Walter Cazzola, Ahmed Ghoneim, and Gunter Saake. System Evolution through Design Information Evolution: a Case Study. In Proceedings of the 13th International Conference on Intelligent and Adaptive Systems and Software Engineering (IASSE'04), *pp. 145–150, Nice, France, July 2004. ISCA.*

System Evolution through Design Information Evolution: a Case Study

Walter Cazzola
Dept. of Informatics and Communication,
Università degli Studi di Milano
Via Comelico 39/41, 20135
Milano, MI, Italy
cazzola@dico.unimi.it

Ahmed Ghoneim, **Gunter Saake**
Institute für Technische und
Betriebliche Informationssysteme,
Otto-von-Guericke-Universität Magdeburg
Germany, D-39016
{ghoneim,saake}@iti.cs.uni-magdeburg.de

Abstract

This paper describes how design information, in our case UML specifications, can be used to evolve a software system and validate the consistency of such an evolution. This work complements our previous work on reflective architectures for software evolution describing the role played by meta-data in the evolution of software systems. The whole paper focuses on a case study; we show how the *urban traffic control system* (UTCS) or part of it must evolve when unscheduled road maintenance, a car crush or a traffic jam block normal vehicular flow in a specific road. The UTCS case study perfectly shows how requirements can dynamically change and how the design of the system should adapt to such changes. Both system consistency and adaptation are governed by rules based on meta-data representing the system design information. As we show by an example, such rules represent the core of our evolutionary approach driving the *evolutionary* and *consistency checker meta-objects* and interfacing the meta-level system (the evolutionary system) with the system that has to be adapted.

Keywords: Software Adaptation, Meta-Data, UML.

1 Reflective System Evolution

Nowadays, evolving a software system is an hot and hard topic especially when evolution must take place without stopping the system execution to face sudden and unexpected events. Evolution must face several issues, the most important consist of:

- planning the best strategy for adapting the system evolution against unexpected events; and
- verifying the consistency of the planned strategy with respects to the original requirements of the system.

In [2, 3] we have described our proposal for system evolution. Our idea consists of a reflective architecture, the system running in the base-level is the one prone to be adapted, whereas the software evolution is the nonfunctional feature realized by the meta-level.

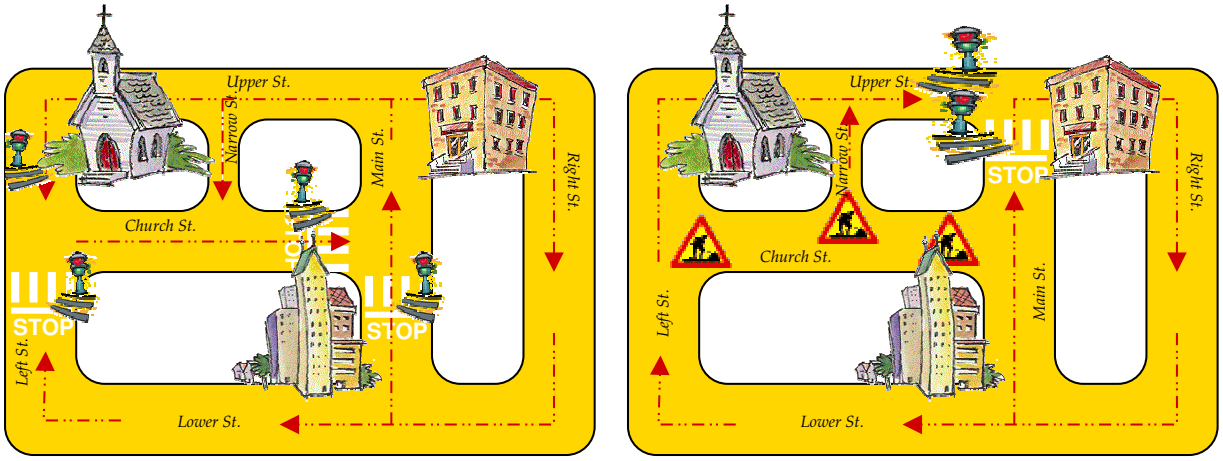
Evolution takes place exploiting the design information of the running systems and it is driven by a set of rules which define how to adapt the system in accordance with the detected event and the meaning of system consistency. These rules are not hardwired in the meta-level system and depends on the system prone to be adapted, moreover, they provide the core mechanism for facing the abovementioned issues.

