A Framework for Automatic Generation of Processes for Self-Adaptive Software Systems

Carlos Eduardo da Silva and Rogério de Lemos
School of Computing, University of Kent
Canterbury, Kent, CT2 7NF, UK
E-mail: {ces26, r.delemos}@kent.ac.uk

Keywords: self-adaptive systems, workflow generation, planning, process, architectural reconfiguration

Received: May 10, 2010

The self-adaptation of software systems is a complex process that depends on several factors that can change during the system operational lifetime. Hence, it is necessary to define mechanisms for providing a self-adaptive system the capability of generating during run-time the process that controls its adaptation. This paper presents a framework for the automatic generation of processes for self-adaptive software systems based on the use of workflows, model-based and artificial intelligence planning techniques. Our approach can be applied to different application domains, improves the scalability associated with the generation of adaptation plans, and enables the usage of different planning techniques. For evaluating the approach, we have developed a prototype for generating during run-time the workflows that coordinate the architectural reconfiguration of a web-based application.

Povzetek: Opisano je okolje za generiranje procesov za prilagodljive sisteme.

1 Introduction

It is commonly agreed that self-adaptive software systems should have the capability of modifying their own structure and/or behaviour at run-time due to changes in the system, its requirements, or the environment. In order to determine the actions to be taken to adapt itself, a self-adaptive software system observes and analyses itself and its environment, and if an adaptation is deemed to be necessary, a plan is generated for altering the system in a controlled manner. These systems are usually implemented in terms of a feedback control loop, and one of its key activities is the generation of adaptation plans [6].

The self-adaptation of a software system is a complex process that depends on several factors that may change during the system operational lifetime. In this context, an adaptation plan should respect the relationships and dependencies among the elements that compose the system, taking into account the actual state of the system and its environment. Thus, it is expected that self-adaptive software systems should be able to generate adaptation plans during run-time, in order to deal effectively with the variability and uncertainty involved in the adaptation. Otherwise, the self-adaptation process would cease to be consistent with the actual system that it controls, since the adaptation process would not be able to consider the actual system state, thus restricting what and how can be adapted. Some existing approaches for software self-adaptation (such as [7] [15]) explore mechanisms for the selection of adaptation plans, where each available plan and the conditions associated with the selection of that plan are defined at design-time through the use of adaptation policies. However, it is difficult to anticipate, at design-time, all the possible contexts of adaptation for some types of systems. For example, resources considered during the design of the adaptation policies may not be available during the system execution, or new adaptation possibilities that have not been considered at design-time may become feasible due the availability of new resources. To deal with this problem, it is necessary the definition of mechanisms for providing a self-adaptive software system the capability of generating, during run-time, the process that controls and coordinates software adaptation. In a previous work [8], we have outlined an approach based on the use of dynamic workflows for coordinating the self-adaptation of software systems. In this paper, we detail that approach by defining a framework that enables the dynamic generation of processes during run-time. This framework consists of a process and collection of languages, mechanisms and techniques, and its supporting computational infrastructure, consisting of open source and in-house tools,

Currently, there is a wide range of techniques for generating processes in different application domains, such as, web service composition [13], grid computing [11], software management [3], and architectural reconfiguration [4], [20]. These approaches present different solutions that are very specific to their respective domains. Moreover, they are difficult to be reused in other domains [11], and in some cases, in other applications inside the same domain [4]. Compared with existing similar approaches, our main contribution is a generic framework for the automatic generation of processes for self-adaptive software systems. The objective is to define a framework that can be applied to different application domains, and that can handle the scalability associated with the generation of adaptation plans. In order to achieve this, our framework is based on a combination of techniques, such as, workflows, artificial intelligence (AI) planning and model transformation. Workflows are used as a means to implement the processes, AI planning is used to dynamically generate the processes, while model transformation is employed in the translation between domain specific models into planning problems. A more specific contribution, which is an integral part of the framework, is the use of model-based technology for the generation of adaptation plans, thus enabling the usage of different planning techniques according to the needs of the application domain. In order to increase the scalability of the framework, the generation of processes is split into two levels of abstraction, namely, strategic and tactical, and to reduce the complexity of the overall interaction between these levels, the framework depends on the explicit representation of feedback loops [6], [18]. In order to evaluate the proposed approach, we have implemented a prototype where the proposed framework is being applied for generating adaptation plans that manage the architectural reconfiguration of a web-based self-adaptive application.

The rest of this paper is organised as follows. Section 2 presents some preliminary information that constitute the basis of the defined framework. Section 3 describes the proposed framework. Section 4 presents the application of the framework in the context of architectural reconfiguration of self-adaptive software systems, and evaluates the overall effectiveness of the proposed framework. Section 5 discuss some related work, while the conclusions and future directions of research are presented on Section 6.

2 Background

This section presents a brief overview of some of the key technologies on which the proposed framework for dynamic process generation is based.

We have adopted as a basis for generating processes the three-layer reference model for architecture-based self-managed systems adopted by Kramer and Magee [18]. In their model, the component control layer (the bottom layer) consists of the components that accomplish the system function, and includes facilities for supporting the manipulation of components (e.g., status query, creation and interconnection). The change management layer modifies the component architecture in response to changes that occur at the bottom layer, or in response to new goals from the layer above. The goal management layer (top layer) deals with high level goals of the system by changing management plans according to requests from the layer below and in response to the introduction of new goals.

