Automating Software Design: Exploring and Evaluating Design Alternatives

Abstract. Development of complex socio-technical IT systems is a very important and a relatively new problem for Software Engineering. Traditional software development methodologies should be revised and improved to capture properties of both human and artificial agents and interactions between them. This thesis proposal aims at building a framework for the automatic selection and evaluation of design alternatives. This is supposed to be done by (i) applying existing planning tools to automate the generation of design alternatives, and (ii) developing methods and algorithms for the evaluation and analysis of design alternatives based on game theory. Supporting tools will be developed to guide their users through the software design process.

Keywords: software design, socio-technical systems, planning, game theory

1 Introduction

Motivation. Development of complex socio-technical IT systems is nowadays a very important issue in Software Engineering [21]. Modern IT systems involve human agents, thus, considering just system functionality is not enough – the interaction between system components and organizational/social environment should be taken into account. Because of this "human aspect" the design of socio-technical IT systems is a relatively new problem for Software Engineering. In addition, such systems are usually large-scale ones, while even for traditional large-scale systems existing software development methodologies are not very successful. Despite of the efforts put into establishing the software development methodologies during the last decades, big projects often run out of time and budget, and are not robust and secure enough to meet continuously increasing user requirements[1]. To face the problem *the methodologies that guide the development process since its early stages should be revised, enriched, and further developed.* Moreover, the complexity of present socio-technical systems is such that to be effective all methodologies have to be equipped with mechanisms for *automation support.*

Problem. What kind of automation a designer needs? One of the proposals is to facilitate the designer's work by automating the specification refinement process. The approach is reflected in Model Driven Architecture (MDA) [16] which focuses on (possibly automatic) transformation from one formal system model to another. Tools supporting MDA exist and are used in the Rational Unified Process for software development in UML. Yet the state-of-the-art is still not satisfactory [20]. Another approach to the problem are design patterns [11] which propose to match standard documented solutions with the design problems arisen in the certain context.

Such approaches only cover part of the designer's work, while there is another activity where the support of automation could be beneficial as well [15]:

> "Exploring alternative options is at the heart of the requirements and design processes".

Indeed, in most current software engineering methodologies the designer has tools to report and verify the final choices (e.g. Goal-models in KAOS [4], UML Classes or Java code), but there is no, actually, the possibility of automatically exploring the alternatives and finding a satisfactory one. *This thesis proposal aims at exploring the problem and building the framework for the automatic selection and evaluation of design alternatives.* The automated selection of alternatives at the early software development stages can

[1] See Standish Group reports at http://www.standishgroup.com/sample_research/.

be the most beneficial and effective. The reason is that at the early stages the design space is larger, it is at these stages when most alternatives are examined (and discarded) and thus a good choice might have significant economic impact. Supporting the selection of alternatives would lead to more alternatives being considered, more thorough analysis of considered alternatives and an overall more complete and trusted design.

Approach. It can be noticed that requirements – at least within the frameworks such as i* [26], Tropos [2] and the like – are conceived as networks of delegations among actors (which are organizational/human/software agents, positions and roles). Every delegation involves two actors, where one actor delegates to the other the delivery of a resource, the fulfillment of a goal, or the execution of a task. The delegatee can deliver/fulfill/execute the delegated service (i.e resource, goal or task), or further delegate it, thus creating another delegation relation in the network. Intuitively, these can be seen as *actions* that the designer ascribes to the members of the organization and the system-to-be. Further, the task of designing such networks can be framed as a *planning* problem for multi-agent systems: selecting a suitable possible design corresponds to selecting a plan that satisfies the goals of the human or software agents. Many off-the-shelves planners are available that, given the problem domain description, generate a sequence of actions (or a plan) that satisfy a set of predefined goals. *One of the aims of this proposal is to embed existing planning tools into the framework to automate the generation of design alternatives.*

