A pattern-like framework to ease dynamical change of components behaviour

Daniela Micucci  
D.I.S.Co. - Università Studi Milano-Bicocca  
Via Bicocca degli Arcimboldi 8, 20126 Milano, Italy  
daniela.micucci@disco.unimib.it

Andrea Trentini  
D.I.Co. - Università Studi Milano  
Via Comelico 39, 20135 Milano, Italy  
trentini@dico.unimi.it

Abstract

RTP (Real-Time Performers) is a software architecture capturing reflection mechanisms to support the dynamics of software real-time systems. Dynamics may comprise both behavioural and temporal issues. RTP relies on a strict separation of concerns between computational entities, alignment entities, and strategies activating both computational and alignments activities. This paper focuses on one of the features of RTP architecture: the dynamical change of components behaviour. The RTP component is based on a reflective framework supporting run-time changes inspired by the Strategy Pattern and state machine concepts.

Keywords: Reflection, software architecture, component-connector, state machine, design patterns, framework

1. Introduction

It is highly probable that a long-time running application must adapt itself to situations that may occur without previous planning. To enforce these adaptations, the application should be able to dynamically support changes on its components behaviour, its overall behaviour (in terms of both timing issues and strategies), and its topology. In other terms, the application should be able to change its structure to reflect these changes.

Unfortunately the knowledge of the overall structure of the system is often implicit in the running system itself, i.e., many architectural issues are realized by hard-coded mechanisms or hidden in the middleware and in the operating system.

Approaches like MILs (Module Interconnection Languages) and ADLs (Architectural Description Languages) do not solve the problem since in both approaches architectural choices are lost inside the code: MILs cannot express them in high level notations, and ADLs adopt notations describing concepts that require conventional programming-in-the-small implementations [2].

Both computational and architectural reflection are the starting points to build dynamic systems. Computational reflection is defined as the activity performed by an agent when doing computations about itself [5]. While Architectural reflection [1] is the computation performed by a system about its own software architecture. It reifies architectural features as meta-objects which can be observed and controlled at runtime. The application of architectural reflection helps bringing visibility over the computation performed by the overall system components at the programming level.

Reflection principles have guided the modelling of RTP (Real-Time Performers) [8]: a software architecture for the design of strongly modular, distributed, and real-time system. RTP exploits reflection mechanisms in order to raise at the application programming level strategies definition (action choice on behalf of events), timing issues management (speed-up/slow-down tuning), component behaviours definition (adding/removing performable commands), and system topology definition (adding/removing connected components).

RTP is based on the following key considerations:

- a system is made up by computational components (Performers);
- computational components exchange information via alignment components (Projectors);
- both computational and alignment components are activated and controlled by supervising components (Strategists).

The identification of these well-distinguished components emphasizes the “separation of concerns” between information processing, information alignment, and their activation.

This article focuses on a particular feature of RTP: its support to the dynamical change of components behaviour.
2. RTP architecture overview

The ideas behind the architecture arise from previous research activities. Works like HyperReal [9], Kaleidoscope [10], and RAID [6], have strongly influenced the design of RTP. HyperReal focuses on timing issues, defining a controller that relies on a time-driven model of control, and separates the definition of plans from the dispatching of actions they define. Special entities named Virtual Clock support the explicit management of time. Kaleidoscope is a software architecture designed for monitoring and control systems. It relies upon a general mechanism for the definition of software components and their composition, and maintains at the application level the definition of the policies controlling information exchange. Finally, RAID is a research project whose aim was the design, the development, and the release of a wireless indoor environmental monitoring system.

Kaleidoscope architecture and HyperReal principles have been exploited in the RAID project. The results achieved have lead to define RTP architecture capturing and formalizing the ideas from both HyperReal and Kaleidoscope. RTP may be considered as a reference architecture for those systems in which the observation and the control of the temporal overall system behaviour is a key issue.

