ABSTRACT

One approach for achieving runtime adaptability in software is to use application frameworks that are tailored for the development of self-adaptive systems. In this paper, we present the Graph-based Runtime Adaptation Framework (GRAF), which enables adaptivity by creating, managing, and interpreting graph-based models of software at runtime. Having a generic graph representation in our approach allows for flexible adaptation via query and transformation operations. The framework is especially suited for the migration of legacy applications towards adaptive software and attempts to reduce necessary changes to the original software. As a proof of concept, we conduct a comprehensive case study of migrating the legacy game Jake2 to achieve runtime adaptivity using GRAF.

Categories and Subject Descriptors

D.3.3 [Programming Languages]: Language Constructs and Features—Frameworks; K.6.3 [Management of Computing and Information Systems]: Software Management

General Terms

Design, Experimentation, Measurement, Performance

Keywords

Adaptation framework, runtime adaptivity, self-adaptive software, model transformation, models at runtime

1. INTRODUCTION

A strong trend in software engineering is to move towards model-based and model-driven engineering. As models provide abstract and customized views of software they can help to reduce complexity by hiding irrelevant details [11]. Accordingly, most ongoing research that is related to self-adaptive software systems (SASS) follows the same trend [4].

In practice though, modeling is one of the major challenges in engineering SASSs [13]. The construction of precise and accurate models of software that address the demands of self-adaptive systems is a hard task, given the highly complex and dynamic nature of such systems. Difficulties arise, especially in the area of model-centric runtime adaptation, where we need to generate, manipulate, and manage models at runtime [5, 20].

The most common reference architecture for SASS comprises two main subsystems: the adaptation manager (autonomic manager) and the adaptable software (managed element) [16]. The adaptation manager acts as an external controller that observes the adaptable software for changes in its operating states and selects appropriate adaptive behavior accordingly. These two subsystems are connected through sensor and effector interfaces, which shapes a closed feedback control loop. In a model-centric approach, an adaptation manager controls the adaptable software by manipulating its runtime model, instead of directly operating on the adaptable software. This reflective architecture [2, 6] is sketched in Figure 1 and demands additional processes to complete the traditional adaptation loop.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Figure 1: Reflective architectural blueprint of a model-centric SASS.
In such a SASS, the state of the adaptable software is propagated to its runtime model (Reflection). These changes are noticed by the adaptation manager (Sensing). The adaptation manager analyzes and plans for the change (Controlling) and adjusts the runtime model accordingly (Effecting). Finally, adaptive changes made on the runtime models are propagated back into the adaptable software (Reflection).

Following this architectural blueprint, we believe that developing comprehensive and flexible frameworks that are able to utilize generic and customizable model representations at runtime, is a promising way to ease the creation of model-centric SASSs. For example, the Rainbow framework [8] successfully incorporates models to achieve adaptivity at the architectural level.

In this paper, we present the Graph-based Runtime Adaptation Framework (GRAF) for engineering SASSs. GRAF relies on the interpretation of behavioral models during the reflection step to achieve compositional adaptation, while also supporting parameter adaptation [18]. It uses a generic graph representation for the implementation of runtime models. The framework can be easily integrated with existing software to enable adaptivity. It extensively supports separation of concerns between the application’s business logic and the logic for adaptation by GRAF’s loose coupling with the legacy system. Additionally, the original system preserves its default (non-adaptive) behavior, once it is decoupled from GRAF.

The rest of this paper is organized as follows: in Section 2, we give a brief overview of our target architecture for model-centric SASSs. Section 3, presents the core features of GRAF. We illustrate how this framework behaves at runtime to achieve adaptivity using models in Section 4. In Section 5, we describe adaptation scenarios to make a legacy game engine adaptive using GRAF, and we evaluate the performance of such scenarios. In Section 6, we discuss the advantages and shortcomings of GRAF. Related work is covered in Section 7. Finally, we conclude this paper in Section 8 and give an outlook on possible future work.

2. ARCHITECTURAL OVERVIEW

In [1], we proposed a model-centric target architecture for a SASS for application-level adaptation. This adaptation approach specifies, that all sensing and effecting operations go through the application itself.

This architecture is based on layers, as depicted in Figure 2. The base-layer of this reflective architecture is the Adaptable Software. It is an evolved version of the original software and represents the managed subsystem of the SASS. Its program elements like classes, methods and attributes extend the elements of the legacy system. Decoupling the adaptable software from the adaptation framework, it behaves as prior to the migration. However, non-functional properties, such as execution performance, may differ between the legacy system and adaptable software.