Of course, the designer remains in the loop: designs generated by the planner are suggestions to be evaluated, amended and approved by the designer. The tricky point here is the solution evaluation which can be complex enough even for very experienced designers with considerable domain expertise. Indeed, a challenging characteristic of the design of socio-technical IT system is that human agents should be taken into account. They can be seen as *players* in a game theoretic sense as they are self-interested and rational. This means they want to minimize the load imposed personally on them, i.e. they want to reduce the number and the complexity of actions they are involved in. In a certain sense non-human agents, i.e. system components, are players as well as it is undesirable to overload them. Each player has a set of strategies he could choose from, e.g. he could decide whether to satisfy a goal himself or to pass it further to another system actor. Strategies are based on the player's capabilities and his relations (e.g. subordination, friendship, or trust) with other human and artificial agents in the system. If we assume that each player ascribes a numerical weight to each possible action, then it is possible to calculate the cost of a given design alternative for the player by summing up the weights of actions in the solution he is involved in. Obviously, each rational player wants to minimize this cost, or at least he "searches for justice", i.e. he wants the load to be more or less equally divided among all the involved actors. How to choose the "right" design alternative that will at the same time satisfy the overall system goal and be "accepted" by all actors? *A part of this research is to develop methods and algorithms for the evaluation and comparative analysis of design alternatives based on game theoretic ideas. The aim is to support the designer with the tool that will evaluate and try to optimize the solution to the design-as-planning problem.*

The rest of the proposal is structured as follows: Section 2 overviews briefly the areas related to the topic of the proposal; in Section 3 the selected approach to the problem is detailed; finally, in Section 4 already obtained results are listed and work plan for the next two years is presented.

2 State-of-the-Art

Requirements Engineering and Software Design. Requirements engineering is considered to be a crucial part of software development process [23]. Careful elicitation and analysis of requirements help to develop a system that meets user's expectations, is trustful and robust. According to [23] requirements engineering involves such activities as domain analysis, elicitation, specification, assessment, negotiation, documentation, and evolution. Modeling requirements to software systems and organizations in terms of goals and their interdependences has been a topic of considerable research interest during the last decades [23]. A number of goal-oriented approaches for requirements representation and reasoning were

introduced. For example, KAOS approach [4] supports modeling goals of different types, allows to define goal attributes, links between goals (e.g. to model the situation when a goal negatively or positively supports other goals), AND/OR goal refinement links, links between goals and agents, etc. The i* model [26] offers primitive concepts of actors, goals and actor dependencies, which allow to model both software systems and organizations. With i* framework one can capture why the software is being developed, while the earlier approaches (e.g. Object Oriented Analysis) reflect only what's and how's.

Software design is an intermediate phase between the user requirements elicitation and analysis and the system implementation. The design process results in software specifications which are formulated in a vocabulary understandable by programmers, while requirements are formulated in terms of objects of the real world and their interconnections, and are understandable by stakeholders. The problem of designing software that meets user requirements is addressed, for example, in [24] where a goal-oriented approach to architectural design based on the KAOS framework is proposed. The authors describe the process of deriving software specifications from requirements, and then building an architectural draft from functional specifications. The obtained architecture is then recursively refined to meet non-functional requirements analyzed during requirements analysis phase.

This research proposal is inspired by and was originated in Tropos project[2]. Tropos is an agent-oriented methodology which covers all software development phases from early and late requirements analysis through architectural and detailed design to implementation [2]. It is based on i* framework with actors, goals, dependencies, plans, resources, and capabilities as basic modeling constructs. The key point in Tropos is in using the same notation through the whole software development process. During the early requirements analysis stakeholders and their intentions are identified, i.e. the organizational environment of the system under development is modeled. Late requirements analysis puts the system-to-be into its operating environment and models its dependencies with the other actors of the organization. The design phase is subdivided into architectural and detailed design and is structured as follows. First, the overall architectural organization is defined, and the capabilities needed by the actors to fulfill their goals and plans are identified. Then, a set of agent types is defined together with assigning each of them one or more different capabilities. Next, agents' micro level is specified during the detailed design phase. Tropos framework includes not only modeling but also reasoning tools which help in requirements analysis, validation and verification (e.g. goal reasoning tools, automatic verification of security and trust requirements in Secure Tropos – see project homepage for the details).

