Modes for Software Architectures*

Dan Hirsch, Jeff Kramer, Jeff Magee, and Sebastian Uchitel

Department of Computing, Imperial College London
{dhirsch| jk | j.magee | s.uchitel}@doc.ic.ac.uk

Abstract. Modern systems are heterogeneous, geographically distributed and highly dynamic since the communication topology can vary and the components can, at any moment, connect to or detach from the system. *Service Oriented Computing* (SOC) has emerged as a suitable paradigm for specifying and implementing such global systems. The variety and dynamics in the possible scenarios implies that considering such systems as belonging to a single architectural style is not helpful. This considerations take us to propose the notion of *Mode* as a new element of architectural descriptions. A mode abstracts a specific set of services that must interact for the completion of a specific subsystem task. This paper presents initial ideas regarding the formalization of modes and mode transitions as explicit elements of architectural descriptions with the goal of providing flexible support for the description and verification of complex adaptable service oriented systems. We incorporate the notion of mode to the Darwin architectural language and apply it to illustrate how modes may help on describing systems from the Automotive domain.

1 Introduction

Distributed systems are very complex dealing with a high number of architectures and communicating infrastructures. Modern systems are heterogeneous, geographically distributed and highly dynamic since the communication topology can vary and the components can, at any moment, connect to or detach from the system. As an answer to these requirements, *Service Oriented Computing* (SOC) has emerged as a suitable paradigm for specifying and implementing such global systems. Engineering issues are tackled by exploiting the concept of *services*, which are the building blocks of systems. Services are autonomous, platform-independent, mobile/stationary computational entities. In the deployment phase, services can be independently described, published and categorized. At runtime they are searched/discovered and dynamically assembled for building wide area distributed systems.

All these require, on the one hand, the development of foundational theories to cope with the requirements imposed by the global computing context, and,

* Partially supported by the Project EC FET – Global Computing 2, IST-2005-16004 SENSORIA and The Leverhulme Trust.

V. Gruhn and F. Oquendo (Eds.): EWSA 2006, LNCS 4344, pp. 113–126, 2006.

on the other hand, the application of these theories for their integration in a pragmatic software engineering approach.

At the architectural level, the fundamental features to take into account for the description of components and their interactions include: dynamic reconfiguration, self-organisation, mobility, coordination, complex synchronization mechanisms, multiple communication contexts, and awareness of quality of service. The variety and dynamics in the possible scenarios implies that considering such systems as belonging to a single architectural style is not helpful. These considerations take us to propose the notion of *Mode* as a new element of architectural descriptions. A mode abstracts a specific set of services that must interact for the completion of a specific subsystem task, i.e., a mode will determine the *structural constraints* that rule a (sub)system configuration at runtime. Therefore, passing from one mode to another and interactions among different modes formalize the *evolution constraints* that a system must satisfy: the properties that reconfiguration must satisfy to obtain a valid transition between two modes which determine the structural constraints imposed to the corresponding architectural instances.

This paper presents initial ideas regarding the formalization of modes and mode transitions as explicit elements of architectural descriptions with the goal of providing flexible support for the description and verification of complex adaptable service oriented systems. We consider that the concept of mode helps on closing the gap between requirements and software architectures by using modes as a *scenario-based* abstraction to relate specific use cases with service configurations. Also, we hope that it will permit the verification of reconfiguration correctness (for example, by a predefined set of reconfiguration operations that will carry one subsystem from one mode to another respecting the mode transition specification). It is worth noticing, that the relation between scenarios and modes is not necessarily one-to-one. The idea is that the scenario-based approach can help in understanding how the scenarios and modes can be related providing feedback for the validation of requirements with respect to reconfiguration issues at the architectural level. In particular, we think that modes can help on verifying correctness of coordination policies and deployment issues for self-organising/healing systems.

