Modelling the Asynchronous Dynamic Evolution of Architectural Types*

Cristóbal Costa-Soria[1], Reiko Heckel[2]

[1] Dept. of Information Systems and Computation, Universidad Politécnica de Valencia, Spain
[2] Department of Computer Science, University of Leicester, UK

ccosta@dsic.upv.es, reiko@mcs.le.ac.uk

Abstract. Self-adaptability is a feature that has been proposed to deal with the increasing management and maintenance efforts required by large software systems. However this feature is not enough to deal with the longevity usually these systems exhibit. Although self-adaptive systems allow the adaptation or reorganization of the system structure, they generally do not allow introducing unforeseen changes at runtime. This issue is tackled by dynamic evolution. However, its support in distributed contexts, like self-organizing systems, is challenging: these systems have a degree of autonomy which requires asynchronous management. This paper proposes the use of asynchronous dynamic evolution, where both types and instances evolve dynamically at different rates, while preserving: (i) type-conformance of instances, and (ii) the order of type evolutions. This paper describes the semantics for supporting the asynchronous evolution of architectural types (ie. types that define a software architecture). The semantics is illustrated with PRISMA architecture specifications and is formalized by using typed graph transformations.

Key Words: dynamic evolution, asynchronous updates, software architecture, typed graph transformations.

1. Introduction

A well-known property of current and future software systems is their increasing scale and complexity [42]. Larger size and complexity requires more management and maintenance efforts, which increase the costs of such systems [39]. In order to alleviate the management issues, self-adaptability is being proposed as a feasible solution [23,24].

Self-adaptability is the ability of a system to manage itself at runtime, that is, to adapt its structure or organization in response to changing operating conditions or user requirements. There are two main engineering approaches realizing self-adaptability, which differ on how the interaction patterns of the system are managed. On the one hand, top-down self-adaptable approaches rely on a centralized representation/model, which describes the relations among the different elements of the system. Self-adaptability is guided by this model: (global) decisions are taken upon this model and changes are applied on the system globally. An example of top-down approaches are self-adaptive systems [12,32], which are based on an architecture model and a set of (high-level) goals to guide the adaptation process. On the other hand, bottom-up approaches are fully decentralized: interactions are managed locally by the elements of the system. Thus, self-adaptability emerges from the local adaptation decisions taken by each component. An example of bottom-up approaches are self-organizing systems [38,41], which are based on algorithmic

* This work has been partially supported by the Spanish Department of Science and Technology under the National Program for Research, Development and Innovation project MULTIPLE (TIN2009-13838), and by the Conselleria d'Educació i Ciència (Generalitat Valenciana) under the contract BFPI06/227.

The original publication is available at www.springerlink.com

functions to guide the (local) adaptation process. These systems, usually of a distributed nature, are composed of several autonomous instances, which run concurrently and organize themselves according to different criteria. However, although self-adaptability can support the management of large software systems, it is not enough to deal with another property: their longevity. Due to their high development costs and mission critical nature, many systems are being used for a long period and with very little time for maintenance. Unforeseen maintenance operations may be required by technology changes, new requirements or necessary corrective measures. Although self-adaptive and self-organizing systems allow the adaptation or reorganization of the system structure, they will not generally allow to introduce new functionality at runtime.

What happens if the behaviour of the components, or the reorganization algorithms, need to be changed without the system being stopped? Then, dynamic evolution is required. Dynamic evolution enables the modification of running software artifacts, thus allowing the introduction of new, non-predicted changes. However, the support for dynamic evolution in distributed contexts, like self-organizing systems, is challenging. Due to the autonomous nature of self-organizing instances, their dynamic evolution cannot be performed synchronously with respect to other instances. That is, while one instance may be able to accommodate new changes, another one may not (due to various constraints, e.g. it might be disconnected). In this context, it is difficult and time-costly to stop (and synchronize) all the distributed entities to introduce a new update.

The solution we propose is the use of *asynchronous dynamic evolution*. Dynamic asynchronous evolution is a feature that allows introducing changes in a running system *concurrently*, but maintaining the order of changes. It allows any type of the system and its instances to evolve at different times. Then, a type may be dynamically updated several times, while each one of its instances: (i) remains out of date, (ii) continues evolving to a previous version, or (iii) starts evolving to the last version. Asynchronous evolution is useful for those situations where dynamic evolutions are required, but synchronisation is not possible and evolutions should be postponed. For instance, this is the case for distributed and/or autonomous systems, because they are composed of entities that sometimes may be inaccessible (i.e. distributed systems) or busy (autonomous systems). Furthermore, asynchronous evolution is also useful for building systems with a dynamic nature, that is, with a high rate of dynamic changes. For instance, systems that are incrementally or collaboratively built at runtime, like dynamic web systems, require to concurrently introduce new changes on the system specification while its instances are still running or evolving to previous versions.

In this paper we introduce a semantic model for asynchronous dynamic evolution of software architectures. The contribution of this paper is twofold: (i) a general and abstract model of asynchronous evolution; and (ii) its application to the evolution of types in PRISMA architectural specifications. Specifically, asynchronous evolution is modelled by the definition of patterns of changes in terms of typed graph transformations [17,21]. This paper describes the approach, but due to lack of space does not contain the complete model. The approach is illustrated by its application to the dynamic evolution of PRISMA architecture specifications (i.e. the type) and its configurations (i.e. its instances).

This paper is organized as follows. Next section introduces the asynchronous evolution of types, the differences with synchronous evolution, and the challenges it poses. Section 3 describes PRISMA architectural types, which are used to illustrate our approach. Section 4 introduces the formalization of asynchronous evolution of architectural types by using graph transformations. Then, sections 5 and 6 discuss our approach and related work. Finally, section 7 presents the conclusion and further works.

2. Asynchronous Evolution of Types

In order to describe what dynamic asynchronous evolution is, first the concept of types and instances should be introduced. A type is an abstract concept which defines the structure (i.e. the

The original publication is available at www.springerlink.com

state) and behaviour (i.e. how this state is modified) of a software artifact. A type is comprised of two elements: a specification, which is the high-level description of a software artifact, and an executable code, which is the realization of this software artifact (and the implementation of the specification). In a Model-Driven-Development approach [6], the executable code is automatically generated from a (possibly partial formal) specification, which reflects how both elements are closely related. The executable code allows the creation and execution of different instances of the software artifact. An instance is the execution of a type on a concrete platform: it behaves as defined by the type specification, and it is characterized by an internal state (i.e. the data stored in the instance), which is different from other instances.

When a type, that is deployed and active in a software system (i.e. it has been instantiated), needs to be changed or updated, *dynamic software evolution* [11,40] (or online software evolution [48]) is used. Dynamic software evolution is a feature that allows changing a type without the need to shut down the system. This is performed by the following evolution process: (1) the modification of the current type specification; (2) the update or regeneration of the executable code of the type, so that instances could be created according to the updated type; and (3) the evolution or migration of the running, stateful instances to integrate the changes. The last step is the longest, because it entails the quiescence (i.e. the safe stopping) of all the instances that are going to evolve. This quiescent [25] or tranquil [47] status[†] of an instance guarantees that there are no pending or running transactions which could be affected by the evolution process.

The dynamic evolution process can be synchronous or asynchronous, depending on how a type and its instances evolve with respect to each other. In *dynamic synchronous evolution*, a type and its instances are evolved sequentially, i.e., the evolution of a type is followed immediately by the update of its instances before the type can evolve any further. In *dynamic asynchronous evolution*, a type and its instances are evolved independently. The type may evolve again before all of its instances have applied the previous changes. Asynchronous evolution advantages synchronous evolution when evolution requests are frequent and/or when instances are highly distributed or partly reachable. On the one hand, in asynchronous evolution changes can be performed earlier: a type can be evolved as soon as required, without waiting to update all of its instances. This is useful for developing highly flexible systems, i.e. those systems with a high probability of changes. On the other hand, changes can be propagated and applied asynchronously, without requiring the instances to be permanently reachable. This is particularly useful for supporting dynamic changes in distributed systems and/or mobile systems.

In order to better understand the implications of synchronous and asynchronous evolution approaches, we need to detail how they manage the execution of multiple evolution processes. But first let us introduce the concepts of type version, activeness, and evolution process.

The different evolutions of a type over time are reflected by means of *type versions*. Each time a type is evolved, a new version is created/generated. This version contains the new (evolved) type specification and the executable code from which new instances will be created. A type version is kept in memory while it is *active*. A version is *active* while: (i) it is the most recent type version, or (ii) it has running instances. The most recent type version is active by default (i.e. loaded in the system) to allow its evolution: only the latest version can be evolved. The previous versions are outdated and are only kept in the system to allow the execution of out-of-date instances. As soon as the out-of-date instances are updated or deleted, then the old type version is not active anymore and can be safely unloaded from the system. The evolution of a type and its instances is performed by an *evolution process*, which: (i) evolves the latest version of the type (generating a new version), and (ii) evolves the instances from the previous version to the new version, preserving their states. In general, the end of an evolution process implies the inactivity of the type version that has been updated, due to the evolution of all of its instances to the new type version.

[†] As introduced by Vandewoude et al. in [47], we will use the distinction between the internal *state* of an element (which is migrated or transformed in the evolution process), and the *status* that describes its condition with respect to the evolution process (idle, active, quiescing, passive, ...)

The original publication is available at www.springerlink.com

Fig. 1 describes visually the execution of several evolution processes and how they are managed in synchronous (Fig. 1(a)) and asynchronous (Fig. 1(b)) approaches. This figure shows the evolution of a type, called *T*, and the evolution of its instances, identified by numbers *1*, *2*, *3*. In Fig. 1 the different versions of the type *T* (i.e. its evolutions) are depicted as squares (T_0, T_1, T_2 and T_3). The position of each square represents the time when the corresponding version was introduced in the system. An adjacent, diamond-ended, dotted line represents the lifetime of a version (i.e. the time that a version is active in the system). Circles with identical number represent instantiations or evolutions of an instance. The vertical position of each circle depicts the version that the instance is conformant to, whereas the horizontal position depicts time: (i) the moment when the instance was created; or (ii) the moment when the instance evolved from a previous version (the instance had a state which has been migrated to the new version). However, the figure does not describe the reason why an instance delays its evolution: this may be due to that: (i) the instance has not yet received the evolution request (i.e. in distributed systems), or (ii) the instance is waiting to reach a quiescent status. Finally, solid round-ended lines (see Fig. 1) depict evolution processes, which start with the generation of a new version and end when all the instances of the old version have been evolved to the new version.

