Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS

Adam Huječek

Meta-Adaptation Strategies for Adaptation in Cyber-Physical Systems

Department of Distributed and Dependable Systems

Supervisor of the master thesis: Doc. RNDr. Tomáš Bureš, Ph.D.
Study program: Informatics
Study branch: Software Systems
Prague 2015

Acknowledgement

I would like to thank everyone whose lectures and classes I had the honor to attend to during my studies at Faculty of Mathematics and Physics at Charles University in both Bachelor and Master study programs. They developed my interest in computer sciences and made this work possible.

Namely, I thank my supervisor Tomáš Bureš for his help and support and for introducing me to the great team which has been working on the IRM and DEECo for years making them well-thought, deep and complex projects. It is a pleasure to work with them. My deepest gratitude goes to Ilias Gerostathopoulos whose advice and guidance helped to overcome my inexperience in the world of research. His encouragement and enthusiasm proved to be exactly what I needed in the time of crisis. I am grateful to Dominik Škoda for fruitful collaboration on implementations of the meta-adaptation strategies as well as the rest of my co-authors Petr Hnětynka and František Plášil.

I would also like to thank my current employer for their patience, support and time flexibility that allowed me to continue my studies while gaining much needed experience in the field.

Above all, I thank my family and friends for creating a perfect environment for my studies and their unshakeable support.

Anotace

Název práce	*Meta-adaptační strategie pro adaptaci v cyber-physical systémech*
Autor	Adam Huječek
Katedra	Katedra distribuovaných a spolehlivých systémů Matematicko-fyzikální fakulta Univerzita Karlova v Praze
Vedoucí diplomové práce	Doc. RNDr. Tomáš Bureš, Ph.D. bures@d3s.mff.cuni.cz (+420) 2 2191 4236
Adresa	Katedra distribuovaných a spolehlivých systémů Univerzita Karlova v Praze Malostranské náměstí 25 118 00 Praha

Abstrakt

Při návrhu komplexních cyber-physical systémů je často nemožné dopředu předvídat všechny potencionální situace a připravit odpovídající taktiky pro adaptaci na změny v dynamickém prostředí, což velmi škodí robustnosti a spolehlivosti těchto systémů. Ze situací mimo očekávanou „obálku adaptability“ mohou povstat všemožné problémy, od poruchy jedné komponenty až po selhání celého systému. Samoadaptační přístupy jsou typicky omezeny na volbu taktiky z pevně dané množiny taktik. Meta-adaptační strategie posouvají hranice adaptability vlastní systému vytvářením nových taktik za běhu. Tato práce rozvíjí a implementuje vybrané meta-adaptace pro IRM-SA v jDEECo a vyhodnocuje jejich účinnost na experimentálním scénáři založeném na případové studii o koordinaci hasičů.

Klíčová slova

Meta-adaptační strategie, adaptační taktiky, cyber-physical systémy

Annotation

Title	*Meta-Adaptation Strategies for Adaptation in Cyber-Physical Systems*
Author	Adam Huječek
Department	Department of Distributed and Dependable Systems Faculty of Mathematics and Physics Charles University in Prague
Supervisor of the master thesis	Doc. RNDr. Tomáš Bureš, Ph.D. bures@d3s.mff.cuni.cz (+420) 2 2191 4236
Mailing address	Department of Distributed and Dependable Systems Charles University in Prague Malostranské náměstí 25 118 00 Prague, Czech Republic

Abstract

When designing a complex Cyber-Physical System it is often impossible to foresee all potential situations in advance and prepare corresponding tactics to adapt to the changes in dynamic environment. This greatly hurts the system's resilience and dependability. All kinds of trouble can rise from situations that lie beyond the expected "envelope of adaptability" from malfunction of one component to failure of the whole system. Self-adaptation approaches are typically limited in choosing a tactic from a fixed set of tactics. Meta-adaptation strategies extend the limits of system's inherent adaptation by creating new tactics at runtime. This thesis elaborates and provides implementations of selected meta-adaptation strategies for IRM-SA in jDEECo as well as their evaluation in a scenario based on a firefighter coordination case study.

Keywords

Meta-adaptation strategies, Adaptation tactics, Cyber-physical systems

Contents

1. Introduction 1
   1.1. Towards Cyber-Physical Systems 1
   1.2. Problem Statement 2
   1.3. Research Goals 3
   1.4. Structure 3
2. Background and Running Example 5
   2.1. Example of a Cyber-Physical System 5
   2.2. Introduction to IRM-SA and DEECo 7

Figure 2: Fragment of IRM-SA model of the case study. 8

   2.3. Case Study Description 10
   2.4. Invariant Refinement Method 12
   2.5. IRM-SA 14
   2.6. DEECo and jDEECo 17
3. Analysis / Goals Revisited 23
4. Meta-Adaptation Strategies 25
   4.1. Tactics Generated by Data Classification 26
   4.2. Tactics Generated by Period Adjusting 28
   4.3. Tactics Generated by Assumption Parameters Adjusting 32
5. Implementation as jDEECo Plugins 36
   5.1. Overall Picture 36
   5.2. MetaAdaptationPlugin 38

Figure 18: Package of MetaAdaptationPlugin inside its project. 38

   5.3. CorrelationPlugin 40

Figure 21: Package of CorrelationPlugin inside its own separated project.........41
5.4. (1+1)-ONLINE EA Plugins 42
5.4.1. PeriodAdaptationPlugin 46
Figure 25: Package of PeriodAdaptationPlugin inside its project. 46
5.4.2. AssumptionParameterAdaptationPlugin 49
6. Experimental Evaluation 52
6.1. Experiment Description 52
6.1.1. IRM-SA Model for the Case Study..52
Figure 32: simplified model of the case study for the evaluation 52
6.1.2. Scenario description 54
6.2. Experiment Evaluation...56
7. Related Work..58
8. Conclusion & Discussion 60
8.1. Improvements of the Current Implementation 60
8.2. Possible Extensions 61
Bibliography...63
List of Abbreviations 67
APA – Assumption Parameters Adjusting strategy...67
Attachments..68

1. Introduction

This chapter contains introduction into the context of Cyber-Physical Systems while explaining basic terms used throughout this work and its main goals as well as the structure of this thesis.

1.1. Towards Cyber-Physical Systems

With the arrival of low-cost mobile embedded devices capable of complicated networking and complex computing comes a great opportunity for large distributed systems which could significantly improve quality of life by providing high-value-added services. Webs of elements interconnected by wireless technologies bringing these services constitute Cyber-Physical Systems (CPS) which addresses various challenges both social and technical in the real world. One example is intelligent transportation system where road infrastructure consisting of traffic lights, digital road signs, car parks and recharge stations for electric automobiles communicates with nearby vehicles in order to achieve efficient usage of limited resources such as road capacity, fuel and parking space. By exchanging information the vehicles can group together into autonomous cooperating fleets. Another example is smart malls, where customers' preferences can be used to advertise favorable deals right on their smart phones as they go by and where crowd can be recommended optimal routes to destination shops during holiday shopping sprees which could also shorten long waiting lines. Other examples range from smart exhibition centers, autonomous robots and smart electric grids to emergency coordination systems. A concrete example of firefighter coordination system is described in detail in Section 2.1.

As seen from the numerous examples, the class of CPS is large and expansive, that is "engineered systems that are built from, and depend upon, the seamless integration of computational algorithms and physical components" [1]. European H2020 research agenda regards CPS to be "the next generation of embedded ICT systems that are interconnected and collaborating, providing citizens and businesses with a wide range of innovative applications and services" [2].

Main features of CPS are high dynamicity, open-endedness, but also dependability and resilience to cope with ever-changing physical environment whose properties are often uncertain. As a significant portion of CPS is life-critical, dependability is a very important requirement. The unpredictability of the environment emphasizes the need of self-adaptability which means systems change their internal state and behavior to react to external impulses. However, typical existing approaches for self-adaptation cannot handle well all situations because they choose tactic from a fixed set which is difficult to design correctly for complex systems so that all mandatory functionality is unaffected under any circumstances.

1.2. Problem Statement

Self-adaptability, i.e. the ability to alter system's behavior or structure in response to external stimuli and changes in the environment, is an important feature of any efficient and dependable CPS. There are three typical ways for achieving self-adaptation in software systems: (i) by designing detailed application mode, e.g., Markov Decision Processes (MDP), and employing simulations or other means of state-space traversal to infer the best response of the system, (ii) by identifying control parameters and employing feedback-based control techniques from control theory, and (iii) by reconfiguring architecture models, typically with the help of Event-Condition-Action rules – architecture-based self-adaptation.

When facing a large complicated distributed systems such as CPS, method (iii), i.e. high level self-adaptation based on architecture models, is preferred in general [3], [4], [5], [6]. Self-adaptation rules in (iii) manifests in invoking certain suitable architecture reconfiguration based on satisfaction of particular conditions [3], [4], [7]. The results of adaptation are usually measured by satisfaction of system goals. The adaptation action enables or disables an activity, generally called tactic [3], in the form of component, component process or binding between components. These methods (i-iii) select an action from a pre-designed fixed set of operations based on observed state of the environment, so self-adaptation in (iii) can be interpreted as choosing subset of tactics from a fixed superset.

The problem lies in inherent unpredictability in the realm of CPS, such as network instability, hardware malfunctions or other physical world hazards, which renders anticipating all potential circumstances in advance at design time infeasible. Therefore, CPS may encounter situations where adaptation by switching between tactics fails as no combination of predefined tactics is applicable in the current context.

1.3. Research Goals

Responding to the challenges presented in Section 1.2, this thesis focuses on providing means to deal with unanticipated runtime situations in CPS which enhance IRM-SA [8], [9] design and runtime by elaborating and implementing the concept of meta-adaptation [10]. These meta-adaptations push the limits of systems' adaptability by creating new tactics at runtime to cope with dynamic changes in the environment and improve overall system utilities. The behavior of meta-adaptations at runtime can be influenced at design time of the CPS but the amount of initial input needed is kept at minimal possible level.

The primary intention is to present the idea of meta-adaptation strategies which is in fact mostly agnostic to adaptation method and implementation framework and to provide implementation of example strategies as a proof of concept to IRM-SA architecture-based adaptation method. The running example of firefighter coordination system in jDEECo [11], [12], [13] serves as a context to experimentation and evaluation of the proposed approach.

This thesis targets the following research goals:

G1 The first goal is to elaborate the proposed meta-adaptation strategies, their potential mutual cooperation and embedding into greater context.

G2 The second goal is to implement examples of meta-adaptation strategies as jDEECo plugins which commence functioning when IRM-SA adaptation method fails to provide suitable adaptation tactic.

G3 The last goal is to prepare the experimental environment in the context of firefighter case study to evaluate the implementation of the meta-adaptation strategies.

1.4. Structure

The thesis is structured in the following way. First, Chapter 2 introduces an example of CPS (Section 2.1), detailed description of the case study based on the example and technological background needed to fully understand the concepts and terms used throughout the thesis. Particularly, IRM (Section 2.4), IRM-SA (Section 2.5) and DEECo & jDEECo (Section 2.6) are presented. Chapter 3 provides an analysis of limitations of combination of IRM-SA and jDEECo and formulates requirements based on thesis goals dealing with these weaknesses. Chapter 4 contains descriptions of meta-adaptation strategies further referenced in the thesis and their relationships focusing on goal **G1**. Chapter 5 presents the architecture of framework supporting meta-adaptation (Section 5.1 and 5.2) as well as information about individual implementations of the meta-adaptation strategies dealing with goal **G2**. The results of the evaluation of the firefighter case study as required by goal **G3** can be found in Chapter 6. The context of the research and the comparison of the meta-adaptation strategies and other related approaches are provided in Chapter 7. Finally, Chapter 8 then concludes the thesis and gives some ideas to improve and extend the outcomes of the thesis.

2. Background and Running Example

The first sections of this chapter contain description of the case study, introduction into Invariant Refinement Method and how it can be exploited to model the case study. Basic concepts of DEECo and particularities of its Java implementation jDEECo can be found in the second half of sections in this chapter.

2.1. Example of a Cyber-Physical System

To better illustrate the context and challenges of Cyber-Physical Systems, the following text describes a simple scenario based on real-life real-scale case study that has been proposed for the evaluation of distributed self-adaptive systems, Firefighter Coordination System [8]. A team of firefighters divided into tactical groups of several firefighters is deployed on the emergency field. Each group is led by a group leader (officer) who aggregates the data of their subordinates' status and environment. The intention is that each leader can deduce whether any of their group members is in danger and take strategic decisions. Example of such mission can be seen in Figure 1.

The communication is done via low-power nodes integrated into their personal protective equipment. Every node is configured at runtime depending on the task assigned to its bearer. For example, a hazardous situation might need closer monitoring of a certain parameter (e.g., temperature).

The group leaders use tablets to display model of the current situation on a map and also detailed information provided by low-power nodes are shown, i.e. information about position, external temperature, battery level and oxygen level. These data are crucial for creating overall picture of the status of the current operation and for giving the appropriate orders or taking corresponding measures to avoid casualties.

Such a coordination system comes with a number of challenges. Its demands on stability, safety and performance are obviously high. Though no guaranties for end-to-end response time are available on top of opportunistic ad-hoc networks assumed to operate beneath the system. Energy consumption should be minimal. Sensors and other component malfunctions cannot be ruled out. What if the temperature starts providing inaccurate readings or fails completely at runtime? What if GPS connection is not available inside the structure firefighters operate in? What if group members lose connection to their group leader?

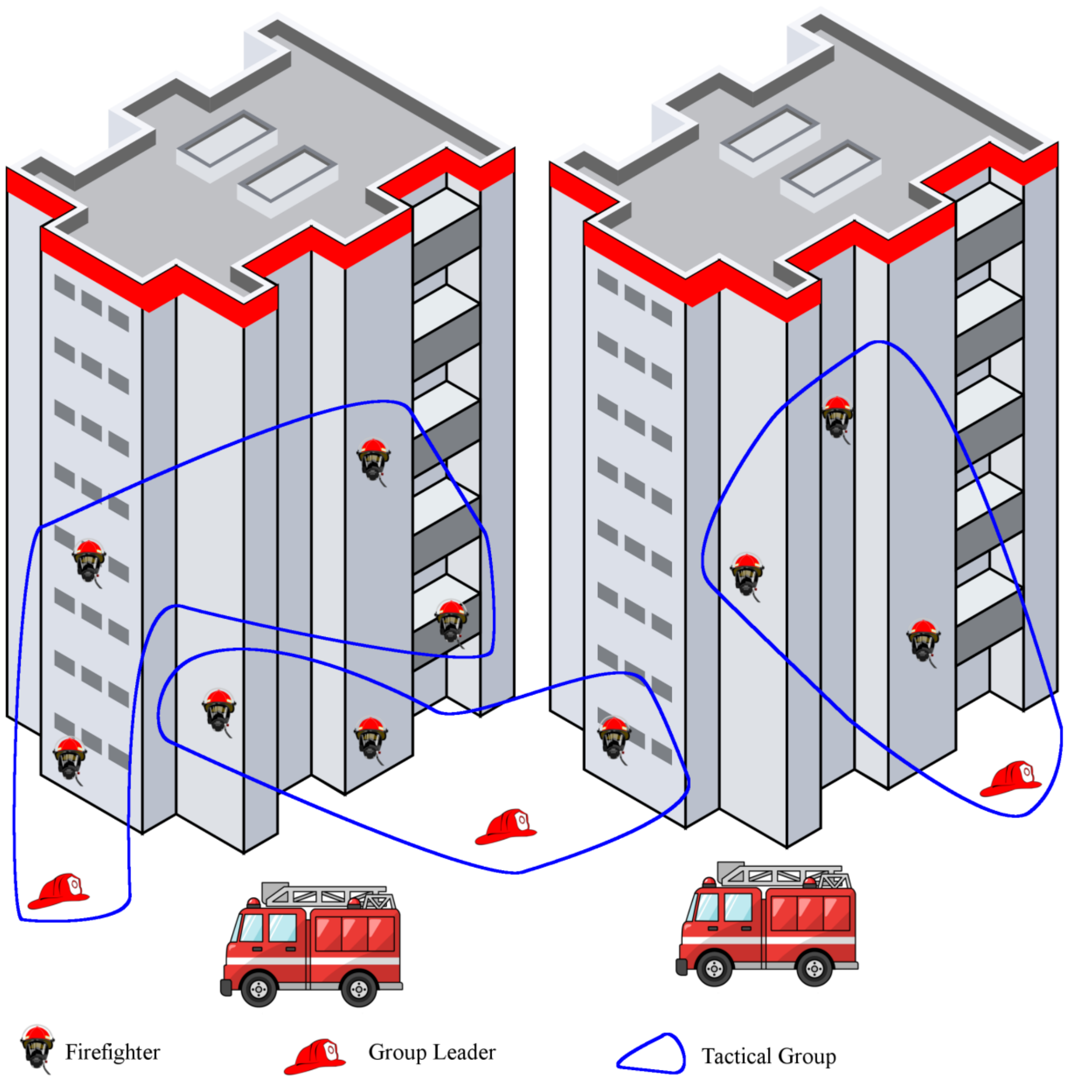


Figure 1: Firefighter coordination case study.

In the circumstances listed above, latest information available is the ground for adaptation of the behavior of every node. For instance, the tactic using indoor tracking system needs to replace the tactic using GPS to detect position if GPS signal becomes too weak. Other tactics ranges from delegating the communication with the group leader to a nearby firefighter if connection to the leader is lost, to changing the frequency of the sensor sampling.

However, it is not possible to list every situation that could trigger adaptation with non-zero probability at design time of the firefighter coordination system. The environment is too unpredictable, complex and dynamic. Far better approach is to build framework where it is possible to dynamically alter its behavior by (i) generating new tactics on demand, and (ii) using these tactics in adaptation actions to deal with unanticipated circumstances.

2.2. Introduction to IRM-SA and DEECo

Invariant Refinement Method for Self-Adaptivity (IRM-SA) [8], [9] is a requirements-oriented design method targeted for domain of CPS. It is based on an iterative approach of refining system requirements from general one to requirements on certain components. This method enables to trace software artifacts to system-level goals and thus contributes to dependability. The different system or component modes emerge from different operational contexts captured by IRM-SA as design alternatives, which greatly boosts adaptability.

The basic concept of IRM-SA is invariant that describe properties of the system-to-be during its whole lifecycle. Invariants express goals and requirements of the system. There are several different sub-kinds of invariants. Process invariants refer only to one component and its fields. Exchange invariants transfer data from fields of one component to other component's field. Assumptions are a special kind of invariant describing conditions expected to hold about the environment, thus an assumption is not expected to be maintained by the system. The invariants constitute a hierarchical system, resembling oriented forest of invariants. The orientation represents refinement of higher level invariants into lower level invariants, either as AND-decomposition or OR-decomposition. The latter can capture different design alternatives with an assumption guarding each variant covering the state of the environment. The IRM-SA design method starts with a set of top-level invariants and ends when every leaf invariants is either Process invariant, Exchange invariant or Assumption.

For example, consider invariant (1) in Figure 2, which declares that the leader of each firefighter group (officer) needs an up-to-date view (encapsulated in the field positionMap) of their group members' location. This "necessity" is AND-decomposed into invariant (2), which happens to be exchange invariant describing the necessities of propagating the position from each member to the leader, and invariant (3) stating necessity of determining the position on the side of each member. Further decomposition of invariant (3) is an example of captured design alternatives. It can be satisfied either by determining the position through an indoors tracking system – invariant (5) – or a Global Positioning System (GPS) – invariant (7). The satisfaction of assumptions (4) and (6) is monitored at runtime and the system switches to the activity bound to the tree's branch currently in effect.

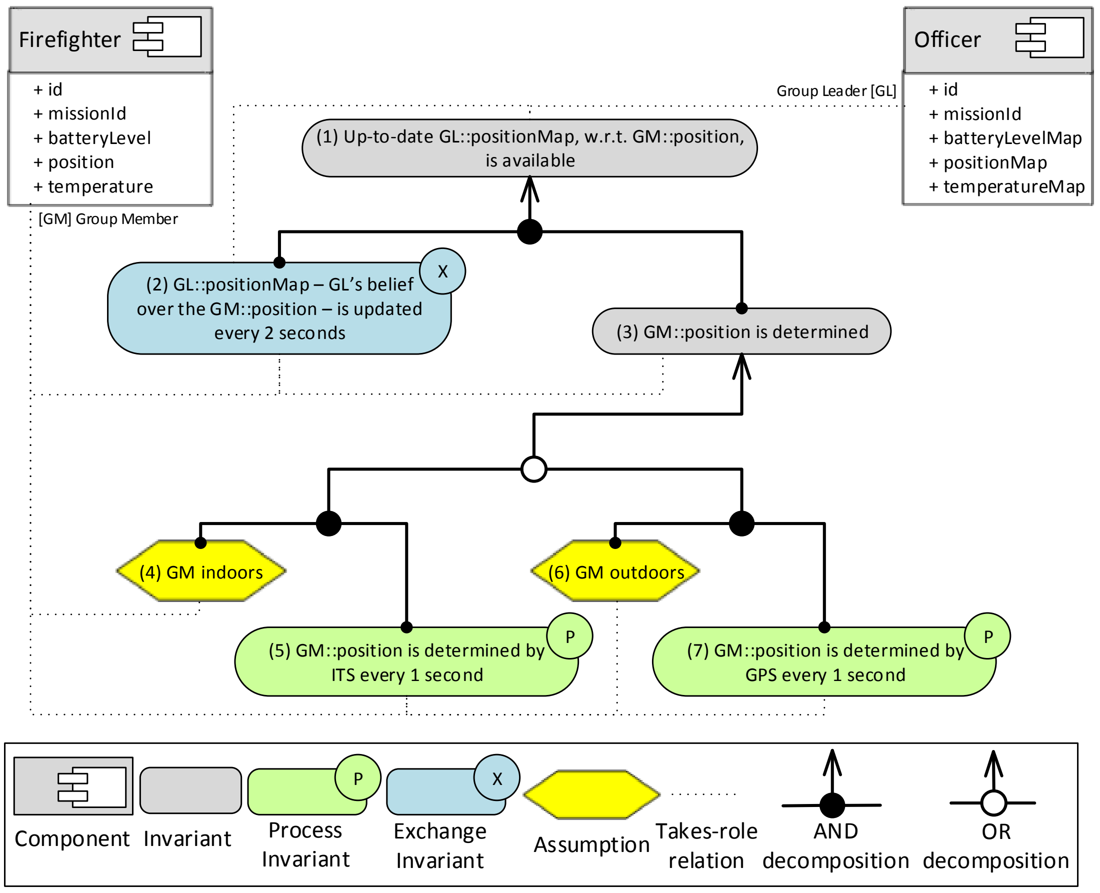


Figure 2: Fragment of IRM-SA model of the case study.

