Representing Topic Event-Based Systems using Pluggable Units

Fernando J. Barros
Departamento de Engenharia Informática
Universidade de Coimbra, Portugal
Email: barros@dei.uc.pt

***Abstract*—In this paper, we present Pluggable Software Units (PUs), a formalism aimed to represent independent and hierarchical software units. PUs extend the request/reply paradigm by introducing full support to anonymous invocation. PUs is a reflective approach supporting the definition of dynamic software topologies. We show that these features enable the representation of modular topic event-based systems using gate-to-gate (g2g) communication. PUs provide a unification of request/reply and event-based paradigms. Our results contradict current research that suggests event-based and request/reply approaches to be independent and intrinsically incompatible. Application examples are described in JUSE, a Java/Groovy implementation of PUs.**

***Keywords*-topic event-based programming; pluggable software units; dynamic software topologies.**

I. INTRODUCTION

Hierarchical and modular design has its origins in the field of General Systems Theory [1] and it has been later adapted by the area of software engineering [2], [3]. Although the advantages of independent software are evident, the definition of reusable software has been elusive for many decades. Early formal frameworks for defining independent software lack the compliance with object-oriented design, becoming virtually useless for software engineering projects. The area of software architecture has produced specifications that have little support from programming languages [4]. Earlier executable specifications supporting the independence between software components have been introduced in the area of event-based programming [5]. Although the event-based paradigm has many features enabling reuse, it is not compatible with the request/reply principles of object-oriented programming, imposing an exclusive choice between programming paradigms [6]. Given the known advantages of both event and request/reply programming it would be desirable to develop a unifying paradigm exhibiting the best of their features.

We have developed PUs [7], a modular and hierarchical software specification framework, based on the General Systems Theory [8]. The PU approach is fully compatible with the request/reply paradigm introducing the complete independence between software units [9]. This approach the anonymous request/reply paradigm, as defined in [6].

In this paper we unify topic event-based and anonymous request/reply programming. In particular, we express events using PUs *gate-to-gate* (g2g) primitives to achieve a framework supporting both styles of programming. This unification allows software models to combine the best features of both paradigms, giving the choice to the modeler to represent parts of the model using the multicast feature of event programming, simultaneously with gate-to-gate communication provided by request/reply.

To obtain the unification of both paradigms we map event publish/subscribe operators into g2g links supported by PUs. Since publish/subscribe operators can be made during application runtime, the key to the unification is given by the ability to support dynamic software topologies that adapts links to these operators.

We shown that design patterns based on implicit invocation, like the Observer pattern [10], can also be represented in PUs, showing the generality of this approach. Application examples are provided in JUSE, a Java/Groovy implementation of PUs.

The paper is organized as follows. Section II provides a formal definition of basic and network pluggable software units (PUs). Section III introduces a representation of topic event-based programming using PUs with a dynamic topology. Related work is described in Section IV. Conclusion and future work are presented in Section V.

II. PLUGGABLE SOFTWARE UNITS

PUs comprises two types of software units: basic and network. Basic PUs provide the actual method invocation, whereas networks are a composition of PUs and provide message passing. In PU composition, both basic and network PUs can be used indistinctly. PUs supports a hierarchical and modular type of software construction. Network definition is dynamic, permitting the specification of adaptive software topologies.

A. Basic PU

Basic PUs define a set of input and output gates. Input gates correspond to object methods, whereas output gates represent an abstract access to external PUs. Output gates are an amendment to the object-oriented protocol and they remove the need for PUs to refer to others explicitly. This construct supports effectively the anonymous request/reply programming. Since PU communication is made exclusively through gates they are completely independent and can be arbitrarily composed. Each basic PU has its own description, referred to as the *PU model*. Let $\widehat{B}$ be the set of names of basic PUS. The PU model associated with $\chi \in \widehat{B}$ is given by:

$$M_\chi = \Big(inGates, \{inSign_g\}, S, s_0, \{a_g\}, outGates,$$
$$\{outSign_k\}, \{outDSign_k\}, \{outFunction_k\}\Big)_\chi$$

where

$inGates$ is the set of PU input gates
$inSign_g$ is the input-to-output signature of every gate g in $inGates$
S is the set of PU states
s_0 is the PU initial state
a_g is an action for every gate g belonging to the set $inGates$

$outGates$ is the set of PU output gates
$outSign_k$ is the output-to-input signature of every gate k in $outGates$
$outDSign_k$ is the intermediate signature of every output gate $k \in outGates$
$outFunction_k$ is the output function of every gate k in $outGates$

An input-to-output signature is a 2-tuple containing the range set of the incoming parameters and the range set of outgoing parameters. For example, if input gate g receives real values **R**, and responds by sending integer values **I**, then its input signature is given by $inSign_g = (\mathbf{R}, \mathbf{I})$.

