Master Thesis

A Simulation Framework for Self-Reconfigurable Socio-Technical Systems

Minh Sang Tran Le

Supervisors

Prof. Gerhard Lakemeyer
RWTH Aachen University
Germany

Prof. Paolo Giorgini
University of Trento
Italy

European Master in Informatics
October 16, 2009

Abstract

Software systems are becoming an integral part of everyday life influencing organizational and social activities. This aggravates the need for a socio-technical perspective for requirements engineering, which allows for modeling and analyzing the composition and interaction of hardware and software components with human and organizational actors.

Socio-technical systems (STS), as opposed to the traditional technical computer base systems, include human agents as an integral part of their structure. One important aspect of an STS is its dynamicity: an STS operates in a continuous evolving environment and, accordingly, its structure changes dynamically. Unlike the technical based system, an STS has the knowledge of how the system should be used to achieve some organizational objectives, and is normally regulated and constrained by internal organizational rules, external laws and regulations.

In this setting, the thesis aims at developing a framework supporting a system able to self-configure, which is to evolve dynamically in response to changes in its environment. A runtime reconfiguration mechanism will be based on AI planning for generating possible system configurations. In particular, the thesis task is to provide a framework that takes an initial organizational structure, and explores the organizational solution space with the help of AI planning technique. Found candidate solutions are simulated with respect to user-defined set of events to evaluate how these solutions adapt to environment changes. Moreover, these solutions are assessed by quantitative evaluators which are accompanied with the framework as well as user-defined ones. Assessment results are visualized to end-user in tree-structure, table, chart to help for a wise decision.

Acknowledgements

I would like to thank all of the people who have provided guidance to me in this thesis work. Particularly, special thank to my supervisors in Trento (Dr. Giorgini), and Aachen (Prof. Lakemeyer), for giving insight on the workings of my project, and for the main discussions on the topic, especially for providing the flexibility to modify the directions when needed. Thanks to Yudis Asnar for his instruction; to Vohla Bryl for dedicating time to answer all my questions; to Fabiano Dalpiaz for his interest, feedback and discussion. Finally, thanks to those who encouraged me along the way.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

Minh Sang Tran Le

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Traditionally, a software system falls into two categories: technical computer-base systems and socio-technical systems [34]. Technical computer-based systems are systems that include hardware and software components but not procedures and processes. According to the 2001 survey [23], only 28% of industry projects are reported as successful, meanwhile 49% of these were challenged, and 23% failed altogether. The success of a project refers to its ability to meet stake-holder's expectation, without going over time and budget constraints. These projects failed because they do not recognize the social and organizational complexity of the environment in which the systems are deployed. The consequences of this are unstable requirements, poor systems design and user interfaces that are inefficient and ineffective [2]. Also in [2], the authors highlight socio-technical systems approaches as a solution.

Socio-technical systems (STS) include one or more technical systems but, crucially, also include knowledge of how the system should be used to achieve some broader objectives. The socio-technical system include in their architecture an operational organizational structure and human actors along with software components. These factors are normally regulated and constrained by internal organizational rules, business processes, external laws and regulations [34]. This raises new challenges related to analyzing and designing of a STS. In particular, in a STS, human, organizational and software actors rely heavily on each other in order to fulfill their respective objectives. Hence, one important element in the design process is to identify list of actors and dependencies among them which, if respected by all partners, will fulfill all stakeholder goals, a golden key to the success of the STS.

KAOS [16] is a requirement elicitation technique that starts with stakeholder goals and derives functional requirements for a system-to-be and a set of leaf-goal assignments to external actors through a systematic, tool-supported process. However, KAOS does not explore the space of alternative assignments. Consequently, designers have no option to decide what an "optimal" assignment is.

Recently, work in [9] has proposed an approach to fill in the gap. The approach is inspired by Tropos [5], an agent-oriented software engineering methodology, which uses the i* modeling framework [39] to guide and support the system development process starting from requirements analysis to implementation. This approach constructs a space of assignment configurations by employing an off-the-shelf AI planner and does evaluation on these assignments to help identifying an optimal design. Particularly, the authors address the following problem: given a set of actors, goals, capabilities, and social dependencies, there is a tool generating alternative configurations which are goal-to-actor assignments and dependencies network among actors. The next step is to evaluate alternatives by assessing and comparing them with respect to a number of criteria provided by the designer.

