Dynamic Management of Integration Testing for Self-Adaptive Systems

Carlos Eduardo da Silva  
Escola de Ciências e Tecnologia  
UFRN, Natal, RN, Brazil  
carlos.silva@ect.ufrn.br

Rogério de Lemos  
School of Computing  
University of Kent, UK  
r.delemos@kent.ac.uk

Abstract

Self-adaptive software systems that rely on dynamic reconfiguration of their architectures invariably assume that the configurations have previously been tested. However, this is an unreasonable assumption to be made considering that one might not be able to account at development-time for all the architectural elements that might be available at run-time. Hence the challenge for generating plans for managing testing activities during run-time. In this paper, we present an approach for dynamically modifying the process responsible for the generation of the processes that manage the testing of a software system. The feasibility of the proposed approach has been evaluated by applying it to the generation of plans that manage at run-time the integration testing of self-adaptive software systems. In particular, we demonstrate how plans that manage the integration testing can be dynamically generated according to the system testing goals (e.g., coverage level).

1. Introduction

Component-based integration testing is concerned with tests involving interfaces between components in order to determine whether these components are able to interoperate [14]. This is a complex process which comprises a variety of activities, techniques and actors, and is further aggravated by the inherent complexity of today’s software systems [2]. Integration testing relies on an incremental approach in which components are gradually added into an architectural configuration and tested. Once a configuration passes the test, a new component is integrated with the configuration that needs to be tested again. These steps are repeated until all software components are integrated and tested, following an integration order based on the component dependencies in a way that minimises the number of stubs required.

With the recent advances on software technology, there is an emergent need for performing integration testing in an automated manner, and during run-time [3] in the context of self-adaptive systems. A self-adaptive system should be able to autonomously modify itself during run-time in order to deal with changes [5]. In this context, the basis of this paper has been our work on the dynamic generation of plans for coordinating integration testing of component-based software systems [7]. In addition to the need for generating processes at run-time, there is also the need for adapting how these processes are actually generated. This is essentially the contribution of this paper compared with our previous work in which there was no means to dynamically modify the mechanisms responsible for generating the processes that manage the integration testing of self-adaptive software systems. Since the criteria associated with integration testing may change at run-time, one would expect this to have an effect on the dynamically generated processes.

In summary, the main contribution of this paper is the definition and development of a framework for the meta-generation of processes, providing support for the dynamic adaptation of the processes that are responsible for generating the processes that manage the integration testing of component-based software systems. In this paper’s proposal, we are essentially dealing with processes at two levels: one at the lower level that is responsible for generating the processes that manage integration testing, i.e., the generation level, and the other at a higher level that is responsible for managing how integration testing processes are actually generated, i.e., the meta-generation level. The goal of our work is to focus on this higher level, which can be accomplished either through parametric or structural adaptation [1]. We focus on parametric adaptation, where the process at the generation level is adapted through the modification of the parameters associated with its activities, while the structure of the process remains the same. The motivation for introducing yet another level of control is to enforce separation of concerns between the decision making associated with the generation of processes and how this generation is performed.

Essentially, our approach for the meta-generation of processes extends our previous work on a framework for the
automatic generation of plans for managing the run-time integration testing [7] by incorporating means that enable to change the way plans are generated. The feasibility of the revised framework is demonstrated in terms of a prototype that is able to generate different plans according to the coverage level established at the system goals.

The rest of the paper is organised as follows. Section 2 presents some background information on our framework for generating processes and its application to component-based integration testing, and some related work. Section 3 introduces our approach for the “meta-generation of processes”, describing how it extends our previous works. Section 4 presents a case study application and describes the experiments that have been conducted to evaluate our approach, concluding with a discussion on the results achieved. Section 5 concludes the paper, identifying some limitations of the proposed approach and presenting some future research directions.

2. Background and Related Work

2.1. Dynamic Generation of Plans for Integration Testing

Our approach for generation of the integration testing process [8] partitions the generation and execution of workflows\(^1\) into three phases: strategic, tactical and operational. At the strategic phase, Artificial Intelligence (AI) planning is used to generate an abstract workflow based on an abstract architectural configuration, the integration order calculated for this configuration and any necessary stubs and a set of task templates. An abstract workflow describes a set of tasks for integrating and testing the system, and the data dependencies among them, without identifying the actual components, nor the test cases, that will be used during the workflow execution. At the tactical phase, the abstract workflow is mapped into a concrete workflow based on a concrete configuration and the test cases that are going to be used with each concrete component. It is important to note that at the strategic phase, the resources associated with the tasks are referred to by a logical name, which should be sufficient to identify the actual resources at the tactical phase. In this way, an abstract workflow can be mapped into different concrete workflows by using different combinations of resources. Finally, at the operational phase, the concrete workflow is executed. Figure 1 presents a simplified view of our approach for the dynamic generation of workflows.

