




## Dynamic Variability in complex, Adaptive systems

Deliverable reference: <b>D4.1</b>	Date: 17. March 2009	Responsible partner: pure-systems
<p>Title:</p> <h3 style="margin: 0;">Survey and evaluation of approaches for the adaptation reasoning framework</h3>		
Editor(s): <b>Michael K. W. Hentze</b>	Approved by: <b>Arnor Solberg</b>	
	Classification: <b>Public</b>	
<p>Abstract / Executive summary:</p> <p>Creation of an adaptation reasoning framework for dynamic, adaptive, and distributed systems is not a trivial task. Such a framework reasons on the variability that is represented by models implementing DiVA's MDE approach and which causes, due to the potential big size of such systems, an explosion of complexity.</p> <p>A dynamic system operates in real-time and is in the potentially constant need to respond fast to component or context changes through global reconfiguration. Complexity explosion would, if not appropriately handled, result in unacceptable calculation costs and too long reaction time spans while deciding which of a large number of possible configuration variants is valid or even optimal.</p> <p>This survey researches and evaluates existing reasoning approaches and shows ways to reduce reasoning complexity in the context of DiVA. Results will be the base of the later Reasoning Framework implementation of WP4 that is coupled to the concepts and implementations of the technical DiVA work packages 1, 2, and 3.</p>		
<div style="display: flex; justify-content: space-between; align-items: center;"> <div style="font-size: small;"> <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> </div> <div style="text-align: center;">  <p><b>COOPERATION</b></p> </div> <div style="font-size: x-small;"> <p>INTERNATIONAL DOCUMENT AVAILABLY: PARTNER + REPORT NUMBER ISBN</p> </div> </div>		



## Table of contents

<b>Survey and evaluation of approaches for the adaptation reasoning framework.....</b>	<b>1</b>
<b>Table of contents .....</b>	<b>2</b>
<b>Version History .....</b>	<b>5</b>
<b>Contributing partners.....</b>	<b>6</b>
<b>List of figures.....</b>	<b>7</b>
<b>List of tables.....</b>	<b>8</b>
<b>1 Introduction .....</b>	<b>9</b>
1.1 PROJECT MOTIVATION AND GOALS .....	9
1.2 REASONING FRAMEWORK .....	10
1.3 OBJECTIVES OF THE SURVEY .....	11
<b>2 Fundamentals.....</b>	<b>12</b>
2.1 ADAPTIVE, DISTRIBUTED SOFTWARE SYSTEM .....	12
2.2 SYSTEM BEHAVIOUR .....	13
2.3 KNOWLEDGE BASED SYSTEMS AND REASONING .....	14
2.4 CONFIGURATION COMPLEXITY .....	15
2.5 COMPLEXITY CLASSES .....	15
2.5.1 Complexity class <i>P</i> .....	15
2.5.2 Complexity class <i>NP</i> .....	16
2.5.3 Complexity class <i>EXP</i> .....	16
2.5.4 Complexity class conclusion.....	16
2.5.5 <i>NP-hardness</i> .....	16
2.6 OPERATIONAL CONTEXT (GENERAL).....	17
2.7 NON-FUNCTIONAL REQUIREMENTS IN GENERAL .....	18
<b>3 Problem definition .....</b>	<b>19</b>
3.1 REQUIREMENTS .....	19
3.2 TASK BREAKDOWN.....	19
3.3 QUESTIONS .....	20
<b>4 Principles for reasoning .....</b>	<b>21</b>
4.1 LOGIC LANGUAGES .....	21
4.2 MODAL LOGIC AND TEMPORAL LOGIC .....	21
4.3 SYMBOLIC REASONER .....	21
4.4 SOUND AND MONOTONIC LOGICAL REASONING .....	22



4.5	MODEL CHECKING .....	23
4.5.1	<i>Model checking approaches</i> .....	23
4.5.2	<i>Logical languages in model checking</i> .....	24
4.6	PROBABILISTIC REASONING.....	25
4.7	COGNITIVE REASONING .....	26
4.7.1	<i>Representational Systems</i> .....	26
4.8	TRADE-OFFS .....	27
4.9	COMPLEXITY REDUCTION.....	27
4.9.1	<i>Exploiting fuzziness with qualitative adaptation policies</i> .....	27
4.9.2	<i>Combinatorial testing</i> .....	28
4.10	HYBRID EVIDENCE .....	28
4.11	PRE-PROCESSING: CONFLUENCE ANALYSIS OF SEQUENCE DIAGRAM ASPECTS .....	30
4.11.1	<i>Confluence</i> .....	30
4.11.2	<i>Complexity reduction</i> .....	30
4.11.3	<i>Outlook for confluence and termination analysis of aspects in DiVA</i> .....	31
4.12	FEATURE MODEL.....	31
4.13	ONTOLOGIES.....	32
<b>5</b>	<b>Approaches &amp; Algorithms .....</b>	<b>34</b>
5.1	OPTIMISATION STRATEGIES .....	34
5.1.1	<i>Assumptions</i> .....	34
5.1.2	<i>Correctness &amp; precision</i> .....	34
5.1.3	<i>Alternative reasoners</i> .....	35
5.2	ADAPTIVE MODEL CHECKING .....	36
5.3	LESS EXPRESSIVE BELIEF NETS .....	36
5.4	PRISM .....	37
5.5	RETE ALGORITHM.....	37
5.6	LOGICAL REASONERS.....	38
5.6.1	<i>SAT solver</i> .....	38
5.6.2	<i>CSP solver</i> .....	39
5.6.3	<i>BDD/ ROBDD data structures and CUDD solver</i> .....	39
5.7	BACKTRACKING ALGORITHMS FOR CASE-BASED REASONING .....	42
5.8	SIMILARITY AND CASE-BASED REASONING .....	42



- 5.9 ARTIFICIAL LIFE ..... 44
  - 5.9.1 *Artificial Neural Networks*..... 44
  - 5.9.2 *Evolutionary Computation*..... 45
- 5.10 PARTITIONING OF REASONING – DIVIDE & CONQUER ..... 49
- 5.11 FAMA ..... 51
- 6 Conclusion and recommendation ..... 53**
  - 6.1 CONCLUSION ..... 53
  - 6.2 EVALUATION CRITERIA ..... 53
    - 6.2.1 *Complexity reduction*..... 53
    - 6.2.2 *Maturity* ..... 54
    - 6.2.3 *Documentation* ..... 54
    - 6.2.4 *Software*..... 54
  - 6.3 EVALUATION ..... 54
  - 6.4 RECOMMENDATIONS ..... 58
    - 6.4.1 *Features and feature value ranges* ..... 58
    - 6.4.2 *Data structures* ..... 59
    - 6.4.3 *Frameworks*..... 59
    - 6.4.4 *Algorithms* ..... 59
    - 6.4.5 *Pre-processing*..... 60
- Glossary ..... 61**
- References..... 65**
- Appendix A - DiVA requirements specification - reasoning specific..... 71**

## Version History

Version	Description	Date	Who
0.1	Document Initialization.	08.07.2008	Michael Hentze
0.2	Outline creation.	18.07.2008	Michael Hentze
0.3	Temporarily deleted references fields.	28.07.2008	Michael Hentze
0.4g	Introduction written, Fundamentals in progress, Problem description finished (except related work).	26.08.2008	Michael Hentze
0.4u	INRIA contribution added (evolutionary algorithms).	28.10.2008	Freddy Muñoz, Michael Hentze
0.4y	SINTEF contribution added (confluence analysis).	30.10.2008	Roy Grønmo, Michael Hentze
0.5	First preview version, still without Chapter 'Conclusion & recommendations'.	10.11.2008	Michael Hentze
0.6	Requirements by Thales/ CAS interpreted, D&C added, smaller overall fixes.	11.11.2008	Dhoua Ayed, Thomas Genßler, Michael Hentze
0.7	Chapter 5 started, Ontologies added.	24.11.2008	Nelly Bencomo, Michael Hentze
0.7p	References substituted.	25.11.2008	Michael Hentze
1.0	Final version for review.	26.11.2008	Michael Hentze
1.0a, 1.0b	Minor corrections and additions.	27.11.2008	Michael Hentze
1.1	Revision past first review; rearrangement of some sections, requirements update, added complexity class section.	16.01.2009	Michael Hentze, Vegard Dehlen
1.2	Revision past second review, added evolutionary algorithms discussion.	28.01.2009	Michael Hentze, Benoit Baudry, Freddy Muñoz
1.2a	Minor corrections (spelling, formatting).	04.02.2009	Michael Hentze
1.3	Minor edits after approval.	17.02.2009	Arnor Solberg Michael Hentze
1.4	Appendix A updated to be aligned with the D6.1 updated requirements list	17.03.2009	Arnor Solberg

## Contributing partners



**pure-systems GmbH**  
Agnetenstraße 14  
39106 Magdeburg  
Germany

**Michael K. W. Hentze**  
[michael.hentze@pure-systems.com](mailto:michael.hentze@pure-systems.com)

**Danilo Beuche**  
[danilo.beuche@pure-systems.com](mailto:danilo.beuche@pure-systems.com)



**IRISA Rennes**  
Campus de Beaulieu  
35042 Rennes Cedex  
France

**Freddy Muñoz Ramirez**  
[freddy.munoz@irisa.fr](mailto:freddy.munoz@irisa.fr) [mailto:](#)

**Reviewer: Benoit Baudry**  
[bbaudry@irisa.fr](mailto:bbaudry@irisa.fr)



**SINTEF**  
Strindveien 4  
7034 Trondheim  
Norway

**Roy Grønmo**  
[roy.gronmo@sintef.no](mailto:roy.gronmo@sintef.no)

**Reviewer: Vegard Dehlen**  
[vegard.dehlen@sintef.no](mailto:vegard.dehlen@sintef.no)



**Lancaster University**  
Lancaster LA1 4YW  
England

**Nelly Bencomo**  
[nelly@comp.lancs.ac.uk](mailto:nelly@comp.lancs.ac.uk)



**Thales**  
Route départementale 128  
91767 Palaiseau Cedex France

**Dhouha Ayed**  
[dhouha.ayed@thalesgroup.com](mailto:dhouha.ayed@thalesgroup.com)



**CAS Software AG**  
Wilhelm-Schickard-Straße 10-14  
76131 Karlsruhe  
Germany

**Thomas Genßler**  
[thomas.genssler@cas.de](mailto:thomas.genssler@cas.de)

## List of figures

FIGURE 1 INTERFACE BETWEEN STAGES OF THE DIVA METHODOLOGY [24].....	10
FIGURE 2 EXAMPLE ADAPTIVE SYSTEM REGARDED FROM THE REASONING PERSPECTIVE. ....	12
FIGURE 3 SIMPLE CONFIGURATION-SPACE SIZE EXAMPLE. ....	15
FIGURE 4 FUZZY CONTROL: CALCULATION OF A SERVER'S CACHE SIZE DEPENDANT FROM SERVER LOAD [16]. ....	27
FIGURE 5 HYBRID EVIDENCE PRINCIPLE IN MODEL COMPARISON [83].....	29
FIGURE 6 FEATURE MODEL AS OF [22] FOR A HOME INTEGRATION SYSTEMS (HIS).....	31
FIGURE 7 LEFT: BINARY DECISION TREE FOR A BOOLEAN FUNCTION. RIGHT: ITS BDD [94]. ....	40
FIGURE 8 DECOMPOSITION TREE IN THE D&C APPROACH. ....	50
FIGURE 9 FAMA ARCHITECTURE.....	51



## List of tables

TABLE 1 EVALUATION OF APPROACHES I .....	55
TABLE 2 EVALUATION OF APPROACHES II.....	56
TABLE 3 EVALUATION OF APPROACHES III .....	57

# 1 Introduction

The DiVA project addresses challenges of dynamic variability in complex, adaptive software systems. Such systems are typically deployed on heterogeneous, distributed platforms.

This document is a survey of reasoning techniques in the context of dynamic adaptation. This chapter offers a brief subsumption of DiVA's motivation; a deeper insight can be found in DiVA's **Description of Work** document [25], further on referenced as **DoW** and in the project's kick-off slides [26]. Chapter 2 assembles fundamentals of the system context that is to be reasoned about, followed by the problem definition in chapter 3. The centre of gravity is put on chapter 4 which offers an overview of a wide range of reasoning principles and algorithms. The conclusion in chapter 6 discusses and recommends approaches that are to be considered for the reasoning adaptation framework planning and implementation.

## 1.1 Project motivation and goals

Technological systems nowadays assemble an increasing number of devices that need in different degrees be adaptable to their environment. This implies their software must be able to recover context changes and can react on them. The overall, heterogeneous conglomerate is a collection of components that offer different resources and that are sinks for resources as well. Components may operate autonomously or be operated by a human user. A motivating example for a complex technological system is a crisis management operation system [25, p.7], where many different services and needs must be balanced and fulfilled. Some components of such a system (databases, crisis team central) are located at fixed places with constant environmental conditions; others (ambulances, crisis teams, satellites) are mobile and must operate in possibly rapid changing, sometimes limiting conditions. Rescue teams working in parallel time but on different locations use diverse technical devices and need specific information at distinct points of time, while their operational surrounding behaves dynamically as well as the devices they work with. Furthermore, single teams' goals usually change over time: they may start with rescuing people, then commence with fire fighting, and finally decide to perform evacuation. Each phase requires specific technical devices support, which causes unforeseeable variability in the overall system.

Managing the operational variability between components is a non-trivial task because of the exponentially increasing amount of possible feature combinations and the complexity explosion of device interactions, as declared in [25, p.8] and [26, fol.14]. Mastering this problem is a central research subject of the project; DiVA's approach is to regard variability dimensions as **crosscutting concerns** and model them as aspects. **Functional and adaptation concerns** captured in models help to break down the large system into clearly laid out parts that are easier to handle. The **separation of concern** paradigm demands for different types of models and levels of model abstractions throughout the system lifecycle which consists of analysis, design, and runtime.

A runtime model holds a component's configuration and state as well as its environment parameter settings. Runtime models are transformed and processed in real-time to calculate an overall state that optimally fulfils global and local system goals, where the resulting system configuration ideally represents the best adapted specificity of many possible variants. Accordingly, the goal stated for DiVA is to "provide a new tool-supported methodology with an integrated framework for managing dynamic variability in adaptive systems" [25, p.6]. The approach is cast to handle a system of co-existing, co-dependant configurations and to master the adaptive software's complexity in an innovative combination of aspect-oriented and model-driven techniques.

## 1.2 Reasoning framework

DiVA project work is organised in eight **work packages (WPs)**, detailed and enumerated as WP1 to WP8 in DoW. Relating to the technical task of reasoning on configurations surveyed in this document, these four WPs are the most relevant ones (arranged alphabetically):

- WP1 - Requirements analysis techniques
- WP2 - Model transformation framework
- WP3 - Adaptation, Models@runtime
- WP4 - Adaptation reasoning framework, Validation

Connections between WPs are illustrated in Figure 1. While WP1 delivers requirements and analysis as a formulation of the system goals and of the environmental context, WP2 creates an operational system model. WP3 handles the management of runtime models using adaptation aspects, supported by the notion of design-by-contract.

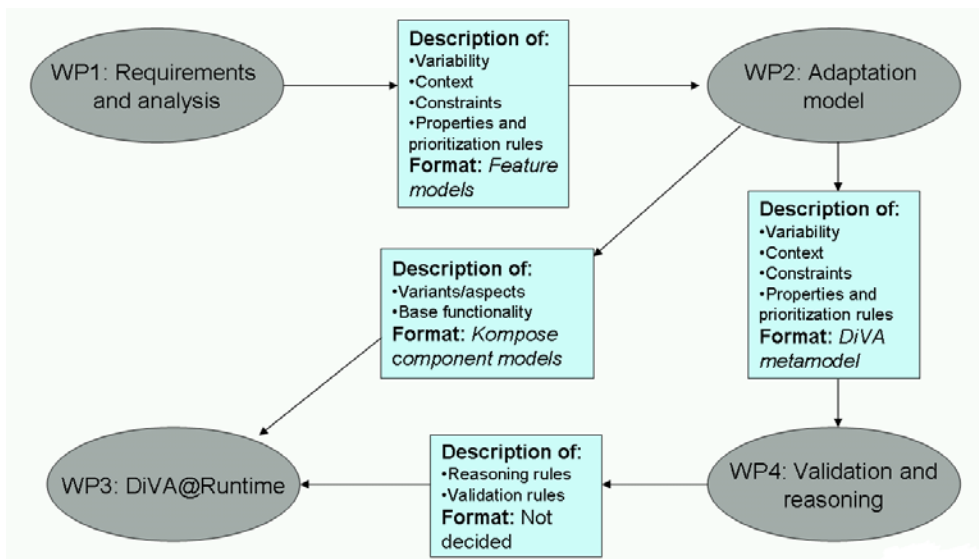


Figure 1 Interface between stages of the DiVA methodology [24].

This deliverable surveys techniques (algorithms, data structures, and frameworks) for the **adaptation reasoning framework (ARF)** to be developed by WP4 for a fast and reliable configuration adaptation based upon the operational model. Techniques implemented later must be able to reason at runtime on the model information to find a fitting overall system configuration in an effective way. Particularly, the combinatorial explosion of possible system configurations must be confronted.

In a later project phase, **validation** techniques will be researched to check whether a specific system configuration fulfils the requirements. In other words: reasoning finds configurations of predefined properties based on the runtime system characteristics of the modules delivered by WP1 - WP3, as requested in [25, p.50]; validation evaluates the configurations found against the requirements.

### 1.3 Objectives of the survey

The DiVA reasoning framework schedule does not imply to develop a novel class of reasoning methods but to adopt, adapt or combine from existing reasoning approaches. This survey therefore presents basic reasoning knowledge and studies existing adaptation reasoning mechanisms together with already available surveys as is established in DoW [25, p.51].

## 2 Fundamentals

This chapter defines and clarifies terms and concepts that are used throughout the survey. It may well be skipped by a reader who is already immersed in the DiVA project's background.

### 2.1 Adaptive, distributed software system

An adaptive software system as is being investigated upon in DiVA consists conceptually of subsystems; single devices represent co-existing, co-dependant lower-level systems that are individually configured. These low level systems may in addition be deployed on remote platform.

A simplified example view on an adaptive DiVA system is given in Figure 2: it consists of three instances of two software component types, where two instances interact by relationships. A component exposes and requests resources. Each component type is represented by its unique feature model (where its properties are visualised by equal numbers of nodes) that acts as the base for adaptation reasoning. However, each feature model is individually configured per component instance (**agent**) at runtime (visualised by equal vs. differing node fill patterns). The example is roughly an analogy of the description in [60]. Dependency relationships arise through resource offers and requests. In addition, system context triggers components to reconfigure. Slightly denoted are priorities, e.g. for adaptation orders, and dependencies between components as well as adaptation rules.

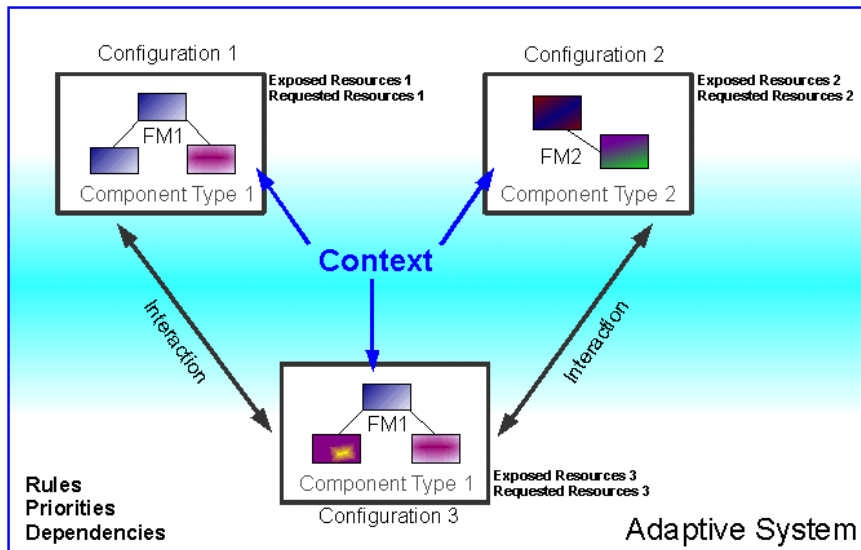


Figure 2 Example adaptive system regarded from the reasoning perspective.