Designing Social Structures. Another perspective of information systems development, inspired by the organizational theory, is reflected in the literature. In [10] ontology for information systems is proposed which adopts i* organizational modeling framework with its actor, goal and social dependency primitives. The paper describes a number of organizational styles (e.g. joint venture, hierarchical contracting, etc.) and social patterns (e.g. broker, mediator, wrapper, etc.). The former describe the overall structure of the organizational context of the system or its architecture, while the latter focus on the social structures necessary to achieve one particular goal. Both organizational styles and social patterns guide the development of the organizational model for an information system. In [6] a methodology for the design of agent societies based on the type of co-ordination structure is described. Following the organizational theory, co-ordination in agent societies is divided into three types: markets, networks and hierarchies. Design steps include selecting a coordination model from ones available in the library; describing interaction between the society and its environment, and behavior of the society in terms of agent roles and interaction patterns; finally, the internal structure of agents is defined.

Automated Software Design. Almost fifty years ago the idea of actually deriving the code directly from the specification (such as that advocated by Manna and Waldinger landmark paper) started a large program of funding for deductive program synthesis that has not gained significant results in the past. The key idea of the approach is the following. A system goal together with the set of axioms are specified, and then a theorem, i.e. goal of the system described in a formal specification language, is proved with

[2] See project homepage at http://www.troposproject.org for the details.

the help of axioms. A program for solving the problem is extracted from the proof of the theorem. The field is still fairly active and several program synthesis systems were proposed (see, e.g. [8, 19]), but they are mainly domain-specific, require considerable expertise, and in some cases do not actually guarantee that the synthesized program will meet all requirements stated by designer [8].

Conceptually, the *automatic selection of alternatives* is done in deductive program synthesis: the theorem prover selects among the appropriate axioms to prove the theorem. Instead, in this proposal it is argued that the automatic selection of alternatives should and indeed can be done at earlier stages. Requirements models are by construction simpler and more abstract than software models. Therefore, techniques for automated reasoning about alternatives at the early stages of the development process may succeed where automated software synthesis has not been able to deliver.

Another approach is to facilitate the work of the designer by supporting the tedious aspects of software development by automating the specification refinement process. Such approach underlies Model Driven Architecture (MDA) [16], which focuses on (possibly automatic) transformation from one formal system model to another. MDA approach, proposed by Object Management Group, is a framework for defining software design methodologies. The central focus of MDA is on the model transformation, for instance from the platform-independent model of the system (PIM) to platform-specific models (PSMs) used for implementation purposes. Models are usually described in some formal language (e.g. UML), and the transformation is performed in accordance with the set of rules, also called mapping. Transformation could be manual, or automatic, or mixed. However, the state-of-the-art is far from being satisfactory [20].

Among the proposals on automating a software design process the one of Gamma et al. on design patterns [11] has been widely accepted. A design pattern is a solution (commonly observed from practice) to the certain problem in the certain context, so it may be thought as a problem-context-solution triple. Several design patterns can be combined to form a solution. Note that it is still the designer who makes the key decision – on what pattern to apply to the given situation.

An interesting work of Gross and Yu [14] should be mentioned here which relates the representation and analysis of non-functional requirements with software design patterns. The proposed approach organizes, analyzes and refines non-functional requirements to provide guidance and reasoning support in applying patterns during a software system design.

AI Planning. The field of AI planning has been intensively developing during the last decades, and has found a number of applications (robotics, process planning, autonomous agents, etc.). Planning approach recently has proved to be applicable in the field of automatic Web service composition [18]. There are two basic approaches to the solution of planning problems [25]. One is graph-based planning algorithms in which a compact structure, called Planning Graph, is constructed and analyzed. In the other approach the planning problem is transformed into a SAT problem and a SAT solver is used.

