Control Encapsulation: A Calculus for Exogenous Composition of Software Components

Kung-Kiu Lau[1] and Mario Ornaghi[2]

[1] School of Computer Science, the University of Manchester
Manchester M13 9PL, United Kingdom
kung-kiu@cs.man.ac.uk

[2] Dipartimento di Scienze dell'Informazione, Universita' degli studi di Milano
Via Comelico 39/41, 20135 Milano, Italy
ornaghi@dsi.unimi.it

Abstract. In current software components models, components do not encapsulate control, and are composed by connection mechanisms which pass control from component to component. Connection mechanisms are not hierarchical in general, and therefore current component models do not support hierarchical system construction. In this paper we argue that control encapsulation by components, together with suitable composition mechanisms, can lead to a component model that supports hierarchical system construction. We show an example of such a model and present a calculus for its hierarchical composition mechanisms.

1 Introduction

In current software component models [10], components are either *objects* or *architectural units*. Such components do not encapsulate control (or computation), and are composed by *connection* mechanisms: delegation for objects (i.e. method call or event delegation), and port linking for architectural units. In general, connection as a composition mechanism does not support hierarchical system construction.

For hierarchical system construction, a composition mechanism should not be defined or applied in an *ad hoc* manner, and it should be compositional in the sense that connecting two components yields another component (of the same type) [1]. Object delegation is not defined in an *ad hoc* manner, since method calls are "hard-wired" in the caller objects, but it is not compositional: two objects connected by delegation do not yield a single object, but remains as two distinct objects (such a pair is not even a standard type).

On the other hand, architectural unit composition is compositional, but its application may be done in an *ad hoc* manner. Ports can be linked if they are compatible, and the resulting composite is another architectural unit; even the connectors can be generated automatically. However, there may be many possible combinations of compatible ports, and therefore a choice of connections has to be made on a case by case basis. Moreover, some ports may also be left unconnected, and the resulting composite unit can be defined in different ways, depending on whether and which of the unconnected ports are exported to the interface of the composite, or simply disposed of.

G.A. Lewis, I. Poernomo, and C. Hofmeister (Eds.): CBSE 2009, LNCS 5582, pp. 121–139, 2009.
© Springer-Verlag Berlin Heidelberg 2009

We believe that control encapsulation in components, and composition mechanisms that are compositional with respect to, i.e. preserve, control encapsulation, can make a significant contribution to component-based development (CBD), by providing component models that support hierarchical system construction. In this paper, we discuss control encapsulation and its role in component models. In particular, we show how we achieve control encapsulation by composition mechanisms in a model that we have formulated [7, 9]. By defining a calculus for these mechanisms, we show that they support hierarchical system construction.

2 Control Encapsulation

A software system consists of three elements: *control*, *computation* and *data*. The system's behaviour is the result of the interaction between these elements. In a component-based system, these elements can be distributed among the components, or they can be shared by the components, to varying degrees. However, the purpose of using components is to maximise distribution and to minimise sharing at the same time. That is, each component should encapsulate as much of these elements as possible, so as to minimise coupling between the components. By 'encapsulation' we mean 'enclosure in a capsule', as defined in the Oxford English Dictionary.

On the other hand, since the whole system is constructed by composing components, encapsulation in the components should not hinder their composition. Ideally it should be possible to do the construction in a hierarchical manner, so that it is easier to construct large systems systematically, and to reason about their properties. That is, ideally encapsulation should not hinder systematic or hierarchical composition.

Therefore an ideal approach to CBD should combine encapsulation and composition in such a way that composition *preserves* encapsulation. This is illustrated by Fig. 1, where two components C1 and C2, each with their own encapsulation, are composed into a composite C3, which also has its own encapsulation as a result of the composition. Such an approach would allow component-based systems to be constructed from decoupled components in a hierarchical fashion, with encapsulation at every level. However, it would require the components to be *compositional*, i.e. for a given definition of components, the result of composing two components is also a component (with encapsulation).

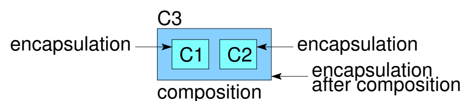


Fig. 1. Compositional encapsulation

Control encapsulation means that a component does not leak control. This means two things. First, it means that any control originated in a component does not leak out to another component. For example, in coordination languages [5], components are active, or have their own threads, i.e. originate their own control, but control never leaks out from components. Rather, components read input values from its (input) ports and outputs values to its (output) ports. The chosen coordination model then coordinates the distribution of the values on the output ports of the components to their input ports. Thus in coordination, components encapsulate control.

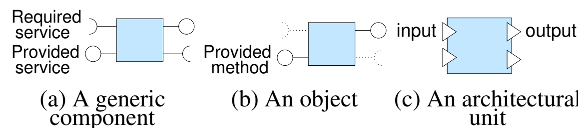
However, coordination is not concerned with preservation of encapsulation in composites, as in Fig. 1, because coordination does not build composites from sub-components. That is, coordination is not a composition mechanism for components: it leaves components ‘as is’. Web service orchestration [17] is an example of such a coordination mechanism.

Secondly, control encapsulation means that when control is passed to a component, e.g. by a caller invoking a method in the component, the component returns the control to the caller upon completing its execution of the called method, without leaking it (to another component) during its execution of the call. Again, web services encapsulate control in this sense, but again, their orchestration does not compose components into composites.

In this section, we consider control encapsulation by composition mechanisms in current software component models [10]. In these models, components are either objects or architectural units. Exemplars of these models are EJB [4] and ADLs (architecture description languages) [11] respectively. In these models, components are composed by connection. We will show that components in these models do not encapsulate control, and connection does not support hierarchical system construction.

2.1 Connection

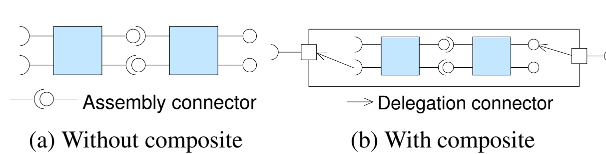
A generic view of a component is a composition unit with required services and provided services. Following UML2.0 [16], this is expressed as a box with lollipops (provided services) and sockets (required services), as shown in Fig. 2(a). An object normally does not have an interface, i.e. it does not specify its required services or its provided services (methods), but in component models like JavaBeans and EJB, beans are objects with an interface showing its provided methods but usually not its required services (Fig. 2(b)). Architectural units have input ports as required services and output ports as provided services (Fig. 2(c).) Therefore, objects and architectural units can both be represented as components of the form in Fig. 2(a).



(a) A generic component (b) An object (c) An architectural unit

Fig. 2. Components

A required service represents an external dependency. A component with an external dependency is not encapsulated, in the sense of ‘enclosure in a capsule’. Therefore, objects and architectural units do not have encapsulation, in particular control encapsulation. We will show that control leaks out of these components.

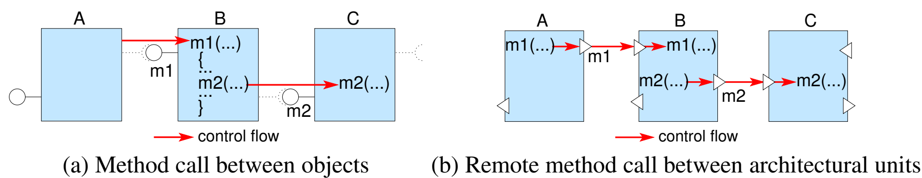


(a) Without composite (b) With composite

Fig. 3. Connection

Objects and architectural units are composed by connection (Fig. 3),[1] whereby matching provided and required services are connected by assembly connectors (Fig. 3(a)). We will show that connection does not always support hierarchical system construction.

