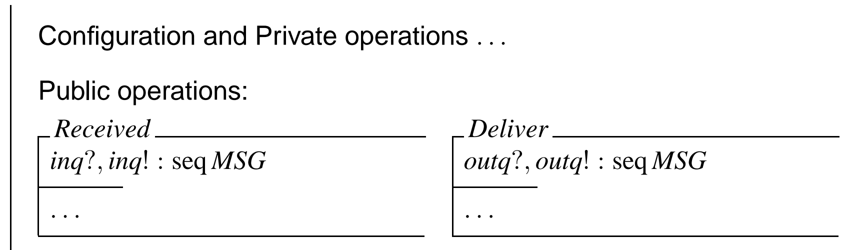
(*c*) And finally, the formal language used to specify controllers should allow the easy derivation of their implementations.

In this section we will discuss about the guided construction of the controller specifications and the second of the outcomes, i.e. the derivation of formal results. With regard to the other two outcomes, CDL is an interface textual language similar to other existing IDLs, but adapted to deal with controllers; the way the controllers are specified in Object-Z allows their interfaces to be easily written down in CDL. And regarding the derivation of code from the specifications, one of the major benefits of using a variant of Z is that this process (called *refinement* in Z terminology) is well documented and understood [15, 19].

Specifying Controllers

Since all controllers have the same structure, their specification obeys a general scheme:





There is a common process for building the class of a given controller:

(1) First, it should inherit class *Controller* and hide operations *Deliver* and *Received* in order to override them.

(2) Apart from the constants in class *Controller*, the constants of this class should represent the configuration parameters that depend on the user preferences, and whose value will be determined when instantiating the class, that is, when attaching the controller to a component.

(3) The state variables will hold the structures and variables used by the controller when accomplishing its task, and will be initialized in the *INIT* operation.

(4) *Configuration* operations are those that allow the user to dynamically consult or modify the controller state variables.

(5) *Private* operations are those auxiliary operations used by the controller for specifying its *public* operations *Deliver* and *Received*.

Deriving Formal Results

One of the goals of our work is to provide developers of controllers with a formal tool to specify the effects of wrapping components with them and to characterize the applications for which the new components can replace the old ones. This section shows the sort of results that can be obtained using our formal framework and how to achieve them. In particular, once we have the Object-Z specification of a controller L, we can study the degree of replaceability between the original component and the one obtained by wrapping it with L. The following process shows how to do it:

(1) Characterize the range of the functions that determine the internal behavior of the controller, namely *In2Component*, *In2Environment*, *Out2Component* and *Out2Environment*, in terms of the original component incoming and outgoing message sets $I(C)$ and $O(C)$.

(2) Characterize the sets $I_E(L)$ and $I_C(L)$ that describe the incoming messages that the controller deals with, using the functions obtained in the previous step.

(3) Characterize the sets of outgoing messages $O_E(L)$ and $O_C(L)$ using the previous results and the expressions [O1] and [O2].

(4) Obtain the interface of the modified component, applying theorem 1 to the expression of the interfaces of the controller in terms of the interface of the original component.

(5) And finally, restrict the sets of message selectors obtained and apply the definitions of equivalence and replaceability given in the previous section.

For example, applying the previous steps to the Object-Z specifications of the controllers of the three properties mentioned at the beginning of this paper, we can determine the degree of replaceability of the components obtained after wrapping a component with them. More precisely, if $C{:}\downarrow Component$ is a component, $\mathcal{I} : Independence$, $\mathcal{S} : SelfProtection$ and $\mathcal{A} : Adaptability$ are the controllers implementing the properties of Independence, Self-Protection and Adaptability, and $C^{\mathcal{I}}$, $C^{\mathcal{S}}$ and $C^{\mathcal{A}}$ are the resulting components after wrapping C with them, respectively, the following results can be proved:

(*a*) C and $C^{\mathcal{I}}$ are equivalent, i.e. $C^{\mathcal{I}} \cong C$.

(*b*) C and $C^{\mathcal{S}}$ are equivalent, i.e. $C^{\mathcal{S}} \cong C$.

(*c*) C is *weakly replaceable* by $C^{\mathcal{A}}$ with regard to the components of any application A, i.e. $\forall A : Application \bullet C \sqsubset_A C^{\mathcal{A}}$.

Intuitively, these results are based on the way those controllers work. In the first place, a controller of the property of Independence may change the target components of the messages, but not their selectors. A Self-Protection controller does not change the interface of the component since it just generates the answers to messages when they are not received. However, an Adaptability controller either extends the functionality provided by the component, or 'translates' its outgoing messages in order to make them compatible with the components in its environment. Extension of functionality does not jeopardizes backwards compatibility since it is just an extension, but translation is achieved through the use of *translators*, independent components which need to join in the application if not already included in it.

RELATED WORK

With regard to the formal aspects of our contribution, they can be related to two main areas of research: the specification of first-class entities that reflectively modify the behavior of components, and the notions of compatibility and replaceability of components. Comparison between our model and other component platforms for open and distributed systems is outside the scope of this paper, and can be found in [17].

In the first place, the meta-entities of most of the reflective models (meta-actors, filters, wrappers, etc.) are usually defined as language extensions and implemented using the base language. Thus, they are determined by several special syntactic constructs and their associated semantics. Some of them, like the Composition Filters [5], have very simple semantics since those filters have no state and cannot accept messages. On the contrary, the Actors model has rather complicated semantics [3] based in λ-calculus, and specifying non-trivial meta-actors (like the Real-Time Synchronizers [14] or the Communicators [16]) is a complex task as claimed by their own authors. Moreover, there seems to be no easy way to generalize those specifications to produce the specifications of general meta-actors. On the other hand, other models like LayOM [6] or the Message Filters [10], sit in between but few of them have actually

defined their semantics. And in all cases, the meta-entities used in most of the reflective architectures lack of uniform structure and definition. Our contribution follows a different approach, using a formal notation for specifying the meta-components and their behavior in a standard and uniform way, and independently from any (object-oriented) base language. This has great advantages in terms of productivity as reuse is fostered, and also eases the problems of formal reasoning about them.