IRM-SA is an independent method without any other dependencies, however its concepts are very well aligned with the abstractions and mechanisms featured in Dependable Emergent Ensemble of Components (DEECo) [11], [12] which is a component system specifically targeted for creating highly dynamic CPS. It features two basic abstractions – components and ensembles. Components are autonomous units of computation and deployment which contain knowledge (their data fields representing the state of the component) and processes operating on knowledge of the individual component. The components are strictly separated and cannot explicitly communicate with each other. The only possible way of communication is indirect via ensembles, which corresponds to how exchange invariants of IRM-SA work, i.e. one component's knowledge is transformed into knowledge of another component. (e.g., Figure 3, lines 25-26). Ensembles are thus groups of associated components exchanging data and cooperating to achieve a specific objective. Components' memberships in ensembles are dynamic, that is periodically updated with regards to component's knowledge accessed through ensemble-specific interfaces.

```
1.  interface GroupMember:
2.      missionId, position
3.  interface GroupLeader:
4.      missionId, positionMap
5.
6.  component Firefighter features GroupMember
7.      knowledge:
8.          id = 59
9.          position = { x = 49.04606, y = 15.093519 }
10.         temperature = 45.2
11.         …
12.     process determinePositionFromGPS
13.             out position
14.         function:
15.             position.x ← GPSSensor.readX()
16.             position.y ← GPSSensor.readY()
17.         scheduling: periodic ( 500ms )
18.     … /* other process definitions */
19.
20. ensemble PositionExchange:
21.     coordinator: GroupLeader
22.     member: GroupMember
23.     membership:
24.         member.missionId == coordinator.missionId
25.     knowledge exchange:
26.         coordinator.positionMap ← ( member.id, member.position )
27.     scheduling: periodic ( 1000ms )
```

Figure 3: Fragment of case study components and ensembles in DEECo DSL.

As hinted above, IRM-SA to DEECo mapping is straightforward. IRM-SA components correspond to DEECo components, process invariants to component processes, exchange invariants to ensembles and assumptions to DEECo runtime monitors. The DEECo process coinciding to process invariant (7) from IRM-SA graph and ensemble matching exchange invariant (2) can be seen in Figure 3, lines 12-17 and 25-26. The adaptation logic used by DEECo systems to turn on and off particular features is exploited to switch to IRM-SA graph tree branch currently in effect at runtime. Technically, a SAT solver is used by a dedicated adaptation manager to reveal a satisfiable configuration, i.e. turns on processes and ensembles corresponding to selected leaf invariants and disabling the other ones.

2.3. Case Study Description

This section contains elaboration and more detailed description of the example of a CPS from Section 2.1. This simple scenario comes from a case study developed in cooperation with professional firefighters.

Let us consider an emergency situation like fire, flood or hurricane. A scouting team of firefighters is deployed in the field by the firefighter department with the objective to survey the criticality of the situation for taking suitable strategic measures. The team consists of tactical groups and every group is organized, commanded and feedbacked by its group leader (officer). The decreasing costs of related technologies enable to equip firefighters with sensing and actuating equipment. This results in improving safety and decision making as firefighting departments and group leaders are provided with large quantity of information collected by firefighters in real time about their position, state of surroundings (temperature, noise, humidity, air composition, etc.), energy level, oxygen supply and health status. These data could prove crucial for making tactical decisions by the group leaders to command the group effectively and to successfully complete the mission.

Members of the firefighter groups have personal protective equipment with integrated low-power nodes capable of wireless communication and thus of capturing and sharing information about the environment as discussed above. Different configurations of the nodes for various types of missions are available, so efficient usage of limited resources is reached depending on the task assigned to the bearer. For example, oxygen level might not be monitored during a flood emergency (resulting in lowering the power consumption). On the other hand, while fighting the fire the monitoring of temperature is of a great importance and this environment parameter should be monitored very closely in such context. The nodes can also exploit other stationary heterogeneous devices in the operation site, e.g. access points, temperature sensors in the buildings, etc. either to gather additional information or boost the wireless coverage and network performance when the additional devices are used as relays.

Group leaders coordinate and command their subordinates through tablets where data collected from individual firefighters are aggregated. The model of the current situation is shown in a visual way (i.e. on a map) on their displays so decisions can be made quickly and efficiently, this greatly helps in avoiding or at least lowering casualties both in lives and heath as well as material damage. When a firefighter is discovered in a potentially dangerous situation, the group leader is notified and can take measures immediately to mitigate the risks before the situation becomes critical.

Designing such safety-critical system is a very challenging task as fulfilling the requirements on stability, performance and dependability is top priority. The environment is highly dynamic and any hardware failure may occur at any time, so sensor readings may become unavailable or completely wrong. Network may prove unreliable in extreme conditions expected in operational field and packets may be delayed or even lost. The length of mission can vary a lot and often cannot be predicted in advance, energy consumption should thus be kept as low as possible to keep the firefighters in the field as long as the circumstances dictate.

The first of two main objectives of the system is to guarantee that individual nodes can operate in any situation, even when the network fails and they are completely isolated. The second objective is to ensure that the nodes can optimally satisfy system-level objectives and constraints even without supervision.

During the analysis of the requirements we may try to capture all possible situations which our system-to-be can get into. What if connection to other group members is lost? What if the oxygen level sensor malfunctions? What if the indoor position system cannot be used due to interference? What if data from others are obsolete because of intermittent network connection? Even this short list of what-ifs serves to indicate that the environment is too dynamic and complex to predict all problems at runtime and their combination ahead at design time.

The ideal solution would be to adjust the system behavior dynamically at runtime to cope with unanticipated situations without the need to provide exhaustive list of pre-designed solutions. This is the subject of our approach. In particular, we build a framework generating new adaptation tactics at runtime to use them in adaptation actions in order to deal with circumstances or combination of failures that have not been expected at runtime either due to their low probability or due to mistakes in design process.

2.4. Invariant Refinement Method

This section expands the description of Invariant Refinement Method (IRM) [8], [9] from Section 2.2 in more detail. IRM is a design method specifically tailored for CPS. It is a requirements-oriented design method focused on distributed cooperation and global perspective on the system-to-be. Both low level software requirements and high level system goals are modeled by the invariant concept. IRM is based on the iterative decomposition of higher level invariants into more specific sub-invariants until all leaf invariants can be implemented by autonomic components or data transfer from one component to the other (i.e. in DEECo performed through the participation of components in an ensemble). This method guides the transformation of initial high level requirements into software architecture of ensembles and autonomous components. IRM provides both traceability of system-level goals to software artifacts and vice versa and captures the design alternatives corresponding to various situations and system deployments mapped to system configurations and component modes.

Components are functional entities of the system-to-be. In IRM, components consist of data fields specific to the domain of the system called knowledge, for example the oxygen level in a firefighter personal equipment or a list of places that a driver wants to visit at particular times. Knowledge is not immutable, but only changes as a result of so-called process invariants (the component itself changes its knowledge) or exchange invariants (the framework transfers knowledge of one component to the other). Components may adopt a particular role in the system if they are referenced by an invariant.

Invariants are the basic concept that IRM is built on. They represent system requirements and goals by describing the desired state of the system-to-be at every moment. Invariants are organized into trees reflecting the decomposition of the top-level system goals. Example of such tree hierarchy from the case study can be seen in Figure 2. Rounded rectangles represent invariants, for instance (1) is a top-level invariant expressing the requirement that group leaders must have information about their subordinates' positions.

There are three sub-kinds of invariants – process invariants, exchange invariants and assumptions. IRM guided design is done if and only if all leaf invariants in IRM trees are one of these sub-kinds of invariants. Assumptions are conditions about the environment expected to hold during runtime and are not maintained explicitly by the system. They are depicted as yellow hexagons in diagrams, for example (4) in Figure 2. Process and exchange invariants are associated with computation activity, i.e. computation producing output knowledge fields given input knowledge fields so that the invariant referencing those fields is satisfied. The computation activity is a second view on the invariant as it provides means for satisfying the operational normalcy described by the invariant.

Every knowledge field of components is an output of a single process or exchange invariant. Process invariants take an input set of knowledge fields of a single component and transform them into an output set of knowledge fields of the same component (of course both sets can be empty for special-purpose invariants). On the other hand, exchange invariants take an input set of knowledge fields of a component and transfer them into an output set of knowledge fields of other component. Invariant (5) in Figure 2 is an example of process invariant which are marked with P in diagrams. Similarly, the letter X marks the exchange invariants, e.g. invariant (2) in Figure 2.

The key mechanism of IRM is decomposition of higher level invariants in a systematic and step-by-step manner. This decomposition results in a set of lower level sub-invariants whose conjunction or disjunction implies the higher level invariant which is depicted in IRM tree via AND- and OR-nodes and their connections. The same behavior expected from the parent invariant is found in children invariants and potentially even more.

This refinement process is recursively applied to system level goals and ends when all leaf invariant are either an assumption, process invariant (invariant referencing only one component) or exchange invariant (invariant referencing ensemble of components).

Figure 2 also demonstrates the refinement. The top-level invariant (1) is refined into a conjunction of two sub-invariants: (2) transferring the information about group member position to group leader and (3) determining the position so it can be transferred. Invariant (3) is further refined by a combination of OR-decomposition and AND-decomposition which is formally not allowed by the IRM. The formal way to handle such situations is the introduction of synthetic invariants corresponding to the abstract-syntax tree of the target formula. However, the graphical notation of diagrams omits these synthetic invariants and decomposition symbols are connected directly, because there is no additional knowledge in the synthetic invariants.

2.5. IRM-SA

Invariant Refinement Method for Self Adaptation (IRM-SA) [8], [9] is an extension of IRM. The design model and process of IRM-SA capture design alternatives (alternative realizations of requirements on the system), applicable configurations and their corresponding circumstances, this enables the running system to adapt to different situations by exploiting architecture variability.

There is at least one applicable configuration for every situation. The number of design alternatives that must be explored to map configurations to situations is usually large. If the design alternatives depend on each other or reference various abstraction levels, the issue is even more problematic. To achieve scalability at design time, decomposition for separation of concerns is introduced. To scale at runtime, SAT solving for selecting the application configuration is employed.

The architecture self adaptation itself runs in iterations consisting of three steps. Firstly, the current situation needs to be identified. Secondly, the configuration suited for the situation must be selected. And finally, the architecture is reconfigured to match the selected configuration.

IRM-SA extends the IRM design model by OR-decomposition, which makes it possible to capture design alternatives. A characterizing assumption in IRM-SA is a top-level assumption specifying the particular situation addressed by the design alternative. Examples of characterizing assumptions can be seen in Figure 2, assumptions (4) “GM indoors” and (6) “GM outdoors” are characterizing assumptions for their respective sub-trees which captures two design alternatives corresponding to the situations where the firefighter is located inside a building or under the open sky.

Characterizing assumptions of several design alternatives may hold at the same time, so the design alternatives are not exclusive. This also serves as built-in fault tolerance mechanism [14].

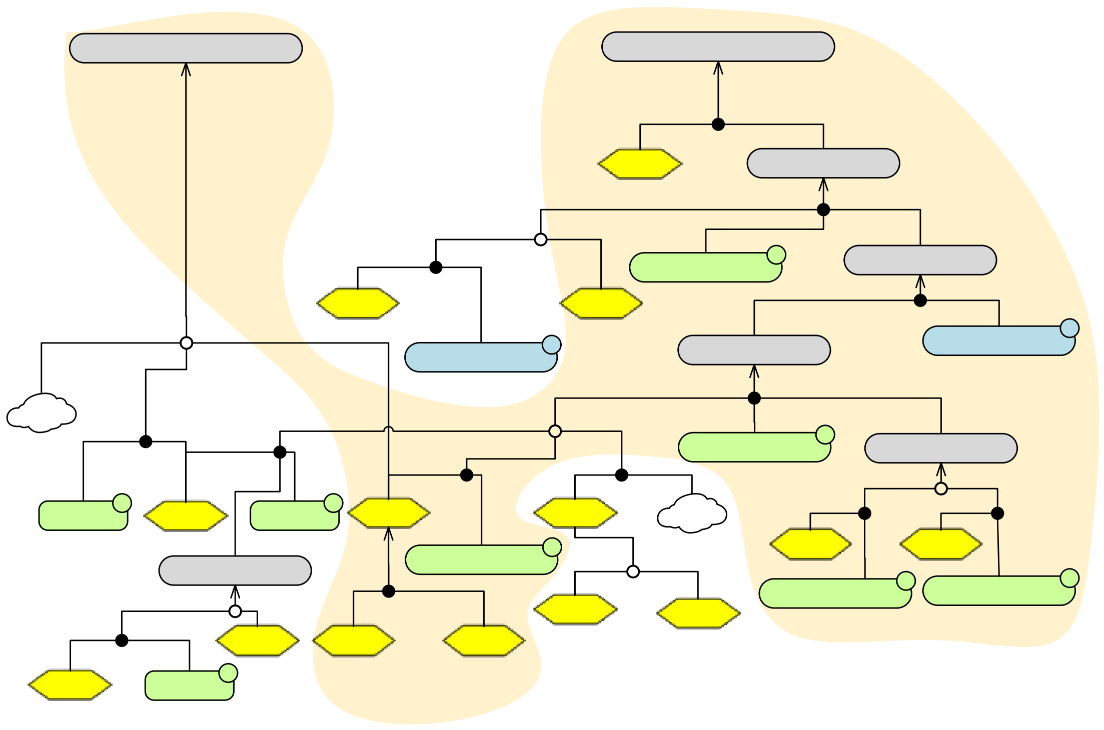


Figure 4: An architecture configuration selected by IRM.

Tracing the low-level processes to high-level invariants is the main way to address dependability. There is no support for other dependability features such as privacy or security. Self-adaptation is not based only on evaluation of snapshots of internal data of other components (i.e. component *belief*), but also on evaluation of various metadata associated with the belief, e.g. timestamp of creating of the belief, timestamp of network activity associated with the belief (timestamp of receiving or sending the data) and so on. This information may help the self-adaptation to predict dangerous situations and further increases dependability.

The CPS sensing and distribution of data causes that belief of the individual components is necessarily outdated. The initial sensor and network latency when disseminating the data as well as other influences are summarized in *inaccuracy* of the belief - this is another example of belief metadata that can be considered while self-adapting. The inaccuracy works well for continuous domains. For discrete ones there is concept of possibility which is a model based on timed automata (see Figure 5). Both inaccuracy and possibility enable invariants to express the need for special adaptation actions when the inaccuracy of the component belief raises too high. This fail-safe mechanism also contributes to overall CPS dependability.

The IRM-SA model of the system can be encoded into a Boolean satisfiability problem (SAT) to easily select an applicable configuration to the current situation. In short, the task to select an applicable configuration is the problem of constructing a set C of selected invariants from the IRM-SA model corresponding to an applicable configuration (example in Figure 2). The following statements must hold to ensure C is well formed with regards to invariant decomposition: firstly every top level invariant is in C. Secondly every child invariant created by AND-decomposition is in C if and only if its parent invariant is in C. Thirdly if a parent invariant is refined by OR-decomposition, then it is in C if and only if at least one of its children invariants is in C, too.

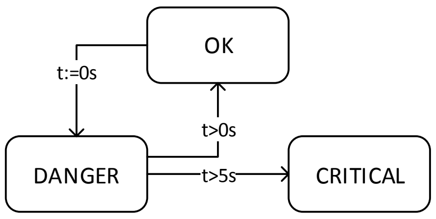


Figure 5: Timed automaton capturing the transitions in the possible valuation of the nearbyGMsStatus field.

Due to the fact that the IRM tree is a directed acyclic graph, invariants on shared paths can be safely duplicated to transform the IRM-SA model into a forest. A Boolean variable s_i is created for every invariant i indicating whether the invariant is in set C or not. Rules described above are transformed to formulas as input of SAT. The variables corresponding to top-level invariants are bound to true. Another Boolean variable a_i is created for every invariant i indicating whether the invariant is acceptable, i.e. whether it can be included in C with regards of the current situation. Formulas "s_i implies a_i" is added to the SAT instance to capture the relationships between these variables. The state of system and its environment is represented in binding variables a_i to reflect acceptability of their corresponding invariant. Every satisfying valuation of such a SAT instance corresponds to an applicable system configuration. If no satisfying valuation exists, then there is no applicable configuration for the current situation.

However, there can be more than one applicable configuration. In such cases there can be mechanism that takes i) IRM-SA model and ii) outputs of the SAT solver (applicable configurations) and chooses one configuration to become the new system configuration. This mechanism can consider many features of individual applicable configurations and compare them based on different criteria. Many different strategies could be employed in this place, e.g. simple total preorder of alternatives in each decomposition, etc.

Every node solves the SAT problem independently exploiting the determinism of SAT solving so that all nodes using the same prioritization method reach the same applicable configuration. However, because of communication delay and unreliability components' knowledge, and therefore also the output of the SAT solvers, gets temporarily desynchronized. This is not harmful in most cases, only the overall system performance is reduced. As an aside, unbounded message delays render both fully centralized and distributed (those requiring distributed consensus) SAT solving methods inapplicable [14].

2.6. DEECo and jDEECo

This section expands the description of the Dependable Emergent Ensemble of Components (DEECo) [11], [12] from Section 2.2 and introduces jDEECo [13] which is its Java implementation.

DEECo is a component model and instantiation of a class of component systems called Ensemble-Based Component Systems (ECBS) which exploits the key ideas of component-based software engineering [15], [16], agent-oriented computing [17], [18] and ensemble-oriented systems [19], [20]. ECBS addresses the dynamic and autonomic nature of Cyber Physical Systems, for which it is specifically tailored and features autonomic components with periodic execution and dynamic ensembles of components controlling data exchange between components.

These dynamic ensembles replace the usual explicit architecture of components and can be characterized as dynamic groups of components cooperating to accomplish joint objectives. The components are autonomous entities inspired by concepts of agent-oriented computing to deal with dynamism. The definition EBCS can be as follows [11]: "Distributed systems composed of components that feature autonomic and (self-) adaptive behaviors and are organized into emergent ensembles to achieve cooperation."

The most important characteristics of ECBS are (i) emergent system architecture represented by bindings of components arising at runtime but based on component and ensemble definitions created at design time, (ii) belief about the environment and the system is managed by the runtime framework for each component, (iii) components are encapsulated and their processes can only employ the knowledge (which includes the belief over the knowledge of other components) of the component with no explicit communication with other components.

DEECo refines the approach of ECBS into software engineering concepts suitable for building actual CPS. There are two main constructs in DEECo: components and ensembles. A component is an independent and autonomous unit of computation, deployment and development. An ensemble is the only way components can interact with each other and serves as a mediator between a set of components which binds them together and arranges their communication. The runtime framework managing both components and ensembles constitutes an essential part of DEECo component model.

A component consists of a set of processes and a set of knowledge fields accessible via a set of interfaces (example of DEECo DSL fragment from the case study can be seen in Figure 3).

Component’s knowledge represents component’s belief of its environment and the rest of the system which means it may become invalid or obsolete and must be handled as such. In essence, knowledge is mapping of identifiers to values, potentially structured further (for these cases structured identifiers are used as shown on lines 15 and 16 in Figure 3).

The knowledge can be accessed by the runtime via a set of interfaces (example definitions on lines 1-4, usage on lines 6 and 21-22 in Figure 3) which offers a limited view on component’s knowledge. Knowledge fields exposed by interfaces do not have to be disjunctive so one field may be exposed by multiple interfaces. Polymorphism may be achieved if one interface is provided by different components. Every knowledge field is an output of exactly one process or ensemble.

A component process is characterized as a function (lines 12-17 in Figure 3) with a list of input and output knowledge fields (line 13) and thus manipulates the knowledge of its component. Processes are periodically scheduled, i.e. the framework executes them repeatedly after a specified period (line 17). The framework also fully manages the processes, i.e. gathers atomically all input knowledge fields, computes the process function and writes all output knowledge fields, again atomically. Executing the process may have side effects (e.g. sensing or actuating), but the explicit communication with other processes or even other components is strictly forbidden.

Ensembles determine component composition and interaction by defining the bindings between them. The ensemble is the only way components can communicate with each other, that is transfer knowledge from one to the other (lines 25-26). Example of an ensemble definition in DEECo DSL from the case study can be seen in Figure 3. In an ensemble, there are two roles that components can play – one of them is the *coordinator* of the ensemble (line 21), and the other components are the ensemble's *members* (line 22). The coordinator is determined by providing the interface specified in definition of the ensemble. The members are components providing required interface and satisfying the ensemble's membership condition which is a predicate about knowledge fields accessible via coordinator and member interfaces (lines 23-24). The ensembles may overlap and one component may be coordinator in one ensemble and member of different ensembles at the same time. Moreover, both ensembles may share the same definition, provided the component is accessible by both member and coordinator interfaces. A new ensemble is dynamically created for every group of components satisfying the membership condition and providing corresponding interfaces by the framework which automatically evaluates the membership condition at suitable times or every specified period (line 27).

