Towards Goal-Oriented Development of Self-Adaptive Systems

Mirko Morandini
FBK-IRST
Via Sommarive 18, 38050 Povo, Trento, Italy
morandini@fbk.eu

Loris Penserini
Utrecht University
Padualaan 14, De Uithof, 3584 Utrecht, The Netherlands
loris@cs.uu.nl

Anna Perini
FBK-IRST
Via Sommarive 18, 38050 Povo, Trento, Italy
perini@fbk.eu

ABSTRACT

Self-adaptive software aims at anticipating changes which occur in a complex environment and to automatically deal with them at run-time. The increasing demand for complex networked software, which makes computing resources available to anyone, anywhere and at any time, is drawing attention to the engineering of self-adaptive software. The objective of our work is to define a process and a tool-supported design framework to develop self-adaptive systems, which consider Belief-Desire-Intention agent models as reference architectures. We adopt an agent-oriented approach, which allows to explicitly model system goals in requirements specification and in the system architecture design. Moreover, goal achievement conditions are specified along with their relationships with the environment and with possible failures, and corresponding recovery actions. This paper aims at motivating and giving an overview of our approach with the help of an example.

Categories and Subject Descriptors
D.2 [Software Engineering]: Design—Methodologies; I.2 [Artificial Intelligence]: Distributed Artificial Intelligence—Intelligent agents

General Terms
Design, Languages

Keywords
Agent-oriented software engineering, software development process, BDI agents, adaptive systems

1. INTRODUCTION

Today’s complex, distributed software systems are expected to be able to autonomously work in an unpredictable and changing environment, avoiding failure and achieving good performance. Self-adaptive systems try to cope with these challenging issues, adapting autonomously to a changing environment to fulfil the objectives of their stakeholders [14, 10]. These systems are supposed to automatically change and improve their own behaviour, basing on the domain information available to them and on the knowledge of system internals.

In this work we adopt the following definition of self-adaptive systems [6]: self-adaptive systems automatically take the correct actions based on the knowledge of what is happening in the system, guided by objectives and needs of stakeholders.

Self-adaptive systems are characterized by three core functionalities:

- monitor (sensing) the environment to recognize “problems”;
- take decisions on which behaviour to exhibit;
- realize the behaviour change by adaptation.

Therefore, to act adaptively, an self-adaptive system, which ideally bases on some specialized architecture to support previously mentioned functionalities, needs knowledge on:

- what to monitor and for which symptoms;
- which alternative behaviours are available;
- decision criteria for the selection of a specific behaviour.

Developing self-adaptive systems poses challenging research issues within the areas of requirements and software engineering. Several requirements analysis and software programming approaches consider a process guided by stakeholders’ needs and objectives, providing human-oriented abstractions such as goals and agents to manage the development of these complex systems. Berry et al. [1] identifies four levels at which requirements engineering for dynamic adaptive systems should be performed, which involve different actors with specific competences, including software designers, the system itself, software engineers and specialist in decision-making system and adaptation techniques. These different levels require appropriate methods to specify and analyse the requirements.

Goals and agents have been proposed as the best suited intentional concepts to catch the properties of complex, dynamic and adaptive systems in software requirements elicitation and analysis. Agent-oriented software engineering (AOSE) methodologies [8], such as MaSE, Prometheus and Tropos, use goal models to capture these stakeholders’ objectives the system under consideration should achieve [23].

For the definition of the system behaviour at run-time, the belief-desire-intention (BDI) model [22] provides an architectural basis.
for agent-oriented systems, endowing them with a reasoning cycle and the concepts of belief, desire (or goal) and plan, and was adopted by several agent-oriented implementation languages and execution platforms. Languages following BDI-architectural abstractions seem to be promising, in particular if they base on well-known languages and allow for imperative programming. Special attention has to be given to the abstractions proposed by the BDI model. They are belief, desire and intention in the original theory by Bratman and developed to different sets of concepts, such as belief, goal and plan [22] and also adding concepts such as events and communication messages.