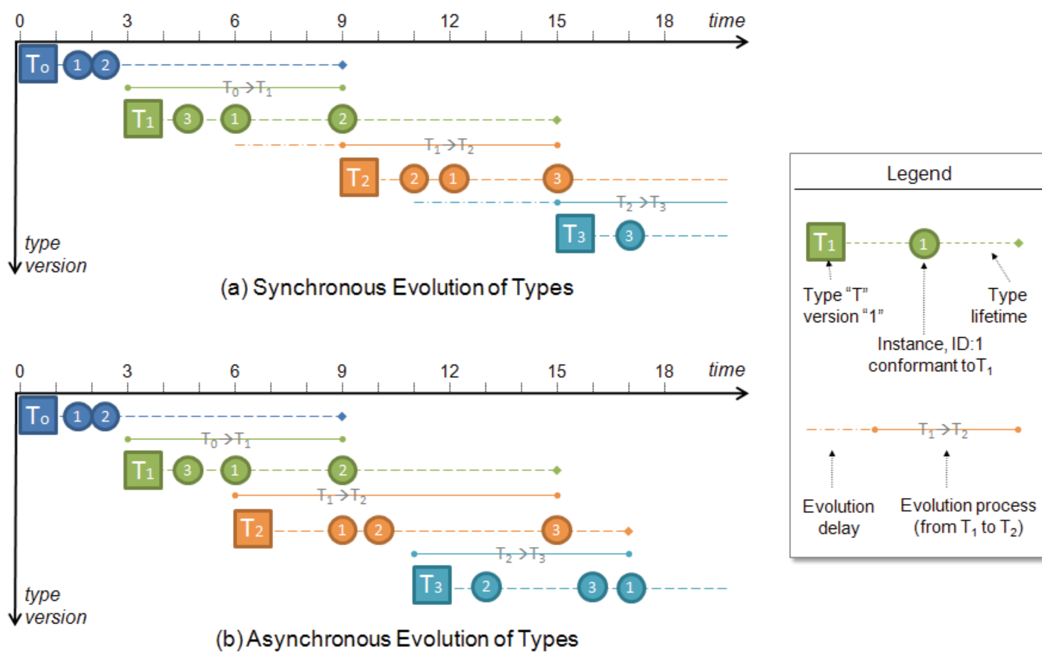

Fig. 1. Synchronous vs Asyncronous Evolution

Fig. 1(a) shows the dynamic evolution of the type *T* following a synchronous approach, as implemented in most of current approaches (e.g. [31,48]). In this approach, a new evolution process cannot be started until the completion of the previous evolution process. This delay is depicted in the figure as a dotted line at the beginning of each evolution process. For instance, although the evolution request "$T_1 \rightarrow T_2$" have been received at time instant 6 (see Fig. 1(a), 2nd evolution process), the evolution process cannot start (i.e. it cannot generate the new version, T_2) until time instant 9, when the previous evolution process "$T_0 \rightarrow T_1$" finishes the evolution of the last instance of T_0, identified as *2*. As it can be seen from the figure, the main disadvantage of synchronous evolution is the time that is needed until a new change can be introduced in the system. This may be an important drawback when evolving either distributed or mobile systems, where network fluctuations and/or the reachability of the instances (e.g. some may be

The original publication is available at www.springerlink.com

disconnected) may increase the time needed to propagate new changes, and thus, the time needed to perform several dynamic changes.

On the other hand, Fig. 1(b) shows the dynamic evolution of the type *T* following an asynchronous approach (e.g. [37]). In an asynchronous evolution approach, type and instances evolve at different times. This allows evolution processes to overlap their execution: as soon as a type has been evolved, it can be evolved again, although some of its instances may still be applying the previous changes. Several examples of overlapping evolution processes are shown in Fig. 1(b). For instance, at time instant 6 the evolution process "$T_0 \rightarrow T_1$" is still active (see the 1st evolution process, started at instant 3), because the instance *2* is still pending to evolve T_0 to T_1. However, as soon as the change request "$T_1 \rightarrow T_2$" is received (see the 2nd evolution process), it is immediately executed: the version T_1 is evolved despite there are instances pending to evolve to this version (i.e. the instance *2*). Another example of overlapping evolution processes (and type versions) is shown at time instant 13, where there are three active type versions: T_1, T_2 and T_3. These versions provide the behaviour of the instances *3*, *1*, and *2* respectively. Two pending evolution processes, "$T_1 \rightarrow T_2$" and "$T_2 \rightarrow T_3$", are still active: the former to evolve the instance *3* to version T_2, and the latter to evolve the instance *1* to version T_3. Each instance can evolve independently of the other instances, without waiting for the other instances to finish the previous evolution process(es). As soon as an instance is ready to evolve to the next type version, it does, although other instances remain in *k*-previous versions.

That is, while in synchronous approaches only one version and one evolution process is active, in asynchronous approaches several versions and evolution processes can coexist at the same time. However, this does not necessarily mean that several evolution paths (i.e. version branching) are allowed. Our approach is focused on an incremental evolution setting: the set of changes performed at time *t* (both at type-level and instance-level) is performed on the result of changes performed at time *t-1*. At the end, all the instances must integrate all the sequence of changes introduced over time. In order to do this, only a single evolution path must be allowed. Only in this case we are able to determine the correct version an instance must be evolved to. A single evolution path can be guaranteed by constraining the evolution to only the latest type version, protecting it from concurrent evolution requests. Thus, evolution processes are sequenced: a new evolution process cannot be started until the previous one has generated at least the new type version (it makes no sense to evolve something that has not been generated yet). And an evolution process cannot finish until the end of the previous evolution process. This is because the previous evolution process may still be evolving instances, which must be also evolved by the newer evolution process in order to integrate the newer changes.

Asynchronous evolution takes a step forward from synchronous evolution: it allows us to introduce multiple (but ordered) change requests, deferring them until they can be effectively applied. For instance, this will allow different stakeholders to update different parts (i.e. types) of a system at different times, without taking into account the update of the running (and perhaps distributed) instances. This update will be carried out by the evolution infrastructure.

However, the consequence of this is that asynchronous evolution is also the most challenging. Most of related works have been focused on the challenges posed by synchronous evolution: the adaptation of runtime data structures and code [27,31,40], the state consistency before and after a dynamic change [25,47], or the migration of the state [37,46]. Asynchronous evolution builds on the features provided by synchronous evolution, but adds new features (asynchrony) which in turn poses new challenges:

- *Type conformance*. Since types evolve at different rates with respect to their instances, it is then difficult to check if the instance-level is conformant to the type-level.
- *Version management*. Different evolutions of a single type entail the existence and management of different versions of this type at runtime (at least where there are running instances of such type), which adds more complexity to the process.
- *Order of evolution processes*. In such a case where a type could require a high rate of evolutions, it could happen that an instance might have two or more pending evolutions.

In this case, the order of pending evolutions must be preserved. This is important in distributed systems, where instances may receive newest evolution changes before the older ones.

- *Coherence of interactions*. Interactions among instances that are in different versions can produce incorrect behaviours. The context where instances evolve is also important.

These challenges are addressed in Section 4. Next we introduce the context of this work, which will allow us to illustrate how we have modelled asynchronous evolution.

3. PRISMA Architectural Types

Software Architecture [36,44] provides techniques for (i) describing the structure (or architecture) of complex software systems (i.e. the key system elements and their organization), and (ii) reflecting the rationale behind the system design. The structure of a software system is mainly described in terms of architectural elements (i.e. components and connectors) and their interactions with each other. This structure is formally specified using an Architecture Description Language (ADL), which is used later to build the executable code of the software system.

Our work is focused on supporting the dynamic evolution of this structure: evolving both the formal system description (i.e. the ADL specification) and its executable code. Among the different formal ADLs from the literature [28], we selected the PRISMA ADL [33,34] because of the benefits it provides for supporting dynamic evolution. First, the PRISMA language allows modelling the functional decomposition of a system (by using architectural elements), and the system's crosscutting concerns (by using aspects), which results in more simple, clear, and concise system specifications. This allows us to separate parts of the software that exhibit different rates of change, and evolve only the interesting parts [29]. Second, PRISMA does not only allow modelling the structure (i.e. the architecture) of a system, but also allows describing precisely the internal behaviour of each architectural element. The behaviour is specified by using a modal logic of actions [43] (for describing services) and π-calculus [30] (for describing interactions among services); see [35] for more details. Thus, since the internal behaviour is formally described, this allows us to automatically interleave the actions required to perform the runtime evolution of its instances: (i) actions to achieve quiescence, and (ii) actions to perform the state migration. Lastly, the PRISMA ADL is supported by a Model-Driven Development framework [6], which allows the automatic generation of executable code from PRISMA models/specifications. This also benefits the support for dynamic evolution. The code generation templates can include not only the code for supporting the runtime evolution of the system, but also the code to reflect the changes on the formal system specification, keeping both in sync. In particular, in this paper we cover the semantics for supporting the asynchronous dynamic evolution of PRISMA architectures. For this reason, next we introduce the main concepts of the PRISMA ADL and a small example, which is used later to illustrate our approach.

The PRISMA ADL defines the architectural elements of a software system at different levels of abstraction: the type definition level and the configuration level. The type definition level defines architectural types, which are instantiated in specific architectures or are reused by other architectural types. The configuration level defines the architecture of a concrete software system, by creating and connecting instances of the architectural types defined at the type definition level. In other words, the configuration level specifies the topology of a specific architectural instance. This separation among the type level and the configuration level allows to easily differentiate-and implement-changes in a type and changes in a configuration (i.e. an *architectural* instance).

An architecture is defined at the type-level as a pattern, so that it can be reused in any other system or architectural type. The architectural type that defines this pattern is called *System*[‡]. A

[‡] In order to avoid confusions, we will use capitalized letters when referring to concepts of the PRISMA metamodel (e.g. System and Configuration).

The original publication is available at www.springerlink.com

System can be used in other architectural types as a single unit, and be treated like other simple architectural types (i.e. components and connectors). This allows PRISMA to support the compositionality, or hierarchical composition, of its architectural elements: the architecture of a complex software system can be described as a composition of several architectural elements which, in turn, can be described as the composition of other architectural elements. Thus, a complex system can be recursively defined as an architecture of architectures, because each composition describes an architecture.

The pattern of a System defines: (i) a set of ports for communicating with its environment (or with other architectural elements); (ii) the set of architectural types it is composed of and the number of instances that can be created of each type; and (iii) the set of valid connections among the architectural types and the number of connections allowed. A port defines a point of interaction among architectural elements; it publishes an interface with a set of provided and/or required services. A connection can be of two kinds: an Attachment, if it links two architectural elements; or a Binding, if it links an (internal) architectural element with one of the ports of the system (i.e. allowing the communication with external architectural elements). For instance, Fig. 2-top shows a System called *Sys*. It consists of two architectural element types, *A* and *B*, with a cardinality of *1..1* and *1..n*, respectively. These types are connected to each other by an Attachment called *Att_AB* with a cardinality *1..1* and *1..n*. These cardinalities mean that only one instance of A is allowed, which can be connected to several instances of B. Finally, *Sys* interacts with its environment by means of the port *p1*. The behaviour of this port is provided by the architectural type *A*, which is connected by means of the Binding called *Bin_p1A*.

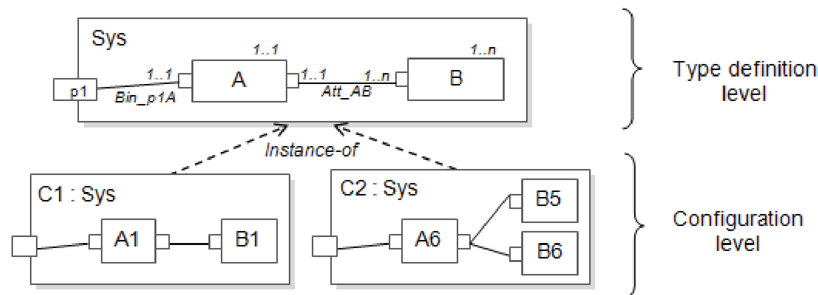

Fig. 2. Example of a PRISMA System and two Configurations

```
System Sys

  Ports
    P1 : interface1;
  End_Ports;

  Import Architectural Elements
    A:TA(1,1), B:TB(1,n);

  Attachments
    Att_AB: A.PServ(1,1) <--> B.PServ(1,n);
  End_Attachments;

  Bindings
    Bin_p1A: P1(1,1) <--> A.PClient(1,1);
  End_Bindings;

  New() { /* Constructor definition */ }
  Destroy() { /* Destructor definition */ }

End_System Sys;
```

(a) Specification of the System Sys

```
Architectural_Model_Configuration C1 =
   new Sys {
     A1 = new A();
     B1 = new B();

     att_A1B1 = new Att_AB(A1, B1);
     bin_A1 = new Bin_p1A(A1);
   }

Architectural_Model_Configuration C2 =
   new Sys {
     A6 = new A();
     B5 = new B();
     B6 = new B();

     att_A6-B5 = new Att_AB(A6, B5);
     att_A6-B6 = new Att_AB(A6, B6);
     bin_A6 = new Bin_p1A(A6);
   }
```

(b) Specification of the Configurations C1 and C2

Fig. 3. Example of PRISMA ADL specifications

The original publication is available at www.springerlink.com

The instantiation of a System is defined at the configuration-level, and is called *Configuration*. A Configuration instantiates a concrete architecture from the different combinations allowed by the pattern: it instantiates each of the architectural types defined in the pattern and connects them appropriately. For instance, Fig. 2-bottom shows two Configurations, *C1* and *C2*, of the System *Sys*. The PRISMA ADL specification of this example is shown in Fig. 3.

