



Dynamic Variability in complex, Adaptive systems


Deliverable reference: D4.2	Date: 22 February 2010	Responsible partner: pure-systems
Title: Adaptation model and validation framework first version		
Editor(s): André Maaß Danilo Beuche Michael K. W. Hentze		Approved by: Arnor Solberg
		Classification: public
Abstract / Executive summary:		
<p>DiVA IS A STREP FUNDED UNDER CONTRACT 215412 TO THE SEVENTH FRAMEWORK PROGRAMME, THEME 1.2: SERVICE AND SOFTWARE ARCHITECTURES, INFRASTRUCTURES AND ENGINEERING</p>		 <p>INTERNATIONAL DOCUMENT AVAILABLY: PARTNER + REPORT NUMBER ISBN</p>



Table of Contents

Adaptation model and validation framework first version	1
Table of Contents	2
Version History	5
Contributing partners.....	6
List of Figures.....	7
1 Introduction	8
2 Premises.....	10
2.1 REASONING-SPECIFIC REQUIREMENTS.....	10
2.2 COMPLEXITY REDUCTION.....	12
3 Reasoning and Validation Concept	14
3.1 ARF STRUCTURE.....	14
3.2 REASONING AND VALIDATION USE CASES	16
3.2.1 <i>Requirements Use Case</i>	16
3.2.2 <i>Design-time Use Case</i>	17
3.2.3 <i>Runtime Use Case</i>	18
4 Design-time Validation through DiVA Context Generation.....	20
4.1 ILLUSTRATIVE EXAMPLE.....	20
4.1.1 <i>Modelling runtime variability and adaptation logic</i>	20
4.2 CONTEXT DATA SELECTION CRITERIA	22
4.2.1 <i>Simple coverage criterion</i>	22
4.2.2 <i>Pairwise coverage criterion</i>	23
4.2.3 <i>Dependency coverage criterion</i>	23
4.2.4 <i>Compound coverage criterion</i>	25
4.3 EXPERIMENTS	25
4.3.1 <i>Test data generation</i>	25
4.3.2 <i>Test data evaluation criteria</i>	26
4.3.3 <i>Experimental results and analysis</i>	26
4.3.4 <i>Threats to validity</i>	28
4.4 RELATED WORK.....	29
4.5 CONCLUSIONS.....	29
5 ARF Design	31
5.1 PRELIMINARY CONSIDERATIONS	31



5.1.1	<i>Query and Question</i>	32
5.1.2	<i>Information and Result</i>	33
5.2	INTERFACE DEFINITIONS	34
5.2.1	<i>Data and Parameter Interface</i>	34
5.2.2	<i>Reasoner Interface</i>	36
5.2.3	<i>Query and Question Interface</i>	37
5.2.4	<i>Information and Result Interface</i>	38
5.2.5	<i>ARF Interface</i>	39
5.3	REASONING AND VALIDATION WALKTHROUGHS	40
5.3.1	<i>Requirements-level Walkthrough</i>	40
5.3.2	<i>Design-time Walkthrough</i>	41
5.3.3	<i>Runtime Walkthrough</i>	42
6	ARF Implementation	45
6.1	REASONING AND VALIDATION QUESTIONS.....	45
6.1.1	<i>Are there valid configurations - Question</i>	45
6.1.2	<i>How many valid configurations do exist - Question</i>	46
6.1.3	<i>Get all valid configurations - Question</i>	46
6.1.4	<i>Get one valid configuration - Question</i>	46
6.1.5	<i>Get best valid configuration - Question</i>	47
6.1.6	<i>Get next valid configuration - Question</i>	47
6.1.7	<i>Is configuration valid - Question</i>	47
6.1.8	<i>Reasoning and Validation Parameters</i>	48
6.2	DiVA ADAPTATION MODEL MAPPING	48
6.2.1	<i>Mapping of DiVA Variability Model</i>	48
6.2.2	<i>Mapping of DiVA Context Model</i>	52
6.2.3	<i>Mapping of DiVA Configuration Model</i>	53
6.3	INITIAL REASONING ENGINE SELECTION	54
6.4	ARF REASONER	54
6.4.1	<i>CUDD BDD Reasoner</i>	54
6.4.2	<i>Alloy based SAT Solver Reasoner</i>	55
7	Conclusion and Outlook to D4.3	56
7.1	COMPARING THE IMPLEMENTED REASONER	56



7.2 FUTURE WORK 57

References..... 58

Version History

Version	Description	Date	Who
0.0	Document initialisation	06.05.2009	Michael Hentze
0.1	Introduction	17.07.2009	Michael Hentze
0.2	Added abstract interfaces definitions (AID)	27.07.2009	Michael Hentze
0.3	First AID feedback added and deliverable outline added	28.07.2009	Michael Hentze, Brice Morin, Vincent Girardreydet
0.4	First structure feedback added, WP3 chapter split to validation & reasoning	29.07.2009	Michael Hentze, Brice Morin
0.5	Added ARF Structure and Interfaces	13.08.2009	André Maaß
0.6	Reasoning and validation use cases added, ARF model mapping added, reasoning and validation questions added, new document structure	12.10.2009	André Maaß, Danilo Beuche
0.61	Reasoning and validation questions	23.10.2009	André Maaß
0.7	Reasoner engines added, References fixed, Chapter 5 added	09.11.2009	André Maaß, Danilo Beuche
1.0	Final version for review	16.11.2009	André Maaß
1.1	Document reviewed and new material added	04.12.2009	Freddy Munoz Benoit Baudry André Maaß
1.2	Reviewer's comments included (1 st Review)	21.01.2010	André Maaß
1.3	Reviewer's comments included (2 nd review) Some restructurings	10.02.2010	Vegard Dehlen André Maaß



Contributing partners



pure-systems GmbH
Agnetenstraße 14
39106 Magdeburg
Germany

André Maaß
andre.maass@pure-systems.com

Danilo Beuche
danilo.beuche@pure-systems.com

Michael K. W. Hentze
michael.hentze@pure-systems.com

Brice Morin
bmorin@irisa.fr

Freddy Muñoz
fmunoz@irisa.fr

Benoit Baudry
bbaudry@irisa.fr

Reviewer: Jean-Marc Jézequel
jezequel@irisa.fr

Franck Fleurey
franck.fleurey@sintef.no

Reviewer: Vegard Dehlen
vegard.dehlen@sintef.no

Vincent Girardreydet
vincent.girardreydet@thalesgroup.com



IRISA Rennes
Campus de Beaulieu
35042 Rennes Cedex
France



SINTEF
Strindveien 4
7034 Trondheim
Norway



Thales
Route départementale 128
91767 Palaiseau Cedex France



List of Figures

FIGURE 1 WP4 AS A CROSSCUTTING WORK PACKAGE.	8
FIGURE 2 DiVA METHODOLOGY WORKFLOW.....	14
FIGURE 3 THREE LEVEL APPROACH.....	15
FIGURE 4 ARF STRUCTURE.....	16
FIGURE 5 MODEL OF THE CONTEXT OF THE SYSTEM.....	21
FIGURE 6 MODEL OF THE VARIABILITY IN THE SYSTEM.....	21
FIGURE 7 CONSTRAINTS BETWEEN THE CONTEXT AND THE SYSTEM CONFIGURATION.....	22
FIGURE 8 GRAPH SHOWING THE DEPENDENCIES BETWEEN DIFFERENT VARIANTS, DASHED BOXES REPRESENT THE CONSTRAINTS (CONDITIONS) ASSOCIATED TO EACH VARIANT.....	24
FIGURE 9 GRAPHICAL ILLUSTRATION OF OUR AUTOMATIC TECHNIQUE FOR SYNTHESIZING DATA SATISFYING SEVERAL COVERAGE CRITERIA.....	26
FIGURE 10 COVERAGE AND TEST SUITE SIZES RESULTS FOR THE “ROBOT” CASE STUDY.....	27
FIGURE 11 COVERAGE AND TEST SUITE SIZES RESULTS FOR THE CRM CASE STUDY.....	28

1 Introduction

DiVA is an acronym for “Dynamic variability in complex, adaptive systems”. A dynamic system holds in the DiVA project (potentially many) variation points which can be selected according to constraints. Moreover, a DiVA system adapts to the context it lives in. Therefore, context’s Quality of Service properties and QoS rules affect adaptation at runtime, too. As such, many possibly valid configurations would have to be taken in account during naive brute force reasoning which is characterised by complexity explosion over a growing system size.

The aim of DiVA work package 4 (**WP4**) is to define and implement advanced techniques for efficient reasoning on co-dependent, co-existing configurations that populate a dynamic system and to define and implement techniques for validation of configuration properties which cannot be ensured at the model level [1, p. 50]. In reasoning, configuration complexity must be confronted. WP4 (Adaptation Reasoning and Validation) is a crosscutting work package which assembles reasoning and validation for the three technical packages WP1 (Requirements Analysis), WP2 (Adaptation Model & Model Transformation) and WP3 (Adaptation, Models@Runtime). This is visualised in Figure 1. WP4 performs requirements level reasoning for WP1, design-time reasoning for WP2 and runtime reasoning together with runtime validation for the WP3 tasks.

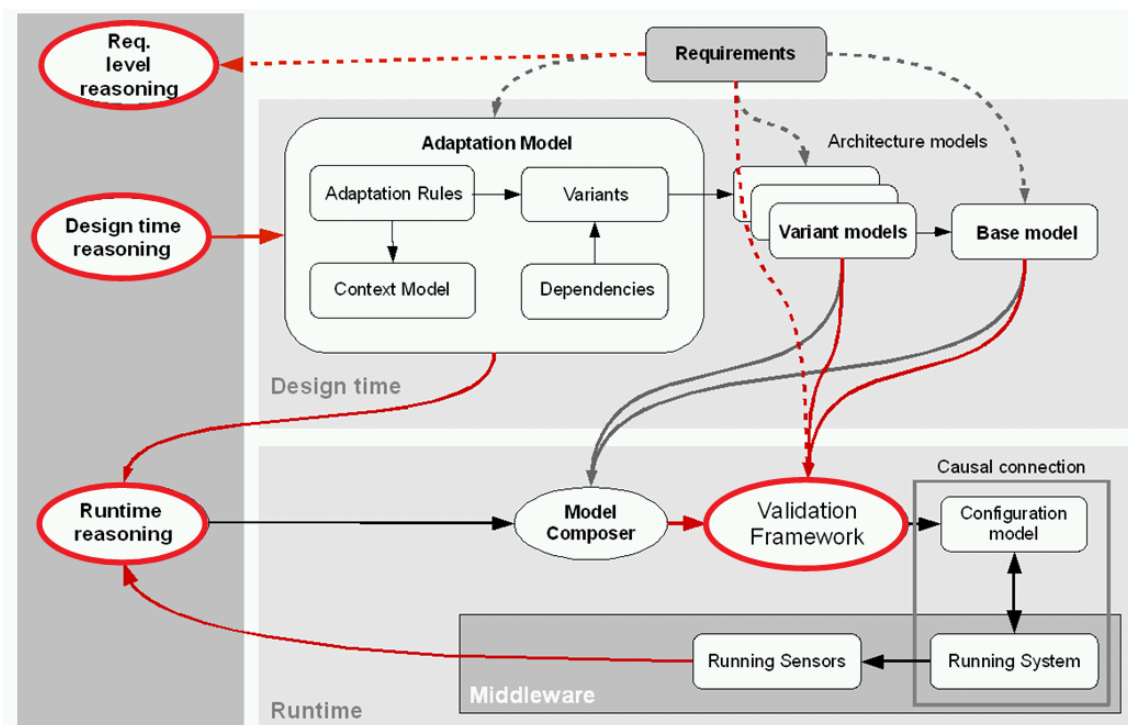


Figure 1 WP4 as a crosscutting work package.

Task 4.2 becomes manifested in the Adaptation Reasoning Framework (**ARF**) and is described in this document. Task 4.2 includes [1, p.51]:

- Definition of a model for adaptation aspects in order to enhance them with assume/guarantee clauses.



- Development of a reasoning framework for analysing the properties of the variability dimensions and select best-fit or at least possible configuration(s).
- This conceptual framework is in WP5 implemented and integrated within the Diva Tool suite.

This technical report D4.2 is the follow-up to D4.1 [2] which holds background information about reasoning principles, algorithms and frameworks. It accompanies the ARF prototype implementation integrated in DiVA Studio as part of WP5. Its purpose is to motivate and explain the ARF interface and to describe its current implementation briefly.

Chapter 2 describes premises such as modelling specific requirements and also reasoning specific requirements, which are covered by the ARF. The concept of the framework is introduced in chapter 3 by discussing its structure and showing its usage with the help of use cases. Since at design-time no real-life system environments or contexts are available reasoning and validation is performed on simulated contexts. Chapter 4 focuses on a set of strategies for such automatic generation of contexts meant to validate the behaviour of the DiVA adaptation models at design-time.

Deducting from the ARF concept chapter 5 explains the design of the ARF and its interfaces. It also describes walkthroughs which refer to the defined use cases of chapter 3. The current implementation of the ARF with its mapping of the DiVA adaptation model to an internal ARF model is explained in chapter 6. It also includes a description of reasoning problems which can currently be solved by the ARF and of reasoners which are currently available in the ARF. First results of the ARF implementation and reasoner tests are shown in chapter 7.



2 Premises

This chapter describes basic requirements defined by the project partners for the design and implementation of the ARF. This can be seen as a supplement to the requirements described in D4.1 [3].

2.1 Reasoning-specific Requirements

The following requirements specific for reasoning were agreed on with WP2 and WP3. Priorities express urgency in the development cycle, e.g. 'high' priority means that this requirement must be implemented as fast as possible.