Our aim is to support software engineers in developing self-adaptive systems. The goals assigned to the system, the environment where the system is situated, and the demand to recover from errors to avoid failures in goal achievement, are crucial aspects to be understood and specified for this kind of systems. As first step towards this objective, we aim at supporting the designer to capture, specify and detail characteristics specific to self-adaptive systems.

More concretely, the objective of our work is to define a process and a tool-supported development framework to develop self-adaptive systems, which consider Belief-Desire-Intention agent models as reference architectures. We adopt the Tropos methodology [3], which provides an agent-oriented approach for the early analysis phases as well as for the design of software, allowing to model explicitly system’s goals in requirements specification and in the system architecture design. We extend it to support for the detailed specification of goals, in relationship with the environment the system will be deployed in, and taking into account possible failures and corresponding recovery actions.

This will result in a development framework in which goals, failures and the environment are treated as first-class abstractions through all analysis and design phases, and give raise to an implementation, which endows the same abstractions.

This paper aims at motivating and illustrating our approach with the help of an example and is structured as follows: Section 2 recalls basic notions of the Tropos agent-oriented methodology, introduces the garbage cleaner example and points out a couple of scenarios in which system self-adaptation is required. Section 3 briefly discusses related work on engineering self-adaptive systems. Section 4 illustrates the proposed analysis and design framework, with the help of the example. Section 5 sketches the transition stage from design to implementation. Finally, Section 6 points out main future work directions.

2. BACKGROUND

In this section we recall key concepts used as basis for our approach, namely goal models in the Tropos methodology. Moreover, we introduce the the garbage cleaner example and point out a couple of scenarios in which self-adaptation is required.

2.1 Methodological Framework: Tropos

The Tropos agent-oriented methodology [3] borrows modelling and analysis techniques from goal-oriented requirements engineering frameworks and integrates them into an agent-oriented paradigm.

In Tropos, the development process is organized into five phases, namely: Early Requirements, whose objective is to produce a model of the environment (i.e. the organizational settings); Late Requirements, in which the system-to-be is introduced in the domain and its impact within the environment is analysed; Architectural Design, whose objective is to obtain a representation of the internal architecture of the system in terms of subcomponents of the system and relationships among them; Detailed Design, which is concerned with the definition of software agents rationale, including capabilities and interaction specifications; Implementation, whose objective is the production of code from the detailed design specification, according to an mapping established between the implementation platform constructs and the detailed design elements [16].

A core activity along this process is conceptual modelling. The modelling language offers concepts such as those of actor, goal, plan, resource, capability, and of social dependency between actors for goal achievement, and a graphical notation to depict views of a model, in terms of actor and goal diagrams. The methodology provides also model analysis techniques and supporting tools [20].

Adopting Tropos into our framework allows to represent and reason on goal models resulting from the analysis of the actor’s point of view. A goal model in Tropos (an example is depicted in Figure 1) is represented in terms of a forest of trees of AND/OR-decomposed goals, along with lateral contributions and dependencies to other actors. Additionally a goal model contains means-ends relationships among plans and goals, to define the means to satisfy a goal.

Starting from a goal-oriented requirements analysis, where goal models are built, the notions of actor, goal, plan and dependency are used also to define the system architecture, where the high-level goals of the system are delegated to specialized sub-actors, refined, and operationalised defining the activities to carry out. In the detailed design phase, the activities are detailed using UML diagrams and defining interaction protocols. The sub-actors will be the building blocks of a multi-agent system in the implementation phase. The behaviour of the agents in this MAS is defined by the variability in the goal models, selecting and executing appropriate activities at run-time. Consequently, the concept of goal is preserved in all development phases, supporting the development of BDI agents and traceability of design choices.

2.2 Motivating Example

To illustrate our development framework, we refer to a simple garbage cleaner agent scenario, which can be found in several variations, in artificial intelligence and multi-agent system fields. We focus on capturing adaptive properties of a system at design time. In particular, with this example we will focus on basic adaptivity requirements rising from dynamic changes in the environment and in the system’s internal state. Adaptivity requirements derived from changes of user needs have been partially addressed in previous works [13, 18].