In order to later illustrate the asynchronous evolution process, we will use this example, assuming that these two Configurations, *C1* and *C2*, are instantiated (and running) in different parts of a large software system.

4. Formalization of the Evolution Process

Dynamic evolution is not a feature of a software system, but a feature of *its* architectural types. The reason is that a software system is composed of several elements, probably heterogeneous, and not all of them require dynamic change support. Some of them may be evolvable whereas others may not. If a type is evolvable, it requires a specific infrastructure to support its dynamic evolution, which is independent of the other types of the system. The structure of a type (i.e. its state space) is different from other types, so different (state-migration) functions will be required to evolve the instances of each type. In our approach, evolvable types are generated with the required evolution infrastructure. We are not going to describe the details of such infrastructure, since it has been covered in other works [13,14]. Here we describe the semantics of the asynchronous dynamic evolution process performed by this evolution infrastructure and how to cope with the problems it poses.

An (asynchronous) dynamic evolution process is triggered when a meta-service called *Reflection,* provided by each evolvable architectural type, is invoked. This service requires a new specification for the type to evolve, which is described in terms of modifications on the current type specification[§]: additions, deletions or updates. An optional parameter allows specifying a set of instances to exclude from the evolution process. This is useful in situations where it is preferable to not evolve an instance but destroy it when it is not longer needed.

The set of modifications that can be performed on an architectural type is defined by its metamodel (recall that a metamodel describes how an architectural type is defined). According to the PRISMA metamodel [35] and the definition of the System element, we have defined 13 evolution operations for System architectural types (see Fig. 4). Each operation involves runtime changes both at type-level and instance-level.

```
AddArchitecturalElement(aeName, type, minCard, maxCard)
AddAttachmentType(attName, sourceAE, sourcePort, srcMinCard, srcMaxCard, targetAE, targetPort, trgMinCard, trgMaxCard)
AddBindingType(bindName, srcPort, targetAE, trgMinCard, trgMaxCard)
AddPort(portName, interface, [playedRole])
RemoveArchitecturalElement(aeName)
RemoveAttachmentType(attName)
RemoveBindingType(bindName)
RemovePort(portName)
UpdateArchitecturalElement(aeName, newType, [newMinCard, newMaxCard])
UpdateAttachmentType(attName, srcMinCard, srcMaxCard, trgMinCard, trgMaxCard)
UpdateBindingType(bindName, trgMinCard, trgMaxCard)
```

Fig. 4. PRISMA evolution operations for System types

§ There is another meta-service, called *Reify*, which provides –at runtime- the current type specification, so it can be evolved.

The original publication is available at www.springerlink.com

For instance, Fig. 5 shows two evolution processes which modify the System *Sys* described previously in Fig. 2. The first evolution process, started in time *t1*, introduces a new architectural type *C*, with a minimum and maximum cardinality of 1 and n, respectively. This type is attached (attachment type called *'A-C'*) to the port *'pA'* of type *A*, with these cardinalities: C→A[1..1], A→C[1..n]. This evolution process also removes the type *B* and the attachment from *A* to *B*. The second evolution process, started in *t5*, introduces a new type *D* that is attached to the type *C*, added in the previous evolution process.

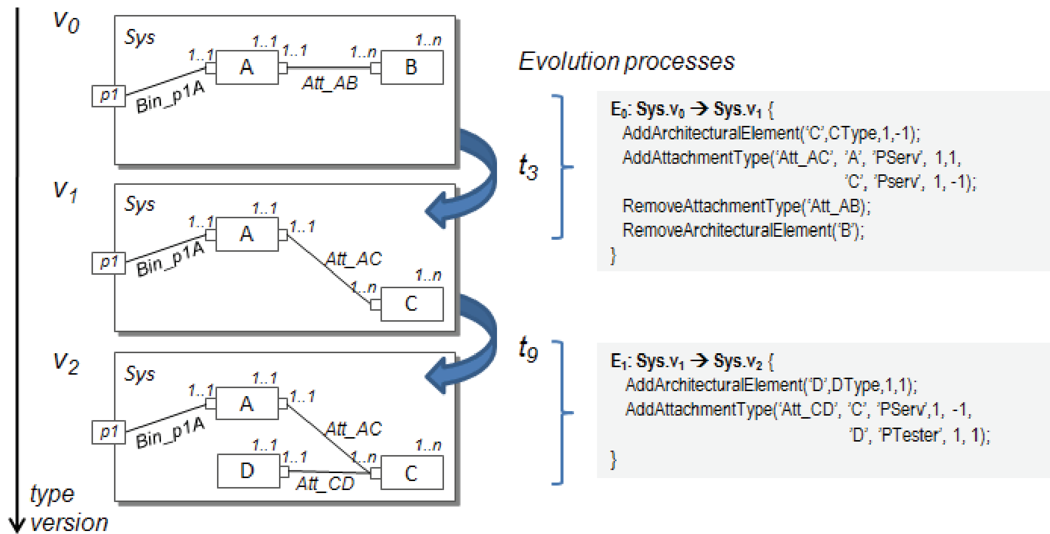

Fig. 5. Example of two evolution processes

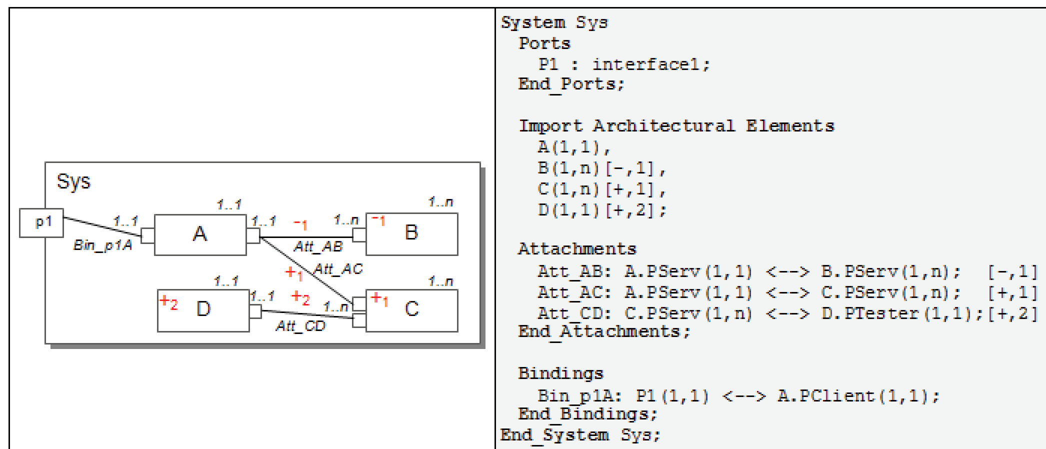

Fig. 6. Example of an architectural type with evolution tags

The successful execution of each evolution process creates a new version (i.e. a new specification) of the architectural type being evolved. Type versions are identified from each other by means of a version number, which is increased each time an evolution process is applied on the type successfully. Each instance also contains an attribute which keeps track of the type version that it is currently an instance of. This is used to control that only the evolution operations leading to the next type version (from the instance perspective) are taken into account, thus preserving the order of evolutions.

The original publication is available at www.springerlink.com

In our approach, only one single type specification is needed to describe both the current and previous type versions. This is possible because each change that is performed on an architectural type is also reflected on its specification, by means of evolution tags. An evolution tag is linked to one of the elements a type specification consists of (e.g. in the case of a System: architectural elements, connections and ports), and describes: (i) the kind of evolution operation performed (i.e. addition, removal or update), and (ii) the type version where the evolution operation took place. Fig. 6 shows an example of such a tagged specification, which describes the different type versions produced by the evolution processes presented in Fig. 5. For instance, the tag *[+,1]* near the declaration of the type *C* in the specification of System *Sys* (see Fig. 6, right) tells us that the type *C* has been imported in version 1 of *Sys*.

With such a tagged type specification, the specification of a type *t* at version *v* can be obtained applying this function:

$$Spec_t(v) = Upd_t\big(\{(Spec_t(v_0) \cup Add_t(v_0, v)) \setminus Rem_t(v_0, v)\},\ v_0,\ v\big) \quad (1)$$

This function builds a set with: (i) the elements of the first version of the type *t* (i.e. before the execution of any evolution operation): $Spec_t(v_0)$; and (ii) all the elements that have been added to the type *t* since the first version, v_0, until version *v*: $Add_t(v_0,v)$. Then, from this set are excluded all the elements that have been removed from the type *t* since the first version, v_0, until version *v*: $Rem_t(v_0,v)$. Finally, from the last resulting set are replaced the elements that have been updated since the first version, v_0, until version *v*: $Upd_t\,(Spec_t,v_0,v)$. Then, an instance *i*, whose version is v_i, will be *conformant with a type t* if its structure satisfies the specification returned by the function $Spec_t\,(v_i)$ defined above.

4.1 Describing the Semantics of Evolution

As described above, an evolution process comprises the execution of several evolution operations; each evolution operation impacts one of the elements an architectural type consists of. The complete set of evolution operations of System types has been formalised by means of typed graph transformations [21]. Graph transformations combine the idea of graphs as a modelling paradigm with a rule-based approach to specify the evolution of graphs. They are supported by an established mathematical theory and a variety of tools for its execution and analysis [17]. The main reasons to choose typed graph transformations as the basis for the formalisation are the following: (i) software architectures can be easily formalised as graphs, as shown in other works [22,49]; (ii) graph transformations are asynchronous, i.e., each rule can be applied once its preconditions are satisfied, which benefits the formalisation of asynchronous evolution; and (iii) typed graphs capture the relation among types and instances, required to model the evolution both at the type-level and instance-level.

In this work, typed graph transformations have been used to describe the *observed* behaviour of evolution operations: i.e. how a System type and its instances change as the execution of evolution operations. These graph transformations are presented using an architecture-based concrete syntax, which is more concise and easier to understand than the graph-based abstract syntax. Instead of using only vertices and nodes (as provided by the graph-based abstract syntax), a concrete syntax allows describing the same behaviour using concepts closer to the area of software architectures (i.e. components, ports, attachments, etc.), but extended with concepts required to deal with asynchronous evolution. The mapping of this architecture-based concrete syntax to the graph-based abstract syntax will be described later in section 4.2.

The semantics description of the entire set of evolution operations (11 in total, see Fig. 4) has produced 43 transformation rules, concerned with the evolution management of the type-level, the instance-level or both. Due to space reasons, we will only describe the formalization of the operations involved with the addition of a new architectural type to the composition of a System, and with the update of an architectural type. We have chosen these operations because they are

The original publication is available at www.springerlink.com

enough to illustrate how the different challenges described in section 2 are addressed by their respective transformation rules. Each rule is self-contained and can be understood without requiring the description of the complete set of rules.

4.1.1 Addition of a new Architectural Type

The addition of a new architectural type to a System is started by the execution of the *AddArchitecturalElement* operation. This operation modifies the composition of a System type to add a new, unforeseen, architectural type at runtime. This operation requires four parameters: *AEName*, the alias of the type to add; *type*, the executable code of the type to add; *minCard*, the minimum number of instances of this type that must exist in each system instance; and *maxCard*, the maximum number of instances that can exist in each system instance (i.e. in each Configuration). Its semantics is described by the rule R1 (see Fig. 7), which acts at the type-level. The left hand side of the rule (see Fig. 7, left side) specifies the condition(s) that must be satisfied to execute such rule: the type to add (i.e. *AEName*) must not already be present in the pattern defined by the System type. The System type to modify is represented as the top-left component named 'S', and the set of its instances as a multiobject (i.e. a collection of elements) named 'S1' at the bottom-left. The right hand side of the rule (see Fig. 7, right side) specifies the consequence of applying the rule: the pattern of the System 'S' has been modified to include a new architectural type, identified by *AEName*. This new type can be instantiated in each one of the System instances, but only while satisfying the *minCard* and *maxCard* constraints.