On top of the adaptable software, we have a three-layered adaptation framework with: (i) the Middleware Layer, which acts as an intermediate layer to facilitate reification and reflection of runtime models via state variable adapters and model interpreters, (ii) the Runtime Model Layer, which is the meta-layer of this reflective architecture, and (iii) the Adaptation Management Layer, which serves as the SASS’s controller unit.

Figure 2: Our model-centric target architecture for a SASS [1].

One of our main objectives is to ensure loose coupling between the adaptable software and the adaptation framework. As depicted in Figure 2, the adaptation framework is glued to the adaptable software via a set of interfaces. Those elements of the legacy software that are in need of adaptation, or can provide information about changes in the application’s operating environment, must be exposed to the adaptation framework. GRAF assumes that any adaptable software provides specific interfaces, as described in Table 1.

Table 1: Interfaces to be provided by the adaptable software [1].

<table>
<thead>
<tr>
<th>Interface</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>StateVar</td>
<td>exposes variables that hold information about the operating environment. Changes in their values are propagated to the model manager, which decides if the value is (i) stored in the runtime model (Reflection), or (ii) discarded.</td>
</tr>
<tr>
<td>SyncStateVar</td>
<td>exposes variables, similar to StateVar, but their representation in the runtime model can be also changed. The new value is then propagated back to the adaptable software.</td>
</tr>
</tbody>
</table>
| Interpreta-
|-------------------|-----------------------------------------------------------------------------|
3. CORE FEATURES OF GRAF

GRAF is a realization of the adaptation framework as described by the target architecture of a model centric self-adaptive software system illustrated in Figure 2. It is implemented in Java, and can be utilized by standard or enterprise Java applications for the purpose of runtime adaptation. We can start and stop GRAF either as a stand-alone Java application, or from within the adaptable software using the provided API.

GRAF uses TGraphs to describe and store state variables and behavioral models. TGraphs are (i) typed, (ii) attributed, (iii) ordered, and (iv) directed graphs [14]. The creation and manipulation of TGraphs is supported by JGraLab [15]. This versatile class library can generate Java source code based on meta-models (schemas). The generated classes can be used in custom applications to utilize TGraphs as data structure. That way, model query and transformation operations can be realized using this lower level Java API.

To create and expose the interfaces described in Table 1 as well as for causally connecting [17] the adaptable software to the adaptation framework, GRAF uses JBoss AOP\(^1\), an aspect oriented programming (AOP) framework for Java. The necessary code for value propagation of state variables from the adaptable software to the runtime model, as well as the code for model interpretation, is injected at previously annotated program elements.

In the rest of this section, we give an overview of the most distinguishable features of GRAF together with the involved technologies.

3.1 Runtime Modeling

The design of GRAF’s architecture is centered around a runtime model. Essentially, this model needs to offer a view on the adaptable software’s state, as well as describe behavior that can be executed by an interpreter. Hence, the model serves as the actual view on the adaptable software, and must be controllable and modifiable by the rule engine.

In GRAF, the runtime model conforms to a schema (metamodel). The schema is customizable and is described as UML class diagrams in standard XMI format. GRAF is able to automatically generate the Java source code to create and manipulate models that conform to the schema as described in UML. The schema that is used in this research is illustrated in Figure 3. According to this schema, every runtime model is composed of StateVariable vertices and Activity vertices.

In this schema, behavioral models are in a form of UML activity diagrams, for modeling behavior at the runtime model level the runtime model schema must express their abstract syntax. However, other types of behavioral models such as Petri nets, state diagrams, etc. can be easily adopted by replacing the schema. Furthermore, state variables are used to store exposed data from the adaptable software.

3.2 Runtime Model Generation

GRAF can generate a default runtime model by utilizing the source code annotations fully automatic. This initialization task can be carried out either before or after running the adaptable software. Generating the default runtime model before startup prevents the overhead of recreating the same model over and over again. On the other hand, this process can ensure that the runtime model matches the current state of annotations in the adaptable software’s source code.

In cases where the automatic generation process is not sufficient, it is desirable to enhance the runtime model manually before startup. The same generation process can be triggered manually first to obtain an initial runtime model that can then be adjusted. For instance, complex behavior descriptions may be added. Following such a refinement, the final model is passed to GRAF to be used at runtime.

3.3 Reification, Reflection and Interpretation

GRAF synchronizes the adaptable software and its runtime model. It dynamically propagates any changes in the application’s state to the runtime model (Reification) and propagates changes in the runtime model back to the application (Reflection).

Reification and reflection is achieved with a set of state variable adapters. For the synchronization of state variables, the injection of state variable adapter code assures that variables in the adaptable software read their representation in the runtime model on use.