## 2.2 System behaviour

During several subsystem's individual reaction to environment changes, overall system behaviour is influenced in an interdependent and therefore complex, generally not foreseeable manner. Changes in environmental parameters often influence a device's capabilities to fulfil its task, while a maximum service quality is desired for the single device as well as for the system as a whole which is built to optimally fulfil a human-specified task. If single devices are mobile, it is likely that they are influenced by uniquely, ever changing environment conditions. Mobility also imposes **constraints** on devices' operation modes for reasons as battery life or network availability; factors as human readability of text on a display due to lighting conditions imply operational requirements. To increase complexity even more, single devices will usually connect and disconnect to the overall system arbitrarily, thus consuming resources, potentially offering resources that could be useful for other devices. Not to forget about security constraints that must be reflected on data transmission channels for example via encryption.

As an example, a single device could be a laptop with large memory that has cached some data being required by a mobile phone, which actually is short of electric power and thus is being prevented from accessing the complete data set over a fast network channel. Under these conditions, the mobile phone's software decides to connect to the nearby laptop via a slower and less power consuming Bluetooth connection to access only a data subset as a trade-off. As another example, a single device could be a sensor which's simple task is to supply systems with context information, to enable a mobile phone to switch to higher display intensity.

Besides the next paragraph, another source of information is [61] as a result of WP3 which gives an overview of what validation means in the case of an adaptive system.

## 2.3 Knowledge based systems and Reasoning

A **knowledge based system** separates the problem descriptions (rules) from the problem solving mechanisms [9, p.11]. Instead of implementing e.g. hard coded if-then rules it is realised as e.g. a database of predicate calculus rules that are interpreted dynamically together with knowledge given as facts. The knowledge base can be divided into the following cases according to [9, p.17]

- **Case-specific knowledge**, which is evident knowledge about one specific instance [9, p.76], e.g. straight observations or deduced knowledge about specific situations.
- **Rule-based knowledge** that contains domain-specific and therefore more global knowledge or other general knowledge like optimisation rules and heuristics for solving problems.

**Inference**, the act of reasoning, is classically split into three types [9, p.23ff] and is here explained by example:

- **Deduction**: performed for example as logical inference, results in safely inferred knowledge: applying the rule that a device needs power to operate on WLAN together with the fact that the device's battery is empty leads to the result that the device cannot run WLAN.
- **Induction**: observing periodically the fact that a device doesn't connect to WLAN as well as that its battery is nearly empty may lead to the new rule that a device cannot operate WLAN with an empty battery. This inferred knowledge is, however, unsafe: the cause for not connecting could also be the absence of a WLAN network in all observed situations.
- **Abduction**: knowing a device with empty battery cannot run WLAN and observing that a device doesn't connect to WLAN leads to the explanation that its battery is empty. This inference is also not sure to be true in all cases, because another cause like antenna damage could be the source of trouble.

**Monotonic reasoning** depicts knowledge that is increasing monotonic during a reasoning process - classical deductive logics reason that way [9, p.27]. The more flexible **non-monotonic reasoning** is, similar to human reasoning, able to revise earlier knowledge and is more complicated to command, because knowledge generated as results of earlier deduction steps will usually be influenced by now obsolete base knowledge that was formerly regarded as valid.

## 2.4 Configuration complexity

A **variant** is a single feature selection of a system. Variants can be applied for a variation point. A **product** is one unique occurrence out of a space of possible system configurations created from a model. A **variation space** is spanned by a model's features dimensions. **Products amount** is determined by the dimensions of the space and by the value range that exist for each feature or property.

A simple example is visualised in Figure 3: a component has 3 variation points with 2 possible variants which means each feature can individually be switched between two states or, in other words, the parameter value range for each feature is two. As a result,  $2^3 = 8$  configurations exist. If we have 20 binary features,  $2^{20}$  (about 1 million) configurations can be built; if we have 20 features with 5 settings each, these are  $5^{20}$  (about 95.000 billion) which are surely much too many to evaluate them all brute force<sup>1</sup>. The number of configurations increases exponentially with the common parameter value range as the base of the exponential function.

Together with preconditions given by rules and dependencies, priorities, and context information, many of these potential configurations are invalid. Furthermore, some that are valid will usually be less useful than others, according to an assessment function. This causes, as a problem to be solved, high reasoning costs for configurations in big systems, because configurations need to be evaluated.

The combinatorial blow-up of the state-space or configuration space<sup>2</sup> of a system is known as the **state explosion problem**. Ways must be worked out to handle the complexity explosion and to enable adaptation reasoning in DiVA to find fast and secure an appropriate configuration.

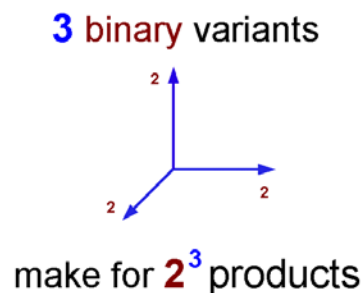


Figure 3 Simple configuration-space size example.

## 2.5 Complexity classes

As a tool for judging the complexity of algorithms, complexity theory from the area of theoretical computer science offers the concept of complexity classes (see [100]). An introduction into complexity theory is out of the scope of this survey. However, a brief overview is given in the following.

### 2.5.1 Complexity class $P$

$P$  is defined as follows for functions [101]:

---

<sup>1</sup> more examples of complexity in the context of adaptive systems may be found in [ScRo2008, p.2f], and [Flo2006]

<sup>2</sup> as each variant can be regarded as a system state

algorithm  $A$  calculates a function  $F$  in polynomial time, if a constant number  $k \geq 1$  exists that for every  $n$   $A$ 's running time is limited by  $O(n^k)$ .

An analogous definition for class  $P$  for deciding sets is:

algorithm  $A$  decides a set  $S$  of size  $n$  in polynomial time, if a constant number  $k \geq 1$  exists that for every  $n$  the  $A$ 's running time is limited by  $O(n^k)$ .

Calculating a function can be mapped to deciding a set. Therefore, in the following just the definition for deciding sets is given.

### 2.5.2 Complexity class $NP$

**NP** is defined as follows [101]:

A set  $S$  is in class  $NP$  if for every input  $x$  is valid that  $x \in A$ , exactly when the following conditions are fulfilled:

- A 'solution'  $y$  exists, which's length  $|y|$  is polynomial time in the length of  $x$  (this mean:  $|y| \leq p|x|$ ) where  $p$  is a polynomial).
- It can be checked in polynomial time whether  $y$  is a valid 'solution' for  $x$  (this means  $(x, y) \in B$  and  $B \in P$ ).

$NP$  designates non-deterministic polynomial time. The before defined complexity classes were deterministic so that at each point of time in calculation the next step is determined. For non-deterministic algorithms, one or several possible next steps exist.

Such, a set  $S$  is in  $NP$  exactly if a non-deterministic program exists that accepts  $S$  in polynomial time.

### 2.5.3 Complexity class $EXP$

**EXP** is defined as follows [101]:

$EXP := \{S: \text{a polynomial } p \text{ exists so that } S \text{ can be decided in } 2^{p(n)}\}$

### 2.5.4 Complexity class conclusion

$P \subseteq NP \subseteq EXP$

$P$  is the class of sets that can be decided efficiently. Sets that are not in  $P$  are not efficiently solvable.

### 2.5.5 $NP$ -hardness

In complexity theory, it is assumed that  $P$  is a proper subset of  $NP$  which means that in  $NP$  sets exist which are not decidable in polynomial time. Such, the hardest sets from  $NP$  are not located in  $P^3$ .

---

<sup>3</sup> to figure out here how to compare hardness of sets resp. problems is out of the scope of this document; this can be learned in [100]

## 2.6 Operational context (general)

An operational context may, seen from the view of the author, be divided in these sub contexts:

- **Social context** - the user's physical surrounding that affect her and that results in desired targets.
- **System context** - the platform infrastructure that runs applications technically.

An operational context of a complex, adaptive software system is influenced by the following factors, which were deduced from [26, fol. 14] and [25, p.8]:

- **Distribution**: software is deployed on distributed platforms.
- **Stationary visa mobility**: usually, few devices are stationary. Some or all components may be mobile devices.
- **Heterogeneity**: platforms usually run on computing devices of different hardware and operation systems.
- **Object driven or functionality driven**: action arises through interaction triggered by objects or through functionality requests.
- **Fluctuations in social context** appear overtime (e.g., working in a fire area versus driving the ambulance).
- **Fluctuations in system context** appear (e.g., low network connection vs. high network connection).
- **Asymmetry**: computation power and targets differ significantly between subsystems.
- **Clusters versus silos**: central data storage with unidirectional data flow versus data distribution.
- **Self-organisation**: the system's components interact to organize themselves.
- **Decentralisation**: aim is to (mostly) prevent central nodes but to force networking
- **Open and networked**: components interact via open networks.
- **Autonomous**: some or all devices may also work stand-alone

## 2.7 Non-functional requirements in general

*Quality of Service (QoS)* as an important non-functional requirement measures how well system properties are optimised in regard to predefined targets. These properties are the following [26, fol.14]:

- *Security*
- *Robustness*
- *Availability*
- *Safety*
- *Reliability*
- *Performance*

## 3 Problem definition

This chapter subsumes the problem definition that is treated as a whole in DoW.

### 3.1 Requirements

The DiVA requirements developed so far, interpreted for the reasoning framework, are appended to this document (see the DiVA requirements specification in Appendix A - DiVA requirements specification - reasoning specific). Requirements codes are marked green in the following section.

In a subsumption, time-efficiency is necessary (R-18). Language conversions will arise from other WPs to the ARF (R-9) and common interfaces are needed (R-10). Reasoning will be performed over non-functional aspects like QoS (R-25) and additional constraints. Dependencies and cardinalities are specified (R-14) and must be reasoned upon together with adaptation concerns. Support for dynamic reasoning (R-15) is the central reasoning task together with support of variability verification (R-19), and variability validation (R-23) for the later validation tasks.

The collected context information by sensors is part of reasoning. A central concern is treating the complexity explosion problem (R-11). The technical infrastructure, mainly the Eclipse Java IDE, must be supported (R-12). Ease of use is a less important aspect for reasoning as it is hidden to the user (R-13) while still documentation must be done well, while compliance to standards (R-16) has not been defined in detail yet. Functional and non-functional properties defined by designers are interpreted during reasoning (R-20, R-21) as well as required configurations (R-22).

### 3.2 Task Breakdown

A non-centralized, adaptive DiVA system as a 'system-of-systems' is considered to be a complex combination of co-dependant, co-existing configuration interactions. A subsystem consumes resources at runtime like memory or bandwidth, which are in practise limited for small, mobile devices. During adaptation to a dynamic system, devices' configurations change and cause additional resources consumption to perform reconfiguration.

A DiVA software system's goal is determined by a set of requirements capturing variability and adaptation rules, which are expressed as components together with their properties, their dependencies, adaptation priorities and other priorities, behaviour of the context they live in, and rules. Requirements are formalized as an operational model which functions as a base structure for invoking variability. They cause, together with varying context parameter assignments, generally many possible product system configurations (**variants**) with the amount of features (formalised as requirements) exponentially increasing the number of variants<sup>4</sup>. A resulting configuration can be better or worse adapted to certain non-functional requirements which are to be expressed by e.g. qualitative evaluation functions.

*Reasoning* is the mechanism of calculating a valid system configuration from a knowledge base which consists of model features, attributes, dependencies, and constraints together with additional rules about context, interaction and so on throughout DiVA. The Adaptation Reasoning/Validation Framework that has to be invented is required "to reduce the testing effort needed to detect and validate the properties of the whole system that cannot be ensured at the model level" [25, p.51], whereas in the first phase adaptation reasoning is the central task of WP4. Reasoning is about features and the following **constraints**:

---

<sup>4</sup> according to [MuBa2008], a varying weaving order of aspects creates additional system variants

- **Dependencies,**
- **Properties,**
- **Priorities,**
- **Context information, and**
- **Rules.**

A brute and therefore computationally expensive way to calculate a well adapted system configuration is to build and evaluate all system configurations and choose the one with the best outcome, respectively kick out all inadequate configurations one by one [35, p.44]. However, due to limited time resources because of aspired real-time characteristics, efficient reasoning mechanisms and surveys on this topic must be researched by WP4. Approaches must be detected to master the variability explosion during rating configurations with predefined properties. The central question is in this regard: how to avoid inefficiency in time or in memory consumption during reasoning?

If reasoning is *rule based* on one hand, the problem arises that it scales badly with the number of context elements and the number of variation points. The resulting combinatorial explosion excludes a simple brute force approach from practicality. If reasoning is based on functions that express the outcome of a configuration via **utility functions** on the other hand, an adapted configuration must assumedly be created before each evaluation, which again would not be feasible due to high calculation costs. A memory-based reasoning imposes limits because of its memory consumptions. So a scalable, heuristic innovative solution must be created, whereby scalability may affect the *granularity* of system adaptation facility. The effort starts by surveying existing work in this field, presented in this document. The concrete structure of the runtime models is still emerging at the time of writing this survey; therefore, different families of reasoning mechanisms are considered.

### 3.3 Questions

The survey treats questions as:

- How to avoid or reduce complexity in reasoning?
- How can fuzziness in reasoning improve time-effectivity?
- What is the state of the art in reasoning on configurations overall?
- How to reason in DiVA's specific circumstances on co-existing, co-dependant configurations?

## 4 Principles for reasoning

### 4.1 Logic languages

For simple **Propositional Calculus** languages presentable as **Horn clauses**<sup>5</sup>, efficient algorithms for reasoning exist [41, p.13] that perform the reasoning task in linear time, e.g. based on **PROLOG**. Propositional Calculus is correct/ sound, specific and time-efficient, but it does not allow for individual items and as such has a limited expressiveness<sup>6</sup>. Therefore, it is questionable if Propositional Calculus languages are sufficient for the DiVA environment. However, they may be adequate for limited problem domains.

**First Order Predicate Calculus (FOPC)** is being implemented in e.g. **Description Logics (DL)** languages. DLs extending semantic nets offer a higher expressiveness as Propositional Calculus at the general price of NP-hardness and perhaps undecidability if not adequately constrained. Thus, a potential DiVA language's expressiveness presumably needs to be reduced to stay below undecidability and to be able to perform time-efficient reasoning. For DLs extending semantic nets which are 'as expressive as possible' polynomial algorithms exist.

### 4.2 Modal Logic and Temporal Logic

Emerson [30, p.2] explains **Modal Logics (ML)** as a class of logics to represent modes of truth. A mode can e.g. be the term 'possibly': an assertion  $\alpha$  in ML may be false in one world and still can be possibly true, if an alternate world exists where an assertion 'possibly  $\alpha$ ' is true.

**Temporal Logic (TL)** allows time-dynamic reasoning in formal systems. TLs are MLs developed for qualitatively describing how the truth values of assertions change temporal. In a TL, temporal operators are provided as **modalities** to describe the variation of assertion truth values with time. As an example, a temporal operator may include 'sometimes  $\alpha$ ' which is true now if there is a future moment at which  $\alpha$  becomes true and 'always  $\alpha$ ' which is true now if  $\alpha$  is true at each future moment. According to [30], "TL has been used or proposed for use in virtually all aspects of concurrent program design, including specification, verification, manual program composition, development, and mechanical program synthesis. In order to support these applications a great deal mathematical machinery connected with Temporal Logic has been developed".

### 4.3 Symbolic Reasoner

Symbolic reasoning is the ability to reason on symbolic information given in a symbolic language and to analyze and present symbolic information to reach valid conclusions. A symbolic reasoner has the ability to understand the logic and validity of an argument through analyzing the relationships of components.

A **symbolic reasoner** is in the following identified with its **symbolic knowledge base SKB** [41, p.2]. The adaptation reasoning framework (**ARF**) feeds its **symbolic knowledge base (SKB)** with information through external queries  $\kappa$  for rules and  $\rho$  for dynamic information like changes in dependencies, properties, priorities, and context information in the form of  $\text{TELL}(\text{SKB}, \kappa)$  or  $\text{TELL}(\text{SKB}, \rho)$ . Such,  $\kappa$  holds information about the global or a local system state. The ARF can then answer questions  $\text{ASK}(\text{SKB}, \rho)$  or  $\text{ASK}(\text{SKB}, \rho, \mu)$  where  $\mu$  is a model that must be checked for validity by the DiVA@Runtime framework (and possibly other frameworks) and will, what depends on the interface definitions still to develop

---

<sup>5</sup> which only allows for conjunctions of disjunctions with a maximum of one positive literal

<sup>6</sup> as it owns no quantors  $\exists$  and  $\forall$  as well as cardinalities

during the work on DiVA, return answers that may be simple *true/ false* decisions or a *valid/ invalid* marked model.  $\kappa$  might also be a rule which implies that rules do not have to be constant as is the case in monotonic reasoning or hard-coded implementations.

A **symbolic encoding** that transports semantic contents offers some benefits over systems that implement perhaps numeric functions in neural networks [41, p.3]:

- A symbolic reasoner can more easily **explain** its calculations and decisions because of its already immanent semantics which allows result insights.
- **Construction** of the SKB is easier for the domain expert and more transparent to maintain and debug due to its human-readable nature.
- **Handling of partial knowledge** (in opposite to complete knowledge) is already inherent to symbolic logics like propositional logics and predicate calculus. Partial knowledge like "reason for not connecting to WLAN is either an empty battery or a low signal quality" can in these logics be represented using disjunction ' $\vee$ ', negation ' $\neg$ ' besides conjunction ' $\wedge$ ' or in more complex cases even quantors ' $\exists$ ', ' $\forall$ '.

Therefore, Greiner et al.'s survey concentrates on two categories of symbolic reasoning: **sound** and **monotonic** logical reasoning, and probabilistic reasoning.

#### 4.4 Sound and monotonic logical reasoning

Soundness guarantees truth-preservation. A sound reasoner's answers are mathematically proven to be correct on a SKB of precise, discrete statements. Reasoning is typically performed on **inference rules** that may be extended by **rewrite rules** to transform and simplify statements. Resulting statements can then be added to the SKB and such speed up calculations. However, the satisfiability problem of checking whether a model can be constructed that fulfils a given proposition or if a given proposition is a tautology is NP-complete: no general  $O(n)$ -algorithm exists to decide the problem, with  $n$  being the number of inputs [35, p.44]<sup>7</sup>, but heuristics may be applicable to the prize of soundness, e.g. the Horn Approximation reasoner mentioned in section 5.1.2. For DiVA, soundness is an important factor as only then reasoning results can be compared to requirements describing the system's aims.

In a dynamic system, a challenge for the Adaptation Reasoning Framework can be not only deciding about a valid, well known model but to work under potentially incomplete knowledge about the overall system due to unattainability of devices at runtime or to calculation time limitations.

Sound deductive techniques exist in many flavours like propositional logics, predicate calculus, and higher order logics as well as natural deduction, ONTIC and equational logic (the latter are discussed briefly in [41, p.3f]). Most of these systems use **resolution** which has been proven by Robin [72] of being able to represent any first order calculus inference.

---

<sup>7</sup> "The above method for testing whether a proposition is satisfiable or a tautology is computationally expensive, in the sense that it takes an exponential number of steps. One might ask if it is possible to find a more efficient procedure. Unfortunately, the satisfiability problem happens to be what is called an NP-complete problem, which implies that there is probably no fast algorithm for deciding satisfiability. By a fast algorithm, we mean an algorithm that runs in a number of steps bounded by  $p(n)$ , where  $n$  is the length of the input, and  $p$  is a (fixed) polynomial." [35, p.44]

## 4.5 Model checking

In this section, foundations of evaluation are presented together with explanation of logical reasoning over semantic languages.

[30, p.46] explains the **model checking** problem as "Given a finite structure  $M$  and a propositional TL formula  $p$ , does  $M$  define a model of  $p$ ? For most any propositional Temporal Logic the model checking problem is decidable since we can do, if needed, an exhaustive search through the paths of the finite input structure". Model checking is a systematic way of checking whether all behaviours of a systems model fulfil its specification. It can be performed automatically given a system model and a set of properties. Model checking is a subtask of reasoning as well as a validation task as it can check a specific instance from the set of all possible product systems against a model definition and thus answers the question of a model's validity.

A brute-force model checking strategy for DiVA would be checking an enumeration of all or very many possible system configurations. However, a smart checking of a heuristically determined subset of all possible result systems can be an adequate approach. Highly optimised model checking algorithms and tools exist and research for more efficient algorithms is an actual interest of research in diverse areas of computer sciences. Model checking tools were initially developed to reason about the logical correctness of discrete state systems, but can now also deal with real-time and limited forms of hybrid systems.