R-01: Accuracy of configuration			
Priority	High	Status	new
Category	Reasoning		
Date Submitted	26.01.09	Last Update	
Reporter	BM, FF	Assigned To	
Stakeholders	INRIA, IRISA		
Description	Find best configuration from context. Determine the most (or the least well) adapted variants, depending on the environment or sensor information at runtime. These variants should respect the constraints defined by WP2 (dependencies, exclusions, etc.). Remark: A framework at INRIA uses Fuzzy Logic to automatically map quantitative values from the sensors to qualitative values (e.g., high, medium, low). For the moment, we can use a hard threshold to map the values (e.g., >75 = high).		

R-02: Stability of adaptation			
Priority	Medium	Status	new
Category	Reasoning		
Date Submitted	26.01.09	Last Update	
Reporter	FF, BM	Assigned To	



Stakeholders	INRIA, IRISA
Description	<p>The reasoning framework should not adapt the system every few seconds even if the environment is changing rapidly. For example if the system is on the limit of reachability of a WIFI network, the sensors will oscillate between "there is a WIFI signal" and "no signal available" but we do not want the system to try to connect each time.</p> <p>If the system keeps on adapting due to minor changes in the environment, this may have a negative impact on the QoS, because dynamic adaptation has a cost.</p> <p>Remark: The generic framework WildCAT (http://wildcat.ow2.org/) allows writing queries like "What is the average value of the WIFI signal during the last 30 seconds". This is a first possible way to introduce some stability.</p>

R-03: Efficiency			
Priority	Medium	Status	new
Category	Reasoning		
Date Submitted	26.01.09	Last Update	
Reporter	FF, BM	Assigned To	
Stakeholders	INRIA, IRISA		
Description	<p>The ARF should react very quickly for critical adaptations but may use more time for non critical adaptation.</p> <p>Remark: We distinguish between two modes: reflection (when we have time) and reflex (when we should adapt rapidly). The reflex mode may simply consist in choosing a reasonable pre-constructed, pre-validated fall-back configuration.</p> <p>Remark: The ARF should also make trade-offs before adapting. Perhaps it is not always optimal to adapt to the best possible configuration if the currently running configuration is good-enough.</p>		

R-04: Fallback configurations	

Priority	Medium	Status	new
Category	Reasoning		
Date Submitted	26.01.09	Last Update	
Reporter	BM	Assigned To	
Stakeholders	INRIA		
Description	Fallback configurations are needed for efficiency.		

2.2 Complexity Reduction

To face and avoid complexity explosion in reasoning is a major concern of the DiVA project. Complexity explosion arises in validating configurations depending on feature models via the exponential increasing number of combinations of features. Details can be found in D4.1 [2, p. 15]. If not appropriately handled, it could cause unacceptable calculation costs and therefore result in much too long reaction time spans for bigger models while computing which of a large number of possible configurations is valid, practical or even optimal.

Results of D4.1 to reach complexity reduction are [2, p. 58ff]:

In the system design phase:

- Reduction of feature value ranges to
 - Binary selectable values, e.g. switch on/ switch off
 - A limited range of discrete, selectable values e.g 100 KB of memory
- Extensive application of constraints to features such as
 - Multiplicities/ bounds to lower an upper number that a feature is allowed to appear in the final configuration
 - Dependencies to tell on which other feature a feature depends on
 - Available to tell in which context a feature may be applied
 - Required to tell in which context a feature must be used

Additional at runtime due to tighter time and resource constraints:

- Use of “Any Time” reasoning approaches. That is, reasoning that delivers results quickly and can be stopped at any time. The result is not always the optimal configuration but the best approximation known at this time.
- Use of pre-calculated configurations (user either directly or as starting point for searching a suitable configuration)
- Use of heuristic approaches to quickly search the configuration space



These issues shall be addressed by using appropriate reasoning approaches. Depending on the size of the system and the available resources different engines and/or approaches are suitable. That requires the ARF to support flexible exchange/selection of reasoners depending on the context. It also requires a certain degree of flexibility in the services provided by the ARF reasoners. Some of the reasoners will not be able to support all use case of the ARF.

3 Reasoning and Validation Concept

The DiVA methodology workflow as shown in Figure 2 is divided into three hierarchical level

1. requirements level
2. design-time
3. runtime

The runtime level is based on the design-time level and the design-time level is based on the requirements level. But as shown by the workflow reasoning and validation is a crosscutting action, whereby validation is applied to design-time and runtime level.

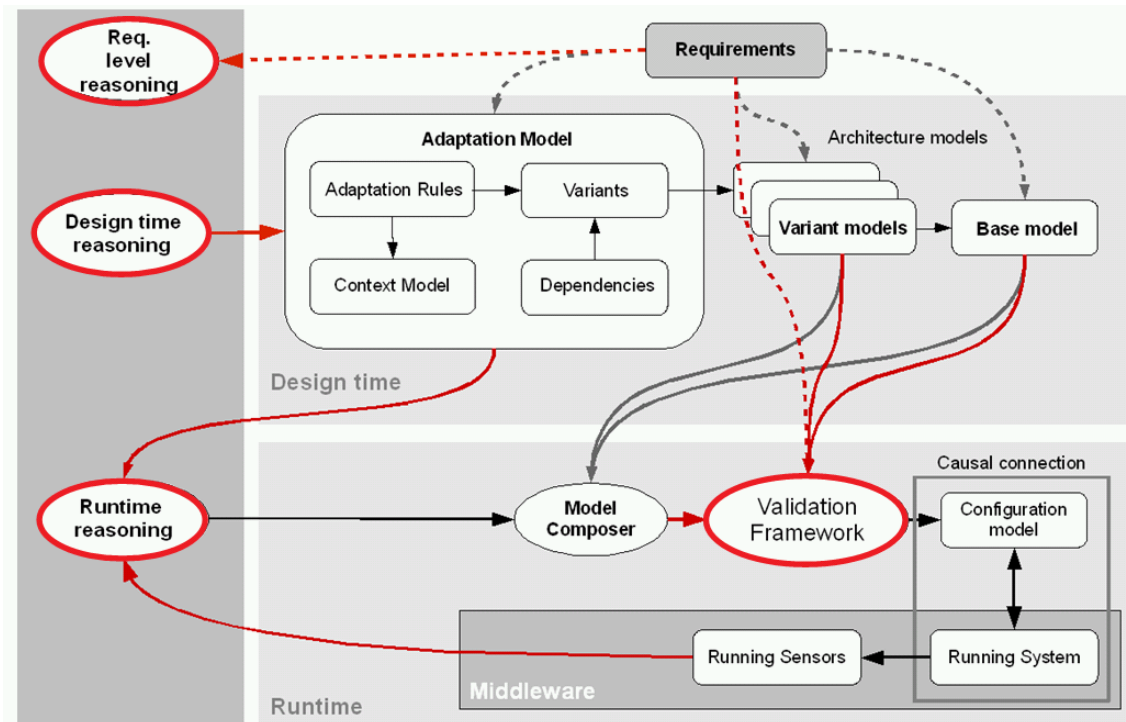


Figure 2 DiVA methodology workflow

Thus the ARF has to provide reasoning and validation to requirements level, design-time level and runtime level. The next section 3.1 discusses the structure of the ARF regarding the three levels and the appropriate work packages and decides how the structure of the ARF will be. To illustrate the usage of the ARF within DiVA section 3.2 describes for all three levels use cases, one for each level.

3.1 ARF Structure

The ARF supports the requirements level, the design-time level and the runtime level. The question is whether the ARF has to provide reasoning and validation as

- requirements level reasoning (and validation) for WP1
- design-time reasoning and validation for WP2
- runtime reasoning and validation for WP3

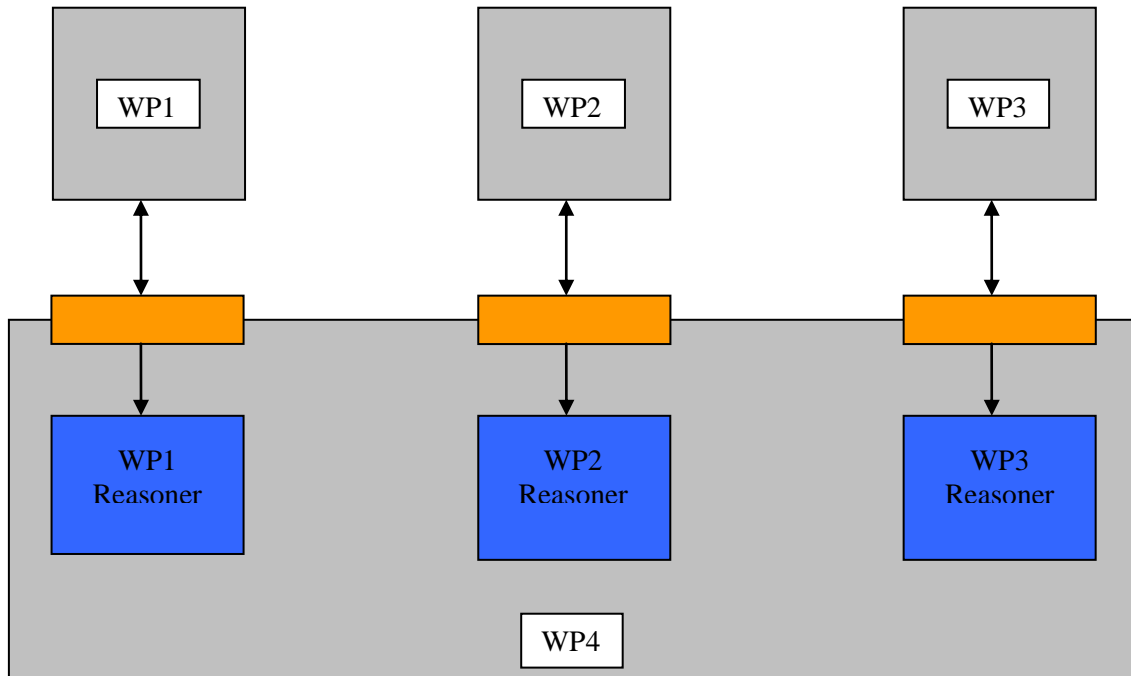


Figure 3 Three level approach

Figure 3 shows a three level approach of a framework, which reflects the dependence on the work packages in its structure. Each work package has its own entry point to the framework and also its own logical reasoner interface. But this implies a large number of reasoners, because one reasoner has to be compatible with the three interfaces, if it is used for all work packages.

Therefore the reasoning problems are tried to map to the same technical base to reduce the number of reasoner. Reasoning problems at requirements level and design-time are similar. Both levels are interested in the existence of configurations or in solving all configurations. Reasoning problems at runtime are subject to resource limitations and refers to getting or optimising one configuration or validate a configuration. Figure 4 points to the ARF structure which captures mapping of reasoning and validation problems to the same technical base. Each work package has its own mapping interface, which maps the level-dependent input models to internal ARF models.

A reasoner reasons on these internal ARF models and is now available for all levels. As mentioned before the ARF reasons on different problem categories. Accordingly a reasoner provides different interfaces

- Solver
 - provides for a given input model access to all (!) configurations
 - tells if there is any configuration
 - for correctness check of input model
- Validator
 - validates a given configuration with respect to
 - being up-to-date regarding the actual context
 - being up-to-date regarding a new context

- Optimiser
 - provides for a given input model access to some (representative) configurations
 - produces configurations optimized with respect to
 - requirements such as a scoring or ranking algorithm
 - Aspect Order Issues

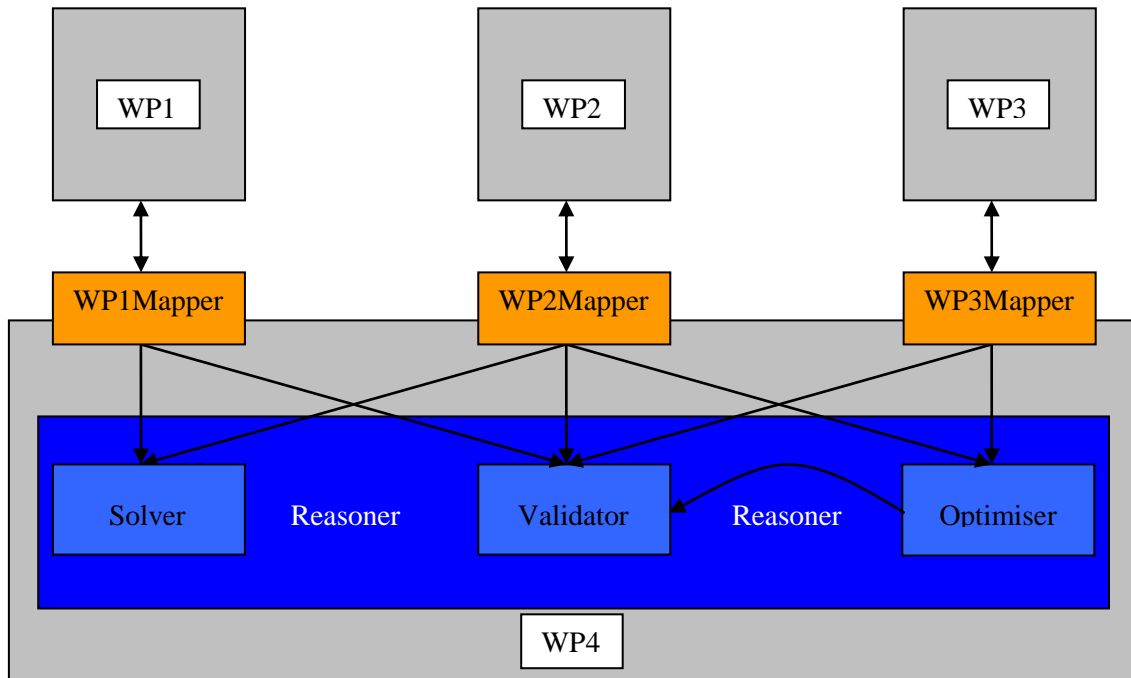


Figure 4 ARF structure

3.2 Reasoning and Validation Use Cases

In order to derive a suitable design for the ARF use cases are defined. These use cases are also the base for deriving the ARF interface. Each use case shows for a different scenario how the ARF can be used and what functionalities the ARF has to provide for requirements level, design-time and runtime reasoning.

