Using Specification Models for RunTime Adaptations*

Sébastien Saudrais, Athanasios Staikopoulos, and Siobhán Clarke

DSG, Trinity College of Dublin, Ireland
{sebastien.saudrais,athanasios.staikopoulos,siobhan.clarke}@cs.tcd.ie

Abstract. For a myriad of reasons, modern applications face constant change to their requirements and working environment, requiring them to adapt accordingly. Increasingly, such adaptation is even required during runtime. In Model-Driven Engineering (MDE) approaches, models are first-class entities in the development of applications, though they have not, to date, been sufficiently taken advantage of in runtime adaptation specification. In many existing approaches, designers are required to consider the execution model when specifying any runtime adaptation, forcing them to understand the different formalisms of both the execution model and the specification model. The focus of this paper is to show how runtime models to monitor an application's execution can be derived efficiently from the specification, and how they support the designer in considering the application's execution in the same formalism as the specification.

1 Introduction

Model-Driven Engineering (MDE) promotes the use of models throughout the development of software. The underlying idea is to promote models as the primary artefacts of software development, making executable code a pure derivative of those models. Models containing adaptation specifications are an increasingly important and frequently encountered part of the development process. This is especially true for modern applications that need to adapt at runtime to cope with constant changes to their requirements and operating environments. Such changes have to be considered at the specification phase, and the models validated before they are transformed to real code. However, despite the importance of specification models, they have, to date, been ignored during the execution of the software. Once the code is generated, the specification models are no longer used with the potential loss of information that would be especially valuable during adaptation specification.

* This work has been carried out within the FP7 project ALIVE IST-215890, which is funded by the European Community. The author(s) would like to acknowledge the contributions of his (their) colleagues from ALIVE Consortium (http://www.ist-alive.eu)

A further difficulty emerges during the process of adapting the execution. While the adaptation may be based on a specification model, the actual adaptation is necessarily performed either by hand on the application's code, or requires a complete regeneration of the system since it is unlikely to match the old specifications. Working directly with the application's code means a move to a different formalism from that of the specification. This change between formalisms has a number of disadvantages. The first is that the adaptations performed in the new formalism must be validated against those in the specification model. An automatic generation of the entire module that is to be adapted can ease this checking, but this needs to be coupled with a reverse-engineering technique to reproduce the specification model. The second disadvantage is that while the architect of the software knows the specification formalism, he is not always familiar with the implementation's one(s). If there are problems with the adaptation at runtime, he has to be able to understand the second formalism in order to solve the errors, or work closely with an implementation team member. Either approach is likely to be difficult where an application's execution context often changes, requiring manual adaptation at runtime. It would be easier for the architect to visualise a snapshot of the actual execution in the same formalism as the specification.

The approach proposed in this paper takes advantage of the specification models during the execution. A runtime model is generated from the specifications, which supports the monitoring of the execution required to supply sufficient information to apply adaptations directly on the specification models of the software. The runtime model contains the information needed to trigger the adaptation and is created based on the adaptations defined at the specification. At runtime, when an adaptation needs to be performed, the specification models are updated to correspond to the actual execution and the adaptation is performed on the up-to-date specification models. The new configuration of the application is, finally, generated from this new specification models. The approach allows a designer to use a single language (the specification language) to design the software and to interact with it during the execution.

Our approach is applied within the ALIVE project[1], which is funded by the EU under Framework 7. ALIVE's objective is to enrich service-oriented architectures with coordination and organisation mechanisms often seen in human and other societies. ALIVE has three levels of design: organisation, coordination and services. The design is hierarchical, and concepts in one level refer to or reuse concepts defined in another. The framework supports adaptations within each level, and caters for adaptation that may affect all design layers because of interlayer dependencies and rules. The remainder of the paper is organised as follows: Section 2 motivates the use of runtime models for the ALIVE architecture. Section 3 presents the different metamodels and how runtime models are generated. Section 4 illustrates the approach with the ALIVE Crisis Management scenario. Section 5 compares our approach with related work and discusses the advantages of runtime models. Finally Section 6 concludes.

2 Background: Use of Runtime Models

Runtime models provide a dynamic view of parts of a system and of operational data observed at runtime [2]. Rules and constraints, which need to be preserved during the system's execution, are captured in runtime models and are validated by the execution framework. Changes to the executing system are propagated to the runtime and vice versa.

The use of runtime models permits the complete or partial reuse of the design models and their adaptation. As observed by France and Rumpe [3], runtime models can be useful in the adaptation of systems through the process of:

- runtime observation;
- adaptation;
- and design update.