### 4.5.1 Model checking approaches

These are some approaches<sup>8</sup> to combat the state explosion problem as explained in section 2.4:

- **Symbolic algorithms** do not build the graph for the **Finite State Machine (FSM)** but implicitly represent the graph by a propositional logic formula using binary decision diagrams (see details in section 5.6.3).
- **Bounded model checking** algorithms unroll the FSM for a fixed number of steps  $k$  and check whether a property violation can occur in  $k$  or fewer steps. This typically involves encoding the restricted model as an instance of SAT (see details in section 5.6). The process is repeated with increasing values of  $k$  until all possible violations have been ruled out<sup>9</sup>.
- **Abstraction** proves a system's properties by first simplifying it, e.g. ignoring the values of non Boolean variables and only considering Boolean variables. The simplified system usually does not satisfy exactly the same properties as the original one so that a process of refinement may be necessary. Generally, one requires the abstraction to be sound<sup>10</sup>; however, most often, the abstraction is not complete<sup>11</sup>. Such a coarse abstraction may in fact be sufficient to prove e.g. properties of mutual exclusion<sup>12</sup>.

---

<sup>8</sup> taken from [http://en.wikipedia.org/wiki/Model\\_checking](http://en.wikipedia.org/wiki/Model_checking)

<sup>9</sup> iterative deepening depth-first search

<sup>10</sup> the properties proved on the abstraction are true of the original system

<sup>11</sup> not all true properties of the original system are true of the abstraction

<sup>12</sup> abbreviated as **mutex**; mutex algorithms are used in concurrent programming to avoid the simultaneous use of a common resource

- **Counter-example guided abstraction refinement (CEGAR)** begins checking with a coarse (imprecise) abstraction and iteratively refines it. When a **counter-example (CE)** as a violation is found, the tool analyzes whether the violation stems from the original model (is feasible) or is the result of a still incomplete abstraction. If the violation is feasible, it is reported and model checking ends; if it is not, the proof of infeasibility is used to refine the model abstraction and checking begins again.

In section 5.2, a single model checking approach is figured out.

#### 4.5.2 Logical languages in model checking

Decidability is NP-complete or higher for expressive logics in general. However, Emerson states in his survey on analysing and classifying logics [30] with respect to the complexity of model checking: "For some logics, which have adequate expressive power to capture certain important correctness properties, we can develop very efficient algorithms for model checking". Therefore it makes sense to investigate on logical languages that perform efficiently and can express the model semantics that appear in DiVA. In the field of **semantic web languages**, logical reasoning approaches for model checking play an increasingly important role.

Some applications of **Temporal Logics (TL)** to program reasoning are:

- checking correctness properties of concurrent programs,
- verification of concurrent programs,
- mechanical synthesis of concurrent programs from TL specifications, and
- automatic verification of finite state concurrent systems.

A DiVA system may be regarded as a program with correctness constraints and can therefore potentially profit from TL approaches.

The computational complexity of different abstract reasoning approaches as theorem proving/satisfiability testing and model checking in practical applications plays a vital role for runtime efficiency. Authors of [48] compare complexity of reasoning in the context of **multi-agent systems** judging three important subsets of the cooperation logic **ATL**<sup>13</sup>. **Alternating-time Temporal Logic** [6] is a temporal extension of **Coalition Logic** that has "attracted much interest from the multi-agent systems community. Using such logics, it is possible to reason about the strategic powers of agents and coalitions of agents in game-like multi-agent scenarios". These requirements fit well to the DiVA scenario where devices can be regarded as agents competing for resources and cooperating for their sharing.

It is being shown that, against earlier assumptions at that time, at least for two important subsets of ATL and ATL itself, represented in **SRML**<sup>14</sup>, automatic model checking for ATL and Coalition Logic has the same complexity as the corresponding theorem proving. Theorem proving may additionally require a human user who gives hints to the system.

---

<sup>13</sup> ATL is a game-based temporal logic for specifying collaborative as well as adversarial interactions between components.

<sup>14</sup> "Simple Reactive Modules Language", a language used in practical model checking systems as SMV for CTL and MOCHA for ATL. SRML is a subset of RML, a high-level language for modelling service-oriented systems.

The authors show the bounds of EXPTIME on the computational complexity of these problems for a practical representation for models which is the same effort as for theorem proving via satisfiability checking, but not higher. Thereby they verify that model checking is not more inefficient in general than satisfiability.

Theoretically, the model checking problems for ATL and CL can be solved in polynomial time in the size of the given formula and the size of the model. However, this only holds for an ATL model represented as an explicit expressed enumeration of all system states in the exponential size  $2^n$  for  $n$  Boolean variables, which is not practical for big models and small devices as are expected in DiVA. Another drawback of an explicit representation is the fact that polynomial time only holds for a fixed number of agents, which violates the dynamic, adaptive nature of DiVA systems. Therefore, an explicit system representation should be prevented for reasoning.

Van der Hoek et al. [48] annotate that the SRML language is a strict subset of the RML language that is used by practical model checking systems, including **Mocha**, the ATL model checker [1], which is yet mostly useful for the validation part of DiVA and obviously not so much for adaptation reasoning<sup>15</sup>. SRML is arguably the smallest “useful” subset of RML: it cannot be simplified without making it unusable in practice. Guarded command structures used to define SRML models are also used for the same purpose in other model checking systems such as SMV (see section 5.6.3).

## 4.6 Probabilistic Reasoning

**Probabilistic Reasoning (PR)** is reasoning using quantitative probability distributions for reasoning about uncertainty. Situations like these may appear in a DiVA system if information is missing or cannot be calculated in a short time. Models are called 'probabilistic' as they encode the probability of making a transition between states and analysis usually means calculation of likelihoods through numerical or analytical methods. As a simple one-dimensional example, the cause probability for a device not connecting to WLAN can be expressed by a numeric rate of 70% for 'no network available' and a rate of 30% for 'battery is empty'.

Basic decision systems need an encoding of all possible system states, which causes exponentially many joint distribution values and makes them hard to implement for complex systems. Basic probabilistic reasoning is therefore NP-complete.

Probabilities are used successfully in engineering systems and medicine diagnostics, which are usually not adaptive systems but rely mostly on statistical measurable previous knowledge. A practical way to implement PR graphically is encoding them as a **belief net** (Bayesian net) or as **Markov net**. Running these nets carries the notion of a simulation that iteratively calculates a solution, e.g. the cause for a device failure. A graphical notation allows for efficient methods of calculation [9, p.360]. Such a system can be even more expressive than symbolic Boolean logic systems. To enable non-monotonic inference, PR can be combined with utility functions [41, p.6].

To set up PR systems, probabilities must exist as a database. To capture them, they must be measured or estimated. Building a global DiVA PR system will usually become complicated to specify and maintain as even experts will hardly be able to overview all probabilistic relations, the more as a dynamic adaptive system may evolve to formerly not invented states. Also, statistical measuring of liked or disliked overall product configurations will induce a hard up to impossible task, because the dynamic system needs to be e.g. simulated or measured first with the limitation that the system may later grow further than the simulated system has been.

---

<sup>15</sup> still it must be annotated that a model validation can be considered a subtask of reasoning as it checks a specific instance (potentially taken from the set of all possible product systems) against requirements

Additionally, size of probability distributions increases exponentially with the number of variables [9, p.360]. However, simplifying assumptions can help in some, but not all cases to reduce complexity. Also, approaches of **factoring** and **NoisyOr representation** which help to reduce redundancy in belief nets. As an example for reduction, [41, p.6] quotes work from [10] where the complete joint distribution for the ALARM belief net with 37 nodes and 47 arcs that causes ca.  $10^{17}$  parameters has been reduced to 752 parameters.

## 4.7 Cognitive reasoning

Cognitive science is the interdisciplinary study of mind and behaviour. It resides in a wide range of disciplines including physics, chemistry, biology, philosophy, and computer sciences. The last is the subject of our study. In particular, we are interested in **artificial intelligence (AI)** and **artificial life (AL)** as the main contributors to the cognitive body of knowledge.

In DiVA, we survey some cognitive approaches in order to solve problems related to decision, learning and optimisation in adaptive systems. Learning capability can help adaptive systems managing the emergency of configuration when adapting. Optimisation and decision techniques can help selecting the best adaptation for a given context or situation.

Early cognitive science was centred on the development of systems capable of representing and using knowledge. These systems called **representational** or **knowledge-based** can provide support in deciding the feasible adaptation for a system given a set of rules (**knowledge**) and environmental stimulus (**query**). Later on, cognitive scientists began to explore a new field, *Artificial Life*, which studies the different systems related to life and tries to reproduce them.

Among all its approaches, artificial life offers some **artificial learning** techniques. In particular, we survey artificial neural networks, which attempt to mimic the behaviour exhibited by natural neural networks. Artificial neural networks are capable of raising emergent behaviour produced by the interaction of its **neurons**. Such emergent behaviour could help handling unpredicted adaptations. Evolutionary computation is a part of artificial life that exploits natural evolution metaphors to solve optimisation problems. We later survey in section 5.9 **Genetic evolution** and **Swarm intelligence** (incarnated in **Particle swarm optimisation** and **Ant colony optimisation**) as methods to solve functional optimisation problems. These algorithms may help finding an optimal adaptation to a given context or environmental change, select a set of adaption for validation purposes or help learning an artificial neural network.

### 4.7.1 Representational Systems

In the early years of cognitive science until the mid 1980s, a majority of the research in AI was devoted to producing computer systems emulating or modelling high-level cognitive functions. Often, these functions were uniquely human, such as natural language interpretation, understanding and translation. The models produced in this stage were representation of facts and the relations between them in such a way that some degree of knowledge can be inferred and applied based on them [63; 15; 96]. These representations are often based on mathematical logic, in which inference logic rules are applied to infer knowledge from existing facts [47]. Symbol manipulation and logic-based programming languages such as LISP [95] and PROLOG [18] were created to support the development of rule-based systems.

In DiVA, representational systems can support the reasoning based on rules. Rules are represented by facts in a database (such as PROLOG), for instance *when low overhead is needed, it implies FTP or UDP communication (overhead low -> FTP or UDP)*. The decision of which adaptation to adopt is performed based on the existing rules (*knowledge*) through a query to the database.

## 4.8 Trade-offs

As a guideline for the understanding of reasoning approaches, mathematical provable impossibility must be accepted, that an ideal reasoning system cannot be constructed that produces "all-and-only the *correct* answers to every possible query, produce answers that are as *specific* as possible, be *expressive* enough to permit any possible fact to be stored and any possible query to be asked, and be (time) *efficient*." [41, p.1]. If improving e.g. expressiveness, a reasoning system can become inefficient or even undecidable. Therefore, the system to reason about must be a trade-off between the four aspects *correctness*, *specificity or precision*, *expressiveness*, and *efficiency*.

In the DiVA context, time efficiency is a hard constraint which reduces negotiation to the first three aspects. In improving reasoning speed in the goal conflict, worst-case time efficiency or average time efficiency may be optimised.

## 4.9 Complexity Reduction

### 4.9.1 Exploiting fuzziness with qualitative adaptation policies

The approach described in **qualitative adaptation policies** [16] uses fuzzy sets to reduce result configurations complexity. Usual sets are associated with strict membership; fuzzy sets are sets of which elements have **degrees of membership**. Result systems as answers to queries using properties settings may be calculated from a function using fuzzy rules as a combination of fuzzy predicates. Following steps are performed in the algorithm: a fuzzy control is applied on all existing rules. Then, values for re-configurations are computed. Subsequent, the most useful re-configuration is selected and applied if its utility is above a predefined threshold.

In the fuzzy logic used, the usual logical operators disjunction ' $\vee$ ', negation ' $\neg$ ' and conjunction ' $\wedge$ ' are extended by modifiers *very*, *slightly*, *moderately*, *above*, *below*. As an example, a server load may be *high* and *slightly medium*. Figure 4 shows the calculation of a server's cache size dependant from its load.

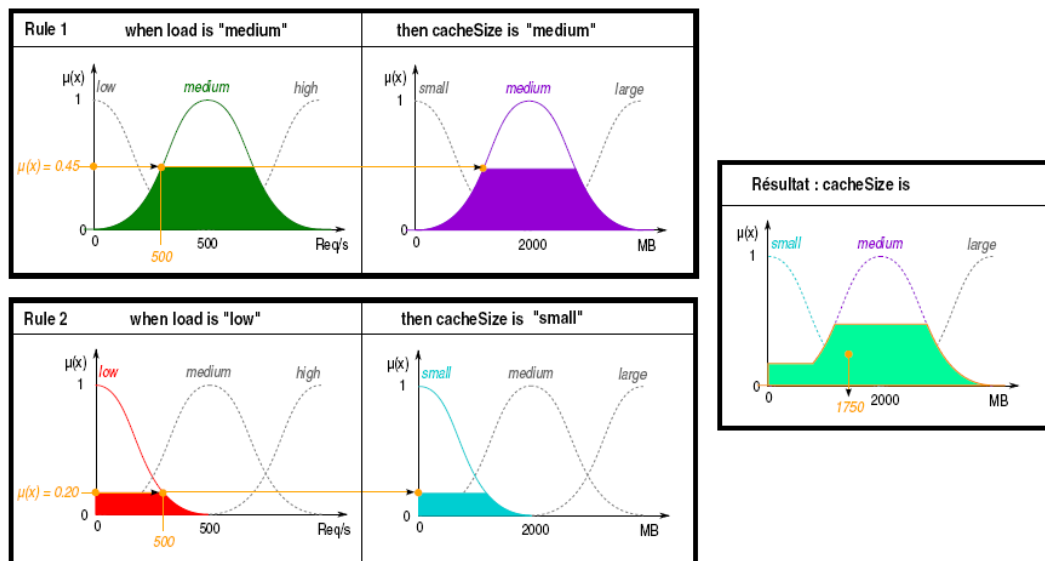


Figure 4 Fuzzy control: calculation of a server's cache size dependant from server load [16].

The approach has already been implemented for the *Fractal*<sup>16</sup> platform.

Fuzzy Control can be seen as a support for parameter space reduction in DiVA. It allows capturing designer knowledge but has the drawback that functions need to be carefully constructed, e.g. by experimental results on an existing system that is reasoned about in DiVA. Also, for higher-dimensional calculations functions can be hard to determine and thus to design.

#### 4.9.2 Combinatorial testing

When the input domain of a system consists of a large number of parameters, or can be configured with a large number of variables, it is usually impossible to test it with all possible combinations of parameters and variables. It is then necessary to apply heuristics or test criteria capable of reducing the number of test data and still guarantee the test coverage. Based on experiments suggesting that software faults often involve a limited number of interactions between variables, **combinatorial testing** [19; 55] proposes to generate **test data** that cover interactions between only 2, 3 or n variables. The generation of test data covering interactions between pairs of variables is known as **pair-wise testing** or **2-way testing** [88]. This is the most popular combinatorial testing approach because it has empirically been shown that it can detect a large number of errors. Furthermore, there exist efficient algorithms and tools that can automate the generation of test data satisfying a pair-wise criterion. However, recent experiments show that for discovering all errors in a system it is usually necessary to test between 4-way and 6-way interactions [55]. Cohen et al. have developed AETG to develop combinatorial testing for more than 2-way interactions [19].

Because the DiVA project aims in building systems that have large input spaces and that should adapt in large number of configurations, combinatorial testing can be an interesting approach for two reasons:

- First, the DiVA project could apply combinatorial techniques to reduce the number of configuration that have to be validated during reasoning.
- Second, the same techniques could be applied to select combinations of test data to test the selected configurations during validation.

#### 4.10 Hybrid Evidence

**Hybrid evidence** as is performed in the system verification/ comparison task of [83] builds a queue of reasoners that use specific reasoning approaches which are in this case simulation, ROBDD (see 5.6), ATPG (Automatic Test Pattern Generation), and SAT (Satisfiability of Boolean notations). A reasoner gets two input models to be compared; in each stage, the result can be 'false' (both the systems are different), true (the systems are identical) or 'open' (no decision yet) as shown in Figure 5. This principle can on one hand be used for verification of a result configuration against the requirements of WP1, although this is not the centre of this survey. What is on the other hand promising is this principle's potential to be exploited as a way to improve reasoning speed. A queue of different reasoners can each be given a time limit; if it can deduce 'true' or 'false' within time, a valid result is computed and reasoning aborts. If it cannot, the single reasoner aborts and the hybrid reasoning system tries to solve the task using another reasoner that may perform faster in certain situations, where additionally also single reasoner order may be changed by the system according to a-priori knowledge. In [83], each reasoner is intended to solve a subset of the overall reasoning task as its contribution to the final solution, so that not each reasoner starts at a 'no solution yet' point. Partitioning of reasoning into well-fit sub-domains may thus support computation complexity reduction. This is similarly mentioned

---

<sup>16</sup> Available at <http://fractal.objectweb.org/>

in [41, p.18]: if it is for example known that attribute values are conditionally independent, a naïve Bayes classifier is advised there.

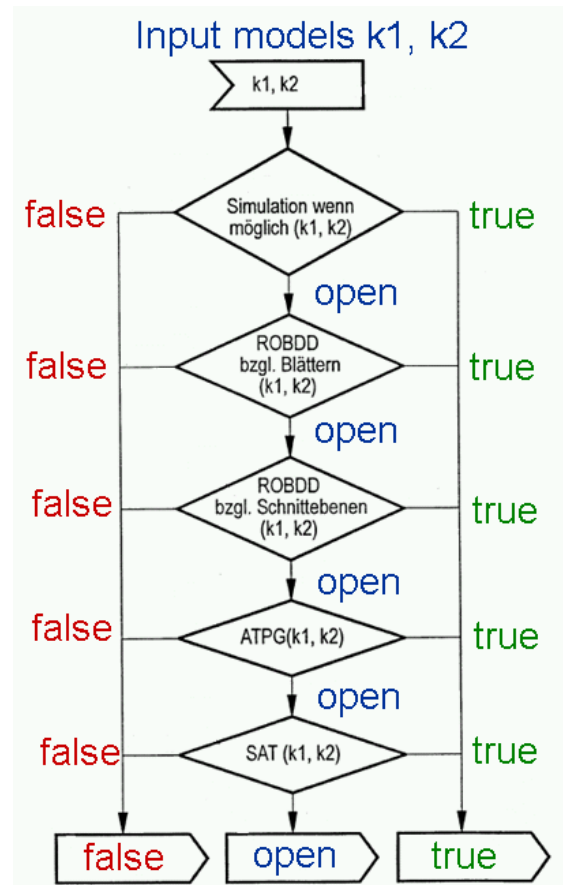


Figure 5 Hybrid evidence principle in model comparison [83].

## 4.11 Pre-processing: Confluence Analysis of sequence diagram aspects

In creation of WP3's runtime models that follow the Aspect-Oriented Modelling (AOM) paradigm, the problem arises that **aspects** must potentially be woven in an adequate order to result in a reproducible result model. Judging runtime models thus involves assuring a reproducible result dependant on the weaving order. This may help to reduce the amount of potentially weavable models that otherwise would be rejected as wrong by the ARF.

The last decade has seen several proposals for aspect diagrams for UML 2 sequence diagrams. Aspect diagrams allow the modeller to define crosscutting concerns/ aspects of sequence diagrams and have these woven with the sequence diagrams of a base model. In a real-world scenario there may be multiple aspects applicable to the same base model. This raises the need to analyse the set of aspects in order to identify possible dependencies and conflicts between applications of aspects.

### 4.11.1 Confluence

A set of terminating aspects that are **confluent** will always yield the same result when applied non-deterministically on the same initial model, i.e. a confluent set of aspects have no dependencies or conflicts between the aspects. Non-confluent aspects often mean that it is necessary to specify an explicit weave order, redesign some of the aspects, or exclude one or more aspects. An important property of a set of aspects is **termination**. A set of aspects is terminating if there exists no base model with infinite weaving of the aspects<sup>17</sup>.

There already exists a well-established theoretical foundation on **graph transformation systems (GTS)** and confluence [68]. The concrete syntax of sequence diagrams and aspects defined upon these are, however, quite different from graphs and GTS rules. This means that the GTS theory cannot be directly applied. Thus, Grønmo et al. [43] establish a specialized confluence theory based upon the concrete syntax of sequence diagram aspects.

### 4.11.2 Complexity reduction

This paragraph describes how a confluence analysis reduces the complexity of modelling rules and dependencies between variants/aspects. WP2 supports modelling of dependencies among aspects. As the number of aspects increases, we get a combinatorial explosion in the number of potential relationships between aspects. Such relationships include aspect application orders, that an aspect requires the application of another aspect, and that an aspect excludes another aspect. To reduce the effort needed by the modeller, a confluence analysis is helpful and we propose the following task order:

1. Define the individual aspects in isolation without bothering about the other aspects.
2. Perform a confluence analysis on the set of aspects. The output of this analysis will be a set of dependant aspect pairs.
3. Define aspect relationships for all the dependant aspect pairs and continue from step 2. The aspect definition process ends when we have full confluence and there are no more dependant aspect pairs.