3.2.1 Requirements Use Case

In comparison to the two other levels at this level reasoning is more or less an informative task. As described in [4] the output of the requirements engineering is a feature model, which is generated after processing the textual requirements documents. The most obvious request to the ARF is the question, if there is at least one valid configuration which can be found for the generated feature model. At requirements level it is of minor interest how the ARF answers this question (i.e. which reasoner is used to answer this question and to a certain extent also how much resources are used to answer this question). Therefore no information about the ARF is necessary e.g. about available reasoners.

Assumption:

- no information about the ARF

Input:

- Data: feature model (mapped on DiVA meta model)
- Question: Is there a valid configuration?

Output:

- Data: Boolean value

At first the generated feature model is set to the ARF for initial processing. The next step is to retrieve the answer to the question (Is there any valid configuration?) by waiting for the ARF. Internally the ARF selects a suitable reasoner, which is able to solve the question, before answering it. If multiple reasoners are available, it decides to use one or more of the reasoners to answer the question. There will be no access to a specific configuration, since depending on the reasoning approach generating an individual configuration is not necessary to answer this question. True will indicate existence of a configuration, false the non-existence.

3.2.2 *Design-time Use Case*

At design-time the DiVA adaptation model is modelled based on the applications requirements [5]. At this level reasoning and validation become more complex. It includes additional rules since more information about the systems internals (the design) are linked to requirements level variability information as expressed in the feature model. This results in increasing size, complexity and expressiveness of the input model.

In difference to reasoning and validation at runtime the ARF at design-time reasons on simulated context instances of the given DiVA adaptation model. Generating such simulated context instances is introduced in the next chapter 4. Also there are not such hard constraints for resource limitations. It is not mandatory to answer questions within hard real-time constraints, but depending on the design task to be solved a quick response from the ARF might be desirable.

One of the reasoning tasks at this level is the calculation of all valid configurations for a given context (or a set of contexts), based on a set of constraints and dependencies. Another question is to get an evaluation (scoring) of the configurations for a context.

At this point the ARF supports user defined specification of the reasoner to be used for answering the question. This was a request made in order to ensure that during design-time and runtime the same reasoner is being used. However, if no reasoner is specified, the ARF automatically selects one or more reasoners to be used for answering the question.

Assumption:

- no information about the ARF

Input:

- Data: DiVA adaptation model
- Data: context instance with simulated values
- Question: get all valid configurations

Output:

- Data: all valid configurations

- Log: protocol and statistics

In this use case the ARF is instructed to use a specific reasoner, by asking the ARF for available reasoners which can answer the question (Get all valid configurations) for the given DiVA model. The model and the context are set to the ARF before asking the question. The ARF returns for instance two suitable reasoners, a Binary Decision Diagram (BDD) reasoner and an “Any Time” (AT) reasoner. Characteristics of the reasoners are described by attributes. The BDD reasoner is a slow reasoner comparable to the other reasoner but its reasoning is optimal. It provides also an optimisation strategy which scores the configurations. The AT reasoner is a fast reasoner, but its reasoning is not optimal and there is also no optimization strategy. The BDD reasoner is selected for instance because of its optimal reasoning capabilities.

The BDD reasoner is selected with a call to the ARF to prevent automatic selection by the ARF. Now the ARF is asked for all valid configurations regarding the set DiVA adaptation model and its context instance(s). The BDD reasoner tries to solve all configurations and afterwards also validates them. Extra validation might be necessary since not always all rules for checking the validity of a configuration can be included in the solving process. Because the used BDD reasoner provides a scoring optimisation strategy all valid configuration are also scored before being returned by the ARF.

Furthermore the ARF returns some logging information like a protocol and statistics. The protocol contains e.g. which reasoner was used and information about its memory usage or time consumption. The statistics contain textual information about reasoning and validation, warnings like inconsistencies in input and errors like runtime exceptions.

3.2.3 Runtime Use Case

At runtime the running system adapts to a new system configuration, if the environment or context of the running system changes [6]. The context is derived from the system’s environment and the adaptation is triggered e.g. by new sensor values (which changes the context) and has to be performed in a pre-defined time span. Therefore also reasoning and validation at runtime are subject to these hard constraints regarding time and memory usage. Because only one configuration is able to run on the system it is not important to get all configurations or the most optimal configuration. In general only one configuration is necessary as long as it is valid for the given context and can be found in desired time.

Since for the example use case it is known that the BDD reasoner is too slow, the AT reasoner is being used as reasoner. Therefore no information about the ARF is collected before. It reasons on a DiVA adaption model which was modelled at design-time and a context instance with actual values of system's environment.

Assumption:

- Reasoner: AT reasoner
- Parameter: Time=FAST, Reasoning=NON-OPTIMAL

Input:

- Data: DiVA adaptation model
- Data: context instance with actual values
- Question: get one valid configuration
- Parameter: TIME=60sec

Output:

- Data: one configuration

First the AT reasoner is set to the ARF, that is configured to provide one non-optimal configuration within 60 seconds. Afterwards the DiVA adaption model, its context instance with actual values and the mentioned parameterised question (Get one valid configuration) are also set to the ARF. Now the ARF is asked for one configuration and the AT reasoner tries to find the first configuration it can within 60 seconds. After these 60 seconds the ARF returns one configuration or none if the reasoner was not able to find any configuration according to the set parameters.

In this use case the validation is a second separate step which checks if the returned configuration can be used for an adaptation of the system represented by the DiVA adaptation model and the context instance. If a configuration is returned by the AT reasoner, it is validated, because in this case the AT reasoner produces configurations without considering constraints. Because no information about validation is known, the ARF selects an appropriate reasoner.

Assumption:

- no information about ARF

Input:

- Data: DiVA adaptation model
- Data: context instance with actual values
- Data: configuration
- Question: is configuration valid

Output:

- Data: Boolean value

First of all again the DiVA adaptation model, the context instance and the solved configuration are set to the ARF. Because no information about any reasoner is known no reasoner is set. To be able to validate the configuration the ARF automatically selects an appropriate reasoner. This reasoner, e.g. the BDD reasoner, validates the configuration with the help of the set model and context instance. The result of the reasoner is returned by the ARF. If the configuration was valid and can be used for the adaptation of the system the ARF returns true and otherwise false.

4 Design-time Validation through DiVA Context Generation

An adaptive system or dynamic product-line is a system, which contains variability and needs to dynamically switch between variants in order to adapt to a changing environment. Modelling adaptive systems thus involves variability modelling, environment modelling and expressing adaptation logics to relate the environment models and the variability model. The purpose of an adaptation logic is thus to specify which variant to use depending on the context. Different techniques have been proposed in the literature to capture and express adaptation logics. Most of these techniques can be categorized under two families of approaches [9; 10; 11; 12; 13]. The first one is the most commonly used and is based on event-guard-action type of rules [9; 10; 11]. In these approaches the context and the configurations are related by a set of rules, which express how the evolution of the context should affect the running configuration of the application. The second family of approaches relies on the optimization of some utility functions [12; 13] associated to the system. In these approaches, changes in the environment trigger an optimization process that evaluates possible alternative configurations and adapt the system to maximize the utility of the running configurations (*e.g.*, trade-off between bandwidth and quality of image on portable video devices). Both these families of approach have their strengths and limitations in terms of usability, efficiency and scalability. In this section we use a hybrid approach, which combines both hard-constraints and optimization rules. The main strengths of this approach are scalability and design-time simulation capability.

In this section we propose a set of strategies for the automatic generation of test vectors to validate the behaviour captured in the adaptation model. These strategies are meant to sample the space of possible contexts to which the adaptation model is supposed to react. This testing approach is intended for design-time validation of the model.

4.1 Illustrative example

To illustrate the approach we use a simple example of a map-building robot. The system is a semi- autonomous exploration robot, which builds a map of an unknown environment when in motion. The robot is connected to a central system, which collects the topographic data and can give directions to the robot. The robot has 3 main modes: idle, going to a specific location or exploring autonomously. It is equipped with three different sensors which can be used alternatively for routing and for drawing the map: i) Camera, which provides the most detailed map, ii) Infrared sensors, which can work without light sources and use limited resources, iii) Ultrasonic sensors, which consumes limited resources while providing good routing capabilities.

While being drawn, the map is either stored locally in the robot's memory with periodic transmissions or directly streamed to a server. To allow for transmissions as well as for receiving commands the robot is equipped with Bluetooth and GPRS networking capabilities. Additionally, the robot can employ different routing and mapping strategies. The local routing strategy uses the sensors to navigate, the map-based strategy uses an exiting map of the area, while external routing involves interactions with a central computer or an operator. Secondly, the robot can either use a simple or detailed map drawing strategy. Depending on its environment, *i.e.* on its mode, on the terrain conditions or on the resources available, the robot has to dynamically adapt in order to optimize the quality of the maps it builds while using appropriate sensors and algorithms.

4.1.1 Modelling runtime variability and adaptation logic

Based on the above description, the first thing that needs to be modeled is the variability in the system and in its environment.

Figure 5 presents the context of the robot as a feature diagram. The context is composed of three things: a given mode for the robot, a set of properties of the surrounding environment and a set of internal

properties of the robot. In the surrounding environment, the availability of Bluetooth signal and the lighting conditions can vary. Inside the robot, the resources in term of power and memory can vary.

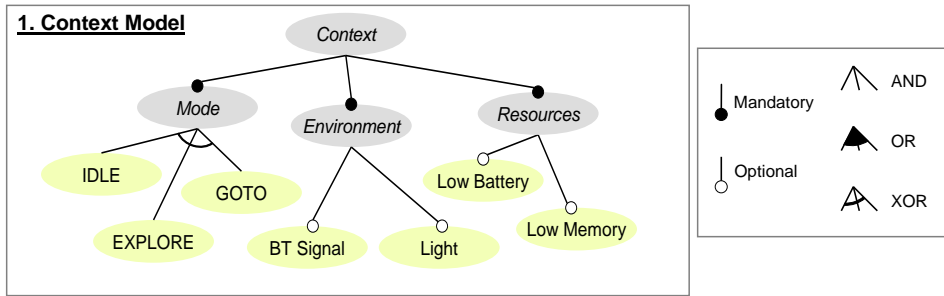


Figure 5 Model of the context of the system

Figure 6 presents the variability in the system as a feature diagram. Each configuration of the robot is composed of two mandatory features (Transmission and Map Strategy) and three optional ones (Routing, Network and Sensors). For each of these variations point a set of alternatives are available. In addition to the constraints defined using the feature diagram notations, cross-tree constraints can be used to reflect dependency constraints between features. For the robot system 3 dependency constraints are presented in Figure 6. The first one expresses that one of the Network feature needs to be present in order to use the External routing strategy.

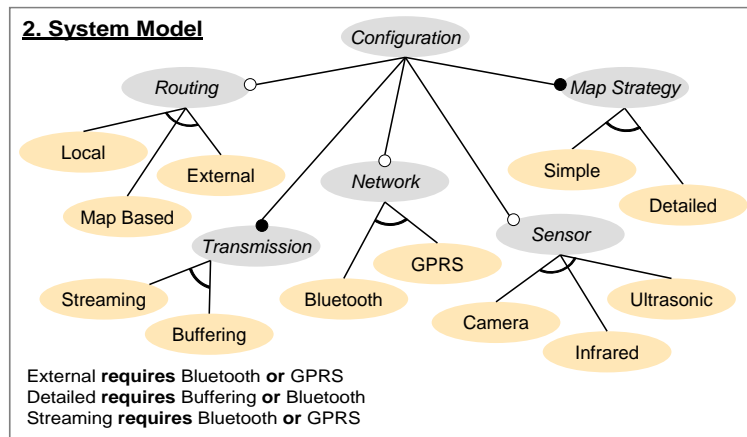


Figure 6 Model of the variability in the system

Once both the variability in the environment and in the system has been defined separately, the adaptation logic can be defined. In the approach we use this is done in two steps. First a set of hard-constraints are defined between the context model and the variability model and second a set of properties and optimization rules are defined. Figure 7 presents the adaptation constraint for the robot. As an example, the second constraint specifies that the “Map based” routing strategy is only available if the robot is in GOTO mode and is not low on memory.

3. Adaptation constraints

Local **requires** GOTO
 Map Based **requires** GOTO **and not** Low Memory
 External **requires** GOTO
 Bluetooth **requires** BT Signal
 GPRS **required not** BT Signal
 Camera **requires** Light **and not** IDLE
 Infrared **requires not** IDLE
 Ultrasonic **requires not** IDLE
 Low Memory **requires** Streaming
 Buffering **requires not** Low Memory

Figure 7 Constraints between the context and the system configuration

The adaptation constraints reduce the number of possible configuration for each context but are not sufficient to pin-point the best configuration for every possible context. Different rule-based or optimization-optimization techniques can be applied to complete the adaptation model. The proposed approach considers this part of the adaptation model as a black-box. The only requirement is the ability to simulate the adaptation model, i.e. to compute which configuration would be chosen for a given context. The approach we used for the experiments is based on the definition of a set of properties, the impact on features on these properties and a set of optimization rules which depend on the context. More details about this technique, demos and tools can be found on the web DiVA project web site <http://www.ictdiva.eu/>¹.

4.2 Context data selection criteria

In this section we describe the different selection criteria we adapted for the particular purpose of testing an adaptation logic model. Notice that we revisit three well-established data selection criteria (Simple coverage [14] and Pairwise [15]), in order to fit them to our need. Additionally, we propose five new criteria, four of them combining the knowledge of the previous. Our intuition here is to combine the abilities of each criterion to target specific faults in order to globally improve the pertinence of the data set.

4.2.1 Simple coverage criterion

The simple coverage (SC) criterion is a black-box testing criterion [14], which requires the existence of all the possible values of a procedure input's domain. In other words, a data set satisfying this criterion is one that includes each possible value for a procedure input parameters. Since the input domain for the adaptation logic is the context space, we revisited this criterion as follows:

SC criterion: a data set that satisfying this criterion is one that includes each possible value that a context variable can adopt.