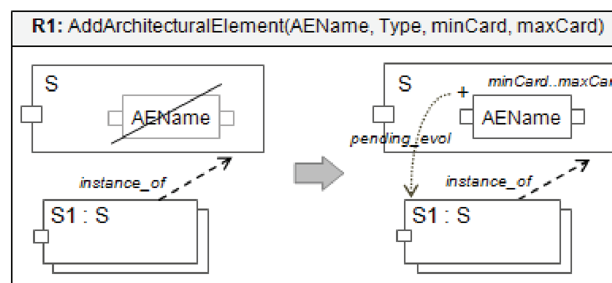

Fig. 7. Rule R1 - AddArchitecturalElement

As described before, a System type keeps track of the changes performed on it, in order to describe its evolution over time and to control the asynchronous evolution of its instances. This is also described in the transformation rules. When the rule R1 is executed, a new *add* tag (represented with the symbol '+') is created and attached to the new architectural element (see Fig. 7, right side). This tag is initialized with the current version number of the System type (whenever a new evolution process is started, the version number of the type is increased; the new evolved type is identified by this version number). This tag is also provided with a link to every instance of the type that must be evolved to the new type specification (remember that we can exclude instances from new evolutions). This relationship is reflected in R1 with the 'pending_evol' link to the set of instances. The use of this link will be shown later.

Changes at the instance-level are performed by reconfiguration operations. A reconfiguration operation can change the configuration (i.e. the topology) of a System instance (those where the service has been invoked), but only while conforming to the pattern defined in its System type. In addition, if a System instance has pending evolutions to apply, a reconfiguration operation also constraints the set of allowed changes in order to converge to the new type. For instance, if a type has been removed from the System pattern, the reconfiguration operation *CreateArchitectural Element* will not allow creating instances of such type. In order to carry out these checkings, a reconfiguration operation takes into account the system instance current version and the set of

The original publication is available at www.springerlink.com

evolution tags corresponding to the instance version + 1 (i.e. to check valid type conformance and to lead the instance convergence to the next type version).

Instances of architectural elements are created by invoking the reconfiguration operation *Create ArchitecturalElement*. This service requires three parameters: *AEType*, the name of the architectural type which to create an instance; *id*, a unique identifier of the instance to create; and *params*, the parameters required to create an instance of the type *AEType*. The semantics of this reconfiguration service is described by the rule R2 (see Fig. 8). In this rule, *S1* is a Configuration (i.e. an instance of a System), and its type is the System *S*.

The left-hand side of the rule describes which conditions must be satisfied to allow the instantiation of an architectural type, *AEType*, in the Configuration *S1* (the context where the reconfiguration service has been invoked). These conditions are the following:

(i) The architectural type *AEType* must be defined in the type of *S1*: *AEType* exists in the pattern defined by the System *S* (see Fig. 8, top, in the box called *S*)

(ii) The instance *S1* cannot have another instance of *AEType* with the same identifier *id*.

(iii) The number of instances of *AEType* created in the Configuration *S1* is lower than the maximum cardinality *maxCard* defined in the System *S* (see Fig. 8, condition 1).

(iv) The type *AEType* has not been removed from the System *S* (i.e. it has not been tagged for deletion, see Fig. 8, cond. 2, first part). Or, if it has been removed, it has not been done in the subsequent evolution of the System type (i.e. it has been removed in a version later than the Configuration version + 1, see Fig. 8, condition 2, second part). Thus, the rule takes into account not only the conformance of a Configuration to its System type (i.e. same version), but also its convergence to the subsequent evolution of its System type.

(v) If the type *AEType* has been added at runtime (i.e. it is tagged with an addition tag), it has been done in previous versions of the System type (i.e. the version of this addition tag is less than or equal to the Configuration version, see Fig. 8, condition 3) or in the subsequent evolution of the System type (i.e. the version of the *AEType* addition tag is equal to the Configuration version + 1). In other words, the type *AEType* cannot be instantiated if it has been added in future versions of the System type (with respect to the Configuration perspective). Thus, the order of runtime type evolutions is preserved.

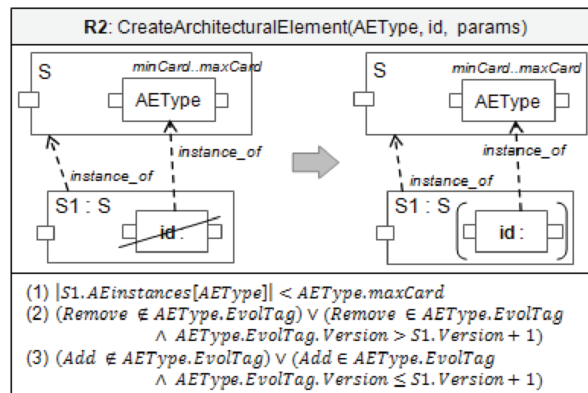

Fig. 8. Rule R2 - CreateArchitecturalElement

The right-hand side of the rule describes how the *AEType* type is instantiated in a Configuration (a System instance). Each new instance created is left in a quiescent state [25] by default, i.e. it cannot start or process any transaction. This is represented by using the symbols '[' and ']' around the software artifact being quiescent (see Fig. 8, right-hand side). The creation of instances stopped by default is to guarantee that the minimum cardinality constraint defined in the System type is satisfied first. Thus, several instances can be created, but they will not be started until their number would not be enough.

The activation of the quiescent instances is performed by the rule R3 (see Fig. 9). This rule is automatically triggered when any System instance (depicted as ‘S1’ in the figure) has so many quiescent instances of the type *AEType* as specified in the property *mincard*, defined in the System type *S* (see Fig. 9, condition 1). The conditions 2 and 3 depicted below the rule are only defined to guarantee that the System type version chosen by the rule conforms to the current version of the System instance. The execution of this rule unblocks (i.e. start the execution of) the set of instances of the type *AEType* (note that the symbols ‘[’ and ‘]’ have been removed from the right hand side).

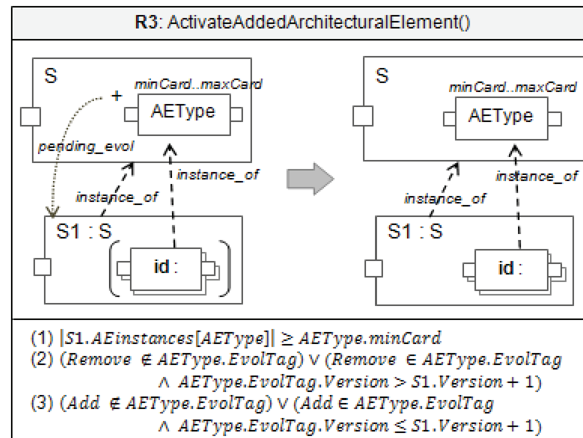

Fig. 9. Rule R3 – ActivateAddedArchitecturalElement

Another effect of the execution of the rule is that it removes the link ‘pending_evol’ among the addition tag (depicted as ‘+’ in the rule) and the System instance *S1* where the rule has been applied. This means that the System instance *S1* has integrated the new type *AEType* as specified by the System type *S*, which was introduced in its version *S1.Version+1*. In other words, the System instance now has integrated one of the evolution operations that were pending.

However, since an evolution process is made of several evolution operations, a System instance could not be promoted to a new version of the System type until all the evolution steps would be performed. This is described by the rule R4 (see Fig. 10). The left-hand side of the rule describes the condition that must be satisfied to promote a System instance to (i.e. be conformant with) the next version of the System type: a System instance *S1*, whose current version is *v*, is said to be conformant to the version *v+1* of its System type if no evolution tag with version = *v+1* exists with a ‘pending_evol’ link to the System *S1*.

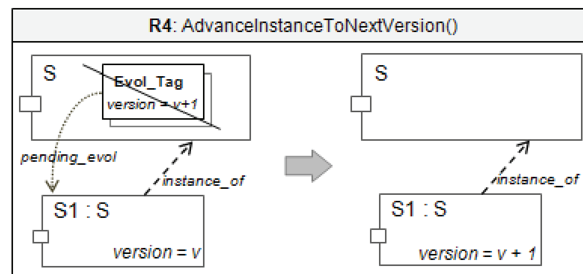

Fig. 10. Rule R4 - AdvanceInstanceToNextVersion

In addition, the ‘pending_evol’ link has an additional use: to monitor the evolution state of each instance at the type-level. By looking the oldest evolution tag (i.e. with the minor version) which is still linked to a System instance, it lets us know: (i) the set of evolution operations the System

The original publication is available at www.springerlink.com

instance is currently involved in, and (ii) the version a System instance currently is conformant to, by decreasing the version provided by the oldest evolution tag found.

4.1.2 Updating of Architectural Types

Finally, in order to fully understand our approach, the update operation is described here. This operation replaces a type version by another and updates its instances, keeping all their existing connections and their internal state. This is formalised by two transformation rules: *UpdateAEType* and *ReplaceAE*, which act at the type-level and the instance-level respectively. The context where these rules are executed are: the System specification that contains the type to update, and the instances (or Configurations) that have imported and instantiated the type to be updated.

On the one hand, the *UpdateAEType* rule models the updating of a type version by a new one, keeping the existing interaction patterns. Let us recall that in the PRISMA model, the type-level defines the interaction patterns among types (i.e. what kind of interactions are allowed), and consequently, the interactions among their instances. When updating a type, the existing interaction patterns must be preserved, in order to guarantee the coherence of interactions at the instance-level: the instances that were connected/attached to the old, updated instance must be able to interact with the new, evolved instance.

The coherence of interactions can be only guaranteed by requiring the new type version to be syntactically and semantically compatible with existing interactions; otherwise, the update operation must not be performed. That is, compatibility evaluation when performing an update is focused on the interactions that the old type has, instead of focusing on the type itself. The reason is that a new type version, despite being incompatible with its previous version, could be semantically compatible with the types that were interacting with the previous version. This is the case when, in the context of an evolution process, the removal of *part of* the original functionality is required: a new version provides less functionality than the previous version (i.e. the new version is not compatible with the old one), and the interactions requiring this functionality have been removed from the initial set of interactions (i.e. the new version is compatible with the resulting set of interactions). If complete compatibility (or subtyping) of the new type with respect to the old type were required, we would never be allowed to remove unused functionality[**].

Since interactions among architectural types are performed through ports (i.e. they are the points of interaction), compatibility for updating is evaluated through ports. Thus, the requirement to perform an update operation is that the new type provides a set of ports syntactically and semantically compatible with the interacting ports of the old type. On the one hand, we define a port p_x as *syntactically compatible* with another port, p_y, if it provides all the services that are required from p_y, which are defined by the set of existing interactions of p_y, with exactly the same signature (i.e. name and parameters) of each service. Note that syntactic compatibility only refers to the minimum set of services *provided* by p_x: this means that a syntactically compatible port may provide additional services or require different services than the original. The goal of the updating operation is to guarantee that the updating does not break the current architecture, without taking care of the introduction of new required services. This is the responsability of other evolution rules, which will evaluate if all the required services are conveniently bound, thus guaranteeing the consistence of the architecture.

On the other hand, we define a port p_x as *semantically compatible* with another, p_y, if it provides the same observable behaviour as other elements expected from p_y. That is, the execution of the services of p_x must produce the same expected results, or sequence of traces, that p_y produces. However, the evaluation of semantic compatibility is challenging and its integration in the evolution model is not trivial. Since this is not the goal of this paper, the reader can refer to other works [18,19] in order to get more details about how some issues have been addressed from a formal perspective. For the sake of simplicity, we have abstracted semantic evaluation this way:

[**] In fact, we could remove unused functionality from a type by removing the old version and adding the new, reduced one. However, in this case, the state migration of its instances would not be performed.