In order to get a required service from another object, an object calls the appropriate method in that object. Thus objects are connected by method calls, i.e. by direct message passing (Fig. 4(a)). For architectural units, connectors between ports provide



(a) Method call between objects (b) Remote method call between architectural units

Fig. 4. Control flow in connection

communication channels for indirect message passing (Fig. 4(b)). In Fig. 4 it is clear that neither objects nor architectural units encapsulate control that either originates in them or is passed to them. When an object A makes a method call to another object B (Fig. 4(a)), it passes control to the callee object B; if during its execution the called method calls a method in another object C, then the object B passes control to C. C in turn may call a method in another object, and so on. The result is that control is leaked by B before it is (eventually if at all) passed back to A. When an architectural unit requires the service of another unit, it initiates control to invoke that service by passing control to that unit, albeit indirectly via their (connected) ports (Fig. 4(b)). Unlike objects, however, in an architectural unit, the required services are not necessarily invoked by provided services; rather, the unit can initiate control independently. For example, whereas in Fig. 4(a), in B the method m2 is invoked by m1, in Fig. 4(b), m2 is invoked not by m1 but by control initiated by B.

Connecting two components may or may not produce a composite. Object delegation does not result in a new (composite) object; rather, it produces a connection between two objects which retain their original identities (Fig. 3(a)). The unconnected services remain available for connecting to other objects. Clearly object delegation does not support hierarchical composition.

Architectural unit composition can produce a composite with its own ports (i.e. another architectural unit with its own identity), but the exact nature of this composite depends on whether and which of the unconnected ports are exported (by delegation connectors) to the composite's interface, or disposed of (Fig. 3(b)). In some component models, the notion of a composite is not clearly defined, and the unexported unconnected ports inside the composite can even remain available for connecting to other units. In general, therefore, architectural unit composition does not always support hierarchical composition.

[1] In [22] Szyperski classifies object delegation as object-oriented composition, and architectural unit composition as connection-oriented composition. We generalise both as connection.

3 Control Encapsulation in Exogenous Composition

We have formulated a component model [9, 7] in which components encapsulate control. In this section we briefly recall our model, and show how it achieves control encapsulation.

In our model, components are encapsulated: they encapsulate control, data as well as computation. Our components have no external dependencies, and can therefore be depicted as shown in Fig. 5(a), with just a lollipop, and no socket. There are two basic

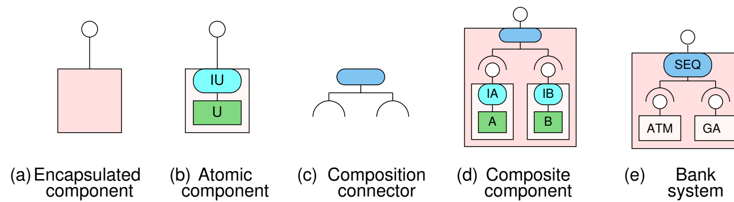


Fig. 5. Encapsulated components and exogenous composition

types of components: (i) *atomic* and (ii) *composite*. Fig 5(b) shows an atomic component. This consists of a *computation* unit (U) and an *invocation connector* (IU). A computation unit contains a set of methods which do not invoke methods in the computation units of other components; it therefore encapsulates computation. An invocation connector passes control (and input parameters) received from outside the component to the computation unit to invoke a chosen method, and after the execution of method passes control (and results) back to whence it came, outside the component. It therefore encapsulates control.

A composite component is built from atomic components by using a *composition connector*. Fig. 5(c) shows a composition connector. This encapsulates a control structure, e.g. sequencing, branching, or looping, that connects the sub-components to the interface of the composite component (Fig. 5(d)). Since the atomic components encapsulate computation and control, so does the composite component. Our components therefore encapsulate control (and computation) at every level of composition. In fact they also encapsulate data at every level of computation [8] but we omit data encapsulation here for simplicity and for lack of space.

Our components are thus passive components that can be invoked. In typical software applications, a system consists of a 'main' component that initiates control in the system, as well as components that provide services when invoked, either by the 'main' component or by the other components. The 'main' component is active, while the other components are passive. Our components are the latter. They receive control from, and return it, to connectors. In a system, control flow starts from the top-level (composition) connector.

Fig. 5(e) shows a simplified bank system with two components *ATM* and *GA*, composed by a sequencer composition connector *SEQ*. Control starts when the customer keys in his PIN (and maybe also the operation he wishes to carry out). The connector *SEQ* passes control to *ATM*, which checks the customer's PIN; then it passes control to *GA*

(get account), which *g*ets hold of the customer *a*ccount details (and possibly perform the requested operation). Control then passes back to the customer. Our composition connectors are *exogenous connectors* [9], encapsulating control as they do outside atomic components. Clearly, exogenous composition is hierarchical: components can be 'recursively' composed into larger composites. This is a consequence of control encapsulation.

3.1 Defining Control Encapsulation

So far we have defined control encapsulation informally. In this section we begin to define it formally. The basic mechanism used for exogenous composition is as shown in Fig. 5(d). We will elaborate on this.

The general picture of exogenous composition is given in Fig. 6, where a composition connector is connected to a component C (which may be atomic or composite). More precisely, a socket of the connector is connected to the lollipop of C. Control is passed from the connector to C via the socket, a method in C is invoked, and upon completion of its execution, control (and result) is passed back to the connector.

To define this control flow, we use $?C.q$ to denote a *request* q received by a component C, and $!C.r$ to denote a *result* r returned by C (Fig. 6(a)). Receiving a request ($?C.q$) and sending a result ($!C.r$) will be called "events". A request to C could be a method call, and a result returned by C the value(s) that result from the method execution. Fig. 6(b) shows a method $m(x:T):R$ in C (with parameter x of type T and result of type R). A method call $m(v)$ to m is denoted by $?C.m(v)$, which means the event "C receives the request $m(v)$". The corresponding result w returned by m is denoted by $!C.m\ w$, which means the event "C returns the result w of m". We write $?C.m(v) \preceq\ !C.m\ w$ to indicate that the request $?C.m(v)$ "causes" the result $!C.m\ w$, where $\preceq$ is a partial ordering on the events. The dot notation $C.m$ follows the object-oriented convention, whilst the punctuation marks ? and ! are often used in the above sense in CCS [13], while causality relations are typical of semantics based on *posets* (partially ordered sets) [21].