Another important aspect of an STS is its dynamicity. The continuous evolving working environment of an STS makes its structure change dynamicity. This attracts a lot of effort of community to address the problem of dynamic configuration and adaptation of software systems. In [11], the authors try to adjust the existing agent-oriented methodologies. Work in [3] aims at developing an adaptive agent framework (need more detail). Another approach presented in [7] mixes the idea of AI planning technique with the Tropos methodology to introduce a general architecture that supports dynamic self-reconfiguration at runtime for STS.

We are interested in both supporting the design of STS and self-configurability of an STS. We realize that one most important aspect of our problem is to have a good enough solution space and a quantitative evaluator assessing the assignment solutions so that the designers or the software systems can make decision per se. Bryl et al [7, 9] present a framework addressing the problem. However, there is still a gap from the theory to the reality. Inspired by the ideas of Bryl et al, this work aims at constructing an open framework that generates goal-to-actor assignment and dependencies configuration (hereafter configuration or solution) and performs assessment on these solutions based on predefined or user criteria.

Contribution The purpose of our work targets to an open framework that supports both configuration generation and evaluation. Our framework inherits and extends that of Bryl. Specifically, the following highlights the difference between Bryl's and ours:

- Early optimization criteria are embedded into the PDDL script in order to prevent the solution space from explosion.

- Solution assessment could be extended at runtime. Users or third-party developers can develop and use their own evaluators in just some simple steps.
- An event-based simulation performs on each solution to assess the adaptation of solution at runtime with respect to a set of events. Users are able to reuse a set of predefined events or to define their own events.

Organization This document is organized as follows:

- *Chapter 1*: this chapter which give introduction to our work.
- *Chapter 2*: presents preliminary knowledge about goal oriented methodology which is adopted to design socio-technical systems, and the Planning Domain Definition Language (PDDL).
- *Chapter 3*: give a glance view about work relevant to self-reconfiguration STS and our framework.
- *Chapter 4*: discusses about the approaches of self-reconfiguration STS. It also present the position of the framework in the general basic structure of a centralized self-reconfiguration STS.
- *Chapter 5*: presents the general architecture of the framework.
- *Chapter 6*: provides detail information about the organizational descriptor model, which is part of the architecture.
- *Chapter 7*: present about the planning problem in both organizational level and instance level.
- *Chapter 8*: discusses about the two kinds of solution assessment, and present some concrete evaluators used in the framework.
- *Chapter 9*: briefly discuss about the prototype of the framework.
- *Chapter 10*: presents the experiment of the framework and the prototype.
- *Chapter 11*: concludes the thesis and raise some future works.

Chapter 2

Background

2.1 Goal-Oriented Methodology

The notion of *goal* is increasingly being used in requirements engineering (RE) methods and techniques today. Goals have been introduced into RE for a variety of reasons - within different RE activities, and to achieve different objectives. Goal-oriented Requirement Language (GRL) [26] is a language for supporting goal-oriented modeling and reasoning of requirements, especially for dealing with non-functional requirements. It provides constructs for expressing various types of concepts that are useful for supporting the requirements and high-level design process. There are three main categories of concepts: *intentional elements, intentional relationships*, and *actors*. GRL *elements* and *relationships* are intentional in that they are used in models that answer questions about intents, motivations and rationales, such as why particular behaviors, informational and structural aspects were chosen to be included in the system requirement, what alternatives to be considered, what criteria were used to deliberate among alternative options, and what the reasons were for choosing one alternative over others. The major elements and relationships are depicted in figure 2.1.

The *intentional elements* in GRL are goal (hard goal and soft goal), task, resource, and belief.

- *Goal* represents stakeholder objectives (or, requirements) for the new system and its operating environment, for example, a library organization has the goal of 'fulfill every book request', or Apple Company has the goal of 'produce 1M Mac5Gs within a year'.

Goals are classified into hard goals and soft goals. Hard goals (or functional goals, or just plain goals) have clear-cut criteria for their fulfillment whereas soft goals (or qualities) are used to define non-functional requirements and do not have clear-cut criteria for their fulfillment. In GRL notation, hard goal is represented by a rounded rectangle and soft goal is represented by irregular curvilinear shape with the goal name inside.