The original publication is available at www.springerlink.com

each port is provided with a function, *CanInteractWith*, which evaluates if another port satisfies a certain contract or interaction protocol, i.e. that the observed behaviour is the required. For instance, in a port that requires compression and decompression services, a very simple function could evaluate that the compression of a sample data is decompressed correctly to the original data. That is, each port has the responsability of validating that their required services are provided correctly. Thus, we could say that a port p_x is *semantically compatible* with another, p_y, if all the ports that are connected to p_y, and request services from p_y, can interact seamlessly with p_x. In case of semantic incompatibility, a factible solution is the use of dynamically generated adaptors [9].

Syntactic and semantic compatibility is reflected in the *UpdateAEType* rule by means of condition (1) (see Fig. 11): the execution of the rule is only performed if the ports of the new type version (i.e. parameter *NewType*) are syntactically (see condition 1.1) and semantically (see condition 1.2) compatible with the ports of the old type version (i.e. parameter *OldType*). The set of types that interact with *OldType* are modelled by means of a multiobject called *AttachedTypes*. The ports of the types that interact with *OldType* (i.e. variable *p3*) are used to evaluate the syntactic and semantic compatibility of the ports provided by *NewType* (i.e. variable *p2*). Moreover, since *AttachedTypes* includes all the types connected to *OldType*, in case *OldType* were connected to itself, *AttachedTypes* would include OldType in its set. This guarantees that the updating of a self-interacting type is made consistently: the ports of the new version must be semantically compatible with the ports of the old version. Self-interacting types are common in self-organised systems: different (distributed) instances of the same type (e.g. agents) interact themselves, sharing a common interpretation of the environment. In this case, semantic compatibility guarantees that instances of the old version can interact consistently with instances of the new version.

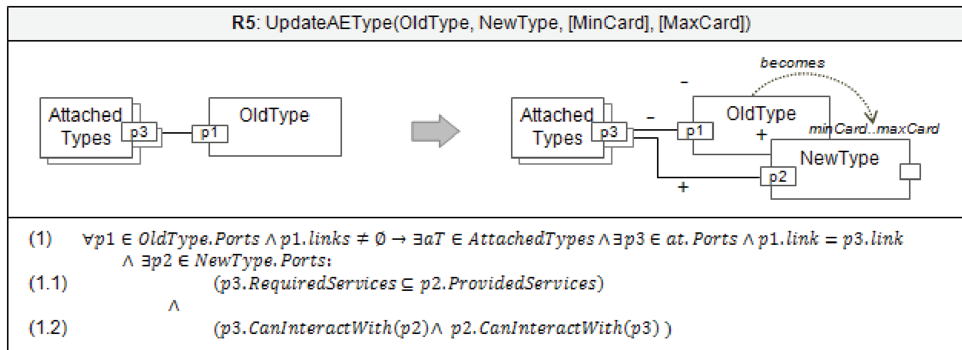

Fig. 11. Rule R5 - UpdateAEType

The execution of this rule introduces the new type version tagged for addition (i.e. see the symbol '+' near the *NewType* component), and tags the old type version for removal (i.e. see the symbol '-' near the *OldType* component). The use of the addition and removal tags activates or constrains the behaviour of other instance-level rules. For instance: the rule R2 (see Fig. 8) not only will create instances of the new type version, but will also avoid the creation of instances of the *old* type version. Another example: the rule R3 (see Fig. 9) will only allow the activation of instances of the new type version if and only if the minimum cardinality is satisfied.

However, in order to distinguish an update operation (which requires state migration) from an addition or removal operation (which also introduce addition/removal tags but do not preserve the instance state), this rule also introduces a relationship among the old type and the new type: the relationship "becomes". It specifically models which type is going to replace the older version, and activates the rule *ReplaceAE* (which is described below) for performing the migration of the instances to the new type version.

With respect to interactions, the existing connections (i.e. links to *AttachedTypes*, a multiobject which represent the set of types that are interacting with *OldType*) are unlinked from the type to

The original publication is available at www.springerlink.com

update (i.e. *OldType*) and linked to the new type version (i.e. *NewType*). This is modelled by tagging the old links with the symbol '-' (i.e. a removal tag) and the new ones with the symbol '+' (i.e. an addition tag). These tags will avoid the creation of new attachments at the instance-level among instances of *OldType* and *AttachedTypes*, promoting instead the creation of attachments with instances of *NewType*. This way, *OldType* will be progressively removed from the System.

On the other hand, the *ReplaceAE* rule models the replacement of an instance by a new one, migrating its previous internal state and updating its existing connections (see R6, Fig. 12). This rule is activated if and only if a component instance is detected in a running System which matches the following conditions: (1) the type of such instance (which matches in the rule with *OldType*) has a "becomes" relationship with another type (i.e. *NewType* in the rule); and (2) such instance has reached a quiescent status (represented in the rule by the symbol "[ ]" around the instance *S1*). The first condition detects that the matching type has been updated. The second condition ensures that the interactions of the instance to migrate are stopped, and that the instance state is consistent, ready to be migrated. This is guaranteed by the quiescent status [25], which is only achieved when there are no running and/or pending transactions.

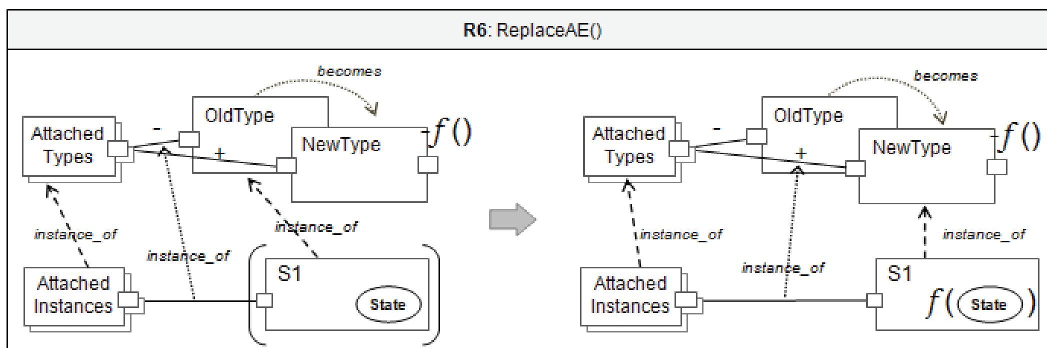

Fig. 12. Rule R6 - ReplaceAE

The result of the execution of the *ReplaceAE* rule is the migration (or transformation) of an instance of the old type to an instance of the new type. This migration is modelled by means of the modification of the *instance_of* relationship and the transformation of the internal state. An instance of a type that has been updated (i.e. *S1* is an *instance_of* the type *OldType*, which will *becomes NewType*) is transformed to an instance of another type (i.e. in the right hand part of the rule R6, *S1* is now an *instance_of* the type *NewType*). This results in that the internal state of the instance (i.e. the *State* element inside the *S1* instance, in the left hand part of the rule R6) is transformed to another (i.e. *f(State)*, in the right hand part of R6), by means of a state migration function. This function is modelled as *f()* and defines the mappings from the old data structures to the new ones.

There are two conditions to enable the state migration of instances: (i) the accessibility of their internal state, and (ii) the availability of a state migration function. On the one hand, if the old component type does not make accessible somehow the internal state of its instances, obviously we cannot migrate their state. For this reason, one of the conditions is that the old type provides a mechanism to get the internal state of its instances at runtime: this can be achieved by means of reflection, or by specialized functions that return the internal state to authorized requests (e.g. the migration function of a new version of the same type). This condition is explicitly included in the rule R6: the *State* of *S1* is visible, at least in the context of the rule. However, the rule does not reflect to *whom* it is accessible. It will depend on the specific implementation.

On the other hand, the new component type must provide a function to create new instances from the data structures of a previous type version. Otherwise, the state of an old instance could not be introduced into a new instance. The implementation of this function can be provided by

The original publication is available at www.springerlink.com

means of a specialized constructor of the new type version which creates a new instance from the state of an instance of the previous type version. This condition is also explicitly included in the rule R6: *NewType* provides a function *f* , which results in the transformation of the original state of *S1* after the execution of the rule. The specific details for the creation of state migration functions are outside the scope of this paper. This rule only models the presence of such functions and what is the result obtained after the execution of the rule. If a state migration function is not provided (or the state of the instance is not accessible), then the old state cannot be migrated and is simply lost. However, the reader can refer to the works of Ritzau & Andersson [37], or Vandewoude & Berbers [46] for further details about how to automate the creation of state migration functions.

Another result of the execution of the *ReplaceAE* rule is the updating of the links of the instance to evolve. The set of instances that are interacting are represented by a multiobject called *AttachedInstances*, and their corresponding types by another multiobject, *AttachedTypes*. Note that, as a result of the execution of the type-level updating rule, the interacting types (i.e. *AttachedTypes*) are semantically compatible with the updated type (i.e. *NewType*). As a consequence, their instances (i.e. *AttachedInstances*) will be also semantically compatible (i.e. they could interact correctly) with the instance after evolution. The updating of links is modelled by means of the modification of *instance_of* relationships (which link the instance-level with the type-level): the attachments (i.e. links among instances) belong to different *attachment types* (i.e. patterns of interaction) before and after the execution of the rule. This means that, when the rule is executed, the existing links with other instances are deleted and replaced by new ones, but which point to the updated instance instead of the old one. Then, all the elements could start interacting.

Finally, note that rule R6 is abstract enough to model two kind of update approaches: *type substitution* and *type transformation*. One the one hand, in type substitution approaches, which has been commonly used in dynamic updating approaches (e.g. [26, 37, 40]), updating is performed through the *replacement* of instances of the old version by instances of the new version. Old instances are completely stopped and their state is migrated to new, updated instances. On the other hand, in type transformation approaches (e.g. [13]), the updating is performed by the *transformation* of the internal structure of old type instances to accommodate the elements introduced by the new type. From outside, an evolved instance keeps the original boundaries, links and state, but also integrates the behaviour added by the new type. From inside, only the elements that have been affected by the change are modified: their state is migrated to new ones, whereas the state of non-changed elements is kept intact. Type transformation approaches are better suited to partially change large architectural types (e.g. servers), because they do not require stopping the entire architectural instance, but only the required parts of the instance.

This is the focus that have been used in our proposal to model recursively the evolution of PRISMA Systems. From outside, the evolution of a System is perceived as a type substitution: its instances are *replaced* by new versions having their state migrated. However, from inside the evolution is performed as a type transformation: the differences among the new type specification and the current specification are used to incrementally change the original instances, by means of a set of evolution operations (e.g. see Fig. 5). This strategy can be recursively applied, until (i) the decomposition of a type is not advisable (i.e. more than the 60% of type structures are going to be changed), or (ii) the internal composition is neither available or modifiable (e.g. COTS).

4.2. Graph-Based Abstract Description of the Evolution Semantics

The semantic rules described in the previous section are formalised by mapping their architecture-based concrete description to a graph-based abstract description. Then, the graph-based abstract rules can be entered in a graph transformation tool, such as AGG [1], to simulate and validate their execution. This will allow us to analyze some properties and dependencies from an high abstraction level (e.g. interaction dependencies, instance-level implications, etc.). In order to do this, the first step is the definition of a *type graph*, i.e., a graph representing a metamodel to define the concepts and relations in the domain. Instances of this metamodel are called instance graphs.

The original publication is available at www.springerlink.com

They represent the runtime states of the system and are subject to modification by transformation rules: given a valid instance graph and a rule, the rule could be applied to the graph to produce a new graph. If this graph satisfies the constraints of the type graph (i.e. the metamodel) it constitutes the successor state in the runtime model of the system. Note that elements of type graphs are not types in the architectural sense, but metamodel elements. In order to model the evolution of both types and instances, an instance graph must include not only instances (e.g. Configurations), but also the types that provide the behaviour of these instances (e.g. Systems). Therefore, a type graph must describe the ontological metamodel [4] of the concepts that are going to be subject to evolution. That is, the type graph must describe both the meta-types (i.e. the properties and relations among types) and the meta-instances (i.e. the properties and relations among instances). In addition, this type graph should also include the concepts required to describe how the concepts evolve over time: the evolution tags.