The techniques for the generation of workflows can be divided into static and dynamic [13]. Static techniques for generating workflows are employed during design or compile-time, when workflows are not expected to change during run-time. On the other hand, dynamic techniques support the generation or modification of workflows at runtime in order to handle changes that may occur. Static techniques can receive as input an abstract model of the tasks that should be carried out, without identifying the resources to be used during execution, and workflow generation consists of selecting the actual resources for the abstract model. While dynamic techniques are able to automatically create the abstract model and select the appropriate resources for its execution. Our approach is classified as a dynamic generation technique where we apply AI planning for create the abstract model.

AI planning endeavours to find from a particular initial state a sequence of actions that are able to achieve an objective. In order to use AI planning, it is necessary to define a domain representation, which identifies among other things the available actions (or tasks) that can be used for generating a plan, and a problem representation, which includes the initial state and the desired goal. Currently, there is a wide variety of planners available that employ different algorithms and techniques, and support different heuristics. These planners allow tasks to be represented in different ways, including, pre- and post-conditions, temporal information (time for executing the task), and hierarchical task networks (tasks that can be decomposed in further tasks). For supporting this wide range of planning systems, there is a standard language called the Planning Domain Definition Language (PDDL), which is used to specify domain and problem representations [14].

Model-based technology explores the use of models at different levels of abstraction, and transformations between levels to manage complexity. These models conform to specific meta-models, which define the relationships among the concepts that can be used in a particular domain. Transformations allow the generation of different artefacts (e.g., source code) based on high level models, ensuring consistency between the models and the generated artefacts [19].

3 Framework for process generation

In our approach, processes are represented through workflows that are dynamically generated. Our framework divides the generation and execution of workflows in three phases: strategic, tactical and operational. At the *strategic phase*, AI planning is used to generate abstract workflows. An abstract workflow describes the set of tasks and the data dependencies among them, but without identifying the actual resources that will be used during the workflow execution. At the *tactical phase*, an abstract workflow is mapped into a concrete workflow which identifies the actual resources associated with the tasks. It is important to note that an abstract workflow can be mapped into different concrete workflows by using different combinations of resources. At the *operational phase*, the concrete workflow is executed. Figure 1 presents a simplified view of our approach for the dynamic generation of workflows.

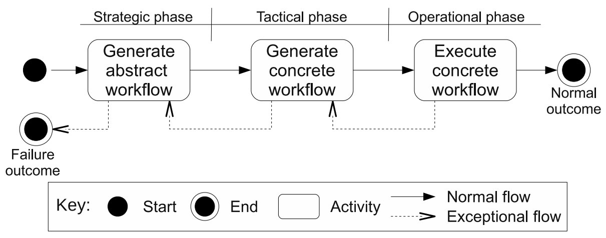
Strategic phase
Tactical phase
Operational phase
Generate abstract workflow
Generate concrete workflow
Execute concrete workflow
Normal outcome
Failure outcome
Key: Start End Activity Normal flow Exceptional flow

Figure 1: Overview of workflow generation.

The proposed approach can be placed in the context of the three layers reference model for self-managed system adopted by Kramer and Magee [18]. At the goals management layer, corresponding to our strategic phase, abstract workflows are generated according to a particular objective (or goal). These workflows are used as a basis for generating the concrete workflows at the change management layer, corresponding to our tactical phase. Once a concrete workflow has been generated, it is executed at the component control layer, corresponding to the operational phase. In our approach, similar to the three layers reference model, if errors occur at a particular phase and that phase is not able to handle the error, these are propagated to the previous phase in which they should be dealt with. In case of a problem occurs during the execution of the concrete workflow, a new concrete workflow is generated at the tactical phase, without the need to generate a new abstract workflow. If it is not possible to generate a concrete workflow (e.g., there are not enough resources), the generation goes back to the strategic phase, where a new abstract workflow is generated. In the eventuality it is not possible to generate an abstract workflow, the generation finishes with an error.

In the rest of this section, we provide more details concerning key aspects of the proposed framework. The first step, though, is to elaborate on the type of workflows supported by our approach, then we describe the task templates that are used for generating workflows, and finally, we present the actual process that enables the dynamic generation of workflows.

3.1 Types of workflows

In the context of the proposed framework for the dynamic generation of processes during run-time, it is important to distinguish how workflows are related to the system itself. Workflows can either be an integral part of the system or they can be peripheral to the system.

A workflow is an integral part of a system if its execution constitutes the system itself. This is the case of workflows that implement business processes, where each executed task contributes to the service to be delivered by the business (the goal of the workflow). In these workflows, the non-functional properties associated with the workflow may influence the non-functional properties of the system itself. For example, the execution time associated with a particular task should influence the time it takes for the workflow to execute. Moreover, the criteria for selecting which tasks should be part of a workflow and how they should be composed should be dictated by the requirements associated with the system. For example, in the case of web service orchestration, depending on the expected overall execution time of the workflow, the appropriate combination of tasks should be identified to compose the workflow.

A workflow is a peripheral part of a system if its execution has as an outcome the system that is expected to deliver the actual services. This is the case of workflows, for example, that coordinate architectural reconfiguration of software systems, where the goal of the workflow is to generate a system that can deliver the actual services. In these workflows, the non-functional properties associated with the workflow have almost no relation with the properties of the system being generated by the workflow. Thus, it is expected that the criteria for generating the workflow, and the criteria for generating the system from the execution of the workflow to be distinct. For example, in the architectural configuration of a software system, reliability could be a key property for that system, and it may not be considered when generating the workflow that would produce the actual system.