RTP architecture individuates three well distinct roles inside a system: computing data, distributing information, and activating both computation and distribution. RTP assigns these roles to three specific components: Performer, Projector, and Strategist respectively. Performer is the entity expressly designed to perform elaboration on its own data, Projector is the entity expressly designed to project (distribute) data between Performers, and Strategist is the entity expressly designed to devise plans toward goals like the activation of Performer computations and Projector alignments.

RTP principles have been translated into a framework developed adopting the Java Programming Language. To further test the validity of RTP ideas, this framework is being exploited in specific application domains.

2.1 Performer

A Performer is the entity executing specific activities, e.g., converting an image, calculating a pollutant concentration, and so on. The Performer may require data to execute its tasks, and the performed activities may produce data. A Performer must be activated in order to perform activities. For this purpose, it must be able to accept a suitable set of commands.

A Performer owns a local environment composed by two different sets of variables: the local variables, visible inside the Performer only, and the set of visible variables, visible also outside the Performer. In turn, the set of visible variables is arranged in two subsets: the set of exported variables, variables writeable by the Performer and readable from the outside (i.e., which can be assigned from the inside and observed from the outside), and the set of imported variables, variables readable by the Performer and writeable by the outside (i.e., which can be observed from the inside and assigned from the outside). This formal separation is important because it allows a Performer not to be aware of the information source and target respectively: a Performer may only publish data on the exported visible and read data from the imported visible. Thus, the set of visible represents the only “port” to the outside world of the Performer. This must be regarded as an advantage because makes Performers completely unaware of the system in which they work, and, consequently, they may be composed under different topologies.

2.2 Projector

A Performer may require data produced by other Performers to carry out its activity. Moreover, a Performer may produce data useful to other Performers at the end of its computation. In RTP a Performer can observe the exported visible of another Performer only if they are projected into its imported visible. For this goal, RTP introduces the concept of Projection. A Projection defines a pair of variables (source, target) where source is an exported visible of one Performer, and target is an imported visible of another Performer. Thus, Projections define mappings from visible exported variables of a Performer into imported visible variables of another Performer. Defining Projections, the system topology will be described without embedding inside the Performers the knowledge neither about the Projections nor of other Performers. Finally, the definitions of Projections do not imply any assumption neither about observation mechanisms, nor about why and when observation is done: Projection reification (Projector) performs alignment activities only upon triggered commands.

2.3 Strategy

Performers and Projections are not aware about the system dynamics: operating such separation of concerns allows reusing the same components under different system dynamics. The definition of the system behaviour must be assigned to a specific entity that has the complete view of both the system topology and the system condition. RTP introduces Strategy as the function that is able to define the future system behaviour given the current state and the set
of the requests (commands) that have to be delivered.

The set of requests are arranged inside Traces. Every entry inside the Trace fully specifies both the request recipient - Performer or Projector - and the type of action the recipient has to perform. In situations in which timing is not relevant, a Trace fully define the system future behaviour. When introducing timing, a Trace becomes a TimedTrace: a set containing special kind of request including timing constraints.

To allow the observation and the control of the system temporal behaviour, RTP introduces Timeline and Virtual-Clock concepts. A Timeline is a monotonic sequence of integers and current time represents a point in a Timeline bounded to change monotonically. A Virtual Clock is an active component that is in charge of advancing the current time of a Timeline. A tick is an increment of the current time of a Timeline, while the period defines the Timeline advancement speed.

A request inside a TimedTrace has its temporal validity defined over a Timeline. Requests are only eligible for being extracted from a TimedTrace when the current time falls inside the associated interval. A Timed Trace keeps track of the current time using Virtual Clocks.

By defining clever Strategies the RTP architecture allows the explicit definition of policies and mechanisms without affecting both Topology and Performers definitions. For instance, an alignment can be triggered according to three different Strategies: push (the Strategy states that a Request is issued to a Projector when its source variable changes), pull (the Strategy defines that a request is issued to a Projector on a change of a suitable exported visible variable of the Performer the target belongs to), and timed (the Strategy triggers actions method according to a specific timing). Similar remarks apply to the Strategies that generate commands to Performers.