Mode	Light	Bluetooth	Low memory	Low battery
<i>IDLE</i>	<i>TRUE</i>	<i>TRUE</i>	<i>TRUE</i>	<i>TRUE</i>
<i>EXPLORE</i>	<i>FALSE</i>	TRUE	<i>FALSE</i>	TRUE
<i>GOTO</i>	TRUE	<i>FALSE</i>	TRUE	<i>FALSE</i>

Table 1 Data set satisfying the simple coverage criterion. Each context variable value appears at least once in the data set.

¹ Tools will be publicly available during the summer. For evaluation purposes a demo and a version of the tools are available at http://www.fleurey.com/diva_preview/.

Table 1 illustrates a data set for our case study, which satisfies the SC criterion. Grey cells indicate the values that actually satisfy the SC criterion. Notice that this is the smallest data set satisfying this criterion.

4.2.2 Pairwise coverage criterion

In order to deal with the combinatorial explosion of combinations of values that have to be tested, combinatorial integration testing (CIT) proposes to select a subset of all combinations while still guaranteeing a certain level of coverage [15]. It is based on the observation that most of the faults are triggered by interactions between a small number of variables [16]. Pairwise testing (PW) or 2-ways coverage criterion is a special case of CIT that samples the set of all combinations in such a way that all possible pairs of variable values are included in the set of test data.

PW criterion: let $C=\{c_1,..c_n\}$ be the set of context variables, let $V_i=\{v_{i1},..,v_{im}\}$ be the set of m possible values for variable c_i , then a set of test context instances TCI satisfies PW if for every pair of context variable, all possible combination of values are present in at least one test context instance. More precisely:

$$TCI \text{ satisfies PW} \Leftrightarrow \forall (c_i, c_j) \in C, \forall v_x \in V_i \wedge v_y \in V_j, \exists tci \in TCI \mid v_x, v_y \in tci$$

Context variable value				
Mode	Light	Bluetooth	Low memory	Low battery
GOTO	FALSE	FALSE	FALSE	FALSE
GOTO	TRUE	TRUE	TRUE	TRUE
EXPLORE	FALSE	TRUE	FALSE	TRUE
EXPLORE	TRUE	FALSE	FALSE	TRUE
EXPLORE	TRUE	TRUE	TRUE	FALSE
IDLE	TRUE	TRUE	TRUE	TRUE
IDLE	FALSE	FALSE	TRUE	FALSE
IDLE	FALSE	FALSE	FALSE	FALSE

Table 2 Data set satisfying the pairwise coverage criterion. It contains each pair or 2-way array of context variable values.

Table 2 illustrates a data set for our case study, which satisfies the PW criterion. Notice that, since the PW requires the appearance of each pair of variable values, it subsumes the SC criterion. That is, this data set also satisfies the SC criterion.

4.2.3 Dependency coverage criterion

The adaption model declares dependencies between variants in the system. For example, in order to use the streaming variant for data transmission, it is necessary to have a connection to a network, either through GPRS or Bluetooth. Some variants also impose some constraints on context variables. So, for a variant which existence depends on other variants, it is possible to derive a constraint on the context that considers the local constraint and the constraint of dependent variants. For example, if streaming is chosen with GPRS, the derived constraint on the context is: $\neg \text{Bluetooth} \wedge \text{LowMem}$. All derived constraints for the example are given

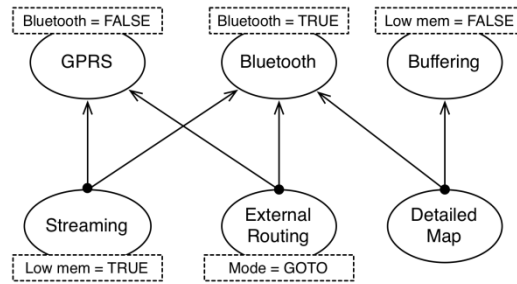


Figure 8 Graph showing the dependencies between different variants, dashed boxes represent the constraints (conditions) associated to each variant.

Figure 8 presents a graph illustrating the dependencies between the system variants (extracted from the *dependency* column). Nodes represent variants; arrows represent directional dependency between variants, and the black dot represents an exclusive option. We construct the dependent variants’ decisions through the conditions (constraints) attached to each variant. More precisely, we joint the *dependent* variant’s conditions into a compound decision, which represents the decision related to the dependent variants. Table 3 presents the constraints derived from Figure 8.

Dependency	Decision (constraint)	#
Streaming and GPRS	LowMem = TRUE && BTSig = FALSE	1
Streaming and Bluetooth	LowMem = TRUE && BTSig = TRUE	2
External routing and GPRS	Mode = GOTO && BTSig = FALSE	3
External routing and Bluetooth	Mode = GOTO && BTSig = TRUE	4
Detailed Map and Bluetooth	BTSig = TRUE	5
Detailed Map and Buffering	LowMem = FALSE	6

Table 3 Decisions derived from the dependencies between variants. We derive each decision from the conjunction (&&) between the dependent variants conditions (constraints)

We introduce the dependency coverage (DEP) criterion to verify the outcome of the constraints related to dependent variants.

DEP criterion: *a data set satisfies DEP if it includes enough values to verify every condition that can change the outcome of a constraint derived from dependent variants.*

Notice that some data sets satisfying the DEP criterion ignore some context variable values. For example, the context variable *light* never appears in the conditions in Table 3. For this reason, we demand from these data sets to satisfy the SC criterion.

Context variable value					Condition	
Mode	Light	Bluetooth	Low memory	Low battery	T	F
GOTO	TRUE	TRUE	TRUE	TRUE	2, 4, 5	1, 3, 6
GOTO	TRUE	FALSE	TRUE	TRUE	1, 3	2, 4, 5, 6
GOTO	FALSE	TRUE	FALSE	FALSE	4, 5, 6	1, 2, 3,
GOTO	TRUE	FALSE	FALSE	TRUE	3, 6	1, 2, 4, 5

EXPLORE	FALSE	TRUE	TRUE	FALSE	5	3, 4, 6
EXPLORE	FALSE	FALSE	TRUE	FALSE	1	2, 3, 4, 5, 6
IDLE	FALSE	TRUE	FALSE	TRUE	5, 6	1, 2, 3, 4
IDLE	TRUE	FALSE	FALSE	FALSE	6	1, 2, 3, 4, 5

Table 4 Data set satisfying the dependency coverage criterion. It contains enough values to verify each condition in Table 3.

Table 4 illustrates a data set for our case study, which satisfies the DEP criterion. It reads similarly to Table 3. Grey cells indicate the values satisfying the SC criterion. Notice that it forces the data set to contain the values *EXPLORE*, and *IDLE* for the variable *mode*, as well as both values *TRUE* and *FALSE* for the *light* context variable.

4.2.4 Compound coverage criterion

We have composed the presented criteria into a set of compound criteria. These criteria mix the conditions that each composed criteria established, then demanding all the criteria to be satisfied. The hypothesis backing this composition is that data sets satisfying several criteria at the same time are more likely to find faults without increasing dramatically the set's size. We seek the best compromise between the data set's size and the coverage of the adaptation logic model, i.e. system variants, adaptation constraints, property values.

DEP and PW: This criterion requires a data set containing the values satisfying the DEP criterion and the PW criterion. Notice that this criterion subsumes the DEP, PW, and SC criteria.

4.3 Experiments

In the previous section we presented a set of coverage criteria for adaptation logic models' testing data. These criteria target different elements of the adaptation logic model in order to ensure an adequate exposure of faults.

Our aim is to find the best compromise between data set's size and the coverage of the adaptation logic elements i.e. system variants, adaptation constraints, property values. On behalf of this purpose, for each criterion we have generated 30 data sets that we used to simulate the adaptation logic behaviour and evaluate the coverage. In this section we present an empirical study that compares the criteria by their adaptation logic model coverage. Paragraph 4.3.1 describes the technique we used to generate the data. Paragraph 4.3.2 introduces the protocol we used to evaluate the adaptation logic coverage. Paragraph 4.3.3 presents our analysis on the experimental results, and paragraph 4.3.4 presents the threats to validity.

4.3.1 Test data generation

In order to evaluate the effectiveness of the different data sets, we have generated 30 data sets satisfying each criterion.

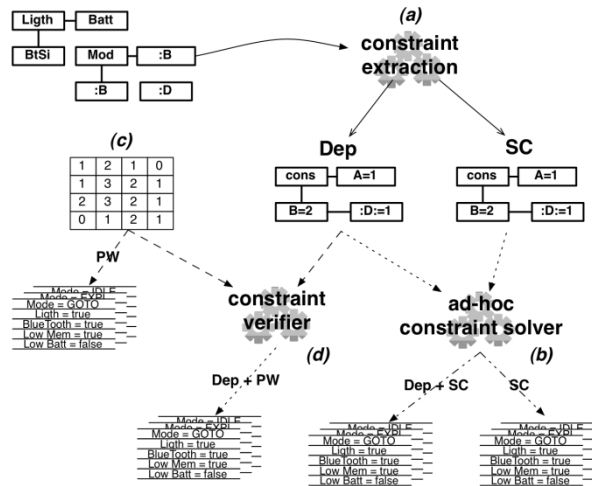


Figure 9 Graphical illustration of our automatic technique for synthesizing data satisfying several coverage criteria.

We automated the generation of the data sets as a constraint satisfaction problem. Figure 9 illustrates our generation technique. From the adaptation logic model, we derive a set of models describing the constraints corresponding to the different criteria (a). Then, we use an ad-hoc constraint solver to generate a solution satisfying the constraints. For the composed criteria, the constraint solver joints and solves the compound criteria’s constraints (b). This generates a data set as smallest as possible, which satisfies the criteria. A special case concerns the generation of the pairwise; we use an existing solution² to generate it, to then transform it into context instances (c). We generate data sets satisfying the criteria that compound the PW criterion, by using a constraint verifier, to enforce DEP and create elements that satisfy the constraint non satisfied by the pairwise (d).

4.3.2 Test data evaluation criteria

In order to evaluate and compare the generated test suites the quality of the test suites needs to be evaluated. In practice the quality of a test suite is a trade-off between its ability to detect fault and its size. The size of the test suite is easy to measure and compare but the ability of a set of contexts to detect errors in the adaptation logic is not as easy to estimate. For our experiments we propose to estimate the fault detection capability of a test suite by measuring its coverage of the adaptation logic. The assumption is we make is that the fault detection capability of a test suite is correlated to the coverage of the model under test (here the adaptation logic). This is a typical assumption made by structural testing techniques.

The coverage measurement we used is based on the coverage of adaptation rules and alternative variants. The adaptation model is fully covered by a set of contexts if all rules are used at least once and each alternative variant of the system is used by at least one configuration. The coverage value of a set of contexts is computed as the percentage of rules and alternative variant which are covered. In order to have significant result, each test generation criteria has been used to generate 30 different test suites and coverage measurement have been carried on all 30 test suites.

4.3.3 Experimental results and analysis

This section presents the results we obtained on two case studies. The first case study is the robot example presented in section 4.1. The second example is part of an industrial customer

² Pairwise generator available at <http://www.mcdowella.demon.co.uk/allPairs.html>

relationship management (CRM) system. While the example of the robot is quite simple and only yields about 200 different configurations, the part of the CRM system used for the case study yields about 1500.

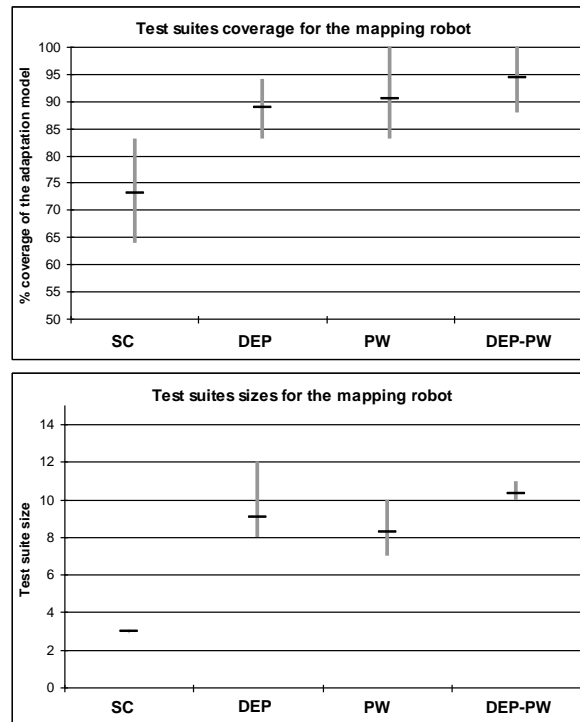


Figure 10 Coverage and test suite sizes results for the “robot” case study.

Figure 10 presents the results obtained for the robot case study. The first graphic presents the coverage of the test suites generated to satisfy the various criteria. The second graphic presents the sizes of the test suites. Each graphic displays the data coming from 30 experiences. The black line corresponds to the average values measured for these 3 experiences. The gray bar goes from the minimal to the maximal observed value. Overall the size of the test suites for the robot is between 3 and 12 contexts and allowed covering between 65% and 100% of the adaptation logic. The simple coverage criteria allow achieving an average coverage of around 74% with a quite important variability from one test suite to the other. This criteria has the benefit of requiring only very small test suites but does not seem to be good-enough for ensuring a sufficient coverage of the adaptation logic. The other three criteria seem to be able to achieve a much higher coverage but imply a significantly higher number of test contexts.