- *Task* specifies a particular way to accomplish goal. In GRL notation taks is a hexagon with the task name inside.
- *Resource* is a physical or information object that is available for use in the task. Resource is represented in GRL as rectangle.
- *Belief* is used to represent assumptions and relevant conditions. This construct is represented as ellipse in GRL notation.

The *intentional relationships* include decomposition, mean-ends, contribution, correlation and dependency.

- *Decomposition* defines sub-components, for example, a goal can be decomposed into subgoals. There are two kinds of decomposition: AND-decomposition and OR-decomposition. An AND-goal is satisfied if all of its subgoals are; and OR-goal is satisfied if at least one its subgoal is.
- *Mean-ends* relationship shows how the goal can be achieved. For example it can be used to connect task to a goal, meaning that it is a possible way of achieving the goal.
- *Contribution* relationship describes how one element influences another one. A contribution can be negative or positive and can be of different extents. Accordingly, contribution types include: help (+)/ hurt (-): one goal contributes positively/negatively towards the fulfillment of another goal, make (++), break (–): one goal subsumes/negates another.
- *Correlation* relationship describes side effects of existence of one element to others.
- *Dependency* relationship describes interdependences between agents.

An *actor* is an active entity that carries out actions to achieve its goal. In GRL notation actor is represented as a circle with the actor name inside.

An *agent* is an actor with concrete, physical manifestions, such as a human individual or a machine.

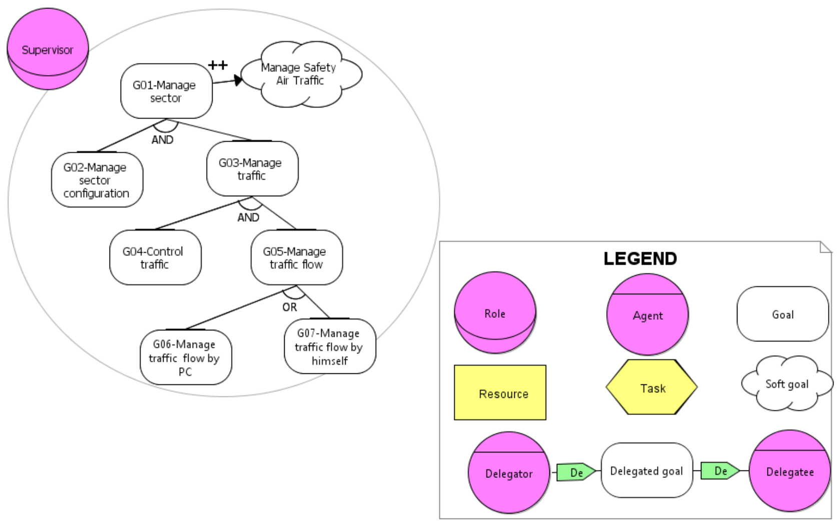


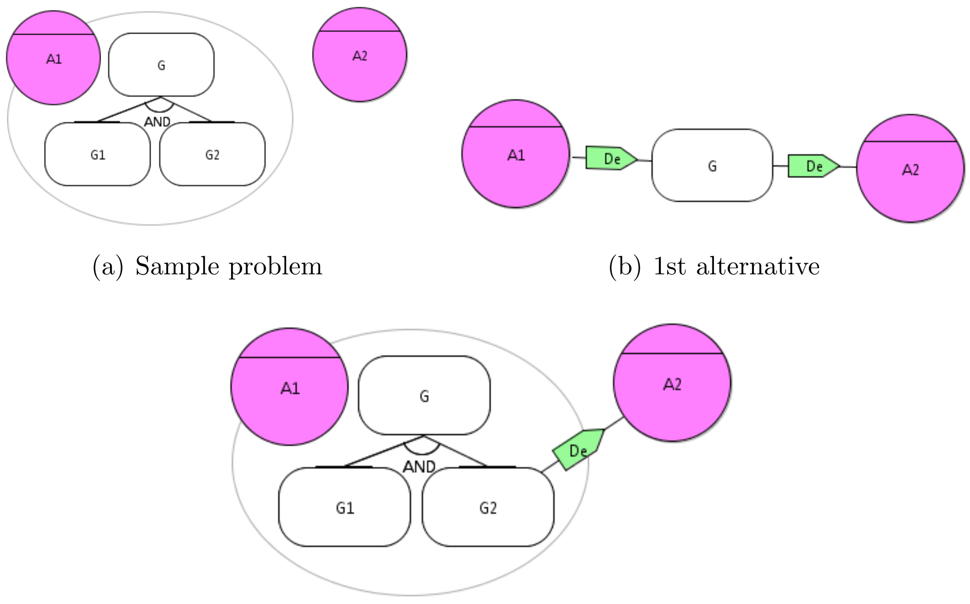
Figure 2.1: Major elements and relationships in GRL.