The meta-level system of our reflective middleware is composed of at least two cooperating meta-objects: *evolutionary* and *consistency checker meta-objects*. Both these meta-objects have two main components: (i) the core which interacts with the rest of the system (e.g., detecting external events/adaptation proposals, or manipulating the reification of the base-level system/applying the adaptation on the base-level system) and implementing the meta-object's basic behavior, and (ii) the engine which interprets the rules driving the meta-object's decisions.

This paper focuses on rules and their role in the adaptation mechanism. The mechanism is explained on a case study: the dynamic adaptation of *urban traffic control systems* (UTCS) on a generic road maintenance event that forces the modification of the vehicular flow.

2 Case Study Overview

The evolution of complex urban agglomerates has posed significant challenges to city planners in terms of optimizing traffic flows in a normally congested traffic network. When designing *urban traffic control systems* (UTCS) of a modern city, the software engineer must model both mobile



(a) Standard situation

(b) Temporary situation

Figure 1: City layout: a) the layout during normal activities b) the layout during road maintenance.

entities (e.g., cars, pedestrians, vehicular flow, and so on) and fixed entities (e.g., roads, railways, level crossing, traffic lights and so on). They have also to face many design issues such as distribution, complexity of configuration, and reactivity to the evolution of the environment. In [7] these issues and many others are illustrated.

Moreover, Software engineers, designing the UTCS, have to deal with a lot of unexpected and hard to plan problems of modern cities such as traffic lights disruptions, roads maintenance, car crashes, traffic jam and so on. On these considerations rely the need of rendering the UTCS dynamically adaptable to the external events.

The map in Fig. 1.a could represent a simplification of a real city map. Notwithstanding that, it can help us in understanding the problems that a city planner has to face when plans the UTCS of its city. The city planner must plan traffic system taking in consideration several issues, two of them, that we consider in our case study, are:

- cars must be able to reach every road from everywhere; and
- opposite traffic lights at the same crossroad (e.g., traffic lights at the crossway between *Church St.* and *Main St.* in Fig. 1.a) must be synchronized or they are useless.

A city map can be easily represented by an oriented graph $G \equiv (\texttt{Crossroads}, \texttt{Roads})$ whose nodes are crossroads and whose edges are roads. Therefore, the first requirement above can be formalized in:

$$\forall c_1, c_2 \in \text{Crossroads} \wedge c_1 \neq c_2 \; \exists p \equiv r_1 \ldots r_n, r_i \in \text{Roads s.t.}$$
$$\forall i, 1 \leq i < n, \exists c, \bar{c}, \bar{c}', c', c'', \in \text{Crossroads s.t.}$$
$$r_i \equiv (\bar{c}, c) \wedge r_{i+1} \equiv (c, \bar{c}') \wedge r_1 \equiv (c_1, c') \wedge r_n \equiv (c'', c_2)$$

That is, all crossroads in the map are connected by a path of roads. Analogously the second requirement can be formalized in:

$$\forall r, p \in \text{Roads s.t. } r \perp p \text{ if } \exists t_r, t_p \in \text{TrafficLights} \implies$$
$$\text{red}(t_r) \iff \text{green}(t_p) \wedge \text{green}(t_r) \iff \text{red}(t_p)$$

Where `TrafficLights` is the set of all the traffic lights marked on the map, t_r and t_p are respectively the traffic lights in r and in p, and `red()` and `green()` are predicates that state if the traffic light passed as argument is or not of the homonym color at the invocation[1].

An urban traffic control system that respects such criteria is consistent with the basic requirements that a livable city must have. Therefore, the software engineer that designs an UTCS should guarantee that such criteria will be respected. *These requirements define the* ***consistency*** *of the system design.*

3 The Role of Design Information

The UTCS for the simple city map showed in Fig. 1.a can be described by the class diagram showed in figure 2, an object diagram (Fig. 4.a) that defines the interconnections among roads and crossroads and a statechart that express the dependencies among traffic lights (see Fig. 3).