The aspect definition process and the confluence analysis are best suited **prior to run-time**. This is because any non-confluence requires a user/ modeller to manually define the aspect relationships. If a confluence analysis is performed **in run-time**, then we must require that the chosen set of aspects is fully confluent in order to perform run-time weaving. If we have non-

---

<sup>17</sup> Termination theory could also prove to be a helpful tool in the DiVA project.

confluence, then the run-time configuration cannot be established with the chosen set of aspects, since we need human-intervention in order to define the necessary aspect relationships.

#### 4.11.3 Outlook for confluence and termination analysis of aspects in DiVA

Although we have only described confluence analysis for sequence diagram aspects, confluence analysis may also apply to other modelling languages to be used in DiVA including class diagrams, component diagrams, and state machines.

Sequence diagrams are different kinds of models than the graphs used in graph transformation. This requires a dedicated confluence theory for sequence diagrams, while many other modelling languages are much closer to graphs (like in graph transformation). The latter fact means that the existing confluence theory and associated tools from graph transformation may be directly applicable, although this remains to be investigated.

The confluence analysis for sequence diagram aspects rely on the computation of a set of extended **critical pairs** (not described here) and a weaving process upon these to see if all such critical pairs are joinable. Although this process is decidable, it remains to investigate the computational tractability and how many aspects the confluence analysis can handle in practice before we encounter performance challenges.

### 4.12 Feature Model

An overview of **Feature Models (FMs)** as the base representation for all potential products of a model is given in [12]. This paragraph assembles the basic concepts in the notation of Czarnecki [22] which are implemented by the **pure::variants** Eclipse plug-in<sup>18</sup>.

A **feature** is a prominent characteristic of a product system. In DiVA, features are components of the system architecture or their services. An **attribute** is a characteristic of a feature, e.g. battery consumption of a service if it is selected. Attributes reside in an **attribute domain** which holds the possible values, e.g.  $\{low, medium, high\}$  for battery consumption. **Extra-functional features** express a relation of one or between several attributes. Examples are *battery consumption = high* or *battery efficiency/ MB data = 0.1*. A **product** is a (hopefully valid) configuration dependant on its FM. The example of Figure 6 shows the features of a Home Integration System. Every feature may additionally have extrafunctional features that can differ in every product.

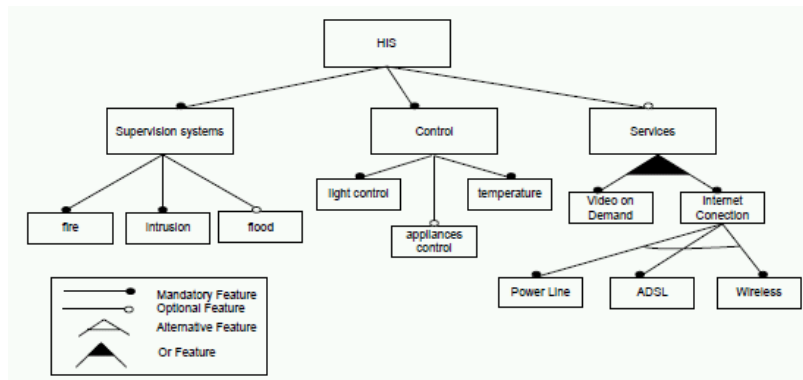


Figure 6 Feature Model as of [22] for a Home Integration Systems (HIS)

<sup>18</sup> available at <http://www.pure-systems.com/>

Four relations are defined in the FM notation: mandatory, optional, alternative, and or-relation:

- **Mandatory** features are selected if the parent feature is selected. They are then always present in the product. E.g., the Control subsystem always has a light control and a temperature control.
- **Optional** features are selected independently. They may or may not be present in the product. E.g., the Control subsystem might in addition have an appliances control.
- **Alternative** features are organised in groups. Exactly one feature has to be selected from a group if the parent feature is selected. The Internet Connection feature, for example, mandatory has one of the three specificities Power Line, ADSL or Wireless, but not more than one.
- **Or** features are organised in groups. At least one feature has to be selected from a group if the parent feature is selected. E.g. Services may be Video on Demand, Internet Connection or both of them.

More possible relations that can be seen as constraints are **requires** and **conflicts**.

## 4.13 Ontologies

There are plenty of definitions for ontologies. Ontology is defined in [65] as a domain's conceptualization agreed upon by a community. [79] offers a more specific definition, "*an ontology is an explicit formal specification of how to represent the objects, concepts, and other entities that exist in some area of interest and the relationships that hold among them*". The specification takes the form of the definitions of representational vocabulary (classes, relations, and so forth), which provide meanings for the vocabulary and formal constraints on its coherent use. In DiVA, ontologies could be used as a representation for model information.

In the research community, there is still confusion between the meaning of the terms controlled vocabulary, taxonomy, thesaurus, meta-model and ontology. Pidcock [67] makes an excellent elucidation of these terms:

- A **controlled vocabulary** is a list of terms that have been enumerated explicitly and can be determined according to unambiguous definition.
- **Taxonomy** is a collection of controlled vocabulary terms organized into a hierarchical structure. Each term in taxonomy is in one or more parent-child relationships to other terms in the taxonomy. Some taxonomies allow poly-hierarchy, which means that a term can have multiple parents.
- A **thesaurus** is a networked collection of controlled vocabulary terms. Therefore, a thesaurus uses associative relationships, most commonly the "synonym of" relation, in addition to parent-child relationships.
- A **meta-model** is an explicit model of the constructs and rules needed to build specific models within a domain of interest.

In general, people use the term ontology to refer to different things, including all of the above.

Taxonomies and thesauri may relate terms in a controlled vocabulary via parent-child and associative relationships. A valid meta-model is an ontology used by modellers, but not all ontologies are modelled explicitly as meta-models. On the other hand, people make commitments to use a specific controlled vocabulary or ontology for a domain of interest.

The application of ontologies for reasoning about variability has been mainly applied through the research area of **product configuration**. Product configuration research is a sub field of artificial intelligence [31]. A configurable product allows the adaptation of individual products to the requirements of a particular customer [7]. The main focus of product configuration is the management of complexity in finding a valid configuration conforming to the requirements of the customer [7] and there is the connection with an adaptive system. The adaptive system can be seen as the “product” to adapt according to requirements. In our case those requirement can be changed or updated during runtime according to changes in the environment. In many cases the process of finding the valid configuration(s) should satisfy optimization criteria [75] very common in Artificial Intelligence (AI) research areas.

In that sense, the research done by Oberle et al [5] [64] is relevant to DiVA. They present an ontology-based approach to support the development and administration of software components in an application server. The proposed ontology captures properties, relationships and behaviours of the components that are required for development and administration purposes. For Oberle et al, the ontology is an explicit conceptual model with formal logic-based semantics. As a result, the descriptions of components can be queried, may foresight required actions, e.g. pre-loading of indirectly required components, or can be checked in order to avoid inconsistent configurations of the system — during development as well as during run time. The ontology, allows for finding APIs that come with certain capabilities (development time support) or for pre-loading components that are required by other components (run time support). The fact that the research pursued by Oberle et al, covers issues about the administration during execution of the application is the key issue why it is relevant to DiVA. Specifically, what Oberle et al call development is what we call design-time in DiVA. Likewise, administration corresponds to run-time.

The ontology-based approach offered by Oberle et al maintains the initial flexibility when configuring and running the application server, but also adds new capabilities for the developer and user of the system. The approach has been prototypically implemented in KAON SERVER [89], an application server running components that support a range of various semantic technologies.

More recent research in the domain of dynamic software configuration using ontologies leading to run-time reasoning and service composition is found in [98] [81] [50]. However, as in DiVA, so far the topic has mainly tackled the research question about how ontologies can support such runtime reasoning. If DiVA adopts ontologies as the support for reasoning we would share the same research question.

## 5 Approaches & Algorithms

This chapter represents the core of the survey. To get an overview of the state of art, a range of reasoning domains are regarded. The survey "Efficient Reasoning" [41] assembles basic knowledge about computational hardness of reasoning algorithms in the domains of logic-based and probabilistic reasoning and is cited thoroughly in the following chapter.

### 5.1 Optimisation Strategies

**Pre-processing** of queries in simple reasoning system like standard DBMS (Database Management Systems) can improve run-time effectiveness to linearity. Although these systems seem to be too limited for the DiVA context, pre-processing like indexing or learning can be considered as well for more complicated reasoning approaches. This induces workload on the system at data acquisition time which in a usual declarative approach is performed via e.g. `TELL(KB, "agent supports TCP")` (see 4.3) or at reasoning idle-time but enhances real-time reasoning performance respectively reduces reasoning complexity at runtime. As two main directions, Greiner et al. mention optimisation systems that on one hand do not modify the reasoning system but perform (usually very expensive) pre-processing steps for just a single problem instance [41, p.19]. On the other hand, systems exist that perform only a single pre-processing step to solve a set of queries and modify the underlying reasoner. The modified reasoner is then able to solve queries unaltered as long as the KB doesn't change remarkably, which regrettably may happen often in a dynamic system. As an optimisation example, **clause ordering**<sup>19</sup> is mentioned which highly influences reasoning effectivity of a PROLOG system.

#### 5.1.1 Assumptions

Simplifying assumptions enable a reasoning system to find solutions although knowledge is incomplete. If for example the KB represents that an agent supports network protocols TCP and UDP, under **Closed World Assumption** the statement `ASK(KB, "agent supports FTP")` deduces the answer *false*. Under **Open World Assumption**, however, answering `ASK(KB, "agent supports FTP")` would be neither *true* nor *false* but *cannot answer at this time* as long as "agent doesn't support FTP" is not known from a potentially later arriving `TELL()` operation. While the first answer leads to a fast decision, the second needs perhaps more calculation effort but can, under certain conditions, be more flexible while it is more expressive.

#### 5.1.2 Correctness & precision

A **correct answer** is an answer that is consistent to the knowledge represented by the given KB. To increase **worst-case efficiency**, trade-offs in the precision of answers can be considered for the DiVA needs: an **imprecise reasoner** is allowed to respond to a query with a correct but nevertheless potentially **vague answer**. A vague answer would be *device supports networking* over a database holding {TCP, UDP} assignments while a precise answer would be *device supports TCP and UDP but not FTP* under closed-world assumption. **Precision bounds** express in that case the quality level of an answer as a reasoning measure. Bounds classify answers as *precise enough* or *not yet precise enough*. As soon as precision is sufficiently fulfilled, reasoning is aborted. For example, a component's configuration can be judged as sufficient if its actual memory consumption constitutes between 80 kB and 120 kB, which would be *true* for 90 kB memory, but also for 119 kB.

A common way in **imprecise logical reasoning** is to simplify queries and/ or abstract the KB which is in DiVA planned to be performed via the aspects modelling approach that bundles

---

<sup>19</sup> the order that logical clauses of a KB are applied during reasoning; it is sensible to order clauses such that average reasoning time is minimized

crosscutting concerns. It is thereby aspired that simplification has no negative consequences on reasoning.

Abstraction is closely related to **approximation** in removing correlation from either KB or query. Approximation may be bounded. The theoretically affiliated and verified by implementation **Horn approximation reasoner** of [80] does not limit statements in the SKB or does not use an incomplete inference mechanism but translates knowledge from propositional languages into Horn theories. These can be answered by efficient, subsequent queries. The procedures developed are extendable to the case of FOPL and show to be useful for a variety of knowledge representation languages. The main feature of their approach is a guaranteed fast response to queries that can be answered directly using approximations together with an incremental off-line compilation which improves response time of the system [80, p.35]<sup>20</sup>.

To increase **average-case efficiency** of a reasoning problem, it can be helpful to classify reasoning tasks in the distribution of all tasks. One exploit of classification is to **ignore reasoning tasks** that turn out to be problematic or not plausible [41, p.18]. However, this is not a general solution and demands for an accurate analysis of each system newly conceptualised and implemented as the result of an underlying scenario. In the case of a dynamic, adaptive system, it can be expensive up to impossible to determine all problematic configurations<sup>21</sup>. A support to an 'ignorance approach' can be explicit formulation of Meta rules, e.g. system size constraints for the system.

**Imprecise probabilistic reasoners** are another way to handle vague answers. However, due to the drawbacks depicted in paragraph 4.5, these are not extensively regarded in this survey.

To allow for (every now-and-then) **incorrect answers** to queries can also improve reasoning time efficiency. In the case of **anytime algorithms**<sup>22</sup> which intuitively, due to their real-time nature, seem to be useful for treating reasoning complexity in DiVA, the algorithm's result may depend on the time it is being assigned to reason. If it can exhaustively calculate the problem, its answer will be correct but before that point of time the answer may be wrong in different characteristics, or accidentally already correct. A system that may deliver incorrect answers can be **deterministic** or **stochastic unsound**. Deterministic means that a wrong answer to a query will arise each time the query is posed; stochastic unsound systems may return different answers to same queries [41, p.16]. As a solution to judge the stochastic unsound algorithms result, it may work out to restart it several times and then take the majority of identical answers as the correct result, what obviously consumes time and thus reduces the benefit of time effectiveness. In the case of a deterministic unsound anytime algorithm, it must be considered that the moment of computation's interruption has an influence on its result.

### 5.1.3 Alternative reasoners

Another average-case efficiency improvement strategy is to exploit classification in the way that specific reasoning tasks with specific calculation demands are assigned to optimised-by-problem-class **alternative reasoners**. This demands for a well known system with well-discriminated sub-reasoning and is surely increasingly sophisticated to implement. Classifications can, additionally to pre-definition, automatically or supervised be learned and result in finding inference procedures that are best suited to the **distribution of queries** that the **Reasoning Framework (RF)** has to answer [41, p.21]. Frequent answers can explicitly be cached and

---

<sup>20</sup> also discussed in [41, p.13]

<sup>21</sup> because the system can potentially grow in an unlimited fashion

<sup>22</sup> a real-time algorithm that provides a (potentially still wrong) result at each time it is aborted

be answered faster than 'exotic' ones which still need to be computed by the chosen runtime reasoning mechanism.

**Hybrid evidence** as discussed in 4.10 is an extended approach to increase runtime efficiency. Also here, the system can benefit from a learner that monitors the different, alternative reasoners at work and exploits the measurement of their effectiveness at system runtime. A learning phase can for e.g. a crisis management system induce much overhead to a DiVA system as the possible crisis situations do not appear regularly but must be simulated in advance with the necessity to capture ideally all possible crisis situations, what may turn out to be impossible.

However, adding a **learning component** to a reasoning system is generally an approach of reducing reasoning effort after a learning phase, e.g. for Case-Based Reasoning. Together with the mentioned drawback, this introduces additional implementation effort as well as computational expense of usually NP-hardness<sup>23</sup> during the learning phase. Knowledge learned additionally needs storage space where in a distributed system question may arise where the central KB will be located.

## 5.2 Adaptive Model Checking

The **Adaptive Model Checking (AMC)** methodology of [42] checks for inconsistencies between a system and its corresponding model. In the case of DiVA, this can be verifying a result system against its requirements for validation or as a reasoning subtask. It may be used for model verification as well as for model reasoning because the approach that integrates methods from **machine learning** and **Black Box Checking (BBC)**, which can be applied to cases where verification starts with an inaccurate model - the model is then refined during verification iterations. This is analogue to the DiVA idea of starting with an earlier system and updating it when e.g. a new feature is added conforming to some changes in environment.

AMC as a CEGAR approach is a variant of BBC and combats the state space explosion problem for finite size systems using CEs to modify incorrect models of a real system to verify. Both CE approaches start with a coarse initial abstraction or over-approximation of a system to check whether a certain property can be verified using this abstraction. If it cannot be verified, one obtains a run in the approximation that violates this property, also called counterexample [52] which is an example of the difference between the model and the system. Learning by example is generally computationally. In comparison, learning **Deterministic Finite Automata** via CE offers the benefit of polynomial-time algorithms availability [4].

The authors supply an AMC implementation prototype written in a mark-up language, handling **Linear Temporal Logics (LTL)** with experimental results. As long as inaccurate, but still relevant models are checked that cause smaller size CEs, AMC performs in experiments from 2.2 up to 100 times faster than BBC [42, p.12]. This benefit holds for larger models as are expected in DiVA, but not for small ones which are better treated by BBC. Also, the changes to the system mustn't be too substantial as these increase model update calculation time. The implemented Vasilevskii-Chou-Algorithm is reported to be a bottleneck in the approach. In this regard, potentially newer developments may be available today.

## 5.3 Less expressive belief nets

Less expressive belief nets for probabilistic reasoning can be learned efficiently [41]. They offer correct and precise answers over a limited presentable knowledge. Efficient on average reasoning is possible using limiting (simple) data structures or limiting queries, although general queries are exponential. Naïve Bayes classifiers that allow no general dependencies between attrib-

---

<sup>23</sup> as it can involve solving the original reasoning task

utes are incapable of catching complex relations like constraints between features at arbitrary feature tree positions are. However, tree augmented Bayesian nets allow for attributes between children in the tree. By adding pre-processing steps per query, that modify data structures or algorithms, efficiency is improved (see example algorithms mentioned in [41, p.20f]) during reasoning but often introduces NP-hardness in pre-processing. An effective inference may instead try to find a procedure that is optimal for the distribution of queries that will appear and such performs pre-processing only once. This surely needs expected queries to have been analysed earlier. Caching of answers to queries can further improve reasoning (**absorbing evidence**) as well as pre-processing at  $\text{TELL}(KB, \kappa)$  time. As in other reasoning domains, choosing a best-fit of algorithms for certain queries at runtime is a possible solution to increase efficiency.

Nevertheless, the maintainability drawback due to its low-level-semantics character excludes a low-level PR algorithm from the list of optimal approaches for the overall system, while the idea of using PR for a subset of reasoning tasks in conjunction with other reasoners in a hybrid approach can be sensible. Also, PR may be useful for the pre-calculation of likely future system states during moments of inactivity to speed up reasoning at trigger times.

## 5.4 PRISM

A free (under GPL) and Open Source software framework for model checking is **PRISM** [57]. It incorporates "state-of-the art symbolic data structures and algorithms" and offers three model checking engines which are based on BDDs and **MTBDDs (Multi-Terminal Binary Decision Diagrams)**. Using MTBDDs, structured probabilistic models are space-efficiently represented. Reachability analysis via BDDs is used as the base of non-probabilistic symbolic model checking.

PRISM constructs models together with their set of reachable states from probabilistic variants of **Reactive Modules** [2] model descriptions. Models are expressed by different representations of Markov processes: **Discrete-time Markov Chains (DTMC)**, **Continuous-Time Markov Chains (CTMC)** and **Markov Decision Processes (MDP)**, model specifications in the probabilistic temporal logics **PCTL** or **CSL**. Performing model checking determines then which states of the system satisfy a specification; it is performed as a combination of reachability-based computation and the solution of linear equation systems respectively linear optimisation problems (e.g. via sparse matrices).

A range of case studies<sup>24</sup> is provided to get an impression on how to build and analyse models. The framework can handle big models with e.g.  $10^{10}$  states in MTBDD and solve these within seconds and far below a minute. PRISM is implemented in Java and C++; it uses the CUDD package (see 5.6.3) written in C.

## 5.5 RETE algorithm

Rules/ production rules in a general sense are formulated as *conditions-conclusions* that are implemented in the form of *if-then* clauses. A set of rules together with an execution strategy forms a program; the programming language PROLOG is an example for a rule-based system. Although rule-based systems existed nearly from the beginnings of research on AI on, they are nowadays important for codification of business rules [54]. Two ways to query rule-based knowledge are **forward chaining** and **backwards chaining**.

Backwards chaining as a demand-driven approach answers queries to the reasoning system. Forward chaining is a data-driven approach which calculates conclusions as soon as new facts

---

<sup>24</sup> available at <http://www.prismmodelchecker.org/casestudies/>

are inserted. In the case of a DiVA model, this means roughly speaking that updating the production system with altered information from e.g. context, a product system configuration is calculated as new system state knowledge. During execution of rules of which conditions fit the input during **matching**, system knowledge may be altered, what leads to a subsequent triggering of more rules, and-so-on, until the algorithm terminates e.g. if all rules were checked.

In a naïve forward-chaining algorithm, all inputs are compared to the conditional parts of all rules in each iteration. This leads to a bad performance, but can be optimised as at each input only a small part of the knowledge is affected and only this partition must be recalculated.

