# Architecture Composition for Concurrent Systems

Thomas Cottenier
UniqueSoft, LLC
thomas.cottenier@uniquesoft.com

Aswin van den Berg
UniqueSoft, LLC
aswin.vandenberg@uniquesoft.com

Thomas Weigert
Missouri University of S&T
weigert@mst.edu

## Abstract

We present a framework to assemble concurrent applications from modules that capture reusable architectural patterns. The framework focuses on concurrent systems where computational processes communicate through asynchronous messages. The language provides support to modularize architectural patterns at different levels of granularity, using agents, regions, aspects and morphing. We present sample implementations of the architectural patterns and show how they are composed using a real-world example. Finally discuss how the deployment and composition of patterns can be further automated.

***Categories and Subject Descriptors*** D.3.3 [**Programming Languages**]: Language Constructs and Features – *classes and objects, modules, packages.*

***General Terms*** Design, Languages

***Keywords*** Architecture, Concurrent Systems, Aspects

## 1. Introduction

Concurrent systems are systems where multiple computational processes or agents can execute independently of each other. Many applications have a high level of inherent concurrency. Components that provide services to clients might need to service thousands of concurrent requests. Another source of concurrency comes from applications that are built by integrating different types of components, which may be distributed over the network or using different technologies [1]. Other applications use concurrency to take advantage of parallelism provided by the underlying platform.

In this paper, we consider concurrent systems where computational processes called agents communicate through asynchronous messages. Messaging provides a simpler concurrency model than the shared-memory interaction model used in general-purpose languages.


*NEMARA'12*, March 27, 2012, Potsdam, Germany.


The architecture of a message-based concurrent system has a determining effect on the performance characteristics of the system such as response time and throughput or robustness features such as availability or connectivity. It is therefore important to be able to fine tune or adapt the architecture of the system when the operating conditions or the platform used to execute the system change. Ideally, changes to the architecture should not require important modifications to the implementation of the functional modules of the system.

In this paper we present a set of concurrent architectural patterns that are expressed in a textual design language called TDL [2]. TDL has two aims. First, it provides first class support for concurrency and messaging. Second, it aims at maintaining a clear separation between the implementation of architectural features and the implementation of functional features. The language therefore supports 3 top level modules: agents, regions and aspects and a generative language construct called morphing.

The paper is organized as follows. First we present the language constructs supported by the language. Second, we introduce a list of architectural patterns and the categories used to classify the patterns. Third, we detail the implementation of selected patterns and show how they are composed to construct a system. Finally, we discuss how the deployment and composition of the patterns can be further automated.

## 2. Language Support

### 2.1 Agents

Agents are a conceptual unit of concurrency independently of the underlying technology used to interleave their execution. An agent has its own thread of control and executes a state machine that reacts to messages and timer expiration events. Agents communicate using messages that are sent or received through ports. The ports of different agents are connected by connectors which define the communication paths between agents. Agents cannot directly access each other's data and do not share global variables. There is therefore no need for explicit locking mechanisms such as mutexes or monitors.

We represent instances of agents using boxes with bold edges. Instances are annotated with multiplicities. Instances communicate with each other through connectors between ports, which are represented as squares.

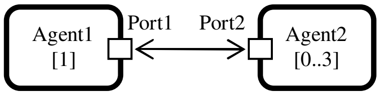


**Figure 1.** Agent Representation

## 2.2 Regions

A statemachine is composed of a set of regions that modularize different features of an agent [3]. Each region encapsulates a set of states, transitions, attributes, and operations that are specific to the feature implemented by the region. Regions interact through:

1. Common inputs: multiple regions respond to the same event, thereby weaving there transitions within a step.
2. Internal signals: an event generated by one region can be consumed by another region, in the next step.
3. Guards: transitions of one region can be guarded by states of another region. If the transition is not enabled, the corresponding event is not consumed by the region.

Regions are represented using boxes within agents. Regions can be associated with ports using arrows which indicate that the signals flowing through the port are handled or generated by the region. Interactions between regions are also represented using arrows. A solid arrow indicates that internal events are generated by one region and consumed by another. Dashed arrows indicate that one region affects the control flow of another region through guards. In Figure 2, both *Region1* and *Region3* consume signals flowing through *Port1*. *Region2* observes *Region1* and *Region3* regulates *Region2*.