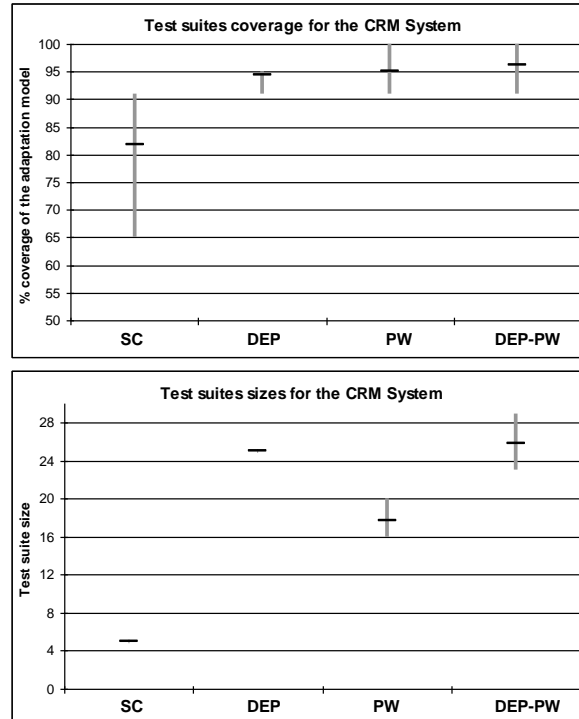


Figure 11 Coverage and test suite sizes results for the CRM case study.

Figure 11 presents the results for the CRM case study. This case study has been initially extracted from an industrial application in order to evaluate the adaptation modelling technique we used for our experiments. The system is assisting its users in their customer relationship activities. It serves different purposes and has to adapt depending on the current situation. The system supports for example connections through a desktop pc, a laptop, a smart-phone or even a basic phone using a voice interface. The part of the system used for this case study comports about 10 different context variables and can use about 1500 different configurations depending on the context. The results of this second case study are very similar to the results obtained with the robot example but on a different scale. The number of test contexts generated by the 4 proposed criteria is between 5 and 29. As on the robot example, the results for simple coverage are quite weak in terms of coverage while the other criteria provide good coverage but with significantly bigger test cases.

On both case studies, the results of simple coverage are weak in terms of the coverage of the adaptation logic. For the three other criteria, the test suites are significantly bigger and the coverage of the adaptation logic seem quite equivalent. In average the combination of “pairwise” (PW) and dependency coverage (DEP) seem to ensure a little bit more coverage but the difference is not statistically significant on the CRM case study. DEP and PW (and the combination of the two) seem to provide equivalent coverage. With respect to the sizes of the test suites, it seems that PW has a small advantage. In practice PW also has the advantage of considering the adaptation model as a complete black-box as opposed to DEP which extracts information from the adaptation model. This result is disappointing for the DEP criterion because despite the use of more information on the adaptation model it does not perform significantly better than PW.

4.3.4 Threats to validity

There exists no perfect data, or perfectly trustable analysis results, and this study is not an exception. For this reason we identify the construction, internal and external threats to validity of this study.



Inter threats lie on the validity of the selected case study, we only studied two adaptation logics, though one is a large industrial test case from a real life project. We ignore whether the constraints specified in these adaptation logic models apply to other scenarios. It is possible that other application context include more complex constraint, which could need more data to satisfy the different criteria.

Construction threats lie on the way we define the coverage, in the way we measure it, and the way in which we generate the data sets. It is possible coverage based on system variants, adaptation constraints, and property values will be not enough to tell that a data set is better than other, especially when dealing with fault detection. We measure the coverage through simulation. However, third parties of the simulator, such as the alloy solver could contain faults. These faults will leak to and degrade the simulation results. Furthermore, the generation of data set includes a third party pairwise generator. We are unaware of possible faults in it, and again, if it contains faults they could leak into the data set, for example making them larger than expected.

4.4 Related work

Some researchers have addresses the validation of dynamically adaptive system, though none particularly address the validation of the adaptation logic.

Zhang *et al.* [17] address the verification of dynamically adaptive systems through modular model checking. They model the adaptive system as finite state machine in which states represent different system variants. For each transition between systems variants, they model check, when possible only the parts of the system that have change product of an adaptation. In [18], they introduce a model-based development process for adaptive software that uses Petri-nets and Petri-net based model checking tools to model and check the system behaviour and properties. Biyani and Kulkarni [19] use predicate detection for testing adaptive systems during adaptation. They extend existing algorithms based of *global predicate evaluation* [20] for testing distributed systems to the system during adaptation. In [21] they introduce an approach using proof-lattice to verify that all possible adaptation paths do not violate global constraints. Allen *et al* [22] used the Wright ADL to integrate the specifications of both architectural and behavioural aspects of dynamically reconfigurable systems. These specifications can be then translated and solved using verification facilities such as model checking. Kramer and Merge [23] use property automata and labelled transition systems to specify and verify adaptive program's properties. The main difference between these verification approaches and ours is the focus of attention. We are interested in verifying through testing the adaptation driver, and not the adaptation process itself. Furthermore, these approaches require computing the entire system configurations and the transitions between them, however sometimes this is not possible.

Lu *et al.* [24; 25] study the test of pervasive context-aware software. They assume context awareness as a series of if-then cases, and starting from that point they formalize the notions of context aware data flow entities, i.e. entities that manipulate data coming from the context. By using this formalization they propose a family of test adequacy criteria that measure the quality of test sets with respect to the context variability. The main difference between this approach and ours is that we revisit existing criteria that explore the environment with criteria that exploit knowledge of the adaptation logic. Whereas we focus principally of exploration and coverage of the context space, this approach focuses on exploiting knowledge of the *context awareness* formalization to qualify data sets. Furthermore, the DEP criterion we introduced in paragraph 4.2.3 is similar to the analysis performed in this approach.

4.5 Conclusions

Modelling the adaptation logic is an important first step for the development of dynamic adaptive systems. We have proposed 4 different strategies for selecting test contexts for the



validation of the adaptation logic of a system. The proposed approach is applied at design-time and relies on the simulation capabilities of adaptation modelling environments. The four proposed strategies go from the simple coverage of individual context element to the combination of relevant context elements. A prototype tool has been implemented to automatically generate test data satisfying the proposed criteria. The approach was applied to two different case studies in order to evaluate and compare the criteria. Results show that the simple independent coverage of the context element is rather insufficient to ensure a good coverage of the adaptation logics. The more elaborate criteria allow ensuring a significantly better coverage but also require much more test contexts. An interesting result is that the black-box criterion “pairwise” was able to perform comparably to the criterion base on dependency analysis. In practice, the initial results presented here suggest the use of “pairwise” as the best alternative to efficiently validate an adaptation model.

5 ARF Design

The requirements level, design-time level and runtime level of the DiVA methodology workflow use the ARF for their own purposes although the data model for all levels is identical since the DiVA meta model is used. But from the perspective of the ARF the main differences are resource constraints (nearly unlimited resources at design-time vs. limited resource at runtime) and questions to be answered by the ARF (checking of existing configuration at requirements level vs. getting all configurations at design-time level).

The use cases of section 3.2 are used as base for designing the initial interface of the ARF which is introduced in form of an informal description in section 5.1. The following section 5.2 defines and formalises the technical interface, which is the base for the implementation of the ARF and its extensions. Finally section 5.3 shows how each use case is converted in a reasoning and validation walk-through which uses the defined formal interface.

5.1 Preliminary Considerations

Coming from the described use cases this section defines the interfaces in a semi-formal textual manner. First of all the use cases are analysed regarding one harmonised common interface of the ARF. Therefore input data as well as output data are collected of them.

As input, the ARF gets from a client module

- [optional] a **reasoner**
- a **reasoning model**
 - (mapped) feature model [WP1]
 - variability model [WP2, WP3]
- a **context model**
 - simulated values [WP2]
 - actual values [WP3]
- [optional] a **configuration model**
- a **reasoning question**
- [optional] **reasoning parameters**
 - resource limitations

As output, the client module gets from the ARF

- a **reasoning result** as answer of the reasoning question
- a **reasoning log** consisting of a **protocol** (e.g. about used reasoner, about used resources) and a **statistics** (e.g. to report errors, to provide information)

Reasoning on a problem is provided by the ARF as answering reasoning questions, because also existing reasoning frameworks such as the FaMa³ Framework use questions to describe a

³ Feature Model Analyser (http://www.isa.us.es/fama/?FaMa_Framework)

reasoning problem. The usage of questions allows also extending the ARF without changing the framework interface. However reasoning on a new problem is enabled by adding a new question to the ARF and not by extending the interface by a new method.

The ARF answers a question by returning a result object. Other than existing reasoning frameworks the result of the ARF contains more than the answer for a question such as a configuration. The ARF can also return logging information. With the help of the logging information reasoning can be reproduced different times. E.g. if the used reasoner is known after reasoning at design-time, it can be reused at runtime to guarantee the same answer.

The following paragraphs specify requests to the ARF such as question and query and responses from the ARF such as information and result in an informal way.

5.1.1 Query and Question

Before asking a question, a client module possibly needs general knowledge about the functionality offered by the ARF. Because the ARF can also be adapted dynamically at runtime, information about the ARF can vary dynamically at system's runtime. Adaption of the ARF means that e.g. a newly developed reasoner has been loaded dynamically in the meantime or a reasoner has been dynamically unloaded because of resource limitations. Getting general information of the ARF is done with the help of a query, which allows interface independent ARF extensions like a question.

The ARF can answer the following queries

- Which reasoners exist overall?
- Which reasoners exist for a specific reasoning model type?
- Which reasoner exist for given input like
 - reasoning model and/or
 - reasoning context and/or
 - reasoning question and/or
 - reasoning parameter?
- Which expected values do have reasoning parameter regarding given input
 - time consumption and/or
 - memory usage?

A reasoning question specifies and describes what to do by the ARF. A question is answered by a reasoner, which is responsible for that question and the given reasoning model. If no reasoner is pre-selected the ARF selects one of its reasoner automatically. Among others the choice is done with the help of question attributes such as

- solving is forced
- validating is forced

Among the described questions in the use cases in section 3.2 the ARF can answer more questions. There can be found a detailed description of all questions in section 6.1. The ARF contains the following set of questions as default that are of concern for

- requirements-level:
 - Are there valid configurations?
- design-time:
 - Are there valid configurations?
 - How many valid configurations do exist?
 - Get all valid configurations.
 - Get one valid configuration.
- runtime:
 - Get best valid configuration for given context.
 - Get one valid configuration for given context.
 - Is configuration valid for given context?
 - Get next valid configuration for given context.

Questions may optionally be parameterised by reasoning parameters like

- time span limitations
- memory amount limits
- enablement of logging

5.1.2 *Information and Result*

The ARF returns information about all available reasoners as answer of a query. The answer of a query depends on the client module input. The same query can return different information, because the ARF has been changed dynamically or e.g. the question to be asked is parameterised. The returned information contains

- a list of matching reasoners and
- values for their attributes
 - time characteristic such as slow, medium or fast
 - memory usage such as low, medium or high
 - reasoning quality such as optimal or none-optimal

A reasoning result is part of an answer of a reasoning question. The ARF has to have a reasoner, which is responsible for that question and can answer it. Depending on the reasoning question the following reasoning results will be returned

- a Boolean value or
- a numeric value or
- one configuration or

- a list of configurations

Additionally the ARF returns a reasoning log object for the asked reasoning question. A protocol is part of the reasoning log object. The protocol can track

- reasoning parameters overall and
 - reasoning's time consumption
 - reasoning's memory usage
 - reasoning's number of configurations
- the used reasoner with reasoning parameters
 - reasoner's time consumption
 - reasoner's memory usage
 - reasoner's number of configurations

Also a statistics is part of the reasoning log object. Statistics can contain

- information and
 - tracing
- warnings and
 - inconsistency of input
- errors
 - exceptions like Out-of-Memory
 - invalid input like unknown model format

5.2 Interface Definitions

On the basis of the previous section this section defines the harmonized interface of each part of ARF in a technical manner. These interfaces are the base for the implementation of the ARF and its extensions.

5.2.1 Data and Parameter Interface

The general approach for the ARF consists in defining a mapping from requirements-level data, design-time data and runtime data to an internal ARF data. Depending on the level the input data consists of different models

- requirements-level
 - (mapped) feature model
- design-time
 - variability model
 - context model

- runtime
 - variability model
 - context model
 - configuration model

Such a mapping as defined by the following type `Mapper` enables the ARF to reason with the same limited set of reasoners on reasoning and validation problems. Although requirements-level produces feature models, its output will be a DiVA adaptation model as described in [4]. Thus all levels use the DiVA adaptation models as data exchange format. The mapping interface of the ARF handles only this model type, which is represented by an EMF⁴ resource.

```
Mapper {
    ARFModel mapModel (Resource);
    Resource mapModel (ARFModel);

    ARFContext mapContext (Resource, Resource);
    Resource mapContext (ARFModel, ARFContext);

    ARFConfiguration mapConfiguration (Resource, Resource,
    Resource);
    Resource mapConfiguration (ARFModel, ARFContext,
    ARFConfiguration);
}
```

The DiVA adaptation models can be exchanged by their EMF resource or their URI⁵. Because of that choice, the input data is encapsulated to reduce the complexity of the interface. The type `ModelHandle` encapsulates the reasoning model and thus includes the fixed input data of the ARF.

```
ModelHandle {
    void setModel (String);
    void setModel (Resource);
    Resource getModel ();
    String getModelURI ();
}
```

The following type `InputHandle` encapsulates a context model and configuration model and thus includes the variable input data of the ARF. This separation of fixed input data and variable input data allows the reuse of the ARF over the time for a given reasoning model.