3. The Performer component behaviour

Speaking in object oriented terms, information resides in object attributes while behaviour in the set of methods exposed by the object interface. The usual way of thinking about an object (and its class) is that the behaviour of any method is defined by the body (source code) and it is set at coding time. Polimorphism adds a somewhat dynamic change of behaviour in the sense that the exact behaviour is chosen at instantiation time by selecting a specialized instance. Then, in the history of object oriented languages, the Strategy Pattern [3] came and added the ability to change behaviour at runtime by changing the ‘engine’ of an interfacing object. The strategy pattern allows the runtime addition and removal of a single behaviour, i.e., a instance of a ConcreteBehavior. The idea of Adaptive Object Models [12] introduced an architectural style [11] supporting runtime behavioural changes. Unfortunately still modeled after standard object oriented concepts with the application of some more clever design pattern.

Instead, the RTP Performer is modeled after a state machine, i.e., it can be defined as:

\[ M = \{ S, A, s, d \} \]

where: \( S \) is the set of possible states; \( s \) is the set of initial states; \( A \) is the set of actions; \( D \) is the transition relation \( d : S \times S \).

The definition above can be refined to model two abstraction levels that are relevant in terms of both model and architecture. Refer to a classical UML state diagram, where states represent macro states corresponding to different behaviors: a command triggers a transition from the current state to a next state. Transitions are associated with actions. A transition is selected according to the incoming command and to a a guard. A guard is a boolean expression depending on the local environment (see below). At the micro level, a Performer has a local environment, i.e., a set of local variables (in the simplest case, the attributes of a class). Guards are expressed in terms of environment variables. Actions manipulate environment variables. This means that the (macro) states evolution is not affected by the performed actions.

Adopting a formal definition a Performer \( i \)-th is a tuple:

\[ P_i = \{ S_i, s_i, T_i, C_i, E_i, G_i, A_i \} \]

where: \( S_i \) is the set of possible states; \( s_i \) is the set of initial states; \( T_i \) is the set of transitions; \( C_i \) is the set of commands, i.e., symbols; \( E_i \) is the local environment, i.e., a set of variables defined in a local name space; \( G_i \) is the set of guards, i.e., boolean expressions defined over \( E_i \); \( A_i \) is the set of actions that manipulate \( E_i \). Moreover, a transition \( t \in T_i \) is defined as:

\[ t = \{ s_{or}, s_{de}, c, g, a \} \]

where: \( s_{or} \in S_i \) is the origin state; \( s_{de} \in S_i \) is the destination state; \( c \in C_i \) is a command; \( g \in G_i \) is a guard; \( a \in A_i \) is an action.

This can be easily expressed in UML by local variables (environment) and methods (actions) in a class diagram, and by states, commands, transitions and actions in a state diagram.

RTP concrete architecture defines a Performer class to model the Performer entity. The abstract class method `accept(Command aCommand)` is responsible for activating the Performer accordingly with the Command instance in input. The semantic of the Command instance is local to the receiving Performer. Concrete Performers should be specialisations of the Performer class, and they have to provide an opportune definition of the `accept()` method definition. Performer visible variables are defined as classes implementing the Visible
interface. This interface represents both imported and exported visible variables whose handling is made using the get() and the set() methods respectively. NotifyingVisible is a special kind of Visible that send a notification when it is updated. This class may be very useful when an event-driven system is preferred to a polling one.