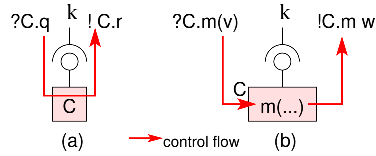


Fig. 6. Exogenous composition

We will use the relation $\preceq$ in a weaker sense: $e_1 \preceq e_2$ indicates that the event e_1 must precede e_2 for some causal or non-causal reason, where an event may be a request or a result. That is, we relax the causal relationship to one of precedence. This allows us to express protocol requirements for the control flow. For example, an ATM terminal *ATM* may have the protocol:

$$\{?ATM.get(card,pin) \rightarrow_{ATM}\ !ATM.get\ ok,\ !ATM.get\ ok \rightarrow_{ATM}\ ?ATM.sel(op)\} \quad (1)$$

where we use $\rightarrow_{ATM}$ to denote the covering relation[2] of the protocol. That is, before asking the user u to select an operation, the login process must end successfully. Conversely, the user (caller) has to conform to the protocol with the following covering relation $\rightarrow_u$:

[2] A finite poset $\langle E, \preceq \rangle$ can be represented by a set of arcs $\{e_1 \rightarrow e'_1, \ldots, e_n \rightarrow e'_n\}$, where $\rightarrow$ is the covering relation ($e \rightarrow e'$ if $e \neq e'$ and $e \preceq e'' \preceq e'$ entails $e'' = e$ or $e'' = e'$).

$$\{!ATM.get(card, pin) \rightarrow_u ?ATM.get\ ok,\ ?ATM.get\ ok \rightarrow_u !ATM.sel(op)\} \quad (2)$$

i.e., send *card* and *pin* to the ATM, wait for the *ok*, and then select the operation. Each event in the *ATM* must synchronise with a dual one in the caller, where for a component C, the dual of $?C.q$ is $!C.q$ (the caller sends q and C receives it) while the dual of $!C.r$ is $?C.r$ (C returns r and the caller receives it). Our dual events correspond to complementary events in [14]. The main difference is that our model components and connectors do not behave symmetrically. Events will always refer to the component C and their duals to the connector or, more in general, to the environment interacting with C. The protocol requirements (1) and (2) together define the control flow for using the ATM to execute an ATM operation.

A protocol specifies the *expected behaviours* of a component C by posets of events such as (1), and a compatible connector k has dual behaviours such as (2). Compatibility means that the events of the expected behaviours of C synchronise with the dual ones in k, while respecting the partial ordering of the protocol (behaviours are related to the execution ordering, in general stricter than the one of the protocol). As for control encapsulation, we begin with a simplistic but very strong requirement, which will be weakened later. We say that:

> A behaviour of a component C has *control encapsulation* iff for every request $?C.q$ and result $!C.r$, $!C.r \rightarrow_C ?C.q$ does not belong to the behaviour.

It follows that:

> A component has *control encapsulation* if all its expected behaviours have control encapsulation.

Intuitively, $!C.r \rightarrow_C ?C.q$ is a "call back requirement": the result $!C.r$, sent to the environment of C, "pretends" that the environment will eventually call C again. We see this as a form of control exported from C to the environment. We show an example of control encapsulation and, then, a counterexample.

Example 1. Consider the component C in Fig. 6(b), with a single method $m(x : T) : T$. The expected behaviours of C are of the form

$$?C.m(v) \rightarrow_C !C.m\ w$$

for every call $m(v)$ and returned result w. Therefore C has control encapsulation. The compatible dual behaviours of the connector contains $!C.m(v) \rightarrow_k ?C.m\ w$. The control flows from the connector to C when $!C.m(v)$ and $?C.m(v)$ react,[3] i.e., when the connector makes the call $m(v)$. During the whole execution of m, control is encapsulated in C and comes back to the connector only when $?C.m\ w$ and $!C.m\ w$ react, i.e., when C returns w and the connector receives it. This is depicted in Fig. 7, where the solid arrows correspond to the covering relation and the dotted arrows to the reaction between dual events.

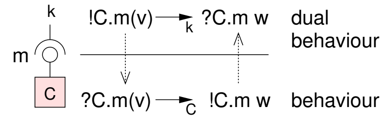


Fig. 7. Encapsulated control

[3] In the sense of CCS.

As counter examples to control encapsulation, consider objects and architectural units composed by connection (Fig. 4) again.

Example 2. Consider the objects A, B and C in Fig. 4(a). Suppose the method in B is $m1(x : T) : T$ and the method in C is $m2$. A calls $m1$ in B, and while executing $m1$, B calls $m2$ in C. In this case, the behaviours of B and the dual ones of A and C are of the form depicted in Fig. 8. The objects A and B, together, form the environment e of B, and A starts the control flow as follows: (1) A calls $B.m1(v_1)$ and B starts; (2) B calls $C.m2(v_2)$ and C starts; (3) B obtains w_1 and uses it to get the final result w_2. The task (1) is performed inside B, (2) inside C and (3) again in B. B does not encapsulate control, because control flows to C before the call $m1(v_1)$ ends. This is reflected by the requirement $!C.m2(v_1) \rightarrow ?C.m2\ w_1$ where a request follows a result.

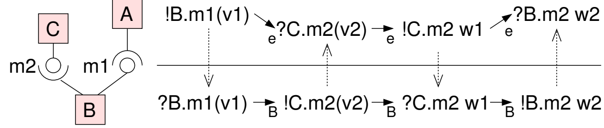


Fig. 8. Non-encapsulated control

Similarly, we can show that the architectural unit B in Fig. 4(b) does not encapsulate control.

The above (informal) definition of control encapsulation is too strong, as the following example shows.

Example 3. Consider Example 1, but this time in a component B allowing C to be called by the connector k a number of times. Suppose we observe a behaviour of C of the form:

$$\{?C.m(v) \rightarrow_C !C.m\ w,\ !C.m\ w \rightarrow_C ?C.m(v'),\ ?C.m'(v') \rightarrow_C !C.m\ w'\}$$

In this case, in each of the calls $C.m(v)$ and $C.m(v')$, the control remains encapsulated in C until the end of the execution of m. However, according to our strong definition of control encapsulation, the behaviours of C violate control encapsulation, owing to $!C.m\ w \rightarrow ?C.m(v')$. The fact that we can observe this kind of behaviour is not a real lack of control encapsulation.

Indeed, the above observed behaviour is compatible with the requirements:

$$?C.m(v) \rightarrow !C.m\ w,\ \ ?C.m(v') \rightarrow !C.m\ w'$$

which are just two instances of the protocol considered in Example 3.1. Thus, according to our model, C has control encapsulation. In Fig. 9, we have used a dashed line for the arrow $!C.m\ w \rightarrow ?C.m(v')$, to put into evidence that it does not come from the protocol of the component C, but from the connector. We remark that the connector does not satisfy control encapsulation. In general, control is not encapsulated in a connector k, since it has to

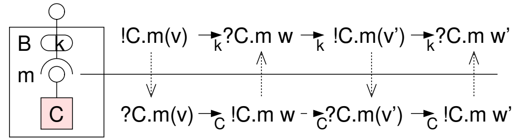


Fig. 9. Encapsulated control and connectors

connect the sub-components and coordinate their computation. A connector is only required to maintain the control encapsulated in the component containing it. In our example, k calls C a number of times, while maintaining control encapsulated in B.

We will relate control encapsulation to the interface behaviour specified by the protocol. In our example, C has control encapsulation since its interface protocol does not contain any "call back" requirement. In contrast, if the requirement $!C.m\,w \to ?C.m(v')$ belongs to the protocol of C, then the control *must* come back to C. This corresponds to the fact that we can use messages as a form of "control". For example, the result w may be a string containing the message "please, call me back again".

The example shows that we have to distinguish between the behaviour exhibited in an observation from the requirements of the protocol. Thus our weaker definition of control encapsulation:

We say that a component violates control encapsulation *if its protocol does*.