System executions utilise real code to perform the functions prescribed by the models. The use of a runtime model, based on *runtime observation* of the execution, supports the creation of an abstract view of the execution, which in turn may be used by an adaptation module. The set of events observed in this process should be generated from the design models, preventing errors that would emerge from manual implementation.

Automatic *adaptation* of the system may be performed that depends on the execution's observation. During the design phase, patterns of adaptation are usually defined by the architect to face the environment's changes, by taking account of some critical execution events. When a predefined set of events is triggered, the adaptation is performed on the specification models and then changes are applied in the generated execution. In this manner, the adaptations defined at the specification are also used at runtime.

Finally, the initial designs may be *re-designed* using the runtime models. The architect, by looking at the runtime models, may decide to modify or add new functionalities to the system. These modifications are then transferred to the execution by production of runtime changes.

3 From Specification to Runtime

Our approach uses the adaptation rules defined during the specification directly when needed at runtime. For the purposes of this paper, we assume these adaptation rules have been proven during the specification of the application and are understandable by the architect. We use the adaptation rules to generate runtime models that will monitor the application and launch an adaptation when needed. Only a subset of the information contained in the adaptation rules is required to produce the runtime models. This subset is composed of the elements of the model that need to be monitored to trigger the adaptation and those that need to be read to perform the adaptation. An overview of the approach is presented in Figure 1. The runtime models are generated from the specification models

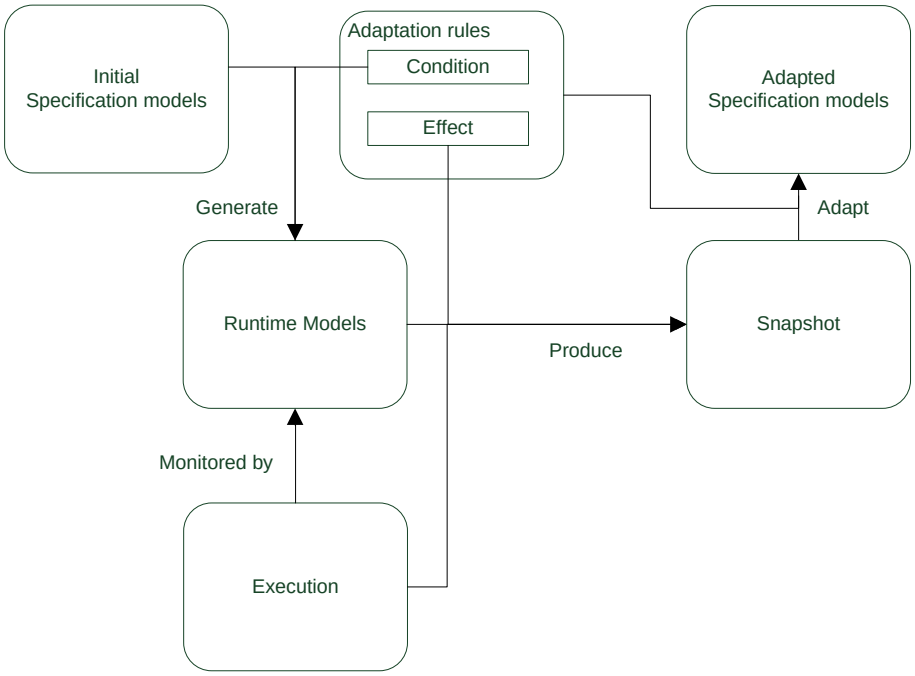

Fig. 1. Approach overview.

and the enabling conditions of the adaptation rules. During execution, the runtime models monitor the application's code. When an adaptation is triggered, the adaptation rules and the runtime models are used to provide a snapshot of the application containing only the part involved in the targeted adaptation. The adaptation is then performed on the specification models obtained from this snapshot. A new configuration of the code is obtained from the new version of the specification models using the same code generation techniques used in the initial generation of the software. In the next section, we define a metamodel for runtime based on adaptation rules. An algorithm is then presented to automatically generate the runtime models. Finally we explain how the adaptation can be performed.

3.1 Adaptation rules

Adaptations specify the appropriate reaction to changes that can occur at runtime and that have an impact on the software. An adaptation rule is composed of a condition and an effect. The condition contains the information triggering the adaptation: for example, an occurrence/absence of an event, a comparison of an object with a value or a number of occurrences of an event. The effect explains how the adaptation is performed and is written in the model transformation language used to specify the adaptation. It explains how the adaptation is ap-