Our work is funded on the basic ideas from software architecture, as it is concerned with the selection of architectural elements, their interactions, and the constraints on those elements and their interactions necessary to provide a framework in which to satisfy the requirements and serve as a basis for the design [16,1]. A fundamental aspect of software architecture is that it is an abstraction that helps manage complexity. To deal with these issues, architecture-based formal modeling notations and analysis and development tools that operate on architectural specifications have been developed (i.e.Architecture Description Languages (ADLs) [15]). Also, notation standards like UML have been proposed to support architectural design [5,6,14]. In this respect, our contribution is the proposal of a (scenario-based) approach that introduces modes as a new first class primitive for languages (with special emphasis on service oriented ones) supporting the

work that has been done in recent years for self-orginising and reconfigurable systems [2,3]. It is worth noticing that modes are already used in other areas such as synchronous programming [13]. In [13], the notion of mode is related to collections of executing states, focusing on behavior. In our case, we are interested in studying modes with respect to system structure. Nevertheless, in future work we plan to study the behavioral side of modes (see Section 4).

In the rest of this paper we present our first ideas on how the notion of mode can be incorporated to an existing ADL (Darwin [9]). The extension is obtained by adding to the language component model an attribute that indicates the component mode in the corresponding architectural instance. We show the usefulness of modes by illustrating how they can help on describing systems from the *Automotive domain*. Also, in Section 4 we will discuss the next steps on possible approaches for the mode based analysis of systems.

Modern automotive systems contain a continuously increasing number of software components that must assist in a big range of operations. These include critical vehicle functions (ABS systems, road repair, etc.) or other less relevant to the vehicle primary function but of importance for nowadays client necessities (for example, road sights, infotainment, etc.). Moreover, these operations are continuously activated and deactivated and component reconfiguration is set up dynamically in a self-organising way. Also, due to advances in mobile technology communication is very complex in automotive software systems, where communication happens within the vehicle (intra vehicle communication), between vehicles (inter vehicle) and between the vehicle and the environment (vehicle-environment). This variety in nature, number, communication and dynamicity makes service-oriented techniques a natural choice for coping with the engineering of automotive systems. We base our work on one of the case studies for the *GC2 EU Project Sensoria (Software Engineering for Service-Oriented Overlay Computers)* [18]. The variety in the possible scenarios in this context provides a interesting testbed for the introduction of modes.

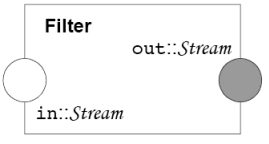
In Section 2 we give a brief introduction of the ADL language Darwin. Then in Section 3 we present modes and apply them for the description of a case study from the automotive domain. In Section 4 we discuss possible approaches for their formalization and how to they can help on the analysis of system properties. Section 5 concludes the paper with final remarks and future work.

2 Darwin

Darwin is a declarative component-based ADL. It supports a hierarchical model, tractable and it is accompanied by a corresponding graphical notation. The overall objective is to provide a soundly based notation for specifying and constructing distributed software architectures [9].

Component Model. The central abstractions managed by Darwin are components and ports. Ports are the means by which components interact. Ports

represent services that components either *provide to* or *require from* other components. A port is associated with a *type*: the interface of the service it provides or requires. Figure 1 shows the textual and graphical representation in Darwin of a filter component which provides an `out` service (filled circle) and requires an `in` service (empty circle), both of type *Stream*.




```
interface Stream {}

component Filter {
    require in:Stream;
    provide out:Stream;
}
```

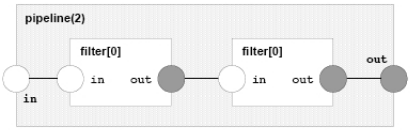
Fig. 1. Filter Component in Darwin

Bindings define a one-to-many mapping relation between provided and required ports. A binding associates the service provided by one component with the service required by another. A provided port may have many required ports bound to it. A required port can be bound to at most one provision. Darwin does not have any special construct to model connectors. Instead connectors, whenever required, are modelled using components and ports.