Furthermore, when we compose components, we *compose their specifications* (which include protocols), i.e., we are looking for composition rules that allow us to reason at the specification level.

Having explained the notion of control encapsulation and how it can be defined using posets as *behaviour specifications*, we now show that exogenous composition (using exogenous connectors) is strictly hierarchical, by presenting a calculus for it.

4 A Generic Calculus for Exogenous Composition

In this section we outline a calculus for exogenous connectors, which allows us to build new components on top of already existing components (atomic or composite) in a hierarchical manner. First we define behaviour specifications (BSP) precisely, and then we introduce components, their specifications and their correctness.

Definition 1. A *BSP instance* with request alphabet Q and result alphabet R is a finite labelled poset $B(Q,R) = \langle E, \preceq, \lambda, Q, R \rangle$ with labelling function $\lambda : E \to Q \cup R$, where E is a finite set of "event indexes" and:

1. for every $i \in E$, if $\lambda(i) \in Q$, then there is an $i' \in E$ such that $i \preceq i'$ and $\lambda(i') \in R$;
2. for every $i \in E$, if $\lambda(i) \in R$, then there is $i' \in E$ such that $i' \preceq i$ and $\lambda(i') \in Q$;
3. for every $i, i' \in E$, if $\lambda(i) \in R$ and $\lambda(i') \in Q$ then $i \not\preceq i'$.

In our approach, a BSP instance represents a "testable" requirement for the behaviour of a component C, so E is finite. Condition 1 says that if the control enters C with a request, it has to leave it with a result. Condition 2 says that every result is caused by a request, i.e., control cannot originate in C, while Condition 3 is our control encapsulation condition.

A label e of the alphabets $Q \cup R$ is called an event. Let i be an event index with label $\lambda(i) = e$. We call the pair $\langle e, i \rangle$ an *event occurrence* and we indicate it by e_i. When e_i has a unique index i, then i may be omitted. We use indexes just to distinguish different event occurrences. Thus behaviours are *equivalent up to reindexing*.[4] According to the

[4] A reindexing is an isomorphism $I : E_1 \to E_2$ preserving $\preceq$ and λ.

above section, we represent a behaviour instance by the labelled arcs $e_i \rightarrow e'_j$ of its covering relation, that we call requirement instances. We indicate by E_Q the subset of the indexes with label in Q, by E_R those with label in R, by $B(Q)$ the request part of $B(Q,R)$ (namely the restriction to E_Q) and by $B(R)$ the result part (namely the restriction to E_R). When we do not need to explicitly mention Q, R, we omit them.

Components are built on top of a repository of *computation units* (Fig. 5(b)). A computation unit U can be used by means of an *invocation connector* IU, giving rise to an *atomic component* also denoted by IU.[5] The connector IU defines the request and result alphabets Q_{IU} and R_{IU}. Furthermore, IU may have access to some permanent data. We model data access by means of a set of "access variables" $x_1,\ldots,x_n$ and we define a component state σ as a set of bindings $x_i \mapsto v_i$ associating each x_i with its accessed data value v_i. Finally, the BSP of IU is denoted by $[IU]$ and defines a set of behaviour triples of the form $\langle \sigma, B(Q,R), \sigma' \rangle$, where σ and σ' are states and $B(Q,R)$ is a behaviour with requests $Q \subseteq Q_{IU}$ and results $R \subseteq R_{IU}$. We will write

$$IU : \sigma \overset{B}{\twoheadrightarrow} \sigma' \tag{3}$$

to indicate that $\langle \sigma, B, \sigma' \rangle \in [IU]$, and we call (3) a BSP-requirement. We attach to each event occurrence of B an *expected property*, to be satisfied after the event occurrence. The expected property of a request concerns the external environment and the data flowing in, whilst that of a result concerns the state of IU and the data flowing out. A *test case* for $B(Q,R)$ is a sequence containing the requests of $B(Q)$ in an order consistent with $\preceq$. Similarly, a result sequence for $B(Q,R)$ is a sequence containing the results of $B(R)$ in an order consistent with $\preceq$. A behaviour triple $\langle \sigma, B(Q,R), \sigma' \rangle \in [C]$ specifies the following expected behaviour:

> if we start C in the state σ and we submit a test case T for $B(Q)$ while satisfying the expected properties of the requests, then C returns a result sequence for $B(Q,R)$ and reaches the state σ' while satisfying expected properties of the results.

We say that the test case T is successful iff C exhibits the expected behaviour. We say that C is *correct* iff for every $\langle \sigma, B(Q,R), \sigma' \rangle \in [C]$ and every test case T for $B(Q,R)$, T is successful.

Example 4. Here we give the BSP requirements of three atomic components. The *ATM* waits for a card c by $?ic(c)$ and a pin p by $?ip(p)$; if $p = pin(c)$ it gives the result ok, otherwise the result err.

It also updates the data access variable $card$. The event $ic(c)$ has the expected property "a card has been inserted, containing the data c", $ip(p)$ the property "p has been correctly composed", ok the property "$card$ contains the data c of the inserted card the composed pin p coincides with $pin(c)$", and err the property "the card has been refused". *GA* waits for the ok event and then uses the data accessed by $card$ to get the account number $num(card)$ and the bank $bank(card)$. It does not update any access variable. The expected properties should be evident from the signature. Finally, *EP* simply propagates the error event. The requirements are given in an open form, with the

[5] The invocation connector provides the interface, hence the overloading of names.

parameters c, p (the data access variables are not parameters). Each open requirement is to be considered as the set of its instances, obtained by grounding the parameters.

$$
\begin{array}{lll}
ATM: \text{if } p = pin(c) & [card \mapsto _] \overset{?ic(c)\to?ip(p),\,?ip(p)\to!ok}{\twoheadrightarrow} card\ [card \mapsto c]; \\
\qquad\quad \text{otherwise} & [card \mapsto _] \overset{?ic(c)\to?ip(p),?ip(p)\to!err}{\twoheadrightarrow} [card \mapsto _] \\
GA: & [card \mapsto c] \overset{?ok\to!acc(num(card),bank(card))}{\twoheadrightarrow} [card \mapsto c] \\
EP: & [] \overset{?err\to err}{\twoheadrightarrow} []
\end{array}
$$

An alternative specification, where the information on *card* is passed using parameters rather than by data access is:

$$
\begin{array}{ll}
ATM: \text{if } p = pin(c) & [] \overset{?ic(c)\to?ip(p),?ip(p)\to!ok(c)}{\twoheadrightarrow} []; \\
\qquad\quad \text{otherwise} & [] \overset{?ic(c)\to?ip(p),?ip(p)\to!err}{\twoheadrightarrow} [] \\
GA & [] \overset{?ok(c)\to!acc(num(c),bank(c))}{\twoheadrightarrow} [] \\
EP & [] \overset{?err\to err}{\twoheadrightarrow} []
\end{array}
$$

Now we define the semantics of the components and of the connectors by means of inference rules of the following form:

$$
\frac{}{IU : \sigma \overset{B}{\twoheadrightarrow}_{\underline{w}} \sigma'}\, IU \qquad \frac{C_1 : \sigma_1 \overset{B_1}{\twoheadrightarrow}_{\underline{w}_1} \sigma_1' \quad \ldots \quad C_n : \sigma_1 \overset{B_n}{\twoheadrightarrow}_{\underline{w}_n} \sigma_n'}{K(C_1, \ldots, C_n) : \sigma \overset{B}{\twoheadrightarrow}_{\underline{w}} \sigma'}\, K
$$