3.2 Defining task templates

In order to use AI planning, it is necessary to define task templates to be used by the planner in terms of their pre- and post-conditions. In our work we have used Planning Domain Definition Language (PDDL) to define task templates, and these templates are implemented as workflows. These implementations are structured in terms of atomic actions [10], as a means for incorporating fault tolerance, based on exception handling, into the workflow execution. Figure 2 presents the base structure defined for implementing task templates. Following the task templates specification, task template implementations include pre- and post-conditions checks, respectively, before and after the execution of the associated task.

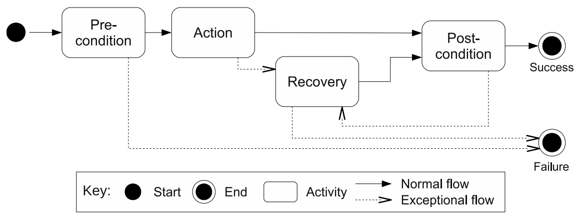

Figure 2: General structure for task templates implementation.

Task templates can have two possible outcomes: Success, representing the successful execution of the task, and Failure, representing failure in the execution of the task. A task template implementation may incorporate a recovery path, which is activated when there is a failure during execution of the task, or a violation of its post-condition. The recovery can be implemented in different ways (forward or backward) according to the application domain, and the task only finishes successfully when the post-condition is fully satisfied. In our approach, it is the system developer who decides how to implement the recovery. For now, we are restricting the implementation of our approach to backward error recovery, where the effects of a task are undone before finishing with a failure.

3.3 Workflow generation process

As previously mentioned, our framework is partitioned into three different phases, where the two first phases are associated with the generation of workflows, and the last phase is related to the workflow execution. Each phase of the generation process is composed by several activities[1], where some of these activities are dependent on the application domain in which the generation framework is being used, while others are completely independent of the application domain. In the following, we detail the activities associated with the strategic and tactical phases, while focusing on the domain independent activities.

3.3.1 Strategic phase

The main objective at this phase is to find the sequence of tasks that will compose an abstract workflow. This is achieved by means of AI planning techniques. In order to generate a workflow, an AI planner receives as input the goal to be achieved, the initial state, and a set of available task templates (with associated pre- and post-conditions).

Figure 3 presents an overview of how a workflow is generated at the strategy phase. In order to use AI planning, it is necessary, first, to obtain the initial state and the goals associated with the workflow, which are represented by the Obtain current state and Obtain workflow goals activity. These activities are dependent of the application domain in which the generation framework is being used.

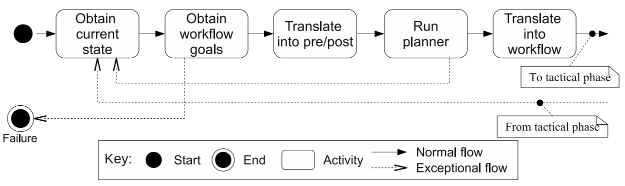

Figure 3: Overview of the activities of the strategic phase.

The Translate into pre/post represents the activity of translating the goal and the initial state from the notation (or representation format) used by the application domain to the notation employed by the planning technique. This translation receives as input one or more models and produces a planning problem representation in PDDL. These input models describe the goal and the initial state associated with the workflow in a domain dependent format. For example, goal-oriented models can be used to represent the objective associated with business processes [16], while architectural models can be used when considering software reconfiguration [20].

Once the pre- and post-conditions have been defined, the next step is the execution of the planner being used (Run planner activity). Depending on the planning technique used, there are several possibilities for generating workflows, and one workflow must be selected based on certain criteria. This selection can be incorporated in the planning technique through the use of different heuristics (such as the number of tasks in the generated workflow, or the time/-cost for executing the generated workflow), or can be a second step after the several possible workflows have been identified. In our approach, we have decided to represent it as a single activity, since it is affected by the planning technique used. Since most planners support PDDL, and receive as input pre- and post-conditions in this language, we can easily change the planner used with no, or very small impact in the task templates specification and the other activities of the strategy level. In case it is not possible to find a plan for a given set of pre- and post-conditions, the Run planner activity finishes with an error, and the strategic phase is restarted.

The Translate into workflow activity receives the output of the planner and translates it into an abstract workflow in the format used for specifying workflows in the execution platform, in our case, the YAWL workflow modelling language [23]. This translation involves the instantiation of each action identified in the PDDL plan as a YAWL workflow task, where the names identified in the action are used for populating the associated workflow task.

It is important to mention that the tasks that compose an abstract workflow are referred to as *strategic tasks*, and the resources associated with these tasks are referred to using logical names, which are sufficient to identify the actual resources at the next phase. In case of a problem at the tactical phase, or during the Run planner activity, the current state is updated (Obtain current state activity) and the Obtain workflow goals activity is activated to decide whether a new goal should be tried, or if the generation finishes with an error.

3.3.2 Tactical phase

The main concern at this phase is to allocate the appropriated resources to the tasks of the abstract workflow. The way in which the resources are identified is dependent on the application domain, and on the type of workflow being considered. When dealing with workflows that are an integral part of the system, the resources of the tactical phase are called *tactical tasks*, which are used for replacing the *strategic tasks* of the abstract workflow. In case of workflows that are a peripheral part of the system, we consider the *strategic tasks* as *parametrised tasks*, where logical names are used as the task parameters. In this case, the resources of the tactical phase are called *concrete pa-*

[1]Activities refer to the steps of the generation, while tasks are associated with the generated workflow.

rameters. Considering those two types of workflows, we have structured the activities of this phase in a way that allows the use of any of the cases (or both at the same time) according to the application domain. Figure 4 presents an overview of the activities at this phase.