There exist several ways to represent the elements of a classical planning problem, i.e. the initial state of the world, the system goal, or the desired state of the world, and the possible actions system actors can perform. The widely used, and to the certain extend standard representation is PDDL (Planning Domain Definition Language), the problem specification language proposed in [13]. Current PDDL version, PDDL 2.2 [7] used during the last International Planning Competition[3], supports many useful features, e.g. derived predicates and timed initial literals.

Design as Planning. A few works can be found which relate planning techniques with software requirements analysis and design. In [1] a program called ASAP (Automated Specifier And Planner) is described, which automates a part of the domain-specific software specification process. ASAP assists the designer in selecting methods for achieving user goals, discovering plans that result in undesirable outcomes, and finding methods for preventing such outcomes. The authors describe the planner they have implemented, which combines adaptive and hierarchical planning techniques (see [1, 18] for references). The problem of their approach is that the designer still performs a lot of work manually determining the

[3] See http://ls5-www.cs.uni-dortmund.de/~edelkamp/ipc-4/ for the details.

combination of goals and prohibited situations appropriate for the given application, defining possible start-up conditions and providing many other domain-specific expert knowledge.

Castillo et al. [3] present an AI planning application to assist an expert in designing control programs in the field of Automated Manufacturing. The system they have built integrates POCL, hierarchical and conditional planning techniques (see [3, 18] for references). The authors consider standard planning approaches to be not appropriate with no ready-to-use tools for the real world, while in this research proposal the opposite point of view is advocated. Another recent application of the planning approach to requirements engineering is proposed by Gans et al. [12]. Essentially, the authors map trust, confidence and distrust described in terms of i* models [26] to delegation patterns in a workflow model. Their approach is inspired by and implemented in ConGolog (see [18] for description and references), a logic-based planning language. However, they focus more on representing/modeling trust in social networks, than on the design automation. The authors do not go far in explaining how they exploit the planning formalism in the design process and, moreover, do not give any examples of modeling.

Game Theory. Game theory is an established discipline which deals with conflicts and cooperation among rational independent decision-makers, or players. A strategic game is defined by a set of players, and, for each player, a set of actions (called strategies) and a payoff function that assigns a numeric value to each of the player's action profile. The theory studies various types of games: non-cooperative and cooperative games, dynamic games in which the order of players' decisions is important, games with incomplete information, etc. The key concept in classical game theory is the notion of equilibrium [17] which defines the strategies of each player in such a way that all players are satisfied to a certain extent. In other words, this set of strategies is a *stable state* which none of the independent rational players wants to deviate from. However, this does not mean that each player maximizes his utility by choosing the equilibrium strategy; rather, we can say that by playing an equilibrium each player maximizes his utility locally, given some constraints (on the other players' actions). For example, playing the *Nash* equilibrium means that no player can benefit when deviating from his equilibrium strategy given that all other players play the equilibrium. Nash equilibrium is proved to exist in mixed strategies (i.e. when a player is randomizing over several strategies), while in pure strategies it might not exist.

Game theory is applied in various areas, especially in economics (modeling markets, auctions, etc.), corporate decision making, defense strategy, telecommunications networks and many others. Among the examples are the applications of game theory to so called network games (e.g. routing, bandwidth allocation, etc.), see [22] for references.

Recently the idea of applying *mechanism design* in the area of multi-agent systems has emerged [5]. Mechanism design can be viewed as a branch of game theory which intends to design systems/game environments so that certain properties are satisfied when the equilibrium state is reached. Another name for this discipline is implementation theory [17] as it implements a particular objective despite the self-interests of individual players. However, a number of fundamental research problems should be solved [5] in order mechanism design to be actually applied to the design of complex distributed systems composed of multiple interacting agents.