The function a_g on input gate g of signature (I_g, O_g) is expressed by

$$a_g : S \times I_g \rightsquigarrow S \times O_g$$

An *action* corresponds to a method in the object paradigm. Action a_g receives input values from $(S \times I_g)$, produces a change in the PU state, and returns a value from O_g. As a side effect, an action on a PU can trigger other actions on the PUs linked to it. The action can also request values from the network where the PU is inserted. We do not formalize here these *side effects* of action behavior.

An output-to-input signature is a 2-tuple (O_k, I_k) containing the range set of the outgoing (direct) parameters O_k and the range set of incoming (return) parameters I_k.

Output functions convert the set of values received by an output gate. These functions are useful when several links are connected to an output gate, and in general, to convert values without creating special PUs.

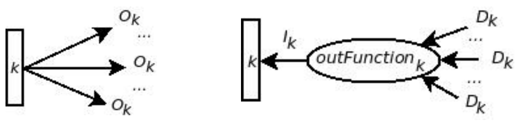
Intermediate signatures define the values, D_k, that can be received by an output gate k. These value are then converted by the output function to the set I_k.

The output function $outFunction_k$ on output gate k of intermediate signature D_k and output signature (O_k, I_k) is expressed by

$$outFunction_k : {D_k}^* \longrightarrow I_k$$

where ${D_k}^*$ is a list of values from set D_k.

Semantics of the output function $outFunction_k$ associated with gate k is graphically sketched in Figure 1, where request and reply semantics are represented in Figure 1(a) and Figure 1(b), respectively.



(a) Request. (b) Reply.

Fig. 1. Semantics of the $outFunction_k$.

In the request phase (Figure 1(a)), values from set O_k are sent to all neighbors of gate k. In the reply phase (Figure 1(b)), values from set D_k are collected and transformed by $outFunction_k$ into a value of the set I_k.

We note that input gates of basic PUs do not define intermediate signatures since these units do not have internal connections, and thus their input gates are terminal.

Given an output gate k with output signature (O_k, I_k), we assume the *head* function when the output function $outFunction_k$ is omitted and the intermediate and input signatures match ($D_k = I_k$). This function returns the first value from a list and it is defined by:

$$head(< arg_0, \ldots >) = arg_0$$

Example: `Position` *PU:* To illustrate an example of a basic PU we employ the `Position` PU represented in Figure 2. This PU has input gates: `ax` and `x`, corresponding to actions it can provide. `Position` has also the output gate `x` that sends the current position to the outside. `Position` receives piecewise constant acceleration values and computes the current position `x` by double integrating the input signal. For simplicity we describe here one-dimension positions. 2D coordinates are used in the next sections.

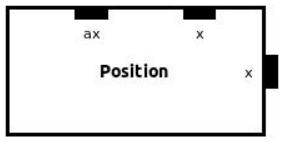


Fig. 2. `Position` PU.

`Position` state keeps the time of the last update ($time$), position (x), velocity (v_x) and acceleration (a_x) values. The PU is described by:

$$\begin{aligned} M_{\texttt{Position}} = (&\{\texttt{ax}, \texttt{x}\}, \\ &\{(\mathbf{R}^2, \emptyset), (\mathbf{R}, \mathbf{R})\}, \\ &\mathbf{R}^3, (time = 0, x = 0, a_x = 0), \\ &\{action_{\texttt{ax}}, action_{\texttt{x}}\}, \\ &\{\texttt{x}\}, \{(\mathbf{R}, \emptyset)\}, \{\emptyset\}, \\ &\{outFunction_{\texttt{x}}(\varnothing, ...) = \varnothing\}) \end{aligned}$$

where $\emptyset$ represents the empty set and $\varnothing$ represents the null/absence of value.

The `ax` action sets the acceleration and is defined by:

```
$action_{\texttt{ax}}(t, a)$
    $\delta \leftarrow t - time$
    $x \leftarrow x + v_x\delta + \frac{a_x}{2}\delta^2$
    $v_x \leftarrow v_x + a_x\delta$
    $a_x \leftarrow a$
    $time \leftarrow t$
    $out.\texttt{x}(x)$
    $\uparrow \varnothing$
```

This action also sends the current position to the outside through gate `x` using the command `out.x(`x`)`. The current position at time *t* is computed by:

```
$action_{\texttt{x}}(t)$
    $\delta \leftarrow t - time$
    $\uparrow x + v_x\delta + \frac{a_x}{2}\delta^2$
```