Components may be nested within composite components to form hierarchical structures. Composite components hide the complexity of the contained structure allowing the specification of the architecture in a varying level of detail. Regular bindings are allowed only between components of the same container. Bindings crossing container boundaries are indirect through port aliases (inward and outward bindings, Figure 2). A port alias is a port in the container component that exports a port from an inner component. Port aliases offer control over the scope of ports in nested constructs while preserving the benefits of nesting. Figure 2 shows a composite component for a pipeline that is obtained from two filters.

Behavioural Specification. The behaviour of components in Darwin is specified both graphically as a Labelled Transition System (LTS) and textually using the Finite State Processes (FSP) notation [12]. The behaviour of an architecture in Darwin is the composition of the behaviours of its individual component constituents, i.e. the parallel composition of the respective LTSs. The resulting LTS can be checked for such properties as the preservation of system invariants or the existence of deadlocks.

Dynamism. Darwins concern in supporting dynamic structures is to capture as much as possible of the structure of the evolving system while maintaining its purely declarative form. Architectural modifications at runtime may cause




```
component Pipeline(int n) {
    require in:Stream;
    provide out:Stream;

    array filter[n]:Filter;
    forall i:0..n-2 {
        inst filter[i];
        bind filter[i].out -- filter[i+1].in;
    }
    bind in -- filter[0].in;
    bind filter[n-1].out -- out;
}
```

Fig. 2. A composite component with an inward (`in` port) and an outward (`out` port) binding

disruption to behavioural aspects of the system such as triggering a deadlock. In [10], it is stated that changes to a systems structure may only be performed when the components involved in the changes are in a quiescent state. A component is considered quiescent if it is not in the process of exchanging application messages with its environment.

Mode Extension. The extension of Darwin with modes is obtained by adding a new attribute to components (boxed names in Figure 3) that indicates the mode in which the component is in the corresponding architectural instance (see Section 3 for the case study details). In the case of basic components, the mode identifies the state of the component. For composite ones, the mode for a composite component is directly related with the modes of its constituents. We assume that each component is in one mode at a time. Ports in gray color mean that the interface port is not "enabled" for binding to another component.

3 Modes for SA: A Scenario for the SENSORIA Automotive Case Study

In this section we introduce an approach to the use of modes at the architectural level by applying it over a case study from the automotive domain. The case study is taken from the *Sensoria Project Automotive Case Study* [18]. The Sensoria case study presents some typical automotive scenarios as they might be available to drivers in the near future. We derive three scenarios which are used to identify the modes and transitions needed to describe the desired evolution of a specific vehicle subsystem. The three scenarios are:

– *Road Sights Scenario:* The driver has subscribed to the dynamic sight service. The vehicles GPS coordinates are automatically sent to the dynamic

sight server. The dynamic sight server searches a sight seeing database for appropriate sights and displays them on the in-car map of the vehicle navigation system.
- *Low Oil Level Scenario:* During a drive, the vehicles oil lamp reports low oil levels. This triggers the in-vehicle diagnostic system. The diagnostic system reports a problem with the pressure in one cylinder head and sends a message with the diagnostic data as well as the vehicles GPS data to the repair server. The service discovery system identifies an adequate repair shop in the area. The repair shop coordinates are sent to the vehicle guiding system to direct the vehicle to the shop.
- *Accident Scenario:* Due to a collision on the route, an automated message is triggered and sent to the accident assistance server that contains the vehicles GPS data. Approaching vehicles are warned about the accident ahead through wireless messages suggesting alternative routes to avoid traffic jams.

In the next section we present a specific vehicle subsystem, the *Route Planning Subsystem*. We will describe the modes for this subsystem and show how they are used to assist modeling architectural instances for the different scenarios we have introduced.