Regarding the concepts of compatibility and replaceability of components, we have limited ourselves to the syntactic level, i.e. to interfaces defined in terms of the component methods, without extending them with constraints [12] or protocols [20]. Our definitions are more in the style of [9], where incoming and outgoing messages are also used for defining the interface of objects, and the notion of 'environment' is introduced. However, both the objects and the environment can block messages in their model, and the internal state transitions of an object are also taken into account, while in our model components are black boxes and therefore we cannot consider their internal state. Besides, we have also introduced the use of temporal logic for reasoning about the component interfaces instead of using traces; this greatly simplifies the definitions and the proofs of many of the results.

Finally, process algebras like π-calculus and other formal notations like CHAM are also used by some authors for specifying their connectors. In our case, benefits of Z and Object-Z overcome their limitations (weak support for reasoning about concurrency and dynamic behavior of systems) since our work is primarily focused on the specification of the model entities, and not so much on their concurrent behavior; this is why we did not even use any mixture of static and dynamic formal notations as other authors do (e.g. [7]).

CONCLUSIONS

The increasing use of open and distributed systems for the development of applications, together with the increasing needs of a global component marketplace, are changing the way software is developed nowadays. Reusability and late composition are two driving forces towards the separation of the computational and interoperational aspects of components, forcing ODS-specific requirements to be incorporated into user applications in an modular and independent manner [4]. The present work focuses on this topic, providing a formal framework built around a component model that uses modular, independent, composible controllers to modify the behavior of components according to the user requirements and other context-specific concerns. More general than the meta-objects of [2, 5, 10], and more specific than aspects [11], controllers not only deal with the interaction between components, but also with the enforcement of properties over them. In this paper we have shown how the framework can be used to specify them, reason about their behavior and characterize the degree of replaceability obtained when wrapping components with controllers.

Having a methodology and a formal framework to reason about reusable components and controllers is a further step towards the goal of having engineering processes for developing software applications from reusable entities, still far from becoming a reality, but close enough to actively pursue it.

References

[1] G.R. Andrews. *Concurrent Programming. Principles and Practice*. The Benjamin/Cummings Publishing Company, Inc., 1991.

[2] G. Agha. *Actors: A Model of Concurrent Computation*. MIT Press, 1986.

[3] G. Agha et al. *A Foundation for Actor Computation*. In Journal of Functional Programming, 1996.

[4] G. Agha. *Compositional Development from Reusable Components Requires Connectors for Managing Both protocols and Resources*. Workshop on Compositional Software Architectures. California, 1998.

[5] M. Aksit et al. *Abstracting Object-Interactions using Composition Filters*. Object-Based Distributed Processing, LNCS 791, Springer-Verlag, 1993.

[6] J. Bosch. *Language Support for Component Communication in LayOM*. Workshop Reader of ECOOP'96. Max Muehlhaeuser (ed.). Dpunkt Verlag, 1997.

[7] P. Ciancarini and C. Mascolo. *Analyzing and Refining an Architectural Style*. In Proc. of ZUM'97, LNCS 1212, Springer-Verlag, 1997.

[8] R. Duke, G. Rose and G. Smith. *Object-Z: a Specification Language Advocated for the Description of Standards*. Tech. Report 94–45, Univ. of Queensland, 1994.

[9] R. Duke, C. Bailes and G. Smith. *A Blocking Model for Reactive Objects*. Formal Aspects of Computing, 8(3):347–368, 1996.

[10] R.K. Joshi, N. Vivekananda and D. Janaki Ram. *Message Filters for Object-Oriented Systems*. Software–Practice and Experience, 27(6):677–699, 1997.

[11] G. Kiczales et al. *Aspect-Oriented Programming*. In Proc. of ECOOP'97, LNCS 1241, Springer-Verlag, 1997.

[12] D. Lea. *Interface-Based Protocol Specification of Open Systems using PSL*. In Proc. of ECOOP'95, LNCS 952, Springer-Verlag, 1995.

[13] B. Moszkowski. *Executing Temporal Logic Programs*. Cambridge University Press, 1986.

[14] S. Ren and G.A. Agha. *A Modular Approach for Programming Distributed Real-Time Systems*. In JPDC, Special Issue on OO Real-Time Systems, 1996.

[15] J.M. Spivey. *The Z Notation. A Reference Manual*. 2nd Ed. Prentice Hall, 1992.

[16] D.C. Sturman. *Modular Specification of Interaction Policies in Distributed Computing*. PhD thesis. University of Illinois at Urbana-Champaign, 1996.

[17] J.M. Troya and A. Vallecillo. *A Reflective Component Model for Open Systems*. In Proc. of the Workshop on Reflective OO Progr. and Systems, ECOOP'98, 1998.

[18] G. Wiederhold. *Mediation in Information Systems*. ACM Comp. Surveys, 27(2):265–267, June 1995.

[19] J. Woodcock and J. Davies. *Using Z: Specification, Refinement and Proof.* Prentice Hall, 1996.

[20] D.M. Yellin and R.E. Strom. *Protocol Specifications and Components Adaptors*. ACM Trans. on Programming Languages and Systems, 19(2):292–333, 1997.