B. PU Network

Hierarchical composition of systems has been used as a powerful construct to manage complex systems. We consider that PUs can be hierarchically composed, being the resultant PU indistinguishable from the basic PU of the last section. This ability permits to handle in a homogeneous form both basic and aggregated components. A PU network is a complex PU built by the composition of other PUs. Let $\widehat{E}$ be the set of names corresponding to PU networks, constrained to $\widehat{E} \cap \widehat{B} = \emptyset$. The model of the network PU $\chi \in \widehat{E}$ is defined by:

$$M_\chi = \Big(inGates, \{inSign_g\}, \{inDSign_g\}, \{inFunction_g\}, \varepsilon, M_\varepsilon, outGates, \{outSign_k\}, \{outDSign_k\}, \{outFunction_k\}\Big)_\chi$$

where

$inGates$ is the set of the network input gates
$inSign_g$ is the input-to-output signature of every gate $g \in inGates$
$inDSign_g$ is the intermediate signature of every input gate $g \in inGates$
$inFunction_g$ is the input function of every gate $g \in inGates$

$\varepsilon \in \widehat{\varepsilon}$ is the network executive
M_ε is the model of the network executive
$outGates$ is the set of the network output gates
$outSign_k$ is the output-to-input signature of every gate $k \in outGates$
$outDSign_k$ is the intermediate signature of every output gate $k \in outGates$
$outFunction_k$ is the output function of every gate $k \in outGates$

with $\widehat{\varepsilon}$ representing the set of all names associated with network executives, constrained to $\widehat{\varepsilon} \cap \widehat{B} = \widehat{\varepsilon} \cap \widehat{E} = \emptyset$.

The PU network has the same type of interface of a basic PU making it possible to use networks as components of other networks, enabling the hierarchical composition of PUs. The network topology is managed by a special PU termed here by *network executive* ϵ. The executive keeps a list of the PUs that compose the network. It also keeps the set of the links existing among PUs. This information is not static, and can be changed by executive actions [11]. The model of the network executive is an augmented PU model defined by:

$$M_{\varepsilon_\chi} = \Big(inGates, \{inSign_g\}, S, s_0, \{a_g\}, \sigma, \widehat{\Sigma}, outGates, \{outSign_k\}, \{outDSign_k\}, \{outFunction_k\}\Big)_{\varepsilon_\chi}$$

Function σ maps the executive state into an network topology. The *topology function* σ is expressed by:

$$\sigma : S \rightarrow \widehat{\Sigma}$$

Each topology $\Sigma \in \widehat{\Sigma}$ is given by

$$\Sigma = \Big(C, \{M_c\}, L, \Xi\Big)$$

where

C is the set of PUs
M_c is the model of each PU $c \in C$
L is the set of links
Ξ is the order function

Given that the current network topology is a function of the executive state, any change in this state can cause a topological change in the network. A link in L is a 3-tuple defined by:

$$\Big((i, g_i), (j, g_j), (dC, rC)\Big)$$

where

i is the name of the source PU
g_i is a gate of the i PU
j is the receiver PU
g_j is a gate of j
dC is the link direct converter
rC is the link reverse converter

Converters transform both the values sent and received by a PU. For example, if a PU works with values in m·s^{-1} and needs to communicate with another PU operating in km·h^{-1}, then adapting capabilities provide a solution to make this conversion without the creation of additional PUs. In this case, the direct converter is given by $dC(x) = 3.6\ x$, and the reverse converter is given by $rC(x) = \frac{x}{3.6}$ to make the conversions m·s^{-1} $\leftrightarrow$ km·h^{-1}. If omitted, converters are considered to be the identity function. We note that reverse converters are a consequence of the request/reply paradigm that imposes the compatibility of the returned values.

$\Xi : L^+ \rightarrow L^+$ is the *order function*, where L^+ is the set of all sets of links (excluding the empty set).

The order function establishes the order of the outside calls when several links are connected at the same output gate. For simplicity, when omitted, a non-deterministic order is assumed. In the JUSE implementation the order function is established implicitly by link declaration order.

The initial network topologyΣ_0 is given by $\Sigma_0 = \sigma(s_{0,\epsilon})$.

Example: Mobile Entity: To illustrate the definition of a PU network, we build a PU to represent an autonomous mobile based on the `Position` PU of last Section. The network PU is depicted in Figure 3 and it is defined by:

$$\begin{aligned} M_{\texttt{Mobile}} =& (\{\texttt{x}\}, \{(\emptyset, \mathbf{R})\}, \varepsilon_{\texttt{Mobile}}, \\ & M_{\varepsilon_{\texttt{Mobile}}}, \{\texttt{x}\}, \{(\emptyset, \mathbf{R})\}, \{\emptyset\}, \\ & \{outFunction_{\texttt{x}}(\varnothing, ...) = \varnothing\}) \end{aligned}$$

where

$$\begin{aligned} M_{\varepsilon_{\texttt{Mobile}}} =& (\{\}, \{\}, \{s_0\}, s_0, \{\}, \sigma, \{\Sigma_0\}, \\ & \{\}, \{\}, \{\}, \{\}) \end{aligned}$$