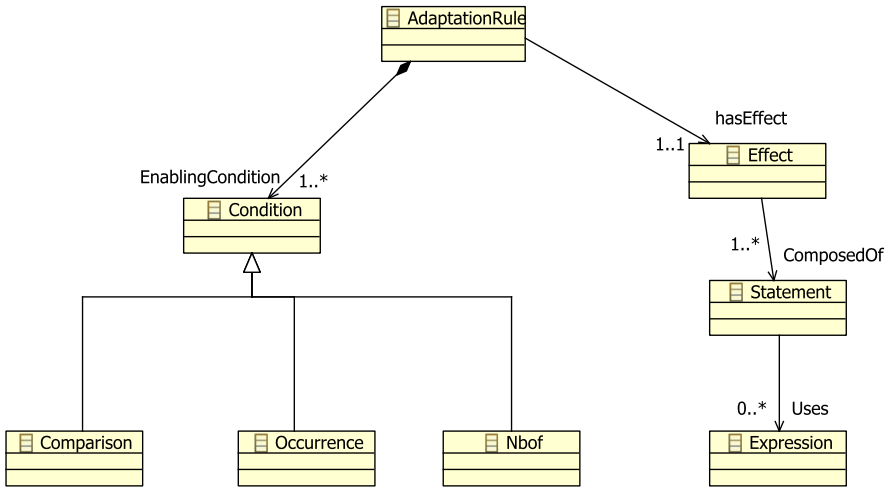

Fig. 2. Adaptation rules concepts.

plied and which part of the application is involved in the adaptation. Figure 2 presents the (simplified) metamodel of the adaptation rules in our approach. The condition is a superset of the possible conditions and can be extended by other types. The effect part only contains the expressions, i.e. the elements of the specification models involved in the adaptation. These elements will be manipulated and updated by the adaptation.

3.2 Runtime Models

Runtime models contain the information needed to support an adaptation when it must be performed. They link the implementation, the specification models and the adaptation rules. We have defined a generic metamodel to represent the different relations between these three elements. The runtime models have the same objective as the condition part of the adaptation rules: triggering the adaptation. As illustrated in Figure 3, the runtime metamodel has as base the metamodel of the adaptation rules relating to the conditions and is extended with information about the platform to monitor the software. The left part of the metamodel corresponds to the enabling condition and the right part to the link with the platform. Each *adaptation rule* has different *triggers*. These triggers are of the same type as the enabing condition and so can be extended with other types of conditions. The class *Element* references the elements of the specification model. For each element to be monitored, the corresponding implementation is obtained through an *AccessPoint*. The access point provides the means to access the value of the element in the implementation, for example, via a method to access the value or an exchange of messages. Only some of the possible types of access points are presented is the metamodel, *method* and

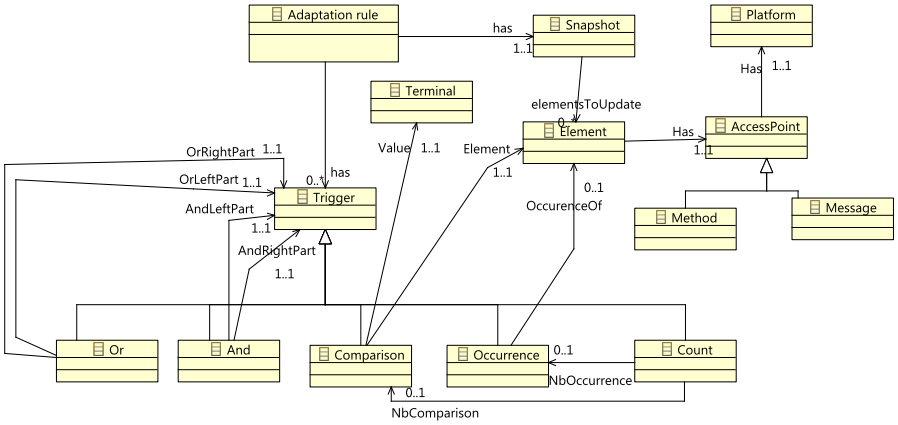

Fig. 3. Runtime metamodel.

message, but extensions can be easily made depending on the requirements of the software.

The runtime metamodel is also used for the snapshot through the class *snapshot*. The purpose of a snapshot model is to give an updated view of the software and links elements of the specification to the platform. It contains only a set of elements involved in the effect part of the adaptation corresponding to the right part of the Figure 3: the *Element* and *AccessPoint*.

3.3 Generation of the Runtime Models

Our approach includes an automated process to apply the adaptation rules on the specification models during the execution. The architect may also add new adaptation rules during the execution that will need to be incorporated in the runtime model without human intervention. The runtime model is automatically generated from the specification, the adaptation rules and the platform specifications. The generation algorithm has two steps. The first step is to select the different classes from the specification that are used by the adaptation rules. For each enabling condition, the set of elements required for monitoring is identified. The trigger is created using the enabling condition of each adaptation rules. The set of elements is then reduced to avoid duplicate elements. This step is designed to ensure that the runtime model contains only the elements required to support adaptation, and is therefore smaller and more efficient to process than would be a runtime model of the complete specification.