The main objective of ensembles is knowledge exchange, i.e. transferring information among components. It is a one-to-many interaction between the coordinator and members of the ensemble. The knowledge exchange is performed as defined in ensemble definition once within a specified period.

The framework jDEECo [13] is an implementation of DEECo in Java programming language for practical usage in real development of CPS. jDEECo provides the runtime environment and programming means to design, develop, deploy and run applications exploiting the concepts of DEECo component model.

The mapping of DEECo features to Java language is based on use of annotations which has an advantage in not introducing any external preprocessors or extensions of the language.

```
1.  @Component
2.  public class FireFighter {
3.      public String id;
4.      public Double temperature;
5.      …
6.      @Process
7.      @PeriodicScheduling(period=1250)
8.      public static void determineTemperature(
9.              @Out(“temperature”) ParamHolder<Double> temperature) {
10.             …
11.     }
12.     …
13. }
```

Figure 6: Simplified fragment of jDEECo component from the case study.

Figure 6 contains a simplified example of a component definition from the case study. The components are defined by creating a class annotated by “@Component” annotation (line 1). The knowledge fields are represented by non-static public fields (lines 3-4). The String field “id” is mandatory so the framework can uniquely identify individual components. Knowledge fields start first level of knowledge hierarchy. They can be primitive types (respectively their object wrapper classes), Lists, Maps or structured classes implementing Serializable interface whose fields recursively represent the knowledge hierarchy. The initial values of knowledge fields are either provided by the class constructors or by static initializers. Note that DEECo interfaces are not mapped to Java interfaces. Instead, similarly to duck typing in dynamic programming languages, the provided interfaces are determined by name convention, i.e. they are implicitly detected based on class field names matching the ones exposed by the interfaces.

Defining the component processes has the form of declaring public static methods annotated by “@Process” annotation in the class representing corresponding component (lines 6-11). The “static” modifier is enforced because of the semantics of component processes, primarily their isolation from the component knowledge fields except their input and output knowledge. Knowledge fields are not static and thus inaccessible from the process. However the manipulation with input and output knowledge fields is allowed by passing them as method parameters which is managed by the framework. The parameters need to be annotated by “@In”, “@Out” and “@InOut” annotations to mark which knowledge fields are input, output or both (line 9). The identifier of the knowledge field is part of all these annotations too. The dot-delimited identifier path can be used to access the internal knowledge node inside the knowledge tree. If a primitive type should be passed as out or inout knowledge field for a process it needs to wrapped inside an ParamHolder object because of immutability of such objects in Java (line 9). Periodicity of the process is expressed in “@PeriodicScheduling” annotation with period in milliseconds as parameter on the method representing the process (line 7).

```
1.  @Ensemble
2.  @PeriodicScheduling(period=1000)
3.  public class PositionExchange {
4.      @Membership
5.      public static boolean membership(
6.              @In(“member.missionId”) String memberMissionId,
7.              @In(“coord.missionId”) String coordMissionId) {
8.          return memberMissionId == coordMissionId;
9.      }
10.     @KnowledgeExchange
11.     public static void exchange(
12.             @In(“member.id”) String memberId,
13.             @In(“member.position”) Position memberPosition,
14.             @InOut(“coord.positionMap”) Map<String,Position> positionMap) {
15.         positionMap.put(memberId, memberPosition);
16.     }
17. }
```

Figure 7: Simplified fragment of jDEECo ensemble from the case study.

Figure 7 contains a simplified example of a component definition from the case study. Similarly to component definitions, ensembles are defined as appropriately annotated classes. The main annotation is “@Ensemble” on the class itself (line 1).

Inside such annotated class there can be defined membership condition and knowledge exchange in a form of public static methods. The annotation “@Membership” marks the membership predicate (line 4), the annotation “@KnowledgeExchange” is intended for methods implementing knowledge exchanges (line 10). Parameters of these methods are also annotated by “@In”, “@Out” and “@InOut” annotation in the same way as parameters of component processes with one exception - identifiers for parameters corresponding to knowledge fields belonging to coordinator component are prefixed with “coord”; prefix for members’ knowledge is “member” (lines 6-7 and 12-14). Analogously to interfaces provided by components interfaces required to become member or coordinator do not need to be defined explicitly. Instead, they are implicitly determined as union of knowledge fields passed to methods implementing membership condition and knowledge exchange. Another similarity to component process is the use of annotation “@PeriodicScheduling” to specify period of the knowledge exchange, however it annotates the ensemble class itself because there is only one knowledge exchange method per ensemble (line 2).

3. Analysis / Goals Revisited

Although self-adaptation is an important characteristic of CPS, the DEECo component model and its implementation jDEECo offer no special means for adaptation. The framework is powerful but all responses to external stimuli have to be devised in advance at design time. The same applies to IRM itself. The IRM-SA provides ways to easily switch between different configurations of the system. Its approach is grounded in employing a certain set of component processes whose selection is based on satisfaction of given pre-defined conditions.

However the success of the system adaptations still relies on careful planning during design time and capturing all possible problematic circumstances in advance and preparing suitable responses. In complicated and complex systems such CPS, it is often impossible to anticipate every point of failure in dynamic and perhaps hostile environment which is inherently unpredictable for the realm of CPS. If hardware malfunction, network breakdown or similar issues occur, the system may find itself with no applicable configuration and is destined to fail its goals and objectives in this context.

The concept of meta-adaptation naturally extends the IRM-SA design and provides dynamic changes to system behavior at runtime to improve the adaptability of the system facing unexpected real-world difficulties. It addresses the limitations of IRM-SA and improves utility of the system in unfriendly conditions by generating new adaptation tactics at runtime, picking ones worth trying and then evaluating their effect in the runtime system.

In contrast to limited number of runtime situations that can be defined at design time, the number of tactics generated at runtime is in principle infinite. The meta-adaptation is independent from underlying adaptation method or implementation framework and can be configured at design time, but the idea is to invest as little effort at design time as possible and still get measurable results.

This thesis focuses on one specific category of meta-adaptations which adapts and changes the self-adaptation logic of systems employing architecture-based self-adaptation (i.e. self-adaptation based on switching between architecture variants/modes) to expand the adaptation envelope of the system. The adaptation envelope can be understood as maximal deviations from the optimal situation that the system can still handle and heal itself by its self-adaptation mechanisms (e.g. IRM-SA). The meta-adaptation helps the current self-adaptation to fulfill its purpose in a domain independent way. The self-adaptation itself is adapted and thus the “meta” prefix in the mechanism’s name.

To cope with this task a framework capable of deriving new strategies at runtime must be developed. This framework needs to meet three basic requirements. Firstly a mechanism monitoring how the system’s inherent self-adaptation is successful needs to be designed. For the scope of this thesis, it does not have to be very complex and complicated, a basic one is sufficient as a demonstration and ground for subsequent research. Secondly, a unified way of describing the meta-adaptation strategies would assist in researching and categorizing the meta-adaptation strategies. The system designers would also benefit from this documentation, because it would make the decision on whether to implement and deploy certain strategies in the system easier by providing all necessary information to make the choice in a readable and formalized way. Finally, actual implementations of the meta-adaptation strategies call for a framework that would enable to plug them in easily and in a straightforward way without reinventing the wheel every time.

As a proof of concept, several meta-adaptation strategies and means for their cooperation and management are introduced in Chapter 4. Their implementations in jDEECo are discussed in Chapter 5 to demonstrate their viability. Chapter 6 provides detailed evaluation of their benefits in a test scenario where a simplified case study model was subjected to malfunctions unforeseen at design time.

4. Meta-Adaptation Strategies

As stated in the chapters above, even self-adaptive systems can be only designed to cope with a finite number of situations that they may encounter at runtime. On the other hand, the number of unique tactics that can be generated – in reaction to reaching the limits of the system design when facing the unforeseen circumstances – is generally infinite. Equally important to generating tactics is to be able to rank them and compare them based on their impact on the system. This enables to select the most auspicious tactic suitable for the current situation or at least filter the unpromising tactics before trying them and wasting time with them. Our current implementation of this adaptation mechanism (see Section 5.2) activates tactics and examines its impact on the system. Should the effect of the changes be positive, they are kept, otherwise they are rollbacked. Figure 8 illustrates basic components of this mechanism in the implementation.

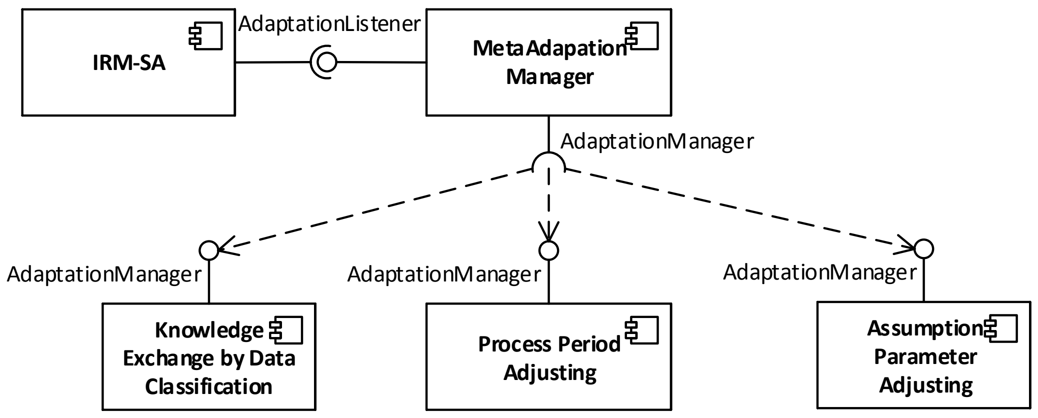


Figure 8: When IRM-SA finishes adaptation, MetaAdaptation Manager is notified and takes over the adaptation if no applicable configuration has been found. In such cases individual strategies are invoked.

The concept of meta-adaptation strategies (MAS) systematizes the creation of new tactics. Every strategy pushes the limits of system's adaptability in a specific manner. There are two objectives of a MAS – to provide (i) an algorithm systematically generating adaptation tactics at runtime, (ii) a metric ranking the generated set of tactics.

The rest of this chapter contains examples of the meta-adaptation strategies whose implementations (see Chapter 5) are evaluated in Chapter 6. The template used to describe them is inspired by the style used by Ramirez et al. to document adaptation design patterns [21] (further based on the template by template of Gamma et al. [22]). Figure 9 contains description of the template format.

Strategy Name: a unique identifier describing the strategy in a compact form

Intent: the strategy's rationale and objectives that it aims to complete

Context: the application independent circumstances required for successful application of the strategy

Behavior: description of the algorithm that generates new tactics accompanied by UML diagrams if needed and the metrics comparing the generated tactics

Contraindications: enumeration of disadvantages and compromises of the strategy

Example: at least one situation that gives an idea of difficulties addressed by application of this strategy and how the difficulties are mitigated

Figure 9: Meta-adaptation strategy template.

4.1. Tactics Generated by Data Classification

The interdependencies among data coming from various real-world physical sensors can be revealed and studied. Particular meta-adaptation strategies could take advantage of them, for example in situations when direct sensing is no longer available. A specific case is dependency based on close physical location of the sensors providing certain measurable attributes, i.e. data with location-dependency. A strategy for automatic creation of definitions of knowledge exchanges is described below. These new knowledge exchanges represent new tactics and can be perceived as a form of "collaborative sensing" when malfunctioning sensor is compensated by a sensor belonging to a node nearby. The description of the implementation of this strategy in jDEECo can be found in Section CorrelationPlugin5.3.

Strategy Name: Knowledge Exchange by Data Classification.

Intent: Improving of system's robustness and extending the period of satisfactory running (or at least result in graceful degradation) under circumstances including unavailability and obsolescence of data.

Context: This strategy deals with scenarios when component's data in a knowledge field become so old that the component's behavior cannot be reliably considered correct, e.g. field's value cannot be updated because of malfunction of the sensor providing the data.

Behavior: A new ensemble specification (see DEECo architecture, Section 2.6) is created to obtain the approximation of actual value of the currently unupdatable field by substituting the up-to-date knowledge of other components. The specification is comprised of (i) membership condition specifying the condition which components must satisfy to interact with each other; and (ii) a knowledge exchange function specifying manipulation with data transferred from a component to the malfunctioning component. For simplicity's sake, only identity function is now used as the knowledge exchange function, i.e. the data are just copied without further altering.

The building of the membership condition is not trivial and requires long observation of the system during normal operation and logging of evolution of components' knowledge with timestamps. This data gathering is usually done in advance as the data analysis and correlation have a performance overhead that cannot be tolerated for embedded mobile devices. The objective of this offline analysis is to find conditional correlations indicating that "closeness" of pairs of values of knowledge fields (A_1, B_1), … , (A_n, B_n) belonging to two different components implies that values of other knowledge fields C, D of the same two components are "close" too. The ensemble specification is created for each such correlation so that the membership condition is the closeness of pairs (A_i, B_i) and the knowledge exchange function is the assignment D := C providing D is the un-updatable knowledge field. These new ensemble specifications will be instantiated when the situation targeted by the strategy is encountered.

Finding the correlations is technically realized by searching for relations

$$\bigwedge_{i=1\ldots n} \mu_{A_i B_i}\left(v_{A_i}, v_{B_i}\right) < \Delta_{A_i,B_i} \rightarrow \mu_{C,D}(v_C, v_D) < T_{C,D}$$

The knowledge logged from runs during normal operation enables to establish these relations on a given confidence level α. The function $\mu_{X,Y}$ is domain-specific and user-provided metrics which defines "distance" between values of knowledge fields X and Y. Typical metric used is a Euclidean distance. The constant $T_{C,D}$ is again domain-specific and user-provided tolerable distance for fields C and D. The value of knowledge field X is represented by v_X. The output parameter is denoted as $\Delta_{A,B}$ and represent maximal distance of values of knowledge fields A and B so that they still imply correspondence of values of C and D.

Possible membership conditions corresponding to different tactics are generated and then a tactic providing the most general condition given the target confidence level is selected (i.e. the membership that is a superset of all other memberships).

Contraindications: Dedicated hardware infrastructure may be needed because the gathered data analysis is a performance heavy task. Also the knowledge logging itself could potentially have high requirements on resources. The demands on performance are particularly serious when there are many knowledge fields or/and when their values change frequently. Moreover, the system may find itself overloaded with needlessly replicated data created by potentially excessive ensembles.

Example: As stated in the description of the firefighter case study Chapter 2.3, each firefighter is equipped with temperature and position sensors whose readings are stored as knowledge. The cooling system of the personal equipment may employ the values of the temperature knowledge, for example. The sensor malfunction clearly endangers the firefighter's security. Fortunately, the firefighters typically cooperate in groups moving close together and thus their sensors measure similar values. So the readings from temperature sensors belonging to nearby colleagues could replace the outdated value from the broken temperature sensor. For instance, the algorithm computes that the temperatures could be exchanged with confidence level 0.8 and tolerated distance 10 degrees (those are user-defined parameters) if the two firefighters are not separated by more than 4 meters.

4.2. Tactics Generated by Period Adjusting

Real-time criteria are usually part of specifications of CPS and are addressed by parameters controlling the scheduling of component processes. Schedulability analysis can most of the times deduce values for these scheduling parameters when the parameters do not have complicated impact on the system. This systematic analysis may not be viable otherwise, i.e. in cases when the parameters affect the system in a complex way, for example when the balance between battery consumption, network utilization and CPU performance is sought. Instead, the systems architects assign values to the scheduling parameters manually. However, if unforeseen circumstances occur, these manually set parameters may no longer meet the criteria imposed on the system, which is exactly the situation addressed by the meta-adaptation strategy introduced in this section. The description of the implementation of this strategy in jDEECo can be found in Section 5.4.1.

Strategy Name: Process Period Adjusting.

Intent: Process scheduling optimization considering overall domain-specific system performance if there are periodically scheduled processes in the system.

Context: This strategy deals with scenarios when violation of timing criteria causes the failure of the system and deduction of scheduling parameters in advance is not viable due to their complicated impact on the system

Behavior: Each process r_i from the set R of real-time processes in the system with scheduling period p_i is provided a corresponding runtime monitor that returns a fitness value f_i which is a real number in range [0-1] that indicates whether the process is still satisfied. If f_i is lower than minimal acceptable value, new tactics that correspond to new real-time processes r_i’ are generated. The processes r_i are transformed to r_i’ by adjusting their periods p_i to p_i’ while keeping them in the pre-defined allowed range. The generic algorithm (1+1)-ONLINE EA [23] is exploited to find the suitable adaptation as can be seen in Figure 10 and Figure 11. Adjusting period p_i of r_i (line 12 in Figure 10) can be understood as substitution of tactic r_i by newly generated tactic r_i’. When there are no processes suitable for adjusting left, i.e. period of every processs has been already adjusted in both directions, or a pre-defined maximal number of tries has been reached, the process period adjusting is finished.

The benefit of individual tactics (i.e. new processes) is measured by comparing the overall system fitness (calculated as a function of individual f_i’s) after an observation period passes and the changes take effect on the running system.

Contraindications: If the periods of component processes are lowered too much, other system resources like battery, network or CPU may be influenced negatively. In such systems, consumption of these resources must be modeled, so the strategy can choose tactics with regards to these requirements.

Example: As stated in the description of the firefighter case study Section 2.3, the operation time of firefighters in field cannot be easily predicted, so battery consumption should be kept minimal. However, the team leader needs as accurate position information for effective firefighter management. These two requirements can be modeled as two contradictory invariants. If the GPS process is invoked too scarcely, the cumulative inaccuracy of the estimated position of a moving firefighter may be too high to fit into a predefined range. On the other hand, too frequent invoking of GPS may result in high battery drainage. As cumulative inaccuracy is the sum of initial inaccuracy of the GPS sensor and the distance a firefighter could have moved since the last GPS readings are obtained, the predesigned process period can become unsuitable if the initial inaccuracy of the sensor rise unexpectedly, e.g. due to the fact that less than three satellites are in sight.

The adaptation can mitigate the risen inaccuracy by scheduling the process determining the firefighter position more frequently. However, the process period must stay in pre-defined bounds and also cannot jeopardize battery life, so balance between these factors must be found by the process period adjusting.

```
begin
  foreach Invariant from Processes.Invariants do
  begin
      Compute fitness for Invariant
      OldFit = *CombineFitness(Processes.Invariants.Fitnesses)
      Adaptees = *SelectProcessesToAdaptTheirPeriods(Processes)
  end
  foreach Process from Adaptees do
  begin
      *Select the direction for period adjustment (up or down) for Process
      *Calculate period delta (difference of old and new period) for Process
      Change period of Process
  end
  ObserveTime = CalculateObserveTime(Processes)
  Run for ObserveTime with no further adaptations for changes to take effect
  foreach Invariant from Processes.Invariants do
      Compute fitness for Invariant
  NewFit = *CombineFitness(Processes.Invariants.Fitnesses)
  if NewFit > OldFit then
      KeepChanges
  else
      Roll-back changes
end
```

Figure 10: Single run of the Process Period Adjusting strategy in a form of pseudocode. “*” denotes variation points of the algorithm.

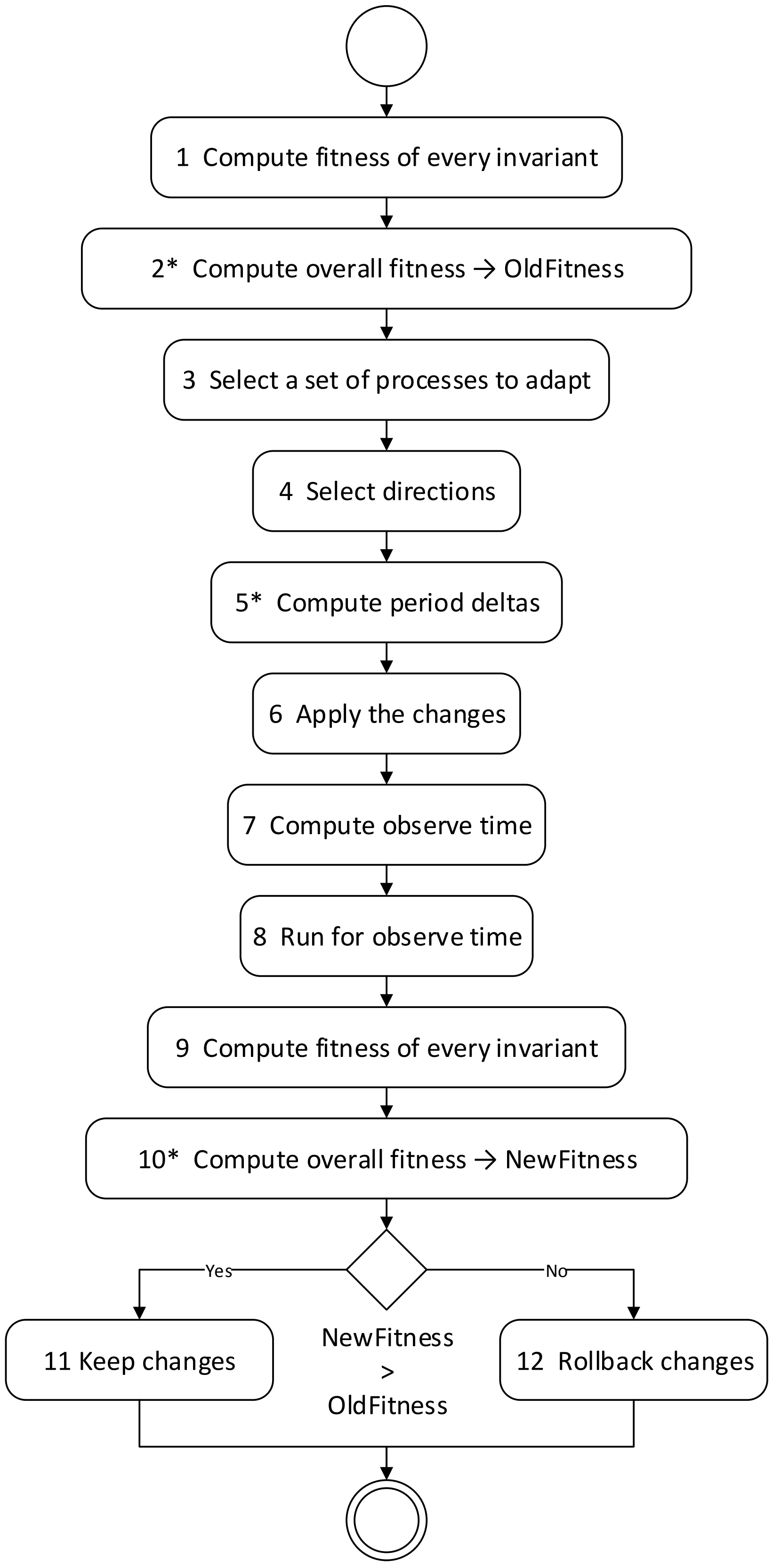

1 Compute fitness of every invariant
2* Compute overall fitness → OldFitness
3 Select a set of processes to adapt
4 Select directions
5* Compute period deltas
6 Apply the changes
7 Compute observe time
8 Run for observe time
9 Compute fitness of every invariant
10* Compute overall fitness → NewFitness
Yes
No
NewFitness > OldFitness
11 Keep changes
12 Rollback changes

Figure 11: Single run of the Process Period Adjusting strategy in a form of activity diagram. “*” denotes variation points of the algorithm.