In the (IU)-rule, IU is an atomic unit, $\underline{w}$ is a subset of the access variables of IU and $IU : \sigma \overset{B}{\twoheadrightarrow}_{\underline{w}} \sigma'$ means that (3) holds and that only the variables in $\underline{w}$ may have different values in σ, σ'. In (K), K is a connector, $K(C_1, \ldots, C_n)$ denotes the component obtained from $C_1, \ldots, C_n$ by means of K and $\sigma \overset{B}{\twoheadrightarrow}_{\underline{w}} \sigma'$ is defined as a function of $\sigma_j \overset{B_j}{\twoheadrightarrow}_{\underline{w}_j} \sigma_j'$. If invocation and composition connectors are introduced by rules of the above kind, the BSP semantic $[C]$ of an atomic or composite component C can be defined as follows:

> $\langle \sigma, B, \sigma' \rangle \in [C]$ iff there is a judgement of the form $C : \sigma \overset{B}{\twoheadrightarrow}_{\underline{w}} \sigma'$ provable by the connector rules.

We remark that we decouple the BSP semantics $[C]$ and the execution semantics of C. The former specifies the expected behaviours, expressed by behaviour triples, whilst the latter corresponds to the implemented behaviour. The two semantics are related by the notion of correctness introduced before. The correctness of the basic components is assumed. The one of the composed components is guaranteed as far as connectors are correctly implemented, i.e., their execution preserves correctness. We assume that a correct implementation is supplied by a *composition environment* CE, depending on the programming language(s), the run time environment and the supported communication mechanisms. We leave the notion of “successful test case” generic, to leave open the choice of CE. For example, we could use CCS or the π-calculus as a formal CE. In

this case we can define both test cases and components as processes, and we can define successful test cases as particular kinds of “experiments” [13]. Alternatively, we could define our *CE* using Petri Nets [18], or we could choose an informal *CE* such as Net Beans. Finally, we could give an execution semantics based on our behaviours, in a way similar to [2] for the internal π-calculus.

$$\frac{C_1 : \sigma \overset{B_1}{\twoheadrightarrow} \underline{w}_1\, \sigma', \quad C_2 : \sigma \overset{B_2}{\twoheadrightarrow} \underline{w}_2 \sigma'}{par(C_1,C_2) : \sigma \overset{B_1 \dot\cup B_2}{\twoheadrightarrow} \underline{w}_1 \cup \underline{w}_2\, \sigma'} \qquad \frac{C_1 : \sigma \overset{B_1}{\twoheadrightarrow} \underline{w}\, \sigma'}{sel(C_1,C_2) : \sigma \overset{1.B_1}{\twoheadrightarrow} \underline{w}\, \sigma'}$$

$$\frac{C_2 : \sigma \overset{B_2}{\twoheadrightarrow} \underline{w}\, \sigma'}{sel(C_1,C_2) : \sigma \overset{2.B_2}{\twoheadrightarrow} \underline{w}\, \sigma'} \qquad \frac{C_1 : \sigma \overset{B_1}{\twoheadrightarrow} \underline{w}_1\, \sigma', \quad C_2 : \sigma' \overset{B_2}{\twoheadrightarrow} \underline{w}_2 \sigma'',}{pipe(p,C_1,C_2) : \sigma \overset{B_1|p|B_2}{\twoheadrightarrow} \underline{w}_1 \cup \underline{w}_2\, \sigma''}$$

$$\frac{C : \sigma \overset{B}{\twoheadrightarrow} \underline{w}\, \sigma'}{ra(C,g) : \sigma \overset{B|g}{\twoheadrightarrow} \underline{w}\, \sigma'} \qquad \frac{C : \sigma \overset{B}{\twoheadrightarrow} \underline{w}\, \sigma'}{la(f,C) : \sigma \overset{f|B}{\twoheadrightarrow} \underline{w}\, \sigma'}$$

$$\frac{B \notin dom(p)}{loop(p,C) : \sigma \overset{B}{\twoheadrightarrow} \underline{w}\, \sigma} \qquad \frac{C : \sigma \overset{B_1}{\twoheadrightarrow} \underline{w}\, \sigma', \quad loop(p,C) : \sigma' \overset{B_2}{\twoheadrightarrow} \underline{w} \sigma''}{loop(p,C) : \sigma \overset{B_1|p|B_2}{\twoheadrightarrow} \underline{w}\, \sigma''}$$

Fig. 10. The basic connectors and their semantics

In Fig. 10 we show a set of basic rules, in part inspired by constructive logic, in particular by [12, 15]. In [15], information values are introduced. An information value $\alpha : A$ for a formula A can be seen as a constructive explanation of the (classical) truth of A and the rules of the related constructive calculus can be interpreted as truth preserving operations on the information values. Here, instead of information values we have behaviour instances (BI) with attached properties. The BI can be seen as an explanation of the attached properties in terms of the events making them true. The rules of the connectors define the BSP of the composite in terms of the BSP of the components and, according to the previous discussion on the *CE*, the connector execution is required to realise the behaviour of the composite while preserving the correctness. The composition operations used in the rules are the following. For conciseness, we will briefly comment on the similarity to the information-value logic (ivl) only for the first two.

- Disjoint Union (*par* rule). The disjoint union $B \dot\cup B'$ is defined in the usual way [3]. The expected properties remain unchanged. In the *par* rule, we require also that $\underline{w}_1 \cap \underline{w}_2 = \emptyset$.

  The *par* rule is similar to the (ivl) ∧-introduction rule. The latter takes two information values $\alpha : A$ and $\beta : B$ and builds $\langle \alpha, \beta \rangle : A \wedge B$. Similarly, *par* takes two BI B_1 for C_1 and B_2 for C_2 and builds $B_1 \dot\cup B_2$ for $par(C_1,C_2)$. In the *par* composition the two components run independently, while preserving correctness. Indeed, no further ordering is imposed in $B_1 \dot\cup B_2$ and the (possible) shared access variables are not updated. There are other forms of parallel composition, involving fork and join, not considered here for conciseness.