Network single topology given by:

$$\Sigma_0 = \sigma(s_0) = (C, \{M_c\}, L)$$

where

$C = \{\texttt{Position}\}$
$\{M_c\} = \{M_{\texttt{Position}}\}$
$L = \{$
$((\texttt{Mobile}, \texttt{x}), (\varepsilon_{\texttt{Mobile}}, \texttt{x}), dC(t) = t, rC(x) = x)$
$((\varepsilon_{\texttt{Mobile}}, \texttt{ax}), (\texttt{Position}, \texttt{ax}), dC(t, a) = (t, a), rC(\varnothing) = \varnothing),$
$((\varepsilon_{\texttt{Mobile}}, \texttt{x}), (\texttt{Position}, \texttt{x}), dC(t) = t, rC(x) = x),$
$((\texttt{Position}, \texttt{x}), (\texttt{Mobile}, \texttt{x}), dC(x) = x, rC(\varnothing) = \varnothing)$
$\}$

The PU network, represented in Figure 3, is composed of one `Position` PU, linked to the executive $\varepsilon_{\texttt{Mobile}}$. The network has the input gate `x` to access the value of the current position. The executive requests the position through the call `x(time)`, where `time` represents the current time.

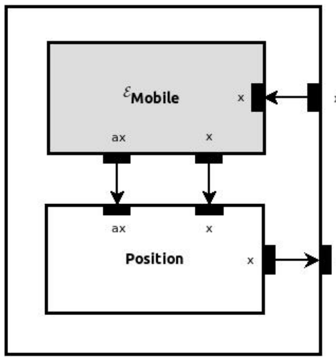


Fig. 3. Block diagram of the `Mobile` network PU.

At a random intervals, the executive updates the current value of acceleration through gate `ax`. This value is integrated by PU `Position` that computes current position and velocity as described in the last section.

C. JUSE

JUSE is a Java/Groovy implementation of PUs and it provides an executable version of software units. JUSE supports the following calls to create software units and to establish links between PUs:

`void addS(Class aClass, String aName)`, creates a PU named `aName` of class `aClass`;

`void linkS(String aName, String aGate, String bName, String bGate, Closure dConverter, Closure rConverter)`, links a PU named `aName` gate `aGate` to gate `bGate` of PU `bName`, establishing direct converter `dConverter` and reverse converter `rConverter`.

Input/output functions are associated with input/output gates, respectively. In JUSE these functions are specified when gates are added to PUs by the call:

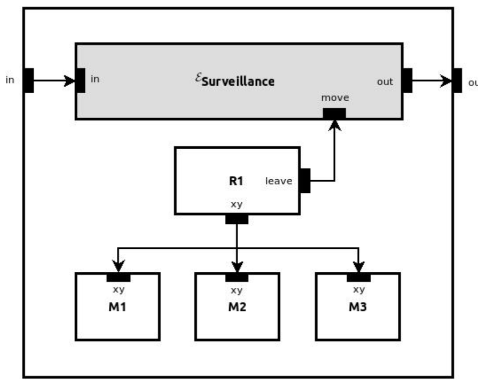
`GateCollection add(Class rSgnt, Class iSgnt, String aGate, ArrayList<Class> dSgnt, Closure aClosure)`, that adds `aGate` with return signature `rSgnt`, intermediate signature `iSgnt` and direct signature `dSgnt`, and associates it with the function `aClosure`.

These primitives are used in the next sections for describing several examples using JUSE. Anonymous Publish/Subscribe JUSE provides support for anonymous publish/subscribe programming enabling both static and dynamic topologies. We consider the surveillance system depicted in Figure 4, with one radar and a variable number of mobiles (ships, aircrafts, ...), that enter and leave radar range. The radar samples the position of all mobiles at a regular rate and receives a list of pairs with mobile position and name. Instead of modeling mobiles as entities that publish their position, as required by push event-based programming, we have considered that mobile positions are pulled by the radar. The main advantage is that we can directly express radar sampling rate instead of handling an arbitrary pushing rate from the mobiles. This representation makes the surveillance model more efficient since it can easily accommodate several radars with different sampling rates. When the radar issues the command `xy` it receives a list of mobile positions at the current time. This example exploits the bidirectional nature of request/reply that unifies the push and pull styles of event-based programming. These two styles can be used independently in some systems like CORBA [12]. To override the default behavior of radar output gate `xy`, we use the following output function that return a list of values:

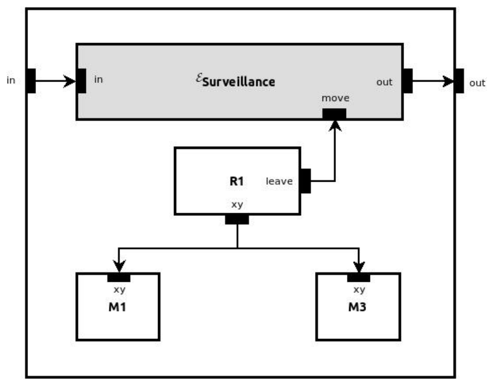
`add(List, XY, 'xy', [], List<XY> list-> list)`