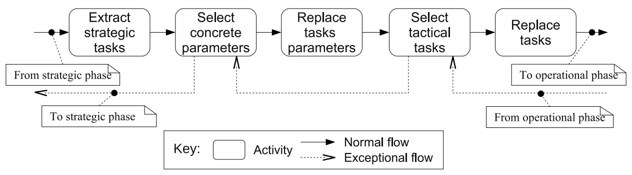

Figure 4: Activities of the tactics level.

The first activity at the tactical phase consists in extracting the strategic tasks that compose the received abstract workflow (Extract strategic tasks). The next activity (Select concrete parameters) is responsible for identifying the concrete parameters of a peripheral workflow. These concrete parameters are then used as replacement for the logical parameters in the extracted tasks (Replace tasks parameters). These tasks, with the concrete parameters identified, are then used in the next activity (Select tactical tasks) for selecting the tactical tasks of an integral workflow. Once the tactical tasks have been selected, they are used as replacement in the tasks of the abstract workflow (Replace tasks), resulting in a concrete workflow that can be executed at the next phase.

In case of failure during the execution of the generated workflow, new tactical tasks are selected for generating a new concrete workflow. If no tactical tasks are selected based on the concrete parameters, new concrete parameters are selected and the process repeated. If there are no available resources for the concrete parameters, the tactical phase finishes with a failure, and the process goes back to the strategical phase. It is important to note that it is necessary to control the relationship between the selection of concrete parameters, and the selection of tactical tasks. For example, in case there are no tactical tasks for a determined set of parameters, this set must be identified, and eliminated from the possible parameters in order to avoid an infinite loop. The same thing happens in the interaction between the execution of a concrete workflow and the selection of new tactical tasks, and between the strategic and tactical phase.

Apart from the changes in the task template implementation, the tactical phase is also customised according with the considered type of workflow. In this way, it is possible to deal with *peripheral workflows* by eliminating the Select tactical tasks activity, only with *integral workflows* by eliminating the Select concrete parameters and the Replace tasks parameters activities, or with both types by not eliminating any of the activities. This decision must be made during the instantiation of the framework into a particular domain. For example, different web service instances could be captured as tactical tasks of an integral workflow.

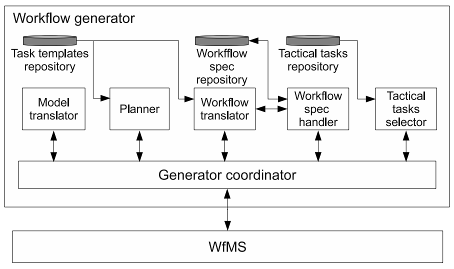

Figure 5: Overview of the infrastructure for supporting process generation.

3.4 Infrastructure

In the sequence, we describe the tools and techniques of the infrastructure that provides the basis for dynamically generating workflows. We present the infrastructure's main components, and identifying those components that must be modified according with the application domain.

Figure 5 presents the infrastructure that underpins the proposed framework. The generation process is managed through a workflow. Workflow specifications are modelled using the YAWL workflow modelling language [23], and are executed in the YAWL Workflow Management System (WfMS). All elements have been implemented using the Java programming language, and the communication among them uses web service technology. The Model translator component must be customised based on the application domain by providing the transformation rules that translates domain specific models into planning problems. We have successfully used different planners, implemented in C and Python (Satplan, Sgplan6, LPG-td, LAMA)[2] by building Java wrappers for each one of them. When dealing with planning based only on pre- and post-conditions, the replacement of the planner used has no impact in the other components of the framework. We also used temporal based planning, requiring the modification of the task template specification for including the appropriated non-functional property (e.g., the cost of each action), and the transformation rules of the Model translator component for including the metric used by the planner (e.g., minimize the total execution time of the generated plan) in the problem representation. The Workflow translator component is responsible for translating a PDDL plan into an YAWL workflow specification. It does that by parsing a PDDL plan and recovering the task implementation associated with each task of the plan. These task implementations are them composed together as the tasks of a new workflow specification using the services of the Workflow spec handler component. The Workflow spec handler provides different services for manipulating workflow specifications, and has been implemented based on the API of the YAWL WfMS. All repositories have been implemented as Java classes. The Workflow spec repository interacts with the YAWL WfMS and its worklets service [1] for load-

[2]Participants of different editions of the International Planning Competition: http://ipc.icaps-conference.org/

ing concrete workflows into the WfMS engine. The Tactical tasks selector is responsible for implementing the decision making associated with the selection of tasks in the tactical phase, while the Tactical tasks repository is used for storing all available tactical tasks.

4 Process generation for architectural reconfiguration

This section describes how the proposed framework is being used for generating the workflows that manage the architectural reconfiguration of a self-adaptive application, presenting the case study application used for its evaluation, its prototype implementation and a brief discussion about some experiments results.

4.1 The reconfiguration process

In the context of the Collect/Analyse/Decide/Act (CADA) feedback control loop [12], our reconfiguration process is related to the decide and act phases, while we assume the existence of mechanisms responsible for the collect and analyse phases. The decide phase of the feedback loop is responsible for identifying what to adapt (the selection of a configuration), and how to adapt (the generation of a workflow for connecting this configuration), while the act phase is responsible for executing the generated workflow. In this scenario, it is important to mention that we are not concerned with the selection of an architectural configuration for the system. This is outside the scope of our work, and there are several existing approaches that can be used to implement this (e.g., [2]). Instead, our focus is in demonstrating how adaptation plans can be generated based on the selected configuration.