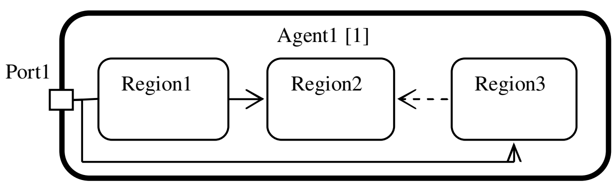


**Figure 2.** Region representation

## 2.3 Aspects

Aspects modularize concerns that cut across multiple agents or regions, or that interact with one or more features in an invasive way. TDL aspects support pointcut expressions to intercept the output of messages and the execution of transitions within regions. Output pointcuts have the form:

```
pointcut outpct (p_t p1, p_t p2) :
  output (s(p1_t, p2_t))
     && args (p1, p2) && within (Region1);
```

Transition pointcuts have the form:

```
pointcut trpct (p_t p1, p_t p2) :
  transition (for S1 input s(p1_t, p2_t) nextstate S1)
     && args (p1, p2) && within (Region1);
```

Intertype declarations can also add states and transitions to specific regions. Within a region, we represent aspects using dashed boxes. Aspects interact through 3 types of interactions: we use an arrow between aspects to indicate that one aspect modifies the state of another. A dashed arrow is used to indicate that one aspect affects the control flow of another. Finally we use a dotted arrow to indicate that an aspect observes the state of another aspect.

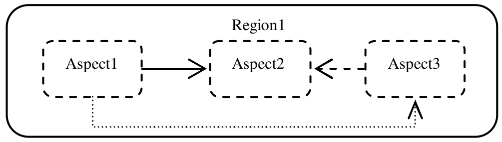


**Figure 3.** Aspect Representation

## 2.4 Morphing

Morphing is a generative technique to generate entities such as attributes, operations or transitions by iterating over the structure of other entities [4]. Morphing transformations follow the structure:

```
'foreach' entity-pattern 'in' entity '.' metafeature
   { morphism-body }
```

The morphism body is expanded for each entity that matches the entity-pattern and that matches the metafeature query. For example, morphing can be used to generate a transition from state *A* to state *B* for all the signals that flow in port *p*, using the form:

```
   foreach signal $s in port p . in {
      forstate A {  input $s {  nextstate B;  }  }
```

In this example, the entity pattern *signal $s* matches any signal that is declared to be accepted by port p and binds the name of the signal to the variable $s. The morphing will expand the transition based on the name of the signal and add it to the region.

# 3. Architectural Patterns

The following architectural patterns are inspired from the patterns presented in [1], but are expressed using the TDL concurrency and modularity constructs. Some new patterns have been introduced to handle interactions between agents and regions. We distinguish between inter-agent patterns, inter-region patterns and endpoint patterns.

## 3.1 Inter-Agent Patterns

Inter-agent patterns define how different agents interact. In particular, it defines how they control each other's life-cycles and how sets of agents of different multiplicities are coordinated. We identified the following inter-agent patterns:

1. Controller: The controller agent controls the life-cycle of a set of agents.
2. Scheduler: The scheduler agent assigns a request to one agent among a set of agents, based on the agent availability.

3. Dispatcher: The dispatcher agent forwards a set of requests to one agent among a set of agents, based on the value of a parameter of the request.
4. Correlator: The correlator agent directs responses to one agent among a set of agents by correlating the id of the response with the sender of the corresponding request.
5. Multiplexer: The multiplexer agent mediates between two set of agents by dispatching each other's requests/responses based on messages ids.
6. Orchestrator: The orchestrator agent defines communication paths between different agents among a set of agent, based on a specification.

### 3.2 Inter-Agent Patterns

Inter-region patterns define how the regions of one agent are composed with each other. They define how external requests are propagated to the regions and how the correspond responses are assembled.