The second step is to identify the access point in the implementation. This step will use information from the specification and platform specifications. The access point is attached to the element in the runtime model and needs code to be generated before it can access the implementation. As software modules do not have a single implementation language, the different access points can be

implemented in different languages. The runtime model is updated with values obtained through the access point during the monitoring process. The actual implementation of the runtime model is done using Kermeta [4]. Kermeta offers calls to Java classes with interfaces to other languages. For each access point, a Kermeta method is created with the intermediate code in Java, if needed, to make the link with the implementation. This access point can be regenerated at runtime if the access point is changing during the execution.

3.4 Adaptation at runtime

Once the runtime model is generated, its monitoring capabilities are executed and the runtime models are automatically updated. When an adaptation is triggered, the specification models are updated with the actual values contained in the runtime model and a snapshot is created. The process of creating the snapshot is based on the same algorithm as the generation of the runtime models but where only the current adaptation's effect's expressions are considered. Once the snapshot of all useful information is created, the adaptation can be performed on the specification models using the adaptation rules. Once the adaptation is performed, the new implementation is generated using the same method as for the first generation of the implementation.

The architect can also use the snapshot process to create a visualisation of the actual execution. This visualisation may consider only a subset of the application and some adaptation rules. The snapshot process is used in this case to support the architect adding new adaptations that take account of the actual execution of the software. A new runtime model is then generated to incorporate the new enabling conditions of the added adaptation rules.

4 Evaluation: Crisis Management Case Study

In this section we show how runtime models are exploited in a use case from the ALIVE project that describes a crisis management scenario defined by Thales[5]. We first present a high-level summary of the specification used in ALIVE applications: metamodels and adaptation rules. We then apply our approach on the example.

4.1 ALIVE's specification

Three metamodels describe the ALIVE layered architecture: organisation, coordination and services. Each one has a different level of abstraction and its own adaptation rules. Model transformations are defined from the metamodels and are bi-directional between the different layers.

Metamodels The organisation level provides context for the two other levels, supporting an explicit representation of the organisational structure of the

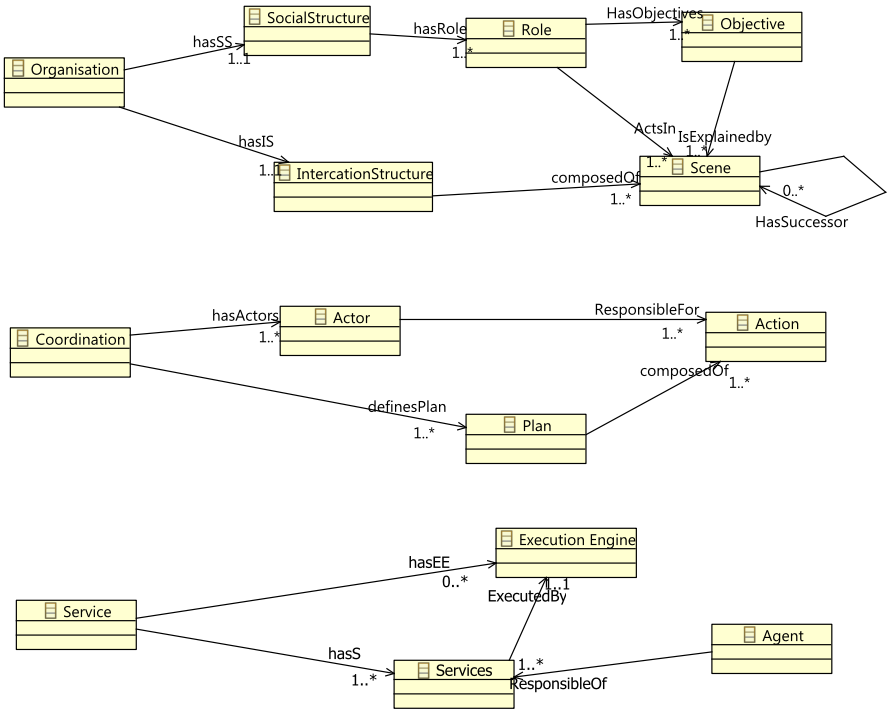

Fig. 4. Abridged ALIVE metamodel.

application. It presents the roles involved in the organisation and their interrelations. Each role has a set of objectives for which it is responsible. A social structure construct contains this information, while an interaction structure construct gives the order of occurrence of the different objectives using scenes. The organisation metamodel is depicted on top of Figure 4.