In our approach, we divided a configuration model in two different levels of concerns, reflecting the division between abstract and concrete workflows. In this way, an *abstract configuration* describes a system configuration in terms of its components, identified by a logical names, their types and connections, identifying the structure of the system, but abstracting away from the actual component instances. A *concrete configuration*, on the other hand, describes a system configuration in terms of actual component instances, and their respective attributes. Similar to an abstract workflow, an abstract configuration can be instantiated into different concrete configurations depending on the availability of actual component instances.

Figure 6 presents an overview of the reconfiguration process. At the strategic phase, the initial state and goal correspond, respectively, to the current and selected configurations. The selected configuration is an abstract configuration, where the affected elements are treated as abstract elements. For example, when dealing with the replacement of a component, the new component can be treated as an abstract component, allowing the reuse of the abstract configuration by replacing this abstract component. If the process is being applied to establish a new configuration, all elements are treated as abstract elements. The Translate into Pre/post activity generates the planner inputs from architectural models using a translation algorithm based on model comparison techniques, where the architectural models are compared, and the results of this comparison are translated into a problem specification in PDDL using a set of model transformation rules. This algorithm follows the idea of model transformation employed in model-driven engineering, where a model (the comparison results) is transformed into another model (pre- and post-conditions expressed in PDDL) [19]. In case the planner can not find a plan for a given set of pre- and post-conditions, a new configuration is selected, or the reconfiguration finishes with a failure.

At the tactical phase, the Obtain concrete configuration is responsible for finding a concrete configuration for the system, which is used for generating a concrete workflow. This activity corresponds to the Select concrete parameters activity presented in Section 3.3.2. It is important to mention that in the present context, the generated workflows are a peripheral part of the system, where abstract workflows are characterised by the use of logical names as tasks parameters. In this way, we do not consider the Selection of tactical tasks activity at the tactics phase.

The generated workflow is then executed changing the target system. In case of a failure during its execution (e.g., a failure while connecting two components), a new concrete configuration is selected, and a new concrete workflow is generated and executed. For now, we are assuming the existence of exception handling mechanisms capable of undoing the effects of the failed workflow before returning to the tactical phase. In case there are not enough resources for establishing a new concrete configuration based on the selected abstract configuration, a new abstract configuration is selected, and a new abstract workflow is generated. If it is not possible to find a new configuration for the system, the process finishes with an error.

4.2 Case study

In the sequence, we present an example scenario of a distributed system that was used to evaluate our work.

The developed prototype application provides stock quotes portfolio reports with suggestions of investments based on historical and current information about the client, and the actual stock quotes values. In this scenario, we consider the existence of different resources that are captured by different component types, which can be combined in different configurations for the provision of the mentioned service. For each of these component types, there are several component instances that can be used. New instances and resource types can become available, resulting in configurations not envisioned at design-time.

Figure 7 presents an example of a configuration for the provision of this service. The Front end component represents the user access point to the service. Report ser-

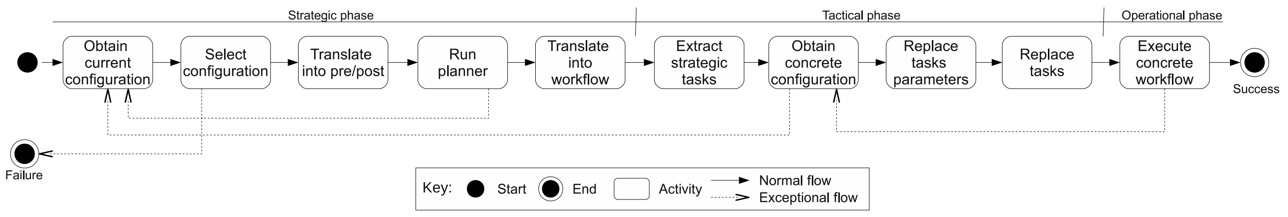

Figure 6: Overview of the process for generating reconfiguration plans.

vice logic represents the application logic of the offered service (stock quote portfolio report), and requires services from internal and external providers. External providers are used for obtaining stock quote values, which can be obtained from different sources. These are captured through Bridge components. A bridge component handles architectural mismatches between the service providers and the system, providing an uniform interface for the different online providers. The Client info logic component, an internal service, provides information about the client, and requires a Database component.

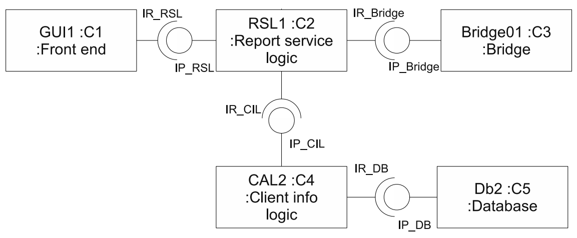

Figure 7: Example of a configuration.

Follow the division between abstract and concrete configurations, each component instance of a concrete configuration is associated to a logical name and type. In this way, GUI1 :C1 :Front End indicates that the component instance GUI1 is associated with logical name C1 with functional requirements associated with components of type Front End.

4.3 Prototype implementation

The infrastructure for supporting the defined reconfiguration process has been implemented based on the architecture presented in Figure 5, and is presented in Figure 8.