Our approach for workflow generation is based on the explicit representation of feedback loops between the three phases, and deals with failures in the following way: in case a fault occurs at a particular phase and that phase is not able to handle it, the processing resumes at the previous phase. The process finishes with an exceptional outcome in case it is not possible to generate another abstract workflow.

2.2. Related Work

Compared to existing approaches for component-based integration testing, our work goes in the direction of automating testing execution during run-time (one of the challenges of the software testing research community [2]), defining means for generation of plans that are responsible for managing the tests and evaluating different candidate components.

There are a number of approaches for dealing with the automation of testing and the testing of software components by the component user [14]. However, these approaches have been targeted to development-time. One interesting approach is the work of Yoon et al. [16], which has focused on testing whether a system can be built across all of its possible configurations when considering different versions of the involved components. Similar to ours, their approach attempts to fully automate the testing process. However, their approach is not suitable to be applied during run-time, different from our proposal, whose generated process obtains a single configuration that passes the integration test during run-time.

Regarding software testing during run-time, it is possible to observe a movement towards online testing, also called testing at service time (opposed to testing at deployment time, where the software system is tested in its deployment environment, but before being accessible to its users) [2, 4]. Brenner et al. [4] have presented an infrastructure for automating run-time execution of tests on software components based on Built-In Test (BIT), monitoring and controlling the system environment and test resources in order to minimize the disruption of testing to the software system. However, their approach is only concerned with the actual execution of tests. Metzger et al. [12] have proposed the use of online testing as the means of augmenting the monitoring process of self-adaptive software systems for increasing the confidence level of predicted failures used to trigger pro-active software adaptation. These approaches not only

\(^1\)In our work, processes are implemented as workflows.
can be used to complement each other, but also provide evidence of the potential of our proposed approach, which can, for example, be employed to provide the means for dynamically adapting the testing process due to the adaptation of the software system (as mentioned by Metzger et al. [12]).

3. Meta-Generation of Processes

This section describes how the generation of workflows for component-based integration testing can be controlled, which extends our previous work on the dynamic generation of processes for self-adaptive systems [6, 8, 7].

3.1. General Model

Our work takes as basis the three-layer reference model for self-managed systems [11], whereas based on high-level goals established by the Goal management layer, our approach at the Change management layer can tune the generation of a workflow that integrates and tests the software system at the Component control layer. A general view of our approach is presented in Figure 2.

Our approach is itself composed by three levels: execution, generation and meta-generation. Following the ideas of computational reflection, each level consists of several models, including those that capture relevant information from the level(s) below. In this way, a concern model captures information related to the level’s objectives, a subsystem model provides a representation of the level below (i.e., the managed subsystem), while an environment model represents the context of the system [15].

The Execution level (bottom level) is responsible for executing the generated workflow. The key concern of the Execution level is related to whether or not the generated workflow can be successfully executed, in other words, if the selected architectural configuration passes all selected tests. The concern model of this level, i.e., (Execution-level concern model), is used for managing the workflow execution, such as, information about the test cases that are part of the workflow.

The Generation level (middle level) reflects upon the Execution level, and is responsible for generating the workflow. The key concern of the Generation level is whether it is possible to build a workflow with the parameters established by the level above (the selected system configuration and its corresponding test suite). It contains three types of models: the subsystem model that captures information about the level below (e.g., information about the generated workflow and its execution), the concern model that captures information about the running of the different steps involved in the generation of workflows, and the environment models that capture those aspects of the application domain that are relevant to the generation level, such as, the concrete configuration to be tested and the test cases that will be part of the generated workflow. This level encompasses the strategic and tactical phases of the generation process depicted in Figure 1.