Alternatives for Satisfying Goals. An *alternative* (solution) to the fulfillment of a goal G consists of one or more leaf goals which together fulfill the root goal. A goal model defines a space of alternatives for the fulfillment of its root goal. Given a set of root goals and soft goals, the space of alternatives can be large and the criteria to evaluate solutions may also differ. For example, and alternative A1 is better than A2 in fulfilling goal G with respect to soft goal G1, G2... if A1's net contributions to G1, G2,... (e.g., positive minus negative contributions) is greater than that of A2. Another example of alternative is illustrated in figure 2.2 where A1 can decide to delegate to A2 all of G (figure 2.2(b)), or a part of it (figure 2.2(c)).

In general, goals and soft goals can be contradictory; hence, there may not be an optimal solution but the search for *good-enough solutions*. As our problem defined in section 1, our framework contributes to have a good enough solution space and quantitative evaluator assessing the assignment solutions so that the system designers can make decision per se.

2.2 PDDL

PDDL is an action-centered language, inspired by the well-known STRIPS formulations of planning problems. At its core is a simple standardization of the syntax for expressing this familiar semantics of actions, using pre- and post-conditions to describe the applicability and effects of actions. The syntax is inspired by Lisp; so much of the structure of a domain description is a Lisp-



(a) Sample problem

(b) 1st alternative

(c) 2nd alternative

Figure 2.2: Sample problem: two alternatives

like list of parenthesized expressions. A PDDL planning task is a combination of following components:

- *Objects*: things in the world correlated with the planning problem.
- *Predicates*: properties of objects that we are interested in, which can be true or false.
- *Initial state*: the state of the world that we are in.
- *Goal specification*: the desired objectives that we want to be true.
- *Actions/Operators*: ways of changing the state of the world.

Planning tasks specified in PDDL are separated into two files called *domain file* and *problem file*. The *domain file* holds *predicates* and *actions*, while *problem file* holds *objects*, *initial state* and *goal specification*. The structure of these files are depicted in figure 2.3

Example 2.1. Let us consider a concrete planning example. There is a robot that can move between two rooms and pick up or drop balls with either of his two arms. Initially, all balls and the robot are in the first room. We want the balls to be in the second room. The table 2.1 summarizes components of this planning problem.

(**define** (**domain** $\langle domain - name \rangle$)
    (**:requirements** $\langle : req_1 \rangle \ldots \langle : req_n \rangle$)
    (**:types**
            $\langle subtype_{11} \rangle \ldots \langle subtype_{1n} \rangle - \langle type_1 \rangle$
            ...
            $\langle subtype_{n1} \rangle \ldots \langle subtype_{nn} \rangle - \langle type_n \rangle$)
    (**:constants** $\langle cons_1 \rangle \ldots \langle cons_n \rangle$)
    (**:predicates** $\langle pred_1 \rangle \langle pred_2 \rangle \ldots \langle pred_n \rangle$)
    (**:action** $\langle action - name_1 \rangle \langle action\ body \rangle$)
    ...
    (**:action** $\langle action - name_2 \rangle \langle action\ body \rangle$)
)

(a)

(**define** (**problem** $\langle problem - name \rangle$)
    (**:domain** $\langle domain - name \rangle$)
    (**:objects** $\langle obj_1 \rangle \ldots \langle obj_n \rangle$)
    (**:init**
            $\langle PDDL\ code\ for\ initial\ states \rangle$
)
    (**:goal**
            $\langle PDDL\ code\ for\ goal\ specification \rangle$
)
)

(b)

Figure 2.3: The PDDL domain file (a) and PDDL problem file (b)

While predicates, initial states, and goal specification can be represented using first-order logic, the actions are described by using precondition and effect as illustrated in table 2.2.