UML [1] provide the designer with a flexible mechanism (its diagrams) that allows of specifying the above-

[1] Note that, traffic light behavior has been simplified removing the yellow color and considering that red and green equally last.

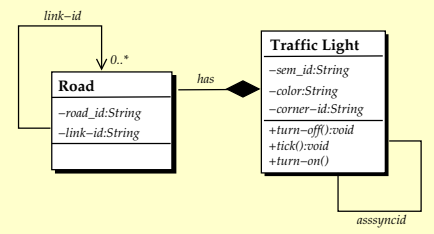


Figure 2: Class diagram of the urban traffic control system

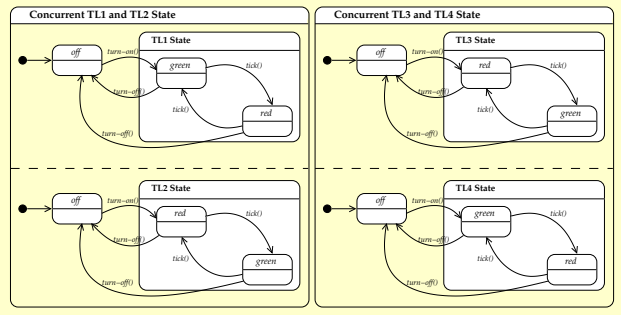


Figure 3: Statechart of the traffic lights in the map

mentioned requirements. Unfortunately, design information expressed through UML are available just at design- and development-time. Therefore, dynamic events such as traffic light malfunctioning and car accidents are hard to be captured at design-time whereas their occurrence surely involves also design information often asking for a complete redesigning, e.g., introducing new traffic lights and barrages. Analogous issues can be raised for the dynamic validation of the system behavior, e.g., to verify whether synchronized traffic lights are still synchronized. Moreover, the UML graphical representation is not suitable for an handy automatic manipulation.

As explained in [3] our reflective architecture provides a dynamic system evolution through the dynamic adaptation of the design information of the system itself (meta-data). The XMI [5] description contains all the design information necessary for checking the consistency of the system and for adapting its design at run-time; it represents the perfect reification of the UML diagrams at run-time [4]. The `Reification` data structure encapsulates the XMI description and provides the meta-level system with an high-level and simple interface for easily observing and manipulating the design information.

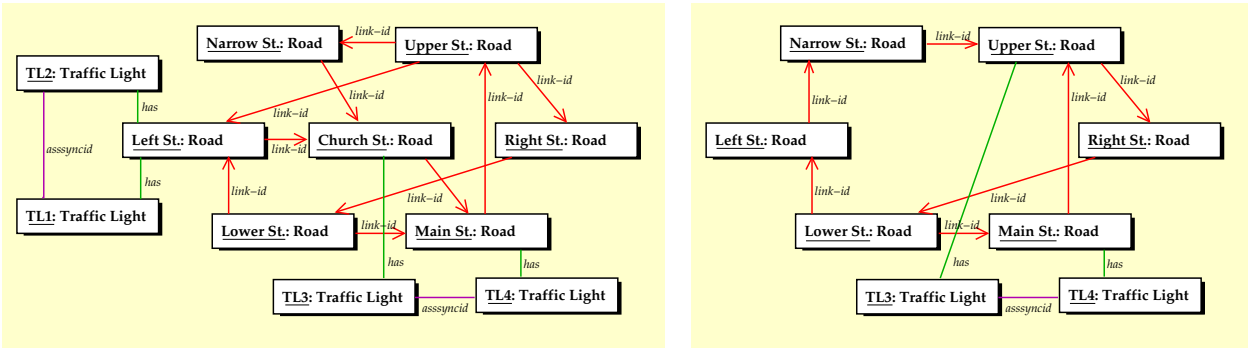
In our reflective architecture, the work of a specific meta-object, called *consistency checker meta-object*, is devoted to validate the consistency of the system prone to be adapted. The consistency checker meta-object uses the rules that define the consistency of the system (see section 2) to validate the system consistency against the system reification. To simplify their definition and to better integrate them with the system reification, rules are not logical expression but Ruby [8] scripts. Ruby is a Turing-equivalent scripting language, that provides a powerful and easy to use tool for describing constraints without the need of coding an ad-hoc interpreter. The work of the consistency checker meta-object basically consists of applying the consistency constraints on the system reification. The following script checks the traffic lights synchronicity at every crossroad.

```
def synchronized?(TL1, TL2)
    # Are the behavior of TL1 and TL2 synchronized?
  StateChart=Reification.getStateChart()
    # S1 and S2 represent the state machine of TL1 and TL2
  S1=StateChart.getStateMachine(TL1)
  S2=StateChart.getStateMachine(TL2)
  return
     # TL1 and TL2 have the same state machine, but ...
   (S1.transitFrom("green").include? "red") and
   (S1.transitFrom("red").include? "green") and
   (S2.transitFrom("green").include? "red") and
   (S2.transitFrom("red").include? "green") and
     # ... they start from the opposite state.
   (((S1.color=="green") and (S2.color=="red")) or
    ((S2.color=="green") and (S1.color=="red")))[2]
end

def all_synchronized?
    # Synchronized traffic lights are really synchronized?
  OD = Reification.getObjectDiagram()
  synchronized = true
     # For each traffic light pointed out in the object diagram ...
  for TL1 in OD.getAllInstances("Traffic Light")
        # ... it checks if it is associated to another traffic light ...
    unless (tl = TL1.asssyncid[2]) then
        # ... if yes, it verifies their synchronization.
      TL2 = OD.getInstance(tl)
      synchronized &&= synchronized?(TL1, TL2)
    end
  end
  return synchronized
end
```