3 Research Contribution

3.1 Problem

As it was already introduced in Section 1, this thesis proposal aims at building a framework for automatic selection and evaluation of design alternatives. This is supposed to be done in two phases. The first is to apply existing planning tools to automate the generation of design alternatives. The second phase is to develop methods and algorithms for the alternatives evaluation and analysis based on game theoretic ideas, and, as a result, to support the designer with the tool to evaluate and optimize a solution to the design-as-planning problem.

Requirements engineer/designer will be supported in the selection of the best alternative by changing the software development process as follows:

- Requirements analysis phase
  - Identify system actors, goals and their properties.
  - Define dependency relationships among actors.
- Design phase
  - Automatically explore the space of design alternatives to identify delegation links and assignments of goals to actors. If no alternatives can be generated, return to the requirements analysis phase and revise the initial structure.
  - With the help of supporting tools evaluate the obtained solutions. If necessary, ask for another, optimized solution.

3.2 Objectives and Approach

Formalizing the design-as-planning problem. We have chosen AI planning approach to support the designer in the process of selecting the best alternative. The motivation of such choice, as it was stated in Section 1, is that the problem of generating design alternatives can be naturally represented as a planning problem. The basic idea behind planning approach is to automatically determine the course of actions (i.e. a plan) needed to achieve a certain goal where an action is a transition rule from one state of the system to another [25, 18]. Actions are described in terms of preconditions and effects: if the precondition is true in the current state of the system, then the action is performed. As a consequence of an action, the system will be in a new state where the effect of the action is true. Thus, once we have described the initial state of the system, the goal that should be achieved (i.e. the desired final state of the system), and the set of possible actions that actors can perform, then the solution to the planning problem is the (not necessarily optimal) sequence of actions that allows the system to reach the desired state from the initial state.

While casting the design process as a planning problem, the following question must be addressed: which are the "actions" in software design? In Tropos approach [2] when drawing the model of a system, the designer assigns goals to actors, defines delegations of goals from one actor to another, and identifies appropriate goal refinements among the predefined alternative refinements. Such actions will be used by a planner to find a way to fulfill the goals of the system actors.

Planning approach requires a specification language to represent the planning domain and the states of the system and its environment. Different types of logic could be applied for this purpose, e.g. first order logic is often used to describe the planning domain with conjunctions of literals specifying the states of the system.

The language for planning domain description should provide support for specifying:

- the initial state of the system;
- the goal of the planning problem;
- the description of actions;
- the axioms of background theory.

To describe the *initial state of the system*, actors' and goal properties, and social relations among actors should be specified. We propose to represent initial state in terms of predicates that correspond to

- the possible ways of goal decomposition;
- actors' capabilities and desires to achieve a goal;
- possible delegation relations among actors.

The desired state of the system (or *goal of the planning problem*) is described through the conjunction of predicates derived from the description of actors' desires in the initial state. Essentially, for each desired goal a predicate is added to the goal of the planning problem.

An *action* represents a temporal activity to accomplish an objective. The behavior of an actor can be formalized by the following actions he can perform.

Goal satisfaction. An actor can satisfy a goal only if achieving this goal is among his desires and he can actually satisfy it. The effect of this action is the fulfillment of the goal.

Goal delegation. An actor may have not enough capabilities to achieve his goals by himself, and so he has to delegate their satisfaction to other actors. This passage of responsibilities is performed only if the delegator wants a goal to be achieved and can depend on the delegatee to achieve it. The effect of this action is that the delegator does not worry any more about the satisfaction of the goal, while the delegatee takes the responsibility for the fulfillment of the goal and so it becomes his own desire to achieve it. The delegator does not care how the delegatee satisfies the goal (e.g. by his own capabilities or by further delegation), it is up to the delegatee to decide it.