For reflection of behavior, the model interpreter executes parts of the behavioral model at predefined points in the control flow (interpretation points). We are motivated by the fact that generic software is often built using runtime interpretation of models, encapsulated in a separate component and used by a large part of the software system. Given the success of model-based software, we follow the idea of model interpretation and apply it to runtime adaptivity.

GRAF’s model interpreter walks along paths of a behavioral model, which describe behavior to be executed at an interpretation point. Java reflection is only used in cases where the described behavior in the activity is marked to be interpreted. Based on the schema depicted in Figure 3, each Action vertex of the associated Activity in the behavior model has a trace link to a method that implements the given actions in the adaptable software. The model interpreter resolves information stored in the runtime model, such as the qualified method names, and uses Java reflection to invoke them.

3.4 Automatic Binding to Adaptable Software

When the adaptable software is started, it passes a set of properties to the Java Virtual Machine to be used for AOP. One of these artifacts is a pointcut definition file, which is provided by GRAF. Based on these pointcuts and with the help of the middleware adapters, byte-code manipulation on the compiled and annotated Java classes is performed at load time of classes to establish a binding between the adaptable software and GRAF. Hence, binding the interfaces listed

\(^1\)http://www.jboss.org/jbossaop/

Figure 3: Excerpt of the runtime model schema.
in Table 1 can be done dynamically either before or after running the adaptable software.

3.5 Model Querying and Transformation

GRAF’s adaptation management layer is built around a rule engine and a repository of adaptation rules that can be thought of as event-condition-action rules. Whereas an adaptation rule is bound to certain types of events, the condition in which it shall be executed is essentially a query on the runtime model. The action, then, is a model transformation, which adapts the runtime model.

Queries are implemented using the Graph Repository Query Language (GReQL) [9], a schema-sensitive expression language developed at the University of Koblenz-Landau. A GReQL expression allows the querying of TGraphs and specifies the range of some free variables, poses conditions on these variables, and finally describes the desired output.

GRAF can transform runtime models via a lower level API that is automatically generated from a runtime model schema using JGraLab tools. To simplify the use of transformations for adaptation, GRAF offers the AbstractTransformation utility class. It is used to represent the action part of an adaptation rule. Potentially, GRAF can also take advantage of Graph Repository Transformation Language (GReTL) [14] to perform transformations on TGraphs. However, GReTL is an ongoing research project (and Java API) and GRAF does not use it at this point in time.

In addition, GRAF supports the validation of constraints on the runtime model, which also uses GReQL expressions. A set of constraints is already defined at the level of the runtime model schema. Examples are the multiplicities for associations that restrict the allowed edges between vertices in the runtime model’s internal TGraph representation. Developers of a SASS can additionally write constraints that are specific to the domain of the adaptable software. For instance, it is possible to express that for a certain behavior (activity), specific actions (methods) are not allowed to be composed in sequence. Constraint validation is out of scope for this paper though.

Finally, GRAF supports the storage of temporal snapshots of the runtime model as additional state information. The adaptation management layer can access this repository by querying it via the model manager. This history repository facilitates the application of data mining techniques to be used by the adaptation management layer.

3.6 Administration and Configuration

GRAF allows administrators to influence the adaptation steps to be performed via a control panel. Administrators get access to the components as well as repositories that are located in the adaptation management layer. Similarly, an external adaptation manager such as StarMX [3] can be integrated with GRAF to automate administration tasks.

As TGraphs are serializable, they can be persistently stored in and retrieved from text files in the JGraLab-specific TG format, which allows to compactly store a graph together with its schema in a human-readable form. In addition, a runtime model snapshot can be exported and visualized based on the DOT format [12].

4. RUNTIME ADAPTATION USING GRAF

After introducing the overall structure and features of GRAF, we discuss the framework’s runtime behavior in this section, covering five main steps as described in Figure 1.

Subsequently, we give an overview on the main steps in the form of scenarios using separate process diagrams for illustration. Note that we ignore all the possible alternate flows and exceptions that might occur during the adaptation scenarios on purpose here. The main idea is to show the control flow of GRAF under typical conditions.

4.1 Reification and Sensing Scenarios

The reification scenario of GRAF involves the propagation of changed values from the adaptable software to the runtime model. This results in a notification of the rule engine in the adaptation management layer, which is the sensing scenario. Figure 4 illustrates the control flow of these two scenarios. Assuming application-level adaptation as discussed in [23], these changes are reflected by the state of the adaptable software and thus can be sensed through its variables and fields.

![Figure 4: Steps involved in the reification and sensing scenarios.](image)

This first scenario starts with a change to the adaptable software and its data, as the system communicates with its operating environment. If the changed variable is an instrumented Java field that is exposed via the StateVar or SyncStateVar interfaces, a state variable adapter of GRAF takes care of sending the updated value to the model manager, which decides if the propagated value should be stored in the runtime model or not. Finally, the model manager notifies the rule engine in case of changes to the model.