4.3. Tactics Generated by Assumption Parameters Adjusting

Monitoring and analysis of the environment conditions are crucial for the CPS to self-adapt itself in face of dynamic changes. Assumptions about the environment of the CPS and its internal state are periodically re-evaluated by specialized (“monitoring”) tactics to decide which pre-designed situation the system is in. Descriptions and specifications of these situations are dependent on domain-specific knowledge in a form of behavioral models, such as timed automata or state-space models). However, if unforeseen circumstances occur at runtime, the expectations that these models are built on may be no longer valid and CPS fails to meet its goals. Such cases when the current situation is not effectively identified by the parameters of assumptions set manually are addressed by the following strategy. The description of the implementation of this strategy in jDEECo can be found in Section 5.4.2.

Strategy Name: Assumption Parameters Adjusting.

Intent: Preventing inappropriate, overdue or early actions by improving the monitoring and analysis phases by making the assumptions parameters as fitting to the present circumstances as possible.

Context: This strategy deals with scenarios when the current situation is not identified correctly because of the predesigned model with hardcoded domain knowledge does not expect such circumstances and thus there are no monitoring assumptions corresponding to the present status. In such cases, early, overdue or inappropriate adaptation actions may be taken which could jeopardize the efficiency or even the functionality of the whole system.

Behavior: The behavior of this meta-adaptation strategy is very similar to the Process Period Adjusting described in the previous chapter. Each assumption a_i from the set A of all assumptions in the system is provided with a corresponding runtime monitor accepting parameters P_i with values V_i. The runtime monitor of a_i returns a fitness value f_i which is a real number in range [0-1] that indicates whether the assumption is still satisfied. If f_i is lower than minimal acceptable value, new tactics that correspond to new assumptions a_i’ are generated. The assumptions a_i are transformed to a_i’ by adjusting values V_i of their parameters P_i to V_i’ while keeping them in the pre-defined allowed range. The generic algorithm (1+1)-ONLINE EA [23] is exploited to find the suitable adaptation as can be seen in Figure 12 and

Figure 15. Adjusting values V_i of a_i’s parameters P_i (line 12 in Figure 12) can be understood as substitution of tactic a_i by newly generated tactic a_i’. When there are no assumptions suitable for adjusting left, i.e. parameters of every assumption have been already adjusted in both directions, or pre-defined maximal number of tries has been reached, the assumption parameter adjusting is finished.

The benefit of individual tactics (i.e. new assumptions) is measured by comparing the overall system fitness (calculated as a function of individual f_i’s) after an observation period passes and the changes take effect on the running system.

```
1.  begin
2.    foreach Assumption from Assumptions do
3.    begin
4.        Compute fitness for Assumption
5.        OldFit = *CombineFitness(Assumptions.Fitnesses)
6.        Adaptees = *SelectParametersToAdaptTheirValues(Assumptions)
7.    end
8.    foreach Parameter from Adaptees do
9.    begin
10.       *Select the direction for param. adjustment (up or down) for Parameter
11.       *Calculate value delta (difference of old and new value) for Parameter
12.       Change value of Parameter
13.   end
14.   Run for predefined ObserveTime with no further adaptations for changes
      to take effect
15.   foreach Assumption from Assumptions do
16.       Compute fitness for Assumption
17.   NewFit = *CombineFitness(Assumptions.Fitnesses)
18.   if NewFit > OldFit then
19.       KeepChanges
20.   else
21.       Roll-back changes
22. end
```

Figure 12: Single run of the Process Period Adjusting strategy in a form of pseudocode. “*” denotes variation points of the algorithm.

Contraindications: Softening of the assumption parameters may be very dangerous in safety critical systems where values of these parameters are not only result of experiences of the application architects but come from strict specifications, such as laws etc. In these situations, relaxing assumptions is not a solution, so parameterizable assumptions can be fine tuned at design time and strict, fail-safe bounds of their parameters can be specified. All these measures ensure that the system is not put in a risk at any time.

Example: Consider altered situation from the case study depicted in Figure 14 where the technology for wireless communication is chosen based on the distance between the communicators. The technology with limited range is more reliable and less energy demanding, but susceptible to interference. The Wi-Fi technology at 5GHz is more power hungry and has longer range. The parameter "25" of assumptions 2 and 3 is a domain-specific knowledge coming from system architects' experience and specifications. However when unanticipated interference prevents the Bluetooth technology to function properly for this distance, this strategy can take care of decreasing the parameter, so more reliable technology is used, even though it drains the batteries more and it would not be efficient to use it for this distance under normal circumstances. Figure 13 illustrates such scenario.

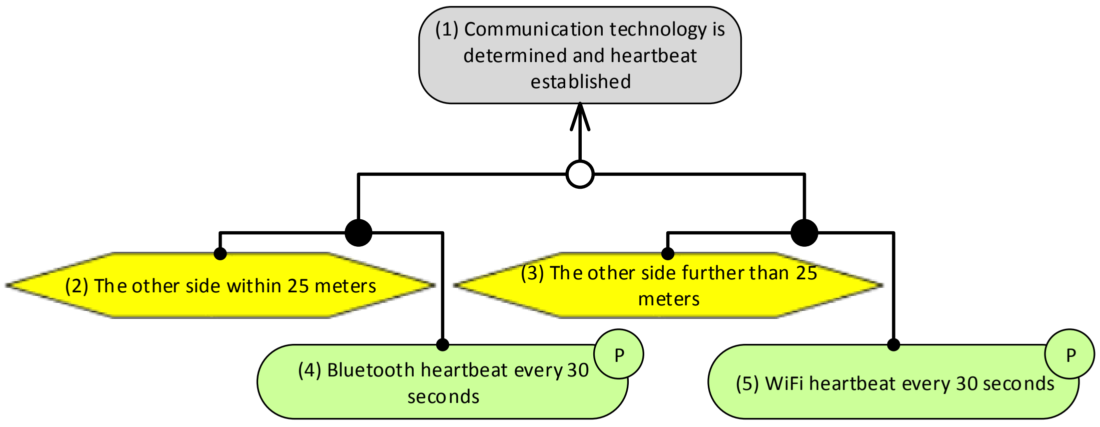

(1) Communication technology is determined and heartbeat established
(2) The other side within 25 meters
(3) The other side further than 25 meters
(4) Bluetooth heartbeat every 30 seconds
P
(5) WiFi heartbeat every 30 seconds
P

Figure 14: Alternative situation of the case study where heartbeat signals are required.

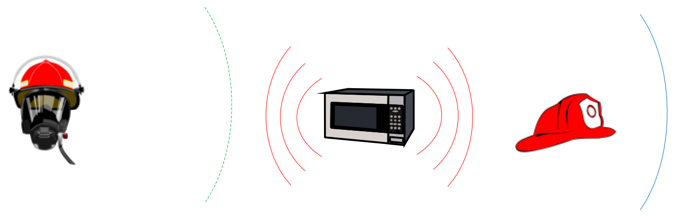


Figure 13: Example of problematic situation where unanticipated interference (red) requires adaptation from original Bluetooth range (blue) to shorter one (green), so 5GHz Wi-Fi is used.

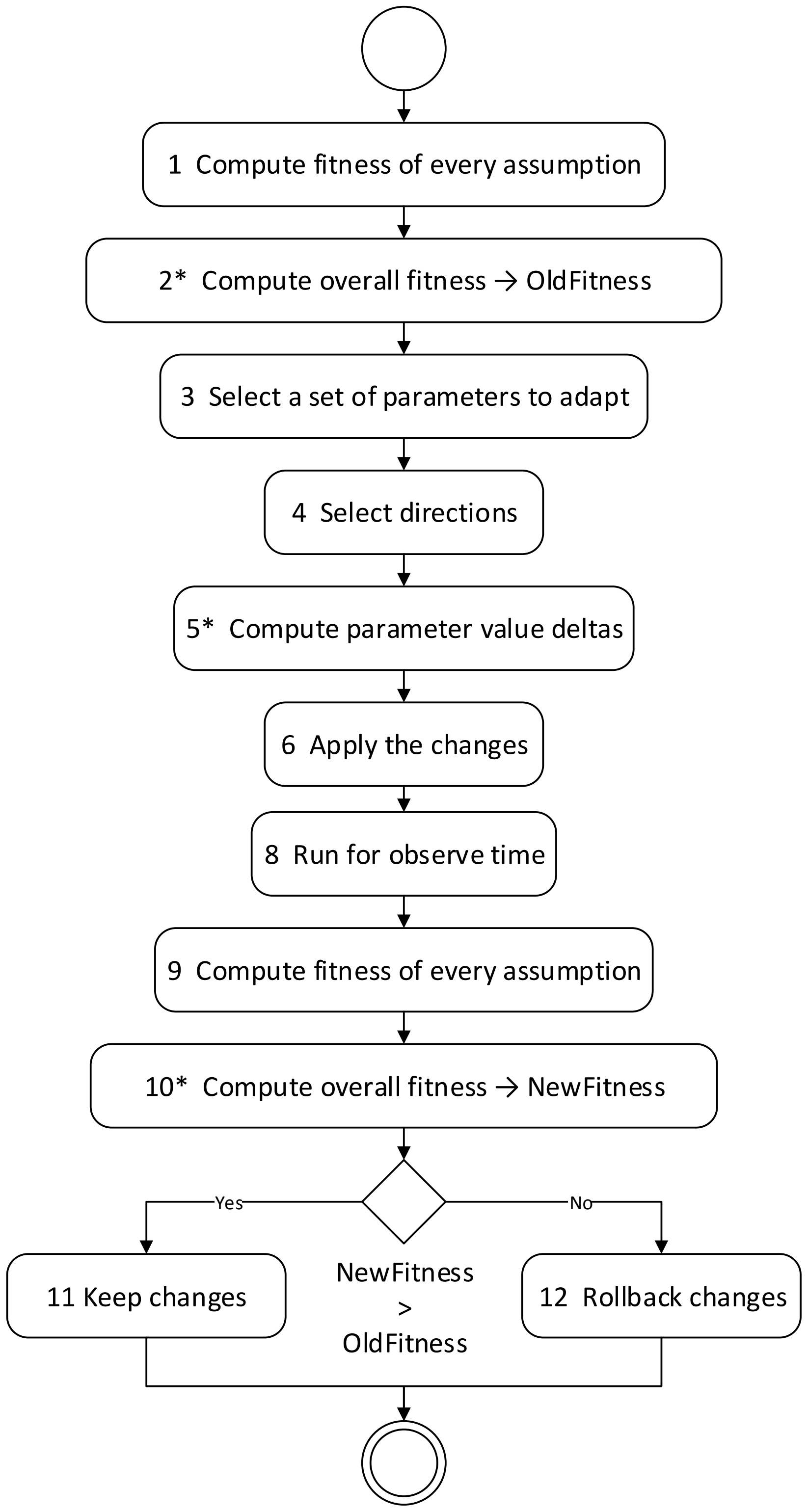

1 Compute fitness of every assumption
2* Compute overall fitness → OldFitness
3 Select a set of parameters to adapt
4 Select directions
5* Compute parameter value deltas
6 Apply the changes
8 Run for observe time
9 Compute fitness of every assumption
10* Compute overall fitness → NewFitness
Yes
No
11 Keep changes
NewFitness > OldFitness
12 Rollback changes

Figure 15: Single run of the Assumption Parameter Adjusting strategy in a form of activity diagram. "*" denotes variation points of the algorithm.

5. Implementation as jDEECo Plugins

This chapter contains information about the jDEECo implementations of the three meta-adaptation strategies described in Chapter 4. The first section provides the overall picture of how IRM-SA jDEECo plugin, the MetaAdaptationPlugin and the plugins corresponding to the individual meta-adaptations are coordinated. The next sections describe the individual components in more detail. The source codes can be found at the attached CD as well as online at project website [24].

5.1. Overall Picture

The existing jDEECo IRM Plugin [25] has been extended, so listeners to the result of the self-adaptation can be registered and serve as adapters between the plugins. The listeners implement AdaptationListener interface and are notified every time IRM-SA adaptation finishes via adaptationResult method. These notifications are the main source of information for MetaAdaptationPlugin managing individual meta-adaptation strategies. This pattern is chosen because IRM-SA should not depend on the meta-adaptation plugin, so it only provides simple interface which is implemented in the meta-adaptation plugin. These relationships are illustrated in Figure 8. The dependencies as defined for jDEECo plugins are depicted in Figure 16.

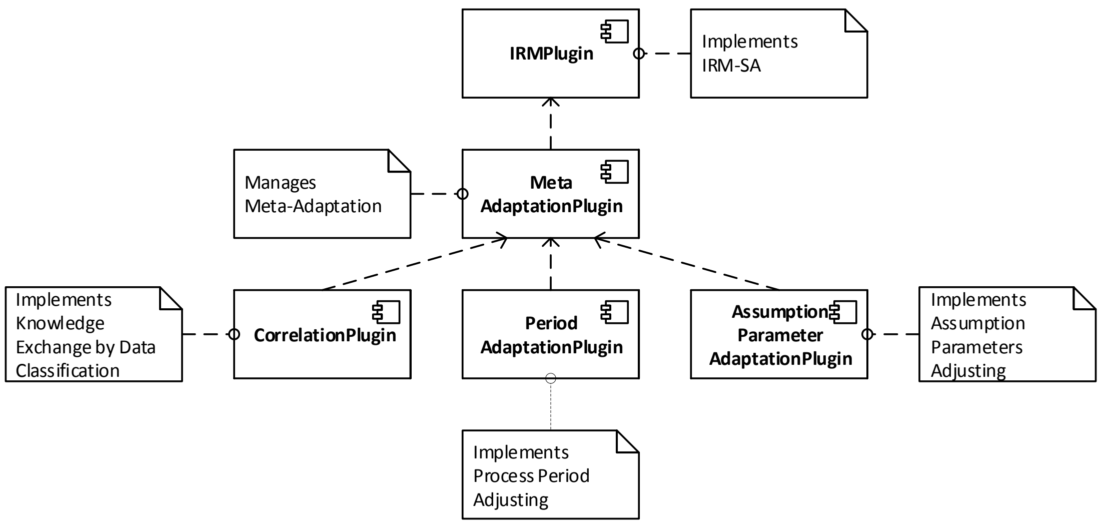

IRMPlugin
Implements IRM-SA
Manages Meta-Adaptation
Meta AdaptationPlugin
Implements Knowledge Exchange by Data Classification
CorrelationPlugin
Period AdaptationPlugin
Assumption Parameter AdaptationPlugin
Implements Assumption Parameters Adjusting
Implements Process Period Adjusting

Figure 16: Dependencies between jDEECo plugins implementing self-adaptation.

Technically, the implementation of the schema above is much more complicated because of limitations brought by jDEECo as shown in Figure 17. Classes representing jDEECo plugins are marked with stereotype instead of implementing

DEECoPlugin interface to keep the diagram clear. The list containing adaptation listeners of IRMPlugin is passed to AdaptationManager (which is jDEECo component responsible for IRM-SA self-adaptation) as part of its internal data. Internal data is a mechanism that allows the communication between jDEECo components deployed to the same runtime outside of ensemble concept and should only be exploited for system components, not application ones. AdaptationManager's static methods representing IRM processes can access this list via ComponentInstance provided by jDEECo framework and notify the listeners and ask them whether the IRM-SA self-adaptation can continue or the IRM-SA adaptation mechanism should be blocked until listeners' work is finished. In other words, the behavior of the AdaptationManager has been altered to not proceed with IRM-SA adaptation if there are listeners that do not wish to be disturbed, such as meta-adaptation strategies that perform self-adaptation of their own. This synchronization is needed as threading model of the jDEECo is not visible to its components and component processes should not block and instead finish as soon as possible, so collisions with process scheduling are avoided.

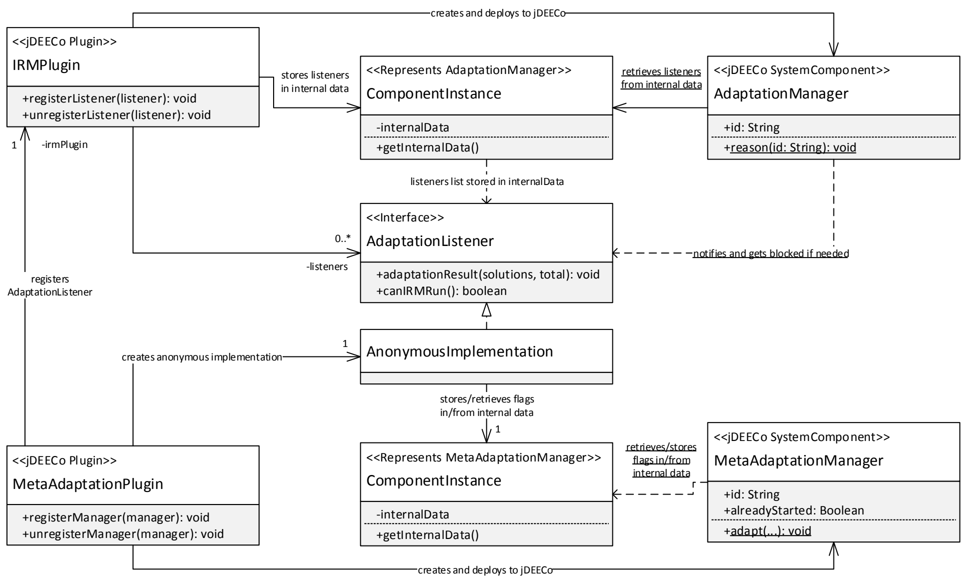

creates and deploys to jDEECo
<<jDEECo Plugin>>
IRMPlugin
+registerListener(listener): void
+unregisterListener(listener): void
stores listeners in internal data
<<Represents AdaptationManager>>
ComponentInstance
-internalData
+getInternalData()
retrieves listeners from internal data
<<jDEECo SystemComponent>>
AdaptationManager
+id: String
+reason(id: String): void
1
-irmPlugin
listeners list stored in internalData
<<Interface>>
AdaptationListener
+adaptationResult(solutions, total): void
+canIRMRun(): boolean
0..*
-listeners
notifies and gets blocked if needed
registers AdaptationListener
creates anonymous implementation
1
AnonymousImplementation
stores/retrieves flags in/from internal data
1
<<jDEECo Plugin>>
MetaAdaptationPlugin
+registerManager(manager): void
+unregisterManager(manager): void
<<Represents MetaAdaptationManager>>
ComponentInstance
-internalData
+getInternalData()
retrieves/stores flags in/from internal data
<<jDEECo SystemComponent>>
MetaAdaptationManager
+id: String
+alreadyStarted: Boolean
+adapt(...): void
creates and deploys to jDEECo

Figure 17: Class diagram depicting actual relations and communication between IRMPlugin and MetaAdaptationPlugin

Both AdaptationManager and MetaAdaptationManager as well as other components excluded from the IRM model need to be marked as such by

"@SystemComponent" (depicted as stereotypes in Figure 17) so the adaptations do not adapt these crucial processes which could result in unexpected behavior.

The whole meta-adaptation extension of the IRM-SA is organized into Eclipse [26] projects. The project cz.cuni.mff.d3s.jdeeco.irm-sa.strategies contains package called cz.cuni.mff.d3s.irm-sa.strategies with all the classes related to MetaAdaptationPlugin described in Section 5.2.

The same project contains also packages for common behavior of (1+1)-ONLINE EA Plugins (see Section 5.4), PeriodAdaptationPlugin (Section 5.4.1) and AssumptionParameterAdaptationPlugin (Section 5.4.2).

Because of organization reasons, the CorrelationPlugin (Section 5.3) has its own separated project cz.cuni.mff.d3s.irm-sa.strategies.correlation dependent on the main project.

5.2. MetaAdaptationPlugin

This plugin is responsible for deploying MetaAdaptationManager into jDEECo runtime which manages plugins/managers implementing meta-adaptation strategies that are described in the following sections. It provides similar interface methods for registering adaptation managers as IRM-SA AdaptationPlugin; their communication patterns are analogous. This can be seen in Figure 19 with example *Plugin instead of actual plugin responsible for meta-adaptation. The package structure of the project is illustrated in Figure 18.