- Prefixing (*sel* rules). The behaviour $h.B(Q,R)$ is obtained by adding the prefix h to the labels ($?q$ becomes $?h.q$ and $!r$ becomes $!h.r$). The *sel* rule is similar to the (ivl) $\vee$-elimination. An information value for $A \vee B$ is of the form $1.\alpha$, with $\alpha : A$, or $2.\beta$, with $\beta : B$. Roughly, $\vee$-elimination works as a case applying a suitable function $f_A(\alpha)$ or $f_B(\beta)$, depending on the index 1 or 2. Similarly, the selector uses the index h to choose the right component to execute while maintaining the correctness. If the signatures of the two components are disjoint, prefixing is not necessary. Furthermore, since the selection is based on the requests, one can apply prefixing only to the request part, as we will do in Section 4.1.
- Relabelling (left and right adapt $la(f,C)$ and $ra(C,g)$). Let $B(Q,R)$ be a behaviour. A left adaptor is based on an injective function $f : Q' \rightarrow Q$. It replaces every q in its range by the q' such that $f(q') = q$. The expected properties of q' must entail those of q. We indicate by $f|B$ the resulting behaviour. Dually, in $B|g$ the right adaptor relabels the results in the domain of g.
- Piping (pipe and loop rules). A pipe function p for $B_1(Q_1,R_1), B_2(Q_2,R_2)$ is a partial function $p : R_1 \rightarrow Q_2$. We require that $B_2(Q_2)$ is of the form $B_2(Q_2') \dot{\cup} B_2(Q_2'')$, where Q_2' is the range of p. The function p is used to pipe the results $r \in dom(p)$ of R_1 into requests $p(r)$ of $B_2(Q_2')$, guaranteeing that if the expected property of r is satisfied, then so is that of $p(r)$. We say that *p can be applied* if for every linear extension S of $B_1(R_1)$, there is a linear extension T of $B_2(Q_2')$ such that $\lambda(T) = p(S)$, where $\lambda(T)$ is the sequence of the labels of T and $p(S)$ is that of the labels of $p(r)$ coming from the sub-sequence of S with labels from $dom(p)$. Intuitively, this means that the pipe connector sends the requests $p(S)$ to C_2. If p can be applied, then the latter are accepted, since they arrive in the right order. Otherwise, we get an abort result. The results with labels not in $dom(p)$ and the requests of $B_2(Q_2'')$ are not piped. To obtain the composite behaviour $B_1|p|B_2$ we *delete the piped events* (while reconstructing $\preceq$, details omitted) and we further impose that *the requests of* $B_1(Q_1)$ *precede those of* $B_2(Q_2'')$ *and the non-piped result of* B_1 *precede those of* B_2.
- Sequencing. Sequencing may be defined as $B_1 \cdot B_2 =_{def} B_1|u|B_2$, where u is the function with an empty domain or, more generally, a function such that $dom(u) \cap R_1 = \emptyset$. We remark that in piping (and hence in sequencing) the requests of B_1 precede those of B_2 and the results of B_1 precede those of B_2, but we do not require that the results of B_1 precede the requests of B_2 to preserve control encapsulation.
- The loop rule of Fig. 10 is only one of the possible iterations. It halts when no result belongs to the domain of p.

We conclude this section by briefly discussing the problem of the implementation in a composition environment *CE*. The latter plays an important role in our approach. The composition rules are used in the logical design phase, to meet the system's requirements. They work at a very high level of abstraction since they compose the BSP of components, rather than their implemented behaviours. The latter will be composed by the *CE*. This may lead to inefficiency. In particular, our strictly compositional approach may introduce a huge number of composition levels, and therefore long delegation chains, which may cause inefficiency. Our idea is that, once we have a satisfactory logical design, we can optimise it by transformations supported by the *CE*. An example

of transformation is shown in Fig. 12, where we use hierarchical Petri Nets to implement the connection rules.

4.1 Typing Connectors

The rules of the previous section give a precise compositional semantics for the connectors. However, there are cases where the piping does not apply to any BSP instance B_2 of the second component. In this case, we say that the pipe aborts. We have a similar situation in the *loop* rule, when we are not in the base case and the pipe of the step aborts. To avoid aborts, we can type connectors. For simplicity, we do not consider here the data access and we show a simplified set of rules, which work in the simpler cases. Types are interpreted as sets of BSP instances and are built up from a family of basic types, which depends on the atomic components, using the following operations: [6]

- $\mathscr{B}_1 \times_p \mathscr{B}_2 = \{B_1 \dot{\cup} B_2 \mid B_1 \in \mathscr{B}_1, B_2 \in \mathscr{B}_2\}$, [7] where we represent disjoint union by $1.\mathscr{B}_1 \cup 2.\mathscr{B}_2$.
- $\mathscr{B}_1 + \mathscr{B}_2 = \{1.B_1 \mid B_1 \in \mathscr{B}_1\} \cup \{2.B_2 \mid B_2 \in \mathscr{B}_2\}$
- $\mathscr{B}_1 \cdot \mathscr{B}_2 = \{B_1 \cdot B_2 \mid B_1 \in \mathscr{B}_1, B_2 \in \mathscr{B}_2\}$

Components are built starting from the atomic ones, using the connectors and the compositional semantics of the previous section. A judgement has the form $C : \mathscr{Q} \Rightarrow \mathscr{R}$, where C is a component, $\mathscr{Q}$ a request type, and $\mathscr{R}$ a result type. It means that for every $B(Q,R) \in [C]$, the request part $B(Q)$ belongs to $\mathscr{Q}$ and the result part $B(R)$ belongs $\mathscr{R}$, and for every $I \in \mathscr{Q}$, there is a $B(Q,R) \in [C]$ such that $B(Q) = I$ (= up to reindexing). The rules are given in Fig. 11, where:

$$\frac{C_1 : \mathscr{Q}_1 \Rightarrow \mathscr{R}_1 \quad C_2 : \mathscr{Q}_2 \Rightarrow \mathscr{R}_2}{par(C_1,C_2) : \mathscr{Q}_1 \times_p \mathscr{Q}_2 \Rightarrow \mathscr{R}_1 \times_p \mathscr{R}_2} \qquad \frac{C_1 : \mathscr{Q}_1 \Rightarrow \mathscr{R} \quad C_2 : \mathscr{Q}_2 \Rightarrow \mathscr{R}}{sel(C_1,C_2) : \mathscr{Q}_1 + \mathscr{Q}_2 \Rightarrow \mathscr{R}}$$

$$\frac{C_1 : \mathscr{Q}_1 \Rightarrow \mathscr{R}_1 \times_p \mathscr{R}'_1 \quad C_2 : \mathscr{Q}_2 \times_p \mathscr{Q}'_2 \Rightarrow \mathscr{R}_2 \quad p : \mathscr{R}'_1 | \mathscr{Q}'_2}{pipe(p,C_1,C_2) : \mathscr{Q}_1 \cdot \mathscr{Q}_2 \Rightarrow \mathscr{R}_1 \cdot \mathscr{R}_2} \qquad \frac{C : \mathscr{Q} \Rightarrow \mathscr{R} + \mathscr{E} \quad p : \mathscr{R} | \mathscr{Q}}{while(p1,C) : \mathscr{Q} \Rightarrow \mathscr{E}}$$

$$\frac{C : \mathscr{Q} \Rightarrow \mathscr{R} \quad g : \mathscr{R} \rightarrow \mathscr{R}'}{ra(C,g) : \mathscr{Q} \Rightarrow \mathscr{R}'} \qquad \frac{C : \mathscr{Q} \Rightarrow \mathscr{R} \quad f : \mathscr{Q}' \rightarrow \mathscr{Q}}{la(f,C) : \mathscr{Q}' \Rightarrow \mathscr{R}}$$

Fig. 11. Typing connector composition

- The adaptors f and g are assumed to be chosen in a set of known adaptors such that $P(f(q))$ entails $P(q)$ and $P(r)$ entails $P(g(r))$, where $P(e)$ is the expected property attached to e. We have *problem domain adaptors* and *type adaptors*. The former transform basic types, the latter adapt the structure of the event terms of a BSP instance B, according to the type of B. Examples are $j1 : \mathscr{R} \rightarrow \mathscr{R} + \mathscr{R}'$ defined by $j1(!e) = !1.e$ and $j2 : \mathscr{R}' \rightarrow \mathscr{R} + \mathscr{R}'$ defined by $j2(!e) = !2.e$.