4.2 Controlling Scenario

The IBM vision of autonomic computing introduces the MAPE-loop as a four step process: (i) monitoring, (ii) analyzing, (iii) planning, and (iv) executing for external controlling of software [16]. The controlling scenario of GRAF is essentially completes the MAPE-loop. The control flow of this scenario is illustrated in Figure 5.

![Figure 5: Steps involved in the controlling scenario.](image)
changes and answers by passing a sequence of adaptation rules to be applied on the runtime model. Optionally, the rule engine may use the runtime model and it’s change history by querying them via the model manager for a more detailed evaluation e.g., taking past actions into account.

Considering the repository of available adaptation rules and based on its own heuristic sand strategies, the rule engine selects a sequence of adaptation rules to be applied to the runtime model. In the last step, the rule engine triggers the invocation of adaptation rules and its transformations on the runtime model, again through the model manager.

4.3 Effecting Scenario

The effecting scenario of GRAF involves a transformation of the runtime model that is based on the adaptation rules, which were selected by the rule engine in reaction to changes in the adaptable software. Figure 6 shows the control flow for this scenario for a single adaptation rule.

![Figure 6: Steps involved in the effecting scenario (for each adaptation rule).](image)

Firstly, the model manager checks the pre-condition of the adaptation rule on the runtime model. If it evaluates to false, the adaptation rule is ignored. Otherwise, the adaptation rule’s transformation is executed. That followed, the model manager checks and validates the runtime model against the schema and its constraints, as well as the adaptation rules’ post-conditions. In case of any conflicts and violation with the executed transformation the model manager rolls back all the changes and notifies the rule engine to take another action.

If the executed transformation is valid, the runtime model is successfully adapted to the new environment based on the existing adaptation rules, given model constraints, and according to the heuristics of the rule engine.

4.4 Reflection Scenario

The reflection scenario of GRAF is responsible for an actual behavior change of the managed adaptable software. Two different ways are supports: (i) adaptation via parameter values, and (ii) adaptation via interpretation of a behavioral model.

4.4.1 Adaptation via Parameters

State variables can be setup so that changes to them are reflected back to the adaptable software. For this mechanism to work, the corresponding variable of the adaptable application must be exposed to GRAF via the SyncStateVar interface. If the runtime model representation of such a state variable is changed by an adaptation rule, then the adaptable software is aware of the new value whenever the variable is used in the adaptable software.

4.4.2 Adaptation via Model Interpretation

GRAF supports the modeling of behavior (using UML activity diagrams at the moment). As illustrated in Figure 7, the model interpreter executes an activity at specific points in the adaptable elements’ control flow.

![Figure 7: Steps involved for interpreting in the reflection scenario.](image)

During execution of some operation of the adaptable software which is exposed to GRAF via the InterpretationPoint interface, the model interpreter is invoked and loads the associated behavioral model. If it describes default behavior, there is no need for adaptation and the interpreter returns this information so that the adaptable element executes the existing default code block in the adaptable software. Otherwise, the corresponding part of the behavioral model has changed to describe something else than the default behavior and the interpreter executes the behavioral model.

Each atomic action used in the behavioral model is associated with methods in the software application via the AtomicAction interface and the interpreter can invoke them. The control flow of the application keeps executing, by returning from the (adapted) method.

5. CASE STUDY: JAKE2 GAME ENGINE

Computer games are among the most resource intensive application domains. Providing the best quality of service and performance is crucial in modern games [21]. Additionally, gamers are always eager to be challenged by more powerful and more intelligent games. We believe that computer games can extensively benefit from autonomic computing and runtime adaptation. Specially, augmenting more intelligent behaviors and adaptive features, in a cost effective way, can prolong the game’s lifetime and increase user satisfaction. Thus, we selected Jake2 legacy game engine as the case study of this paper to demonstrate the usage and evaluate the performance of GRAF.

Jake2 is a Java port of the GPL release of the Quake II game engine, a first person shooter game originally written in C. The 0.9.1 version of Jake2 was shown by the JOGL team for JavaOne 2004, to present an example of Java-OpenGL interoperability. Jake2 has since been used by Sun as an example of Java Web Start capabilities for games distribution over the internet [19].

[^2]: http://bytonic.de/html/jake2.html
5.1 Adaptation Requirement

In Jake2, a player is able to select the game’s difficulty level as the game starts. Naturally, players improve their performance as they spend more time playing the game. However, the game’s difficulty level is fixed unless the player restarts the game and as the system is not getting any feedback about the player’s performance, the game might become too easy or too hard for the player. Hence, we set our main goal of adaptation as following: “As the game is in progress, the bots (the computer controlled artificially intelligent enemies) should adapt their behavior according to the expertise level of the human player”.