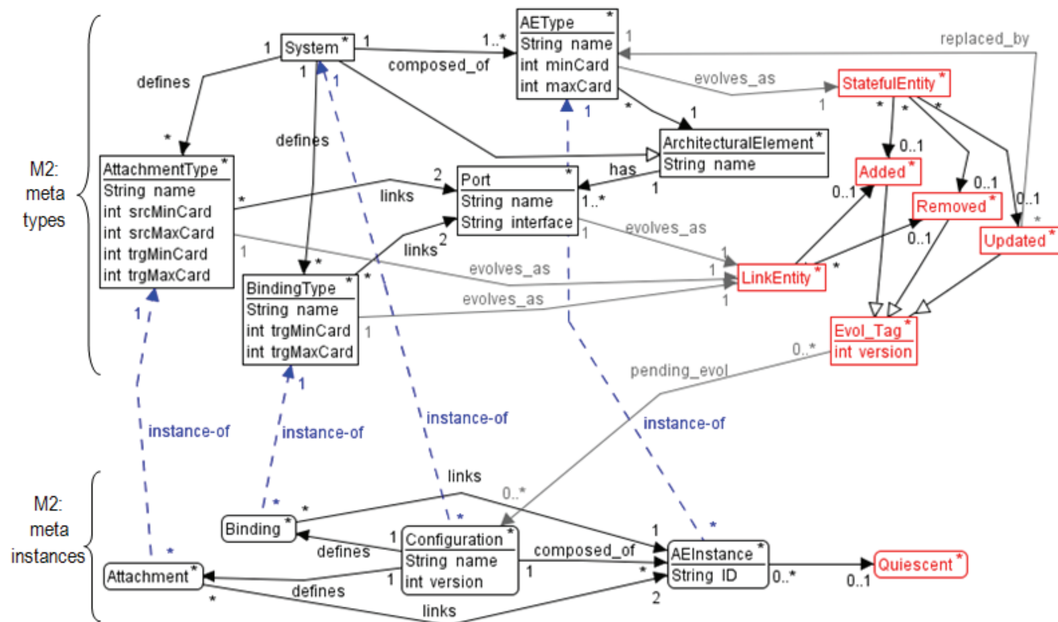

Fig. 13. Type Graph of PRISMA with Evolution Tags

Fig. 13 shows the type graph that includes the concepts of the PRISMA metamodel and the concepts related to the evolution of types and instances. The *ArchitecturalElement* concept represents simple PRISMA architectural elements (i.e. not composed elements), which provide or request services through a set of *Ports* (see the *Port* concept in the metamodel). The *System* concept represents composed PRISMA architectural elements: they are composed of several architectural elements (*AEType*), which are connected by means of attachments (*AttachmentType*) and/or bindings (*BindingType*). Since a System is also an architectural element (e.g. it has ports), it inherits its behaviour from the ArchitecturalElement concept. The *AEType* concept provides an alias for an architectural element that is imported into a System and the allowed cardinality in such System. There are three kind of evolution tags: *Added*, *Removed*, and *Update*. However, the latter is only used by stateful entities, i.e. architectural elements. The reason is that the update of stateless entities (like attachments, bindings and ports) can be reflected by means of a removal tag and an addition tag. In the case of stateful entities, this is not enough, since the state must be migrated from the old entity to the new one. For this reason, the *updated* concept has a link (called *replaced_by*) to the architectural element that is going to replace the tagged element. All the evolution tags (see the concept *Evol_Tag*) have a link to the System instances (i.e. Configurations) that are pending to be evolved. Finally, the metamodel defines what are the elements of the

The original publication is available at www.springerlink.com

instance-level, their properties and relationships (see Fig. 13, meta-instances). Note how the quiescent status has been added to the concept *AEInstance*, in order to reflect when an instance is ready to be evolved.

Fig. 14 illustrates how an instance graph looks like: it includes type-level concepts and instance-level concepts. In particular, the type-level concepts included in this graph are related to the System *Sys*, after the type-level execution of the evolution process "E_0:*Sys.v*$_0$→*Sys.v*$_1$": it removes the architectural type *B* and adds the type *C* (see Fig. 5). The System *Sys* is modelled as a graph, and each one of its structural elements (ports, architectural types, and connections) are modelled as nodes of this graph, according to the type graph described above. For instance, the type *A* is modelled with two graph nodes: (i) an *ArchitecturalElement* node named "Comp_A" which defines the behaviour of the type and which is linked to the *Port* nodes named "PClient" and "PServ"; and (ii) an AEType node named "A" which defines the usage restrictions of such type in *Sys*: a minimum and maximum cardinality of 1. Note how the elements that have been affected by the evolution process (i.e. the attachment types *Att_AB*, *Att_AC*, and the architectural elements *B*, *C*) have been tagged with removal or addition tags. Finally, the instance graph shows some elements of the instance-level (see M0): the Configuration *C1*, which is still conforming to version 0 of *Sys*.

This graph is a clear example of the benefits of using an architecture-based concrete description instead of a graph-based abstract description to describe the evolution semantics: the former is more concise than the latter. However, the former cannot be automatically validated and/or formally analysed to detect inconsistencies.

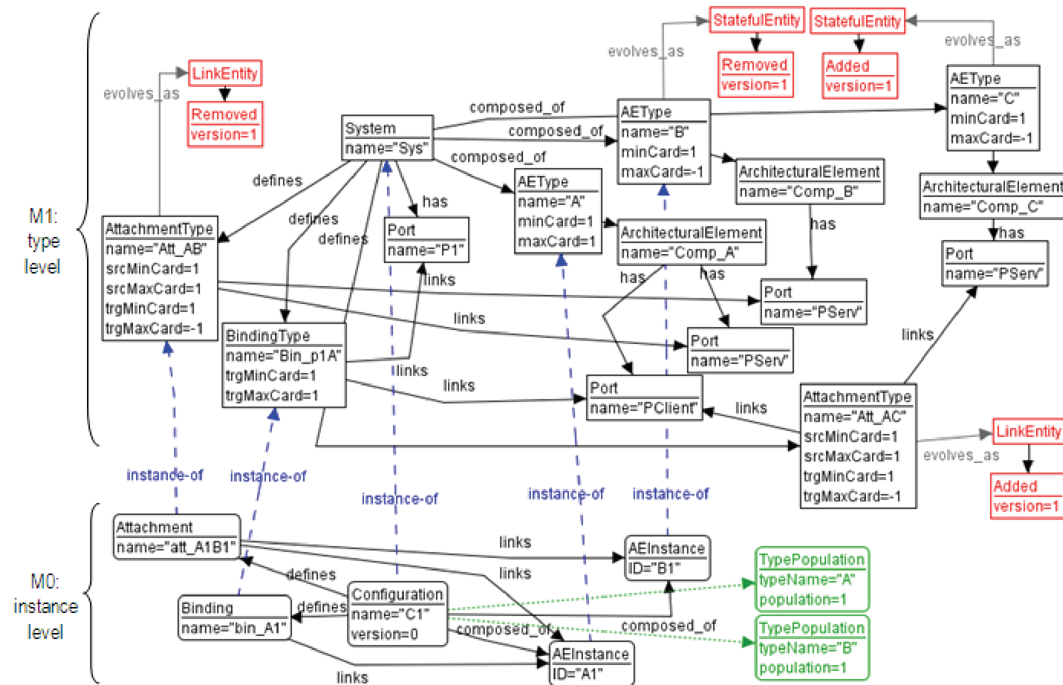

Fig. 14. Instance graph of the Sys type (at version 1) with the C1 instance (at version 0)

An example of the mapping of an evolution rule to a graph transformation rule is shown in Fig. 15. This figure shows how the reconfiguration operation *CreateArchitecturalElement* has been modelled in AGG, renamed as *CreateAE*. The execution of this rule requires three input parameters: *configName*, the name of the Configuration where a new instance is going to be created; *typeToInstantiate*, the name of the type to instantiate; and *newID,* the identifier of the new instance to create. The left-hand side of the rule describes the initial matching (see Fig. 15, center)

The original publication is available at www.springerlink.com

required to execute the rule. The instance graph must contain a Configuration node whose attribute “name” equals to *configName*. This node must be linked to a System node (by an *instance-of* relationship), which in turn is linked to an AEType node with an attribute “name”=*typeTo Instantiate*. This checks that the type to instantiate is declared in the Configuration type (i.e. the System node *sysName* linked to the Configuration). The right-hand side of the rule describes the result of the transformation rule: a new AEInstance node, whose attribute “ID” is *newID*, has been created and linked to the selected Configuration node (i.e. *2:Configuration*). In order to reflect the fact that the new instance is in a quiescent status, it is also linked to a Quiescent node.

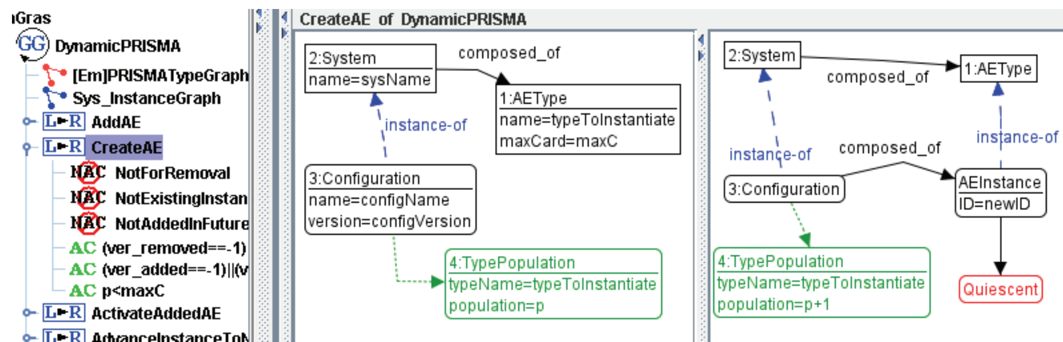

Fig. 15. CreateAE: mapping of rule R2 in the AGG [1] tool

The complex conditions defined by the rule R2 (see Fig. 8) have been modelled by a set of NACs (Negative Applicable Conditions) and attribute conditions (see Fig. 15, left). These conditions allow the execution of a graph transformation rule only if: (i) none of the defined NACs matches with the instance graph, and (ii) all of the attribute conditions are true. Here is described how complex R2 execution conditions have been modelled in AGG:

(i) If *typeToInstantiate* has been removed from the System type (i.e. it has been tagged for deletion), it must have been removed in a System version greater than the current Configuration version. This is checked by a NAC and an attribute condition. NAC1 (see Fig. 16) checks if the type to instantiate (i.e. the node *3:AEType*) is tagged for deletion (i.e. it is linked to a Removed node). If true, the variable *var_removed* will contain this tag version number; otherwise (the type has not been deleted yet), it will contain the value -1. This variable is compared with the current Configuration version (see the first attribute condition in Fig. 17), checking that it is greater. If it is evaluated to false, then the rule *CreateAE* cannot be executed.

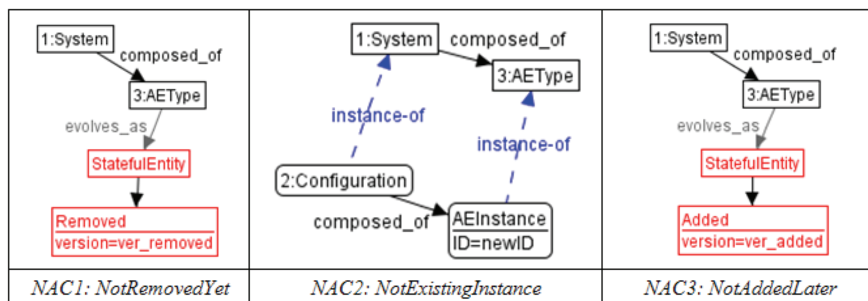

Fig. 16. NACs of the graph transformation rule *CreateAE*

(ii) The identifier of the new instance, *newID*, must not have been used previously. This is checked by NAC2 (see Fig. 16): if the selected configuration (i.e. the node *2:Configuration*) is linked to an AEInstance node with an ID=*newID*, then the NAC condition is true and the rule *CreateAE* cannot be executed.