As you can note, the script exploits the state machine defined for each traffic light in the statechart encapsulated in

[2]Note that, for sake of clarity, attributes specified in the class diagram are used as well as attributes of the class object instead of using the appropriate methods `getAttributeValue`.



(a) Standard layout

(b) Modified layout

Figure 4: Object diagrams: a) object diagram for UTCS during normal activities b) modified object diagram during road maintenance.

the `Reification` data structure. Exploring the statechart the script gets the traffic light state machine, its initial state and the transitions from a state to the others. Therefore it can verify whether the traffic lights at the same cross-point are synchronized. Analogously, a scripts that check the reachability on the map has been written but it is not reported for sake of brevity.

4 Road Maintenance: what does it happen?

Road maintenance is a typical event hard to predict at design time. Such a kind of prediction is difficult because the event can become manifest in many variants, e.g., it can involve just a road or a group of roads, the involved area can be completely close to the traffic or the traffic can become limited in one-way or limited in time, and so on. At design-time, the software engineer can provide some scenarios or some patterns of behavior (e.g., he can suggest how to hijack the traffic flow into the near roads) for managing such unexpected events but often their management is demanded at run-time.

What if an external event, such as a car accident or the road surface becomes unpaved, temporarily block the traffic flow in a road as showed in Fig. 1.b?

Removing *Church Street* from the city map, as depicted in Fig. 1.b, involves several small changes in the whole map and to the traffic flow. Some of the surrounding streets must be followed in a different direction to allow cars of reaching every place in the map. Traffic lights governing the traffic flow in and out of *Church Street* must be turned off (as the one at the corner with *Main Street*) or moved in a different place (as the one in *Main Street* moved at the corner with *Upper Street*).

Of course, a similar reorganization must involve also the UTCS that supervises the traffic flow and coordinates the work of traffic lights and other entities. Usually, such a kind of software adaptation is performed by redesigning the whole system and implementing the changes. As showed in Fig. 4.b, it is necessary to remove the object representing the interrupted street (or at least to disconnect it from the other objects) and the objects representing turned off traffic lights from the object diagram in Fig 4.a; to modify the connection link[3] of the remaining roads accordingly to the map in Fig 1.b. Few changes should also occur to the system statechart but they are not significant for the purposes of this work and for sake of brevity they will not be treated and the corresponding diagram will not be reported. Besides, in our case study, we did not consider the necessity of signalling that the access to *Church Street* is forbidden by adding a barrage; a similar approach to the road management would imply the introduction of a new class `Barrage` to the class diagram and the adaptation of the class `Road`, generating a lot of changes also in the other diagrams as a domino effect, augmenting the complexity of the case study to the detriment of its clarity.