Surveillance initial topology is given by Listing 1 where the name of mobiles is introduced by reverse converters using the method `setSource` in lines 9-11.

```
void structure() {                                                       1
   super.structure();                                                    2
   addS('R1', Radar);                                                    3
   addS('M1', Mobile);                                                   4
   addS('M2', Mobile);                                                   5
   addS('M3', Mobile);                                                   6
   linkS('Network', 'in', 'Executive', 'in', {List<IOutput> m->          7
         [m]}, {Void x-> x});
   linkS('Executive', 'leave', 'Network', 'move', {List<String>s ->      8
         [s]}, {Void x-> x});
   linkS('R1', 'xy', 'M1', 'xy', {->}, {XY p-> p.setSource('M1')});      9
   linkS('R1', 'xy', 'M1', 'xy', {->}, {XY p-> p.setSource('M2')});      10
```



(a) Initial topology.



(b) Topology after the removal of mobile `M2`.

Fig. 4. Block diagram of the surveillance network.

```
    linkS('R1', 'xy', 'M1', 'xy', {->}, {XY p-> p.setSource('M3')});
}
```

Listing 1. JUSE definition of the surveillance initial topology

One of the radar tasks is to keep track of the mobiles within its range. Mobiles out of range are removed and sent to another surveillance system through output gate `out`. These operations are described in Listing 2 where executive action `move` receives a list of mobiles to be removed from the current surveillance system.

```
Void move(List<String>mobiles) {
    List<IOutput> leaving = new List<IOutput>();
    mobiles.each{String m-> leaving.add(remove(m)};
    out.out(leaving);
    return null;
}
```

Listing 2. Removing mobiles

III. TOPIC EVENT-BASED PROGRAMMING

We consider event-based programming as a special case of anonymous publish/subscribe systems with a dynamic topology. This interpretation implies that the same type of invocation is used, the difference being the way dynamic topologies are specified. The event invocation is less expressive than anonymous publish/subscribe due to unidirectional information flow. The publish/subscribe mechanism characteristic of event programming is kept since it enables simpler specifications.

A. Basic Operators

A key feature introduced in publish/subscribe systems, and not supported by many modular systems, is the ability to define changes in topology. In fact, publish/subscribe operators can be regarded as implicitly defining dynamic topologies that link publishers and subscribers. However, in a reflective framework like PUs, that provides full support for dynamic topologies, the ability to change links between components can be easily supported. While PUs topology is kept and controlled by the executive, this does not mean that changes in topology must be decided in the executive. In fact, decisions can be made anywhere in the system, but they are only effective when they are enforced by the executive. To provide similar operators as defined by event-based systems we consider the executive to act as the implicit hidden middleware supporting events. The result is an explicit construct where publish/subscribe messages can be handled. For this purpose, we provide each PU with the output gate `command`. The executive is provided with the input gate `command`, that receives a command and the origin of the command. To support hierarchical event-based systems, we extend PU network with the output gate `command`, so publish/subscribe commands can be sent to upper levels of the hierarchy.

JUSE supports the following executive commands to represent publish/subscribe systems:

`Void publish(String aName, String aGate, Closure dConverter, Closure rConverter)`; where PU named `aName` publishes output gate `aGate`, with direct converter `dConverter` and reverse converter `rConverter`;

`Void subscribe(String aName, String aGate)`; where PU named `aName` subscribes gate `aGate` and will receive notifications at gate `aGate`.

These operators establish g2g channels whenever there is an intersection between the interest of publishers and subscribers. The definition of hierarchical event-based systems is enabled by the possibility to send/receive messages to/from the network PU.

JUSE supports the following executive commands to destroy links in publish/subscribe systems:

`Void unpublish(String aName, String aGate)`;

`Void unsubscribe(String aName, String aGate)`.

Although this can be seen as an exercise of expressing one paradigm into another, this mapping combines advantages of both techniques. The unifying approach allows multicast and g2g topologies to co-exist. If we take the example of the last section, while the links between the radar and the mobiles can be easily expressed using publish/subscribe operators, a pursuer launched to cancel a specific mobile will be better specified using g2g operators. In every case the specification ends up becoming g2g, but when using publish/subscribe operators links are defined implicitly, providing a short notation that in many cases becomes more convenient. The addition of a new radar, for example, becomes quite simple, since the radar needs only to publish its output gate `xy`.

The new representation of the Mobile PU presented in Section II-B is given in Figure 5. Default gate `command` and g2g links are provided to each PU so commands can be transmitted. While the executive has the input gate `command` linked to the output gate `command` of all other PUs, executive output gate `command` is linked to network output gate `cmd` so commands can be sent hierarchically to the upper level, enabling dynamic scoping [6].