The system in its simplest form contains only one agent, the CleanerAgent, situated in an environment, which changes dynamically over time. The main goal of the CleanerAgent is to maintain a building clean, searching dust in rooms (partitioned into a grid of fields) and cleaning them appropriately. To do this, the agent will be equipped with a dust sensor, mop and broom, and a dust-box. The building contains dustbins and battery charging stations. To achieve its main goal, if its internal dustbox is full, the agent has to move to a dustbin to empty it. Similarly, if its battery is nearly empty, it has to recharge it at a battery charging station.

To emphasize adaptivity requirements, two scenarios are illustrated below in order to understand how the system has to deal with environment changes and failure symptoms.

Environment changes. To clean up the building, the main activities of the agent are to wander around searching for dust and to clean. Let us now consider that, while cleaning, the agent senses that its

1 For more details on modelling activities see [3].
internal dustbox is full: it has to 'adapt', changing its behaviour in order to empty its dustbox, carrying it to a dustbin.

This scenario shows the need for internal (e.g. dustbox) and external (e.g. dustbin) resources, that can be monitored, sensed, and manipulated. Also the dust located on fields of a room can be seen as an external resource. Internal resources, which are generally under control of the agent itself (besides failure), include cleaning and movement components, sensors, dustbox, and battery.

Developing the design framework, we have to consider that the state of these resources affects the goal achievement process. The implemented system has to adapt its behaviour to these changes in order to properly achieve its main goal(s). Thus, it should be possible already at design time, to capture how the agent’s behaviour is affected by the environment.

**Failure prevention.** Let’s consider the risky error of a broken battery. This error would surely lead immediately to system failure and has thus to be anticipated by the system whenever possible. Known symptoms of battery degeneration could be a suddenly reduced lifetime or voltage. Continuing with the scenario previously described, while searching for a dustbin (which can be a time-demanding task, if we suppose that the simple agent has no complete information on the building), it is possible that one of these symptoms is observed. To avoid system failure, the agent has to change behaviour, initiating a recovery action, e.g. to alert the technician or, in a broader, open environment, to go to a body shop and to ask for battery replacement.

Such a scenario emphasizes the need of a modelling process for faults (symptoms) in a design framework, in order to avoid system failure and to achieve its goals correctly.

### 2.2.1 Goal-oriented modelling of the example

For the scope of this paper, we do not consider modelling of the early requirements analysis phase of the Tropos methodology, where external stakeholders and the system as it is prior to the introduction of the agent, are defined.

Using the basic Tropos diagrammatic notation [3], a possible model for the late requirements analysis of the CleanerAgent is shown in Figure 1. It details the stakeholders’ goals, the agent has to achieve, and tasks, it has to be capable of, in order to achieve these goals.

For instance, the CleanerAgent’s top goal CleanRoom is AND-decomposed into the two sub-goals DealWithDust and MaintainBatteryLoaded, where the first goal deals with cleaning activities, and the second goal expresses the need for the robot’s battery to be maintained above a certain threshold level of charge in order to make the robot working. The DealWithDust goal is further decomposed into the leaf goals FindDust, CleanField, and EmptyFullDustbox. Leaf goals are operationalised through plans (tasks), which are the means to achieve a goal (means-end relation). The use mop and use broom plans represent two different means to achieve the goal CleanField.

The purpose of the Tropos architectural design phase is usually to split the system into smaller sub-components. Since the system is very simple, we consider the CleanerAgent system as a single component.

Clearly, the Tropos model in Figure 1 lacks of many important requirements elicited in the previous scenarios, because it provides only a static view of the system’s goals. The modelled goals are persistent and can not be correlated to any specific system state or event. Moreover, the goal model contemplates only the (optimistic) way of goal achievement and does not model explicitly, for example, what to do if some error occurs.