[6] For conciseness, we mix syntax and semantics.

[7] Different kinds of parallel compositors C have different $\times_C$.

- The pipe function $p : \mathscr{R}'_1|\mathscr{Q}'_2$ is such that for every $R' \in \mathscr{R}'_1$, there is $Q' \in \mathscr{Q}'_2$ such that R' pipes to Q'. It is chosen in a set of known pipes for the types $\mathscr{R}'_1, \mathscr{Q}'_2$. Like adaptors, pipe functions are required to maintain the expected properties. We may have problem domain and type pipes. Examples of type pipes are $e1 : !\mathscr{B}+!\mathscr{B}' \to ?\mathscr{B}$, defined by $e1(!1.e) = ?e$ and $is1 : !\mathscr{B}+!\mathscr{B}' \to ?\mathscr{B}+?\mathscr{B}'$, defined by $is1(X) = j1(e1(X))$. Both are partial (only the labels of the form $1.e$ are piped), but $e1$ destructures $1.e$ into e while $is1$ leaves $1.e$ unchanged.
- The while connector is a special case of loop, defined by:

$$while(p,C) =_{def} ra(loop(p1,C),e2)$$

The pipe function $p1$ is defined by $p1(1.r) = p(r)$, while $p1(2.r)$ is undefined. This guarantees that the loop halts when we get a result of the form $2.B$, with $B \in \mathscr{E}$. The adaptor $e2$ eliminates the prefix 2, giving rise to $B \in \mathscr{E}$. There are other specialisation of *loop*, giving rise to components that transform an input stream into an output stream. Here we do not discuss the issue of termination.

Now we discuss the application of typed rules in a hierarchical top-down or bottom-up development. Let $\mathscr{Q} \Rightarrow \mathscr{R}$ be a problem specification and *CR* be a component repository. To get a composed component C solving our problem, we look for a proof tree applying the inference rules, with conclusion $C : \mathscr{Q} \Rightarrow \mathscr{R}$ and assumptions containing components of *CR*. In the bottom-up approach we start by applying the rules to components chosen in *CR*, in such a way that the behaviour of the composites becomes "nearer and nearer" to the wanted one. We stop when we reach it. Here the problem is that it is difficult to establish what is "nearer". On the other hand, in simple cases a brute-force approach could work. The top-down approach is goal oriented. To solve $\mathscr{Q} \Rightarrow \mathscr{R}$ we have to choose one of the rules with a conclusion $C : \mathscr{Q}' \Rightarrow \mathscr{R}'$ matching $\mathscr{Q} \Rightarrow \mathscr{R}$. The premises of the rule contain the sub-problems. We stop the decomposition process when we reach a sub-problem solved by a component of the repository. Rules *sel* and *par* are top-down deterministic, i.e., the types of the premises are uniquely determined by the one of the conclusion. The adaptor and while rules are top-down non deterministic, because the types of the of the premises are determined by those of the consequence and by the non-deterministic choice of the adaptor (f, g) or type pipe (p) function. Finally, the pipe rule is not purely top-down. Indeed, the types of the premises depend also on the components we want to pipe. On the other hand, if we are looking for a pipe, likely we have an idea of the components we want to pipe. We illustrate the above discussion by an example.

Example 5. Let ATM and GA be the atomic components of Example 4 and let us assume that we have to solve the problem specified by

$$ic(c) \to ip(p) \Rightarrow err + acc(num(c), bank(c)) \tag{4}$$

i.e., the user inserts card c and pin p and the component returns either err (operation refused) or $acc(num(c), bank(c))$ (account number and bank of c). The types of ATM and GA are shown below, where we omit ?, ! since on the left hand side of $\Rightarrow$ we have requests, and results on the right hand side:

$$\begin{array}{ll} ATM: & ic(c) \to ip(p) \Rightarrow err + ok(c) \\ GA: & ok(c) \Rightarrow acc(num(c), bank(c)) \end{array}$$

We observe that: the request type of the problem coincides with the one of the ATM component; $ok(c)$ pipes to $ok(c)$ by the identity pipe $I : \mathcal{Q}|\mathcal{Q}$; and the result type of GA entails the one of the problem. Thus it is reasonable to look for the following pipe

$$\frac{ATM : ic(c) \to ip(p) \Rightarrow err + ok(c), \quad C : err + ok(c) \Rightarrow err + acc(num(c), bank(c)), \quad I}{pipe(I, ATM, C) : ic(c) \to ip(p) \Rightarrow err + acc(num(c), bank(c))}$$

where C can be likely obtained from GA by the sel rule, as suggested by the + in its result type. Indeed, our top down process can be continued until we reach the following solution for C:

$$\frac{\dfrac{EP : err \Rightarrow err}{ra(EP, j1) : err \Rightarrow err + acc(num(c), bank(c))} \qquad \dfrac{GA : ok(c) \Rightarrow acc(num(c), bank(c))}{ra(GA, j2) : ok(c) \Rightarrow err + acc(num(c), bank(c))}}{sel(ra(EP, j1), ra(GA, j2)) : err + ok(c) \Rightarrow err + acc(num(c), bank(c))}$$

where $EP : err \Rightarrow err$ is a propagator. With request $?1.err$, the selector sends $?err$ to $ra(EP, j1)$, which returns $j1(!err) = !1.err$. With request $?2.ok(c)$, $?ok(c)$ is sent to $ra(GA, j2)$, which returns $!2.acc(n, b)$ (where $n = num(c)$ and $b = bank(c)$).

As already remarked, our approach may introduce redundancies and many composition levels and this problem is to be solved at the composition environment level. We conclude this section by an example on this issue.

Example 6. Let us assume to use Petri Nets as a EC. We label transitions by request types (to fire, a transition waits for a mark) and places by result types (a place receives a mark). A component is represented by a unique transition with an input place waiting for requests and an output place receiving the results, as shown in Fig. 12. If a

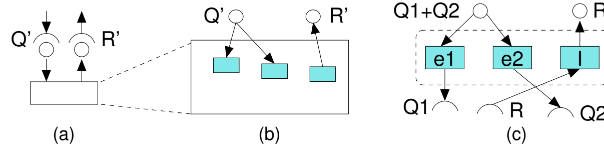


Fig. 12. Petri-net components and connectors

component is not atomic, its place can be exploded into type sub-nets, i.e., we have a hierarchical representation of composites (Fig. 12 (b)). Connectors are net schemas, with plugs for the components they connect. In Fig. 12 we show a plug (a) and the sel connector (b). Composing means plugging the input and output places in the corresponding connectors. Connectors and components are typed. The types of a connector are parametric and a component can be plugged in only it its types match those of the connector. For example, the type $acc(db, c) + err$ matches $Q_1 + Q_2$, where Q_1, Q_2 are type variables.

Fig. 13(a) is the net representation of the proof-tree of (4). In Fig. 13(b) we show a simplification of the net in (a), based on the fact that the pipe function I (for Identity) maps x into x, EP is a propagator, and transitions $e1(1.x) = x$ and $j1(x) = 1.x$ compose into $is1(1.x) = 1.x$ (the latter simply controls that the form of the token is $1....$).