This goal can be achieved by altering the way bots play the game at runtime, thus we need to compose the behavioral model of bots at runtime. Such a compositional adaptation may re-orchestrate current atomic actions, adjusting decision criteria of behavioral models, or use set of new atomic actions. In this case study, we consider the following candidate solutions to satisfy our adaptation requirement:

- **Approaching Enemy Differently**: use different path algorithms for approaching the enemy. For example, instead of running straight towards the target position, the bot could start strafing left and right to be harder to hit by a player. This adaptation can be used to make the game harder for good players.
- **Adaptive Weapon Selection**: instead of a fixed weapon, the bot can adaptively select the most appropriate weapon. For example, set the attack state randomly to either missile or melee, or always use a melee weapon, even when far away to reduce the game difficulty.
- **Cloning and Morphing**: the bot creates clones of itself or re-spawns itself as a stronger/weaker bot. We can consider a number of clones, cool-down times, and bot types. New bots can be created by initializing a set of the parameters, such as current and max health, weapons, and ammo.

As a proof of concept, we decided to present the implementation details for cloning and morphing in this paper.

5.2 Preparing Adaptable Jake2

The first step in integrating Jake2 with GRAF is to evolve its original code and make it adaptable. In [1], we proposed an evolution process that supports this initial step. In this study, we need to comprehend the original source code of Jake2 and locate those elements that implement the bot behavior, in accordance with the state variables that carry information about the performance of human players. Then, we need to refactor current elements or add the missing ones. Finally, we tag these program elements using Java annotations.

Jake2 exhibits a well-structured modular design. Top level modules (packages) of Jake2 are listed in Table 2. As our adaptation requirement asks for a modification in game’s logic, we further study the jake2.game package. The jake2.game.GameAI class of this package embodies a set of methods that implement the intelligent behavior of bots. We investigate the methods of GameAI to locate how the bots perceive, decide, and react in their world. In Jake2, each bot is always standing or running in its partially observable world (idle state). As a bot perceives a target its state changes from idle to attack. This transition is implemented by calling

![GameAI.ai_checkattack()](image)

GameAI.ai_checkattack(). It performs a sequence of tasks to handle the situation. We used UML activity diagrams during the program comprehension step, as this representation is close to the behavioral model representation in this version of GRAF.

A bot runs towards his next target without firing first and checks if this target is dead or not. If it is dead, the bot tries to find a new target. Otherwise, it will show its hostile animation and update its own knowledge about his own distance to the target. If the target is still available and visible, the bot tries to attack in various ways.

In order to modify the behavior of bots, we need to change GameAI.ai_checkattack(). Thus, we annotate this method as @InterpretationPoint. In GRAF, each action in the behavioral model maps to a unique method in code. Therefore, we refactored the original ai_checkattack() method and extracted each action as a new method. Additionally, as the required actions to perform morphing and cloning were missing, we added two methods to GameAI: ai_tryClone() and ai_tryMorph(). We annotated each of the extracted and the two new methods as @AtomicAction.

For measuring a player’s performance we decided to simply monitor the amount of killed monsters. This data is stored in the level_locals_t class, a structure that is cleared, as each map is entered as well as read/written to a save-file for persistency. Hence, we annotated the level_locals_t.killed_monsters integer field as @StateVar. That way, its value is exposed to GRAF and stored in the runtime model.

5.3 Setting up GRAF for Jake2

The setup process of GRAF with any adaptable software consists of a couple of steps, some of them are optional, depending on the adaptation requirements. We introduce these steps briefly here the way they were performed in this case study, neglecting implementation details.

5.3.1 Generating/Refining the Runtime Model

GRAF needs a runtime model specific to the adaptable software. This runtime model can be generated from annotations in the source code of the adaptable software. This can be done automatically at startup of the SASS by passing
in the path to the compiled class or off-line. In the second case, the runtime model is stored persistently in a .tg file, which is then passed to GRAF at startup. That way, a pre-processing step can adjust the runtime model further, e.g., manipulate the default behavior representation using transformations.

Figure 8 illustrates an excerpt of the runtime model, which represents the default attack behavior for bots. We use the concrete syntax of UML activity diagrams here, the internal representation of the actual TGraph conforms to the runtime model schema. This activity has the name of the method that is annotated as @InterpretationPoint.