⁴ Eclipse Modeling Framework (<http://www.eclipse.org/modeling/emf>)

⁵ Uniform Resource Identifier (<http://tools.ietf.org/html/rfc1630>)

```
InputHandle {
    void setContext(String);
    void setContext(Resource)
    Resource getContext();
    String getContextURI();

    void setConfiguration(String);
    void setConfiguration(Resource);
    Resource getConfiguration();
    String getConfigurationURI();
}
```

As described in previous paragraph reasoners and questions have attributes and reasoning can be parameterised. The following public types describe reasoner attributes, question attributes and reasoning parameters and their possible values. Attribute values in type `ReasonerAttributes` are qualitative ratings of reasoner features and attribute values in type `QuestionAttributes` characterise a question. All parameter values of the type `ReasoningParameters` can be set by a client module to specify reasoning and validation.

```
ReasonerAttributes {
    TIME {SLOW, MEDIUM, FAST, UNKNOWN}
    MEMORY {LOW, MEDIUM, HIGH, UNKNOWN}
    REASONING {OPTIMAL, NONE-OPTIMAL, UNKNOWN}
}

QuestionAttributes {
    SOLVING {TRUE, FALSE}
    VALIDATION {TRUE, FALSE}
}

ReasoningParameters : QuestionAttributes {
    LOGGING {TRUE, FALSE}
    TIME_CONSUMPTION {DONT-CARE, SECONDS as Integer, UNKNOWN}
    MEMORY_USAGE {DONT-CARE, BYTES as Integer, UNKNOWN}
}
```

Coming from these base types the next paragraphs define all ARF types which are used to reason on several problems.

5.2.2 *Reasoner Interface*

As described by the use cases a reasoner reasons on several questions. There are questions regarding getting or optimising one configuration, solving any configurations or validating a configuration. Thus reasoner's work is applied to three tasks

- optimising to provide for given input models access to some configurations
- solving to provide for given input models access to all configurations
- validating to check correctness of a configuration

According to these three tasks there are the three base types `Optimiser`, `Solver` and `Validator`. The type `Reasoner` extends these base types. If a reasoner is responsible for a question, reasoning means validating or either optimising or solving. Because a reasoner is not directly available for a client module, a handle of type `ReasonerHandle` is used, which represents the reasoner and also its parameters.

```
Optimiser {
    ConfigurationResult optimise(ModelHandle, InputHandle);
}

Solver {
    ConfigurationResult solve(ModelHandle, InputHandle);
}

Validator {
    BooleanResult validate(ModelHandle, InputHandle);
}

Reasoner : Optimiser, Solver, Validator {
    String getName();

    Result reason(ModelHandle, InputHandle, QuestionHandle);

    Boolean optimising();
    Boolean solving();
    Boolean validating();

    ReasonerAttributes getAttributes();

    List<String> getSupportedQuestions();
    List<String> getSupportedHandles();
}

ReasonerHandle {
    String getReasoner();
    ReasonerAttributes getAttributes();
}
```

5.2.3 Query and Question Interface

The type `ReasonerQuery` represents a query, which is asked for information about reasoners of the ARF. The query is parameterised both by input data models and a question handle. Because a query is part of the ARF, the query can not directly be used by a client module for security reasons. Thus it is described by a handle of type `ReasonerQueryHandle`.

```
ReasonerQuery {
    String getName();
    String getQuery();

    Information ask(ModelHandle, InputHandle, QuestionHandle);
}

ReasonerQueryHandle {
    String getQuery();
}
```

A question specifies what to do by the ARF and is characterised by its question parameters. The question is asked for a result as part of the answer. The used reasoner reasons on the input data model with attention to the set reasoning parameters. Same to a query a question is part of the ARF. Thus a handle for a question is used of type `QuestionHandle` which encapsulates both the question and the reasoning parameters.

```
Question {
    String getName();
    String getQuestion();
    QuestionAttributes getAttributes();

    Result ask(ReasonerHandle, ModelHandle, InputHandle,
    QuestionHandle);
}

QuestionHandle {
    String getQuestion();
    ReasoningParameters getParameters();
}
```

5.2.4 Information and Result Interface

Information of type `Information` is returned by the ARF as answer of a query. This information can vary, because as described before e.g. reasoner can be loaded or unloaded dynamically. The information consists of list of reasoner handles.

```
Information {
    List<ReasonerHandle> getReasoner();
}
```

Depending on the question there are three result types of the ARF. A question can return a Boolean value, a numeric value and a list of configurations. That means an answer with no configuration corresponds to an empty list and an answer with one configuration corresponds to a list with one member. According to this there are the three results of type `BooleanResult`, `NumericResult` and `ConfigurationResult`.

```
Result {
    void setAnswer(Object);
    Object getAnswer();
    void setLog(Log);
    Log getLog();
}

BooleanResult : Result {
    void setAnswer(Boolean);
    Boolean getAnswer();
}

NumericResult : Result {
    void setAnswer(Integer);
    Integer getAnswer();
}

ConfigurationResult : Result{
    void setAnswer(List<Resource>);
    List<Resource> getAnswer();
}
```

If logging is enabled during reasoning and validation a log of type `Log` is also part of the answer. The log is divided into a protocol of type `Protocol` and a statistics of type `Statistics`, containing information, warnings and errors. The protocol gives information about the used reasoner and provides values for reasoning parameters regarding the reasoning in general and the used reasoner in special.

```
Log {
    Protocol getProtocol();
    Statistics getStatistics();
}

Protocol {
    ReasoningParameters getParameters();
    ReasoningParameters getParameters(ReasonerHandle);
    List<ReasonerHandle> getUsedReasoner();
}

Statistics {
    List<String> getInfos();
    List<String> getWarnings();
    List<String> getErrors();
}
```

5.2.5 ARF Interface

Client modules use the ARF by the public interface `ARF`. This interface uses only public types, which are defined in the previous paragraphs. It is possible to set a reasoner handle, if it is

known, a model handle and input handle representing the input data to the ARF. After setting these objects the ARF can answer either a query or a question. If the ARF answers a query it returns an information object and if it answers a question it returns a result object.

```
ARF {
    void setReasoner(ReasonerHandle);

    void setModel(ModelHandle);

    void setInput(InputHandle);

    Result ask(QuestionHandle)

    List<ReasonerParameter> ask(ReasonerQueryHandle,
    QuestionHandle);

    ReasonerHandle loadReasoner(String);
    void unloadReasoner(ReasonerHandle);
}
```

5.3 Reasoning and Validation Walkthroughs

Deducing from the three uses cases and the harmonised interface definitions this section describes the three appropriate walkthroughs. Each walkthrough refers to the appropriate use case of the DiVA methodology workflow level and is divided into three phases

1. initialisation of input
2. reasoning and validation
3. analysing of output

5.3.1 Requirements-level Walkthrough

- initialisation phase
 - 1) getting variability model
 - 2) creation of model handle with variability model
 - 3) creation of a handle for ARE_THERE_VALID_CONFIGURATIONS-question

```
1: Resource vm = getVariabilityModel();
2: ModelHandle mhandle = new ModelHandle(vm);
3: QuestionHandle qhandle = new QuestionHandle(
    ARF.QUESTIONS.ARE_THERE_VALID_CONFIGURATIONS);
```

- reasoning and validation phase
 - 4) creation of a local ARF
 - 5) setting model handle to the ARF



- 6) asking question to the ARF
 - a) ARF selects a reasoner automatically
 - b) selected reasoner reasons on the input

```
4: ARF arf = new ARF();
5: arf.setModel(mhandle);
6: Result result = arf.ask(qhandle);
```

- analysis phase
 - 7) casting result to question-dependent result
 - 8) getting Boolean value from result

```
7: BooleanResult bresult = (BooleanResult) result;
8: Boolean answer = bresult.getAnswer();
```

5.3.2 Design-time Walkthrough

- initialisation phase
 - 1) getting variability model
 - 2) creation of model handle with variability model
 - 3) getting context model
 - 4) creation of level-dependent input handle with context
 - 5) creation of a handle for GET_ALL_VALID_CONFIGURATIONS-question
 - 6) creation of a handle for GET_ALL_REASONER-query

```
1: VariabilityModel vm = getVariabilityModel();
2: ModelHandle mhandle = new ModelHandle(vm);
3: Context context = getContext();
4: InputHandle ihandle = new InputHandle(context, null);
5: QuestionHandle qhandle = new QuestionHandle(
    ARF.QUESTIONS.GET_ALL_VALID_CONFIGURATIONS);
6: ReasonerQueryHandle rqhandle = new ReasonerQueryHandle(
    ARF.QUERIES.GET_ALL_REASONER);
```

- reasoning and validation phase
 - 7) creation of a local ARF
 - 8) setting model handle to the ARF
 - 9) setting input handle to the ARF
 - 10) asking query to the ARF

11) selecting reasoner handle of an optimal reasoner (BDDReasoner)

12) setting reasoner handle to the ARF

13) asking question to the ARF

```
7: ARF arf = new ARF();
8: arf.setModel(mhandle);
9: arf.setInput(ihandle);

10: Information information = arf.ask(rqhandle, qhandle);
    List<ReasonerHandle> handles =
        information.getReasoner();
11: ReasonerHandle rhandle = null;
    for (ReasonerHandle handle : handles) {
        if (handle.getAttributes().REASONING ==
            ReasonerParameters.OPTIMAL) {
            rhandle = handle;
            break;
        }
    }
12: arf.setReasoner(rhandle);
13: Result result = arf.ask(qhandle);
```

- analysis phase

14) casting result to question-dependent result

15) getting configurations from result

16) getting log from result

17) getting protocol from log

18) getting statistics from log

```
14: ConfigurationResult cresult =
    (ConfigurationResult) result;
15: List<Configuration> answer = cresult.getAnswer();
16: Log log = cresult.getLog();
17: Protocol protocol = log.getProtocol();
18: Statistics statistics = log.getStatistics();
```

5.3.3 Runtime Walkthrough

In opposite to the previous walkthroughs, the runtime walkthrough has to capture two separate tasks of the runtime use case. The first task solves one configuration and the second task validates the configuration. For this also this walkthrough is divided into two tasks, whereby the second task reuses data from the first task.

Solving Task

- initialisation phase

- 1) getting variability model
- 2) creation of model handle with variability model
- 3) getting context model
- 4) creation of input handle with context
- 5) creation of a handle for GET_ONE_VALID_CONFIGURATION-question
- 6) setting TIME-parameter of question handle to 60 seconds
- 7) getting AT reasoner handle

```
1: VariabilityModel vm = getVariabilityModel();
2: ModelHandle mhandle = new ModelHandleImpl(vm);
3: Context context = getContext();
4: InputHandle ihandle = new InputHandle(context, null);
5: QuestionHandle qhandle = new QuestionHandle(
    ARF.QUESTIONS.GET_ONE_VALID_CONFIGURATION);
6: qhandle.getParameters().TIME_CONSUMPTION = 60;
7: ReasonerHandle rhandle = getATReasoner();
```

- reasoning and validation phase
 - 8) creation of a local ARF
 - 9) setting reasoner handle to the ARF
 - 10) setting model handle to the ARF
 - 11) setting input handle to the ARF
 - 12) asking question to the ARF

```
8: ARF arf = new ARF();
9: arf.setReasoner(rhandle);
10: arf.setModel(mhandle);
11: arf.setInput(ihandle);
12: Result result = arf.ask(qhandle);
```

- analysis phase
 - 13) casting result to question-dependent result
 - 14) getting first configuration from result

```
13: ConfigurationResult cresult =
    (ConfigurationResult) result;
14: Configuration configuration =
    cresult.getAnswer().get(0);
```

Validation Task

- initialisation phase

15) setting configuration to input handle

16) creation of a handle for IS_VALID_CONFIGURATION-question

```
15: ihandle.setConfiguration(configuration);
16: qhandle = new QuestionHandle(
    ARF.QUESTIONS.IS_VALID_CONFIGURATION);
```

- reasoning and validation phase

17) setting no reasoner to the ARF

18) asking question to the ARF

```
17: arf.setReasoner(null);
18: result = arf.ask(qhandle);
```

- analysis phase

19) casting result to question-dependent result

20) getting Boolean value from result

```
19: BooleanResult bresult = (BooleanResult) result;
20: Boolean answer = bresult.getAnswer();
```

6 ARF Implementation

The ARF as described in the previous chapters provides a uniform interface and execution environment for reasoning and validation tasks within the DiVA project. It is not the intention of the DiVA project to develop entirely new reasoning engines but to harvest as much as possible from existing solutions. Based on the survey of D4.1 suitable algorithms and engine frameworks with different characteristics are candidates for inclusion in the ARF.

The next section 6.1 describes all reasoning questions in detail, which are implemented and part of the ARF by default. As written in chapter 5 the data model for all three levels is identical since the DiVA adaptation model is used. Section 6.2 defines the mapping of the DiVA meta model to the internal ARF model. This chapter describes also the selection criteria for reasoning engines in section 6.3 and gives a brief description of implemented reasoner their capabilities and characteristics in section 6.4.

6.1 Reasoning and Validation Questions

The ARF offers a fixed set of reasoning and validation questions, which are described in the next paragraphs. These questions represent the most obvious requests of all work packages, which shall be answered by the ARF. It is also possible to extend this set of questions by own questions. There are differences in the usage of questions regarding the DiVA workflow level, because each level has its own environment and requests. In the overview tables the usage is denoted by the three characters 'X', 'O', and '-':

- **X:** This state describes, that the level is interested in the question and it is in the main focus. It is used.
- **O:** This state describes, that the level is more or less interested in the question and it is nice to have. It can be used.
- **-:** This state describes, that the level is not interested in the question, because it makes no sense in this manner. It is available.

At requirement level a textual requirements documents is processed. The output of this process is a feature model, which is mapped to a variability model. From the perspective of the DiVA methodology workflow this level captures requirements and models variability and adaptation. Thus reasoning and validation has to support modelling by some questions for correctness or consistency check.

Design-time level adds information about system internals. At this level reasoning and validation works on context models with simulated values with e.g. unlimited resources like memory and time. Therefore it is more a simulation without real-time constraints and further questions has to support this simulation aspect.

At runtime reasoning and adaptation has to fulfil real-time constraints. If an environment of a running system changes it has to adapt to a new system configuration. Conditions like memory or time constraints as well as the knowing that at most one configuration is needed to run have to be supported by more questions. All questions may be parameterised with resource usage constraints.