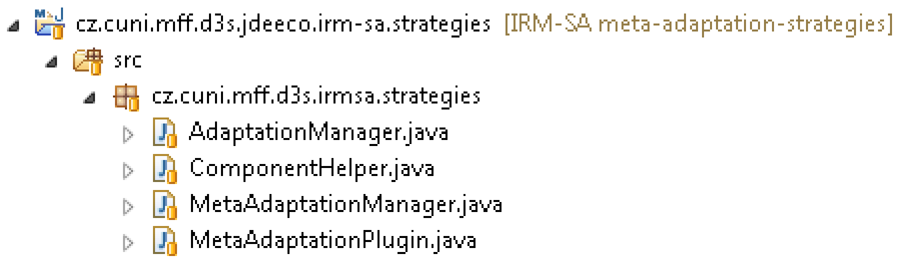

cz.cuni.mff.d3s.jdeeco.irm-sa.strategies [IRM-SA meta-adaptation-strategies]
src
cz.cuni.mff.d3s.irmsa.strategies
AdaptationManager.java
ComponentHelper.java
MetaAdaptationManager.java
MetaAdaptationPlugin.java

Figure 18: Package of MetaAdaptationPlugin inside its project.

The MetaAdaptationPlugin takes advantage of jDEECo dependency management and its only dependency is the IRM-SA AdaptationPlugin. It becomes the dependency of the three meta-adaptation strategies (Figure 16).

MetaAdaptationManager is a jDEECo component which is notified by the IRMPlugin via anonymous implementation of the interface AdaptationListener and

ComponentInstance’s internal data as shown in Figure 17. Analogously, interface AdaptationManager serves as an adapter between MetaAdaptationManager and components responsible for individual meta-adaptation strategies. When IRM-SA fails to find an applicable configuration, anonymous implementation of the AdaptationListener sets flag RUN_FLAG in internal data of the ComponentInstance corresponding to MetaAdaptationManager. Having fetched this flag, MetaAdaptationManager uses the list of AdaptationManagers stored in its internal data and starts the meta-adaptation strategies that correspond to the AdaptationManagers, it keeps a flag alreadyStarted indicating whether the managers have been already started to avoid multiple calls of AdaptationManagers’ run() methods. The AdaptationListener blocks IRM-SA self-adaptation until MetaAdapationManager sets the appropriate flag IRM_CAN_RUN (again in the internal data) indicating that all AdaptationManagers signal that their adaptations are finished, either successfully or not.

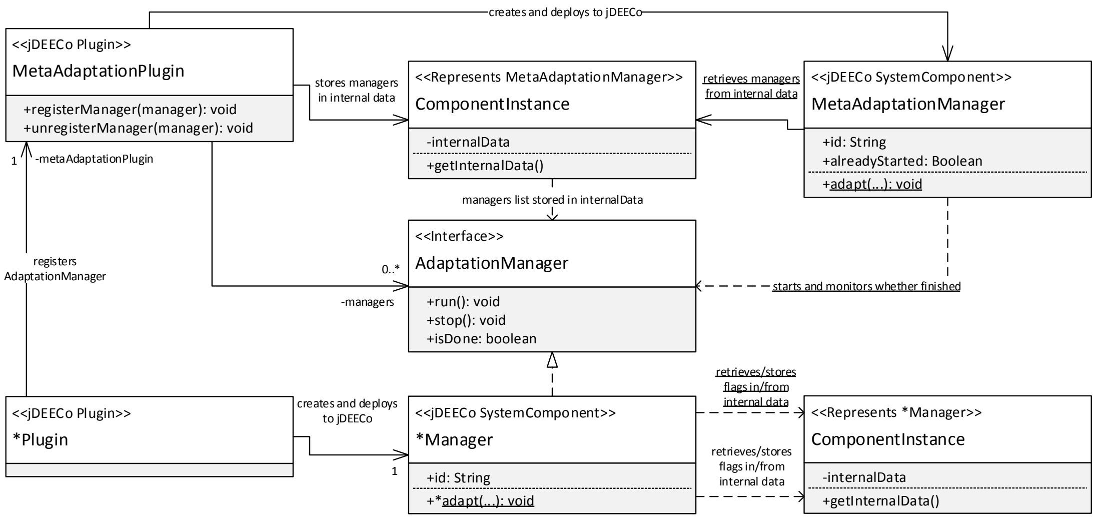
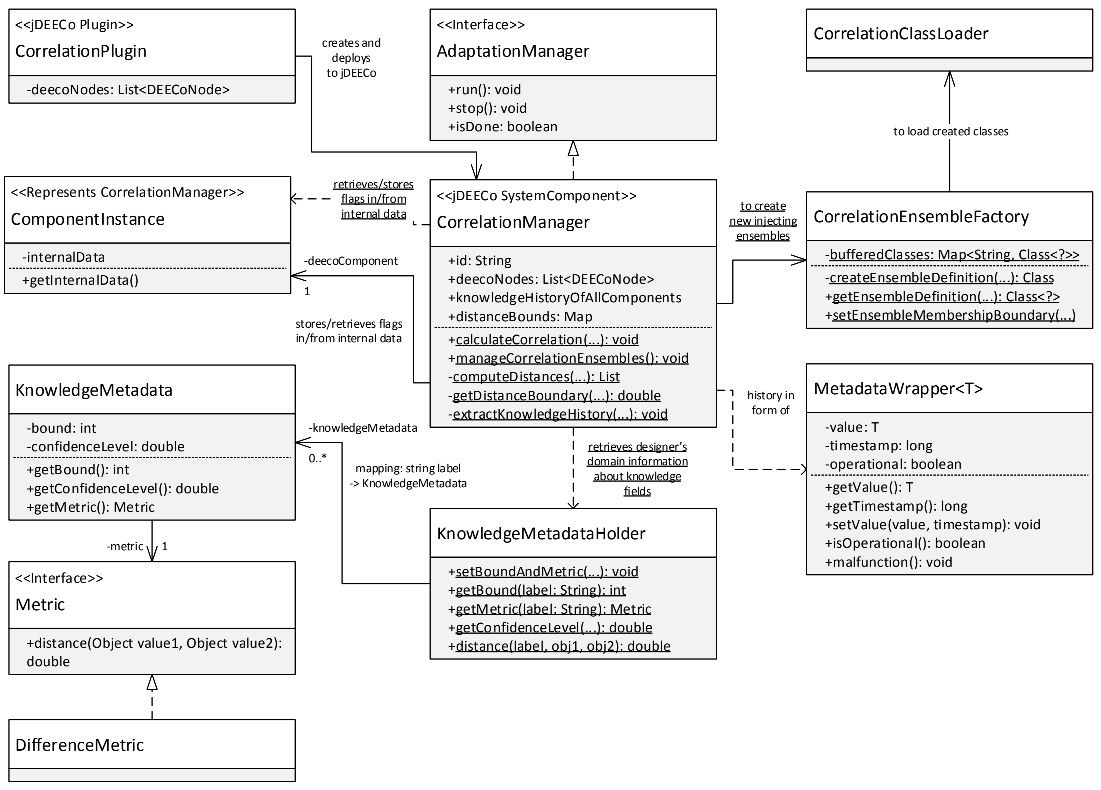

creates and deploys to jDEECo
<<jDEECo Plugin>>
MetaAdaptationPlugin
+registerManager(manager): void
+unregisterManager(manager): void
stores managers in internal data
<<Represents MetaAdaptationManager>>
ComponentInstance
-internalData
+getInternalData()
retrieves managers from internal data
<<jDEECo SystemComponent>>
MetaAdaptationManager
+id: String
+alreadyStarted: Boolean
+adapt(...): void
1
-metaAdaptationPlugin
managers list stored in internalData
<<Interface>>
AdaptationManager
+run(): void
+stop(): void
+isDone: boolean
registers AdaptationManager
0..*
-managers
starts and monitors whether finished
retrieves/stores flags in/from internal data
<<jDEECo Plugin>>
*Plugin
creates and deploys to jDEECo
1
<<jDEECo SystemComponent>>
*Manager
+id: String
+*adapt(...): void
retrieves/stores flags in/from internal data
<<Represents *Manager>>
ComponentInstance
-internalData
+getInternalData()

Figure 19: Class diagram depicting relations and communication between MetaAdaptationPlugin and example *Plugin implementing a meta-ataptation strategy.

Current implementation of the plugin is very simplistic and does not apply any sophisticated mechanism to manage the AdaptationManagers. This is definitely room for improvement as elaborated in Chapter 8 – however the AdaptationManager interface would probably need to be overhauled to provide detailed information about the adaptation method along with descriptions of both the situations that the adaptation method is appropriate for.

5.3. CorrelationPlugin

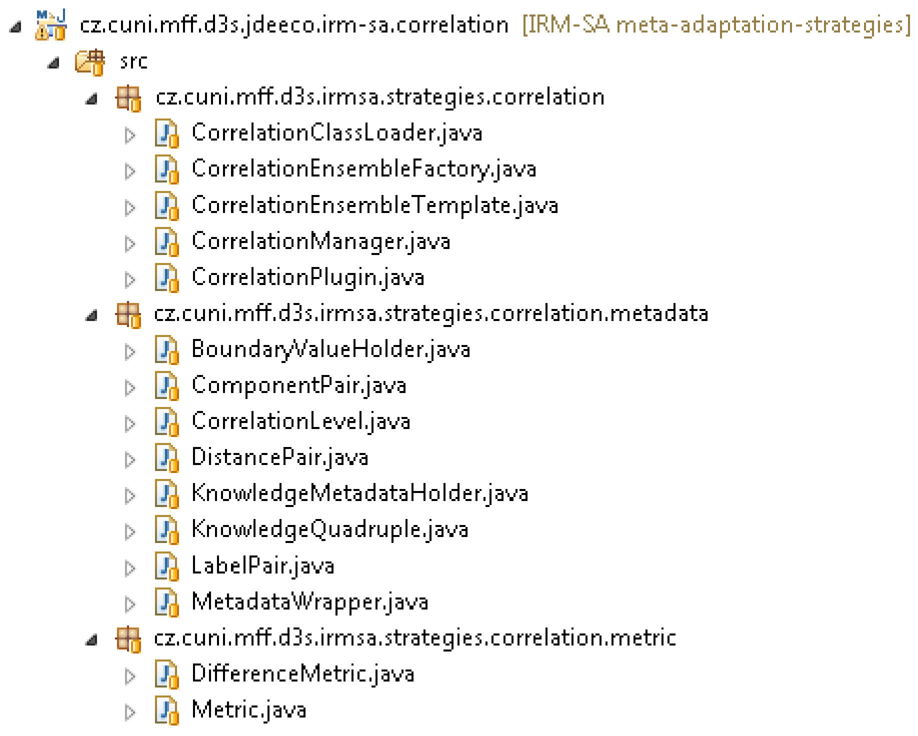
This plugin implements Knowledge Exchange by Data Classification meta-adaptation strategy (Section 4.1). Because of organization reasons, it resides inside its own project cz.cuni.mff.d3s.irm-sa.strategies.correlation dependent on the main meta-adaptation project. Its structure can be seen in Figure 21. The relationships and communication between classes are illustrated in Figure 20.



<<jDEECo Plugin>>
CorrelationPlugin
-deecoNodes: List<DEECoNode>
creates and deploys to jDEECo
<<Interface>>
AdaptationManager
+run(): void
+stop(): void
+isDone: boolean
CorrelationClassLoader
to load created classes
<<Represents CorrelationManager>>
ComponentInstance
-internalData
+getInternalData()
retrieves/stores flags in/from internal data
<<jDEECo SystemComponent>>
CorrelationManager
+id: String
+deecoNodes: List<DEECoNode>
+knowledgeHistoryOfAllComponents
+distanceBounds: Map
+calculateCorrelation(...): void
+manageCorrelationEnsembles(): void
-computeDistances(...): List
-getDistanceBoundary(...): double
-extractKnowledgeHistory(...): void
to create new injecting ensembles
CorrelationEnsembleFactory
-bufferedClasses: Map<String, Class<?>>
-createEnsembleDefinition(...): Class
+getEnsembleDefinition(...): Class<?>
+setEnsembleMembershipBoundary(...)
-deecoComponent
1
stores/retrieves flags in/from internal data
KnowledgeMetadata
-bound: int
-confidenceLevel: double
+getBound(): int
+getConfidenceLevel(): double
+getMetric(): Metric
-knowledgeMetadata
0..*
mapping: string label -> KnowledgeMetadata
history in form of
MetadataWrapper<T>
-value: T
-timestamp: long
-operational: boolean
+getValue(): T
+getTimestamp(): long
+setValue(value, timestamp): void
+isOperational(): boolean
+malfunction(): void
retrieves designer's domain information about knowledge fields
KnowledgeMetadataHolder
+setBoundAndMetric(...): void
+getBound(label: String): int
+getMetric(label: String): Metric
+getConfidenceLevel(...): double
+distance(label, obj1, obj2): double
-metric
1
<<Interface>>
Metric
+distance(Object value1, Object value2): double
DifferenceMetric

Figure 20: UML class diagram depicting relationships and communication between classes from CorrelationPlugin's project.

The communication pattern is the one depicted in the template of Figure 19. The current implementation of CorrelationPlugin is for demonstration purposes intended for online data correlation analysis. For that reason, it needs to pass deecoNodes to the CorrelationManager that is a jDEECo component implementing the meta-adaptation strategy itself to extract knowledge from all accessible adaptable components (done in the static private method extractKnowledgeHistory) to knowledgeHistoryOfAllComponents in a form of MetadataWrapper which must wrap knowledge fields that are supposed to be adaptable. At runtime, distance boundaries for pairs of component knowledge fields (mapped in field distanceBounds) are computed (in jDEECo process calculateCorrelation) using domain information about knowledge fields passed to the strategy at design time to KnowledgeMetadataHolder (see below). The distance may be NaN to indicate that no correlation is found. In this meta-adaptation data of a specific knowledge field from other components are injected to a component that cannot obtain fresh values anymore (e.g. due to sensor failure). Data are injected based on data correlation – the correlation with the highest confidence level is picked if there are more than one. Technically, CorrelationManager creates a new ensemble class (in jDEECo process manageCorrelationEnsembles) via CorrelationEnsembleFactory loaded by CorrelationClassLoader and deploys it to the jDEECo framework. The same process also again un-deploys this ensemble in cases when the correlation is no longer valid.



```
cz.cuni.mff.d3s.jdeeco.irm-sa.correlation [IRM-SA meta-adaptation-strategies]
  src
    cz.cuni.mff.d3s.irmsa.strategies.correlation
      CorrelationClassLoader.java
      CorrelationEnsembleFactory.java
      CorrelationEnsembleTemplate.java
      CorrelationManager.java
      CorrelationPlugin.java
    cz.cuni.mff.d3s.irmsa.strategies.correlation.metadata
      BoundaryValueHolder.java
      ComponentPair.java
      CorrelationLevel.java
      DistancePair.java
      KnowledgeMetadataHolder.java
      KnowledgeQuadruple.java
      LabelPair.java
      MetadataWrapper.java
    cz.cuni.mff.d3s.irmsa.strategies.correlation.metric
      DifferenceMetric.java
      Metric.java
```

Figure 21: Package of CorrelationPlugin inside its own separated project.

This meta-adaptation strategy requires initial input at design time to be passed to KnowledgeMetadataHolder: (i) required confidence level, (ii) Metric providing distance between values and (iii) bound that separates values considered close from the rest. The Metric is a simple interface providing distance between two objects representing values of knowledge fields that are analyzed. For convenience there is an implementation of DifferenceMetric for descendants of Number class, but for more complex types there must be user-defined one. For example Euclidean distance for two-dimensional position was used in the evaluation scenario.

The knowledge fields that are supposed to participate in this meta-adaptation needs to be wrapped in MetadataWrapper. This is required because the plugin needs to distinguish whether the sensor is operational (MetadataWrapper‘s boolean field operational) and also needs to pair the values by their timestamps (MetadataWrapper’s long field timestamp) to correctly discover the correlation.

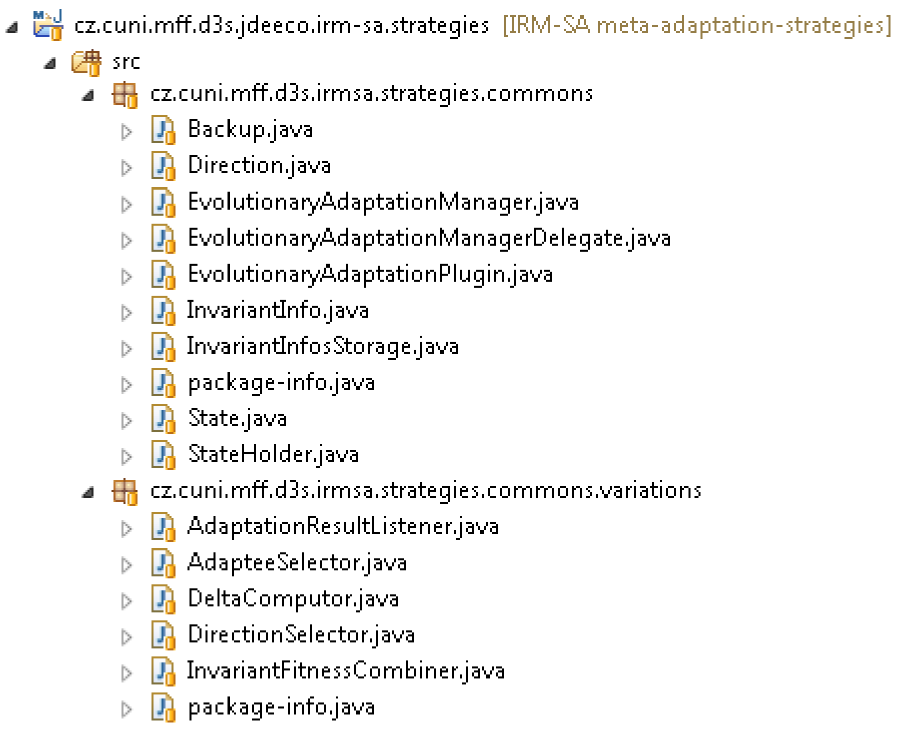
Also note that KnowledgeMetadata is a private static inner class of KnowledgeMetadataHolder, so it is not visible at project overview in Figure 19. One more discrepancy in that figure is class CorrelationEnsembleTemplate which is not used at all, but serves only as an example of a class generated by CorrelationEnsembleFactory at runtime via javassist library [27].

The interesting feature of this implementation of the meta-adaptation strategy is the fact that it is completely independent of IRM-SA invariant tree and hierarchy, unlike the two other meta-adaptation strategy implementations.

5.4. (1+1)-ONLINE EA Plugins

Process Period Adjusting and Assumption Parameters Adjusting strategies have very similar structure, as they both exploit (1+1)-ONLINE EA and IRM-SA invariant hierarchy; therefore code-reuse is desirable between the jDEECo plugins implementing these two strategies. The common parts of the plugins are located in package cz.cuni.mff.d3s.irmsa.strategies.commons and its sub-package variations in project cz.cuni.mff.d3s.jdeeco.irm-sa.strategies (Figure 22).

The relationships and communication of classes in these packages are depicted in Figure 23. The entry point to the common functionality provided by these packages is EvolutionaryAdaptationPlugin depending on MetaAdaptationPlugin which is an abstract class intended to be ancestor of plugins implementing meta-adaptation strategies based on the evolutionary algorithm. The communication pattern is the one depicted in Figure 19. Descendants of the plugin need to provide implementation for several methods, mostly default implementations of interfaces from sub-package variations which represent variation points of the strategy as described in Chapter 4. Moreover, the method provideDataToManager() can be overridden to provide more data to the EvolutionaryAdaptationManager which is the main jDEECo component responsible for the adaptation. The information about IRM-SA invariants with additional information needed for the algorithm is stored and passed along in a form of collections of instances of InvariantInfo class.



```
cz.cuni.mff.d3s.jdeeco.irm-sa.strategies [IRM-SA meta-adaptation-strategies]
  src
    cz.cuni.mff.d3s.irmsa.strategies.commons
      Backup.java
      Direction.java
      EvolutionaryAdaptationManager.java
      EvolutionaryAdaptationManagerDelegate.java
      EvolutionaryAdaptationPlugin.java
      InvariantInfo.java
      InvariantInfosStorage.java
      package-info.java
      State.java
      StateHolder.java
    cz.cuni.mff.d3s.irmsa.strategies.commons.variations
      AdaptationResultListener.java
      AdapteeSelector.java
      DeltaComputor.java
      DirectionSelector.java
      InvariantFitnessCombiner.java
      package-info.java
```

Figure 22: Packages with common functionality for meta-adaptation plugins exploiting (1+1)-ONLINE EA.