3.1 Route Planning Modes

Our case study is a Route Planning Subsystem (RPS) for a vehicle, which is in charge of providing guiding indications to the driver. The RPS has three possible modes of operation that are specified with Darwin in Figure 3 (described below). To simplify the example we omit port names and port types which are not relevant in this case.

The RPS architecture is composed of three basic components and is shown in Figure 3:

- *Planner (P)*: This component is in charge of determining the solutions for the trip to be done by receiving routing information from the environment and sending it to the User Prompt. Depending on its mode the Planner may send also planning information to the environment for example to guide another vehicle.
- *User Prompt (UP)*: This component receives the information from the Planner and follows and provides the User Interface with the information relative to the real time progression of the trip (i.e. actual position, next turns, etc.).
- *User Interface (UI)*: This component handles the information arriving from the User Prompt and how it is visualized by the Driver. Depending on its mode the User Interface can be reconfigured to connect directly to the environment.

Each diagram in Figure 3 shows the acceptable configurations of the RPS in a specific mode. The interpretation of these diagrams can be dual. In the first place, a composite component mode can be seen as constraining the instances,

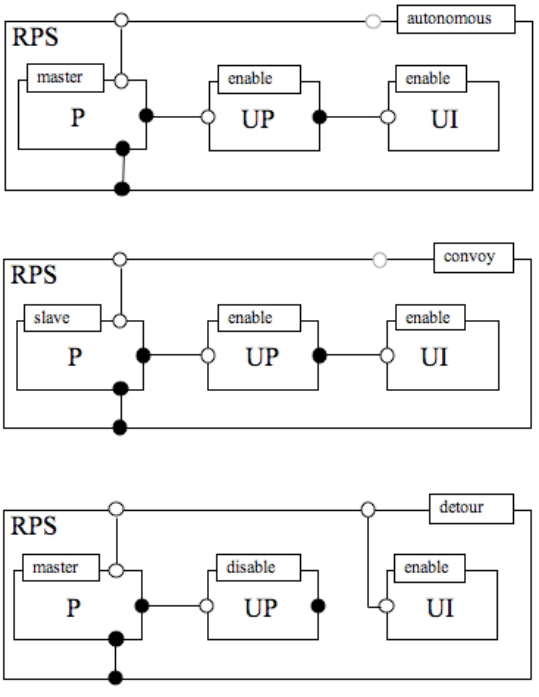


Fig. 3. RPS modes

bindings as well as the modes of its inner components. On the other hand, it can be interpreted as the modes and bindings of the basic components determine the valid mode for the corresponding composite component.

The first RPS mode is the `autonomous mode`. In this mode the driver indicates his destination and the route planner proposes the best route for him. The second mode is the `convoy mode` where the driver has to follow another vehicle from whom is receiving the indications to destination. And finally, the third one is the `detour mode` where the route planner is overrode by an external authority that guides the driver to a detour and in this way avoiding some problem in the route (i.e., an accident or works in the street). In more detail:

- `autonomous`: This mode represents the scenario where the vehicle is planning the trip autonomously (Planner in `master mode`), for example by using the information provided by a *GPS* system in the vehicle or internal information already present in the Planner. As you can see, in this mode the Planner is in `master mode`, and User Prompt and User Interface are in `enable mode`. For User Interface, `enable` is the only mode allowed for this component. In `autonomous mode` we have only intra vehicle communication.
- `convoy`: This mode represents the scenario where the Planner component is guided by the information received by another vehicle in front of him.

The binding to the master vehicle is done through a binding between the required and provided port aliases of the vehicle Planners (see Figure 4). In `convoy mode` the Planner is set to `slave mode` identifying the change in the configuration. In this mode we have inter vehicle communication.