6.1.1 *Are there valid configurations - Question*

The simplest request to the ARF is the question: Are valid configurations for a given reasoning model and/or context model available. The answer gives a first insight, if the given reasoning model is correct and consistent. This means, at least one configuration has to be found because a

running system can be described by it. Otherwise the reasoning model has to be changed and has to be improved.

Level	Work package	Usage
Requirements level	WP1	X
Design-time	WP2	O
Runtime	WP3	-

6.1.2 How many valid configurations do exist - Question

The possibility to ask if there are configurations or not does not say anything about the total number of configurations. Reducing the number of possible configurations in many cases reduces reasoning resource usage.

Level	Work package	Usage
Requirements level	WP1	X
Design-time	WP2	O
Runtime	WP3	-

6.1.3 Get all valid configurations - Question

With the help of this question all possible configurations of a reasoning model and/or context model are solved. This request is useful for a simulation environment than for a real life environment. Because there is more than one configuration an additional analysis or further processing like scoring, sorting or comparing can be done.

Level	Work package	Usage
Requirements level	WP1	O
Design-time	WP2	X
Runtime	WP3	-

6.1.4 Get one valid configuration - Question

Among the request for the existence of at least one configuration the request to get this configuration is just as obvious. This question implies completing the reasoning, if one configuration is found. Thus if only one configuration is required, this question enables getting it as fast as possible. But there can be given no statement about goodness of this configuration, because it is the first found configuration. Perhaps this configuration is best-fitting, but it could also be the worst-fitting configuration.

Level	Work package	Usage
Requirements level	WP1	-
Design-time	WP2	O

Level	Work package	Usage
Runtime	WP3	X

6.1.5 Get best valid configuration - Question

Compared to the previous question the request for the best configuration is in general much more resource intensive. Depending on the reasoning approach information for all possible configurations is required as input for a rating/scoring. Since in combination with time/memory resource constraints reasoning might stop before checking all configurations, the result will be the best known valid configuration.

Level	Work package	Usage
Requirements level	WP1	-
Design-time	WP2	O
Runtime	WP3	X

6.1.6 Get next valid configuration - Question

If reasoning returned a valid configuration previously, but it does not fit in the running system for any reason, a new configuration can be requested. Reasoning engines should deliver a new valid configuration. Reasoning engines are not required to ensure uniqueness of configurations. That is, a configuration might be returned several times by the reasoning engine. However it is not permitted to return the same configuration as the previous configuration. If a reasoning engine can detect that all possible configurations have been returned an empty result should be returned to indicate this condition.

Level	Work package	Usage
Requirements level	WP1	-
Design-time	WP2	-
Runtime	WP3	X

6.1.7 Is configuration valid - Question

With respect to the validation rules of a work package a solved configuration has to be validated, if it fit in a running system. Thus the ARF has to handle this request and check whether the configuration fulfils these validation rules. Otherwise the ARF has to be requested for a next valid configuration as provided by the previous question.

Level	Work package	Usage
Requirements level	WP1	
Design-time	WP2	
Runtime	WP3	X

6.1.8 Reasoning and Validation Parameters

As mentioned at the beginning reasoning and validation is more than answering a question. A reasoning request can also include some parameters which describe memory usage and time consumption as resource constraints and enables logging for tracing purposes. Answering a question has to deal with these parameters and the answer of a question depends always on these parameters. If the ARF has to solve e.g. the best configuration, it solves the best configuration within the time limit and/or memory limit. But a better configuration could be possible, if there is more time or more memory available.

In difference to the logging parameter, which is a Boolean value, the memory usage parameter and the time consumption parameter are numeric values. Memory usage is given in mega bytes and time consumption in seconds. Setting a value to these parameters is not required, if no values can be set for any reason. In this case the default is to have no limit for memory usage and time consumption.

6.2 DiVA Adaptation Model Mapping

As mentioned before reasoning problems are mapped to the same technical base. The current ARF implementation uses the *pure::variants*⁶ meta model for this purpose. In future versions this is intended to be replaced by the EMF feature model meta model, which will provide an open source API for feature models. According to the three DiVA models the next sections describes the mapping of the DiVA variability model, the DiVA context model and the DiVA configuration model to the appropriate *pure::variants* meta model.

6.2.1 Mapping of DiVA Variability Model

The DiVA variability model[5] is mapped to the internal ARF model, which is based on the *pure::variants* feature model meta model. A *pure::variants* feature model contains elements which are so called features. Among others a feature consists of the variability type, a unique name and a visible name. Each element of type `diva.NamedElement` of the DiVA variability model is mapped to a feature with the following rule:

- Supported feature group variability types are:
 - Mandatory [n]
 - Optional [0..n]
 - Alternative [1]
 - Or [1..n]
 - free range [x..y]
- Feature's unique name is composed of
 - a prefix followed by an underline such as `Diva_` and
 - the id of the `diva.NamedElement` (`diva.NamedElement->Id`)
- Feature's visible name is set to

⁶ *pure::variants* is a tool for variant management of product line based software development. (<http://www.pure-systems.com/Products.products.0.html>)

- the name of the `diva.NamedElement` (`diva.NamedElement->Name`)

In the following paragraphs a feature will be notated as follows:

[Variability Type] Unique Name (Visible Name)

An element of type `diva.DivAModelElement` of the DiVA variability model can contain attributes. These attributes are mapped to attributes of the corresponding feature in the *pure::variants* feature model. An attribute of a feature consists of a type, a name and one value or an array of values. An attribute of a `diva.DivAModelElement` will be mapped to an attribute of a feature with the following rule:

- Attribute's type corresponds to
 - the type of the `diva.DivAModelElement`'s attribute such as `Integer`
- Attribute's name is set to
 - the name of the `diva.DivAModelElement`'s attribute such as `DivaAttribute`
- Attribute's value contains
 - the value of the `diva.DivAModelElement`'s attribute (`diva.DivAModelElement->Attribute`)

In the following paragraphs an attribute of a feature will be notated as follows:

[Type] Name = Value(s)

Some elements of the DiVA variability model among others can contain a constraint expression of type `diva.Expression`. A *pure::variants* feature model allows also the definition of constraints, which have a name and a rule and can be associated with a feature. A `diva.Expression` will be mapped to a constraint with the following rule:

- Constraint's name is set to
 - the type of the `diva.Expression` reference link such as `DivaConstraint`
- Constraint's rule contains
 - the term of the `diva.Expression` (`diva.Expression->Term`)

In the following paragraphs a constraint associated with a feature will be notated as follows:

Name: Rule

Base Structure

Before mapping elements of the DiVA variability model a *pure::variants* feature model is created with the following fixed base structure:

- [Mandatory] `VariabilityModel` (Variability Model)
 - [Mandatory] `Variables_Group` (Variables)
 - [Mandatory] `Properties_Group` (Properties)

- [Mandatory] Dimensions_Group (Dimensions)
- [Mandatory] Rules_Group (Rules)

Variable Mapping

DiVA context variables of type `diva.Variable` are mapped to mandatory or optional features. The prefix of the unique name is `Variable`. Referring to the base structure this feature is put to the `pure::variants` feature model as follows:

- [Mandatory] Variables_Group (Variables)
 - [Variability Type] Variable_diva.Variable->Id (diva.Variable->Name)

The variability type is derived from the context variable type. There are Boolean variables and enumeration variables. A Boolean variable is mapped to an optional feature, because it can be part of the configuration or not. An enumeration variable is mapped to a mandatory feature and its literals to alternative features, because one of these literal must be part of the configuration. The model structure for a Boolean variable is as follows:

- [Mandatory] Variables_Group (Variables)
 - [Optional] Variable_diva.BooleanVariable->Id (diva.BooleanVariable->Name)

And the model structure for an enumeration variable and its literals is as follows:

- [Mandatory] Variables_Group (Variables)
 - [Mandatory] Variable_diva.EnumVariable->Id (diva.EnumVariable->Name)
 - [Alternative] Variable_diva.EnumVariable->Id_diva.EnumLiteral->Id (diva.EnumLiteral->Name)

Property Mapping

DiVA properties of type `diva.Property` are mapped to mandatory features, because they must be part of the configuration. The prefix of the unique name is `Property`. Additionally a DiVA property has the integer attribute named `Direction`. This attribute is mapped to an attribute of the corresponding feature. The base structure is extended as follows:

- [Mandatory] Properties_Group (Properties)
 - [Mandatory] Property_diva.Property->Id (diva.Property->Name)
 - [Integer] Direction = diva.Property->Direction

Dimension Mapping

DiVA dimensions of type `diva.Dimension` are mapped to mandatory features, if they must be part of the configuration due to a lower range boundary greater than 0. An optional feature is created, if they can be part of the configuration due to lower range boundary equals 0. The prefix of the unique name is `Dimension`. Additionally a DiVA dimension has the attribute named `Property`, which contains an array of DiVA properties. This attribute is mapped to an attribute of the corresponding feature. The model structure is as follows:

- [Mandatory] Dimensions_Group (Dimensions)
 - [Variability Type] Dimension_diva.Dimension->Id (diva.Dimension->Name)

- [String[]] Properties = [diva.Dimension.Property->Id]

Variant Mapping

DiVA variants which are located below DiVA dimensions are mapped to alternative or or-features. An alternative feature is created, if the DiVA dimension's upper range boundary equals 1 and thus only one variant may be chosen. Otherwise an or-feature of range [diva.Dimension-> Lower, diva.Dimension->Upper] is created to allow a range of chosen variants. The prefix of the unique name of such a feature is Variant. The model structure of a DiVA variant is as follows:

- [Optional | Mandatory] Dimension_diva.Dimension->Id (diva.Dimension->Name)
 - [Variability Type] Variant_diva.Variant->Id (diva.Variant->Name)

Mapping of `diva.PropertyValue`:

A DiVA variant contains property values of type `diva.PropertyValue`. A property value is mapped to an integer attribute of the feature corresponding to the DiVA variant. The name of the attribute is the identifier of the appropriate DiVA property and the value is set to the value of the property value. The model structure is as follows:

- [Alternative | Or] Variant_diva.Variant->Id (diva.Variant->Name)
 - [Integer] diva.PropertyValue->Property->Id = diva.PropertyValue->Value

Mapping of `diva.Expression`:

A DiVA variant also consists of expressions of type `diva.Expression`. These expressions are mapped to constraints. The name depends on the type of reference link and the rule is in general like X implies Y. *pure::variants* fully supports the DiVA expression semantics with its own constraint language.

Expressions of type `diva.ContextExpression` refer to DiVA context variables. There are the two reference links *Available* and *Required*, which are mapped to the appropriate two constraints as follows:

- [Alternative | Or] Variant_diva.Variant->Id (diva.Variant->Name)
 - Available: Variant_diva.Variant->Id *implies* diva.ContextExpression->Term
 - Required: diva.ContextExpression->Term *implies* Variant_diva.Variant->Id

The second kind of expression is of type `diva.VariantExpression` and refers to DiVA variants. There exists one reference link named *Dependency*, which is mapped to a constraint as follows:

- [Alternative | Or] Variant_diva.Variant->Id (diva.Variant->Name)
 - Dependency: Variant_diva.Variant->Id *implies* diva.VariantExpression->Term

Priority Rule Mapping

DiVA priority rules of type `diva.PriorityRule` are mapped to optional features, because these rules are automatically chosen by their context expressions. The prefix of the unique name is Rule. The base structure of the *pure::variants* feature model is extended as follows:

- [Mandatory] Rules_Group (Rules)

- [Optional] Rule_diva.PriorityRule->Id (diva.PriorityRule->Name)

Mapping of `diva.ContextExpression`:

A DiVA priority rule has one context expression of type `diva.ContextExpression`. This expression refers to a DiVA context variable and is mapped to a constraint. The constraint named Context has a rule of syntax (X implies Y) and (Y implies X). The model structure of a DiVA priority rule including its context expression is as follows:

- [Optional] Rule_diva.PriorityRule->Id (diva.PriorityRule->Name)
 - Context: (diva.ContextExpression->Term *implies* Rule_diva.PriorityRule->Id) *and* (Rule_diva.PriorityRule->Id *implies* diva.ContextExpression->Term)

Mapping of `diva.PropertyPriority`:

A DiVA priority rule has also several property priorities of type `diva.PropertyPriority` which are mapped to attributes of integer type. The model structure of a DiVA priority rule including its property priorities is as follows:

- [Optional] Rule_diva.PriorityRule->Id (diva.PriorityRule->Name)
 - [Integer] diva.PropertyPriority->Property->Id = diva.PropertyPriority->Priority

Oracle Mapping

A DiVA context, which is part of the DiVA variability model, can contain an expression as oracle. This expression describes which variants are expected to be part of the final configuration. This oracle has to be mapped to a feature of a *pure::variants* feature model. Therefore its base structure is extended as follows:

- [Mandatory] VariabilityModel (Variability Model)
 - [Mandatory] Oracles_Group (Oracles)

As mentioned before among others a DiVA context consists of an oracle expression of type `diva.VariantExpression`. This expression is mapped to a constraint named Oracle and a rule of syntax (X implies Y) and (Y implies X). This constraint is associated with an optional feature with unique name prefix Oracle and represents the DiVA context. The model structure is extended as follows:

- [Mandatory] Oracles_Group (Oracles)
 - [Optional] Oracle_diva.Context->Id (diva.Context->Name)
 - Context:(diva.VariantExpression->Term *implies* Oracle_diva.Context->Id) *and* (Oracle_diva.Context->Id *implies* diva.VariantExpression->Term)

6.2.2 Mapping of DiVA Context Model

The DiVA context model is mapped in general to the internal ARF context. In its current implementation the ARF uses *pure::variants* variant description models for this purpose. A *pure::variants* variant description model contains elements which are so called selections. Among others a selection consists of the selection type and a reference to a feature of a *pure::variants* feature model. A selection is mapped by the following rule:

- Selection's selection type is even:

- Excluded or
- Selected
- Selection's reference is composed of
 - an appropriate prefix followed by an underline like `Divu_` and
 - the id of an `diva.NamedElement` (`diva.NamedElement->Id`)

In the following a selection will be notated as follows:

[Selection Type] Reference

DiVA context elements define values for all context variables of a DiVA variability model. Values of context variable of type `diva.BooleanVariable` are of type `diva.BoolVariableValue` and values of context variable of type `diva.EnumVariable` are of type `diva.EnumVariableValue`. Because the selection represents the value of a context variable it references the appropriate feature. Therefore the prefix, which is used for the reference, is `Variable`.