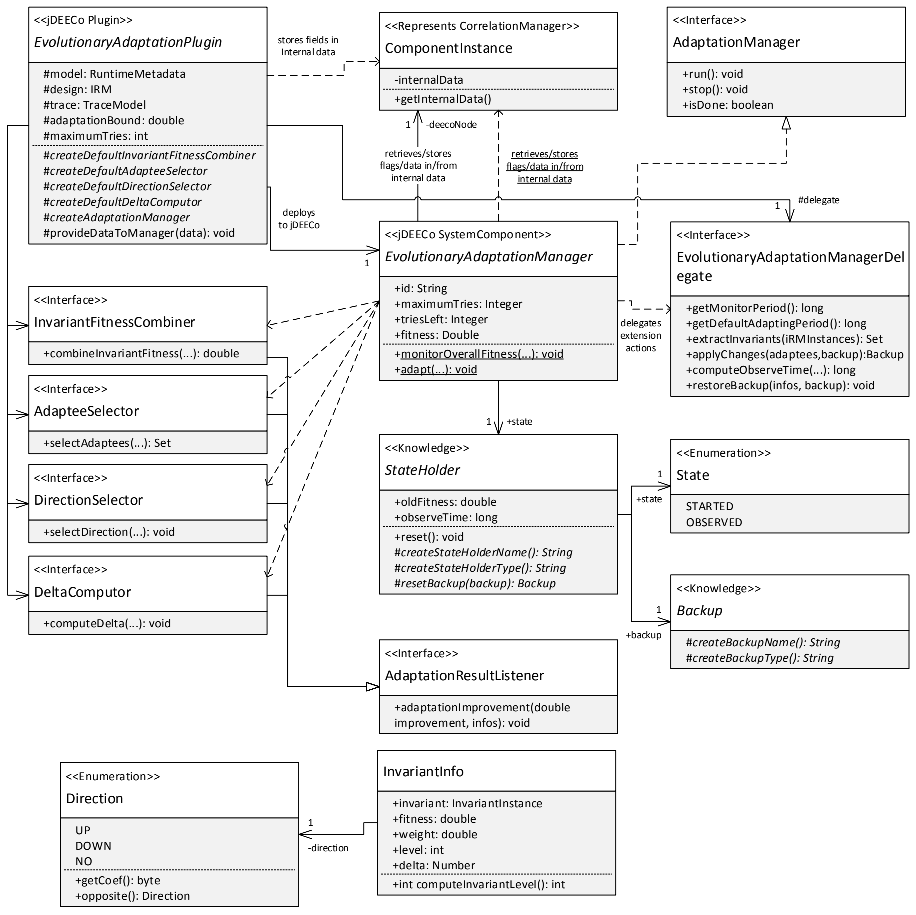
The adaptation process is divided into two steps and thus slightly differs from the flow charts from the meta-adaptation strategies templates. However, it complies with the original specifications in all other aspects. The whole process is started by MetaAdaptationManager indicating that IRM-SA could not find applicable configuration. Firstly, the overall system fitness is monitored in monitorOverallFitness with help of InvariantFitnessCombiner. The result is stored in knowledge field fitness and auxiliary structures needed for computation are stored for later use in the second step. Secondly, the adaptation itself is handled in process adapt. The state of the adapt process is stored in knowledge field StateHolder containing the flag (field state) indicating whether new tactic should be devised or new tactic has been observed and now it is time to evaluate it and revert it if it proves disadvantageous. All information to restore the state before application of the tactic is part of the StateHolder knowledge in field backup. After an adaptation is accepted or discarded, interfaces representing variation points are notified via AdaptationResultListener.

If the overall fitness is above adaptationBound or the maximal number of tries is reached, the self-adaptation stops and indicates it to the MetaAdaptationManager.

There is one inconvenience in jDEECo process scheduling that the plugin must work around: the process is already scheduled for the next run when its method is called. That means that the adapt process can only change its own period with lag of one run. It causes no significant problems when the observation time is equal to or larger than adapt's period. Otherwise the observation takes longer than required, but this should not be harmful in most cases. More aggressive solution manipulating with jDEECo internal process scheduling queue may remove these issues, but would also bind the plugin with private API of jDEECo which is not desirable.

Because Process Period Adjusting and Assumption Parameters Adjusting strategies are similar but not identical, the extension points where the behavior of the EvolutionaryAdaptationManager and related classes can be customized are provided. To implement new meta-adaptation based on this template plugin, one has to extend the EvolutionaryAdaptationPlugin, StateHolder and Backup classes. The main extension point is interface EvolutionaryAdaptationManagerDelegate which is delegated all extension actions from EvolutionaryAdaptationManager using delegate pattern [22]. This approach is chosen because static nature of jDEECo components prevents the convenient extension based on inheritance. At least one implementation of each variation point of the algorithm must be provided.

A lot of the mentioned classes are generic. The main reason behind this is the fact that this enables better type safety (that is hardcode overtyping is rarer, e.g. for Backup and StateHolder) and various "withers" (i.e. setters returning the object itself to enable fluent typing) can return the correct object type instead of the instance of class where these withers are declared.



<<jDEECo Plugin>>
EvolutionaryAdaptationPlugin
#model: RuntimeMetadata
#design: IRM
#trace: TraceModel
#adaptationBound: double
#maximumTries: int
#createDefaultInvariantFitnessCombiner
#createDefaultAdapteeSelector
#createDefaultDirectionSelector
#createDefaultDeltaComputor
#createAdaptationManager
#provideDataToManager(data): void
stores fields in Internal data
<<Represents CorrelationManager>>
ComponentInstance
-internalData
+getInternalData()
1
-deecoNode
retrieves/stores flags/data in/from internal data
retrieves/stores flags/data in/from internal data
<<Interface>>
AdaptationManager
+run(): void
+stop(): void
+isDone: boolean
1
#delegate
deploys to jDEECo
1
<<jDEECo SystemComponent>>
EvolutionaryAdaptationManager
+id: String
+maximumTries: Integer
+triesLeft: Integer
+fitness: Double
+monitorOverallFitness(...): void
+adapt(...): void
delegates extension actions
<<Interface>>
EvolutionaryAdaptationManagerDelegate
+getMonitorPeriod(): long
+getDefaultAdaptingPeriod(): long
+extractInvariants(IRMInstances): Set
+applyChanges(adaptees,backup):Backup
+computeObserveTime(...): long
+restoreBackup(infos, backup): void
<<Interface>>
InvariantFitnessCombiner
+combineInvariantFitness(...): double
<<Interface>>
AdapteeSelector
+selectAdaptees(...): Set
<<Interface>>
DirectionSelector
+selectDirection(...): void
<<Interface>>
DeltaComputor
+computeDelta(...): void
1
+state
<<Knowledge>>
StateHolder
+oldFitness: double
+observeTime: long
+reset(): void
#createStateHolderName(): String
#createStateHolderType(): String
#resetBackup(backup): Backup
1
+state
<<Enumeration>>
State
STARTED
OBSERVED
1
+backup
<<Knowledge>>
Backup
#createBackupName(): String
#createBackupType(): String
<<Interface>>
AdaptationResultListener
+adaptationImprovement(double improvement, infos): void
<<Enumeration>>
Direction
UP
DOWN
NO
+getCoef(): byte
+opposite(): Direction
1
-direction
InvariantInfo
+invariant: InvariantInstance
+fitness: double
+weight: double
+level: int
+delta: Number
+int computeInvariantLevel(): int

Figure 23: UML class diagram depicting relationships and communication between classes from EvolutionaryAdaptationPlugin's project.

The integration to jDEECo framework and IRM-SA is depicted in Figure 24. Several extensions of IRM-SA to support meta-adaptations are described there. The IRM-SA model has been extended with weight of invariant (how important the invariant's fitness should be to the overall fitness for some InvariantFitnessCombiners) and minimal and maximal periods for process invariants. New type of monitor has been introduced for monitoring invariants' fitnesses returning double value instead of old satisfaction monitor's boolean value. IRMInstanceGenerator now handles these new monitors and sets their outputs into new field fitness of InvariantInstances. New annotation "@AssumptionParameter" is intended to provide default value, minimal and maximal bound, scope and initial adaptation direction for parameters of assumption monitors (for detailed information see Section 5.4.2) and is exploited by IRMInstanceGenerator when preparing non-adapted parameters for calling assumption monitors (both satisfaction and fitness monitors).

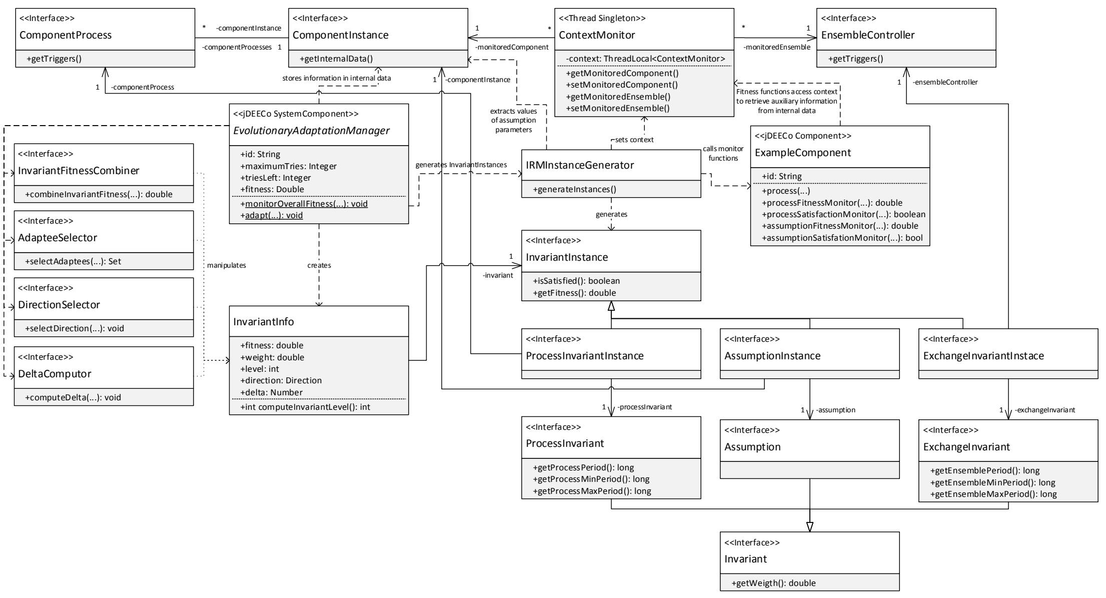


Figure 24: EvolutionaryAdaptationManager set in greater context of jDEECo framework and IRM-SA

5.4.1. PeriodAdaptationPlugin

This plugin extends EvolutionaryAdaptationPlugin and implements Process Period Adjusting meta-adaptation strategy. The structure of its package is illustrated in Figure 25.

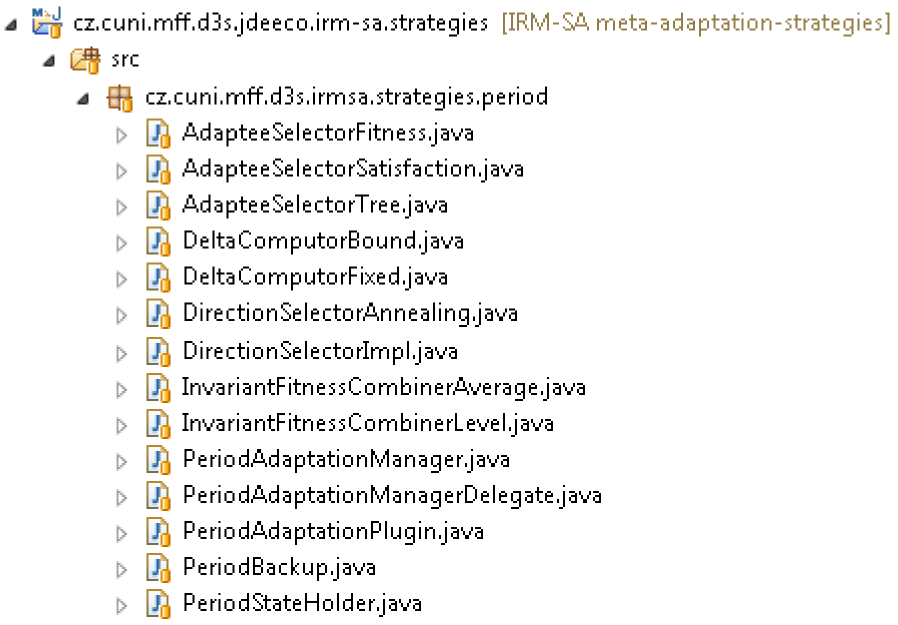

```
◢ cz.cuni.mff.d3s.jdeeco.irm-sa.strategies [IRM-SA meta-adaptation-strategies]
  ◢ src
    ◢ cz.cuni.mff.d3s.irmsa.strategies.period
      ▷ AdapteeSelectorFitness.java
      ▷ AdapteeSelectorSatisfaction.java
      ▷ AdapteeSelectorTree.java
      ▷ DeltaComputorBound.java
      ▷ DeltaComputorFixed.java
      ▷ DirectionSelectorAnnealing.java
      ▷ DirectionSelectorImpl.java
      ▷ InvariantFitnessCombinerAverage.java
      ▷ InvariantFitnessCombinerLevel.java
      ▷ PeriodAdaptationManager.java
      ▷ PeriodAdaptationManagerDelegate.java
      ▷ PeriodAdaptationPlugin.java
      ▷ PeriodBackup.java
      ▷ PeriodStateHolder.java
```

Figure 25: Package of PeriodAdaptationPlugin inside its project.

The code of the plugin and its delegate is very straightforward thanks to robust infrastructure provided by EvolutionaryAdaptationPlugin. The periods of the component processes are changed via getTriggers() method of

ComponentProcessClass and locating TimeTrigger object in the returned collection. This object can be used to set the period to a new value. The overview of the most important classes can be seen in Figure 26.

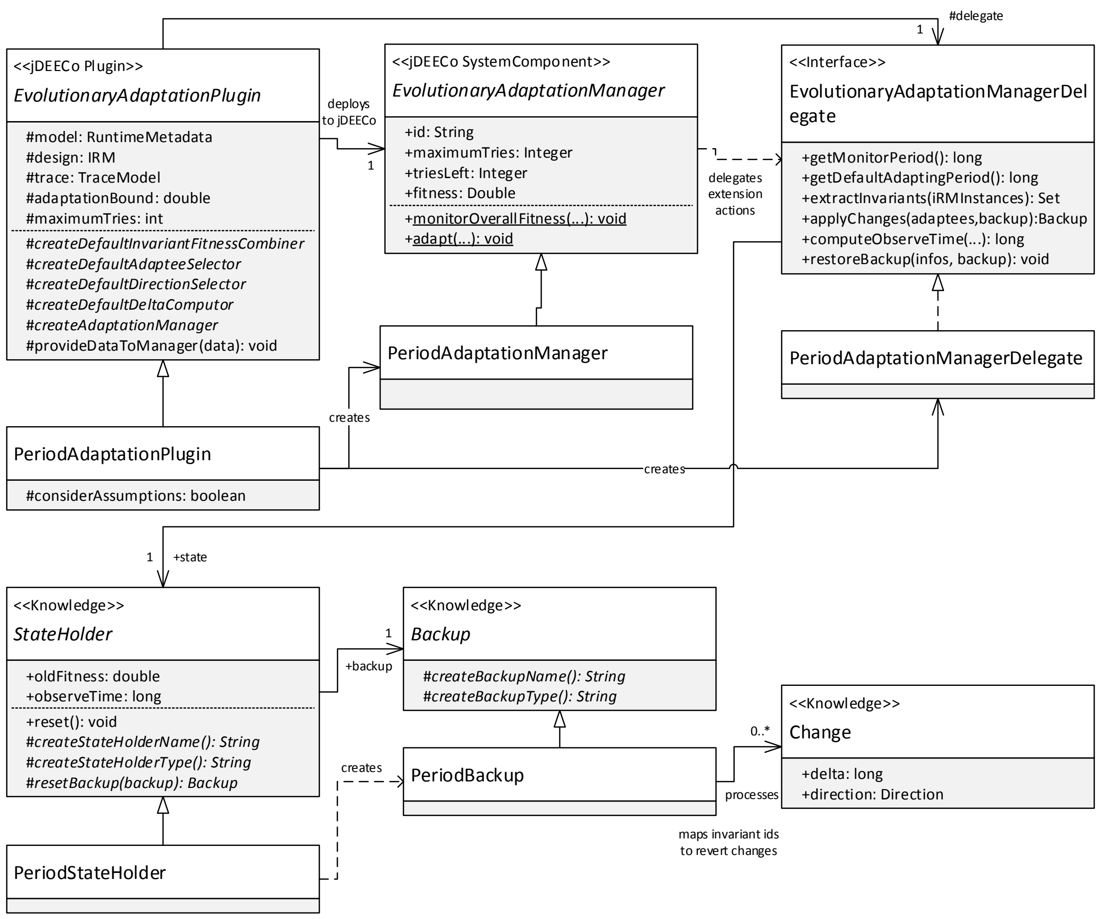

#delegate
1
<<jDEECo Plugin>>
EvolutionaryAdaptationPlugin
#model: RuntimeMetadata
#design: IRM
#trace: TraceModel
#adaptationBound: double
#maximumTries: int
#createDefaultInvariantFitnessCombiner
#createDefaultAdapteeSelector
#createDefaultDirectionSelector
#createDefaultDeltaComputor
#createAdaptationManager
#provideDataToManager(data): void
deploys to jDEECo
1
<<jDEECo SystemComponent>>
EvolutionaryAdaptationManager
+id: String
+maximumTries: Integer
+triesLeft: Integer
+fitness: Double
+monitorOverallFitness(...): void
+adapt(...): void
delegates extension actions
<<Interface>>
EvolutionaryAdaptationManagerDelegate
+getMonitorPeriod(): long
+getDefaultAdaptingPeriod(): long
+extractInvariants(iRMInstances): Set
+applyChanges(adaptees,backup):Backup
+computeObserveTime(...): long
+restoreBackup(infos, backup): void
PeriodAdaptationManager
PeriodAdaptationManagerDelegate
creates
PeriodAdaptationPlugin
#considerAssumptions: boolean
creates
1
+state
<<Knowledge>>
StateHolder
+oldFitness: double
+observeTime: long
+reset(): void
#createStateHolderName(): String
#createStateHolderType(): String
#resetBackup(backup): Backup
1
+backup
<<Knowledge>>
Backup
#createBackupName(): String
#createBackupType(): String
0..*
<<Knowledge>>
Change
+delta: long
+direction: Direction
creates
PeriodBackup
processes
maps invariant ids to revert changes
PeriodStateHolder

Figure 26: Overview of the most important classes of PeriodAdaptationPlugin.

The implementation provides a great number of variation points. The overall system fitness (variation point InvariantFitnessCombiner) can be computed in two ways: (i) compute weighted average fitness by invariants' weight defined at design time (InvariantFitnessCombinerAverage) or (ii) compute weighted average fitness using invariant level as weight (InvariantFitnessCombinerLevel). Potential adaptees (variation point AdapteeSelector) can be selected in two ways: (i) the ones with lowest fitness (AdapteeSelectorFitness), (ii) the ones higher in the IRM tree (AdapteeSelectorTree). The direction of the adaptation (i.e. determining whether the adaptee's period should increase or decrease, variation point DirectionSelector) can be determined in two ways: (i) try down and then up if previous try is not successful (DirectionSelectorImpl) or (ii) by simulated annealing [28]

(DirectionSelectorAnnealing). The deltas of period (variation point DeltaComputor) can be computed in two ways: (i) fixed delta provided at design time (DeltaComputorFixed), (ii) delta to set the period in the middle of current period and its bound (DeltaComputorBound). The users of the plugin are encouraged to provide own implementations of the variation points of the algorithm as those provided by default are merely examples and proof of concept.

The required user input at design time is as follows. IRM-SA model must contain minimal and maximal process periods. Monitors returning double value representing fitness value must be provided along with existing IRM-SA satisfaction monitors returning boolean values. Both monitors use the “@Monitor” annotation. When creating the plugin, the user specifies the variants of the algorithm to be used by passing implementations of corresponding interfaces to the plugin before adding it to jDEECo framework. Example of the plugin’s creation is depicted in Figure 28.

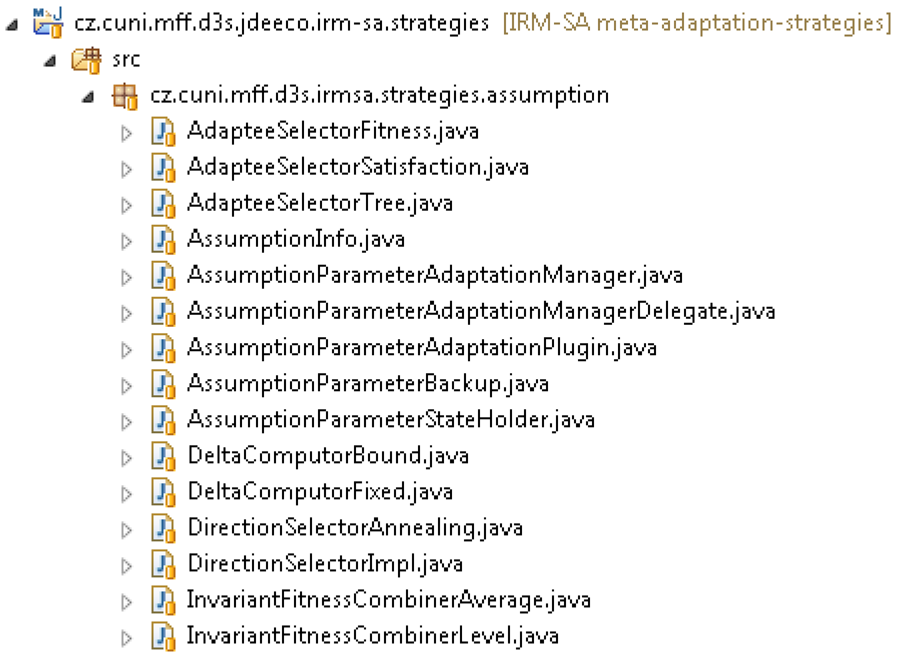
```
IRM design = (IRM)
        EMFHelper.loadModelFromXMI(DESIGN_MODEL_PATH);
IRMPlugin irmPlugin = new IRMPlugin(design).withLog(false);
MetaAdaptationPlugin metaAdaptationPlugin =
        new MetaAdaptationPlugin(irmPlugin);
RuntimeMetadata model = RuntimeMetadataFactoryExt.eINSTANCE
        .createRuntimeMetadata();
PeriodAdaptationPlugin periodAdaptionPlugin =
        new PeriodAdaptationPlugin(
            metaAdaptationPlugin, model, design, irmPlugin.getTrace())
        .withInvariantFitnessCombiner(
            new InvariantFitnessCombinerAverage())
        .withAdapteeSelector(new AdapteeSelectorFitness())
        .withDirectionSelector(new DirectionSelectorImpl())
        .withDeltaComputor(new DeltaComputorFixed(250))
        .withConsiderAssumptions(true)
        .withAdaptationBound(0.8)
        .withMaximumTries(3);
```

Figure 27: Code snipped illustrating the creation of PeriodAdaptationPlugin.