The following is a concrete example of the domain file and problem file encoded in PDDL code:

```
(define (domain gripper-strips)
    (:predicates (room ?r) (ball ?b) (gripper ?g) (at-robby ?r)
        (at ?b ?r) (free ?g) (carry ?o ?g))
    (:action move
        :parameters (?from ?to)
        :precondition (and (room ?from) (room ?to) (at-robby ?from))
        :effect (and (at-robby ?to) (not (at-robby ?from))))
    (:action pick
        :parameters (?obj ?room ?gripper)
        :precondition (and (ball ?obj) (room ?room) (gripper ?gripper)
    (at ?obj ?room) (at-robby ?room) (free ?gripper))
        :effect (and (carry ?obj ?gripper) (not (at ?obj ?room))
                (not (free ?gripper))))
    (:action drop
        :parameters (?obj ?room ?gripper)
        :precondition (and (ball ?obj) (room ?room) (gripper ?gripper)
            (carry ?obj ?gripper) (at-robby ?room))
        :effect (and (at ?obj ?room) (free ?gripper)
            (not (carry ?obj ?gripper))))
)

(define (problem strips-gripper2)
    (:domain gripper-strips)
    (:objects rooma roomb ball1 ball2 left right)
    (:init  (room rooma)
            (room roomb)
            (ball ball1)
            (ball ball2)
            (gripper left)
            (gripper right)
            (at-robby rooma)
            (free left)
            (free right)
            (at ball1 rooma)
            (at ball2 rooma))
    (:goal (at ball1 roomb)))
```

■

From the very first version 1.0 in 1998, PDDL is evolved with each International Planning Competitions. PDDL 2.1 was created in 2002 by Maria Fox and Derek Long for IPC3, adding main features of *fluents* (or functions), plan quality measures (or *metrics*), and durative actions. Fluents are predicates that allow handling of numeric values and can be uses in actions preconditions (with relational operators: `>, <, <=, >=`) and effects; value is given in problem file. Metrics are the use of fluents to allow defining plan quality measures different from typical ones (time, number of actions). Actions can have a duration which can be given as a numeric value, an interval, or be calculated using fluents.

PDDL 2.2 was created by Stefan Edelkamp, Jrg Hoffmann and Michael Littman in 2004 for IPC4. PDDL 2.2 introduced *derived predicates* and *timed initial literals.* Derived predicates are predicates that are not affected by any of the action available to the planner; actually it is already at PDDL 1.0 as axioms. Timed initial literals are facts that will become true or false at time points that are known in advance.

Objects	The two rooms, four balls and two robot arms
Predicates	`room(x)` - true iff x is a room `ball(x)` - true iff x is a ball `gripper(x)` - true iff x is a gripper (robot arm) `at-robby(x)` - true iff x is a room and the robot is in x `at-ball(x, y)` - true iff x is a ball, y is a room, and x is in y `free(x)` - true iff x is a gripper and x does not hold a ball `carry(x, y)` - true iff x is a gripper, y is a ball, and x holds y
Initial state	All balls and the robot are in the first room. All robot arms are empty. [and so on]
Goal specification	All balls must be in the second room.
Actions/Operators	`MOVE(x, y)` - the robot moves from room x to room y. `PICKUP(x, y, z)` - the robot picks up ball x in room y by arm z. `DROP(x, y, z)` - the robot drops ball x in room y from arm z.