**Figure 1:** The CleanerAgent modelled in a Tropos goal model.

### 3. RELATED WORK

In [1], authors identify a goal-oriented approach for engineering adaptivity requirements including monitoring, decision, and adaptation functionalities. In this four levels framework, each level corresponds to the objectives of a different stakeholder. Level 1 comprises traditional requirements engineering, fulfilled by the system developer in order to elicit customers’ and users’ objectives. Level 2 considers requirements engineering, fulfilled by the system itself at run-time in order to determine if and how to adapt. Level 3 includes requirements engineering to determine the system adaptation architecture. Level 4 spans research on adaptation. On level 1, [7] proposes goal-oriented modelling of every possible system configuration in a distinct goal model (using KAOS). Our work covers level 1 of this categorization and aims at defining the foundation necessary to address the implementation of level 2.

Also [11] can be categorized at level 1. It enriches i* models to obtain a design-level view, aiming at a specification of autonomic systems. Annotations such as sequence, priority, and conditions are introduced for decompositions, while, expressions can define variable contribution to soft-goals. The obtained models are a first step towards the goal model approach to get a more detailed system design. Nevertheless, the environment and its influence on the system behaviour can be only modelled at high-level through i* delegation. Moreover, the concept of goal lacks of an important run-time property such as the life-cycle with creation and achievement conditions. This work lacks also a definition of a complete development process.

At the requirements stage, in [25] the authors analyse all possible scenarios generated from a formal model of an adaptive system, in order to check for failures. Here, the main idea is to anticipate failures, by introducing the concept of obstacle, used to model a set of undesirable behaviours, which prevents goal achievement. Such a requirements analysis is conducted by exploiting a formalization of goal models in temporal logic, an approach which may become infeasible for large, complex systems. The same limit is shown by Formal Tropos [5], a first order temporal logic specification of a Tropos requirements model, which allows to perform verification by analysing huge number of scenarios with the purpose to find those which may lead to system failure (i.e. counterexamples). Our main concern is instead on how to model requirements for adaptive systems which may foresee and manage potential failures at runtime.

The concept of antigoal [24] is an obstacle obtained by negating security-related goals and is thus related to some attacker agents.
which might benefit from it. This approach differs from and complements our work at the same time. In fact, despite we do not consider goals belonging to malicious agents, we deal with agent failures from the developer viewpoint.

Also [9] argues that it is not feasible to anticipate and prevent all possible errors in complex agent systems. The approach presented therein attempts to give to a software system the capability to recognize incorrect behaviour at run-time and to initiate recovery actions, avoiding failure. The system monitors itself to detect symptoms related to errors, and can execute implemented recovery actions. In alternative, to avoid to get stuck, the system tries different available actions in the domain, which may allow overcoming the possible failures, and learns from their success. However, they do not address issues related to modelling systems which endow such features.

Relevant to our work is also research in AOSE methodologies, which consider goal modelling a core activity along the software development [8]. In particular, recent work on characterizing goal types [4] are considered in our approach. Still in the AOSE context, worth mentioning are recent proposals to explicitly model the environment where software agents are situated in, introducing the concept of artifact [15]. In our approach, we intend to exploit basic ideas of this work.

4. APPROACH

With the Tropos methodology as foundation for our development framework, we illustrate some extensions to the architectural design phase in order to explicitly model the important properties of an adaptive system, basing on the motivating example in Section 2.2 and its Tropos architectural design goal model in Figure 1. This will include modelling of the environment with its relationships to the goals of the system and modelling of possible faults in the system, along with appropriate recovery activities to avoid system failure. Within the architectural design phase we are considering, a goal model represents the agent intentionality in terms of how the agent perceives the environment, applies strategies to fulfill its responsibilities, and chooses alternative ways to adapt to requirements changes. Like a human being, the agent should be able to perceive the environment and to exhibit a suitable behaviour.

4.1 Goal types

In the standard use of Tropos, goals are general states of affairs to be achieved. Thus, every goal modelled has to be achieved by the system. Thus all goals are simultaneously in process for satisfaction, and have to be achieved once to satisfy the stakeholders’ objectives. Goals to achieve and plans to perform characterize the behaviour of an agent. Therefore, in our example, CleanerAgent has the goal DealWithDust, which –once satisfied– would not have to be achieved again.