AND/OR goal decomposition. As in different goal-oriented modeling frameworks (e.g. as in Tropos and KAOS) two types of goal refinement are supported: OR-decomposition, which suggests the list of alternative ways to satisfy the goal, and AND-decomposition, which refines the goals into subgoals which all are to be satisfied in order to satisfy the initial goal. An actor can decompose a goal only if he wants it to be satisfied, and only in the way which is predefined in the initial state of the system. The effect of decomposition is that the actor who refines the goal focuses on the fulfillment of subgoals instead of the initial goal. It is assumed that different actors can decompose the same goal in different ways.

In addition to actions, *axioms* of the planning domain can be defined. These are rules that hold in every state of the system and are used to complete the description of the current state. For example, to propagate goals properties along goal refinement the following axiom is used: a goal is satisfied if all its AND-subgoals or at least one of the OR-subgoals are satisfied.

The proposed formalization of the design-as-planning-problem should be viewed as a starting point of our research. We foresee that the evaluation of the proposed approach on the basis of real case studies may cause the refinement and further development of the formalization.

Applying planning. The next step, after the design problem is formalized, is to choose the "right planner" among off-the-shelves tools available. In the last years many planners have been proposed [18]. In order to choose one of them the following requirements are considered:

- The planner should not produce redundant plans. Under non-redundant plan we mean that, by deleting an arbitrary action of the plan, the resulting plan is no more a "valid" plan (i.e. it does not allow to reach the desired state from the initial state).
- The planner should use PDDL (Planning Domain Definition Language) since it is becoming a "standard" planning language and many research groups work on its implementation.
- The language should support a number of "advanced" features (e.g. derived predicates) that are essential for implementing our planning domain, i.e. it should be at least PDDL 2.2.

The first requirement is concerned with the optimality of the generated design decisions. We argue that it is not necessary to focus on the optimal design: human designers do not prove that their design is optimal, why should a system do it? Instead, in our framework the plan is required to be non-redundant, which guarantees at least the absence of alternative delegation paths since a plan does not contain any redundant actions.

The situation in which no solution can be found by the planner might be caused either by an error in the requirements, or by the lack of capabilities the human and software system actors were ascribed. At this point the designer needs to find the way to relax the initial constraints, i.e. to revise actors' desires and capabilities, and possible social dependencies. The problem with our approach is that the planner does not usually provide the point where failure occurred. Thus, our goal is to interfere into the

planning process to find the failure cause and/or to invent heuristics to "play" with initial constraints for replanning the design till the solution is found.

Solution evaluation. Another important issue is related to the evaluation of multiple alternatives, when they are available. However, most standard planning tools do not provide all the solutions to the given problem, but they stop when the first plan is found. This is a natural restriction because there can be exponentially many solutions to the problem. Our approach is to treat this case by iteratively generating next alternative on the basis of the evaluation of a current one. In the following the solution evaluation issue is detailed.

Solutions to the design problem can be evaluated both from *global and local* perspectives, i.e. from the designer's point of view and from the point of view of individual actors. The optimality of a solution in the *global* sense could be evaluated with respect to

- the length of the obtained plan;
- time required to execute the plan;
- overall plan cost (the idea of ascribing a cost to each action by each actor is described in Section 1);
- the degree of satisfaction of non-functional requirements in case the plan is adopted.

Regarding the first case, the number of actions in the obtained plan is often the criteria for the planner itself to prefer one solution to another. Thus, it can be assumed that the obtained plan is already (locally) optimal in the sense of the length minimization.

The second and third cases are closely related with the idea of plan metrics introduced in PDDL 2.1 [9]. Plan metrics specify the basis on which a plan is evaluated for a particular problem, and are usually numerical expressions to be minimized or maximized. Of course – and this is often the case for available planners – a planner could ignore the metrics and just evaluate a solution post hoc, which might lead to sub-optimal and possibly poor quality plans. For the minimization of time required to execute the plan the *total-time* variable together with the idea of durative actions can be used (the latter refers to the possibility to ascribe duration to each action of the planning domain). However, the complexity of the problem of optimizing a solution with respect to the defined metrics is very high and the feature is still poorly supported by the available planning tools [9].