The coordination level uses the organisation level as a starting point, and provides coordination plans to achieve the objectives of the organisation. Its metamodel has the concepts of plans and actors. Actors correspond to the synthesis of roles to agents. As agents can play different roles in an organisation, the concept of actor captures the goals of an agent playing a specific role. The coordination plans are made more precise than the interaction structure by introducing a lower-level concept of actions. For example, an organisation model may describe payment while a coordination model will describe cash, paper payment or electronic payment. The coordination metamodel is shown in the middle of the Figure 4.

The service level supports the semantic description of services and the selection of the most appropriate service for a given task. It connects the executing environment and the two other levels, which are input to the service level. It

contains agents, the different services and the execution engine. The agents are connected to the actors of the coordination. The services are refinements of the coordination actions, for example, the electronic payments become different services from each bank that offers an electronic payment. The execution engine is responsible for service execution against the plan described in the coordination model. The service metamodel is presented on the bottom of Figure 4.

Adaptation rules The adaptation rules of the different levels are based on the occurrence of specific events or properties. An adaptation is triggered if certain conditions are verified. Properties from all three levels may trigger an adaptation to an ALIVE application. Depending on the level where the adaptation trigger occurs, the adaptation will have a different impact on the application. Adaptations affecting the service level will be performed without impacting the two others. An adaptation that impacts the coordination level is also likely to impact the service level. An adaptation at the organisation level is likely to impact all three levels, in particular the relations between agents as their roles and objectives may have changed. All the adaptations, for the three levels, are expressed with the same language.

4.2 Initial specification

The use case describes a system to handle emergency situations.The organisation includes a police station, first-aid station, emergency centre and fire station. The main objective of the fire station is to evacuate people. Other objectives of the different roles are to identify the emergency location, to provide an ambulance service and to regulate traffic. These objectives are delegated through the arrows to the other roles as depicted on the top part of Figure 5. The coordination level describes a generic plan to achieve the evacuation objective. The plan describes the different steps of the evacuation: selection of the transport vehicle, provision of an itinerary to the accident location, collection of injured people, provision of an itinerary to the hospital. This plan is a generic one that can be used and refined by the service level. The middle part of the Figure 5 shows the coordination level.

During an accident, the fire station makes decisions relating to the evacuation of people. The evacuation plan is called at the service level. Specific services are used: an ambulance, the emergency centre, itinerary software and the police station. The bottom part of the Figure 5 shows the different services in play.

Adaptation rules are defined to handle common failures that can happen to this type of application: traffic jams, engine failure, escalation of the danger level. For example, Figure 6 illustrates the code for different adaptations to the crisis management case study. Adaptation A1 concerns engine failure. Depending on the position of the ambulance and on the level of risk for rescued people, different choices can be made: ambulance change, people transfer or ambulance repair. This adaptation concerns only the service level. Adaptation A2 concerns a failure relating to difficulties encountered by the rescue personnel in achieving

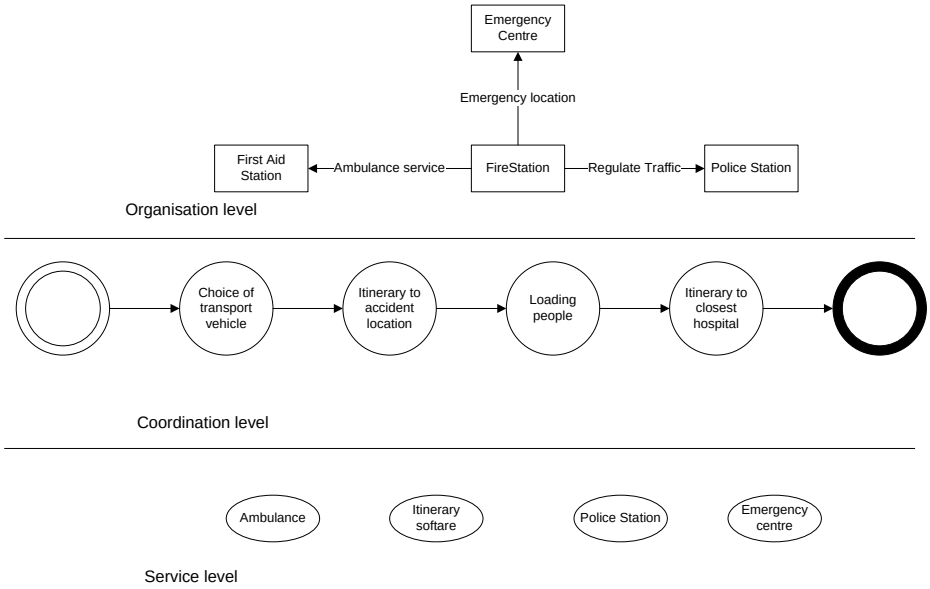

Fig. 5. Initial specification of the crisis management scenario.

their objectives. The ambulance has a problem and no other terrestrial vehicle, as needed by the plan, is available. Alternative transport has to be considered, either by air or by sea and a new plan has to be given to the service level. This adaptation concerns both the coordination and the service levels. Adaptation A3 is triggered when the coordination level is unable to find a new plan when the ambulance fails. The organisation level needs to adapt to the situation and may incorporate new roles. In this case, private companies can be added, like private helicopters, to evacuate people. While this adaptation will impact the three levels, some parts of each level can be reused, like the abstract plan and different services.