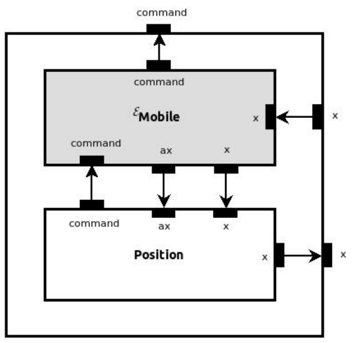


Fig. 5. Mobile PU supporting publish/subscribe operators.

A major contribution of the publish/subscribe paradigm is a set of compact operators to express structural changes. Gate-to-gate links require source and destination information in order to establish a new channel. In some situations this information is cumbersome to gather and publish/subscribe operators can represent models in a simpler manner. However, since the resulting topologies can still be expressed by g2g networks, no increasing expressiveness is actually obtained. We next provide an example that demonstrates the advantages of the publish/subscribe operators.

B. Event-Based Surveillance System

We consider the event-based representation of the surveillance system described in Section 3 and depicted in Figure 6. After the commands of Listing 3, the topology is changed from the initial structure given by Figure 6(a) to the topology of Figure 6(b).

```
out.command({String s, Executive e-> e.publish(s, 'xy', {->},        1
      {String source, XY p-> p.setSource(source)})}) "In radar R1"
out.command({String s, Executive e-> e.subscribe(s, 'xy')}) "In      2
      Mobile M1"
out.command({String s, Executive e-> e.subscribe(s, 'xy')}) "In      3
      Mobile M2"
```

Listing 3. JUSE publish/subscribe commands to create the topology in Figure 6(b).

The commands originated at mobiles `M1` and `M2` are issued by the respective executive and sent to the Surveillance executive through gates `command`. Line 1 publishes `R1` output gate `xy`. Line 2 subscribes to all output gates `xy` that have been publishes and tries to create g2g channels to `M1` input gate `xy`. Line 3 is similar and it applies to mobile `M2`. Mobile `M3` did not subscribe any gate and it becomes stealthy since it cannot be detected by any radar. The advantages of this approach are self-evident. The introduction of new radars and mobiles becomes very simple since the event-based executive handles publications and subscriptions, freeing the modeler from specifying g2g links. These links can be cumbersome to establish as shown in this particular system.

The support in JUSE for systems requiring a hybrid specification using events and request/reply can be exemplified by the creation of a pursuer launched to cancel a specific threat. Upon detection, the radar sends a signal to the executive to request the creation of a pursuer and to make g2g links between the pursuer and a specific mobile identified by the radar. Figure 7 depicts the surveillance system with pursuer `P1` attached to mobile `M1`. The radar and executive PUs are extended with gate `pursuer` so the radar can make requests for purser creation.

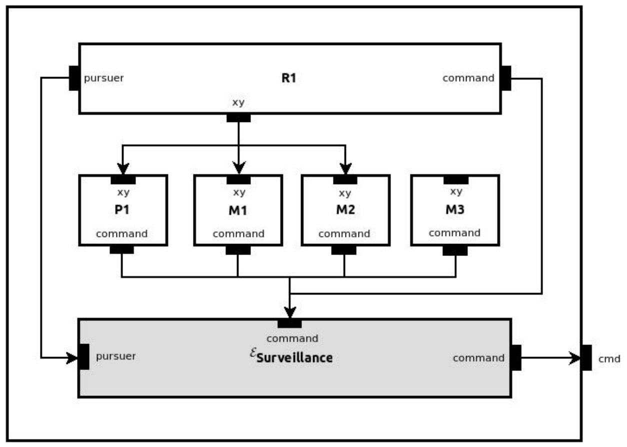


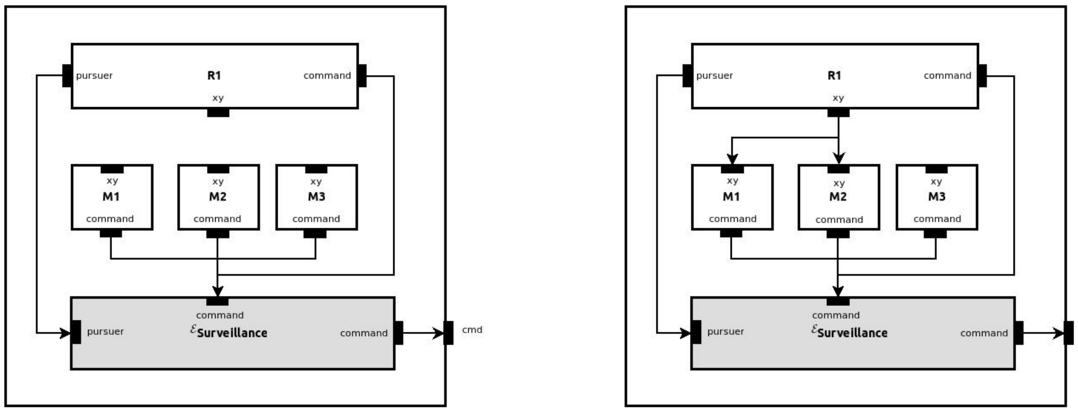
Fig. 7. Surveillance system with purser `P1` targeting mobile `M1`.