Table 2.1: PDDL elements in the robot example

MOVE(x, y)
precondition: $room(x) \wedge room(y) \wedge at - robby(x)$ **effect**: $at - robby(y) \wedge \neg(at - robby(x))$

PICKUP(x, y, z)
precondition: $ball(x) \wedge room(y) \wedge gripper(x) \wedge at - ball(x, y) \wedge at - robby(y) \wedge free(z)$ **effect**: $carry(z, x) \wedge \neg(at - ball(x, y)) \wedge \neg(free(z))$

DROP(x, y, z)
precondition: $ball(x) \wedge room(y) \wedge gripper(z) \wedge carray(z, x) \wedge at - robby(y)$ **effect**: $at - ball(x, y) \wedge free(z) \wedge \neg(carry(z, x))$

Table 2.2: PDDL actions for planing problem in example 1

PDDL 3.0 was developed Alfonso Gerevini and Derek Long for IPC5 (ICAPS 2006). The motivation is to introduce goal preferences and state trajectory contraints into PDDL, focusing on quality instead of on planning time or plan length (new for propositional domains). State trajectory contraints express a goal that can be seen as intermediate, i.e, goal which can be met not at the end but at certain moments of the plan. Since both goals and constraints must be accomplished for the plan being valid, preferences are introduced and applicable to goals or constraints. Preferences can be given a weight to establish which is more important, and hence, lead to plan quality. For example, two plans, one accomplishing preferences and other now, are both valid but first is better than second.

PDDL 3.1 was introduced at IPC6 (ICAPS 2008). PDDL 3.1 introduces state variables which are neither binary (true/false) nor numeric (real-valued), but instead map to a finite domain. This is achieved by adding object fluents to the language, which are analogous to the numeric fluents introduced in PDDL 2.1. Where numeric fluents map a tuple of objects to a number, object fluents map a tuple of objects to an object of the problem. In addition, PDDL 3.1 also introduces a new requirement for specifying numeric action costs in language fragments that do not normally permit numerical features.

Chapter 3

Related Work

The field of socio-technical systems emerged with the early work of Trist and Emery of the Tavistock Institute [21, 20, 36], can be view from two different complementary perspectives: a social sciences [21, 30, 37] and a design or engineering sciences [33, 34]. Social sciences study psychological, managerial, and organizational aspects of a socio-technical phenomenon. For example, they are interested in relationships inside workgroups, roles, supervision, motivation and the like. Differently, the latter perspective focuses on the design of socio-technical systems given sets of requirements, as well as their system properties, leading to important design principles, such as Chern's [12] and Pava's extension to advanced information technologies [36]. One important development has involved the work of Majchrzak and Gasser [27] on the design of TOP Modeller, a well tested tool and framework for designing, and analyzing, a socio-technical system in several diffrent problem domains. Unfortunately this work is outdated.

Socio-technical Systems Design (STSD) is a research paradigm for modeling the social and technical aspects of organizations [31]. Methods for designing socio-technical systems have been studied and proposed (with varied results) for decades, see [2] for an overview. Unlike similar work in organizational theory that have crossed over into computer science (business processes is one popular example), it has been difficult for such systems to become adopted in the software engineering community. This is primarily because of very wide interpretations of the subject. Baxter [2] summarizes these problems, namely inconsistent terminology in defining the technical and social systems, difficulties in finding proper levels of abstraction, conflicting value systems, lack of agreement on success criteria, lack of synthesis of ideas, multidisciplinary of the field, failure to keep up with current advances, and problems of defining details needed in fieldwork. All these have made STSD less attractive to the software engineering community, but while these present considerable obstacles, the field is regaining attention (eg. [31, 13, 10, 29, 25]. "There is still a role for humanistic, socio-technical ideas" that give "equal focus to the employee as to the non-human system," [2]. In answer to this problem, recent work has proposed to develop the discipline by using STSD throughout the systems engineering life cycle [2, 31].

Goal -based requirements modeling for socio-technical systems and organizations has been a topic of considerable research interest during the last fifteen years [49]. In [6], Bryl et al proposed a planing-based extension of Tropos [5], which is an agent-oriented software engineering based on goal methodology, to support designer in exploring the space of alternative designs of a socio-technical system, leading to automating the design. The fulfillment of each of the system goals is related to a number of choices of how the goal is decomposed and which are the actors the goals are delegated to; hence, alternative designs can be derived and evaluated in accordance with criteria that measure their effectiveness.