4.3 Runtime models

The runtime model obtained from the specifications to support adaptation A2 is depicted on Figure 7. The enabling condition from adaptation A2 has the occurrence of the message *ambulance_blocked* and the occurrence of the properties *no_repairable* and *no_terrestrial_vehicule_available*. This trigger is added to the runtime model. The next step in the creation of the runtime model is to link with the implementation. For the purpose of the evaluation, we are using the Thales simulation workbench to simulate the different services. For each of the three elements, the corresponding access point is provided according to the platform specification. The ambulance provides its status and position through the methods *Ambulance_position* and *Ambulance_status*. The emergency centre provides

```
A1:Condition:
        Message={ambulance_accident}
    Effect:
        IF (major_injuries) THEN
                send_another_ambulance
        ELSE
                send_repair_vehicule
        END
A2:Condition:
        Message={ambulance_blocked} AND no_repairable
        AND no_terrestrial_vehicule_available
    Effect:
        IF (Action_aerial) THEN
                create_plan_aerial_evecuation
        END

A3:Condition:
        Message={ambulance_blocked} AND no_repairable
        AND no_vehicule_available
    Effect:
        add_role(private_transporter,ambulance_service)
        create_plan_private_evacuation
        add_service(private_helicopter)
```

Fig. 6. Adaptation rules.

the transports' availability through *Transport_Availability*. The three methods are implemented in Java and interact with the workbench using messages.

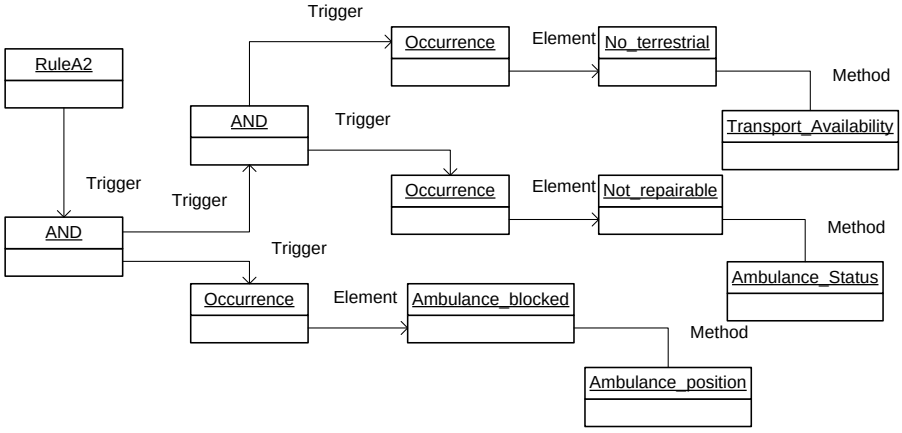

Fig. 7. Runtime model.

Once the adaptation is triggered, a snapshot of the part of the application of interest to the adaptation is made. This snapshot is represented on Figure 8. The *create_plan_aerial_evecuation* call needs nothing at the coordination level as a new plan is created. Once the evacuation plan is created, the status of

different aerial transport is needed to select one available to execute the plan. The snapshot contains two elements *helicopter* and their corresponding access point. The specification models are updated using both the runtime model and the snapshot model, and the adaptation is performed.

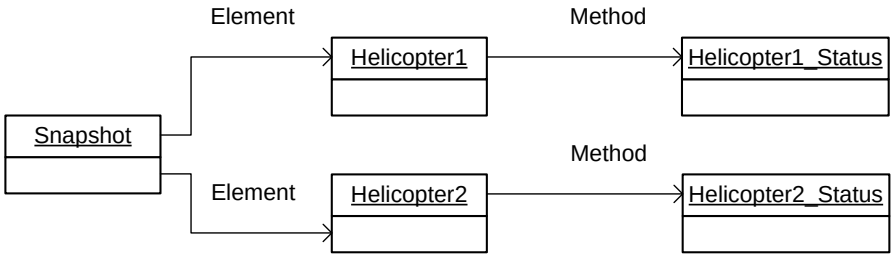

Fig. 8. Snapshot model.