The original publication is available at www.springerlink.com

(iii) If *typeToInstantiate* has been added at runtime (i.e. it is tagged with an addition tag), it must have been added in a System version less than the current Configuration version. This is similarly checked as with the removal case. See NAC3 in Fig. 16 and the second attribute condition in Fig. 17.

(iv) The number of existing *typeToInstantiate* instances in the Configuration must be lower than the maximum cardinality defined in the System type. This is checked with the help of an auxiliary node, TypePopulation (see Fig. 15), which keeps the number of instances (attribute *population*) of a type (attribute *typeName*) that have been instantiated in the Configuration it is linked to. Then, an attribute condition (see the third attribute condition in Fig. 17) checks that the population, *p*, of the selected TypePopulation node (i.e. this which attribute *typeName* equals to the type to instantiate, *typeToInstantiate*) is less than the maximum cardinality, *maxC*, of the selected *AEType* node.

Conditions

Expression	OK
(ver_removed==-1)||(ver_removed>configVersion+1)	✔
(ver_added==-1)||(ver_added<=configVersion+1)	✔
p<maxC	✔
	☐

Fig. 17. Attribute conditions of the graph transformation rule *CreateAE*

5. Discussion

The main goal behind the use of self-adaptive/self-organizing systems is the realization of quality attributes (e.g. performance, reliability, dependability, etc.). These quality attributes are measured at runtime to prevent their degradation, and in such a case, corrective actions are automatically performed by the system [23,32]. These corrective actions may involve runtime adaptations or re-organizations of the system, following some rules or guidelines introduced at design-time. Therefore, in such systems, runtime adaptations/reorganizations are part of the system behaviour. Dynamic evolution is beneficial in this kind of systems not only for introducing unforeseen changes to the functional concerns of the system, but also for changing the self-adaptive/self-organizing concerns of the system, like the adaptation rules or the algorithms that guide the local organization of system nodes.

However, the question here is to what extent a dynamic evolution process may impact, or be in conflict with, existing self-adaptive/self-organizing behaviour. For instance, the dynamic update of a subsystem may affect (at least temporarily) a quality attribute (e.g. performance) of the whole system. This could trigger the activation of self-adaptive behaviour, which may be in conflict with the dynamic evolution process. In order to address these possible conflicts, two possible scenarios must be considered. On the one hand, if self-adaptive behaviour is triggered by the instance (or subsystem) that is being evolved, the dynamic evolution process must prevail: external, ad-hoc changes upon a type should be prioritary instead of programmed management operations. The reason is that incorrect or conflicting self-adaptive behaviour may prevent the installation of updates, leading an instance to be indefinitely out-of-date. This does not mean that current executing processes must be aborted, but that new processes must be postponed in benefit of the execution of a dynamic evolution process. That is, when a self-adaptive action is going to be performed, the dynamic type change should be executed first, and then the need for the self-adaptive action should be reevaluated. This is reflected in our approach: there may be conflicts among the execution of a dynamic reconfiguration rule (based on an outdated type) and the integration of a new type version (e.g. see rules R2 and R3). The transformation rules take this into account: in presence of a new type version, the dynamic reconfiguration behaviour is constrained to fulfill the changes imposed by the new, evolved type, instead of the current type.

On the other hand, if self-adaptive behaviour is raised by other subsystems (i.e. not being the subject of an evolution process), the presence of dynamic evolution processes must be notified to

The original publication is available at www.springerlink.com

these subsystems to avoid conflicts. For instance, if a dynamic evolution process is applied on a server component instance, then the performance of the processing rate of requests will invariably decrease. Then, if a self-adaptive subsystem detected this decrease on performance, it might trigger a dynamic reconfiguration policy for creating new instances of the "delayed" server component. In our approach, this decision would not be a problem: since a type is the first on being evolved, the new instances of the server component would be conformant to the new type version instead to the old one. And when the server instance that was evolving would finish its evolution, it could continue serving requests (probably better, since it has been updated/corrected). However, if the self-adaptive subsystem had decided to disconnect all the server component instances of the type that is being evolved and use a different server type, then the dynamic evolution of the server type would have been useless. A way to overcome these unpredicted situations is to generate an event to notify that an instance is temporarily unavailable due to an evolution process. This event should be notified to the neighbours of the evolving instance and/or to the self-adaptive subsystems. This way, a self-adaptive subsystem could take this information into account to postpone the evaluation of the quality attributes that are affected by the evolving instance. We have modelled this event in our approach by using the quiescent status: when an instance achieves this status is because it is engaged in an evolution process. Then, this information can be used by other subsystems (like a self-adaptive subsystem) to avoid reconfiguration conflicts.

In general, the execution of a dynamic evolution process will have an impact on some quality attributes of the instance to evolve and its context (mainly performance). In most cases, this impact will be temporary, since it is related to the execution of an evolution process and this evolution process has a limited duration. In fact, after the finalization of the evolution process, generally the quality attributes of the system will recover its previous values (or better values, if the installed updates benefit these attributes). However, in the worst case, it may happen that the installed updates degrade the overall quality of the system (e.g. some interrelations have not been considered by the architect). In this case, the dynamic updates can also be reverted by using the version management we have described in this paper: since a type specification keeps all the information of previous versions, in case anything fails, the changes could be reverted.

The approach presented in this paper captures the dynamic evolution process at a very high level of abstraction, as a sequence of gradual, asynchronous changes on both type and instance-level. We have chosen graph transformations to describe the evolution process because they naturally model both the system itself (as a graph of types and configurations) and the asynchronous nature of its evolution (by individual application of rules without global control). As opposed to logic-based formalisations [43] or process calculi approaches [10, 15], the use of graphs allows us to represent the system and its runtime state at a high level of abstraction. The direct mapping between graphs and visual models resembles that between UML diagrams and their metamodel-based abstract representations. We have shown that graph transformations allow to describe precisely and concisely the principles and mechanisms of dynamic evolution: (i) how, in presence of change, will the involved type and instances react, and (ii) how will their interactions with other elements be managed.

However, the details of how the quiescence status is achieved by instances have been explicitly excluded from the model. The reason is that this would require modelling also the instance execution model: for modelling the safe stopping of running transactions and the blocking of new service requests, the model should also include how instances process and execute service requests. This would add excessive complexity on the evolution model, thus eliminating the benefits of a concise description. We have decided to simply model quiescence as an attribute of an instance, which describes when this status is reached. The concrete semantics have been left to the infrastructure (i.e. the middleware). More details of the semantics can be found in [25, 47].

Finally, one consideration remains concerning the management of inactive type versions (ie. old versions without instances). Version inactivity (as described in section 2) must be considered *within the context of a System*. Every System has a local copy of a type, and each local copy can integrate the updatings (i.e. versions) at different times: a version may be inactive in one System

The original publication is available at www.springerlink.com

(because it has been evolved), but still be active in another System (because due to semantic incompatibility the new version cannot be introduced). For this reason, the accessibility of inactive versions must be guaranteed, by storing them in a database or filesystem for future reference.

6. Related Work

Several works have addressed the dynamic evolution of software systems. Segal and Frieder [40] reviewed the first approaches (around the late 1970s and earlier 1980s) to support *dynamic program updating*. Such earlier works were mainly based on supporting the evolution of procedure-oriented programs, with few emphasis on the concepts of types and instances. However, Fabry's work [20] investigated the updating of (abstract) data types and the migration of their instances. His work provided deferred updates, using version attributes to distinguish the out-of-date instances. The main mechanism used was the use of indirections at code segments: all the calls to the old code were updated with the address of the new code segment. Thus, running processes could continue executing the old code, whereas new processes would start executing the new code. However, this means that not only the old code segment is modified, but also the calling programs, which results in invasive evolutions. The benefit of current architectural approaches, as our work, is that interactions among processes are made explicit and separated from the functional behaviour, thus facilitating their evolution. Another limitation of Fabry's work is that external interfaces could not be modified.

With the expansion of object-oriented languages and frameworks, the distinction among types and instances (e.g. classes and objects) become evident, and also the need for dynamically updating both. JDRUMS [2,37] provides transparent dynamic updating features to Java programs, by means of a modified Java Virtual Machine. It extends the Java class loader (i.e. the Class and Object Java meta-types) to include a link to the class (or object, respectively) that replaces it. When a class is dereferenced, JDRUMS checks if it has been updated and then returns the last version. Object updates are performed when they are dereferenced; their old, internal state is converted to the structure of the new class. Thus, different versions of the same class, as well as their objects, can coexist simultaneously: asynchronous evolution is supported. Other works also address the dynamic evolution of Java programs by extending the default class loader, such as Malabarba et al. [26] and Wang et al. [48]. However, both approaches perform a synchronised dynamic evolution, not suitable for distributed systems.

Most of the techniques for supporting dynamic evolution only differ on how the indirection is managed. An interesting review of the wide range of techniques proposed can be found at [27]. For instance, PROSE [31] supports the dynamic weaving of new concerns (which are implemented as methods) to the existing code. It uses code interception and redirection to introduce the new code. However, the use of multiple type versions is not considered, neither the migration of running instances. In general, the existing works describing how to support dynamic evolution are focused on the implementation details. The contribution of our work is that we provide a high-level model for describing the evolution.

Dynamic evolution has also raised the interest in the area of software architecture [7,10,15,32]. However, the interest has been mainly focused on the description and/or support of the dynamic (self)reconfiguration of a single system instance (i.e. a composite component instance), according to an overall architectural specification, style or pattern [12,14,24]. However, the evolution of a type specification has not explicitly taken into account. For instance, focusing on graph-based approaches, the works of Hirsh et al [22], Wermelinger et al. [49], or Bucchiarione et al. [8] use typed graph grammars to describe an architectural type (i.e. a style or a pattern) and graph instances to describe architecture instances. Then, typed graph transformation rules are used to model the dynamic reconfiguration, keeping the architectural type intact. We have also used typed graph grammars as the basis for the formalization of the dynamic evolution process. However, we have used typed graph grammars to represent a software architecture metamodel (i.e. PRISMA)

The original publication is available at www.springerlink.com

and graph instances to represent both an architecture type and the set of its instances. Thus, graph transformations can change the type and its instances, while taking into account both type constraints (defined in the left-hand side of rules) and the metamodel constraints (defined in the typed graph).

Other approaches that have addressed dynamic evolution from an high-level of abstraction are those based on reflective concepts, such as [3,5,11,15]. Reflection is a powerful concept to describe the ability of a system to reason about itself and act upon itself. However, these approaches do not perform asynchronous evolution as covered in this work.

7. Conclusion and Further Work

In this paper we have introduced the semantics of the asynchronous evolution of types in the context of software architecture. This is an important feature for the design and development of large systems with a long-time usage, since it allows the introduction of concurrent changes at runtime without waiting to sequence all changes, which would delay the introduction of needed capabilities or problem fixes. When considering the use of dynamic evolution support, its benefits (i.e. supporting the correction or improvement of executing software artifacts) should be evaluated against its disadvantages (i.e. a negative impact on the system quality attributes during the execution of an evolution process).

This work has been focused on the dynamic evolution of the structural view of systems (i.e. the architecture of their subsystems). It has been addressed from a white-box perspective: the internal structure of the architectural instances are evolved gradually, by adding or removing its elements at runtime, connecting/disconnecting them, etc. From a black box perspective, this is perceived as a replacement of the entire architecture and the migration of their state. However, internally, only some parts have been changed, while keeping the state of the other elements.