1. Monitor: The monitor region observes the execution of another region
2. Regulator: The regulator regions controls the execution another region
3. Abstractor: The abstractor region provides a mapping between a set of concrete external requests and a set of more abstract internal requests.
4. Composer: The composer agent assembles different types of agents from different regions and aspects, based on the type of service provided.
5. Distributor: The distributor region distributes a set of requests to other regions.
6. Collector: The collector region collects responses from a set of regions.
7. Transaction: The transaction allows a set of region to agree on the outcome of a request and produce consistent responses and resulting configurations.

### 3.3 Endpoint Patterns

Endpoint patterns define how an agent handles a set of external signals and are associated with a port of that agent.

1. Timeout-Retry: The timeout-retry pattern retries sending requests a specific number of times when a response is not received within a time interval.
2. Filter: The signal filter modifies or eliminates a set of requests before they are processed by the main state machine.
3. Buffering : The buffering pattern buffers requests when the number of outstanding requests exceeds a specific threshold.
4. Throttling: Throttling limits the rate of requests by buffering messages.
5. Splitter: The splitter segments requests into multiple requests when the payload exceeds a certain size. The corresponding responses are reassembled into a single response.

## 4. Implementation

In this section we detail how a selected subset of these patterns are implemented using the modularity constructs of TDL.

### 4.1 Controller

The Controller pattern allows one agent to control the lifecycle of a set of agents. Typically, the controller creates a session to handle a specific request, and terminates the session when the request has been processed. Listing 1 presents an example implementation of the *Controller* pattern in TDL.

The controller agent contains a set of *Session* agents and a map that correlates an index to a *Session* agent. The *Controller* accepts the external signals *createSession* and *deleteSession*, and can send the shutdown signal to *Session* agents through the *session_out* port. The *session_out* port is connected to the *ctrl_in* port of the *Session* agent by a connector.

```
agent Controller {
  Session [0..MAX] sessions;
  Map <Index, Session> sessionMap;
  Session session;
  port env in with createSession, deleteSession;
  port dispatch out with shutdown;
  connector from dispatch to sessions.ctlr_in;
  region controller {
    start {
      nextstate Active;  }
    forstate Active {
      input createSession (index) {
        sessions.append (new Session() );
        session := offspring;
        sessionMap.add(index, session);
        nextstate Active;  }
      input deleteSession (index) {
        output session.shutdown();
        sessionMap. remove(index, session );
        nextstate Active; } } } }
agent Session {
  port ctrl_in in with shutdown;
  region session { ...  } }
```

**Listing 1.** Controller Pattern implementation

### 4.2 Dispatcher

The dispatcher pattern is used when a signal needs to be sent to a specific agent among a set of agents. The dispatching is performed based on a mapping between the values of a key or index and the agents managed by the dispatcher. In most cases, the index is passed as a parameter of the signal or computed from these parameters.

The pattern uses morphing to iterate over all the signals that flow through a specific port. In the example of Listing 2, the dispatcher is implemented using morphing to iterate over all the signals that flow in the *env* port and that have two parameters, the first parameter being of type *Header_t*. For each match, a transition using the values of the identifiers variables *$Request* and *$Request_t* is generated.

```
region dispatcher {
 Header_t h;
 foreach signal $Request (Header_t, $Request_t)
       in port env. in {
  $Request_t req;
  forstate Active {
   input $Request (h, req) {
    if (sessionMap.find(h.index , session)) {
     output session.$Request(h, req) via dispatch;
    } else {
     sendResponse (SessionNotFound); }
     nextstate Active; } } } }
```

**Listing 2.** Dispatcher pattern implementation

### 4.3 Buffering

The buffered pattern is an endpoint pattern used to control the number of outstanding requests on an external port. This is typically used to mediate between two sets of agents that have different multiplicities, or when two sets of agents support different levels of concurrency, as is the case when TDL models need to interface with Java or C code where concurrency is implemented using threads.