The evaluation of design alternatives with respect to non-functional requirements satisfaction is discussed in [15]: *"Different alternatives contribute to different degrees of achievement of non-functional goals about system safety, security, performance, usability, and so forth"*. The authors of this paper define rules to identify application-specific parameters and functions to quantify impacts of different explored alternatives on goal satisfaction.

Local evaluation of the obtained plan is performed for each actor and reflects actors' absolute (individual) or relative (in comparison with other actors) assessment of the solution. For example, actor may have some upper bound on his personal load which he does not want to exceed, or he may want to minimize his individual absolute deviation from the mean utility value (where the player's utility is defined as some "upper bound of outcomes" minus the personal outcome of the game, i.e. it says how much a player "saves").

Another approach we will follow in this research to evaluate the obtained solutions is based on game theory (e.g. on the use of equilibrium concept). The substantial difficulty in applying game theoretic ideas to our problem is that all human and software agents of a socio-technical system should work as a solid mechanism satisfying the overall organizational goal. Differently from classical non-cooperative game theory, where all players choose their strategies independently and simultaneously before the game, in our problem actors' choices are closely interrelated. A player cannot independently change his strategy because the new action sequence will very likely be unsatisfactory, i.e. it will not be a solution anymore. Thus, to satisfy the system goals it will be necessary to impose some additional load (to compensate the one this player tries to avoid) on some other actors – and it might happen that they will not be satisfied with their new utilities, and will try to deviate from the strategy they were imposed, and so on and so

forth. We intend to build the recursive "replanning-towards-optimality" procedure and see whether it will converge to some sort of equilibrium.

A very important step of our research will be connected to the evaluation of the selected approach with the help of *real-life case studies*. Such case studies will serve for verification of the proposed problem representation, and for testing the supporting tools based on methods and heuristics described in this proposal.

The main *application* of the proposed approach lies in the area of software development – mainly, it concerns the automation of passage from requirements analysis to design. Additional applications can be found in designing social structures/organizations (e.g. in business process reengineering), or in the domains where replanning and re-evaluation at runtime is needed (e.g. when queries to evaluate are assigned to the nodes of P2P database).

4 Current Results and Research Plan

What is done so far:

- Initial formalization of the design-as-planning problem was done. The problem representation was translated into PDDL.
- Experiments both to evaluate the approach on simple examples and to choose the appropriate planner were conducted.
- Preliminary tool for applying planning techniques was developed.
- The first part of the approach (application of planning, without evaluation and optimization) was applied to Secure Tropos domain[4].
- Preliminary work on iterative solution building and evaluation was done.

Research plan:

- Spring 2006 – Summer 2006:
  - Explore solution evaluation issues in the light of game theory. Assess the possibility to use mechanism design.
  - Explore other evaluation techniques, and the case of the absence of solution.
  - Planning: on the basis of case studies enrich formalization, assess different planners from the point of view of correctness and performance, try enhanced planning techniques (e.g. planning with duration/metrics support).
- Summer 2006 – Autumn 2006:
  - Automation framework: provide tool(s).
- Winter 2007 – Summer 2007:
  - Consider large real-life case studies to assess/verify the approach.
  - Provide description of the (enriched) methodology.
- In parallel with other activities:
  - Consider framework applications: P2P databases, secure systems design, BPR, etc.
- Autumn 2007:
  - Write the thesis.

[4] The resulting paper is under review for CAiSE'2006.