The Meta-generation level (top level) reflects upon the Generation level, and is responsible for managing workflow generation according to the system goals, which establish the criteria for generating workflows. The goals received at the top level encompass the different attributes that might affect the generation/selection of test harness (e.g., test cases), such as, the desired coverage level and adequacy criteria. The key concern of this level is whether it is possible to achieve the desired coverage level based on the test cases associated with the components of the selected architectural configuration. Once the test suite for achieving the desired coverage level is selected, it is used for configuring the generation process of the level below. At the Meta-generation level the concern model captures information about the goals of the system.

In order to present a general view of the concerns associated with the three levels, Table 1 summarises some of the elements that are monitored/controlled at each level of our approach. The Meta-generation level gets as input the test cases available for the system configuration, and the respective coverage level associated with a particular test suite. Regarding control, this level is responsible for deciding whether it is possible to achieve with the available resources the desired coverage level. The Generation level monitors the available resources for the selected configuration and the selected test cases during the building of the workflow. In this way, the generation of a workflow can be interrupted in case a particular component or test case is no longer available. The Execution level is responsible for
monitoring the outcomes of the tests to decide whether or not the workflow has been successfully executed, and controlling the components of the configuration and the execution of the test cases.

<table>
<thead>
<tr>
<th>Level</th>
<th>what is monitored</th>
<th>what is controlled</th>
</tr>
</thead>
<tbody>
<tr>
<td>Meta-generation</td>
<td>Available test cases</td>
<td>Selection of test cases</td>
</tr>
<tr>
<td>Generation</td>
<td>Achieved coverage level</td>
<td>Concrete configuration</td>
</tr>
<tr>
<td>Execution</td>
<td>Test case outcome</td>
<td>Test case execution</td>
</tr>
</tbody>
</table>

Table 1. What is being observed/controlled at each level.

As previously mentioned, compared with our previous work on the framework, which implemented the Execution and Generation levels, the work described in this paper extends that framework by including a Meta-generation level for managing the workflow generation process. In the sequence, we describe the Generation level, identifying what needs to be adapted in the generation of workflows, and the problems that must be tackled. This serves as basis for identifying the needs for the Meta-generation level, which is presented afterwards.

3.2. Generation Level

This level is responsible for generating the workflows that conduct the integration testing of the managed system. The activities of this level, presented in Figure 3, correspond to an instantiation of the generation process provided by our framework, and have been presented in [7].

![Figure 3. Overview of the generation level](image)

The first two activities, Obtain current configuration and Obtain abstract configuration, obtain the architectural configurations of the system, respectively, the current configuration and the abstract configuration that should be tested. If no abstract configuration is available, the process finishes with an exceptional outcome.

Calculate integration order represents the identification of the integration order for the selected abstract configuration. Translate into pre/post translates the configuration models into a model with the pre- and post-conditions that will be used by the next activity (Run planner), which runs an AI planner for producing a plan model. In case it is not possible to find a plan for the given inputs (e.g., lack of stubs) the process finishes with an exceptional outcome. The Translate into workflow activity translates the generated plan model into an abstract workflow, including several sub-workflows that are dynamically generated for changing the system configuration at the different integration steps.

Extract abstract tasks extracts the abstract tasks that compose the abstract workflow, while Obtain concrete configuration is responsible for obtaining the concrete configuration of the system that will be used by the next activity (Replace tasks parameters) to replace the logical parameters of the abstract tasks. The Obtain test cases activity is responsible for obtaining the test cases that will be used for testing the components of the concrete configuration. Each component might need to be tested with a different number of test cases, defined by the Meta-generation level, and this activity is responsible for obtaining the selected test cases. These test cases are used for creating sub-workflows that will be used to compose the generated workflow by replacing its abstract tasks, and are the ones that control the execution of the test cases selected for its corresponding component. For example, if the Meta-generation level selects five test cases for a particular component, its corresponding sub-workflow will be composed of five tasks, one for each test case.

Finally, Associate concrete tasks replaces the abstract tasks of the abstract workflow with the concrete tasks created by the previous activity, resulting in a concrete workflow that is passed to the Execution level. For these concrete tasks, the concrete components under test are identified, together with the test cases that have been selected based on the received goals. In case there is a problem while generating the concrete workflow (i.e., either obtaining the concrete configuration or the test cases that will be used), the process finishes with an exceptional outcome, which is dealt with by the Meta-generation level.

3.3. Meta-generation Level

This level is responsible for adapting the Generation level according to the testing goals (e.g., desired coverage level). However, before the generation process can be dynamically modified, it is necessary to identify what can be changed and how.