However, to exhibit the desired behaviour, it can be necessary to compose goals that have to be achieved for the whole agent lifecycle, goals that have to be satisfied only in certain situations (or only once), and goals that give simply a means to perform specific activities without having a particular achievement condition. To deliver on such requirements, we extend the Tropos framework, introducing several different goal types, as proposed in [4] and available within most BDI-agent based languages and platforms, such as JACK, Jason, and Jadex, namely maintain-goals, achieve-goals, and perform-goals.

Therefore, the first extension of Tropos has been to apply these goal types to the goal model in Figure 1, resulting in the goal model of Figure 2.

In the Tropos analysis phase, the two subgoals of the main goal (CleanRoom), can be described as maintain the room clean (goal DealWithDust) and maintain the battery loaded (goal MaintainBatteryLoaded). Thus, these two goals describe a state condition to be maintained through the whole life of the agent. This means that every time the specific state is not maintained, the agent has to deliberate some activity, to bring about the agent to a valid state.

Next, DealWithDust is decomposed into FindDust, CleanField, and EmptyFullDustbox. We now analyse, to which goal these types correspond and how they are related to one another. In this simple example, we decided to set the subgoal FindDust as a perform-goal, performing a ‘blind’ step-by-step search for dust at any time the system has not to do tasks that are more urgent. To the goal CleanField, we assign an achieve-goal flag, where the state to achieve will intuitively be a clean field. Similarly, the third goal will be to achieve the empty condition for the agent dustbox, but only when the dustbox is full.

Furthermore, we recognize the need to have some additional relations between the subgoals, in order to give to the agent the intended behaviour. We introduce only one new basic relationship, the inhibition. This relationship, used also in [21] for the implementation phase, expresses a simplified form of priority, which can be easily expressed graphically, with the following intended meaning: if goal A inhibits goal B, any time A has to be achieved, the achievement process of B has to be stopped until A is achieved (or, if A is a maintain-goal, until the desired state is reached).

In our example (see Figure 2), we identified four inhibition relations. They express priority of ‘battery loading’ over all cleaning activities (i.e. goal MaintainBatteryLoaded inhibits the goal DealWithDust) and of ‘emptying the cleaner’s dustbox’ over the remaining cleaning activities (i.e. goal EmptyFullDustbox inhibits FindDust and CleanField). An additional inhibition link asserts that cleaning a field has always precedence over searching for other dirty fields, i.e. the goal CleanField inhibits the goal FindDust.

4.2 Environment Modelling

The first scenario in Section 2.2 emphasizes that internal agent abilities need be correlated with the state of environmental entities. For example, the CleanerAgent has to start and stop searching and cleaning activities in relation to state changes of environmental resources, e.g. ‘dustbin is full’. To effectively deal with our objectives, the general Tropos goal model has to be extended along the following main directions:

1. possibility to model the environment (the domain knowledge)
2. possibility to specify several types of goals, like defined in [4], and constraints between goals (i.e. precedence, priority);
3. possibility to correlate goals with changes in the domain knowledge.

The modelling process. As first step of the process, by analysing modelled goals and textual requirements, the designer tries to capture non-intentional entities involved in the desired behaviour. For a graphical representation of these entities, in this first proposal we selected standard UML classes, grouped by packages (in the lower part of Figure 2). Regarding the previous characterization of internal and external resources of the CleanerAgent, the resources

\[\text{Following the Tropos requirements analysis phase, possibly involved intentional entities (namely, stakeholders and other human and software actors in the system) are already modelled.}\]
(entities) we captured can be classified into two groups: entities belonging to the agent and entities belonging to its environment. The former include, among others, a dustbox and a battery. The latter are the rooms with dust and possible other objects, including dustbin and battery loader. As next step, the designer tries to elicit the relationships of these entities to the goals in the goal model. As a next step, the designer tries to elicit the relationships of these entities to the goals in the goal model. As a next step, the designer tries to elicit the relationships of these entities to the goals in the goal model. As a next step, the designer tries to elicit the relationships of these entities to the goals in the goal model.

Figure 2: A fragment of the extended Tropos model for Cleaner Agent that shows correlations between environment and system entities.