The selection type of a variable of type `diva.BooleanVariable` depends on the value. If the value is true, the feature is selected by the selection. Otherwise the feature is excluded by the selection. The ARF context contains a selection for a Boolean context variable as follows:

- `[Excluded | Selected] Variable_diva.BoolVariableValue->Variable->Id`

All possible values for a variable of type `diva.EnumVariable` are defined by its literals. Therefore the selection does not reference the feature of the appropriate context variable, but references the feature of a mapped literal. Because literals are mapped to alternative features, one selected feature of that group excludes the others automatically. A literal value is mapped to a selection of the ARF context as follows:

- `[Selected] Variable_diva.EnumVariableValue->Variable->Id_`
`diva.EnumVariableValue->Literal->Id`

6.2.3 Mapping of DiVA Configuration Model

The DiVA configuration model is mapped in general to the internal ARF configuration, which is based on a *pure::variants* variant description model. As described before a *pure::variants* variant description model contains selections and will be notated as follows:

[Selection Type] Reference

The DiVA configuration model contains DiVA configurations. These configurations define DiVA aspect elements which refer to variants of a DiVA variability model. Because the selection has to represent a variant it has to reference the appropriate feature. Therefore the prefix, which is used for the reference, is `Variant`. These variants are part of the final system configuration and have to be selected by a selection:

- `[Selected] Variant_diva.Variant->Id`

All other variants have to be excluded by a selection, because they are not part of the final system configuration:

- `[Excluded] Variant_diva.Variant->Id`

6.3 Initial Reasoning Engine Selection

Conceptually there is no limitation on the number of reasoning engines included in the ARF. To meet the requirements set forth in chapters 2 and 3 for larger systems multiple reasoners with different algorithms are necessary. However, to quickly support all use cases under the assumption of a limited size input models, only a single reasoner is necessary if it provides answers to all questions within the maximum time span defined by the DiVA WP6 use cases.

Reasoners based on propositional logic like Binary Decision Diagrams (BDD) & Satisfiability (SAT) solvers can be used to answer all questions with the exception of WP2 scoring and ranking [5] to find the best configuration for a given context and the validation of the runtime system. The scoring as described in [5] is a simple calculation and for a relatively limited number of configurations the score can be calculated for all configurations to select the configuration with the highest score. Validation of the runtime system is already implemented as a separate engine by WP3 which is to be integrated into the ARF. Thus BDD/SAT solver based engines are selected as initial reasoning approach.

In addition to these reasoning engines a “dummy” engine is provided for testing and fast checking whether the ARF works. This engine accepts every question but returns a default result such as

- False for a Boolean result
- ‘0’ for a numeric result
- Empty list for a configuration result

6.4 ARF Reasoner

These following two implementations of reasoner are currently available in the initial ARF. One is based on the BDD reasoning engine and is provided by *pure::variants* (based on *pure::variants*) and is described in section 6.4.1. The other one is a port of the initial WP2 simulator which is based on Alloy [7] and described in section 6.4.2.

6.4.1 CUDD BDD Reasoner

The CUDD BDD reasoner is provided by *pure::variants*. Internally *pure::variants* uses several different techniques for its feature model evaluation, including a mapping to the CUDD binary decision diagram library. CUDD has been described as part of the survey in [2].

The BDD reasoner in *pure::variants* does not provide support for the full *pure::variants* meta model but is restricted to evaluation of propositional logic. Reasoning on feature attributes is not supported. However, this is currently not a relevant restriction, since the DiVA meta model does not require use of these concepts in feature models.

The BDD reasoner basically builds a full representation of the logic formulas in a way that when all formulas are represented, all results are known and can be derived relatively quickly. The time required to build the BDD increases exponentially, thus for larger input the construction takes very long time. Additionally the order in which the formulas are processed plays an important role for efficiency. The *pure::variants* BDD reasoner has been tuned to provide good performance for feature model structures. However, due to the time it takes to build the BDD for larger input models, it does not qualify as a reasoning approach generally usable for “Any Time” reasoning, which could be proven by experiments (see section 7.1).

6.4.2 Alloy based SAT Solver Reasoner

Alloy [7] is a structural modelling language based on first-order logic, for expressing complex structural constraints and behaviour. The Alloy Analyzer is a constraint solver that provides fully automatic simulation and checking. Alloy has been developed by the Software Design Group at MIT⁷. The Alloy Analyzer is, technically speaking, a 'model finder'. Given a logical formula (in Alloy), it attempts to find a model -- a binding of the variables to values -- that makes the formula true. Alloy provides a (textual) language to express models for which it solves the constraints

Alloy in its current release is based on Kodkod [8], which in turn uses different well known open source SAT solvers (e.g. SAT4J, zChaff or MiniSAT as described in [2]).

The ARF generates an alloy specification (in textual form) from the feature model and reads the results from Alloy also in textual form.

⁷ Massachusetts Institute of Technology (<http://web.mit.edu/>)

7 Conclusion and Outlook to D4.3

7.1 Comparing the implemented Reasoner

The expressiveness of the Alloy language is comparable to the language supported by the CUDD BDD Reasoning Engine. The main difference is the speed for delivering the first valid configuration. Since a BDD contains all solutions once constructed, and construction time corresponds to the size of the problem, it can be very long to build the BDD and derive the first configuration.

Initial experiments showed that for small problems (initial WP6 CAS use cases) both reasoning engines perform well and deliver all configurations in about the same time through the ARF. For larger models there are significant differences. The test models were enlarged by creating 2 to 5 independent identical clones of the original model, the means size of the input model grows up to 5 times and number of possible configurations is number of initial configurations power number of cloned instances. Tests were run on a 2,2 GHz 4 Core system. Both reasoning engines are single threaded, that is, only one core was used.

Model	CAS		CAS x 2		CAS x 3		CAS x 4		CAS x 5	
#Configuration (maximum)	44		1936		85184		3748096		164916224	
#Dimensions	8		16		24		32		40	
0 : 1 Range	3		6		9		12		15	
0 : -1 Range	2		4		6		8		10	
1 : 1 Range	3		6		9		12		15	
1 : -1 Range	0		0		0		0		0	
#Variants	21		42		63		84		105	
#Variants per Dimension	2,625		2,625		2,625		2,625		2,625	
Reasoner	BDD	SAT	BDD	SAT	BDD	SAT	BDD	SAT	BDD	SAT
Are there configurations (ms)	625	403	4161	2800	19770	11122	107096	31507	498141	60746
How many configurations (ms)	2057	20210	91083	TIME	MEM	TIME	MEM	TIME	MEM	TIME
Get one configuration (ms)	635	391	4188	2828	19890	10818	-	-	-	-
Get all configurations (ms)	2057	20799	93076	TIME	MEM	TIME	MEM	TIME	MEM	TIME
Get best configuration (ms)	2062	20401	93567	TIME	MEM	TIME	MEM	TIME	MEM	TIME

Table 5 Comparison CUDD BDD and Alloy SAT solver

All times in the table are given in milliseconds. ‘MEM’ indicates that available memory on target machine (1.5GB) was used up by the BDD reasoner before delivering an answer to the ARF application. However, further research showed that this is due to an implementation issue in the BDD reasoner (it tries to keep all calculated results in memory). ‘TIME’ means that within the given time (10 min) the Alloy SAT reasoner was not able to answer this question. ‘-’ indicates that no measurement was done.

As expected, times for getting the first valid configuration and getting an answer if there is any valid configuration are more or less identical. For larger models only the later question was timed.

It is clearly visible that getting a first configuration is much faster with the Alloy SAT reasoner, however it delivers results at a much lower rate. Also of interest is the much slower increase of this time to the first result compared to the BDD reasoner. It seems that at least for this problem the Alloy SAT reasoner's reasoning time grows almost proportional for larger input models. Further investigation is required here.

With this property, the Alloy SAT reasoner is a good candidate for an "Any Time" reasoner. But even though it delivers the result much faster than the BDD reasoner for larger systems, it fails in scalability considering real-time requirements.

7.2 Future Work

Task 4.2 provided the ARF in its initial form with set of reasoning engines which perform its task "perfect" in the way of delivering always (given enough time) the optimal result. Task 4.3 will have to extend the base of available reasoning engines by exploring alternative approaches based on heuristics, partitioning and combination of different reasoning approaches, so that a reasonable performance for the reasoning can be achieved for typical sizes of systems. Especially the use of pre-computed partial results as input for reasoning during runtime could provide benefits.

References

- [1] DiVA (2007): "Annex I - Description of work". DiVA project.
- [2] Hentze, M. Beuche, D. Muñoz, F. Grønmo, R. Bencomo, N. Ayed, D. Genßler, T. (2009): "Survey and evaluation of approaches for the adaptation reasoning framework". Deliverable 4.1, DiVA project.
- [3] DiVA (2009): "D6.1 - Case Study Specification and Requirements". Work package 6 deliverable. DiVA project.
- [4] Chitchyan, R. (2009): "Survey and Evaluation Document of the Requirements Engineering for Dynamic Variability". Deliverable 1.1, DiVA project.
- [5] Fleurey, F. Dehlen, V. Solberg, A. (2009): "An adaptation meta-model supporting specification, simulation and execution of adaptive systems". ECMDA 2009.
- [6] Morin, B. (2009): "Survey and evaluation of approaches for runtime variability management". Deliverable 3.1, DiVA project.
- [7] Seater, R., Dennis, G. (2006): "Tutorial for Alloy Analyzer 4.0". <http://alloy.mit.edu/alloy4/tutorial4/>.
- [8] Torlak, E., Dennis, G. (2006): "Kodkod for Alloy Users". First ACM Alloy Workshop, <http://alloy.mit.edu/kodkod/>.
- [9] David, P., Ledoux, T. (2006): "Safe Dynamic Reconfigurations of Fractal Architectures with FScript". Proc. In Proceeding of Fractal CBSE Workshop, ECOOP'06, Nantes, France.
- [10] Fleurey, F., Dehlen, V., Bencomo, N., Morin, B., Jézéquel, J.-M. (2008): "Modeling and Validating Dynamic Adaptation". Proc. In the Models@run.time at MODELS 2008, Toulouse, France.
- [11] Zhang, J., Cheng, B.H.C. (2005): "Specifying adaptation semantics". Proc. In Proceedings of the 2005 Workshop on Architecting Dependable Systems WADS '05, St. Louis, Missouri, US.
- [12] Floch, J., Hallsteinsen, S., Stav, E., Eliassen, F., Lund, K., Gjørven, E. (2006): "Using architecture models for runtime adaptability", Software IEEE, 2006, pp. 62-70.
- [13] Hallsteinsen, S., Stav, E., Solberg, A., Floch, J. (2006): "Using product line techniques to build adaptive systems". Proc. In SPLC'06: 10th Int. Software Product Line Conference, Washington, DC, USA.
- [14] Binder, R.V. (1999): "Testing Object-Oriented Systems: Models, Patterns and Tools" (Addison-Wesley).
- [15] Cohen, D.M., Dalal, S.R., Parelius, J., Patton, G.C. (1996): "The combinatorial design approach to automatic test generation", IEEE Software, 1996, pp. 83 – 88.
- [16] Kuhn, D.R., Wallace, D.D. (2004): "Software Fault Interactions and Implications for Software Testing", IEEE Transactions on Software Engineering, 2004, pp. 418 - 421
- [17] Zhang, J., Goldsby, H., Cheng, B.H.C. (2009): "Modular Verification of Dynamically Adaptive Systems". Proc. Proceedings of Eighth International Conference on Aspect-Oriented Software Development (AOSD'09), Charlottesville, Virginia, USA, March 2009.



- [18] Zhang, J., Cheng, B.H.C. (2006): "Model-based development of dynamically adaptive software". Proc. Proceedings of the 28th international conference on Software engineering (ICSE'06), Shanghai, China.
- [19] Biyani, K.N., Kulkarni, S.S. (2007): "Testing Dynamic Adaptation in Distributed Systems". Proc. Proceedings of the Second International Workshop on Automation of Software Test.
- [20] Babao, z., lu, Marzullo, K. (1993): "Consistent global states of distributed systems: fundamental concepts and mechanisms": "Distributed systems (2nd Ed.)" (ACM Press/Addison-Wesley Publishing Co.), pp. 55-96.
- [21] Kulkarni, S.S., Biyani, K.N. (2004): "Correctness of Component-Based Adaptation": "Component-Based Software Engineering", pp. 48-58.
- [22] Allen, R., Douence, R., Garlan, D. (1998): "Specifying and Analyzing Dynamic Software Architectures": "Fundamental Approaches to Software Engineering", pp. 21.
- [23] Kramer, J., Magee, J. (1998): "Analysing Dynamic Change in Software Architectures: A Case Study". Proc. Proceedings of the International Conference on Configurable Distributed Systems.
- [24] Heng, L., Chan, W.K., Tse, T.H. (2008): "Testing pervasive software in the presence of context inconsistency resolution services", in Editor (Ed.)^(Eds.): "Book Testing pervasive software in the presence of context inconsistency resolution services" (ACM, 2008, edn.), pp. 61-70.
- [25] Heng, L., Chan, W.K., Tse, T.H. (2006): "Testing context-aware middleware-centric programs: a data flow approach and an RFID-based experimentation", in Editor (Ed.)^(Eds.): "Book Testing context-aware middleware-centric programs: a data flow approach and an RFID-based experimentation" (ACM, 2006, edn.).