Listing 3 shows an *Interface* region that forwards requests through a port, and sends back responses to an agent. Listing 4 shows how the buffering pattern is implemented as an aspect that applies to the *Interface* region. The *Buffering* aspect keeps track of the number of outstanding requests sent through a port. If the number of outstanding requests exceeds a threshold, new requests are stored in a buffer until a response is received.

```
region Interface {
 state ACTIVE;
 state OVERFLOW;
 start {
  nextstate ACTIVE; }
 forstate ACTIVE {
  input Request (request) {
   output Request via OUT;
   nextstate ACTIVE; }
  input Response (response) {
   if (sessionMap.find(response.index , session) )
    output session.Response (response); }
    nextstate ACTIVE; } } }
```

**Listing 3.** Interface Region

```
aspect Buffering{
 intertype Interface {
  Integer outstandingRequests := 0;
  before (Request_t req) :
    output (Request (Request_t) ) && args (req) {
   if (outstandingRequests >= NbrThreads ) {
    if (!buffer.isFull()) {
     buffer.put(req) {
     nextstate ACTIVE;
    } else {
     nextstate OVERFLOW; }  } }
  after () : output (Request (Request_t)) {
   outstandingRequests := outstandingRequests + 1;}
  after () : transition ( input Response ) {
   outstandingRequests := outstandingRequests - 1;
   if (! buffer.isEmpty()) {
    request := buffer.get();
    output Request (request);
    outstandingRequests := outstandingRequests+1;
    } } } }
```

**Listing 4.** Buffering pattern implementation

## 5. Deployment

Figure 4.a shows an architecture diagram for an application that performs analysis on network traffic that transits between the external ports *IN* and *OUT*. The application supports a large number of concurrent sessions, and should minimize the CPU utilization and memory usage. The application is composed of 4 functional units, modularized as regions. The system contains a *Dispatcher* to forward client requests to the right session. A *Multiplexer* is used to concentrate the client requests into a smaller number of outgoing connections. Finally, the *Correlator* correlates responses from the external component to the sessions.

Figure 4.b shows an alternative architecture that uses dedicated sessions to perform the analysis. On average, only 5% of the sessions require analysis. The *AnalysisSession* sessions are connected to the *ProxySession* sessions by a buffering scheduler, which absorbs peaks of analysis requests. This architecture has the following advantages. First, the proxy sessions do not need to load analysis data when no analysis is being performed for the session. The architecture therefore consumes less memory. Second, multiple analysis can be performed concurrently for the same proxy session, which is not possible with the first architecture. The evolution from architecture 1 to architecture 2 can be performed without modifications to the implementation of the functional modules *Proxy*, *Monitoring*, *Analysis1* and *Analysis2*.

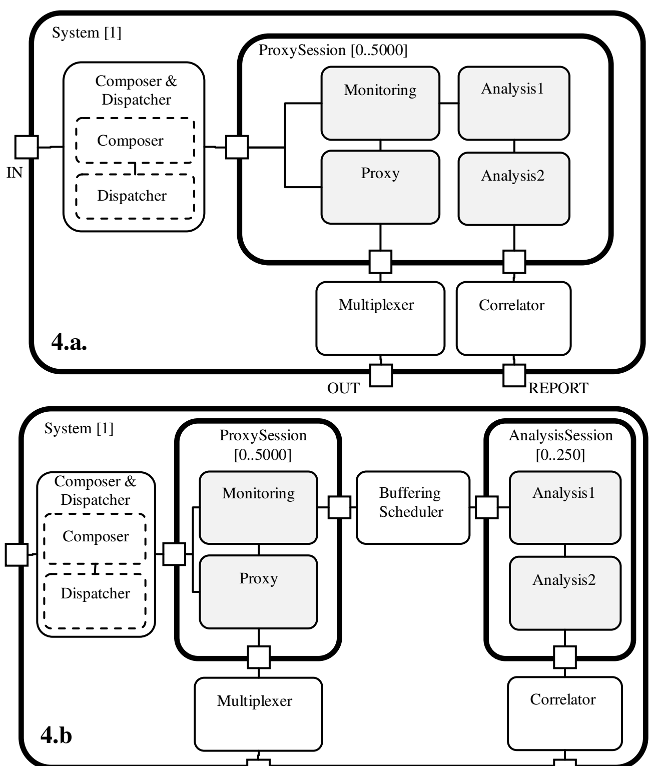


**Figure 4.** Architectures of the Traffic Monitoring System

## 6. Automation Challenges

The instantiation of architectural patterns is performed manually by customizing the pattern to the data types, signals and protocols of the application. The language support for regions and aspects make this task easier, because the implementation of the patterns is more modular. Generative techniques such as morphing also help by automatically generating and contextualizing parts of the pattern implementation. However, most of the contextualization and deployment of the patterns is still done by hand. Different approaches can be used to automate the deployment of architectural patterns.