JUSE executive action `pursuer` is given in Listing 4. The executive finds the nearest purser to position aXY (Line 3), adds the purser (Line 4) and links it to the mobile aName (line 5). These links are specified using g2g operations and they would become difficult to be specified using publish/subscribe operations.

```
Void pursuer(String aName, Point aXY) {                 1
   pursuer = findPursuerAt(aXY);                        2
   add(pursuer);                                        3
   link(pursuer _name, 'xy', aName 'xy');               4
}                                                       5
```

Listing 4. Creation of pursuer `P1` using g2g specification.

This solution requires the ability to program the executive that no longer can be an implicit and hidden middleware layer and needs to be made visible and reprogramable. A solution in the event paradigm would require the use of content event-based programming and the pursuer would only receive messages from a specific mobile. This solution, however, would



(a) Initial topology.

(b) Topology after the radar `R1` has published output gate `xy` and `M1-M2` have subscribed gate `xy`.

Fig. 6. Event-based surveillance system.

not be so elegant and mostly no so efficient since it would require message conversion from a possibly large number of mobiles. This solution would also require to disclose the mobile name to the pursuer, breaking the modularity of the approach.

C. Event-Based vs. Request/Reply

The multicast nature of events poses no difficulty in being mapped into gate-to-gate communication. In the example of the previous section, when a new mobile is added to the surveillance system and subscribes gate `xy` it is immediately linked to all radars in the systems, without the need to explicitly find what are the current set of active radars, since the executive can retrieve this information and use it to make the required links. Analogously, when a new radar is created and publishes gate `xy` it starts receiving data from all mobiles, since the executive keeps tracks of all PUs having published gate `xy`. The complementary situation is treated in a similar form: when a mobile leaves the system by unsubscribing gate `xy`, all radars stop receiving mobile position. Event-based systems have been considered orthogonal to anonymous request/reply programming [6]. This seems to be the case for some request/reply systems, but it does not hold for PUs, as shown here.

Event-based programming, through publish/subscribe constructs, has introduced a set of operators that provide a compact specification for some types of changes in software topology. However, these operators can be mapped into g2g links defined in PUs. This situation is quite fortunate since it allows the expression of event-based programming with modular software units. Events can thus be integrated with anonymous request/reply instead of being an additional form of software specification.

The difficulty in integrating event-based and request/reply approaches seems to be caused by the requirements event systems impose on software topologies. As shown in the examples, anonymous invocation is just one of the key factors to integration, the other is the ability to modify software topology during runtime. Anonymous requests/reply solutions not exhibiting both features will thus provide limited support for event-based systems.

We consider that a unifying effort is currently required given the multiplicity and apparent disparate paradigms proposed for software development. This situation forces practitioners to master a large variety of paradigms, or in alternative, to map models to a known paradigm that does not yield be best representation. Since reality is, for complex systems, multi-faceted, a unifying approach, like PUs, permits to choose the best paradigm to each aspect of the system while guaranteeing the overall integration.

IV. Related Work

Hierarchical and modular principles have been used as a powerful heuristic for handling complex problems in many fields. One of the first formal descriptions of modular decomposition have been made in the area of General Systems Theory [1]. The decomposition of software into modules has later been advocated in software engineering [3]. On this latter work, however, the hierarchical decomposition of software has not been really introduced but rather hierarchy is used as synonymous of layered (software). Recently, there has been a growing interest in modular representations and large variety of formalisms have been modified/created [2], [13]–[15]. Likewise General Systems Theory representations, these formalisms are not compliant with request/reply principles, imposing awkward software specifications. Given these limitations, many formal models become virtually useless for practical use in software projects.

To overcome the limitations of formal descriptions, so called Architecture Definition Languages (ADLs) have been developed [4], [16], [17]. However, ADLs are mainly fa ades decoupling specification from implementation as pointed in [18]. ADLs description need thus to be translated into a programming language. This process is somewhat similar to the one used by the Unified Modeling Language [19] with the limitations and drawbacks of separating specification from implementation.

To bridge this gap, hierarchical and modular constructs have been introduced into existing programming languages [18], [20], [21]. However, none of theses approaches provide the general support to modular hierarchical software as provided by PUs. Limitations include the lack of converters and input/output functions. Additionally, these systems do not provide full support for topology adaptation. In particular, they lack the ability to represent hierarchical mobility, as supported by PUs [7].