5.4.2. AssumptionParameterAdaptationPlugin

This plugin extends the EvolutionaryAdaptationPlugin and implements Assumption Parameters Adjusting meta-adaptation strategy. The structure of its package is illustrated in Figure 28.

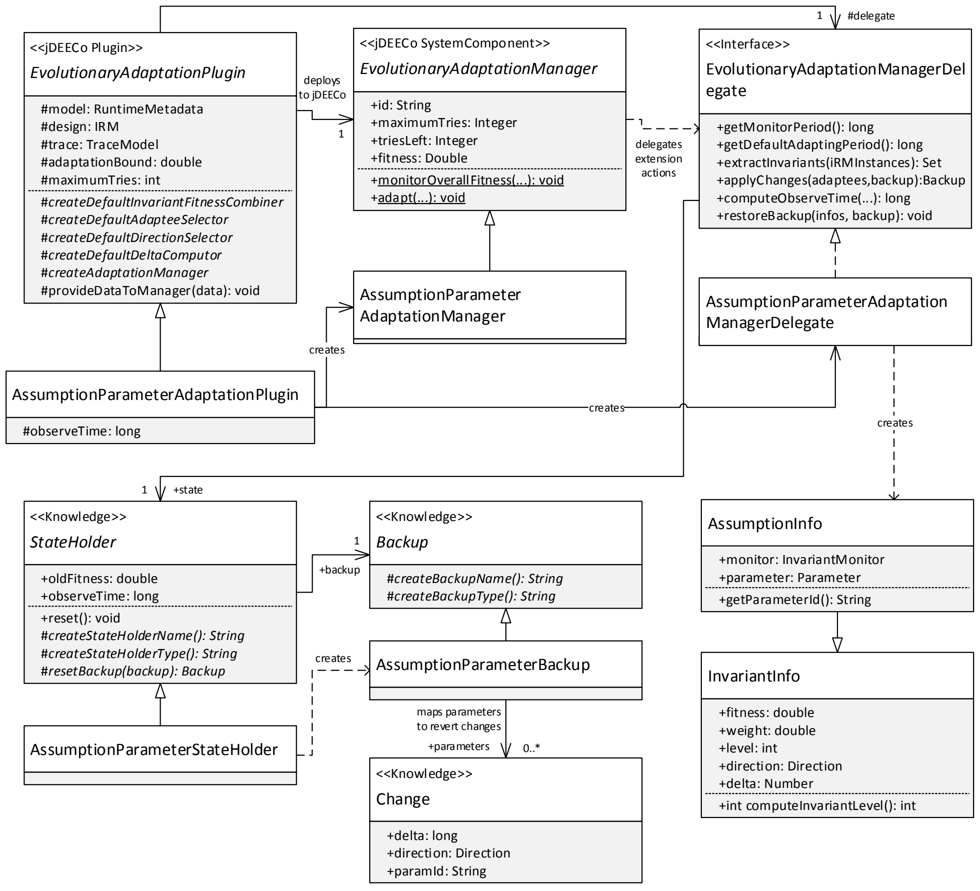
The code of the plugin and its delegate is very straightforward thanks to robust infrastructure provided by EvolutionaryAdaptationPlugin. The values of assumption parameters for satisfaction and fitness monitors are changes by setting appropriate values in components' internal data. Name convention is used to identify parameters' values by both AssumptionParameterAdaptationManagerDelegate and IRMInstaceGenerator which extracts the values of monitor methods' parameters and calls the monitors. AssumptionInfo extends InvariantInfo by containing the invariant monitor object and parameter. It is more finely grained because the adaptation affects individual parameters, not just assumptions as a whole. It also provides convenient method getParameterId() to get to parameter id which follows the naming convention mentioned above. The overview of the most important classes of the plugin can be seen in Figure 29.



```
cz.cuni.mff.d3s.jdeeco.irm-sa.strategies [IRM-SA meta-adaptation-strategies]
  src
    cz.cuni.mff.d3s.irmsa.strategies.assumption
      AdapteeSelectorFitness.java
      AdapteeSelectorSatisfaction.java
      AdapteeSelectorTree.java
      AssumptionInfo.java
      AssumptionParameterAdaptationManager.java
      AssumptionParameterAdaptationManagerDelegate.java
      AssumptionParameterAdaptationPlugin.java
      AssumptionParameterBackup.java
      AssumptionParameterStateHolder.java
      DeltaComputorBound.java
      DeltaComputorFixed.java
      DirectionSelectorAnnealing.java
      DirectionSelectorImpl.java
      InvariantFitnessCombinerAverage.java
      InvariantFitnessCombinerLevel.java
```

Figure 28: Package of AssumptionParameterAdaptationPlugin inside its project.

The implementation provides great variety of variation points. More detailed description of provided implementations of interfaces from package cz.cuni.mff.d3s.irmsa.strategies.commons.variations is presented in Section 5.4.1 as analogous implementations are prepared for this plugin.



<<jDEECo Plugin>>
EvolutionaryAdaptationPlugin
#model: RuntimeMetadata
#design: IRM
#trace: TraceModel
#adaptationBound: double
#maximumTries: int
#createDefaultInvariantFitnessCombiner
#createDefaultAdapteeSelector
#createDefaultDirectionSelector
#createDefaultDeltaComputor
#createAdaptationManager
#provideDataToManager(data): void
deploys to jDEECo
1
<<jDEECo SystemComponent>>
EvolutionaryAdaptationManager
+id: String
+maximumTries: Integer
+triesLeft: Integer
+fitness: Double
+monitorOverallFitness(...): void
+adapt(...): void
delegates extension actions
1
#delegate
<<Interface>>
EvolutionaryAdaptationManagerDelegate
+getMonitorPeriod(): long
+getDefaultAdaptingPeriod(): long
+extractInvariants(iRMInstances): Set
+applyChanges(adaptees,backup):Backup
+computeObserveTime(...): long
+restoreBackup(infos, backup): void
AssumptionParameter AdaptationManager
creates
AssumptionParameterAdaptation ManagerDelegate
AssumptionParameterAdaptationPlugin
#observeTime: long
creates
creates
1
+state
<<Knowledge>>
StateHolder
+oldFitness: double
+observeTime: long
+reset(): void
#createStateHolderName(): String
#createStateHolderType(): String
#resetBackup(backup): Backup
1
+backup
<<Knowledge>>
Backup
#createBackupName(): String
#createBackupType(): String
AssumptionInfo
+monitor: InvariantMonitor
+parameter: Parameter
+getParameterId(): String
creates
AssumptionParameterBackup
AssumptionParameterStateHolder
maps parameters to revert changes
+parameters
0..*
<<Knowledge>>
Change
+delta: long
+direction: Direction
+paramId: String
InvariantInfo
+fitness: double
+weight: double
+level: int
+direction: Direction
+delta: Number
+int computeInvariantLevel(): int

Figure 29: Overview of the most important classes of AssumptionParameterAdaptationPlugin.

The required user input at design time is as follows. Assumption monitors returning double fitness value must be provided along with existing IRM-SA satisfaction monitors returning boolean values. Both monitors use the "@Monitor" annotation. Their parameters must be marked with "@AssumptionParameter" annotation. The following parameter properties must be specified: name, default value, minimal and maximal values. The scope of the parameter is optional and defines whether the parameter value is shared among assumption monitors (COMPONENT) or is localized to this monitor only (MONITOR). The other optional property is initial direction for parameter adaptation, either UP or DOWN. Figure 30 contains code snipped illustrating the use of the annotation. When creating the plugin, the user specifies the variants of the algorithm to be used by passing implementations of corresponding interfaces to the plugin before adding it to jDEECo framework.

```
@Component
@IRMComponent("FireFighter")
public class FireFighter {
…
    @InvariantMonitor("A02")
    public static boolean positionAccuracySatisfaction(
        @AssumptionParameter(name = "bound", defaultValue = 1.5,
            maxValue = 1.9, minValue = 1.1,
            scope = Scope.COMPONENT, initialDirection = Direction.UP)
        double bound) {
        …
    }
…
}
```

Figure 30: Code snipped illustrating the use of AssumptionParameter.

```
IRM design = (IRM)
        EMFHelper.loadModelFromXMI(DESIGN_MODEL_PATH);
IRMPlugin irmPlugin = new IRMPlugin(design).withLog(false);
MetaAdaptationPlugin metaAdaptationPlugin =
        new MetaAdaptationPlugin(irmPlugin);
RuntimeMetadata model = RuntimeMetadataFactoryExt.eINSTANCE
        .createRuntimeMetadata();
AssumptionParameterAdaptationPlugin apap =
        new AssumptionParameterAdaptationPlugin (
            metaAdaptationPlugin, model, design, irmPlugin.getTrace())
        .withInvariantFitnessCombiner(
            new InvariantFitnessCombinerAverage())
        .withAdapteeSelector(new AdapteeSelectorFitness())
        .withDirectionSelector(new DirectionSelectorImpl())
        .withDeltaComputor(new DeltaComputorFixed(5))
        .withAdaptationBound(0.4)
        .withMaximumTries(3);
```

Figure 31: Code snipped illustrating the creation of AssumptionParameterAdaptationPlugin.

6. Experimental Evaluation

This chapter contains a description of the simplified case study that is used to evaluate the three proposed meta-adaptation strategies, their impact on the running system and a comparison with the scenario where no self-adaptation is employed at all.

6.1. Experiment Description

The experiment exploits meta-adaptation strategies implemented as extension of IRM-SA plugin for jDEECo as described in Chapter 5. In Section 6.1.1 there is an explanation of simplification of the original case study and Section 6.1.2 describes the specific scenario which has been used for gathering the data for the evaluation itself.

6.1.1. IRM-SA Model for the Case Study

The overall case study is very complex and multi-layered which could interfere with evaluation of the meta-adaptation approach. To mitigate this risk only part of the system is simulated to focus on the most essential areas the adaptation could significantly improve the behavior of the system in unanticipated situations and harsh circumstances.

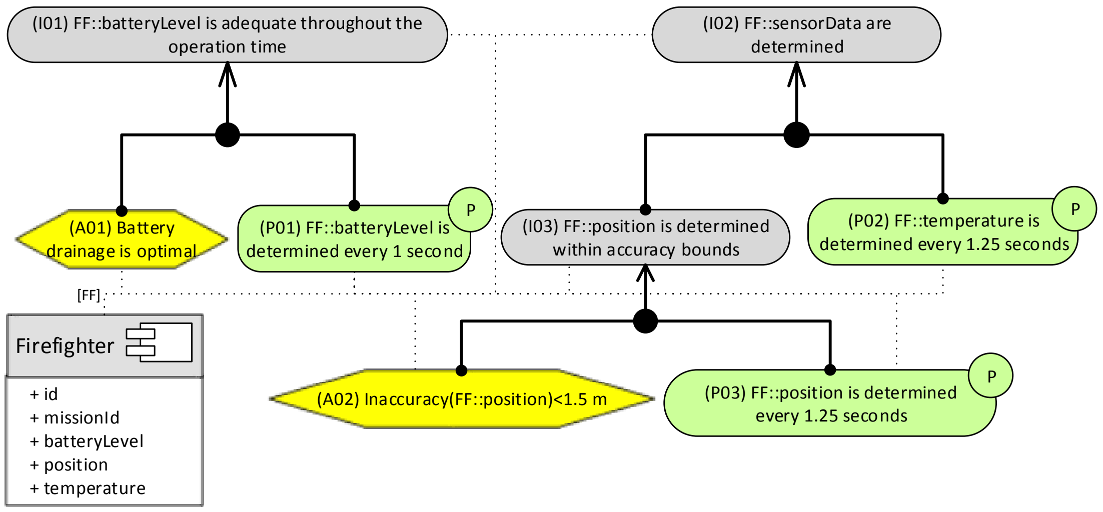

(I01) FF::batteryLevel is adequate throughout the operation time
(I02) FF::sensorData are determined
(A01) Battery drainage is optimal
(P01) FF::batteryLevel is determined every 1 second
P
(I03) FF::position is determined within accuracy bounds
(P02) FF::temperature is determined every 1.25 seconds
P
[FF]
Firefighter
+ id
+ missionId
+ batteryLevel
+ position
+ temperature
(A02) Inaccuracy(FF::position)<1.5 m
(P03) FF::position is determined every 1.25 seconds
P

Figure 32: simplified model of the case study for the evaluation

The main goal of the simplified system is gathering the information about firefighters' environment and equipment. The only component considered is the FireFighter as can be seen in Figure 32. The battery level, the temperature of environment and firefighter position are chosen to represent various important data that could be monitored to detect possible danger to the group members. There are two top level invariants:

1. [I01] Protective equipment of the firefighter has enough energy to stay operational during the predefined mission time
2. [I02] Data about firefighter's environment are collected

Both of them are further refined via AND-decomposition. The invariant [I01] has naturally sub-invariants related to battery, i.e. process invariant [P01] to determine battery level and assumption [A01] guaranteeing optimal energy drainage. The invariant [I02] is refined by process invariant [P02] determining environment temperature and inner invariant [I03] which is again further refined by finally leaf invariants, that is process invariant [P03] determining position by using particular technology and assumption [A02] ensuring the position inaccuracy is within predefined bounds.

The knowledge fields of the FireFighter component are wrapped inside MetadataWrapper to support certain types of adaptation described in more detail above in Section 5.3. PositionKnowledge is a type integrating two real values representing position in two-dimensional coordination system used by the simulation and a real number as accuracy of this position information provided by the position sensor. Battery level is integral for clarity's sake and temperature is stored as real value in degrees of Celsius.

There is no pre-designed ensemble in this simplified scenario as the FireFighter components do not need to communicate with each other to gather the required information about their environment because every unit is provided with all necessary measuring equipment.

The idea is to provide transparent scenario where different automatic meta-adaptation strategies could be evaluated, for results of this evaluation see Section 6.2. There are no pre-designed adaptations that would benefit from employing the IRM-SA to show example situation where imperfection inherently contained in complex and complicated CPS could be mitigate by general approach of meta-adaptation strategies.

6.1.2. Scenario description

The simplified case study model is used in the evaluation scenario where there is a building on fire that is being explored by three firefighters (FF1, FF2 and FF3) whose map is depicted in Figure 33. Firefighter FF3 is moving on their own, while firefighters FF1 and FF2 are moving together in a group. Every firefighter is modeled as s DEECo component that gathers information about its battery level, position and environment temperature. The IRM-SA model is shown in Figure 32.

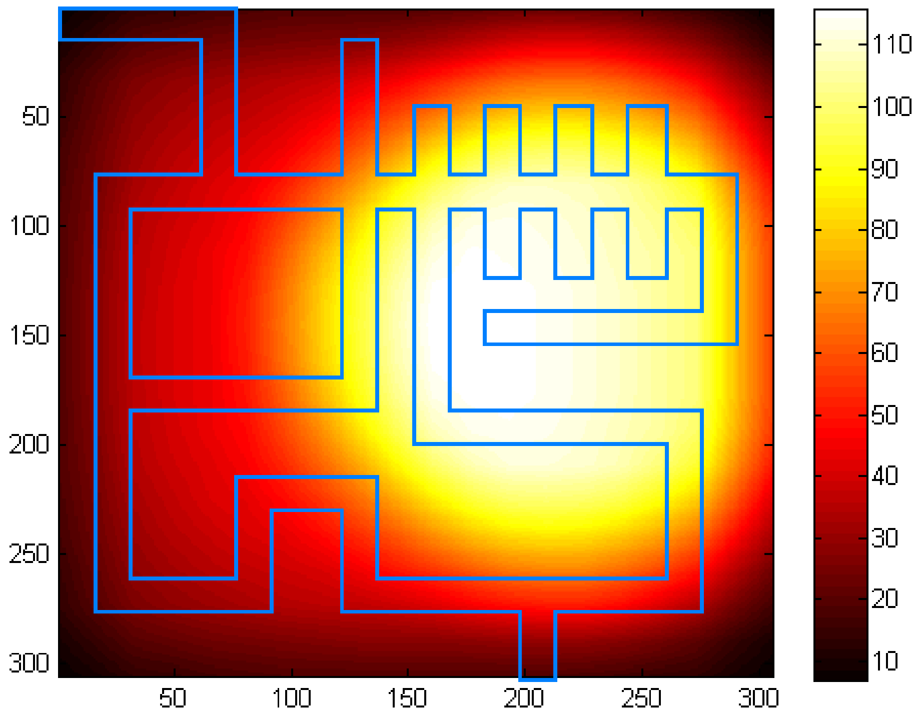


Figure 33: Heat and corridor map used in the simulation: blue lines marks the corridors in the building, background color depicts environment temperature

IRM-SA is responsible for adaptation of the system. If there is no applicable configuration that can be used to adapt to the current situation, the IRM-SA notifies the meta-adaptation manager described in Section 5.2. Then the individual meta-adaptation strategies are employed to adapt the system for unexpected circumstances.

To simulate such unforeseen conditions, two artificial malfunctions are introduced to the scenario:

The first malfunction occurs after 50 seconds – FF1's position tracking system begins to provide inaccurate readings, so inaccuracy of position of FF1 oversteps the pre-designed boundary (1.5 m). Meta-adaptation is started because IRM-SA is not able to discover application system configuration. The fact that the sensor is still operation, only more inaccurate, prevents the use of Data Classification strategy (DC, see Section 4.1). The situation can be though saved by both Process Period Adjusting strategy (PPA, see Section 4.2) and Assumption Parameters Adjusting strategy. (APA, see Section 4.3). Invariant P03 “FF::position is determined every 1.25 seconds” is chosen by PPA as perspective candidate for adaptation and the corresponding process’s period is lowered for FF1. This decreases inaccuracy of the position between two consecutive position determinations which in turn to some extent alleviate problems with increased inaccuracy of the position sensor.

Due to the pre-designed lower bound for the position process (250 ms), PPA strategy is not enough to fully recover the system, i.e. lower the position inaccuracy of FF1 back under the acceptable threshold (1.5 m). Fortunately, APA is prepared to deal with such situations. Assumption A02 is chosen as a perspective candidate for adaptation and its parameter is relaxed to allow higher position inaccuracy, but still within pre-designed bounds. The system recovers and the IRM-SA can again find a satisfiable configuration.

The second malfunction occurs after 150 seconds – FF1’s temperature sensor breaks completely. The IRM-SA again cannot find an applicable configuration and triggers meta-adaptation via meta-adaptation manager. Changing process periods by PA has no impact. APA is not able to improve the situation because the sensor provides no readings whatsoever. However, DC finds the correlation between the distance between firefighters and environmental temperature their sensors measure. The closer the firefighters stand, the more similar the environmental temperatures are. A new ensemble created by DC at runtime is deployed to the system. This ensemble injects the temperature readings of other firefighters to firefighter FF1 if they are close enough. Usually FF2’s temperature is injected to FF1’s field because these two firefighters move as a group. Yet, when FF1 is closer to FF3 than he is to FF2, FF3’s temperature is injected instead.

This scenario combining two different malfunctions provides opportunities for all suggested strategies to improve the system performance while proving that cooperation is achievable for both very different and very similar strategies which might not be the case if more specific and specialized examples for each strategy are evaluated. The results of the evaluation can be found in the next section.

6.2. Experiment Evaluation

In this section, the results of the evaluation of the experimental scenario can be found including figures comparing self-adaptation approach of meta-adaption strategies with control run with no adaptation at all.

Figure 34 shows the differences between the two approaches after the first malfunction, i.e. position sensor provides readings with higher inaccuracy after 50th second of the experiment. The differences between the FF1's belief of his position and his actual position are depicted by the box plots. The common period [0, 50]s, i.e. before the malfunction occurs, is the same for both meta-adaption approach and control sample and is displayed by the left box. The rest of the boxes depicts period (50, 300]s, the middle box shows data from the simulation with meta-adaptation and finally the right box corresponds to the control simulation without any adaptations. The increased inaccuracy of position sensor obviously raises the difference between the belief and the actual values as depicted by the last box. Figure 34 also captures the assumed limits on the position inaccuracy – the original limit is represented by the horizontal dashed line; the limit relaxed by APA strategy is represented by the horizontal dotted-dashed line.

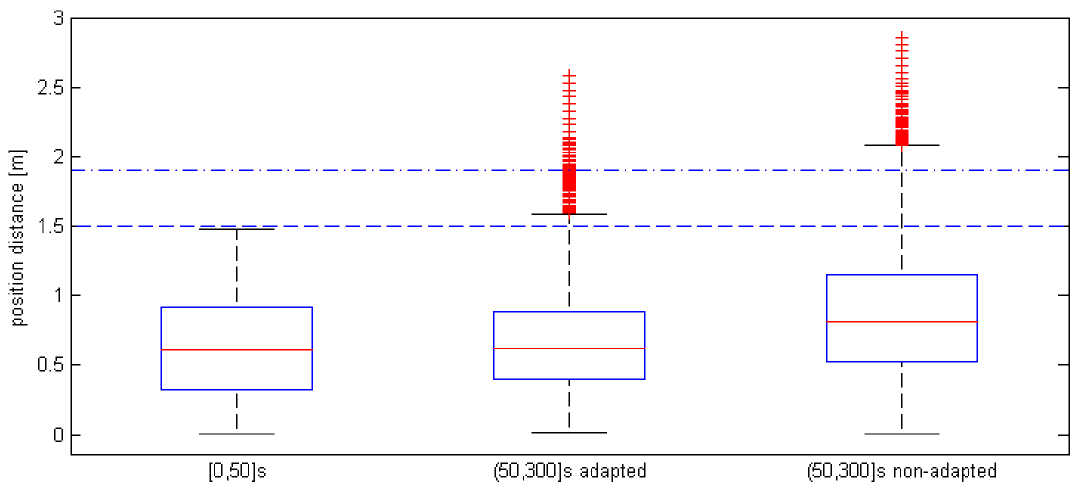


Figure 34: The Euclidean difference between the actual position and belief about the position. The first box depicts situation before malfunction, the second one illustrates results of the meta-adaptation strategies and the third one is control sample without any adaptations.