The new configuration is then produced from the adapted specification models as shown on Figure 9. The plan is modified and the services *helicopter1* and *helicopter2* are added.

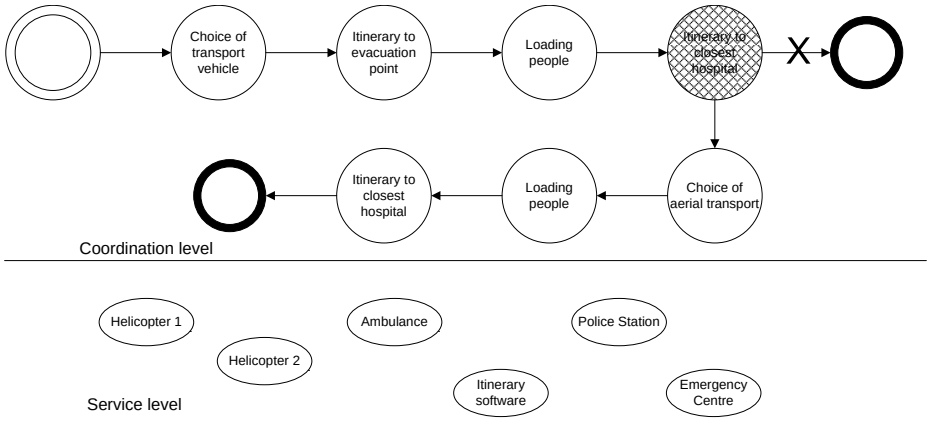

Fig. 9. New specification.

5 Discussion and Related Work

Discussion Our hypothesis related to the efficiency of this approach is based on an assumption that only a subset of the application is subject to adaptation. The approach generates a runtime model based on only those elements required

to support adaptation, thereby reducing its size relative to the full application, making it more efficient to work with. Given this, our approach is therefore well-suited for applications where a big part of the specification is static (in other words, not expected to require adaptation over the execution of the application) and mainly used to understand the objectives of the application. A good example of this is ALIVE's organisation level. The static part of ALIVE applications do not, therefore, require permanent monitoring at runtime. In applications where adaptation rules cover a bigger part of the specification, the runtime model will be a correspondingly bigger proportion of the full specification, reducing the extent of the efficiencies. Further experiments are needed to identify the maximum coverage percentage that will still result in efficiency benefits in the monitoring process. Figure 10 illustrates the placement of the size of runtime models for ALIVE applications against possible coverage of adaptation rules over the specification models. The evaluation runtime model contains 10 elements to monitor when the specification models contain 50 elements. The snapshot models need an average of 10 elements to update.

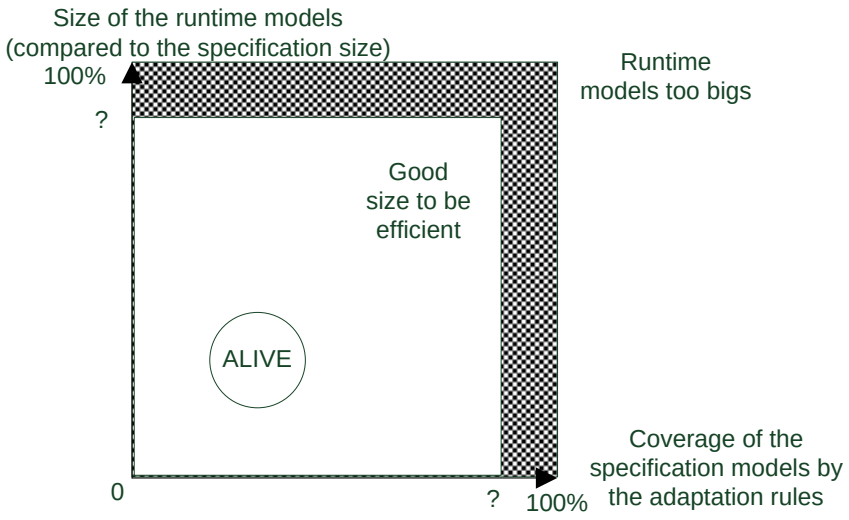

Fig. 10. Adaptation rules coverage of application.

A second potential limitation is the feasibility of performing the adaptation on the models at runtime. If the application is centralised, different transformation languages can be used but as modern applications are often distributed, including ALIVE applications, the adaptation may also be distributed. Few transformation languages focus on ensuring a light execution footprint, which may be problematic in a distributed setting. The current version of our runtime models is implemented using Kermeta but it requires at least a Java virtual machine. A more optimal approach would be a transformation language than can be

interfaced with multiple implementation languages but without any constraints on the execution platform.