The design adaptation must be propagated to the system code by mapping the changes into the original classes and introducing new classes and new instances. Some advanced tools as Rational Rose® and Poseidon4UML® could help the designer/programmer to automate this last phase.

Notwithstanding that, the proposed situation is quite simple and just few changes must be done to the original design, the nonstopping nature of the UTCS does not permit a similar approach that foresee a system stop. Moreover, in the UTCS, the reactiveness to sudden and unex-

[3]The connection link is a road attribute that represents which roads can be reached following that road; this attribute implicitly represents which direction can be followed.

pected events is particularly important. Therefore the described approach cannot be applied to the case study without an infrastructure that dynamically and promptly exploits design information for system adaptation.

5 Dynamic Design Adaptation

Our reflective middleware [3] has been designed to evolve nonstopping systems by adapting their design information. In the meta-level, a specific meta-object called *evolutionary meta-object* plans how the system must be adapted when an external event occurs. The plan designed by the evolutionary meta-object is a set of UML diagrams derived from the original system design (represented by `Reification`) by manipulating the involved entities. In our test case, the evolutionary meta-object must transform the object diagram showed in Fig. 4.a into the object diagram showed in Fig. 4.b. It must carry out an analogous transformation also to the statechart showed in Fig. 3.

The plan of evolution is prepared on a clone of `Reification` to preserve system consistency. `Reification` provides an high-level API invocable from Ruby that easily allows to add entities to, to remove entities from and to modify entities in the UML diagrams through their XMI representation.

As well as for the consistency checking also evolution is driven by a set of Ruby rules indexed on events. The evolutionary meta-object has just to invoke the right rule with the appropriate parameters. A plan for managing the maintenance of a generic road is prepared by invoking the following Ruby script.

```
def plan_inaccessible_road(r, map, tls)
   # Planning system adaptation when road r is inaccessible.
  OD = Reification.getObjectDiagram()
  StateChart = Reification.getStateChart()
   # Adapt the link-id of each road as specified by map.
  for R in OD.getAllInstances("Road")
    if map[R.road-id[4]] != nil then
      R.link-id[4] = map[R.road-id]
    end
  end
   # Traffic lights next to r have to be removed
  for T in OD.getAllInstances("Traffic Light")
    if T.corner-id.include? "Church" then
      OD.removeInstance(T)
      StateChart.removeStateMachine(T)
    end
  end
   # Traffic lights in tls must be added at new crossroads.
  tls.each_key {|name|
    theTL = OD.addInstance(name, "Traffic Light")
    tls[name].each_key {|attribute|
      theTL.addAttribute(attribute, tl[attribute])
    }
  }
   # Traffic lights at the same corner must be synchronized.
  tls = OD.getAllInstances("Traffic Light")
  state = "Synch State 0"
  for T in tls
      # finds S the TL on the same crossroad of T.
    S = onSameCorner(T)[5]
    S.asssyncid = T.asssyncid[4] = state.succ
    StateChart.addConcStateChart("TL", T, S, state)
  end
   # At last, r can be removed.
  OD.removeInstance(r)
end
```

The plan consists of modifying the object model and the statechart of the system, so that, (i) all the roads next to the inaccessible road change their flow direction according to the planner information, (ii) all the traffic lights near the inaccessible street are removed, (iii) new traffic lights are introduced at crossroads created after the changes in the flow direction, the behavior of such traffic lights will be synchronized, and (iv) at last the inaccessible road is removed from the system (it will be reintroduced when it is accessible again).

In our test case the maintenance of *Church Street* is managed by the following invocation:

```
plan_inaccessible_road("Church St.",
  { "Left" => "Upper", "Upper" => "Right",
    "Narrow" => "Upper" },
  { "TL-Upper" => { "sem_id" => "s1-U1",
        "corner-id" => "Upper St. & Main St." },
    "TL-Main" => { "sem_id" => "s2-M12",
        "corner-id" => "Main St. & Upper St." }
  })
```

The adaptation plan created by the evolutionary meta-object is not immediately applied on the system that must be adapted. It is passed to the consistency checker meta-object that verifies whether the system will remain consistent after the evolution. As explained in section 3, the con-

[4] Note that, for sake of clarity, attributes specified in the class diagram are used as well as attributes of the class object instead of using the appropriate methods `getAttributeValue` and `setAttributeValue`.