In this prototype, architectural models are represented using Eclipse Modelling Framework (EMF) models based on xADL 2.0 [9]. In order to run our experiments, we have implemented a simplified execution platform in which components must be blocked (a kind of quiescent state [17]) before being involved in a reconfiguration, blocked components are not considered when selecting a new configuration, and all architectural elements provide two different types of interfaces, application and configuration services interfaces, as explained in [10]. The Model translator component has been implemented based on EMF

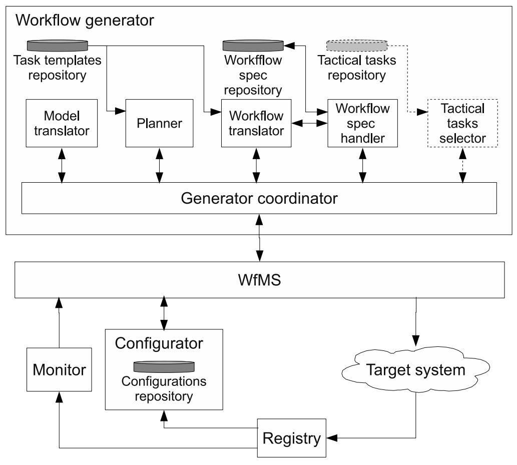

Figure 8: Overview of the overall system architecture.

and Atlas Transformation Language[3]. This component receives architectural models as input, and applies a set of ATL transformation rules for generating a problem description in PDDL. Figure 9 presents an example of a PDDL problem description generated by the Model translator component. This example considers the establishment of the configuration presented in Figure 7. Thus, the inputs used are an empty xADL description, corresponding to the current configuration, and a xADL description of the configuration of Figure 7, corresponding to the selected configuration. The header has an identifier for the problem (reconfigurationProblem) and a reference to the domain representation where the actions and predicates are described. The list of objects involved includes all components (identified by logical names) and their provided and required interfaces. In this example, all involved components are unblocked, and all connections are not established.

Since we consider the generation of peripheral workflows, the Tactical tasks selector component is deactivated by removing the respective activity from the workflow that controls the reconfiguration. The generated workflow changes the configuration of the Target system. The Registry is where all available components, their respective attributes, and a current model of the target system are stored. We assume that the Registry is responsible for monitoring the available resources, and for providing an accurate view of the Target system. Any changes in

[3]http://www.eclipse.org/atl

```
(define (problem reconfProblem)
(:domain reconfigurationDomain)
(:objects C1 C2 C3 C4 C5 - component
  IR_RSL IR_Bridge IR_CIL IR_DB - requiredInterface
  IP_RSL IP_Bridge IP_CIL IP_DB - providedinterface
)
(:init
  (not (blocked C1)) (not (blocked C2)) (not (blocked C3))
  (not (blocked C4)) (not (blocked C5))
  (not (connected C1 IR_RSL C2 IP_RSL))
  (not (connected C2 IR_Bridge C3 IP_Bridge))
  (not (connected C2 IR_CIL C4 IP_CIL))
  (not (connected C4 IR_DB C5 IP_DB))
)
(:goal (and
  (not (blocked C1)) (not (blocked C2)) (not (blocked C3))
  (not (blocked C4)) (not (blocked C5))
  (connected C1 IR_RSL C2 IP_RSL)
  (connected C2 IR_Bridge C3 IP_Bridge)
  (connected C2 IR_CIL C4 IP_CIL)
  (connected C4 IR_DB C5 IP_DB)
  )
))
```

Figure 9: Example of pre- and post-conditions in PDDL.

the resources' availability should be reflected at the Registry. The Monitor component represents the mechanisms used for the collect and analysis phases of the feedback control loop. This component observes the components of the established configuration, checking the values of the component attributes (e.g., response time) against a defined threshold. In case of violation, it starts a reconfiguration by starting the execution of the reconfiguration workflow. The Configurator is responsible for selecting a configuration for the system. It has been implemented based on the use of a utility function that is a linear combination of the utilities of the different elements. The Configurations repository stores the configuration models used during the reconfiguration.

4.4 Discussion

In order to demonstrate and evaluate the proposed approach, we have conducted some experiments involving the architectural reconfiguration of a case study application.

Among the experiments, we have considered the establishment of a particular configuration, which for the planner, is the toughest scenario, since it involves all components of the configuration. In this experiment, we have observed the search space (number of possible actions) of the LPG-td planner, changing the number of resources available and the size of the generated workflow.

The size of the generated workflow depends on the number of components and connections in the selected configuration. Every component of a configuration must be blocked before being connected, and unblocked at the end of the reconfiguration. In this way, a configuration with *n* components and *n-1* connections will have *3n-1* tasks. For example, the configuration of Figure 7, with five components and four connections, requires a workflow with 14 tasks (blocking the five components, establishing the four connections, and unblocking the five components). In these experiments we have considered three different abstract configurations, which require three, four and five components, and contains respectively two, three and four connections.

We have implemented two variations of our approach.

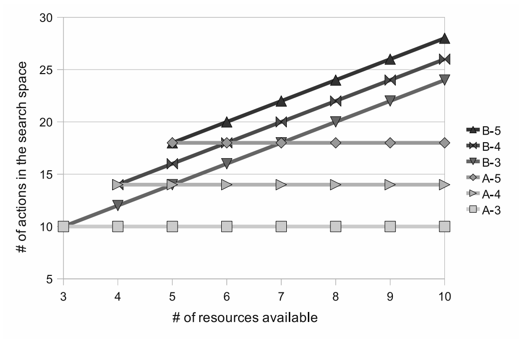

Figure 10: Search space variation changing the resources available.