Related work Many approaches adapt applications using a different formalism than the specification. In such approaches, the adaptation module can be seen as a runtime model because it has its own representation of the execution. However, the gap between the specification and the execution requires a re-test of the adaptation even though it has already been proven at the specification phase. For example, Pickering et al [6] propose an approach to manage complex systems with runtime models. The management of the systems is defined in specification models that are transformed to runtime models in a specific infrastructure, IBM WebSphere and so are expressed in a different language than the specification. Rainbow [7] provides an adaptation framework based on an abstract architectural model to monitor runtime properties to accommodate resource variability, system faults, etc. In our approach, runtime model is built on dynamic parts of the specification models and not on an abstract model to apply architectural adaptations.

Other approaches are in a position to use the specification models at runtime because of the specific platform they provide. For example, Fractal [8] monitors the execution and performs the adaptation using the reflexivity of its own language. The ALIVE approach uses standard languages, and therefore assumes different languages at the implementation level. The Diva [9] approach considers both design and runtime phases of development. At design time, an application is modelled using a base model (containing the common/core functionalities), a set of variant models (capturing the variability of the adaptive application) and an adaptation model (specifying which variants should be used according the rules and current context of the executing system). At runtime, the models are processed by model composers that produce the system's configuration. The application is fully monitored and is based on the reflexivity of the underlying language.

6 Conclusion

In this paper, we presented an approach to using specification models to derive efficient runtime models that support runtime adaptation. We defined a metamodel for runtime models based on adaptation rules. Runtime models are automatically generated from the specification. Adaptation is performed at runtime using the specification models. The approach is designed to address two main objectives. This first is to use the same formalism for adaptation both at design and runtime. This reduces the potential for introduction of errors, by avoiding the transformation to another formalism, and aids the architect's understanding of the execution without requiring him to learn additional languages. The second objective is to optimise the efficiency of the runtime models. This is achieved as the runtime models monitor only the parts of the application that are involved

in adaptation. A snapshot is taken of only those elements of interest to the adaptation. A full snapshot is available when the architect wants to have an overview of the system or wants to introduce new adaptation rules. The automation of the generation of runtime models supports this addition of new adaptation rules. We illustrated an evaluation of the approach through application on a case study.

Our approach is currently being tested on different industry-specified use cases covering different types of implementations. Further refinements of the adaptation at runtime implementation are ongoing. Further, we are investigating alternatives to using Kermeta to avoid requiring the use of a Java virtual machine. Possibilities require choosing a transformation language with tool support that would not restrict the model to a particular platform.

References

1. ALIVE: Coordination, organisation and model driven approaches for dynamic, flexible, robust software and services engineering, http://www.ist-alive.eu/
2. Calinescu, R.: Methodology for the model-driven development of self-managing systems. In: CF '08: Proceedings of the 2008 conference on Computing frontiers, New York, NY, USA, ACM (2008) 115–116
3. France, R., Rumpe, B.: Model-driven development of complex software: A research roadmap. In: FOSE '07: 2007 Future of Software Engineering, Washington, DC, USA, IEEE Computer Society (2007) 37–54
4. Muller, P.A., Fleurey, F., Jézéquel, J.M.: Weaving executability into object-oriented meta-languages. In L. Briand, S.K., ed.: Proceedings of MODELS/UML'2005. Volume 3713 of LNCS., Montego Bay, Jamaica, Springer (October 2005) 264–278
5. Aldewereld, H., Dignum, F., Penserini, L., Dignum, V.: Norm dynamics in adaptive organisations. In Boella, G., Pigozzi, G., Singh, M.P., Verhagen, H., eds.: NORMAS. (2008) 1–15
6. Brian Pickering, Sylvain Robert, S.M., Mengusoglu, E.: Model-driven management of complex systems. In: Proceedings of the 3rd International Workshop on Models@Runtime, at MoDELS'08, Toulouse, France (oct 2008)
7. Huang, A.C., Garlan, D., Schmerl, B.: Rainbow: Architecture-based self-adaptation with reusable infrastructure. In: ICAC '04: Proceedings of the First International Conference on Autonomic Computing, Washington, DC, USA, IEEE Computer Society (2004) 276–277
8. Bruneton, E., Coupaye, T., Leclercq, M., Quéma, V., Stefani, J.B.: An open component model and its support in java. In Crnkovic, I., Stafford, J.A., Schmidt, H.W., Wallnau, K.C., eds.: CBSE. Volume 3054 of Lecture Notes in Computer Science., Springer (2004) 7–22
9. Fleurey, F., Delhen, V., Bencomo, N., Morin, B., Jezequel, J.M.: Modeling and validating dynamic adaptation. In: Proceedings of the 3rd International Workshop on Models@Runtime, at MoDELS'08, Toulouse, France (oct 2008)