A condition relates the state of an environmental entity to the state of a goal. The use of different types of conditions allows to guide the goal achievement process, triggering transition from a goal state to another, and to guard transitions between goal states, as described in [2]. At run-time, this would enable the system to be aware of and thus to react to environmental changes.

Regarding the scenario of adapting to a full dustbin, we identified the goal EmptyFullDustbox to be an achieve-goal. This achieve-goal has to be ‘enabled’ every time the dustbox is full, thus we create a CreationCondition, which relates the goal with the agent’s dustbox. Moreover, the goal is achieved only if the dustbox is empty again, which can be expressed stating an AchievementCondition. Therefore the agent, at runtime, will try to carry out those plans recognized to be the means to this goal (in this case we modelled only the plan carry to bin), until the achievement condition is true. Our first proposal for a graphical language to model these concepts is shown in Figure 2.

4.3 Fault Modelling

In order to achieve their objectives, a main feature for self-adaptive systems is to autonomously prevent system failure by adapting correctly to new circumstances. To effectively deal with required adaptivity properties, we propose to complement the Tropos agent goal model (which is concerned with the modelling of desired states of affairs) with the possibility to model undesirable (faulty) states, which are known to the designer for possibly leading to system failure. In general, the negation of these undesired states would also be modelisable by goals, but the idea is, that following the “human way” of reasoning, it can often be easier to enumerate undesirable things than to precisely define desirable objectives.

The following is the expected run time behaviour: at the time that indications (or symptoms) for these undesirable states (which can be seen as faults or errors in the system) are noticed by the system, appropriate recovery activities should be promptly initiated. These recovery activities can include not only simple activities to be performed, but also new goals, which have to be achieved or complex diagnosis activities which have to be started.

It could be argued that a truly adaptive (or even autonomic) system has to properly adapt to avoid any failure, also if it was not recognized at design time. The following citation captures, why it is not possible to build agents able to avoid failure fixing faults which were not previously modelled.

For an autonomous system (AS) to fix any fault, the humans implementing the AS have to have anticipated the fault, and if a fault is anticipated, it is not a design fault [1]. Therefore, we want to give to the designer the possibility to model failure conditions and fault recovery activities in a simple, intuitive way. Our approach to system design gives a way for modelling how a system should self-adapt to avoid (or, if it is not possible to do so, at least to recover from) faulty situations. Worth to notice, this approach gives not the same amount of flexibility than an exhaustive formalization of goals, activities and possible states, together with a planning algorithm. However, for complex systems in an unpredictable, non-discrete environment, a formal approach seems not to be feasible.

The steps of the proposed modelling process for error recovery are illustrated by analysing a failure situation in our motivating example. Basing on Tropos architectural design, we introduce an error modelling dimension, which includes new abstractions of error, symptom, and recovery activity and includes several modelling steps:

1. Capture possible errors which may lead to system failures.
2. Reveal symptoms for (or, if it is feasible, to predict) the modelled errors.
3. Model possible recovery activities for error symptoms. Detail of new goals/plans.
4. Formalize such symptoms, evolving them to creation conditions for the associated recovery activities.

The fragment of the Tropos model sketched out in Figure 3 is related to the motivating example. The design process fulfilled to come out with such a model is described in the following.

As first step, the designer will try to elicit possible undesired states (error states) of the system, such as battery broken, represented by a ‘jagged’ circle. As a possible symptom for this error the designer recognized a very rapid discharge, modelled by an ellipse, related to the error and to the interested environmental entities, in this case the agent’s battery. The next step is to model possible recovery activities for each recognized error.

In this first proposal, we relate recovery activities directly to possible symptoms, leaving out a true diagnosis phase. We do it this way, because our aim is to create intuitive and easily implementable models. Relating a recovery action directly to a symptom, we capture the simplest form of diagnosis. However, more complex diagnosis mechanisms can always be either manually added in the implementation phase or modelled as a first step in recovery activities.
5. FRAMEWORK INSTANCE WITH AN IMPLEMENTATION IN JADEX

In this section, we give guidelines for a mapping from the newly introduced design concepts to BDI agent definitions. Following them, we realized a Jadex based implementation of the CleanerAgent example, which will be illustrated and used to discuss some preliminary results.