Our framework for workflow generation has been designed with the purpose of being adjustable to different application domains. This can be done by either modifying the structure of the generation process (including/removing activities), or by modifying parameters associated with activities of the process. In this paper, we focus on paramet-
ric adaptation, whilst introducing support for dynamically modifying the generation process.

Taking as basis the activities of the Generation level, we have defined a “meta-generation process” responsible for managing the generation process at the level below. A general view of the activities defined for the Meta-generation level is presented in Figure 4.

![Figure 4. Overview of the meta-generation level](image)

The **Obtain goals** activity is responsible for obtaining the system goals that are relevant for the meta-generation level. As high-level goal, we consider the desired coverage level that must be achieved by the integration testing process. For calculating the coverage level of a test suite, it is necessary to define the corresponding coverage criteria. For now, we assume the invocation of the methods provided by the components’ interfaces as coverage criteria, thus considering that the testing coverage level can be calculated based on the number of methods that are activated by a test suite in relation to the total number of methods provided by the components of a particular architectural configuration. We could also consider the interfaces of all components or another criteria [14].

The second activity, **Obtain concrete configuration**, represents the decision making associated with the selection of the system configuration (not within the scope of this paper) that will be tested. This selected configuration is then used for identifying the different component methods that need to be tested, serving as basis for calculating the coverage level achieved by a particular test suite.

The next activity (**Find test cases**) is responsible for the decision making associated with whether or not it is possible to obtain the desired coverage level with the test cases that are available for the components of the configuration. For now we assume that test cases are available and associated to each concrete component in the form of meta-data, although they could be dynamically generated. Test cases also contain meta-data information identifying which component methods they activate. Based on this, this activity involves the transformation of the goals, concrete configuration, and test cases information into a format that can be used by a test case selection tool for finding a test suite that achieves the desired coverage level. After running the test cases selection tool, this activity interprets its output, producing a list with the test cases that have been selected for each component of the selected configuration.

The last activity, **Configure “Obtain test cases”**, is responsible for configuring the **Obtain test cases** activity of the generation level. This activity receives as input the lists produced by the **Find test cases** activity, and uses those lists for changing the parameters of the generation workflow (i.e., the test cases that will be used for each component of the configuration).

As previously mentioned, the Meta-generation level can be activated when either the goals change or the Generation level below is not able to generate a workflow for a given set of inputs. In the latter scenario, the process returns to the **Obtain concrete configuration** activity and tries to obtain new concrete configuration for the system, which implies a change in the set of test cases available. In case of a problem in the **Find test cases** activity (e.g., it is not possible to achieve the desired coverage level with the selected configuration and its respective test cases), the process also tries to obtain a new concrete configuration, with a new set of available test cases. If there is no other configuration available, the process goes back to the **Obtain goals** activity where a decision about whether to attempt a new goal (possibly relaxing the criteria being considered, e.g., reducing the desired coverage level in 10%), or ending the whole process with an exceptional outcome is made.

In case of a problem during execution of the generated workflow, i.e., the configuration does not pass the tests, a new workflow is generated based on another configuration and its respective test cases. In case no configurations are able to achieve the specified coverage level, the generation process is modified for using a different coverage level, and possibly, different adequacy criteria.

### 4. Case Study

In order to evaluate our approach, we have implemented a prototype by extending the supporting infrastructure defined for our framework for the dynamic generation of workflows [7]. This extension involves the inclusion of a new component that is used to select the test cases that will be used during the integration testing. As previously mentioned, we assume that all test cases are available, and stored in a repository. Different from our previous work [7] where all test cases of a component were exercised, we now try to minimise the number of test cases that are selected for achieving a particular coverage level. For doing so, we employ MINTS [10], a multi-criteria test suite minimisation tool based on integer linear programming (ILP). MINTS expects a series of test related data encoded as a binary ILP problem consisting of a set of linear equality and inequality constraints, and a linear objective function. These are converted into a Pseudo Boolean Evaluation format that is fed into an ILP solver. In this way, this new component
encapsulates the transformations from goals, configurations and test case information into a MINTS format, the running of MINTS for finding a minimal test suite that can achieve the desired coverage level, and the interpretation of MINTS output and its transformation into a format that can be used by the remain of the Meta-generation process.