- `detour`: This mode is considered for scenarios like the *Accident* and *Low Oil Level* scenarios. In this mode the User Prompt is in `disable mode` and the User Interface is reconfigured to attend instructions from an external system (Police or Highway Emergency System for example). Also, the Planner maintains its `master mode` as the vehicle may be used by the external system as relaying point for additional planning information to be passed to other vehicles (that switch to `convoy mode`) that are approaching, but are further away and may have more alternatives to choose from. The external system sends instructions directly to the driver redirecting the vehicle to avoid traffic problems or to guide him to the required assistance. In this mode we have vehicle-environment communication.

The point is that in the highly dynamic domain of automotive systems, there are different scenarios that this subsystem must handle that imply reconfigurations changing the architectural style of the system, i.e., the architectural constraints that configurations must satisfy. We use modes to capture these changes of scenarios. A mode is related to a (possible set of) configurations which characterized it. The diagrams in Figure 3 define the RPS component mode types, while in Figures 4, 6 and 7, system configuration instances using the RPS subsystem present different alternatives of these modes. For example, Figure 4 shows a configuration for three RPSs of three cars where the head is in `autonomous mode` and the other two are in `convoy mode`. Figure 6 shows an instance of the `detour mode` where the *Highway Emergency System* is taking control of the planner and sending additional information to the Planner (the ??? mode name indicates that it is not relevant for the example). Finally, Figure 7 shows the `autonomous mode` with its planner connected to an external GPS system.

Figure 5 shows a way of modeling multiple alternatives using $\star$ symbol in the mode attribute (although, other less restrictive alternatives can be proposed) meaning that that component can be in any mode. In this case, Figure 5 identifies the set of possible configuration instances using one RPS in `convoy mode` which can be connected to another RPS in any of the three possible modes. Note that all these configurations follow similar structural constraints.

Modes of a composite component depend on their constituents modes defining mode-based composition. Note that modes not only determine configuration but also coordination and communication mechanisms. In our case study, each one of the RPS modes requires a different type of automotive communication. Figure 8 shows the modes for the RPS and the possible transitions (i.e. reconfigurations) among them. This transition system at the architectural level can help in understanding how the scenarios can be related providing feedback for the validation of requirements.

It is worth noticing, that the relation between scenarios and modes is not one-to-one. For example, a system including the RPS can have a mode for the

Accident scenario that may combine the `detour` with the `convoy modes` of the RPS, where the external system only passes the alarm to the nearest cars approaching the accident zone, which in turn forward the alarm to the other cars behind them by using the `convoy mode`. These may be of help on providing feedback from the architectural level to the requirements by indicating that some scenario may need refinement in more detailed or specific ones.

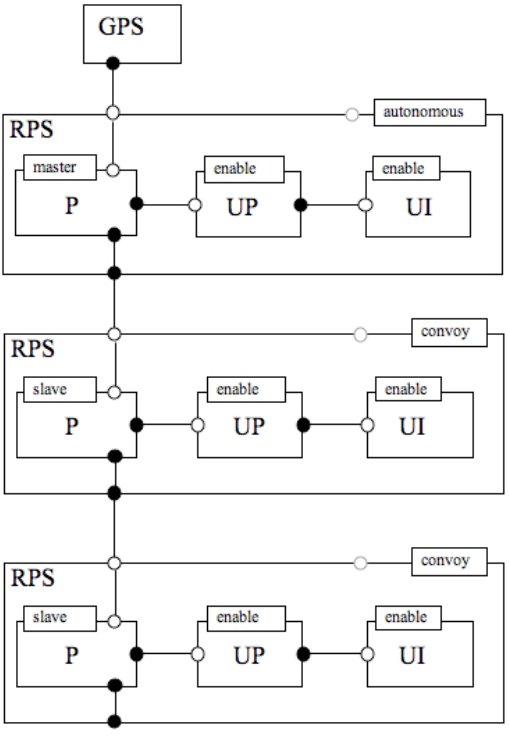


Fig. 4. Convoy mode configuration

With this example we have shown how modes relate scenarios with architectural configurations defining the structure of the system (and type of communication and/or coordination). Also, we can see how modes can help on specifying the possible reconfigurations that are allowed (or not) to handle the relationship among different scenarios (self-organisation), or on guiding the system to take repairing actions in case of some problem (self-healing/reparing). For example, if a system is in a mode where some component fails ending in a non valid configuration (i.e., non valid mode), then depending on the last mode it was, it can determine the resources and operations necessary to reach a valid (maybe the same) mode.