The use of event-based programming has also been advocated as an alternative to anonymous request/reply [22], (the reverse we have described in this paper). However, the advantages are only apparent since signal wiring diagrams [22] need to be manually mapped into object-oriented request/reply languages becoming, in some aspects, similar to ADLs.

V. Conclusion and Future Work

PUs provide a powerful framework for developing reusable software units. This approach supports hierarchical and modular software development, permitting to handle complexity by partitioning large models into smaller and independent units. The introduction of converters and input/output functions provide a great flexibility to software interconnection. PUs also provide full support for dynamic software topologies. In particular, we have shown that the ability to add and remove software channels at runtime permits to describe the topic event-based programming style using gate-to-gate connections supported by PUs. This work has demonstrated that topic event-based programming can be regarded as a particular case of anonymous request/reply. Callbacks, used in the Observer pattern and the Composite pattern, were also shown to be particular cases of an anonymous request/reply representation supporting converters and input/output functions. As future work we intend to study the requirements for supporting content event-based programming and to introduce the required operators to represent this paradigm in PUs. The representation of event-based scoping exploiting hierarchical modeling also looks promising.

Acknowledgment

This work was supported by the Portuguese Foundation for Science and Technology under project PTDC/EIA-EIA/100-752/2008.

References

[1] A. Wymore, *A Mathematical Theory of Systems Engineering: The Elements*. Krieger, 1967.

[2] R. Allen and D. Garlan, “A formal basis for architectural connection,” *ACM Transactions on Software Engineering and Methodology*, vol. 6, no. 3, pp. 213–249, March 1997.

[3] D. Batory and S. O’Malley, “The design and implementation of hierarchical software systems with reusable components,” *ACM Transactions on Software Engineering and Methodology*, vol. 1, no. 4, pp. 355–398, 1992.

[4] D. Garlan, R. Monroe, and D. Wile, “ACME: An architecture description interchange language,” in *Conference of the Centre for Advanced Studies on Collaborative Research*, 1997.

[5] D. Luckham and J. Vera, “An event-based architecture definition language,” *IEEE Transactions on Software Engineering*, vol. 21, no. 9, pp. 70–93, 1996.

[6] G. Mühl, L. Fiege, and P. Pietzuch, *Distributed Event Based Systems*. Springer, 2006.

[7] F. Barros, “System and method for programming using independent and reusable software units,” US Patent 6851104 B1, February 2005.

[8] ——, “Modeling formalisms for dynamic structure systems,” *ACM Transactions on Modeling and Computer Simulation*, vol. 7, no. 12, pp. 505–515, 1997.

[9] ——, “Achieving reuse with pluggable software units,” in *12th International Conference on Software Reuse: Top Productivity through Software Reuse*. Lecture Notes in Computer Science, Volume 6727, 2011, pp. 183–191.

[10] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[11] F. Barros, “Representing hierarchical mobility in software architectures,” in *International Workshop on Software Engineering for Adaptive and Self-Managing Systems*, 2007.

[12] OMG, *CORBA Component Model Specification*, 2006.

[13] J. Bradbury, “Organizing definitions and formalisms for dynamic software architectures,” Queens University, Canada, Tech. Rep. 2004-77, 2004.

[14] F. Oquendo, “Formally modelling software architectures with the UML 2.0 profile for π-ADL,” *ACM SIGSOFT Software Engineering Notes*, vol. 31, no. 1, pp. 1–13, 2006.

[15] F. Arbab, “Reo: A channel-based coordination model for component composition,” *Mathematical Structures in Computer Science*, vol. 14, pp. 329–366, 2004.

[16] N. Medvidovic and R. Taylor, “A classification and comparison framework for software architecture description languages,” *IEEE Transactions on Software Engineering*, vol. 26, no. 1, pp. 70–93, 2000.

[17] M. Shaw and P. Clements, “The golden age of software architectures: A comprehensive survey,” Carnegie-Mellon University, USA, Tech. Rep. CMU-ISRI-06-101, 2006.

[18] J. Aldrich, C. Chambers, and D. Notkin, “ArchJava: Connecting software architecture to implementation,” in *International Conference on Software Engineering*, 2002, pp. 187–197.

[19] J. Arlow and I. Neustadt, *UML 2 and the Unified Process: Practical Object-Oriented Analysis and Design*. Addison, 2005.

[20] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J. Stefani, “The FRACTAL component model and its support in Java,” *Software Practice and Experience*, vol. 36, no. 11–12, pp. 1257–1284, 2006.

[21] V. Sreedhar, “Mixinup components,” in *International Conference on Software Engineering*, 2002, pp. 198–207.

[22] T. Faison, *Event-Based Programming: Taking Events to the Limit*. Apress, 2006.