Furthermore, it is set to be not interpreted, this is noted in Figure 8, by using a UML comment, since the UML activity does not have this property. Finally, the two floating actions Try Cloning and Try Morphing represent the atomic actions that can be used within this behavior description. The second annotation in this figure denotes the declaringElement attribute of the Action vertex, which is used by the model interpreter to invoke an actual method of the adaptable software. This meta-data is just shown for one action only here.

5.3.2 Specifying Adaptation Rules

Achieving the adaptation requirements is mainly done by writing adaptation rules. The runtime model that is finally passed to GRAF at startup of the SASS acts as a baseline for these transformations. In the context of GRAF, adaptation rules are event-condition-action rules that describe atomic adaptation tasks, using GReQL for expressing conditions and the generated Java API from the runtime model schema, for describing transformations. GRAF also comes with a set of (abstract) helper classes for reducing boiler-plate code.

In this case study, the adaptation rule’s action (transformation) that enables cloning redirects the outgoing edge of Update Knowledge to point to Try Cloning, so that it does not connect to the second DecisionNode vertex any more. A new edge is created to connect Try Cloning to a new, third DecisionNode vertex, where the output of Try Cloning is checked using a simple GReQL expression. These expressions can make use of the names of OutputPin vertices (shown in schema, but hidden in this instance view). For the case that cloning was successful, the third decision node is connected to the FinalNode vertex. Otherwise, AI should check for a missile or melee attack as before.

5.4 Evaluating GRAF/Jake2 Performance

To investigate the usability of GRAF for enabling runtime adaptivity in Jake2, we prepared five system variations, as follows:

- **S0**: Original version of Jake2.
- **S1**: Refactored and annotated version of S0 as prepared by following Subsection 5.2.
- **S2**: Starts GRAF. Aspect codes are woven at annotations. The system is integrated with GRAF. No adaptation rules, which means that the default behavior is always under effect. The default behavior is achieved by executing the source code (no model interpretation).
- **S3**: Similar to S2, but the default behavior is achieved by interpreting the default runtime model.
- **S4**: Adaptation rules and model constraints are added to S3. GRAF transforms and interpret the behavior.

The transient variations from S1 (original version) to fully adaptive Jake2 in S4 allow us to measure the overhead of refactorings to prepare atomic actions, AOP, runtime model, model interpretation, adaptation rules (as queries and transformations on runtime model). Table 3 summarizes the difference among all variations.

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>ID</th>
<th>S0</th>
<th>S1</th>
<th>S2</th>
<th>S3</th>
<th>S4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Atomic Actions &amp; Annotations</td>
<td></td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Start GRAF &amp; Runtime Model</td>
<td></td>
<td>-</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Interpretation &amp; Reflection</td>
<td></td>
<td>-</td>
<td>-</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Adaptation Rules</td>
<td></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>

All the experiments are run on an Intel Core 2 Duo E8400 workstation with 4.0GB DDR2 memory, running MS Windows XP Professional 32-bit SP3 and Java Platform, Standard Edition 6. We used JConsole (from Oracle JDK 6),
the Eclipse Test and Performance Tools Platform (TPTP)\(^3\), 4.7.1, and FRAPS\(^4\) 3.2.6 for the purpose of profiling and benchmarking Jake2 variations.

Executing each variation, we waited for one minute on the game’s start menu and started a new game with default settings and beginner difficulty. As the game started we killed the first bot (to ensure that the condition for the adaptation rule evaluates to true at least once). Following a fixed path, we played the game for 2 more minutes before returning back to the spawn location in the game.

### 5.4.1 Memory Utilization

Investigating the utilized resources among different variations of Jake2 can help us to study the usability and scalability of GRAF. Table 4 summarizes the memory utilization of each variation and the total number of the classes loaded. The results for S0 and S1 are almost identical, which indicates that we can safely replace original software with its adaptable version from this perspective.

<table>
<thead>
<tr>
<th>Metric</th>
<th>ID</th>
<th>S0</th>
<th>S1</th>
<th>S2</th>
<th>S3</th>
<th>S4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Max Used (MB)</td>
<td></td>
<td>98.2</td>
<td>98.8</td>
<td>170.5</td>
<td>165.9</td>
<td>171.0</td>
</tr>
<tr>
<td>Max Committed</td>
<td></td>
<td>112.4</td>
<td>112.4</td>
<td>213.6</td>
<td>213.1</td>
<td>215.2</td>
</tr>
<tr>
<td>Loaded Classes</td>
<td></td>
<td>3799</td>
<td>3799</td>
<td>5648</td>
<td>5660</td>
<td>5754</td>
</tr>
</tbody>
</table>

As we integrated and started GRAF in S2, S3, and S4, we observe a constant increase in memory utilization. There is an average of 72% increase in the total amount of used memory (heap and non-heap), 30% increase in the maximum committed memory, and 50% increase in the total number of the loaded classes. In conclusion, in the case of Jake2 GRAF we had a quite significant but linearly constant overhead in memory use as we move from non-adaptive variations to the self-adaptive ones. However, model interpretation and the adaptation rules do not add a significant memory overhead.