Figure 35 concentrates on the situation after the 150th second of the experiment, i.e. after the second malfunction prevents FF1's temperature to operate completely. The actual environmental temperature is represented by the red line. The component's belief about the temperature in simulation with meta-adaptation is represented by the blue line. And finally the component's belief about the temperature in simulation without adaptation is represented by a green line. As can be seen, the belief is rather accurate before the malfunction occurs because there is only interference caused by firefighter's movement between measurements and by random noise. After the malfunction, the belief that is not adapted is no longer usable as it is not updated at all and temperature field can no longer be relied on. On the other hand, the meta-adaption causes the belief to be updated by injecting readings from nearby firefighters which provides useful belief in spite of some delay and inaccuracy.

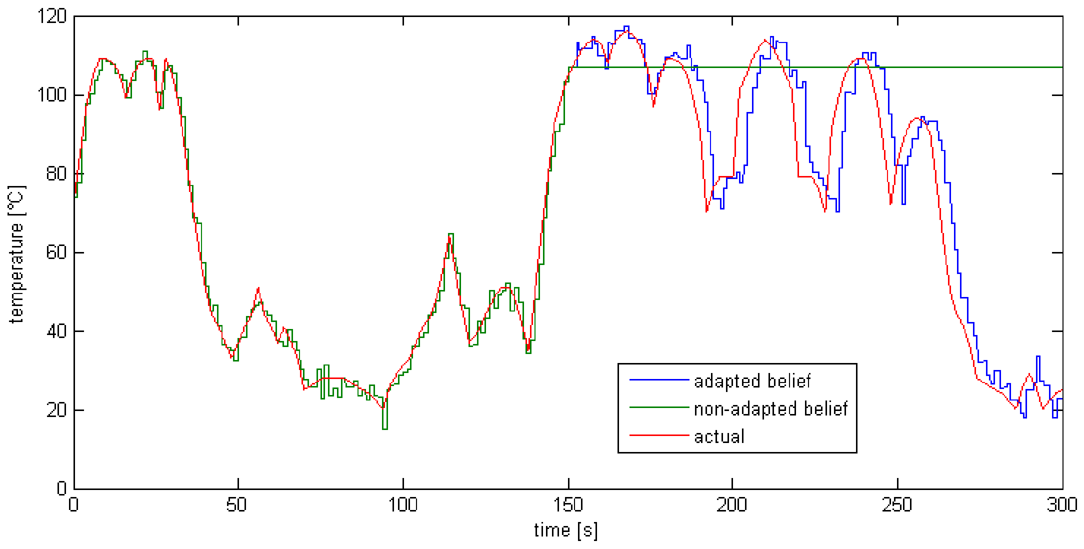


Figure 35: The evolution of the actual temperature and belief about the belief temperature.

7. Related Work

The self-adaptation belongs among thriving research subjects of the software engineering [29], [30], [31]. The main research areas dealing with it are (i) modeling and model-driven engineering [32], [33], [34], (ii) control theory [35], [36], [37] and (iii) software architecture [3], [5], [38], [39].

The MAS elaborated in this thesis could be applied to model many diverse approaches from (i)-(iii) that push the borders of adaptability of a system, in spite the fact that it is designed primarily with architecture-based self-adaptation in mind. For instance, a model-driven approach is likely to be easily converted to a MAS strategy, should it be able to generate new behavior models and to pick from them in the running system. One of the promising approaches is AVIDA-MDE [34]. In AVIDA-MDE, a MAS tactic takes a form of a behavior model of the system (state diagram in UML) and a MAS strategy corresponds to generating behavioral models by a digital evolution-based approach exploiting an evolutionary computation platform [40]. The definition of a MAS metric consists of both the latent functional properties exhibited by the generated models, and the non-functional characteristics. Despite the fact that offline approach is employed in AVIDA-MDE, the automatic generation of tactics is its important feature that MAS counts on. Other sources of inspiration for MAS are various methods for synthetizing computationally diverse program variants [41].

The inspiration for Process Period Adjusting and Assumption Parameter Adjusting strategies is a method using evolutionary computation to adapt test cases [42] introduced not long time ago called Veritas. Its primary concept lies in application of (1+1)-ONLINE algorithm generating new test cases. At runtime, less false negatives are produced by the new test cases; that is the test cases correspond better to the current changes in the self-adaptive system behavior. A MAS tactic takes form of a test case; iterative application of the evolutionary algorithm that generates new test cases corresponds to a MAS strategy. Fitness functions measure the overall system fitness which is similar to Process Period Adjusting and Assumption Parameters Adjusting strategies. The MAS metric of this strategy can take the form of these fitness functions. Proteus [43] is a framework incorporating Veritas that addresses assurance when confronted with uncertainty in the running system. In order to enhance self-adaptative systems with runtime compliance checking [44], component-based integration testing is also encouraged along with online adaptive testing described above.

There are many different levels where the adaptability of a system can be improved to cope with uncertainty. FLAGS [45], [46] and Evolution Requirements [47], [48] are examples of approaches originating from requirements engineering community that come with important ideas.

The main concept of FLAGS lies in “adaptation requirements”. They describe requirements on the counteractions applied when application requirements fail. KAOS [49] object models, operation and goals are exploited in FLAGS. When satisfaction criteria of a “conventional” goal are not met, the system takes a countermeasure which corresponds to an *adaptation goal,* which is a special goal type. FLAGS introduces a fuzzy goal whose satisfaction is the result of a fuzzy membership function. That allows modeling of the satisfaction criteria for conventional goals. RELAX language [50] is exploited [51] for formal specification of the fuzzy goals. Adaptation goals in the running system may be triggered based on the level of goals’ satisfaction. This starts countermeasures including altering the operations’ pre- and post-conditions, changing the goals’ membership functions or adding or removing objects, operations or goals [46].

Requirements causing the evolution of other requirements are the main concept of Evolution Requirements (EvoReqs). On one hand, traditional requirements are modeled as goals (i.e. EvoReqs is model-based), on the other hand the evolutional requirements take the form of event-condition-action rules whose events serve as a guard condition for the goals. When the satisfaction of a requirement is no longer possible, the requirement is altered, e.g. the requirement are relaxed, retried later, delegated to a human actor or a system task takes place of a domain assumption [48].

MAS differs from FLAGS, EvoReqs and similar approaches (e.g. [52]) by (i) focusing mainly on runtime behavior of the system, while their focus lies mainly in requirements specification, and (ii) the fact the unanticipated circumstances cannot be coped with by these approaches because every situation and corresponding tactic (plan or task) must be foreseen in advanced to model them, i.e. they do not define how design and runtime flexibility should be achieved, only provide means to it.

8. Conclusion & Discussion

Meta-adaptation strategies elaborated in this thesis enhance IRM-SA capabilities to deal with unanticipated situations at runtime by creating new tactics dealing with dynamically changing environment. This extends the adaptation envelope of the system and provides self-healing mechanisms to put in use when unforeseen circumstances jeopardizing the functionality of the system are encountered. A mechanism for management and activation of different meta-adaptation strategies is introduced to provide common means for easy implementation of various approaches. These implementations cover a large spectrum of meta-adaptation strategies, including not only the ones elaborated in the thesis, but also many others that are out of the scope of this thesis.

The proposed meta-adaptation strategies are described in detail in Chapter 4. IRM-SA has been altered to provide interface to plug in various extensions listening to the results of IRM-SA self-adaptation covering goal **G1**. This interface is exploited by MetaAdaptationManager (Section 5.2) that serves as a controller managing individual implementations of meta-adaptation strategies which are documented thoroughly in Chapter 5, as required by goal **G2**. The experimental scenario based on the firefighter coordination case study to evaluate the implementations of meta-adaptation strategies has been prepared and the promising results are presented in Chapter 6, satisfying goal **G3**.

8.1. Improvements of the Current Implementation

There are several ways to improve the current implementation of the meta-adaptation strategies and to remove their current limitations. Some of them are elaborated in this section.

First, the simplistic implementation of the MetaAdaptationManager could be extended to choose the meta-adaptations to run according to a more sophisticated algorithm. For this reason, interface AdaptationManager should be extended. One possibility is to provide some kind of rich communication protocol providing information about the suitable situations to deploy this meta-adaptation to the MetaAdaptationManager and let it to compare it with the present circumstances and decide if the strategy is really to be employed. In such a case, the strategy would provide preconditions that must be satisfied for the meaningful execution of the strategy. The other option is to add only one method returning a boolean value indicating whether the strategy is useful in the current situation or not, so the responsibility for monitoring the current state of the system lies on meta-adaptation strategies themselves and not their manager.

The individual implementations of meta-adaptation strategies could also be improved. Process Period Adjusting strategy may be expanded to adapt also ensemble periods, not only process periods. However, ensemble scheduling periods are not available in the current jDEECo API, so this improvement would need coordination with the main jDEECo project development.

Assumption Parameter Adjusting strategy has also room for improvement. The current implementation is limited to adjust parameters of assumptions that are related to the knowledge of one component only. The workaround consisting of auxiliary ensemble (gathering knowledge needed) and component (storing knowledge and hosting the assumption monitor) is obviously clumsy. Adjusting parameters of assumptions dealing with knowledge of multiple components natively is a natural extension of the current implementation, however it must be thought well because it may easily introduce communication between components bypassing the standard jDEECo communication model. A place inside jDEECo framework for defining such assumption monitors and auxiliary data must be defined, too.

8.2. Possible Extensions

Implementations of other meta-adaptation strategies can be seen as possible extensions of the work presented in this thesis. The following meta-adaptation strategy is an example that could extend the self-adaptation capabilities of the systems even more and that would nicely fit into the mechanisms introduced in this thesis.

Consider a scenario where multiple sensors measure a physical phenomenon, for example temperature. These components have role *TemperatureProvider*. At some point, component *C* (one of the sensors) malfunctions and starts emitting temperatures that are not at all close to the values provided by the rest of the components. A meta-adaptation strategy based also on data correlation could discover that this phenomenon is taking place and remove the role

TemperatureProvider from component *C* so that unusable data are not spread in the system.

Bibliography

[1] MODELS 2014. Call for ACM Student Research Competition. [Online]. Available: http://models2014.webs.upv.es/acmsrc.htm

[2] European Union Horizon 2020, Smart Cyber-Physical Systems, ICT-01-2014. [Online]. Available: http://ec.europa.eu/research/participants/portal/desktop/en/-opportunities/h2020/topics/78-ict-01-2014.html

[3] S.-W. Cheng and D. Garlan, “Stitch: A Language for Architecture-based Self-adaptation,” *J. Syst. Softw.*, vol. 85, no. 12, pp. 2860–2875, Dec. 2012.

[4] P.-C. David, T. Ledoux, M. Léger, and T. Coupaye, “FPath and FScript: Language support for navigation and reliable reconfiguration of Fractal architectures,” *annals of telecommunications-annales des télécommunications*, vol. 64, no. 1-2, pp. 45–63, 2009.

[5] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, and P. Steenkiste, “Rainbow: Architecture-based self-adaptation with reusable infrastructure,” *Computer*, vol. 37, no. 10, pp. 46–54, 2004.

[6] D. Hirsch, J. Kramer, J. Magee, and S. Uchitel, “Modes for software architectures,” in *Software Architecture*. Springer, 2006, pp. 113–126.

[7] T. Batista, A. Joolia, and G. Coulson, “Managing dynamic reconfiguration in component-based systems,” in *Software Architecture*. Springer, 2005, pp. 1–17.

[8] T. Bures, I. Gerostathopoulos, P. Hnetynka, J. Keznikl, M. Kit, F. Plasil, and N. Plouzeau, “Adaptation in Cyber-Physical Systems: from System Goals to Architecture Configurations,” Department of Distributed and Dependable Systems, Charles University in Prague, Tech. Rep. D3S-TR-2014-01, March 2014.

[9] J. Keznikl, T. Bures, F. Plasil, I. Gerostathopoulos, P. Hnetynka, and N. Hoch, “Design of ensemble-based component systems by invariant refinement,” in *Proceedings of the 16th International ACM Sigsoft symposium on Component-based software engineering*. ACM, 2013, pp. 91–100.

[10] I. Gerostathopoulos, T. Bures, P. Hnetynka, A. Hujecek, F. Plasil, and D. Skoda, “Meta-Adaptation Strategies for Adaptation in Cyber-Physical Systems,” Technical Report, Tech. Rep., 2015.

[11] T. Bures, I. Gerostathopoulos, P. Hnetynka, J. Keznikl, M. Kit, and F. Plasil, “DEECO: an Ensemble-Based Component System,” in *Proceedings of the 16th International ACM Sigsoft symposium on Component-based software engineering*. ACM, 2013, pp. 81–90.

[12] R. Al Ali, T. Bures, I. Gerostathopoulos, P. Hnetynka, J. Keznikl, M. Kit, and F. Plasil, “DEECo: an Ecosystem for Cyber-Physical Systems,” in *Companion Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 610–611.

[13] "jDEECo Website", 2015. [Online]. Available: https://github.com/d3scomp/JDEECo

[14] I. Gerostathopoulos, T. Bures, P. Hnetynka, J. Keznikl, M. Kit, F. Plasil, and N. Plouzeau, “Self-Adaptation in Cyber-Physical Systems: from System Goals to Architecture Configurations,” 2015.

[15] I. Crnkovic and M. Larsson, “Building Reliable Component-Based Software Systems, Artech House,” *Inc., Norwood, MA*, 2002.

[16] I. Crnkovic, M. Chaudron, and S. Larsson, “Component-based development process and component lifecycle,” in *Software Engineering Advances, International Conference on*. IEEE, 2006, pp. 44–44.

[17] N. R. Jennings, “On agent-based software engineering,” *Artificial intelligence*, vol. 117, no. 2, pp. 277–296, 2000.

[18] Y. Shoham and K. Leyton-Brown, *Multiagent systems: Algorithmic, game-theoretic, and logical foundations*. Cambridge University Press, 2009.

[19] M. Hölzl, A. Rauschmayer, and M. Wirsing, “Engineering of software-intensive systems: State of the art and research challenges,” in *Software-Intensive Systems and New Computing Paradigms*. Springer, 2008, pp. 1–44.

[20] M. Hölzl *et al.*, “Engineering Ensembles: A White Paper of the ASCENS Project,” *ASCENS Deliverable JD1*, vol. 1, 2011.

[21] A. J. Ramirez and B. H. Cheng, “Design patterns for developing dynamically adaptive systems,” in *Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*. ACM, 2010, pp. 49–58.

[22] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*. Pearson Education, 1994.

[23] N. Bredeche, E. Haasdijk, and A. Eiben, “On-line, on-board evolution of robot controllers,” in *Artifical Evolution*. Springer, 2010, pp. 110–121.

[24] "Meta-adaptation Strategies Website". [Online]. Available: https://github.com/-d3scomp/IRM-SA/tree/meta-adaptation-strategies

[25] "IRM-SA Website". [Online]. Available: https://github.com/d3scomp/IRM-SA

[26] "Eclipse Website". [Online]. Available: https://eclipse.org/

[27] "Javassist Website"". [Online]. Available: http://jboss-javassist.github.io/javassist/

[28] S. Kirkpatrick, C. D. Gelatt, M. P. Vecchi *et al.*, “Optimization by simulated annealing,” *science*, vol. 220, no. 4598, pp. 671–680, 1983.

[29] G. Di Marzo Serugendo, B. Cheng, R. De Lemos, H. Giese, P. Inverardi, J. Magge, J. Andersson, B. Becker, N. Bencomo, Y. Brun *et al.*, “Software Engineering for Self-Adaptive Systems: A Research Roadmap,” 2009.

[30] R. De Lemos, H. Giese, H. A. Müller, M. Shaw, J. Andersson, M. Litoiu, B. Schmerl, G. Tamura, N. M. Villegas, T. Vogel *et al.*, “Software engineering for self-adaptive systems: A second research roadmap,” in *Software Engineering for Self-Adaptive Systems II*. Springer, 2013, pp. 1–32.

[31] M. Salehie and L. Tahvildari, “Self-adaptive software: Landscape and research challenges,” *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, vol. 4, no. 2, p. 14, 2009.

[32] M. Morandini, L. Penserini, and A. Perini, “Towards goal-oriented development of self-adaptive systems,” in *Proceedings of the 2008 international workshop on Software engineering for adaptive and self-managing systems*. ACM, 2008, pp. 9–16.

[33] J. Zhang and B. H. Cheng, “Model-based development of dynamically adaptive software,” in *Proceedings of the 28th international conference on Software engineering*. ACM, 2006, pp. 371–380.

[34] H. J. Goldsby and B. H. Cheng, “Automatically generating behavioral models of adaptive systems to address uncertainty,” in *Model Driven Engineering Languages and Systems*. Springer, 2008, pp. 568–583.

[35] A. Filieri, C. Ghezzi, A. Leva, and M. Maggio, “Self-adaptive software meets control theory: A preliminary approach supporting reliability requirements,” in *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, 2011, pp. 283–292.

[36] T. Patikirikorala, A. Colman, J. Han, and L. Wang, “A systematic survey on the design of self-adaptive software systems using control engineering approaches,” in *Proceedings of the 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. IEEE Press, 2012, pp. 33–42.

[37] A. Filieri, H. Hoffmann, and M. Maggio, “Automated design of self-adaptive software with control-theoretical formal guarantees,” in *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 299–310.

[38] D. Sykes, W. Heaven, J. Magee, and J. Kramer, “From goals to components: a combined approach to self-management,” in *Proceedings of the 2008 international workshop on Software engineering for adaptive and self-managing systems*. ACM, 2008, pp. 1–8.

[39] J. Kramer and J. Magee, “A rigorous architectural approach to adaptive software engineering,” *Journal of Computer Science and Technology*, vol. 24, no. 2, pp. 183–188, 2009.

[40] C. Ofria and C. O. Wilke, “Avida: A software platform for research in computational evolutionary biology,” *Artificial life*, vol. 10, no. 2, pp. 191–229, 2004.

[41] B. Baudry, M. Monperrus, C. Mony, F. Chauvel, F. Fleurey, and S. Clarke, “DIVERSIFY: Ecology-inspired software evolution for diversity emergence,” in *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week-IEEE Conference on*. IEEE, 2014, pp. 395–398.

[42] E. M. Fredericks, B. DeVries, and B. H. Cheng, “Towards run-time adaptation of test cases for self-adaptive systems in the face of uncertainty,” in *Proceedings of the*

9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems. ACM, 2014, pp. 17–26.

[43] E. M. Fredericks and B. H. Cheng, "Automated generation of adaptive test plans for self-adaptive systems," in *Appear in Proceedings of 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, ser. SEAMS*, vol. 15, 2015.

[44] C. E. da Silva and R. de Lemos, "Dynamic plans for integration testing of self-adaptive software systems," in *Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. ACM, 2011, pp. 148–157.

[45] L. Baresi and L. Pasquale, "Live goals for adaptive service compositions," in *Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*. ACM, 2010, pp. 114–123.

[46] L. Baresi, L. Pasquale, and P. Spoletini, "Fuzzy goals for requirements-driven adaptation," in *Requirements Engineering Conference (RE), 2010 18th IEEE International*. IEEE, 2010, pp. 125–134.

[47] V. E. S. Souza, A. Lapouchnian, and J. Mylopoulos, "(Requirement) evolution requirements for adaptive systems," in *Proceedings of the 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. IEEE Press, 2012, pp. 155–164.

[48] V. E. S. Souza, A. Lapouchnian, K. Angelopoulos, and J. Mylopoulos, "Requirements-driven software evolution," *Computer Science-Research and Development*, vol. 28, no. 4, pp. 311–329, 2013.

[49] A. van Lamsweerde, "Requirements engineering: from craft to discipline," in *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*. ACM, 2008, pp. 238–249.

[50] J. Whittle, P. Sawyer, N. Bencomo, B. H. Cheng, and J.-M. Bruel, "RELAX: a language to address uncertainty in self-adaptive systems requirement," *Requirements Engineering*, vol. 15, no. 2, pp. 177–196, 2010.

[51] L. Baresi and L. Pasquale, "Adaptation goals for adaptive service-oriented architectures," in *Relating Software Requirements and Architectures*. Springer, 2011, pp. 161–181.

[52] B. H. Cheng, P. Sawyer, N. Bencomo, and J. Whittle, "A goal-based modeling approach to develop requirements of an adaptive system with environmental uncertainty," in *Model Driven Engineering Languages and Systems*. Springer, 2009, pp. 468–483.

List of Abbreviations

APA – Assumption Parameters Adjusting strategy

API – Application Programming Interface

CPS – Cyber-Physical Systems

CPU – Computer Processing Unit

DC – Data Classification strategy

DEECo – Dependable Emergent Ensemble of Components

DSL – Domain-Specific Language

ECBS – Ensemble-Based Component Systems

FF – Fire Fighter

GM – Group Member

GPS – Global Position System

ICT – Information and Communication Technologies

IRM – Invariant Refinement Method

IRM-SA – Invariant Refinement Method for Self Adaptation

jDEECo – java implementation of DEECo

MAS – Meta-Adaptation Strategies

MDP – Markov Decision Processes

NaN – Not a Number

PPA – Process Period Adjusting strategy

SAT – Boolean Satisfiability Problem

Attachments

The CD attached to this thesis contains the following:

- thesis.pdf
    - electronic version of the thesis
- workspace
    - Eclipse workspace containing the sources related to the thesis and dependencies needed to run the experimental scenario
- thesis
    - directory containing the thesis sources and figures