RETE<sup>25</sup> was improved as a first version RETE I in 1970's and has become widely used, e.g. in Product Configuration Systems and Business Rule systems/ Production Rule Systems<sup>26</sup> which are in need of efficiency [54] as e.g. CLIPS, Jess, JBoss Rules, and Soar. The RETE-Algorithm optimises rules application in rule-based systems with very many rules to increase efficiency in the following way: as long as the system's memory changes slowly, most facts stay the same between two points in time  $t$  and  $t+1$ . Conditions are therefore built into a network of nodes holding tests so that conditions that are common to several rules must only once be tested – matching needs not be repeated every iteration. Changes in the facts memory are propagated non-expensively through the net at once, causing nodes to be annotated when the fact matches the test. When a combination of facts satisfies all tests for a rule, a net leaf node is reached and the rule is triggered<sup>27</sup>. Annotation costs memory, so that RETE improves calculation speed through memory consumption, what can lead to memory shortness for very large systems. So RETE 1, which was disclosed to the public, is less efficient when handling large amounts of data or very rapidly changing data, which may be the case for a DiVA system. The later on created RETE II algorithm which was not disclosed solves these shortcomings and "is dramatically faster than the original algorithm when dealing with large amounts of data or rapidly changing data" [70]: on tests, the Rete II based system (CLIPS/R2) ran several times faster than the system based on standard RETE on moderately complex problems, and on complex problems, the RETE II system was over 100 times faster [70].

The publicly available RETE 1 can be considered a candidate solver, as it is an approach to reuse existing knowledge about a DiVA system that has been reasoned about in an earlier iteration in an increasingly effective way.

## 5.6 Logical reasoners

### 5.6.1 SAT solver

The Boolean satisfiability problem in propositional logic, denoted by **SAT**, consists of deciding whether logical values can be assigned to the variables of a given propositional formula that render the formula *true* [12]. SAT is NP-complete [77, p.78]. In the context of a DiVA model this means to decide whether a valid product can be configured.

SAT can be solved by highly efficient propositional solvers to implement bounded model checking (see subsections of 5.6). One of these is the **DPLL (Davis-Putnam-Logemann-Loveland)** algorithm which is a complete, backtracking-based algorithm for deciding the satisfiability of propositional logic formulae in **CNF** [23]. However, models arising from the DiVA

---

<sup>25</sup> rete (lat.): net; basics of RETE can be found in [34] and are explained in [http://en.wikipedia.org/wiki/Rete\\_algorithm](http://en.wikipedia.org/wiki/Rete_algorithm)

<sup>26</sup> a computer program typically used to provide some form of artificial intelligence, which consists primarily of a set of rules about behaviour.

<sup>27</sup> a simple example for RETE execution is given in [54]

use cases will presumably be more complicated as they will surely not only contain variables but in addition constraints on variables.

### 5.6.2 CSP solver

The **Constraint Satisfaction Problem (CSP)** not only treats variables on finite domains, e.g. an integer value set  $\{-1, 0, 1\}$  but additionally constraints on them as they will appear in DiVA. **Constraint Programming (CP)** consists of algorithms or heuristics that solve the CSP for a given model such that values for variables are found which satisfy all constraints.

The approach in [11] extends feature models to deal with extra-functional features which are expressed as constraints. Further, it uses **objective functions** which obviously are identical to utility functions to judge the quality of a valid constraint solution. Such a **Constraint Satisfaction Optimisation Problem (CSOP)** not only calculates a solution space but an optimum space of best solutions. *Filters* are introduced to act as limitations for the products that may be transformed from a model, e.g. for only products that support FTP networking. The approach can easily be extended.

While CP is more flexible than the SAT solver or the general BDD/ROBDD data structures and CUDD solver in performing operations on feature models, the CSP solver of [12] performs slow on bigger models in operations like e.g. calculating the number of possible combinations of features. However, it is suited to specific queries as is mentioned there.

### 5.6.3 BDD/ROBDD data structures and CUDD solver

A **Binary Decision Diagram (BDD)** is a data structure for the representation of Boolean functions as a Negation Normal Form (NNF) that can be used as a SAT solver.

An example is given in Figure 7: on the left, a binary decision tree for the Boolean function  $f(x_1, x_2, x_3) = -x_1 * -x_2 * -x_3 + x_1 * x_2 + x_2 * x_3$  and its truth table is shown, on the right, the BDD following from this function. The resulting graph is acyclic, directed and owns one root node; it can therefore also be a **Propositional Directed Acyclic Graph (PDAG)**. An edge from a node to a child represents an assignment of the variable to 0 or 1. A BDD is called *ordered* if different variables appear in the same order on all paths from the root. It is called *reduced* if the graph is minimized according to the rules of

- **Merging:** merge any isomorphic sub graphs.
- **Elimination:** any node whose two children (representing the 0 and 1 variable assignment) are isomorphic is eliminated.

Many logical operations as conjunction  $\wedge$ , disjunction  $\vee$ , negation  $\neg$ , existential abstraction  $\exists$  and universal abstraction  $\forall$  can be implemented on BDDs by polynomial-time graph manipulation algorithms. An Introduction to BDDs may be found in [3].

A **ROBDD (Reduced Ordered Binary Decision Diagram)** guarantees uniqueness due to its canonical form; it can therefore easily be used in functional equivalence checking as formal verification and for model checking.

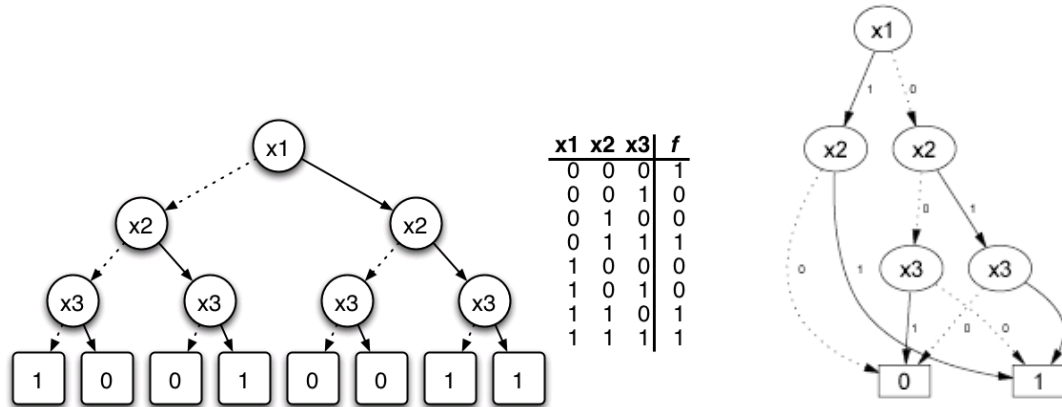


Figure 7 Left: binary decision tree for a Boolean function. Right: its BDD [94].

An important point for reducing *space complexity* in BDDs is variable ordering to reduce graph size. However, the problem of finding the best variable ordering for a BDD is NP-hard [14]. For any constant  $c > 1$  it is even NP-hard to compute a variable ordering resulting in an OBDD with a size that is at most  $c$  times larger than an optimal one [83]. Luckily, efficient heuristics exist to tackle the problem.

The weakness of this kind of representation is the size of the data structure which may vary between a linear to an exponential range depending on the variable ordering as mentioned. However, the memory consumption is clearly compensated by the time-performance results offered by BDD solvers [12].

### CUDD

Many implementations for BDDs are available (see an elaborate list on [http://en.wikipedia.org/wiki/Binary\\_decision\\_diagram](http://en.wikipedia.org/wiki/Binary_decision_diagram)). One is the **CU Decision Diagram (CUDD)** package. The framework's description is available at [85]. The CUDD package provides functions to manipulate BDDs (besides **Algebraic Decision Diagram (ADDs)**), and **Zero-suppressed Binary Decision Diagrams (ZDDs)**. BDDs are used to represent switching functions; ADDs are used to represent a function from  $\{0,1\}^n$  to an arbitrary set. ZDDs represent switching functions like BDDs; however, they are much more efficient than BDDs when the functions to be represented are characteristic functions of cube sets, or in general, when the ON-set of the function to be represented is very sparse. They are inferior to BDDs in other cases.

The CUDD package provides a set of operations on BDDs, ADDs, and ZDDs, functions to convert BDDs into ADDs or ZDDs and vice versa, and a large assortment of variable reordering methods. It can be used in three ways:

- As a **black box**. In this case, the application program that needs to manipulate decision diagrams only uses the exported functions of the package. Functions included in the CUDD package allow many applications to be written in this way. An application written in terms of the exported functions of the package needs not concern itself with the details of variable reordering, which may take place behind the scenes.
- As a **clear box**. When writing a sophisticated application based on decision diagrams, efficiency often dictates that some functions be implemented/ added as direct recursive manipulation of the diagrams, instead of being written in terms of existing primitive functions. A recursive function can be interrupted by dynamic variable reordering.
- Through a C++ **interface**, which offers almost all the functionality provided by the CUDD and is thus especially recommended for fast prototyping by the creators. A Perl5 interface also exists and is distributed separately.

CUDD is implemented in ANSI C and C++. However, a Java interface/ implementation exists in JavaBDD<sup>28</sup>, so it can be added to the Eclipse Java environment. CUDD is used in frameworks like PRISM (see 5.4) and NUSMV 2 (see 5.6.3). It has also proved to be a worthy tool in the implementation of the variant management product pure::variants<sup>29</sup>, while this doesn't support cardinalities yet.

To be able to judge how CUDD can improve complexity reduction, the DiVA use cases need to be exploited to explore adequate representation in BDD, ADD or ZDD.

## NuSMV 2

Another Symbolic Model Checker for BDDs under LGPL Open Source licence is **NuSMD 2**, a "well-structured, flexible, and documented platform for model checking close to industrial standards" [17, p.1]. It is an extended version of the long-established NuSMV package and adds model checking based on SAT which is nowadays popular and successful in several industrial fields. Both BDD and SAT can complement each other as they are often able to solve different classes of model checking problems (see section 5.11 for details). Because of its aim in making as much functionality as possible independent of the model checking engines, it can be extended by new solvers. The framework is programmed by an extension of the SMV language which can describe **Finite State Machines (FSMs)**, instantiate modules and asynchronous processes and express requirements in CTL and LTL. Input models are in the BDD and SAT case internally encoded (*flattened*) as Boolean models without scalar variables. Properties may also be flattened and a subsequent **cone of influence** reduction step can be "extremely effective in tackling the state explosion problem", Clarke et al. claim<sup>30</sup>. The resulting internal model representation can then be processed by either BDD or SAT solvers, offering verification operations like for BDD:

- reachability analysis,
- fair CTL model checking,
- LTL model checking,
- Computation of quantitative characteristics.

---

<sup>28</sup> <http://javabdd.sourceforge.net/>

<sup>29</sup> <http://www.pure-systems.com>

<sup>30</sup> papers regarding these optimisation steps are referenced there

For SAT, bounded model checking is performed via counterexamples up to specified maximum bounds, based on the well-known Davis-Logemann-Loveland-algorithm [23] which can produce “dramatic speed-ups in the overall performance” [17, p.4]. At the time of the cited papers writing, new state of the art SAT solvers were under development<sup>31</sup>. The user may select the solution method for a property manually. Counter-example traces are logged and can be analysed by the user, which may be helpful during development time of the ARF. The user may also simulate a system’s behaviour via BDD or SAT.

## 5.7 Backtracking algorithms for Case-based Reasoning

Reasoning on a DiVA system carries the notion of **Constraint-based reasoning** as is discussed in the section after this section. A constraint network is defined by a set of variables, a range of values for each variable and a set of constraints between the variables. Solving the network means an assignment of values so that all constraints are satisfied [53]. Such networks may be solved by basic algorithms as classical **Backtracking (BT)** and elaborated BT algorithms as Backmarking, Backjumping, Forward-Checking, and Conflict-Directed Backjumping, where different average performance efficiency in reasoning appears. The decision which is the best fitted for a certain domain heavily depends on the specific problem to be solved and such cannot be judged in this survey while the problems depend on scenarios that have not yet been formalized.

For a theoretical comparison of algorithms, two measures are advised by Kondrak and van Beek [53, p.2]: the **number of consistency checks** performed and the **number of nodes in the backtracking tree**. The authors analyze the above mentioned BT algorithms as well as hybrid algorithms and construct hierarchies of them related to the performance measures. These can later on be used as a base to select a potentially needed BT algorithm for the above mentioned solvers, e.g. backtracking is a basic approach in solving BDDs.

## 5.8 Similarity and Case-based Reasoning

In **Case-based Reasoning (CBR)**, a preserved set of cases and their solutions called **episodes** form the base of reasoning. Cases consist of features and their relationships [84]. A solution can usually only be applied if the new problem stems from the same or a similar domain as the base episodes. The more predictive features of base cases are indexed and the known solutions (**targets**) are searched, where search is a transformation from base cases to targets. In a first retrieval stage, similar candidate episodes are selected. These are mapped in stage two from a selected case to a target using adaptation operators. The problem solver thus finds a sequence of state-transformation operators. This can be, dependant on the problem complexity, a simple or an even unsolvable task.

To fit a base case to a target, adaptation of operators is performed. Complexity in reasoning and cost of search appear through the **operator chaining problem** together with the **operator conflict problem**. In case of a DiVA system where entities have conflicting and interacting goals, backtracking increases worst case reasoning costs assumedly remarkable above linearity. Reasonable database setup in CBR may reduce backtracking effort. Different operator chaining techniques exist like heuristic search, hierarchical planning, least commitment strategy, and goal regression. An example of similarity measures in CBR are **Nearest Neighbour (NN)** estimation [49, p.22ff].

An extensive overview of CBR techniques together with constraint based, fuzzy and probabilistic modelling is given in [49]. As a remark to this paragraph, from this paper may be cited: "Approaches such as CBR are often criticized for their ad-hoc character. Even though methods of

---

<sup>31</sup> like e.g. CHAFF; an algorithm for solving instances of the Boolean SAT problem

this kind proved to be successful in practice, this criticism might be intelligible from a theoretical point of view. Indeed, it seems to be true that many CBR applications, even if successfully solving the problem at hand, lack a sound theoretical basis". This may turn out to be problematic for an initially only conceptually drawn dynamic system the behaviour of which cannot generally be pre-calculated due to e.g. random real-time events. It will surely be necessary that a specifically considered system which cannot be observed in reality has to be **simulated** first to create a KB of episodes or to introduce a user guided learning phase where results must be judged. This implies an initial problem solution stage with statically fixed interim results. Getting a solution out of such a database is quite a simple task for non-demanding surroundings, as e.g. is a system that finds a house out of a catalogue of estates, because here parameters are relatively few and success is easily measurable in customer's demands. Yet, in a dynamic, adaptive system, reasoning causes timely changing and more conflicting goals what causes much bigger search effort.

[84, p.4] informally classifies problem solving complexity in CBR as follows:

- **Simple:** the calculated case is very similar to an already existing target case. This requires just simple substitutional adaptation.
- **Routine:** the calculated case will still be similar to the target, but requires more extensive reconfigurations of the case model, e.g. changing structure or single elements, causing a transformational adaptation.
- **Innovative:** complex changes in the case are necessary. This most likely requires user support during adaptation in order to resolve a solution case at all. Solutions are generative in the way that formerly non-existing solutions are to be calculated similar as is performed by humans using intuition and heuristics.

Regarding DiVA, routine, medium hard complexity is premised throughout this survey: reasoning is not simple due to the dynamic nature of the system, but doesn't need to be innovative if appropriate constraints in system design are applied in an early design stage. The target of the system and the behaviour of its components must thus be elaborately described by rules. DiVA domains may be incomplete; its problem & solution specifications can be conceptually distant. Conflicts between the dependant domain components are likely because of their rush for resources. Such limited, a basically complex DiVA CBR system structure will cause expensive, but not unsolvable calculations.

The enemy 'exponential calculation complexity' could in a brute CBR approach theoretically be reduced by trying to create a huge KB of mostly all possible cases and solutions in so-called instance-based methods, but this does not solve reasoning complexity overall, as exponential space complexity would be the result and is therefore out of discussion [49, p.21]. However, learning knowledge in **Instance-based Learning (IBL)** methods which reason in a locally limited way can be enhanced by organizing an optimally structured memory of cases and even by removing cases in a later stage like in non-monotonic reasoning as a reflection of past reasoning decisions, with the drawback that removal may require human supervision and that memory management can become a bottleneck together with its naturally high memory consumption for KB representation.

In **Model-based** approaches, a preferable low-dimensional model of the system is constructed and observed of how well its parameter values fit actual system data, e.g. the context information. A model can be used for performance tasks, such as explanation of the observed data, problem solving, and future prediction. If ways can be defined throughout the DiVA methodology to influence future decisions of agents and thus their choices, an estimation of future situations can help to constraint ARF reasoning at runtime based on earlier reasoning results. This

leads to the more globally oriented **Model-based Reasoning** in the field of AI [49, p.18f], where surface models and deep models<sup>32</sup> are differentiated.

An example for a transformational adaptation of a medium hard reasoning system is given in [84, p.6ff]. Adaptation is seen as a solution transformation problem of several parameters, where earlier cases with problems and solutions exist and the new problem has to be derived by a solution transformation in a **transformation space**. However, this simple example is far away from the complexity of a DiVA system where many agents compete in parameter adaptation.

**Hierarchical approaches** as used in this CBR example can help in speeding up reasoning as they retrieve cases at different levels of abstraction and detail. Higher level results are adapted to sub-tasks of targets, e.g. in a **blackboard control architecture**, where local observations and transformations of primitive transformation operators can be monitored by other operators. Together with an appropriate strategy, this helps to handle complex and perhaps ill-defined, partly contradictory problems [84, p.8ff]. Additionally, hierarchical reasoning supports a potential parallelization of calculation effort by distributing it over the system network. However, the distribution of knowledge necessary for distributed reasoning implies new challenges. As a base for level of abstraction judgment and preferred treating of certain reasoning aspects, priorities may be used.

## 5.9 Artificial Life

Artificial life is a field in cognitive science that studies different systems related to life, their processes, and evolution. Such study is carried out through simulations using computer and mathematical models. Artificial life attempts to imitate traditional biology and chemistry to recreate the biological phenomena.

Research in artificial life concentrates considerable effort on the *emergent* phenomena. Complex but coherent global phenomena arise (*emerge*) from the interaction of small component systems where the components are simple in relation with the global phenomena. An example of such phenomena is an ant colony, where the constituent elements of the colony (ants) interact between each other and move as a whole. While the behaviour of each ant can be described and modelled by each agent in the group following a small number of rules; the coordinated behaviour – emerges – from the interactions of the constituent elements.

In DiVA, artificial life approaches can help managing the emergence of behaviour product of unpredicted configurations of the system [39]. They can also be useful to solve learning [91; 59] and combinatorial optimisation problems [36; 28; 29] related to select the best adaptation given a deterministic goal.

### 5.9.1 Artificial Neural Networks

An **artificial neural network (ANN)** is a computation model based on the metaphor of biological neural networks [91, 59]. It is rooted on the concept of parallel-distributed processes [20; 21], where the combined activity of many simple processing units (**neurons**) operating in parallel could give rise to an overall complex computation.

ANNs typically model each **processing element (PE)** as an operation that takes as input some scalar values and applies a nonlinear mathematical function to those values, one that produces a single scalar number. The input of a PE can come from the environment external to the system or from other PEs within the system. Analogously, the output value of a PE could form part of

---

<sup>32</sup> A **deep model** is an explicit system model with respect to the system's structure and behaviour. A **surface model** as in conventional rule-based systems describes the system by rules and facts.

the overall output of the system, or input of other PEs within the system. PEs are related to each other through a connection, which passes the values produced by one to another. The strength of the connection defines how much related two PEs are and enables the learning capability. For instance, a particular single PE could have a large effect on some units while having only a small influence on others.

A particular property of neural networks is the so-called **graceful degradation**, which means that if connections or PEs are removed from the network or noise is introduced to the inputs; performance of the network will decline gradually but not completely and abruptly. This behaviour is preferred to system suffering serious failures, which is typically expected. Furthermore, the transition between states of a neural network gives rise to emergent behaviour that otherwise may not arise [74].

The principal property of neural networks is their ability to **learn from experience** [90; 46] which they acquired through a learning algorithm. Learning from experience refers to the tuning of the connections inside the network through the results of a series of **learning trials**. Results produced by the trials are mapped to the desired output and the corrections to the connections are performed. In the literature, there are three kind of learning algorithms: **supervised**, **semi-supervised**, and **unsupervised learning**.