This prototype has been used for conducting some preliminary experiments for demonstrating the feasibility of our approach on generating different workflows depending on changes in the high-level goals.

In the sequence, we present the application that has been used as case study during our experiments, followed by a demonstration of the meta-generation process for showing that different workflows are indeed generated depending on the high-level goals. We conclude this section with a brief discussion about our approach.

### 4.1. Example Application

As a case study application, we have used an XML parser implemented as a Java component library, called NanoXML\(^2\). We have also considered an application program JXML2SQL that uses the services of NanoXML.

Following the work by Orso et al. [13], we have grouped the NanoXML library into four components: Parser, Validator, Builder, and XMLElement handler, plus one component for the JXML2SQL application. Figure 5 depicts a possible architectural configuration for the NanoXML library and the JXML2SQL application, following the grouping of components previously established. According to the division between abstract and concrete configuration, each component instance has a logical name and type associated with it. The following convention is followed in the naming of components: BuilderV4 :C4 :Builder indicates that the component instance BuilderV4 is associated with the logical name C4 and of type Builder.

![Figure 5. Architectural configuration of the NanoXML library and JXML2SQL application.](image)

This application has been obtained from the Software-artifact Infrastructure Repository (SIR) [9], which is a repository containing applications and test related data that can be used for performing controlled experiments. SIR provides different versions of NanoXML, together with test suites and other test related elements for those different versions. Among the test related information provided by SIR, we can find a test suite with test drivers, inputs and expected outputs for each test case, and trace files mapping test cases to methods they activate.

Since in SIR no trace information is stored regarding the JXML2SQL application, the selection of test cases considers only the components of the NanoXML library. In this way, the test cases of the JXML2SQL application have not been included in the MINTS tool, and this should not affect the outcome of our evaluation since the JXML2SQL application is being used just as a driver for NanoXML.

### 4.2. Demonstration

In the sequence, we describe one of the experiments conducted for demonstrating the generation of different workflows depending on changing goals. For this experiment, we assume that an SLA specifies different acceptable coverage levels for the integration testing of the system.

Starting with the Meta-generation level, we considered that the goals obtained correspond to a coverage level of 70\%, while the concrete configuration considered is the one presented in Figure 5.

The Find test cases activity transforms this information into the format supported by the MINTS tool [10]. Table 2 presents parts of the input created for this example. The first line of the table identifies the test cases available. The first group identifies the component under test by each test case available. For example, test case \(t_1\) has been made for testing component Parser. The second group captures information on method coverage. We have considered all public methods provided by the components’ interfaces that are activated by any of the available test cases. In this way, we have a total of 45 provided methods distributed amongst the four components that compose the NanoXML library. For example, test case \(t_1\) activates methods in all components of the NanoXML library. The third group corresponds to the coverage level achieved by each test case in terms of the number of methods activated. For example, test case \(t_1\) activates a total of 16 methods from the 45 available.

<table>
<thead>
<tr>
<th>Component Under Test</th>
<th>(t_1)</th>
<th>(t_2)</th>
<th>(t_3)</th>
<th>(t_4)</th>
<th>(t_5)</th>
</tr>
</thead>
<tbody>
<tr>
<td>XML Element</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Validator</td>
<td></td>
<td></td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Builder</td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Parser</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Methods</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XMLElement.getName()</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XMLElement.hasChildren()</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Validator.elementStarted()</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Builder.addAttribute()</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Parser.parse()</td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td></td>
</tr>
</tbody>
</table>

| Coverage level | 16 | 24 | 11 | 15 | 15 | ...

Table 2. Test related data fed into MINTS tool.

\(^2\)http://nanoxml.sourceforge.net
For calculating the coverage level, we considered a simple sum of the coverage level achieved by each test case. In this way, the test suite provided activates a total of 4436 methods, and this number has been used as a reference for calculating the number of methods that need to be activated for achieving a desired coverage level. Thus, a desired coverage level of 70% corresponds to a test suite that activates 3105 methods.

After all this information is assembled, the Find test cases activity runs the MINTS tool, which tries to minimise the test suite, whilst trying to maintain the desired coverage level. Among the considered criteria, we have included a constraint that at least one test case must be selected for each component.

In the current example, considering a desired coverage level of 70%, the tool selected a total of 161 test cases. In this particular example, the number of test cases selected per component is as follows: XMLElement – 54, Validator – 34, Builder – 32, and Parser – 41. The resulting output, which identifies the selected test cases for each component, is interpreted and used by the Configure “Obtain test cases” activity, which changes the “Obtain test cases” activity of the generation level accordingly.