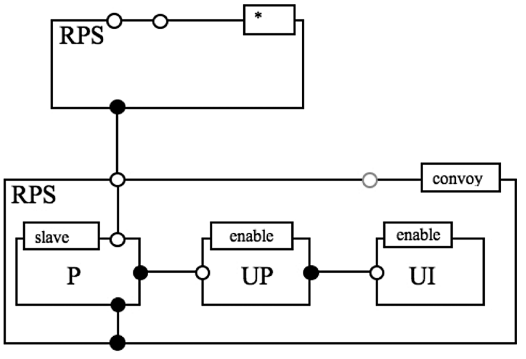


Fig. 5. Unconstrained Leader of a 2-RPS convoy

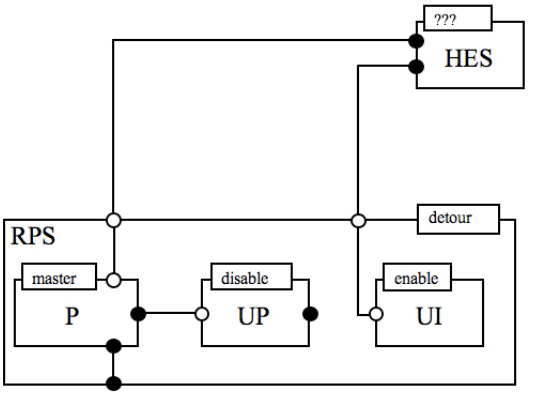


Fig. 6. Detour mode configuration

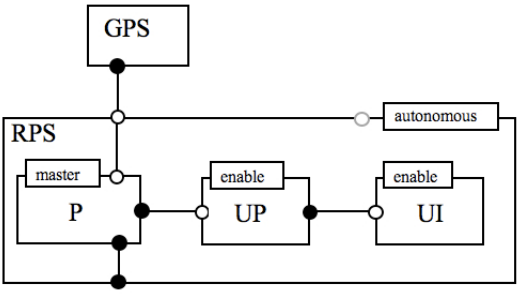


Fig. 7. Autonomous mode configuration

4 Discussing Modes

In Section 3 we have introduced modes over the Darwin language and shows how it is applied to the Automotive domain. We have seen how the mode abstraction