- *Supervised learning* involves training the system to replicate some given set of input-output mapping [73; 93]. It consists in presenting inputs taken from pre-established training sets of input-output pairs. The intention of supervised learning is that the network does not only learn to replicate the response of the training set, but also that it generalizes to other input spaces. In this way the network should be able to produce the desired outputs even when it has not been previously exposed to the inputs.
- In *reinforced learning* there is no direct relation between input-output. Instead of training the systems with a pre-established set of input-output pairs, it is rewarded or punished whether it produces the correct or incorrect output [86; 87; 92]. In reinforced learning there is still the notion of **expected output**.
- On the other hand, *unsupervised learning* does not provide feedback to the network and there is no expected result. Unsupervised learning seeks a model (typically statistical) to determine how inputs relate the outputs.

In DiVA, ANNs can be trained to select/ to learn an adaptation that besides respecting some invariants fulfils functional and extra functional system goals. The learning ability of ANNs provides support for managing the emergence of behaviour when adapting to unpredicted environments.

### 5.9.2 Evolutionary Computation

Evolutionary computation uses metaphors based on biology, chemistry or natural phenomena to solve functional optimisation problems. That is, trying to find a combination of parameter values that minimises or maximises some function. It is called **evolutionary** because most of the approaches it embodies are sustained by the natural selection evolution theory – **survival of the fittest**. Evolutionary algorithms are often called *search algorithms*, because they explore a search space to find a solution that is close to the optimal.

Evolutionary computation and other search techniques have been applied in software engineering [78] to solve problems related to testing [8; 58; 71], maintenance [5; 45; 37], requirements engineering [32] and model synthesis [38]. DiVA can benefit of evolutionary computation to explore the adaptation space. In this way, evolutionary algorithms can find the best fitted adaptation, or select a reasonable amount of adaptations scenarios for validation purposes. They could also be used as learning support by tuning the parameters of a neural network. In the fol-

lowing sections, three classes of evolutionary algorithms are figured out: Genetic algorithms, Ant colony optimisation, and Particle swarm optimisation. These three classes of meta-heuristics are thereafter discussed in a self-contained section.

### Genetic Algorithms

**Genetic algorithms (GA)** [36] are a style of evolutionary algorithms based on the genetic evolution metaphor. The genetic structure between different generations of individuals may adapt and evolve to provide better responses to the changing environment.

The mechanics underlying this kind of algorithm are governed by a function to optimise. The parameters of such functions are *encoded* as strings of characters. For instance, parameters can be encoded as string of binary digits. Each string is referred to as *gene (individual)* of and *individual* in the *population*. Initializing randomly several strings creates an initial population, which later evolves. Evolution consists in an iterative loop of *evaluation* and *breeding*. Evaluation consists in measuring the quality of each individual through a single numeric *fitness* value, which indicates how good they are. Once the population has been evaluated, the algorithm *breeds* a new population (*generation*) by selecting the fitter members of the population as *parents*. Evolution is achieved by mixing the genetics of parents in a variety of ways (typically in literature this is called a *movement* in the regard that it moves one generation to another). In order to explore un-explored portions of the search space, evolution can also introduce random *mutations* to genes.

The mixing and reproduction process involve the application of genetic operators. For instance, a typical operator –*crossover*– recombines randomly the genes from different parents and produces a new individual. Often a pair of parents is *crossed* and its selection is based on their fitness value, in this way the new individual inherits some of the genetic material with good fitness.

A common technique in GA is *elitism*. It consists in selecting the best-fitted individual of the evaluated population and always adding it into the population of the next generation. This ensures that the best individual encountered so far is always preserved. Another form of elitism consists on selecting a percentage of the population best fitted to the breeding process and excluding the rest.

As said before, the goal of GA is optimising a function value. Each possible gene is a point in the genotype space, and the function used to calculate the fitness defines a surface over this space. It is worth mentioning that the fitness function maps the target *to optimise function* into the encoding (genotype) space. GA aims at finding the optimal (best value) points in this space. Whenever the execution of a GA always finds the same *optimal* point, it is said to *converge* on peaks in the fitness surface.

### Ant Colony Optimisation

**Ant colony optimisation (ACO)** [28] is an evolutionary algorithm based on the ant colonies metaphor. In nature, ants can select the optimal path to a food source by following *pheromone* trails left by explorer ants. These ants wander at random where the food source is located [40]. When an ant travels a path to the food source, it leaves a pheromone trace; other ants may follow this path to find food and will leave more pheromone [62]. Over time, pheromone trails start to evaporate, hence reducing their attractive strength. While ants take more time to travel a long path down and back again, more pheromones evaporate over time. A short path, by comparison, gets re-filled with pheromones faster, and thus its pheromone density remains high as it is laid on the path as fast as it can evaporate.

In ACO ants explore the solution space of an optimisation function. Each artificial ant visits iteratively the parameter values of the function and lays **artificial pheromone** in amount pro-

portional to the **fitness** value of the parameters. Through the iterations, ants tend to visit the closer parameter values with more pheromone than others, thus re-filling the path with pheromone. This selection process amplifies previously reinforced parameter values and leads to the emergence of good solutions. Artificial pheromone decays over time preventing mediocre parameter values from being amplified by accident.

ACO results to be very good for dynamically changing problems. It maintains a pool of alternative portions of solution, the pheromone decay equation in these solutions ensures that every link has at least a small amount of pheromone, then a weakly used link can be quickly reinforced and replace a missing or failing link. However, ant colonies can reinforce portions of solutions that belong to many good solutions. While a given portion belongs to more good solutions, more virtual pheromone it receives. If a large number of portions of solution are equally inclined to be part of good solutions, ACO cannot differentiate them and performs poorly.

### Particle swarm optimisation

**Particle swarm optimisation (PSO)** [51; 29] is an evolutionary algorithm based on a social-psychological metaphor. It simulates for instance the behaviour of bird flocking. A group of birds are randomly searching food in an area. There is only one piece of food in the area being searched. No bird knows where the food is but all birds know how far the food is after every iteration. The best strategy that birds can adopt is to follow the companion bird that is nearest to the food.

In PSO, an individual (bird) is represented by a single solution or **particle** to an optimisation problem. Each particle has a fitness value that is evaluated by a **fitness function** to be optimised. Besides, each particle has a velocity (**vector**), which directs its flying. The particles fly through the problem space following the current optimal particle.

Given a function to optimise, encoded as a fitness function, the PSO is initialized with a group of random particles and then searches for the optimal solution by updating generations. Through iterations, each particle is updated following two best values. The first one *-pbest-* is the best solution the particle has achieved so far. The second one *-gbest-* is the best value obtained so far by any particle in the swarm. By using these two values *pbest* and *gbest*, the position and velocity of the particle is updated. Particles will follow the best particle if any exists or may explore the solution space for a new optimum.

### Artificial life applied to adaptive systems

Recent work in adaptive systems [38] uses digital evolution to handle the emergence of properties and configuration of adaptive systems.

The approach to digital evolution adopted by Goldsby and Cheng relies on a reward and punishment evolution style [66] where the most rewarded beings prevail. Digital evolution is then used to synthesize a suite of models representing a target adaptation of the system. Simple organisms coexist in a limited environment that can hold a fixed population. Each organism is a **model** (state machine), which self-replicates based on a reward model and evolves randomly each time. Evolution is performed during the replication and consists in mutating the organism by adding, removing or modifying its constituent elements (transitions and labels). The **likelihood** to replicate for each individual is evaluated according to its performance. The individual performance of each individual is evaluated according to a set of desired tasks, for instance, the functional properties satisfied by an individual. Performing a task increases the *merit* of an organism, which determines its access to the available resources. Better performance implies a better reward. Organisms with more access to resources are more likely to self-replicate. As the population has a fixed size, organisms with better merit have more chances to survive than their peers with less merit. This *kills* progressively the population of poorly performing organisms.

Competition for survival ensures that, over time, the population comprises organisms that increasingly achieve more goals.

The result of the described evolution is a set of target adaptations (models) that satisfy the overall functional invariants. The models exhibit different emergent functional and non-functional behaviour suitable for domains not previously specified. For instance, developers can discover that one model is more fault-tolerant and energy efficient, whereas another is more secure and resource intensive. This assists developers in distinguishing the generated solutions and making informed trade-off between models in order to select models to use as target systems.

### Evolutionary computation discussion

The implications and applications of each evolutionary algorithm in DiVA are still uncertain. The use of one algorithm over another is subject to the reasoning model, the size of the reasoning space, etc.. Deciding which algorithms are the best suited for DiVA is difficult at this point; instead we study their advantages and disadvantages.

A common ground for these algorithms is the need to define an optimisation or fitness function, which encodes the objectives to maximise. Furthermore, each algorithm requires a special encoding for the particular problem it tries to solve. GA uses genes and chromosomes, PSO uses particles and ACO uses ants and trial paths. The advantages and disadvantages of the three evolutionary algorithm classes are the following.

- **Genetic algorithm** is a meta-heuristic ideal for problems where the optimisation goals are complex, discontinuous, noisy, and changing over time and with many local optima. Its main advantage is that it can solve problems knowing almost nothing about them from the start converging very quickly to an optimal solution. However, it needs a proper representation of the problem solution because its performance may rely on it. Whether its implementation is relatively easy compared with PSO and ACO, it requires the setting of many parameters such as population size, mutation rate, crossover rate, etc. Moreover, it cannot solve problems where the solution gene has variable length; adaptation of GA such as bacteriologic algorithm can provide solutions to variable length problems. Finally, GA relies strongly on the initial solution and suffers of *premature convergence*. That is, an individual that is better fit than others at early stages may dominate on the reproduction process, leading to a local optimum rather than a search for a global optimum.
- **Particle swarm optimisation** is a meta-heuristic capable of providing high-quality solutions within shorter calculation time and stable convergence. It performs very well for problems of variable length and contrarily to GA it is less dependent on a set of initial solutions. This implies that its convergence is robust. PSO is easy to implement in comparison to other meta-heuristic and has only a few parameters to set. However, it requires a complex encoding for its particles, and has a slow convergence in refined search stages because of its weak local search ability.
- **Ant colony optimisation** is a meta-heuristic efficient for solving dynamically constrained discrete problems. It is capable of adapting to changing problems quickly absorbing to changes such as new elements. Analogous to PSO, it is less affected by poor initial solution. In problems of middle and medium size it generally outperforms GA finding better solutions. However, it is complex to implement, and require an encoding for the ants and the paths traversed by them. Moreover, even when its convergence is guaranteed, the convergence time is uncertain.

Although in DiVA any of these algorithms could be used to determine the best configuration to adopt, or the better component arrangement or evolution order, is more likely that a combination of them will give a better solution rather than a single one.

## 5.10 Partitioning of Reasoning – Divide & Conquer

The **Divide & Conquer (D&C)** approach proposed in [76] as a contribution to the MUSIC project<sup>33</sup> is developed in the context of adapting large, distributed applications in mobile environments where frequent context changes appear, where reasoning is similar to DiVA's reasoning task. It tackles the problem of exponential growth via partitioning (*divide*) reasoning over the whole system in smaller parts and combining (*conquer*) smaller solutions in such a way that the overall calculations are reduced. Applications are split into **application parts (parts)** and the algorithm organises **collections of parts (packs)**. As such, D&C tries not to optimise the utility of a product system globally but applies heuristics to solve the overall problem in (perhaps many) local sub-computations. The approach does not require complete knowledge about the environment and the applications.

Parts are conglomerates of components and have an own utility function. Packs are logical assemblies of parts; parts in packs are treated as a whole and are also adapted as a whole. In forceful exhausting such partitioning, it becomes possible to determine which parts can be adapted independently. The strategy is to put parts into different packs to reduce the cross-product of possible adaptations to a sum, such that computation time decreases from exponential time to linear time.

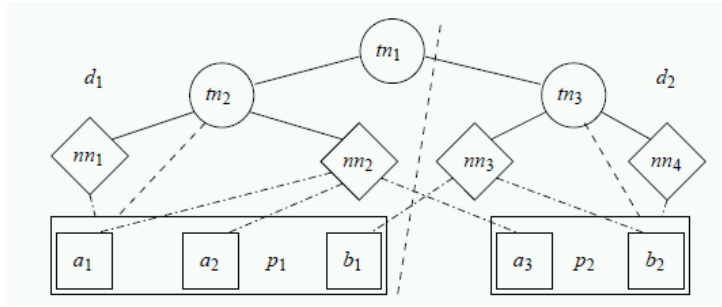
Packs may be merged or split to let an algorithm adapt to the reasoning conditions. Of course, this leads to a worse (but hopefully still acceptable) overall utility as only reasoning over the whole system optimises utility. On the other hand, utility outcome can be balanced against reasoning time and adaptation quality, so that the algorithm may scale to a weaker or stronger hardware it is running on. Bigger packs lead to a better utility but may require more reasoning effort, smaller to a weaker utility but to an increased reasoning time.

The authors additionally introduce **rating**: a user may decide if an application is important to her or not so that a priority of reasoning can be applied. This leads to an improved user satisfaction, but demands for human intervention.

Data structures are organised in a **Directed Acyclic Graph (DAG)** as a *decomposition tree* shown in Figure 8 which represents the result of organisation at a certain time. Squares  $\square$  denote packs, diamonds  $\diamond$  denote negotiation nodes which distribute system resources of devices or hold information about dependencies of the parts an application consists of. Other tree nodes are shown circular  $\circ$ . Nodes are annotated with local information about adaptation, their parents and children and about earlier decisions. They are changed by the middleware according to rules at certain context changes. Usually, only a few nodes are updated (are active), activity is handed over from active node to active node. The operations on the decomposition tree are divided into three types: *atomic operations* (realized by a middleware), *strategies* (which complex operation to perform in which situations) and *complex operations* (reactive actions to apply on the tree).

---

<sup>33</sup><http://www.ist-music.eu/MUSIC>



**Figure 8 Decomposition tree in the D&C approach.**

**Operations** are explained in detail in [76, p.7ff]. Roughly these are:

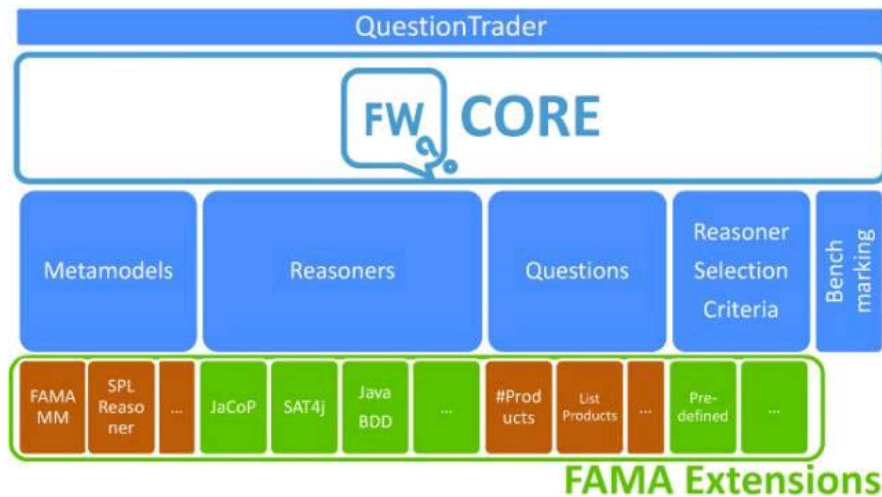
- Splitting and merging of packs.
- Assignments or removing of application to packs at their start-up/ termination.
- Handling of disappearing devices.
- Decomposition tree balancing.

**Strategies** are condition-action rules for the principles of reasoning; each node has its own rule-based control loop. Strategies control organisation of adaptation, e.g. splitting and merging of packs. They are also necessary to prevent problems as deadlocks, unreachable states, or to prevent an oscillating tree [76, p.9]. Strategies are still under development, the authors mention.

Implementation of the approach seems not to be trivial; additionally, the control loop algorithm is regrettably not explained in depth and the approach is still under progress as the research group is still developing a simulator to develop different strategies at the time of publication in 2008. Besides the general benefit of time-effectiveness, a D&C approach offers the possibility of parallel computing on multiple processors.

## 5.11 FAMA

**FAMA**, the **FeAture Model Analyser** [12], is a framework for automated analysis of Feature Models which are explained in chapter 4.10. It looks solidly packaged which makes it easier to integrate in the DiVA platform as e.g. CUDD. FMs are represented via a feature markup language in XML. The tool suite FaMaTS<sup>34</sup> is built following the SPL paradigm supporting different variability metamodels, reasoners or solvers, analysis questions and reasoner selectors, easing the production of customized VM analysis tools.



**Figure 9 FAMA architecture.**

FAMA allows for integration of several solvers to be able to choose a best fitting solver for a specific reasoning operation while solvers may be selected dynamically at runtime. Currently, different logic representation like BDD (Binary Decision Diagrams), SAT (Boolean Satisfiability Problem) and CSP (Constraint Satisfaction Problem) are implemented. FMs are translated into per-solver-fitting logical representations to reason on tasks of interest for DiVA as

- Validating a FM to find out whether a product exists satisfying its constraints.
- Calculate the total number of products that can be created from a FM.
- List all products that can be created from a FM and its constraints.
- Calculate the number of products a feature appears in.

The Authors have shown that one optimal reasoner for all these operations does not exist but that switching reasoners for specific model queries is a solution for improved overall performance (see Hybrid Evidence in section 4.10). Together with several available operations and the flexibility to be able of adding additional reasoners the FAMA approach looks promising. The implementation is working and actually being further developed by the authors. It is available under the GNU Lesser General Public License (LGPL). Third-parties are allowed to develop and integrated their own FAMA extensions for automated reasoning. Some basic services are provided by default. Among these features are VM file storing, a benchmarking system to compare the performance of several reasoners and VM random generation.

<sup>34</sup> homepage <http://www.isa.us.es/fama>,  
download <http://code.google.com/p/famats/>

In Figure 9 a detailed description of FAMA architecture is given that is briefly described to understand the functionality and the flexibility of the tool<sup>35</sup>:

- *QuestionTrader*: interfaces for a query-like interaction with FAMA. They are uncoupled from FAMA Core so changes on it do not affect the final user.
- *FAMA Core*: communicates different extensions among them.
- *FAMA Extensions*: a set of modules that allows FAMA FW to adapt to user needs:
  - *Metamodels*: to integrate own variability or feature metamodel and file loaders/ savers into FAMA. FAMA supports a FAMA Eclipse plug in metamodel and XML and X3D (3D representation of feature models) file formats by default.
  - *Reasoners*: new reasoners or solvers developed by third parties can be added. By default, JavaBDD, JaCoP and SAT4j are provided with the tool.
  - *Questions*: incorporate new reasoning operations. Products counting and listing, error detection and explanation, and products filter are included.
  - *Reasoner Selector*: select at runtime the reasoner to answer an analysis question among all the available reasoners and based on a previously defined criterion (performance, accuracy, ...). FAMA allows to develop own smart selectors.
  - *Benchmarking*: compare different reasoners performance for the same operation. A random feature model generator is provided which allows storing the data and results for a later analysis.

Authors state that they are currently working on integrating attributes and new extensions such as extended feature metamodels (those containing extra-functional attributes), new reasoners as Choco, VM refactoring and merging. Not supported until now are constraining rules. Such, the framework is not still ripe but on its way to be a productive tool.

---

<sup>35</sup> see

[http://www.isa.us.es/fama/modules/portalWFInterface/init.php?module=83&id=fama\\_web&auth=a0793fd12e5413cccbdac5c18986c60a&fragment=fama\\_architecture&#fama\\_architecture](http://www.isa.us.es/fama/modules/portalWFInterface/init.php?module=83&id=fama_web&auth=a0793fd12e5413cccbdac5c18986c60a&fragment=fama_architecture&#fama_architecture)

## 6 Conclusion and recommendation

### 6.1 Conclusion

Reasoning about knowledge is a wide, interdisciplinary field. This survey offered an overview of basic reasoning aspects as well as of data structures, algorithms and frameworks. While loads of reasoning approaches exist for different domains, approaches in the context of reasoning on variability in complex, adaptive systems are still in development. So no well-established 'off-the-shelf' solution for DiVA's specific reasoning needs was identified. A still pre-mature treatment was found in the D&C approach of [76] described in 5.10.

The ARF will obviously not be a simple reasoner but a Meta system, as the reasoner to choose for a specific reasoning domain highly depends on the problem definition, its features and parameters. Its inner reasoner may be one reasoner, alternatively chosen reasoners as in the Hybrid Evidence idea mentioned in 4.10 or as in the FAMA framework from paragraph 5.11. Also, several reasoners that operate in hierarchies or networks are possible.