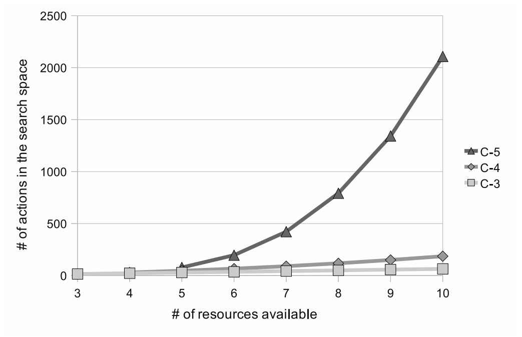

Figure 11: Search space variation considering the selection of a configuration by the planner.

The main difference between these variations is the amount of information passed to the planner. Our approach (identified by A in the graphs) explores the division between strategy and tactics using logical names for representing the components of a configuration. The second approach (identified by B) separates the selection of a configuration from the generation of the plan, but includes all available resources in the planner. While the third approach (identified by C), based on [4], uses the planner for selecting a configuration for the system, including all possible configurations and all available resources in the planner.

Figure 10 shows the variation in the search space as we vary the number of resources available. In this graph, A-3 means the numbers obtained by our approach (A) for a workflow related with a configuration that requires three different components. Since we use logical names to represent the resources required by a configuration, and do not consider all available resources in the planner, the number of resources available does not affect the search space. We notice a linear progression of the search space when we include all available resources in the planner (B-3, B-4 and B-5). The starting points of each curve represents the minimum number of resources required by the correspondent configuration.

Figure 11 presents the change in the search space when the selection of a configuration is combined with the generation of the plan. Based on this experiment, it is clear the overhead caused by mixing selection and plan generation.

As expected, the search space increases with number of resources known by the planner, and the selection of a configuration by the planner further aggravate its scalability. The division between strategy and tactics helps to reduce the search space since the planner does not need to know about all available resources, but only about those involved in the reconfiguration, which are represented by logical names.

5 Related work

The focus of this section is to review how existing approaches generate adaptation plans for self-adaptive software, and we start with those approaches that specify adaptation plans at design-time. Cheng et al. [7] capture adaptation plans as a set repair strategies, consisting of condition-action rules, where conditions are evaluated based on utility functions. Georgas and Taylor [15] propose the use of adaptation policies, also captured as condition-action rules, that indicate what actions should be taken in response to events. Both approaches are focused on the definition of the mechanisms for selecting one adaptation plan from the set of available plans, where each available plan, identifying what and how to adapt, is defined at design-time. A major limitation with these approaches is the difficulty in anticipating all possible contexts in which system adaptation may take place. This has a major impact on identifying the appropriate adaptation plans because of the combinatorial nature between conditions and actions, which also affects the management of the available plans. Different from these approaches, our approach provides the means for defining adaptation plans during run-time.

There are other approaches (such as [5]) that provide support for selecting what to adapt at run-time. However, there is a cost to be paid in these approaches, since the adaptation is enacted by deactivating and reactivating the whole software system, even for the case when a single element needs to be replaced. Moreover, the main focus of these approaches is on the selection of what to adapt based on the state of the environment and the resources required for enabling it, and not on how to enact the selected adaptation. Differently from these approaches, our focus is on the dynamic generation of the process that enacts the adaptation in a way that does not require the complete deactivation of the system.

Some approaches use model comparison techniques, in the context of architectural reconfiguration, for generating adaptation scripts that impact only those elements affected by the adaptation. Alia et al [2] compare the models of the current and the selected system configuration for identifying the actions of the adaptation script. Morin et al. [20] build on top of model comparison by applying priority rules for ordering the identified adaptation actions. However, model comparison and priority rules are not enough to generate plans with complex relationships among the adaptation actions and the involved system components. For it, it is necessary to consider pre- and post-conditions associated with the actions, as demonstrated in [22]. Moreover, the focus of these approaches is not the generation of adaptation scripts, but the selection of the system configuration [2], and coping with the exponential growth in the number of possible configurations [20]. Thus, they do not consider the possible issues associated with the generation of the adaptation scripts. Our approach also applies model comparison for generating adaptation plans, but the results of the comparison are used for identifying the inputs for an AI planner, supporting the generation of plans that can deal with complex relationships between its constituent tasks.

Other approaches have applied AI planning for selecting a configuration and deciding how to change the system at the same time [3], [4]. Both these approaches require the inclusion of all available resources in the planner, which, together with the mix of configuration selection and plan generation, affects the scalability of the planer. Moreover, both approaches require the specification of the current system state and the target system state using PDDL. In our approach, we are not restricted to a fixed set of resources for generating adaptation plans, and we employ model-based technology for translating domain specific models into planning problems. The partition of our approach into different levels of abstraction provides support for dealing with variations in the resources availability and reduces the search space considered by the planner, increasing its scalability, while more specific and scalable techniques can be used for selecting an adaptation for the system (such as [2]).

Concerning workflow generation, several approaches apply AI planning techniques in different domains [21], such as, grid computing [11] and web service composition [13]; however, these approaches are very specific to their respective domains. Our approach is based on ideas from [11] [13], in which the generation is partitioned into strategical and tactical for increasing scalability. Differently from these approaches, we are focused on providing an framework that can be applied to different domains, and consider two types of workflows, integral and peripheral, supporting different decision making and planning mechanisms according with the application domain.