### 6.1 Domain-Specific Languages

Frameworks such as Spring Integration [6] and Apache Camel [7] use domain specific languages to capture architectural patterns. Such languages allow a pattern to be configured and contextualized using entities of a base model passed as parameters. A graphical representation, such as the one presented in the paper, allows architectural patterns to be deployed using a drag and drop approach.

Such approaches support architecture evolution because the DSL abstract from many details about the implementation of the architectural patterns. Architectures can therefore quickly be deployed and modified by composing the patterns at the level of the DSL.

However, architectural patterns have a large number of context parameters that need to be defined before they can be deployed. If the language constructs provided by a DSL abstract too much of the details of the pattern, the construct becomes less usable: it cannot be deployed in certain contexts because the pattern needs to be adapted in a way that the language does not support. On the other hand, DSL's that expose too many details of the pattern implementation are more complex and have a steep learning curve. The abstraction benefits provided by the DSL are reduced.

### 6.2 Round-Trip Engineering

Architecture specifications can co-exist with code written in a general purpose language using round-trip engineering environments such as Rational Software Architect [7]. Round-trip engineering approaches maintain two separate levels of specification of the system, and resolve inconsistencies between these descriptions by propagating modifications from one level to the other. Round-trip engineering can be applied to domain-specific languages by synchronizing a representation of the system defined in a domain-specific language with a representation of the system defined in a general-purpose language, or a language such as TDL presented above.

The advantage of the round-trip approach is that the domain-specific language can be more abstract. The details of the pattern contextualization are implemented at the code level. In such environments, it is especially important to keep the representations of the patterns modular. If the implementation of the patterns become tangled at the code level, it will become more difficult to modify the definition of the system at the level of the DSL. The language constructs of TDL support this objective.

## 7. Conclusions

We build application by assembling contextualized instantiations of architectural patterns with modules that implement functional features of the system. Architectural decisions have tremendous impact on the performance, scalability and quality attributes of a system. It is therefore important to be able to modify or fine tune architectural elements of a system, without requiring important changes to the implementation of the functional features of the system.

We present a language for the development of asynchronous systems called TDL. We show that TDL language features such as agents, regions, aspects and morphing can support the modular implementation of architectural features. We discuss different type of architectural patterns and detail the implementation of selected patterns in TDL. We show how these patterns are deployed and composed to build systems, and how the TDL language construct support architecture evolution. Finally, we discuss how domain-specific languages and round-trip engineering can automate the deployment and composition of architectural patterns.

## References


[1] Hohpe, G. and Woolf, B. 2003. Enterprise Integration Patterns. Addison Wesley

[2] Cottenier, T., van den Berg, A. and Weigert, T. 2012. Management of Feature Interactions with Transactional Regions. In Proceedings of the International Conference on Aspect-Oriented Software Development, Postdam, Germany.

[3] Harel, D. 1987. Statecharts: A visual formalism for complex systems, Science of Computer Programming, Volume 8, Issue 3. 231-274.

[4] Huang, S.S., Zook, D. and Smaragdakis Y. 2007. Morphing: Safely shaping a class in the image of others, In Proceedings of the 21st European Conference on Object-Oriented Programming, Berlin, Germany, LNCS 4609, 399-424.

[5] Fisher, M., Partner, J., Bogoevici, M. and Fuld, I. 2009. Spring Integration in Action. Manning.

[6] Ibsen, C and Anstey, J. 2010. Camel in Action. Manning.

[7] Liu, C. 2010. Round Trip Engineering Scenario using Rational Software Architect and ClearCase Remote Client. IBM developerWorks.