This mapping process is founded on works already extensively discussed in [16] and in [12]. Specifically, such works proposed a mapping from the most important concepts in Tropos goal models to the Jadex agent definition language, along with an automatic code generation tool, called r2x (Tropos to Jadex), which was recently integrated into the Tropos modelling tool TAO4ME [19]. This Section briefly recalls how we exploited this mapping to realize the CleanerAgent example and emphasizes how we extended it in order to support developing self-adaptive systems, along the approach described in Section 4, namely, by explicitly coding goal types, entities in environment, error states and recovery actions. A discussion on the observed behaviour of the resulting code in preliminary experiments concludes the section.

5.1 Mapping

As a first step, the CleanerAgent Tropos model, including its main goal model with root goal CleanRoom, and all the goal models related to recovery activities, is mapped to a Jadex XML Agent Definition File (ADF) and Java code following the approach in [16]. The generated code implements the agent’s reasoning mechanisms needed to select correct plans at run-time to achieve desired goals.

The r2x tool analyses a goal model exploring goal decomposition trees. The goal hierarchy is mapped to Jadex goals along with Java files containing the decomposition logic. This choice is emphasized by the fact that goal hierarchies are not a native feature of Jadex. For Tropos plans, Java file skeletons are generated and connected to the relative goals using the Jadex triggering mechanism. The goal decomposition graph is also stored in the agent’s belief base, together with all contributions to softgoals and dependencies to other agents. An architectural choice makes the agent aware about its abilities, namely, at run-time the agent can control its behaviour by navigating the modelled goal graph. This includes a basic failure handling skill, which is made possible exploiting alternatives included in the goal model. The generated code skeleton is ready to be executed on the Jadex platform, exhibiting a basic set of behaviours corresponding to the designed source model. It can be modified and customized as needed. It does however not contain the concrete activities the system has to be able to perform, which have to be implemented manually or by following proposals such as capability modelling [17].

Goal types. The goal types defined in Section 4.1 can be directly mapped to each Jadex goal defined in the generated ADF, which represents the Tropos goal hierarchy. In this approach we limit the mapping of goal types to only leaf-level goals, to avoid conflicts in the goal achievement process.

Regarding the example, we map the leaf goal MaintainBatteryLoaded straight-forward to a Jadex MaintainGoal. However, even considering only leaf-level goals, the mapping is not always straight-forward. For example, it is not enough to map the perform-goal FindDust to the corresponding Jadex goal type, because at execution time it would be achieved only once, executing associated plans. The appropriate goal type, corresponding to recurrently perform the goal, is called perform goal with retry-flag in [4], and corresponds to a RecurrentPerformGoal in Jadex. The two goals annotated as achieve-goal are also mapped to maintain-goals. This choice makes possible to act like an achieve-goal (having the same target condition, and being activated by the maintain condition, which corresponds to a negated creation condition), preserving the goal hierarchy.

The inhibition link is a concept explicitly taken from the Jadex language, can clearly be directly mapped to an ADF flag of the form <inhibits ref="goal_to_inhibit"/> in the definition of a goal which inhibits other goals.

The environment. Environmental and system entities have been mapped to objects in the agent belief base. They do not define physical objects, but the agent view on these objects, and form thus part of the local knowledge of a single agent. The modelled class diagrams of Figure 2 have been implemented as Java classes. These classes, including details on variables and methods that will be needed, can also be automatically generated using available UML tools. Instances of the Java classes are then defined in beliefs in the CleanerAgent’s ADF belief base section.

For example, the dustbox entity defined in Figure 2 maps to a Java class with the same name and is instantiated as a fact in the belief named dustbox (Figure 4) and can be accessed from other sections of the ADF as well as from Java classes through an API delivered with Jadex. Beliefs sets can be used if more than one instance of a class have to be stored.