As previously mentioned, the generation level of our framework is divided into two phases. At the strategic phase, an abstract workflow is generated, which is presented in Figure 6, where we can notice tasks for integrating (IntegrateComponent) and testing (TestComponent) components. Each IntegrateComponent task is associated with an abstract sub-workflow that changes the system configuration for including the component being integrated (showed in [7]). On the other hand, each TestComponent task is an abstract task that represents a placeholder for a sub-workflow that controls the execution of the test cases associated with its respective component.

At the tactical phase, the logical names of the workflow are replaced by concrete components, and the TestComponent tasks are associated with sub-workflows created by the Obtain test cases activity. For example, the sub-workflow associated with the task TestComponent (C4), responsible for testing component BuilderV4, is composed of 32 tasks, corresponding to the 32 test cases that have been selected for testing the Builder component.

In case the goal management selects a different coverage level, the set of test cases selected for each component might be different, which would result in a different concrete workflow for integrating and testing the system. To verify this, we have repeated this experiment, forcing a failure in the generated workflow, and observed whether the meta-generation process is able to generate a new concrete workflow. In this case, the SLA specifies that a coverage level of 60% is acceptable with a reduced price, and the system automatically tries to generate a new workflow with the new coverage level. In this scenario, MINTS selected a total of 158 test cases, with the following distribution: XMLElement – 55, Validator – 34, Builder – 20, and Parser – 49. Thus, resulting in different concrete workflows regarding the TestComponent sub-workflows, e.g., the TestComponent (C4) task is now associated with a sub-workflow with 20 tasks.

4.3. Discussion

In this section, we briefly evaluate our approach, and identify potential directions for improvement.

Our initial experiments have shown that by changing the parameters of an activity, we are able to change how workflows are generated, resulting in the generation of different workflows. More specifically, we have focused on how different goals affect the generation of workflows based on the selection of the test cases that will be used during the integration testing of a particular configuration.

Although we focused on a single parameter (i.e., selection of test cases based on coverage level), other parameters can also be subject to changes based on high level goals. One example is the definition of the integration order. The calculation of the integration order can be divided into two phases: the generation of a dependency graph capturing
the relationships between the architectural elements being considered, and the application of an algorithm on this dependency graph. The first phase can be tuned by modifying the coverage criteria (e.g., provided/required interfaces or methods) that should be considered based on the testing goals. For now, we considered the dependency between provided/required interfaces of components, but it is in our plans to investigate how the integration order is affected by the use of different coverage criteria specified by high-level goals and its impact in the generation of workflows.

The experiments conducted, although preliminary, have also shown the feasibility of integrating testing related tools with our framework for the dynamic generation of workflows, demonstrating the potential of our approach in providing subsidies for the development of a fully automated testing environment. This is further evidenced by the use of a real software application as case study.

However, several technical challenges still need to be solved. For example, how to represent other high-level testing goals (e.g., coverage criteria and software quality characteristics) in a way that can be used by a machine, and how to derive different testing scaffolding from high level testing goals. Other issues involve the selection and generation of test cases, for example: how to effectively capture the properties of existing test cases, and how to dynamically generate meaningful test cases for dynamic configurations in a way that can also be used during run-time.

5. Conclusions

This paper has presented a framework for the metageneration of plans for self-adaptive software systems, which provides support for the dynamic adaptation of the processes that are responsible for generating the workflows that manage the integration testing of component-based software systems. The objective of the work was to demonstrate that we are able to adapt the way processes are dynamically generated according to system goals, which can change during run-time. In the context of component-based integration testing, the dynamic management of the generation process is guided by the testing goals, like, coverage level, that affect the selection of the test cases to be used for testing the system. We have implemented a prototype for evaluating the feasibility of our approach, showing that depending on the desired coverage level, different tests need to be performed, which implies the generation of different workflows.

For now, adaptation is restricted to changing the parameters associated with the activities of the generation process. However, as our overall aim is to build an automated software testing environment that can be dynamically adapted depending on high level goals, we intent to investigate the structural adaptation of the generation process as future work. Our initial plan is to introduce support for structural changes at the generation level based on the identification of process templates that can be used for building the generation process.

References