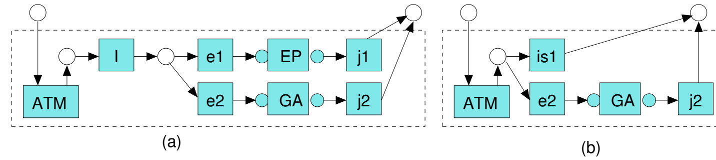


Fig. 13. Petri-net simplification

5 Discussion and Concluding Remarks

The calculus for exogenous composition in Section 4 shows that exogenous composition supports hierarchical system construction. Control encapsulation allows us to consider the request and result types separately. This facilitates both hierarchical decomposition and composition of the functions of the system, while offering the facility for specifying protocols. Types and inference rules for them make the calculus non *ad hoc*. Indeed, we say that exogenous composition is both *functional*, i.e. it can be defined explicitly as a function, and *algebraic*, i.e. it results in a unit of the same type as the sub-units. A composition mechanism that is functional can be fully automated as a composition operator, since it is fully defined. By contrast, a non-functional mechanism can only be applied manually, since it requires glue. A composition mechanism that is algebraic supports hierarchical (recursive) composition, since each composition step yields a construct of the same type. Such mechanisms are most desirable since they can constitute a component algebra [1]. By contrast, a non-algebraic mechanism cannot support hierarchical composition, since each composition step may yield a construct of a different type.

The calculus for software composition in [1] is also based on the π-calculus. However, it is not based on control encapsulation, and the composition mechanism is connection. Units (called *forms*) are linked by scripts via their services. So the composition mechanism is like architectural unit composition, using connection (with glue); it is algebraic but non-functional.

In our calculus, we have chosen to stay at a very abstract level, to decouple the calculus (meant to be used in the design phase) from the composition environment (in the deployment phase) for implementing a design expressed in the calculus. Our study of the latter is only at an initial stage. Some ideas come from [6], where connectors are implemented in the formalism of hierarchical Petri nets [19]. Using Petri nets we do not have a natural representation of streams, which can be treated in the calculus by means of a special kind of loop connector. So we are considering the π-calculus as a candidate for building a formal model of a composition environment. The advantage is that our behaviours fit with a a restricted form of π-calculus. The fact that restricted forms of process calculi give rise to a compositional semantics is not new [20, 2]. An example is the *internal π-calculus* [2], which admits *event structures* [23] as a compositional semantics [2], and has a simpler notion of equivalence than the full π-calculus. Interestingly, the internal π-calculus comes with the idea of formalising internal mobility. Our approach comes with the idea of control encapsulation, which allows us to give a calculus for typed components where types have a "request-result" form $\mathscr{Q} \Rightarrow \mathscr{R}$,

highlighting the functional behaviour of components. Event structures are posets with a conflict relation. In our case, we have a further simplification and we use only posets. We have not yet studied the possibility of extending our approach to event structures.

Our system is Turing complete using *pipe*, *sel* and *while*, and atomic units for *succ* and '=', over natural numbers. This shows that our calculus is powerful but undecidable. In particular, termination is undecidable and must be treated *ad hoc*. Iteration rules that are not *ad hoc* and more controlled can be given, and we could argue that they are sufficient for the purpose of building applications from components at a high granularity level.

Finally, using the calculus will naturally lead to multiple levels of composition. So it is important to be able to optimise such a design by simplifying the composition at a particular level, or even reducing the number of levels. Here, the typed inference rules should be able to offer useful help. For instance, ideas from proof-theory (in particular, cut elimination) would seem to be able to offer strategies for optimising a pipe composition.

Acknowledgements

We wish to thank the reviewers for their detailed and constructive comments that have helped us to improve the paper.

References

1. Achermann, F., Nierstrasz, O.: A calculus for reasoning about software composition. Theoretical Computer Science 331(2-3), 367–396 (2005)
2. Crafa, S., Varacca, D., Yoshida, N.: Compositional event structure semantics for the internal *pi*-calculus. In: Caires, L., Vasconcelos, V.T. (eds.) CONCUR 2007. LNCS, vol. 4703, pp. 317–332. Springer, Heidelberg (2007)
3. Davey, B.A., Priestley, H.A.: Introduction to Lattices and Order, 2nd edn. Cambridge University Press, Cambridge (2002)
4. DeMichiel, L., Keith, M.: Enterprise JavaBeans, Version 3.0. Sun Microsystems (2006)
5. Gelernter, D., Carriero, N.: Coordination languages and their significance. Comm. ACM 35(2), 97–107 (1992)
6. Lau, K.-K., Ntalamagkas, I., Tran, C.: Composite software composition operators using coloured Petri-nets. Technical report, Computer Science, Univ. Manchester (in preparation)
7. Lau, K.-K., Ornaghi, M., Wang, Z.: A software component model and its preliminary formalisation. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 1–21. Springer, Heidelberg (2006)
8. Lau, K.-K., Taweel, F.: Data encapsulation in software components. In: Schmidt, H.W., Crnković, I., Heineman, G.T., Stafford, J.A. (eds.) CBSE 2007. LNCS, vol. 4608, pp. 1–16. Springer, Heidelberg (2007)
9. Lau, K.-K., Velasco Elizondo, P., Wang, Z.: Exogenous connectors for software components. In: Heineman, G.T., Crnković, I., Schmidt, H.W., Stafford, J.A., Szyperski, C., Wallnau, K. (eds.) CBSE 2005. LNCS, vol. 3489, pp. 90–106. Springer, Heidelberg (2005)
10. Lau, K.-K., Wang, Z.: Software component models. IEEE Trans. on Soft. Eng. 33(10), 709–724 (2007)

11. Medvidovic, N., Taylor, R.N.: A classification and comparison framework for software architecture description languages. IEEE Trans. on Soft. Eng. 26(1), 70–93 (2000)
12. Miglioli, P., Moscato, U., Ornaghi, M., Usberti, G.: A constructivism based on classical truth. Notre Dame Journal of Formal Logic 30(1), 67–90 (1989)
13. Milner, R.: A Calculus of Communicating Systems. Springer, Heidelberg (1980)
14. Milner, R.: Communicating and Mobile Systems: the π-Calculus. Cambridge University Press, Cambridge (1999)
15. Ornaghi, M., Benini, M., Ferrari, M., Fiorentini, C., Momigliano, A.: A constructive object oriented modeling language for information systems. ENTCS 153(1), 67–90 (2006)
16. OMG. UML 2.0 Infrastructure Final Adopted Specifcation (2003)
17. Peltz, C.: Web services orchestration and choreography. Computer 36(10), 46–52 (2003)
18. Petri, C.A.: Kommunikation mit Automaten. PhD thesis, University of Bonn (1962)
19. Reisig, W., Rozenberg, G. (eds.): APN 1998. LNCS, vol. 1492. Springer, Heidelberg (1998)
20. Sangiorgi, D.: π-calculus, internal mobility, and agent-passing calculi. Theoretical Computer Science 167(1&2), 235–274 (1996)
21. Schröder, B.S.W.: Ordered Sets: An Introduction. Birkhäuser, Basel (2003)
22. Szyperski, C.: Universe of composition. Software Development (August 2002)
23. Winskel, G., Nielsen, M.: Models for concurrency. In: Handbook of Logic in Computer Science. Semantic Modelling, vol. 4, pp. 1–148. Oxford University Press, Oxford (1995)