References

1. J. S. Anderson and S. Fickas. A proposed perspective shift: viewing specification design as a planning problem. In *IWSSD '89: 5th Int. workshop on Software specification and design*, pages 177–184, 1989.
2. P. Bresciani, P. Giorgini, F. Giunchiglia, J. Mylopoulos, and A. Perini. TROPOS: An Agent-Oriented Software Development Methodology. *JAAMAS*, 8(3):203–236, 2004.
3. L. Castillo, J. Fdez-Olivares, and A. Gonzlez. Integrating hierarchical and conditional planning techniques into a software design process for automated manufacturing. In *ICAPS 2003, Workshop on Planning under Uncertainty and Incomplete Information*, pages 28–39, 2003.
4. A. Dardenne, A. van Lamsweerde, and S. Fickas. Goal-directed requirements acquisition. *Science of Computer Programming*, 20:3–50, 1993.
5. R. K. Dash, N. R. Jennings, and D. C. Parkes. Computational mechanism design: A call to arms. *IEEE Intelligent Systems*, 18(6):40–47, Jan./Feb. 2003.
6. V. Dignum and F. Dignum. Modelling agent societies: Co-ordination frameworks and institutions. In *Portuguese Conference on Artificial Intelligence*, pages 191–204, 2001.
7. S. Edelkamp and J. Hoffmann. Pddl2.2: The language for the classical part of the 4th international planning competition. Technical Report 195, University of Freiburg, 2004.
8. T. Ellman. Specification and synthesis of hybrid automata for physics-based animation. In *ASE*, pages 80–93, 2003.
9. M. Fox and D. Long. Pddl2.1: An extension to pddl for expressing temporal planning domains. *J. Artif. Intell. Res. (JAIR)*, 20:61–124, 2003.
10. A. Fuxman, P. Giorgini, M. Kolp, and J. Mylopoulos. Information systems as social structures. In *Proc. of the 2nd Int. Conf. on Formal Ontology in Inform. Sys. (FOIS'01)*, pages 10–21. ACM Press, 2001.
11. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
12. G. Gans, M. Jarke, S. Kethers, and G. Lakemeyer. Modeling the impact of trust and distrust in agent networks. In *Proc. of AOIS'01*, pages 45–58, 2001.
13. M. Ghallab, A. Howe, C. Knoblock, D. McDermott, A. Ram, M. Veloso, D. Weld, and D. Wilkins. PDDL – The Planning Domain Definition Language. In *Proc. of AIPS'98*, 1998.
14. D. Gross and E. S. K. Yu. From non-functional requirements to design through patterns. *Requirements Engineering*, 6(1):18–36, 2001.
15. E. Letier and A. van Lamsweerde. Reasoning about partial goal satisfaction for requirements and design engineering. *SIGSOFT Softw. Eng. Notes*, 29(6):53–62, 2004.
16. Object Management Group. Model Driven Architecture (MDA). http://www.omg.org/docs/ormsc/01-07-01.pdf, July 2001.
17. M. J. Osborne and A. Rubinstein. *A Course in Game Theory*. MIT Press, 1994.
18. J. Peer. Web Service Composition as AI Planning – a Survey. Technical report, University of St. Gallen, 2005.
19. S. Roach and J. Baalen. Automated procedure construction for deductive synthesis. *Automated Software Engineering*, 12(4):393–414, October 2005.
20. R. K. Runde and K. Stølen. What is model driven architecture? Technical Report UIO-IFI-RR304, Department of Informatics, University of Oslo, March 2003.
21. I. Sommerville. *Software engineering (7th ed.)*. Addison-Wesley, 2004.
22. E. Tardos. Network games. In *Proceedings of the Annual ACM Symposium on Theory of Computing*, 2004.
23. A. van Lamsweerde. Requirements engineering in the year 00: a research perspective. In *Proc. of ICSE'00*, pages 5–19. ACM, 2000.
24. A. van Lamsweerde. From system goals to software architecture. In *SFM*, pages 25–43, 2003.
25. D. S. Weld. Recent Advances in AI Planning. *AI Magazine*, 20(2):93–123, 1999.
26. E. S.-K. Yu. *Modelling strategic relationships for process reengineering*. PhD thesis, University of Toronto, 1996.