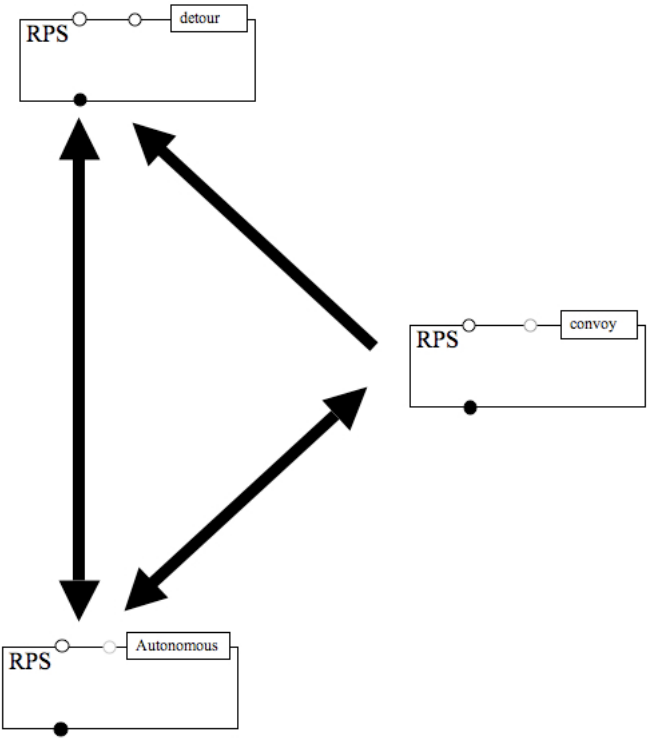


Fig. 8. RPS Modes

can help on closing the gap from requirements to the architectural level by a scenario-based approach which allow to capture the highly dynamics arising in configuration and coordination of service oriented systems. At the same time, this new notion of mode arises interesting questions that our research must try to answer. In this section we will discuss on some alternatives to study in our future work.

Mode Formalization. In Section 3.1 we have incorporated to Darwin, in an *informal* way, the notion of mode as an attribute of component interfaces. Our idea (at least initially) is to obtain a conservative extension of Darwin that respects its original semantics. In fact, in the original language the mode attribute can be modeled by including a dedicated required port to each component. This port is bounded to a special *mode component* whose only function is to indicate the mode in which the component is. This shows that Darwin has enough expressive power to capture the abstraction we needed. Although, incorporating modes as a first class primitive is fundamental to our goals of providing new language abstractions that can cope with the new requirements of global and service oriented systems.

Another future step for the formalization of mode is to study possible extensions of UML. The relation between scenarios and modes, may indicate that extending UML with modes may be useful for relating scenario-based notations with structural ones. One possible way of incorporation modes to UML can be via the definition of specific stereotypes.

Self-* and Reconfiguration Support. Given that self-organisation is fundamental for service oriented computing, another approach we are investigating for the formalization of modes is based on the work done in [4]. The work of [4] presents a software architecture based approach for the specification and verification of self-organising systems. A declarative method is introduced to describe systems underlying software architecture style and then use it to build and maintain system structure during its runtime evolution.

Specifically, based on Darwin component model, system valid structural and evolution constraints are described using the Alloy Language [7]. Alloy specifications consist of definitions of sets, relations among them, and constraints over sets and relations. We plan to extend the Alloy model for Darwin in [4] by adding modes as a new basic element of the model. One benefit of using Alloy is to use the analyser of Alloy models [8] that can be used to generate sample instances that conform to the model or to verify properties of behaviour over a given space of instances. We consider that by incorporating modes to the Alloy constraint model we may be able to identify more clearly configuration issues of self-organising systems.

Another motivation to apply the ideas from [4], is that it provides a method to specify reconfiguration rules over Alloy models as a constraint satisfaction problem. A set of configuration actions is generated when the structure of the system is no longer valid with respect to its architectural description due to either a scheduled change or a failure. Then, the execution of these actions should lead to an architectural instance that remains valid with respect to the style model. In the same way, we can think of using a mode-based approach to reconfiguration and self-organisation which we consider can help to manage the increased complexity is software systems, specially in highly dynamic reconfigurable systems.

Alloy allows us to add structural and evolution constraints to Darwin models obtaining more detailed characterizations of system configurations and their properties. For example, taking the diagram in Figure 5, we may be able to add a structural constraint that only allows `autonomous` and `detour` as the leader mode, exactly identifying the valid configurations with two RPSs. Anyhow, we do not think of Alloy as the final or best approach but as a first step and benchmarking of other languages we plan to study.

Analysis. A main goal for the introduction of modes is in helping on the verification and validation of systems. Our initial approach focuses on the study of techniques for analysing systems structure, but also our future plans will profit from previous experience on scenario-based synthesis of behaviour models from Message Sequence Charts (MSCs) for the identification and validation of modes.