![Figure 3: A fragment of the extended Tropos model for CleanerAgent that shows correlations between possible system errors and environment entities. Here, feasible recovery activities are also described.](image)

![Figure 4: Excerpt of the belief base in the CleanerAgent ADF.](image)
Now we can map the conditions defined in Section 4.2 to Jadex code. Conditions can be stated for each goal and plan in the Jadex ADF, by referring to beliefs defined in the belief base, similarly to Java method invocation. The supported conditions depend on the goal type. Common to all goal types are the creation, context, and drop conditions. Using our example, for the achieve-goal EmptyFullDustbin, we can directly map modelled creation and achievement conditions to the ADF (Figure 5).

Figure 5: Excerpt of goal definitions in the CleanerAgent ADF.

Failure prevention. The last step is to map failure prevention mechanisms, modelled in Section 4.3. A modelled fault/error, represented by a jagged box in Figure 3, is a concept needed in the design phase, but is not mapped to BDI code. On the contrary, related concepts require to cope with faults, such as symptoms and recovery activities, have to be implemented into agent code. The goal models for recovery activities were already mapped in the first mapping step and appear as loose parts of goal model structures, within the CleanerAgent definition. Here, our idea is to achieve a run-time behaviour such that goal models referring to an appropriate recovery activity are activated each time modelled symptoms, correlated to a specific fault, occur. Therefore, symptoms can be seen like creation events for the related recovery activities goal models. To achieve this, first, symptoms have to be formalized, regarding available environmental entities of interest for the represented error. Then, they can be mapped to creation conditions for the top-goals or plans of the repair activities. Moreover, if more alternative recovery activities are modelled for one symptom, a selection algorithm is needed. The meta-reasoning functionality provided by Jadex is used for this purpose.

In our motivating example we modelled battery failure as a state to avoid to avoid system failure. An important symptom which anticipates this failure is a dramatically rapid battery discharge. A creation condition for the top goal of the recovery activity GetNewBattery is set to monitor battery usage and lifetime, accessing the battery instance in the belief base and becomes true if specific values are exceeded.

The code skeleton obtained in this way has to be completed by programming the plans as traditional Java code. The next section provides simple tests, verifying that under specific environment changes the generated agent exhibits the intended behaviour.

5.2 Discussion of a prototype

A first implementation of the modelled CleanerAgent has been realized on the basis of the code skeleton generated by the t2x tool and following the mapping described in the previous section for the goal model and the environmental entities. Then, the modelled plans were implemented in Java, to match the expected activities, and a simple environment for a simulation is implemented.

The implementation of the failure scenario is very similar to the one just described. The behaviour of getting a new battery is activated if the modelled symptom is recognized (i.e., the associated creation condition is true) and achievement of this goal has precedence over all other goals in the goal model.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we introduced an agent-base development framework for self-adaptive software. The agent-oriented approach allows designer to abstract from the complexity of system technologies, focusing more on the knowledge level issues, i.e., explicitly
modelling agents, goals, plans and their relationships to characterize the system requirements specification.

The proposed framework is founded on the Tropos methodology. In this paper we describe several extensions required to deal with self-adaptive software. Through the use of a motivating example, we showed the need of enriched goal models in order to specify goal achievement conditions in relationship with the environment, and the possibility to model faults and corresponding recovery activities. Moreover, we illustrated how adaptivity requirements naturally fit within a BDI agent architecture, giving an implementation of the motivating example and sketching the transition phase from design to BDI-based agent code for the Jadex platform.

Several aspects of this work should be further investigated in order to complete the objective of our long-term-research, which consists in defining a complete process and tool-supported design framework to develop self-adaptive systems. First of all, the proposed extensions to the modelling language (goal types, faults, symptoms, and recovery activities) should be formalized and integrated into the language meta-model. The graphical notation needs to be revised and modelling tools should be extended accordingly in order to support model building.

Goal analysis techniques can be extended to reason on these new abstractions. Moreover, the mapping towards a BDI architecture should be completed, tested, and automated by tool support. Validation of the whole approach on more realistic scenarios will also be performed, focusing on adaptivity qualities of the system to develop.

7. REFERENCES