3.1. The ProgrammablePerformer

To create a truly dynamically (at runtime) changeable Performer, the ProgrammablePerformer class was defined as an extension. The ProgrammablePerformer represents an explicit state machine, it reifies states with State instances. Figure 1 shows the UML class diagram of this class with its set of states and ECA\(^2\)-transitions. The ProgrammablePerformer has a set of States, one is marked as “initial” and there is always a reference to the “current” one. Each State has a set of Transitions that will be examined when interpreting acceptable commands. Each Transition reifies an ECA transition, i.e., it is associated with an (optional) AcceptableCommand, a (optional) Condition, a (required) next State, and a (optional) Action. The accept() method on the State class cycles through all the associated Transitions until the following two conditions hold: a Transition that matches the actual received Command exists, and the (optionally) associated Condition is evaluated as true.

If such a Transition is found, its corresponding Action is executed and its “next” State becomes the “current” one. Both Condition and Action need a link to the Performer (marked “context” in the Figure 1) because they respectively need “read”\(^3\) and “read and write”\(^4\) access to the Performer.

In this model, each class reifies an aspect of the formal state machine described in Section 3:

ProgrammablePerformer represents the state machine: it exposes a current state, the accept() method (to make it execute commands), and the methods to manipulate the set of states (i.e., getState(), getStates(), addState(), getInitialState(), setInitialState())

State (set of, associated to the Performer) represents a particular public state of a Performer: it has a name and the methods to manipulate the set of transitions (i.e., getTransition(), addTransition())

---

\(^2\)Event-Condition-Action

\(^3\)A condition is evaluated against the state of the Performer.

\(^4\)An action may also modify the Performer.
new AcceptableCommand("OFF"),
getState(0));
// initial state is (0)off
setInitial(0));}

Let now assume the need to change the behaviour of a single instance of this class. This may be the case when, for example, that instance is driving a high-load device and the Strategist (the entity reifying strategy) wants to limit usage of the device during peak hours. The behaviour change may be done as simply as adding a condition, in this case a Condition instance that checks for the current time, after having instantiated the switch:

Switch s1=new Switch();
s1.accept("OFF"); // put it in (0)ff state
s1.getState("OFF").getTransition("ON").
setCondition(new
NoPeakHoursCondition());
// from now on, the 's1' switch will not
// change from (0)ff to (1)on
// during peak hours

The class NoPeakHoursCondition is an implementation of the Condition interface by defining the evaluate() method. The method body will be a simple check of the current time against a static (or dynamic) peak-hour table, as in the following example code:

public class NoPeakHoursCondition
implements Condition{
public boolean eval(){
return System.getTime()<9 ||
System.getTime()>18;}}

Note that System.getTime() does not exists in Java but it is very easy to write a method to get the current time through the use of Date and Calendar classes.

With the same mechanism, the Strategist may revert to the original behaviour by only removing the condition instance from the transition from “off” to “on”. This technique may be applied, of course, to bigger and more complex ProgrammablePerformer examples. To the extent of having an Action instance that modifies the ECA rules of the state machine itself. In this extreme case the ProgrammablePerformer becomes a self-modifying reflective state machine.

4. Conclusions and Future works

The paper introduces RTP, a software architecture planned for the design of strongly modular, distributed, and real-time system. RTP exploits reflection mechanisms in order to raise at the application programming level the definition of strategies, the management of timing issues, the component behaviours definition, and the system topology definition.

By considering computational entities, alignment entities, and strategies activating both computational and alignments activities as orthogonal issues, RTP operates a strict separation of concerns. In particular, the architecture refines the paradigm of the Strategy Pattern by applying the state machine model to the computational component with the purpose of dynamic behaviour change. By using reflective methods, the programmer (actually, the Strategist) may modify any part of the state machine, thus building a (potentially) completely different Performer than the one initially instantiated. The idea behind the choice of state machines (as inspiration for the computational component) is that there is a lot of research already available on FSA [4] (Finite State Automata) and their properties.

The framework is currently under test in a vision/robotics 3D reconstruction application based on trinocular stereo vision, a complex distributed movement tracking system based on webcams [7], and workflow modeling system based on distributed document management.

References

[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns Elements of Reusable Object-Oriented Software. Addison-Wesley, 1994.