Problems which are currently developed in DiVA's case study specification will be the base for deciding on individual reasoners. We seek reasoning approaches respectively single reasoners with an optimal performance specifically for a DiVA system, where performance is a combination of *{correctness, precision, expressiveness, time-efficiency}* [41, p.22]. Efficiency (average, worst-case) is a hard constraint.

A utility function as **performance measure** can later help to evaluate reasoners and compare them based on the function's score. These comparisons can be performed during the implementation phase when reasoners are selected for a specific reasoning task. *Robustness* is also a concern as for certain approaches as the D&C tree for [76, p.9] problems like deadlocks, unreachable states, and oscillating tree behaviour must be prevented in implementation.

### 6.2 Evaluation criteria

For implementation of the ARF, approaches (algorithms), data structures and frameworks must be selected. To choose from the approaches that have been discussed in this document, the following criteria were applied. As information sources are quite heterogeneous and sparsely carried out, not all criteria could always be determined for an approach. Beside basic considerations a source for these criteria are the DiVA requirements [27] which are explained in section 3.1. Central tasks are

- **Support for dynamic reasoning.** This implies the need for *flexibility* in the ARF.
- **Support of variability verification.** A need for *accuracy* follows from this.

#### 6.2.1 Complexity reduction

The main goal of the ARF is to reason fast which can for big models only be achieved by complexity reduction of the underlying models or in sophisticated calculation, e.g. by **heuristics** or by trade-offs in the reasoning result's **accuracy**. Accuracy reduction does not mean that a reasoning answer is completely useless but that it is not necessarily the optimal answer. This may be sufficient for certain or even many reasoning situations. As such, it is important that an approach offers opportunities to reduce time-complexity, which can be 'bought' by e.g. a higher memory consumption.

**Approach complexity** can roughly be taken as a real-world performance measure: an approach which is mathematically linear in time but also needs a long time span for a small number of computations is not usable in practise. Another complexity which is taken into account for algorithms is the **input data effort** for calculating the data base to reason upon. E.g. it is theoretic-

cally possible to pre-calculate all results for a fixed system but this usually results in immense memory consumption.

**Flexibility** expresses how well an algorithm may be adapted to changing reasoning environments/ tasks.

### 6.2.2 *Maturity*

**Maturity** is a measure for how well-established a technology is. If it has been applied in many earlier projects chances are good that it is highly usable and will support reaching a successful project result. On the other hand, if a work still in progress offers a high **potential** to reach an innovative result, this is also annotated.

### 6.2.3 *Documentation*

A well thought-out algorithm that is not detailed described cannot be easily implemented; the same goes for a framework that is potentially mighty but not well **documented**. If **case studies** of problems similar to DiVA's exist, this is considered positively.

### 6.2.4 *Software*

The software being used for implementation is to be deployed publicly under an **Open Source** licence. Also, the software must fit into the DiVA studio infrastructure that consists of **Eclipse**<sup>36</sup> as an integration platform.

## 6.3 Evaluation

The following table evaluates approaches that have been treated throughout this document.

- The left hand captions express the optimal case of a value, e.g. 'Accuracy (high)' means that a high accuracy is better than a medium one.
- Positive judgements are marked **green**, medium **blue**, negative **red** together with their in-between colours **turquoise** and **lilac**.
- Because many approaches have not been explained in detail or have not been extensively evaluated in the source papers, several positions marked *cursive* must be assumed from the underlying technologies, e.g. data structures.
- Best judgements from each category are marked **bold**.
- These best judgements are added in the last table line 'Overall score', where all judgements were equally weighted.

It still must be mentioned that trade-offs between e.g. accuracy and approach complexity are irrefutable (see 4.8).

---

<sup>36</sup> a Java programming language based integrated development environment for rapid application development

**Table 1 Evaluation of approaches I**

Name / Category (Desired value)	Adaptive Model Checking	Probabilistic reasoning	PRISM	RETE 1	DPLL	CSP solvers
Approach	General Approach	General algorithms	Framework (probabilistic)	Algorithm (rule-based solver)	Algorithm	Algorithm
Paragraph link	5.2	5.3, 4.6	5.4	5.5	5.6.1	5.6.2
Application/Heuristics	Verification/Model Learning; AMC/CEGAR/BBC	probabilities	BDD, MTBDD; DTMC, CTMC, MDP	Forward chaining	SAT for CNF; backtracking	CSOP on feature models
Accuracy (high)	medium	low to medium	medium	medium to high	high	high
Approach complexity (low)	e.g. polynomial	medium	roundabout medium (approach dependant)	lower to medium; high memory effort	lower to medium	up to exponential; high
Input data effort (low)	medium	high	High to medium	medium	medium to high	low to medium
Flexibility (high)	medium	high	unknown	high to medium	low to medium	unknown
Maturity (high)	medium	high (approach dependant)	high (approach dependant)	high	high	medium
Documentation (much)	papers	papers	many case studies (but in another context), tutorials; web page	papers, tutorials	papers, tutorials	papers
Open Source (yes)	no implementation focussed	No concrete implementation focussed	GPL	yes	yes	yes
Eclipse compatible (yes)	-	-	Java; CUDD; yes via interface	-	-	-
score	1	1	4	4	4	2

Scale: undesirable - worse than medium - medium desirable - better than medium - desirable

**Table 2 Evaluation of approaches II**

Name / Category (desired value)	CUDD	NUSMV 2	CBR	Divide & conquer	FAMA	
Approach	Framework (integrated in NUSMV 2, PRISM etc.)	Framework	General approach	Algorithm & strategies concept	Framework	
Paragraph link	5.6.3	5.6.3	5.8	5.10	5.11	
Application/Heuristics	BDD; ADD; ZDD	BDD, SAT (based on CUDD) DLL algorithm	Cases	Divide & conquer	BDD; SAT; CSP; expandable; uses FMs; several querying tasks	
Accuracy (high)	high	high to medium	high to low (algorithm dependant)	medium to high	high to medium (solver dependant)	
Approach complexity (low)	medium to high	medium to low	medium to low; potentially high memory effort	medium to low; can be linear	low to medium (solver & task dependant)	
Input data effort (low)	medium to high	medium to high	high	medium to high	medium to high (solver dependant)	
Flexibility (high)	unknown	unknown	high	medium	high	
Maturity (high)	high	high	high, depends on approach	medium, but has a high potential	medium	
Documentation (much)	web site; tutorials, papers	web site, tutorials, papers	Papers etc., depends on approach	paper; potentially author contacts	papers; implementation, author contacts, user guide	
Open Source (yes)	GPL	LGPL v.2.1	depends on specific approach	no implementation at hand	LGPL	
Eclipse compatible (yes)	interfaceable via JavaBDD	C++; via interface	-	-	-	
score	6	5	3	3	6	

Scale: undesirable - worse than medium - medium desirable - better than medium - desirable

**Table 3 Evaluation of approaches III**

Name / Category (desired value)	Artificial life	Artificial neural networks (ANN)	Evolutionary computation	Genetic algorithms	Ant colony optimisation	Particle swarm optimisation
Approach	Algorithms	Algorithms	Algorithms	Algorithms	Algorithms	Algorithms
Paragraph link	5.9	5.9.1	5.9.2	5.9.2	5.9.2	5.9.2
Application/Heuristics	Digital evolution	Machine learning	Functional optimization	Functional optimization	Functional optimization	Functional optimization
Accuracy (high)	high to medium	high to medium (depends on training)	high to medium (depends on parameters and encoding)	high to medium (depends on parameters and encoding)	high to medium	high to medium
Approach complexity (low)	medium to low	low	medium to low	low	medium to low	low
Input data effort (low)	high; hard to encode	high; hard to encode	medium; expressed as optimization problem; requires numerical encoding or cost function	medium; expressed as optimization problem; requires numerical encoding or cost function	medium; expressed as optimization problem; requires numerical encoding or cost function	medium; expressed as optimization problem; requires numerical encoding or cost function
Flexibility (high)	high	medium	high	high	high	high
Maturity (high)	medium to high	high	high	high	medium	medium
Documentation (much)	books + research paper	books + research paper	books + research paper	books + research paper	books + research paper	books + research paper
Open Source (yes)	-	several libraries available	several libraries available	several libraries available	several libraries available	-
Eclipse compatible (yes)	-	Java; C++; via interface	Java; C++; via interface	Java; C++; via interface	Java; C++; via interface	C++; via interface
Overall score	2	5	6	7	4	3

Scale: undesirable - worse than medium - medium desirable - better than medium - desirable

## 6.4 Recommendations

Time-efficiency is a hard constraint for reasoning in DiVA. Models conversions for heterogeneous models, e.g. from a Models@Runtime representation to a feature model and from the results back to a Models@Runtime model, can be performed efficiently if both the models are semantically equivalent. In other cases, conversion can be NP-Hard, e.g. when converting to an ROBDD representation. However, it is being assumed that the ARF's repeated calculations over the target representation result in an overall increased time-efficiency.

As such, the main opportunities to gain time-efficiency are offered on the modelling stage and by the selected reasoning algorithms. The selection of an underlying framework to implement new respectively reuse exiting algorithms is motivated by approach complexity, maturity, flexibility, documentation and Eclipse compatibility.

### 6.4.1 Features and feature value ranges

Besides reconfiguration resources like calculation power or memory consumption, information about which resources on the one hand are offered (e.g. network access), on the other hand are claimed by components are integrated in the runtime model.

Parameterisation of models increases the combinatorial complexity problem [33, p.108ff]. Therefore, two categories of resource selections are proposed to consider:

- **binary selectable** via a *yes/ no* decision, e.g. *switch on/ switch off*,
- **discrete selectable**, e.g. 'use 100 KB of memory', perhaps as a range of maximum and minimum number

but **not continuously selectable**, as this would increase the amount of variants remarkably, depending on the range of continuous values.

Continuous values on a digital computer are in fact not infinitively analogous but are represented by fine grained floating point values. These still leave a big numerous range which can potentially appear in a variant. Similarly, a big amount of discrete values would result in a vast amount of variants, as the same problem will appear. To reduce complexity here, it is advisable to limit the number of possible values in modelling. Another possibility on reducing pseudo-infinitely many numbers is a coarse discretisation at run-time.

At the modelling stage, the *reduction of values* to binary selectable values or a limited range of discrete selectable values helps to reduce the variation space. This, on the other hand, reduces expressiveness, but will for many features be appropriate, e.g. it is rarely necessary to have dozens of settings for display brightness.

Dependent features should be combined at the architectural level to reduce complexity. Via a heuristic combination, the D&C approach (which is regrettably still immature, see 5.10) aims at reducing complexity automatically down to linearity in optimal cases in a pre-processing step.

The space of possible configurations can be reduced, if constraints are applied to features, e.g.<sup>37</sup>

- **Multiplicities/ bounds**: lower and upper number that a feature is allowed to appear in the final configuration.
- **Dependencies**: tells on which other feature a feature depends on.

---

<sup>37</sup> according to WP2's analysis of the CAS case study from the middle months of 2008

- *Available*: in which context situation a feature may be applied.
- *Required*: in which context situation a feature must be used.

In applying *priorities* to features, their calculation order can play a role in anytime algorithms that then start reasoning on higher prioritised features to assure that the most important aspects are treated first.

#### 6.4.2 Data structures

A variability model is needed to structure features and their variability. In WP4, orthogonal variability of components should generally be represented in a *feature model* as the basis for calculations on variants (see 4.10). These are then to be converted to data structures necessary for specific algorithms.

While there are different representations for variant models nowadays, feature models which are explained deeply as CONSUL models in [13] are recommended, because this formulation has proven in practice to be a powerful yet clearly laid out model representation. The *pure::variants tool* of pure-systems already contains an editor for parameter adaptation [33, p.24f] of feature models. This involves the modification of variables that determine program behaviour together with the mathematical formulation of feature constraints, restrictions and relations/ dependencies. The approach of CONSUL models divides the variability creation task into problem space and solution space. The editor is useful for experimentation due to its flexibility and high maturity. However, to stay Open Source, an *additional editor* will be provided based on presumably EMF/ Ecore; all APIs will be open.

#### 6.4.3 Frameworks

The *FAMA framework* (see 5.11) reached a good score in evaluation. It offers basic reasoning algorithms like BDD, SAT and CSP solvers that are optimised for specific reasoning queries. Additionally, it can be extended by more solver algorithms which offer, together with its Eclipse plugin implementation, a solid base for experiments and an iterative development cycle. Iteration will be necessary for optimisation reasoning in a specific problem domain, as solver's efficiency depends on an actual problem as had been said by many sources throughout this survey. While *BDDs* and *SAT* are more general approaches, the *CSP* solver may be useful for the DiVA reasoning tasks as it constraints variables (see 5.6.2); constraints are an integral part of reasoning in DiVA. However, some work needs to be put into FAMA to develop it to the needs of the DiVA project. FAMA is in use by Universidad Politécnica de Valencia and the University College Dublin.

The *PRISM framework* of paragraph 5.4 seems a mighty alternative; however, its extensive source of use cases is mostly relevant for communication protocols.

The highly mature *CUDD framework* offers *BDD*, *ADD* and *ZDD*. It is less easy to integrate into the DiVA platform as it is written in C/ C++. However, it may be interfaced by JavaBDD. During development, it can be considered as an extension top FAMA; it can be checked whether it offers reasoners that can help in defeating complexity in a specific problem context as an extension to FAMA.

#### 6.4.4 Algorithms

In the field of algorithms, a combination of *Cognitive Approaches* like genetic algorithms and evolutionary computation (see 5.9) can be a sensible extension to FAMA, as these heuristics reached a high score in evaluation. They engage in finding best-fitted adaptations or can support learning. The functional optimisation that these approaches carry out usually offers, after a shorter amount of iterations and therefore less time, less accurate results and, after a bigger number of iterations, increasingly precise results. Such behaviour can be exploited for anytime

algorithms. However, the challenge is to formulate appropriate evolutionary optimisation functions for the selection and mutation of entities.

A cognitive approach should be complemented by approaches from other fields of reasoning. The long-established and widely used *RETE I algorithm* (see 5.5) may speed up reasoning remarkably by the costs of high memory consumption which appears in its knowledge propagation. It is useful for a system which changes slowly and not abruptly over time.

A newly developed approach in the field of dynamic adaptive systems is the still immature but actually further developed *Divide & Conquer approach* described in section 5.10. It implements, together with strategies and heuristics, pre-processing of input data in a sophisticated way. If it is possible to collaborate with the authors and under the assumption that more details will be published soon it may be a promising approach for tackling reasoning complexity. However, this approach is, due to many open questions, at the time being only a potential candidate.

#### 6.4.5 Pre-processing

A pre-processing as in *Combinatorial Testing* (see 4.9.2) can help in reducing reasoning complexity while generating test data that covers interactions between variables. For systems that have large input spaces and that should adapt in a large number of configurations, it can potentially reduce the number of configurations that have to be reasoned upon.

However, before applying this technique, it makes sense to *combine dependent variables* at the architectural level. Also, the above mentioned algorithms should be studied and exploited, as adding more and more components to the ARF may easily exceed DiVA's capabilities.

Reasoning in an AOM environment which is dependent from the aspects weaving order may benefit from the *Confluence Analysis* approach of 4.11 to establish a valid weaving order. However, it must be established in subsequent work if the task of reasoning on weaving orders is part of the WP4 work or if this task will be integrated into the WP3 framework, as the ARF is not engaged in weaving aspects but in reasoning over valid defined model definitions.

## Glossary

### A

ACO	Ant colony optimisation, an optimisation approach that applies a swarm of agents to a reasoning task.
ADD	Algebraic Decision Diagram
AI	Artificial Intelligence is a field of engineering machine intelligence.
AL	Artificial Life, a discipline where evolutionary concepts are used as a base of computation.
AMC	Adaptive Model Checking, a methodology of checking for inconsistencies between a system and its corresponding model. → BBC
ARF	Adaptation Reasoning Framework
ATL	Alternating-time Temporal Logic, a temporal extension of Coalition Logic that is used to reason in game-like multi-agent scenarios.
ATPG	Automatic Test Pattern Generation
ANN	Artificial Neural Network

### B

BBC	Black Box Checking, an automatic verification mechanism, where verification starts without any model; the model is obtained by repeated verification attempts → AMC
BDD	Binary Decision Diagram, useful as →SAT solver
BT	Backtracking, an algorithm for finding ideally all solutions to a computational problem by abandoning ‘dead branches’ in a decision tree.

### C

CE	Counter-example
CEGAR	Counter-Example Guided Abstraction Refinement, a model checking approach.
CBR	Case-based Reasoning
CNF	Conjunctive normal form in Boolean logic
CP	Constraint Programming
CSP	Constraint Satisfaction Problem
CSOP	Constraint Satisfaction Optimisation Problem
CTL	Computation Tree Logic, a temporal logic
CTMC	Continuous-Time Markov Chain

**D**

D&C	Divide & Conquer
DBMS	Database Management System
DiVA	Acronym for the research project "Dynamic Variability in complex, adaptive Systems"
DoW	"Description of Work" document [25] for →DiVA
DTMC	Discrete-Time Markov Chain

**E**

EXPTIME	Complexity class that holds problems which are solvable by a deterministic Turing machine in exponential time.
---------	----------------------------------------------------------------------------------------------------------------

**F**

FTP	File Transfer Protocol, a network communication protocol with lower overhead than →TCP.
FSM	Finite State Machine
FOPC	First-Order Predicate Calculus

**G**

GA	Genetic Algorithm
GPL	GNU General Public Licence, an Open Source licensing model

**H****I/J**

IBL	Instance-based Learning
IDE	Integrated Development Environment

**K**

KB	Knowledge Base
----	----------------

**L**

LGPL	GNU Lesser General Public Licence, an Open Source licensing model
LISP	is a logic-based programming language.
LTL	Linear Temporal Logics

**M**

MDP	Markov Decision Process
ML	Modal Logics, a class of logics to represent modes of truth.
MTBDD	Multi-Terminal Binary Decision Diagram

**N**

NP	Complexity class that holds problems which are solvable by a non-deterministic Turing machine in polynomial time $\rightarrow$ NP-complete
NP-complete	Complexity class that holds the hardest problems in the class of $\rightarrow$ NP.

**O****P**

PE	Processing Element
PCTL	Probabilistic Temporal Logics
PDAG	Propositional Acyclic Graph
PR	Probabilistic Reasoning
PROLOG	is a logic-based programming language.
PSO	Particle Swarm Optimisation, an evolutionary algorithm.

**Q****R**

RML	Reactive Modules Language, a simple but expressive formalism for specifying game-like distributed system models, used as the model specification language for several model checkers, e.g. MOCHA.
-----	---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

**S**

SAT	Satisfiability Problem in propositional logics that decides whether a logical formula can be fulfilled to 'true'.
SKB	Symbolic Knowledge Base, a $\rightarrow$ KB with symbolic content used by a symbolic reasoner.
SPL	Software Product Line
SRML	Simple Reactive Modules Language, a simplified version of $\rightarrow$ RML.

**T**

TCP	Transmission Control Protocol, a network communication protocol family.
TL	Temporal Logics, extended $\rightarrow$ ML to allow time-dynamic reasoning in a formal system.

U

UDP User Datagram Protocol, a network communication protocol with lower overhead than →TCP.

UML Unified Modelling Language, a graphical founded software engineering language.

VW

WP Work Package

X

XML eXtensible Markup Language, a general-purpose specification for creating custom markup languages.

YZ

ZDD Zero-suppressed binary Decision Diagram.