[5] The function `onSameCorner(T)` looks in the street or square for another traffic light that is in front of `T`. For sake of brevity we do not detail its code further.

sistency checker meta-object will invoke all the rules that define the consistency of the system against the redesigned meta-data. The plan will be applied to the base-level system only if it passes the consistency check otherwise the evolutionary meta-object tries to design a novel evolutionary plan.

Finally, the implicit causal connection of the reflective middleware is responsible of modifying the base-level system in accordance with the proposed evolution. The adopted mechanism for transposing design information into the real system is based on the UML virtual machine [6].

6 Summary

A good design is the basis of every good project, this is particularly true when design information should be used to verify and adapt a running system without stopping it. In this paper we have shown the role of design information in the adaptation of software systems at run-time. The evolution is managed by a reflective middleware that reifies and, when events occur, manipulates system design information (the system meta-data).

Manipulation is realized by Ruby scripts that drive both the adaptation plan and the consistency checks. We have chosen to write evolutionary and consistency rules as Ruby scripts rather than as logic formulas because: they are expressive as well as (maybe more than) logic formulas; (ii) we don't have to write an interpreter for the rules; finally (iii) they can directly interact with the base-level representation through the reflective facilities of Ruby without extra efforts.

The paper focuses on a case study but it should be clear that the approach is general and usable to evolve most of the software systems. One problem still open with this approach is related to writing the rules driving the whole evolutionary mechanism: they are not simple and require a deepen knowledge of the system to be adapted. Moreover, not always the required changes in the design can be automatically mapped in changes to the code, this could limit the applicability or could require to manually generate the code for the system adaptation to be used by the middleware. In the future we will address these issues.

References

[1] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Object Technology Series. Addison-Wesley, Reading, Massachusetts, third edition, February 1999.

[2] Walter Cazzola, James O. Coplien, Ahmed Ghoneim, and Gunter Saake. Framework Patterns for the Evolution of Nonstoppable Software Systems. In Pavel Hruby and Kristian Elof Søresen, editors, *Proceedings of the 1st Nordic Conference on Pattern Languages of Programs (VikingPLoP'02)*, pages 35–54, Højstrupgård, Helsingør, Denmark, on 20th-22nd of September 2002. Microsoft Business Solutions.

[3] Walter Cazzola, Ahmed Ghoneim, and Gunter Saake. Software Evolution through Dynamic Adaptation of Its OO Design. In Hans-Dieter Ehrich, John-Jules Meyer, and Mark D. Ryan, editors, *Objects, Agents and Features: Structuring Mechanisms for Contemporary Software*, Lecture Notes in Computer Science 2975, pages 69–84. Springer-Verlag, Heidelberg, Germany, July 2004.

[4] Jernej Kovse and Theo Härder. Generic XMI-Based UML Model Transformations. In Zohra Bellahséne, Dilip Patel, and Colette Rolland, editors, *Proceedings of the 8th International Conference on Object-Oriented Information Systems (OOIS'02)*, LNCS 2425, pages 192–198, Montpellier, France, September 2002. Springer-Verlag.

[5] OMG. OMG-XML Metadata Interchange (XMI) Specification, v1.2. OMG Modeling and Metadata Specifications available at `http://www.omg.org`, January 2002.

[6] Dirk Riehle, Steven Fraleigh, Dirk Bucka-Lassen, and Nosa Omorogbe. The Architecture of a UML Virtual Machine. In Linda Northrop and John Vlissides, editors, *Proceedings of the 2001 Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'01)*, pages 327–341, Tampa Bay, Florida, USA, October 2001. ACM Press.

[7] Andrea Savigni, Filippo Cunsolo, Daniela Micucci, and Francesco Tisato. ESCORT: Towards Integration in Intersection Control. In *Proceedings of Rome Jubilee 2000 Conference (Workshop on the International Foundation for Production Research (IFPR) on Management of Industrial Logistic Systems – 8th Meeting of the Euro Working Group Transportation - EWGT)*, Roma, Italy, September 2000.

[8] David Thomas and Andrew Hunt. *Programming Ruby: The Pragmatic Programmer's Guide*. Addison-Wesley, 2001.