This technique allows the user to specify scenarios in the form of MSCs that capture a desired set of actions, and then to combine them to form one or more state machines (LTS) [17] for deriving the tasks . This allows to use the LTSA tool [11] to analyse models for safety, liveness and temporal logic properties. We consider that the introduction of the mode abstraction in this context can help on reducing complexity and facilitating the validation of correctness between requirements and scenarios derived from modes at the architectural level. It is interesting to think that a component mode may be visible for the state machines that describe the behaviour of the component.

5 Conclusions

In this paper we propose to exploit the notion of modes at the architectural level, where modes are related with specific architectural constraints over the corresponding subsystem configurations. This allows us to assign specific modes to components defining their style and also allows us to specify the interactions and transitions among different modes.

We have introduced the notion of mode by proposing a case study from the Automotive Domain. Our goal is to provide flexible support for the description and verification of complex service oriented systems. We consider that the concept of mode helps on closing the gap between requirements and software architectures by using modes as a *scenario-based* abstraction to relate specific use cases with service configurations.

References

1. Garlan, D. and Shaw, M. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.
2. Garlan,D., Kramer, J., and Wolf, A. Woss '02: Proceedings of the first workshop on self-healing systems. New York, NY, USA, 2002.
3. Garlan,D., Kramer, J., and Wolf, A. Woss '04: Proceedings of the 1st acm sigsoft workshop on self-managed systems. New York, NY, USA, 2004. ACM Press.
4. Georgiadis, I. *Self-Organising Distributed Component Software Architectures*. PhD thesis, Department of Computing, Imperial College of Science, Technology and Medicine, University of London, January 2002.
5. Hofmeister, C., Nord, R., and Soni, D. *Applied Software Architecture*. Addison-Wesley, 1999.
6. Hofmeister, C., Nord, R., and Soni, D. Describing software architecture with UML. In *First Working IFIP Conference on Software Architecture*, San Antonio, Texas, February 1999. Kluwer Academic Publishers.
7. D. Jackson. Alloy: A lightweight object modelling notation. Technical report, MIT Lab for Computer Science, July 1999.
8. Jackson, D., Schechter, I., and Shlyakhter, I. Alcoa: The alloy constraint analyzer. In *International Conference on Software Engineering*, pages 730–733, Ireland, June 2000.

9. Magee, J., Dulay, N., Eisenbach, S., and Kramer, J. Specifying distributed software architectures. In *Fifth European Software Engineering Conference (ESEC95)*, Barcelona, September 1995.
10. Magee, J. and Kramer, J. Dynamic structure in software architectures. In *Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE4)*, ACM Software Engineering Notes, pages 3–14, San Francisco, October 1996.
11. Magee, J. and Kramer, J. *Concurrency: State Models and Java Programs, 2nd Edition*. Wiley, 2006.
12. Magee, J., Kramer, J., and Giannakopoulou, D. Analysing the behaviour of distributed software architectures: a case study. In *5th IEEE Workshop on Future Trends of Distributed Computing Systems*, pages 240–245, 1996.
13. Maraninchi, F. and Rémond, Y. Mode-automata: About modes and states for reactive systems. In *European Symposium On Programming*, volume 1381 of *Lecture Notes in Computer Science*, pages 249–250. Springer Verlag, March 1998.
14. Medvidovic, N. and Rosemblum, D. Assesing the suitability of a standard design method. In *First Working IFIP Conference on Software Architecture*, San Antonio, Texas, February 1999. Kluwer Academic Publishers.
15. Medvidovic, N. and Taylor, R. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, January 2000.
16. Perry, D. and Wolf, A. Foundations for the study of software architecture. *ACM SIGSOFT*, 17(4):40–52, 1992.
17. Uchitel, S., Chatley, R., Magee, J., and Kramer, J. System architecture: the context for scenario-based model synthesis. In *Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE'04)*, ACM Software Engineering Notes, pages 33–42, Newport Beach, CA, 2004.
18. Angelika Zobel. Sensoria deliverable 8.0: Description of scenarios for the automotive case study. 2006.