## References

- [1] Alfaro, de L. (2001): "jMocha: A Model-checking Tool that Exploits Design Structure". Proceedings of the 23rd Annual IEEE/ACM International Conference on Software, IEEE Computer Society Press, pp. 835-836, January, 2001.
- [2] Alur, R. Henzinger, T. (1996): "Reactive modules". In Proc. LICS'96, 1996.
- [3] Andersen, H. R. (1997): "An Introduction to Binary Decision Diagrams". Lecture Notes. Technical University of Denmark.
- [4] Angluin, D. Punyakanok, V. (1978): "Learning Regular Sets from Queries and Counterexamples". In: Information and Computation, 75, 87-106.
- [5] Antoniol, G. Penta Di, M. Harman, M. (2005): "Search-Based Techniques Applied to Optimisation of Project Planning for a Massive Maintenance Project". Proceedings of the 21st IEEE International Conference on Software Maintenance ISBN 0-7695-2368-4, IEEE Computer Society: 240-249.
- [6] Alur, R. Henzinger, T. A., and Kupferman, O. (2002): "Alternating-time temporal logic." Journal of the ACM, 49(5):672–713, September 2002.
- [7] Asikainen, T. Soininen, T., and Männistö, T. (2004): "A koala-based approach for modelling and deploying configurable software product families". In van der Linden, F. (editor), Software Product-Family Engineering, 5th International Workshop, 2003, volume 3014 of Lecture Notes in Computer Science, pages 225 – 249. Springer.
- [8] Baudry, B. Fleurey, F. et al. (2005): "Automatic test case optimisation: a bacteriologic algorithm". Software, IEEE 22(2): 76-82.
- [9] Beierle, Chr. Kern-Isberner, G. (2006): "Methoden wissensbasierter Systeme". 3. Auflage, Vieweg, Wiesbaden, Germany.
- [10] Beinlich, I. A. Suermondt, H. J. Chavez, R. M. & Cooper, G. F. (1989): "The ALARM Monitoring System: A Case Study with Two Probabilistic Inference Techniques for Belief Networks". Second European Conference on Artificial Intelligence in Medicine, London.
- [11] Benavides, D. Ruiz-Cortés, A. Trinidad, P. (2005). "Automated reasoning on feature models". LNCS, Advanced Information Systems Engineering: 17th International Conference, CAiSE 2005, 3520:491–503, 2005.
- [12] Benavides, D. Segura, S. Trinidad, P. Ruiz-Cortés, A. (2007): "FAMA: Tooling a framework for the automated analysis of feature models". In Proceeding of the First International Workshop on Variability Modelling of Software intensive Systems (VAMOS) 2007, p.129-134.
- [13] Beuche, D. (2003): "Composition and Construction of Embedded Software Families". PhD thesis, University of Magdeburg, Germany.
- [14] Bollig, B. Wegener, I. (1996): "Improving the Variable Ordering of OBDDs Is NP-Complete". IEEE Transactions on Computers, 45(9):993—1002, September 1996.
- [15] Charniak, E. McDermott, D. (1985): "Introduction to artificial intelligence". Addison-Wesley Longman Publishing Co., Inc.
- [16] Chauvel, F. Barais, O. Plouzeau, N. Borne, I., Jezequel, J.-M. (2008): "Qualitative adaptation policies". Slide show from DiVA Technical Meeting Lancaster by Fleury, F., Sept. 16, 2008. IRISA, Université de Rennes, France.

- [17] Clarke, E. Giunchiglia, E. Giunchiglia, F. Marco Pistore, M. et al. (2002): "NuSMV 2: An OpenSource Tool for Symbolic Model Checking". In: Lecture Notes In Computer Science; Vol. 2404. Proceedings of the 14th International Conference on Computer Aided Verification, 359 - 364.
- [18] Clocksin, W. F. Christopher, S. M. (1984): "Programming in Prolog", 2nd ed., Springer-Verlag, New York, Inc.
- [19] Cohen, D. M. Dalal, S. R. et al. (1997): "The AETG System: An Approach to Testing Based on Combinatorial Design". IEEE Transactions on Software Engineering 23(7): 437 - 444.
- [20] CORPORATE PDP Research Group, Ed. (1986a): "Parallel distributed processing: explorations in the microstructure of cognition", vol. 1: foundations, MIT Press.
- [21] CORPORATE PDP Research Group, Ed. (1986b): "Parallel distributed processing: explorations in the microstructure of cognition", vol. 2: psychological and biological models, MIT Press.
- [22] Czarnecki, K. Eisenecker, U.W. (2000): "Generative Programming: Methods, Techniques and Applications". Addison-Wesley, August 2001.
- [23] Davis, M. Logemann, G. Loveland, D. (1962): "A machine program for theorem-proving". Communications of the ACM archive, Volume 5 , Issue 7 (July 1962), Pages: 394 - 397.
- [24] Dehlen, V. (2008): "WP2: Methodology status - creating the adaptation model". Powerpoint presentation, DiVA consortium meeting, Lancaster, England.
- [25] DiVA (2007): "Annex I - Description of work". DiVA project.
- [26] DiVA (2008): "Understanding DiVA - Kick-off Lillehammer 13-14/3 2008. PowerPoint presentation.
- [27] DiVA (2009): "Case Study Specification and Requirements". DiVA project.
- [28] Dorigo, M. Maniezzo, V. et al. (1996): "Ant system: optimisation by a colony of cooperating agents". Systems, Man, and Cybernetics, Part B, IEEE Transactions on 26(1): 29-41.
- [29] Eberhart, R. Yuhui, S. (2001): "Particle swarm optimisation: developments, applications and resources". Evolutionary Computation, 2001. Proceedings of the 2001 Congress on.
- [30] Emerson, E. A. (1990): "Temporal and modal logic". In: Handbook of Theoretical Computer Science, p. 995-1072, Elsevier.
- [31] Faltings, B. Freuder, E.: "Configuration". IEEE Intelligent Systems, 13(4), 1998.
- [32] Finkelstein, A. Harman, M. et al. (2008): "'Fairness Analysis' in Requirements Assignments". Proceedings of the 16th IEEE International Requirements Engineering Conference (RE '08). Barcelona, Spain, IEEE.
- [33] Floch, J. (2006): "Theory of Adaptation". Deliverable D2.2, SINTEF, The MADAM - Project.
- [34] Forgy, Ch. (1982): "Rete: A Fast Algorithm for the Many Patterns/Many Objects Match Problem". Artif. Intell. 19(1): 17-37 (1982).
- [35] Gallier, J. H. (2003): "Foundations of Automatic Theorem Proving". University of Pennsylvania, USA.
- [36] Goldberg, D. E. (1989): "Genetic Algorithms in Search, Optimisation and Machine Learning". Addison-Wesley Longman Publishing Co., Inc.
- [37] Gold, N. Harman, M. et al. (2006): "Allowing Overlapping Boundaries in Source Code using a Search Based Approach to Concept Binding". Proceedings of the 22nd IEEE International Conference on Software Maintenance. ISBN 0-7695-2354-4, IEEE Computer Society: 310-319.

- [38] Goldsby, H. J. Cheng, B. H. C. et al. (2008): "Digital Evolution of Behavioral Models for Autonomic Systems". *Autonomic Computing*, 2008. ICAC '08. International Conference on.
- [39] Goldsby, H. Cheng, B. H. C. (2008): "Automatically Generating Behavioral Models of Adaptive Systems to Address Uncertainty". *Model Driven Engineering Languages and Systems*: 568-583.
- [40] Goss, S. Aron, S. et al. (1989): "Self-organized shortcuts in the Argentine ant". *Naturwissenschaften* 76(12): 579-581.
- [41] Greiner, R. Darken, Chr. Santoso, Iwan (2001): "Efficient Reasoning.". University of Alberta. ACM Computing Surveys, USA.
- [42] Groce, A. Peled, D. Yannakakis, M. (2002): "Adaptive model checking". In: *Tools and Algorithms for Construction and Analysis of Systems*, p.357-370. Springer Verlag.
- [43] Grønmo, R. Sørensen, F. Møller-Pedersen, B. Krogdahl, S. (2008): "Semantics-based Weaving of UML Sequence Diagrams". *International Conference on Model Transformation - Theory and Practice of Model Transformations (ICMT)*, Zürich, Switzerland. July 2008.
- [44] Grønmo, R. Sørensen, F. Møller-Pedersen, B. Krogdahl, S. (2008): "A Semantics-based Aspect Language for Interactions with the Arbitrary Events Symbol". *The European Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA)*, Berlin, Germany. June 2008.
- [45] Harman, M. (2006): "Search-Based Software Engineering for Maintenance and Reengineering". *Proceedings of the Conference on Software Maintenance and Reengineering*. ISBN 0-7695-2536-9, IEEE Computer Society: 311.
- [46] Hinton, G. E. (1992): "How neural networks learn from experience." *Sci Am* 267(3): 144-51.
- [47] Hodges, W. (1977): "Logic: An Introduction to Elementary Logic". Penguin.
- [48] Hoek van der, W. Lomuscio, A. Wooldridge, M.: (2006) "On the complexity of practical ATL model checking". *Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*, p.201 - 208, Japan.
- [49] Hüllermeier, E. (2007): "Case-Based Approximate Reasoning". *Series: Theory and Decision Library B*, Vol. 44. Springer. Dordrecht, The Netherlands.
- [50] Kaviani, N. Mohabbati, B. Gasevic, D. and Fink, M. (2008): "Semantic annotations of feature models for dynamic product configuration in ubiquitous environments". In *4th International Workshop on Semantic Web Enabled Software Engineering*, 2008.
- [51] Kennedy, J. Eberhart, R. (1995): "Particle swarm optimisation". *Neural Networks*, 1995. Proceedings., IEEE International Conference on.
- [52] König, B. Kozioura, V. (2006): "Counterexample-guided Abstraction Refinement for the Analysis of Graph Transformation Systems". Universität Stuttgart, Germany. [http://elib.uni-stuttgart.de/opus/volltexte/2006/2533/pdf/TR\\_2006\\_01.pdf](http://elib.uni-stuttgart.de/opus/volltexte/2006/2533/pdf/TR_2006_01.pdf)
- [53] Kondrak, G. Beek, van P. (1997): "A Theoretical Evaluation of Selected Backtracking Algorithms". *Artificial Intelligence* Vol. 89, p.541-547, 1997.
- [54] Kruse, R. (2006): "Wissensbasierte Systeme". *Scriptum*, chapter 11. University of Magdeburg, Germany.
- [55] Kuhn, R. Lei, Yu Kacker, R. (2008). "Practical Combinatorial Testing: Beyond Pairwise." *IEEE IT Professional* 10(3): 19 - 23.
- [56] Kwiatkowska, M. (2003): "Model checking for probability and time: from theory to practice". <http://www.prismmodelchecker.org/>

- [57] Kwiatkowska, M. Norman, G. Parker, D. (2002): "PRISM: Probabilistic Symbolic Model Checker". University of Birmingham, United Kingdom.
- [58] Lakhotia, K. Harman, M. et al. (2007): "A multi-objective approach to search-based test data generation". Proceedings of the 9th annual conference on Genetic and evolutionary computation. ISBN 978-1-59593-697-4. London, England, ACM: 1098-1105.
- [59] Lawrence, J. (1994): "Introduction to Neural Networks". California Scientific Software.
- [60] Morin, B. et al (2008): "An Aspect-oriented and Model-Driven Approach for managing Dynamic Variability" (draft version). IRISA/ INRIA Rennes, France.
- [61] Freddy Munoz, F. Baudry, B. (2008): "Validation challenges in model composition: The case of adaptive systems". In the proceedings of: First International Workshop on Challenges in Model-Driven Software Engineering, 2008. INRIA/ IRISA, France.
- [62] Nicolis, S. C. Despland, E. Dussutour, A. (2008): "Collective decision-making and behavioral polymorphism in group living organisms". *Journal of Theoretical Biology* 254(3): 580-586.
- [63] Nilsson, N. J. (1980): "Principles of artificial intelligence". Morgan Kaufmann Publishers Inc.
- [64] Oberle, D. Eberhart, A., Staab, St., and Volz, R. (2004): "Developing and managing software components in an ontology-based application server". In *Middleware '04: Proceedings of the 5th ACM/IFIP/USENIX international conference on Middleware*, pages 459–477, New York, NY, USA, 2004. Springer-Verlag New York, Inc.
- [65] Oberle, D. Staab, St. Studer, R. and Volz, R. (2005): "Supporting application development in the semantic web". *ACM Trans. Interet Technol.*, 5(2):328–358, 2005.
- [66] Ofria, C. Wilke, C. O. (2004): "Avida: a software platform for research in computational evolutionary biology". *Artificial Life* 10: 191-229.
- [67] Pidcock, W.: 'What are the differences between a vocabulary, a taxonomy, a thesaurus, an ontology, and a meta-model?' Technical report, <http://www.metamodel.com/article.php?story=20030115211223271>, 2003.
- [68] D. Plump, (2005): 'Confluence of Graph Transformation Revisited', In 'Processes, Terms and Cycles: Steps on the Road to Infinity: Essays Dedicated to Jan Willem Klop on the Occasion of His 60th Birthday', *Lecture Notes in Computer Science* 3838, pages 280-308, Springer-Verlag, 2005.
- [69] Pohl, K. Böckle, G. van der Linden, F. (2005): "Software Product Line Engineering – Foundations, Principles, and Techniques". Springer, Heidelberg, Germany.
- [70] Production System Technologies Website (2008). Accessed 2008-11-07. <http://www.pst.com/rete2.htm>
- [71] Qiang, G. Hierons, R. M. et al. (2007): "Heuristics for fault diagnosis when testing from finite state machines: Research Articles." *Softw. Test. Verif. Reliab.* ISSN 0960-0833 17(1): 41-57.
- [72] Robinson, J. A. (1965): "A machine-oriented logic based on the resolution principle". *JACM* 12, p.23-41.
- [73] Rumelhart, D. E. Zipser, D. (1986): "Feature discovery by competitive learning". *Parallel distributed processing: explorations in the microstructure of cognition*, vol. 1: foundations, ISBN 0-262-68053-X, MIT Press: 151-193.
- [74] Rumelhart, D. E. Smolensky, P. et al. (1986): "Schemata and sequential thought processes in PDP models". *Parallel distributed processing: explorations in the microstructure of cognition*, vol. 2: psychological and biological models ISBN 0-262-13218-4, MIT Press: 7-57.

- [75] Sabin, D. Weigel, R. (1998): "Product configuration frameworks - a survey". *IEEE Intelligent Systems*, 13(4):42–49, 1998.
- [76] Scholz, U. Rouvoy, R. (2008): "Divide and Conquer – Organizing Component-based Adaptation in Distributed Environments". *Proceedings of the First International DisCoTec Workshop on Context-aware Adaptation Mechanisms for Pervasive and Ubiquitous Services (CAMPUS 2008)*.
- [77] Schrijver, A. (2003): "A Course in Combinatorial Optimisation". University of Amsterdam, the Netherlands.
- [78] Sebase (2008): "Software Engineering By Automated SEArch". Retrieved 20 October, 2008 from <http://www.sebase.org>.
- [79] SeCSE Project. Deliverable a1.d9.1 - qos ontology implementation. Technical report, 2007.
- [80] Selman, B. Kautz, H. (1996): "Knowledge compilation and theory approximation". *Journal of the ACM* 43, 193–224.
- [81] Shahri, H. H. Hendler, J. and Porter, A. (2007): "Software configuration management using ontologies". In *Proceedings of the 3rd International Workshop on Semantic Web Enabled Software Engineering at the 4th European Semantic Web Conference (ESWC'07)*, 2007.
- [82] "Method and Device for comparing technical systems with each other". WIPO, International Application No.: PCT/DE1999/003500
- [83] Sieling, S.: (2002): "The nonapproximability of OBDD minimization." *Information and Computation* 172, 103–138. 2002.
- [84] Smyth, B. (1993): "Complexity of adaptation in real-world case-based reasoning systems". In *Proc 6th Irish Conference on AI & Cognitive Science*. Trinity College Dublin, Ireland.
- [85] Somenzi, F. (2008): "CUDD: CU Decision Diagram Package. Release 2.4.1". Weblink, 2008-10-10: <http://vlsi.colorado.edu/~fabio/CUDD/>. University of Colorado at Boulder, USA.
- [86] Sutton, R. S. (1984): "Temporal credit assignment in reinforcement learning". University of Massachusetts Amherst: 223.
- [87] Sutton, R. S. (1988): "Learning to Predict by the Methods of Temporal Differences". *Machine Learning*, ISSN 0885-6125 3(1): 9-44.
- [88] Tai, K.-C. Lei, Y. (2002): "A Test Generation Strategy for Pairwise Testing". *IEEE Transactions on Software Engineering* 28(1): 109 - 111.
- [89] Volz, R. Oberle, D. Staab, St. and Motik, B. (2003): "Kaon server - a semantic web management system". In *Alternate Track Proceedings of the Twelfth International World Wide Web Conference, WWW2003, Budapest, Hungary, 20-24 May 2003*. ACM, 2003.
- [90] Wasserman, P. D. (1989): "Neural computing: theory and practice". Van Nostrand Reinhold Co.
- [91] Wasserman, P. D. (1993): "Advanced Methods in Neural Computing". John Wiley & Sons, Inc.
- [92] Watkins, C. J. C. H. (1989): "Learning from Delayed Rewards". Computer Science. London, King's College. PhD.
- [93] Werbos, P. J. (1994): "The roots of backpropagation: from ordered derivatives to neural networks and political forecasting". Wiley-Interscience.
- [94] [www.wikipedia.org](http://www.wikipedia.org), plus search phrase.
- [95] Winston, P. H., Horn, B. K. (1988). "LISP". Addison-Wesley Longman Publishing Co., Inc.

- [96] Winston, P. W. (1992): "Artificial intelligence". 3rd ed., Addison-Wesley Longman Publishing Co., Inc.
- [97] Zhang, J. Cheng, B.H.C. (2006): "Model-based development of dynamically adaptive software". In ICSE '06: Proceedings of the 28th international conference on Software engineering. 2006, ACM, p. 371–380. Shanghai, China.
- [98] Zhou, Y. Zhao, Q. and Perry, M. (2005): "Reasoning over ontologies of on demand service". In IEEE International Conference on e-Technology, e-Commerce and e-Service, 2005. EEE '05, 2005.
- [99] Zilberstein, S. (1993): "Operational Rationality through Compilation of anytime algorithms". Dissertation, University of California, Berkeley, USA.
- [100] Cormen, Th. Leiserson, Ch. Rivest, R. (2001): "Introduction to Algorithms". Second Edition. MIT Press.
- [101] Smid, M. (2001): "Theoretische Informatik für Computervisualisten". 3rd edition. Fakultät für Informatik, Otto-von-Guericke-Universität Magdeburg, Germany.

## Appendix A - DiVA requirements specification - reasoning specific

Requirements assembled in [27] define DiVA's development needs. In the following, a subset which is relevant for the ARF is listed:

Requirements number (previous, new)	Requirement name (previous, new)	Category	Priority	Description (shortened)	Consequences for reasoning
R-10	Support of technology heterogeneity.	Runtime	high	Ability to assemble heterogeneous systems, implemented with different technologies, needed.	Common interface definitions necessary.
R-11	(Need to reduce complexity of variable systems at design time).	Modelling	high	Support of exponential growth of the number of potential system configurations.	Treat complexity explosion problem.
R-12	Technical infrastructure/ platform support.	Non-functional	high	DiVA technology must support the case study's technical infrastructure as specified in D6.1.	Must also be fulfilled by reasoning.
R-13	Ease of use.	Non-functional	low	Easy use of technology and tools, including documentation, integrated tool support including user guidance.	Only secondarily important during runtime because reasoning is hidden to users; but important during development cycle



Requirements number (previous, new)	Requirement name (previous, new)	Category	Priority	Description (shortened)	Consequences for reasoning
R-14	Support specification of variability at different phases of development lifecycle).	Modelling	high	Composition policies include - dependencies between components / services (X only in conjunction with Y, X not in conjunction with Y, versions, ...) - Cardinalities (i.e. "there must at least be one instance of a certain service type") - Specification of security / privacy constraints.	Take these in account during reasoning
R-15	Support for Dynamic Reasoning (best-fit selection).	Runtime	high	Analysis of the variability dimensions, properties and adaptation rules. Selection of best-fit. At least possible configurations.	The central reasoning task.
R-16	Compliance to standards.	Non-functional	high	Notations, description languages etc. should be compliant to existing standards as far as possible.	Applies to reasoning framework.
R-19	Support of verification of adaptation models.	Verification/ Validation	high	Ability to check correctness, consistency of a configuration that adapts to the context. - Ability to check if the selected configuration is the best one. - Dynamic composition policy checking: Before services are added/ removed/ updated policies must be checked whether system remains in consistent state.	Central reasoning tasks.



Requirements number (previous, new)	Requirement name previous or (previous, new)	Category	Priority	Description (shortened)	Consequences for reasoning
R-20, R-21	Support of functional & non-functional description of variants.	Modelling	medium	Enable designers to describe functional properties & QoS properties of each variant, i.e. information about performance, optimality, required resources.	Reasoning interprets these descriptions.
R-18	Specification of adaptation policies).	Modelling	medium	Ability to specify required configurations according to context (in terms of required functionalities of the system and required QoS).	Reasoning interprets these specifications.
R-9	Support of variability validation of the system.	Verification/ Validation	high	Once the configuration model of the system that adapts to the context is selected, it must be validated.	Central validation task.
R-22	Support of reasoning about QoS properties.	Runtime	high	Reasoning framework is able to take into account QoS properties. For example, prioritize performance.	Extended reasoning functionality.



Ω