6 Conclusions and future work

This paper has presented a framework for the automatic generation of processes for self-adaptive software systems based on workflows, AI planning and model transformation. The framework can be applied to different application domains by supporting the use of the most suitable generation techniques according to the application domain. Moreover, our approach reduces the search space considered by a planner by splitting the generation in two levels of abstraction, and provides support for generating different types of workflows. In order to evaluate the proposed framework and its respective computational infrastructure, a prototype was developed for experimenting our approach, and comparing it with similar approaches. The effectiveness of the whole approach was evaluated in the context of a web-based self-adaptive application for obtaining stock quotes reports, where the processes generated at run-time are responsible for coordinating the architectural reconfiguration of the system.

Although the proposed approach for the dynamic generation of process for self-adaptive software systems has produced quite promising results, we have identified a couple of limitations that if properly handled could enhance overall effectiveness of the approach. First, the type of workflows being generated are simple sequential workflows, however, the intent is to incorporate non-determinism and other planning techniques to capture uncertainty into the generation of workflows, and to support different control flow constructs, such as, conditional and parallel execution branches. Second, although the task templates for our framework are structured using atomic actions, one issue that was not yet fully investigated is how to exploit this for the provision of fault tolerance. The intent is to incorporate exception handling mechanisms for tolerating faults that might occur during the execution of generated workflows.

As future work, we would like to simplify the reuse of the framework, and this could be achieved by using meta-transformation languages for translating domain specific models into pre- and post-conditions. Also in this direction, the intent is to incorporate ideas from software product lines into our framework for dealing with the variability of processes. Another future work would be the application of the proposed framework into other application areas to evaluate its overall effectiveness since our initial idea was to see this framework applied to support software self-adaptation whenever processes need to generated during run-time. Concerning the domain of reconfiguration, we intent to consider more complex scenarios in which, for example, there exists transfer of state between components).

Acknowledgement

Carlos Eduardo da Silva is supported by the Programme Alβan, the European Union Programme of High Level Scholarships for Latin America, scholarship No. E07D401107BR.

References

[1] M. Adams et al. Worklets: A service-oriented implementation of dynamic flexibility in workflows. In *Proc. of the OTM CoopIS'06*, pages 291–308, 2006.

[2] M. Alia et al. A component-based planning framework for adaptive systems. In *Proc. of the OTM DOA'06*, pages 1686–1704, 2006.

[3] A. Andrzejak et al. Feedbackflow-an adaptive workflow generator for systems management. In *Proc. of the ICAC'05*, pages 335–336, 2005.

[4] N. Arshad et al. Deployment and dynamic reconfiguration planning for distributed software systems. *Software Quality J.*, 15(3):265–281, 2007.

[5] M. Autili et al. Towards self-evolving context-aware services. In *Proc. of the DisCoTec CAMPUS'08*, 2008.

[6] Y. Brun et al. Engineering self-adaptive systems through feedback loops. In *Software Engineering for Self-Adaptive Systems*, pages 48–70. 2009.

[7] S.-W. Cheng et al. Architecture-based self-adaptation in the presence of multiple objectives. In *Proc. of the ICSE SEAMS'06*, pages 2–8, 2006.

[8] C. E. da Silva and R. de Lemos. Using dynamic workflows for coordinating self-adaptation of software systems. In *Proc. of the ICSE SEAMS 2009*, pages 86–95, 2009.

[9] E. M. Dashofy et al. A comprehensive approach for the development of modular software architecture description languages. *ACM Trans. Softw. Eng. Methodol.*, 14(2):199–245, 2005.

[10] R. de Lemos. Architectural reconfiguration using coordinated atomic actions. In *Proc. of the ICSE SEAMS'06*, pages 44–50, 2006.

[11] E. Deelman et al. Mapping abstract complex workflows onto grid environments. *Journal of Grid Computing*, 1(1):25–39, 2003.

[12] S. Dobson et al. A survey of autonomic communications. *ACM Trans. Auton. Adapt. Syst.*, 1(2):223 – 259, 2006.

[13] S. Dustdar and W. Schreiner. A survey on web services composition. *International Journal of Web and Grid Services*, 1(1):1–30, 2005.

[14] M. Fox and D. Long. Pddl2.1: An extension to pddl for expressing temporal planning domains. *J. of AI Research*, 20:61–124, 2003.

[15] J. C. Georgas and R. N. Taylor. Towards a knowledge-based approach to architectural adaptation management. In *Proc. of the WOSS'04*, pages 59–63, 2004.

[16] D. Greenwood and G. Rimassa. Autonomic goal-oriented business process management. In *Proc. the ICAS'07*, page 43, 2007.

[17] J. Kramer and J. Magee. The evolving philosophers problem: Dynamic change management. *IEEE Trans. Softw. Eng.*, 16(11):1293–1306, 1990.

[18] J. Kramer and J. Magee. Self-managed systems: an architectural challenge. In *Proc. of the FOSE'07*, pages 259–268, 2007.

[19] J. Miller and J. Mukerji. MDA guide version 1.0.1. Technical report, OMG, 2003.

[20] B. Morin et al. An aspect-oriented and model-driven approach for managing dynamic variability. In *Proc. of the MoDELS'08*, pages 782–796, 2008.

[21] D. Nau. Current trends in automated planning. *AI Magazine*, 28(4):43–58, 2007.

[22] C. Shankar and R. Campbell. Ordering management actions in pervasive systems using specification-enhanced policies. In *Proc. of the PERCOM'06*, pages 234–238, 2006.

[23] W. M. P. van der Aalst and A. H. M. ter Hofstede. Yawl: Yet another workflow language. *Inf. Syst.*, 30(4):245–275, 2005.