### 5.4.2 Execution Performance

All variations exhibit a similar overall game performance: \(46 \pm 1\) average CPU% and \(67 \pm 1\) frame per second. This means that the memory overhead of using GRAF and runtime adaptation does not effect the overall game experience.

![Figure 9: ai_checkattack() execution times box-plot.](http://www.fraps.com/)

We further investigate the execution performance of GameAI class and specifically `ai_checkattack()` as summarized in the box-plot of Figure 9. The evaluation results reveal that adaptable Jake2 (S1) with the refactored `ai_checkattack()` method has almost identical performance compared to S0. S2 has a light overhead as GRAF is started and the annotated elements are wrapped by aspect codes. The execution times are higher, but still in an acceptable range, in S3 and S4. In this two adaptive versions, the desired behavior of bots are achieved via interpretation. S4 has higher upper quartile than S3 as the alternate behavior model in S4 also performs bot cloning. Additionally, there are more upper overlays in S4 as the adaptation rule is enabled and occasionally transforms the runtime model.

Another observation from Figure 9 is the overhead of dynamically weaving the aspect code. This only affects the first call to each annotated program element and we can see a large gap between the first execution and the rest of the method calls. However, this can be prohibited by weaving the aspect code offline.

### 6. DISCUSSION

In Section 5, we put GRAF into practice. First, it is noteworthy that for preparing the adaptable software, developers can decide whether to achieve adaptivity by adding branches to the source code that depend on an exposed state variable, or if functionality is made available as atomic actions to be used for modeling (alternative) behavior. From the framework’s perspective though, adaptivity is achieved mainly by querying and transforming the runtime model. Hence, GRAF covers parameter and compositional adaptability in a unified way.

Second, we decided to use UML activity diagrams to model behavior, since they are close to the source code representation, as we focus on adaptivity at the method-level. There are other alternatives though, such as state charts, petri nets, or feature trees. The appropriateness of these languages could be evaluated depending on (i) the desired self-* properties, (ii) the software application’s domain, or (iii) on associated implementation (dis-)advantages. To introduce a new modeling approach for behavioral models, a new runtime model schema (description of syntax), as well as a model interpreter (description of semantics) need to be provided.

Furthermore, we observed that the precision of the sensed data may vary. Thus, it might be enough to reify changes to the runtime model less often (reification frequency), or developers need to define thresholds so that data is only propagated to GRAF, if a sensor holds a value that is below/above pre-defined limits.

On the implementation side, GRAF uses GReQL expressions frequently at runtime, for example on each access of a state variable from the middleware to update it with a newly reified value. Profiling results revealed that caching Java references to subgraphs of the runtime model can reduce the overhead of query evaluation in cases where nothing changed.

Finally, it is important to describe potential internal and external threats to the experimental validity of this research, as well as possible approaches to prevent the occurrence of these threats. We cover internal and external validity subsequently.
6.1 Internal Validity

There could be several other factors which influence the results and affect our inferences. The tools developed/used to collect data can potentially have defects or effect the performance results. Efforts to restrict this threat were dealt with by considering various profiling tools, applying them to several case studies, and study their impact on results.

We tried to reproduce the exact same scenario for each experiment. However, this was not fully achieved due to the game randomness. One solution to this threat would be to use customized test maps and eliminate any randomness by using pseudo-random generators with fixed seeds.

Another noteworthy point is that the case study is not performed in an embedded and real-time environment, which may effect the results. Although we tried to decrease the background processes, we preferred a more realistic gaming environment over an isolated one.

The JVM provides dynamic memory management and garbage collection. This feature can affect the captured memory utilization results. We manually trigger the garbage collection and use averaging to tackle this threat.

6.2 External Validity

The general threats in this facet are mono-operation and mono-method biases. These threats refer to the cases where only one measure, one case study, or one single method are used as a proof of concept. In this article, we prepared Java2, integrate it with GRAF and manage and interpret its behavioral model to satisfy a given adaptation requirement.

We believe that the selected scenario is detailed enough to show a reasonably complex behavioral model. However, this study is limited to a single high-level adaptation requirement. Nonetheless, we considered several candidate solutions to achieve the adaptation goal. Each of these solutions (e.g. add cloning behavior for bots) can be realized differently as well. For example, the developer may add all missing branches and decision nodes in the source code and control the flow by just tuning the criteria at the decision nodes. Although this solution is not as flexible as the solution we presented in this study and requires more changes in the original source code, we can achieve similar adaptivity.