The evolution semantics described here allows the concurrent, but ordered, dynamic evolution of both the type and its instances. The evolution is concurrent because both the type and its instances evolve asynchronously, independently of each other. But the evolution is also ordered because instances evolve towards the last updated version of the type, while preserving the order in which different evolution processes (i.e. those that create new type versions) were introduced. Version management is carried out by means of evolution tags, which allow keeping evolution traceability, so that each instance can follow the evolution of its type across the time. Thus, although a type would evolve several times, at the end each instance would converge to the last version of the evolved type.

The evaluation of the evolution semantics, by simulating all the rules in a tool like AGG, is an ongoing work. However, there are issues that have not been covered, such as the description of how runtime faults are managed. Since rules only describe preconditions and post-conditions of actions, it is not addressed what happens in the middle of such actions, for instance, if an evolution operation cannot be finished. Another issue is the correct evaluation of semantic compatibility: our model would have to be complemented with a mechanism to establish compatibility among types. In addition, further work remains, such as the definition of a fully distributed and decentralized asynchronous evolution model. The model presented is decentralized at the instance-level: each instance evolves at different times, without requiring to be located at the same node where the evolution started. However, the model relies on a centralized type which receives the evolution requests and propagates the changes to its (distributed) instances. In a fully decentralized model, a type may also be distributed among different nodes. In this case, when an evolutionary change is requested on a type, it should be propagated properly to the other nodes. Then, some issues arise like: (i) how to keep (distributed) type specifications synchronized; (ii) how to manage concurrent type evolution requests and avoid type version branching. These issues are similar to those dealt in versioning management approaches [16,45]. This is an ongoing work which has not been covered in this paper.

The original publication is available at www.springerlink.com

References

1. AGG: Attributed Graph Grammar System Tool, http://user.cs.tu-berlin.de/~gragra/agg/
2. Andersson, J., Comstedt, M., Ritzau, T.: Run-time support for dynamic Java architectures. In: *ECOOP'98 workshop on Object-Oriented Software Architectures (WOOSA'98)*. Brussels (1998)
3. Andersson, J., De Lemos, R., Malek, S., Weyns, D.: Reflecting on Self-Adaptive Software Systems. In: *ICSE workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'09)*. Vancouver, Canada (2009)
4. Atkinson, C., Kühne, T.: Model-Driven Development: A Metamodeling Foundation. *IEEE Software* 20(5), 36-41 (2003)
5. Bencomo, N., Blair, G.S., Flores-Cortés, C.A., Sawyer, P.: Reflective Component-based Technologies to Support Dynamic Variability. In: *2nd Int. Workshop on Variability Modelling of Software-Intensive Systems (VaMoS'08)*. Universität Duisburg-Essen, Germany (2008)
6. Beydeda, S., Book, M., Gruhn, V.: *Model-Driven Software Development*. Springer (2005)
7. Bradbury, J.S., Cordy, J.R., Dingel, J., Wermelinger, M.: A Survey of Self-Management in Dynamic Software Architecture Specifications. In: *Workshop on Self-Managed Systems (WOSS'04)*. Newport Beach, CA (2004)
8. Bucchiarone, A., Melgratti, H., Gnesi, S., Bruni, R.: Modelling Dynamic Software Architectures using Typed Graph Grammars. *Graph Transformations for Verification and Concurrency*. ENTCS vol. 213(1), 39-53. Elsevier (2007)
9. Cámara, J., Salaün, G., Canal, C.: Composition and Run-time Adaptation of Mismatching Behavioural Interfaces. *J. of Universal Computer Science* 14(13), 2182-2211 (2008)
10. Canal, C., Pimentel, E., Troya, J.M.: Specification and Refinement of Dynamic Software Architectures. In: *Working IFIP Conference on Software Architecture (WICSA'99)*. San Antonio, Texas, USA (1999)
11. Cazzola, W., Ghoneim, A., Saake, G.: Software Evolution through Dynamic Adaptation of Its OO Design. In: *Objects, Agents, and Features*. LNCS, vol. 2975, pp. 31-48. Springer, Heidelberg (2003)
12. Cheng, S., Garlan, D., Schmerl, B.R.: Making Self-Adaptation an Engineering Reality. In: *Self-star Properties in Complex Information Systems*. LNCS, vol. 3460, pp. 158-173. Springer, Heidelberg (2005)
13. Costa-Soria, C., Hervás-Muñoz, D., Pérez, J., Carsí, J.A.: A Reflective Approach for Supporting the Dynamic Evolution of Component Types. In: *14th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'09)*. Potsdam, Germany (2009)
14. Costa-Soria, C., Pérez, J., Carsí, J.A.: An Aspect-Oriented Approach for Supporting Autonomic Reconfiguration of Software Architectures. In: *2nd Workshop on Autonomic and SELF-adaptive Systems (WASELF'09)*. San Sebastián, Spain (2009)
15. Cuesta, C.E., Romay, P., Fuente, P.d.l, Barrio-Solórzano, M.: Reflection-Based, Aspect-Oriented Software Architecture. In: *European Workshop on Software Architecture (EWSA'04)*. LNCS, vol. 3047, pp. 43-56. Springer, Heidelberg (2004)
16. De Lucia, A., Deufemia, V., Gravino, C., Risi, M.: Behavioral Pattern Identification through Visual Language Parsing and Code Instrumentation. In: *13th European Conference on Software Maintenance and Reengineering (CSMR'09).* Kaiserslautern, Germany (2009).
17. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: *Fundamentals of Algebraic Graph Transformation (Monographs in Theoretical Computer Science. An EATCS Series)*. Springer (2006)
18. Engels, G., Heckel, R., Küster, J.M., Groenewegen, L.: Consistency-Preserving Model Evolution through Transformations. In: Jézéquel, J.M., Hußmann, H., Cook, S. (eds.): «UML» 2002 - The Unified Modeling Language. LNCS, vol. 2460, pp. 212-227. Springer, Heidelberg (2002)
19. Engels, G., Heckel, R., Küster, J.M.: Rule-Based Specification of Behavioral Consistency Based on the UML Meta-model. In: Gogolla, M., Kobryn, C. (eds.): «UML» 2001 - The Unified Modeling Language, Modeling Languages, Concepts, and Tools. LNCS, vol. 2185, pp. 272-286. Springer, Heidelberg (2001)
20. Fabry, R.S.: How to design a system in which modules can be changed on the fly. In: *2nd International Conference on Software Engineering* (ICSE'76). San Francisco, California, USA (1976)
21. Heckel, R.: Graph Transformation in a Nutshell. *School on Foundations of Visual Modelling Techniques (FoVMT 04)*. ENTCS, vol. 148(1), pp. 187-198. Elsevier (2006)
22. Hirsch, D., Inverardi, P., Montanari, U.: Graph grammars and constraint solving for software architecture styles. In: *3rd Int. Software Architecture Workshop (ISAW-3)*. ACM Press (1998)
23. Kephart, J.O., Chess, D.M.: The Vision of Autonomic Computing. *IEEE Computer*, 36(1), 41-51 (2003)
24. Kramer, J., Magee, J.: Self-managed systems: an architectural challenge. In: *ICSE - Future of Software Engineering (FOSE'07)*. IEEE (2007)

The original publication is available at www.springerlink.com

25. Kramer, J., Magee, J.: The Evolving Philosophers Problem: Dynamic Change Management. *IEEE Transactions on Software Engineering* 16(11), 1293-1306 (1990)
26. Malabarba, S., Pandey, R., Gragg, J., Barr, E., Barnes, J.F.: Runtime support for type-safe dynamic Java classes. In: Bertino, E. (ed.): *ECOOP 2000- Object-Oriented Programming*. LNCS, vol. 1850, pp. 337-361. Springer, Heidelberg (2000)
27. McKinley, P.K., Sadjadi, S.M., Kasten, E.P., Cheng, B.H.C.: Composing Adaptive Software. *IEEE Computer* 37(7), 56-64 (2004)
28. Medvidovic, N., Taylor, R.N.: A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering* 26(1), 70-93 (2000)
29. Mens, T., Wermelinger, M.: Separation of concerns for software evolution. *J. of Software Maintenance and Evolution* 14(5), 311-315 (2002)
30. Milner, R.: *The Polyadic π-Calculus: A Tutorial*. Laboratory for Foundations of Computer Science Department, University of Edinburgh (1993)
31. Nicoara, A., Alonso, G., Roscoe, T.: Controlled, Systematic, and Efficient Code Replacement for Running Java Programs. *SIGOPS Operating Systems Review* 42(4), 233-246 (2008)
32. Oreizy, P., Gorlick, M., Taylor, R.N., et al.: An Architecture-Based Approach to Self-Adaptive Software. *IEEE Intelligent Systems* 14(3), 54-62 (1999)
33. Pérez, J., Ali, N., Carsí, J.A., Ramos, I., et al.: Integrating aspects in software architectures: PRISMA applied to robotic tele-operated systems. *Information & Software Technology* 50(9-10), 969-990 (2008)
34. Pérez, J., Ali, N., Carsí, J.A., Ramos, I.: Designing Software Architectures with an Aspect-Oriented Architecture Description Language. In: Gorton, I. et al. (eds.) *Component-Based Software Engineering*. LNCS, vol. 4063, pp. 123-138. Springer, Heidelberg (2006)
35. Pérez, J.: *PRISMA: Aspect-Oriented Software Architectures*. PhD Thesis, Universidad Politécnica de Valencia, 2006.
36. Perry, D.E., Wolf, A.L.: Foundations for the Study of Software Architecture. *SIGSOFT Software Engineering Notes* 17(4), 40-52 (1992)
37. Ritzau, T., Andersson, J.: Dynamic Deployment of Java Applications. In: *Java for Embedded Systems*. London (2000)
38. Rogers, A., Jennings, N.R., Farinelli, A.: Self-Organising Sensors for Wide Area Surveillance using the Max-Sum Algorithm. In: *WICSA/ECSA Workshop on Self-Organizing Architectures (SOAR'09)*. Cambridge, UK (2009)
39. Rombach, H.D.: Design for Maintenance - Use of Engineering Principles and Product Line Technology. In: *13th European Conf. on Software Maintenance and Reengineering (CSMR'09)*. Kaiserslautern, Germany (2009)
40. Segal, M.E., Frieder, O.: On-the-Fly Program Modification: Systems for Dynamic Updating. *IEEE Software* 10(2), 53-65 (1993)
41. Serugendo, G.D.M., Gleizes, M.P., Karageorgos, A.: Self-organisation and emergence in MAS: An Overview. *Informatica (Slovenia)* 30, 45-54 (2006)
42. Software Engineering Institute: *Ultra-Large-Scale Systems: Software Challenge of the Future*. Technical Report, Carnegie Mellon University, Pittsburgh, USA (2006)
43. Stirling, C.: Modal and Temporal Logics. *Handbook of Logic in Computer Science, vol. II*. Clarendon Press, Oxford (1992)
44. Taylor, R.N., Medvidovic, N., Dashofy, E.M.: *Software Architecture: Foundations, Theory and Practice*. Wiley (2009).
45. Thao, C., Munson, E.V., Nguyen, T.N.: Software Configuration Management for Product Derivation in Software Product Families. In: *15th Int. Conf. on Engineering of Computer Based Systems (ECBS'08)*. Belfast, Northern Ireland (2008)
46. Vandewoude, Y., Berbers, Y.: Component state mapping for runtime evolution. In: *Int. Conf. on Programming Languages and Compilers*. Las Vegas, Nevada, USA (2005)
47. Vandewoude, Y., Ebraert, P., et al.: Tranquillity: A low Disruptive Alternative to Quiescence for Ensuring Safe Dynamic Updates. *IEEE Transactions on Software Engineering* 33(12), 856-868 (2007)
48. Wang, Q., Shen, J., Wang, X., Mei, H.: A Component-Based Approach to Online Software Evolution. *Journal of Software Maintenance and Evolution* 18(3), 181-205 (2006)
49. Wermelinger, M., Lopes, A., Fiadeiro, J.L.: A graph based architectural (re)configuration language. *SIGSOFT Software Engineering Notes* 26(5), 21-32 (2001)

The original publication is available at www.springerlink.com