However, as the notable strength of GRAF, which distinguishes it from other adaptation frameworks, lies in its ability to support method-level compositional adaptation by interpreting runtime models, we set the focus of this study to present this and other model-centric features of GRAF.

Other legacy software with different characteristics (e.g. mission and safety critical systems) may be subject to different adaptation concerns and the overhead of integrating and using GRAF may differ. Controlling this threat can be achieved by applying additional empirical studies to various software projects, preferably from different domains.

7. RELATED WORK

There are several successful frameworks and projects for software runtime adaptation [23]. Some of these frameworks have more common core features with GRAF.

The Rainbow framework [8] uses an abstract architectural model to monitor the runtime properties of an executing system. It has a quite similar model-centric reference architecture as we mentioned in Figure 1. Similar to GRAF, certain components of Rainbow are reusable and Rainbow is able to perform model constraint evaluation and adaptation at runtime. However, this framework mainly targets architecture-level adaptation, where adaptive behaviors is achieved by composing components and connectors, instead of handling behavioral models (such as activity models).

DiVA project [20] considers design and runtime phases of adaptation. It is based on a solid four dimensions modeling approach to SASS [10]. At design time, a base model and its variant architecture models are created. The models include invariants and constraints, which can be used to validate adaptation rules. In comparison to our approach, DiVA also focuses on adaptation at the architectural level, whereas we target adaptation at a lower level by incorporating methods and fields. We also generate models at runtime, where they weave aspects into explicit models of software.

Vogel and Giese [24] propose a model-driven approach which provides multiple architectural runtime models at different abstraction levels. They derive abstract models from the source models. Subsets of the target models focus on a specific adaptation concern to simplify autonomic managers. Similar to DiVA, this research also targets adaptation from an architectural point of view. Both of these approaches rely on the enterprise and component-based architectures, such as Java EE, which make them not suitable for enabling adaptivity in legacy and standard applications.

TRAP/J [22] is a software tool which enables the addition of new adaptable behavior to existing Java applications without requiring changes of the original source code or the virtual machine. TRAP/J utilizes behavioral reflection and AOP techniques. In this approach, developers choose a subset of existing classes and generate a set of aspects and reflective classes for an adapt-ready software. TRAP/J focuses on the technological approach of how to make an existing Java application adaptable. TRAP/J supports refication of method invocations but not the refication of read or write operations on fields.

StarMX [3] is a generic configurable adaptation management framework. It separates management logic from application logic by using explicit sensing and effecting interfaces. StarMX has no dependency on application characteristics (e.g. architecture or environment) or any particular self-* property. Designed for Java-based systems, it incorporates Java Management Extensions (JMX) technology and is capable of integrating with various policy/rule engines.

Similar to some of the presented works, our approach makes use of Java reflection and AOP for the instrumentation of legacy software. In contrast to mentioned research projects, we focus on the interpretation of behavioral models for effecting the managed software application at runtime, while also supporting parameter adaptation. Additionally, the JGRaLab [15] toolkit offers a variety of flexible and powerful ways to operate on models at runtime, including the query language GReQl [9] and a rich Java API that is automatically derived from the runtime model schema.

8. CONCLUSIONS AND FUTURE WORK

Following our proposal for a model-centric SASS in [1], we implemented GRAF to support the migration of existing software towards a SASS. GRAF inherits most of the common features of current adaptation frameworks, such as support of generic and extensible rule engines, and external control panels. Additionally, runtime adaptivity can be achieved using GRAF based on a combination of (i) query-
ing, (ii) transforming and (iii) interpreting behavioral models as well as state variables, which are causally connected to a software application.

We see plenty of interesting directions which can be followed to improve GRAF. One interesting future direction is supporting concurrent behavioral models. Concurrency can be captured and modeled in runtime models, and by extending the interpreter concurrent activity models can be executed at runtime.

Additionally, advanced AI techniques can improve the overall performance of GRAF. For example, a machine learning-based decision maker may be used to improve the performance of the adaptation manager, or pattern matching techniques can facilitate analyzing the queries on a runtime model history by reveal patterns of successful or faulty adaptation steps. In the same direction, data mining techniques can be used to manage large sets of adaptation rules and constraints more efficiently.

We can also improve the usability of GRAF by adding a graphical control panel to provide humans with intuitive ways of inspecting parts of the runtime model. Moreover, a set of plugins and wizards can be created for the Eclipse IDE to support developers. For instance, adaptation rules could be generated from templates and based on a graphical selection of exposed state variables and actions available.

9. REFERENCES