The work of Bryl et al [6] employed AI (Artificial Intelligence) planning [38] techniques and use PDDL (Planning Domain Definition Language) 2.2 [18] to formally specify the initial organizational setting and actions of the domain. The task was framed as a planning problem where selecting a suitable social structure corresponds to selecting a plan that satisfies the stakeholders goals. A basic set of first-order predicates has be presented to formalize the organizational setting in terms of actors and goals, their properties (e.g. actor capabilities), and social dependencies among actors. An off-the-shelf planning tool, LPG-td [28], is adopted for the implementation of the planning domain. Plan metrics uses here are the global and local evaluation based on complexity (e.g, cost and duration) of an actor in satisfying a goal [7, 9].

The above framework is then extended in [1], uses risk-based evaluation metrics for selecting a suitable design alternative, and aims at agent-based safety critical applications. In this work [1], the risk-based criteria (e.g. related to the criticality of a goal satisfaction or minimum acceptable level of trust between agents) is incorporated into the planning-based procedure which supports a socio-system design (as well as a system redesign at runtime). This work aims at proposing a design that maintains the risk level within the acceptable limits.

Yet another extension of the framework proposed the use of location as a metrics for evaluating alternatives [14]. Due to the fact that location properties associated to variation points can be used to (a) limit the range of alternatives an agent can choose among; (b) express location-dependent contribution to soft-goals, the authors suggest a planning-based approach that can be customized to discard unavailable options and to exploit softgoal satisfaction for ranking availble alternatives.

Nevertheless, design-time support is not sufficient to provide a comprehensive support for socio-technical systems. Runtime violation of requirements [24] is recognized as an open problem and has been explored since several years. Feather et al [22] propose an approach to reconcile requirements with runtime behavior, where both the design- and run-time phases are covered: (a) anticipate as many violations as possible at specification time, and (b) detect and resolve the remaining violations at runtime. Though Feather's work is not targeted for socio-technical systems, it points out problems and proposes solutions which apply also in the context of STSs. In [7], Bryl et al presented a *centralized self-reconfiguration* mechanism to dynamic reconfiguration of a socio-technical system structure in response to internal or external changes. The approach is the same as [6] with the reconfiguration mechanism added. The evaluation on the both system state will decide whether the system needs to be redesigned in response to external or internal change. And, if the above evaluation shows that the reconfiguration is need, replans the system structure would be done. The work gives the idea; however, the set of event is still in preliminary

Another and complementary approach is *decentralized self-reconfiguration* mechanism. Talos [14, 15] is an architectural for self-reconfiguration where self-reconfiguration is seen from the perspective of each agent/component. Three different subsystems are the core of the self-reconfiguration process each agent performs: *Monitor*, *Diagnose*, and *Compensate*. The agent should continuously monitor both its internal state and the location where it is running. Triggering events are then diagnosed. If there is failure, a compensate plan should be executed to "undo" the effects of the failed plan.

Our work focuses on extending the work in [7, 9]. Local evaluation is no longer exists but early encoded into the planning problem, hence, limit the range of the alternatives/solutions space that an agent can choose among. Evaluation could also be extended at runtime by adding new evaluators from independent developers. Due to the fact that the best solution at the first round planning may be the worst after triggering events, an event-based simulator performs on each solution to see its adaptation at runtime.

Chapter 4

Self Reconfigurable Socio-Technical Systems

4.1 Socio-Technical Systems Design

According to Tropos methodology, the process of modeling and analyzing requirements for STS is the following. Starting from the stakeholder's request, the designer identifies goals and actors for the system to-be. The designer refines the goals and relations between the actors goals to provide the detail specification of the system. In particular, the designer constructs a goal model in which each of top goals is decomposed into AND- or OR-subgoals. Goals are decomposed from a very high abstract level to concrete tasks that can be assigned to human actors of software components. When being in charge of a goal, an actor can continue decompose this goal, or fulfills it if it is a concrete task, or delegates it to another actor. Obviously, there are more than one way to decompose a goal, and assign goals to a actor as well. It produces many alternatives for satisfying original top goals. The STS designer faces to a question which is *what is an optimal or, at least, good enough assignment of leaf goals to actors so that to root goal is satisfied when all actors deliver on the goals assigned to them?* Such a question is the problem of exploring the space of alternative delegation and assignment network. Each of alternatives then is evaluated in particular criteria for the comparison.

To address this problem, Bryl et al [9] propose a requirements engineering approach that complements the Tropos requirements analysis and design process. Then general schema of the approach is illustrated in figure 4.1.

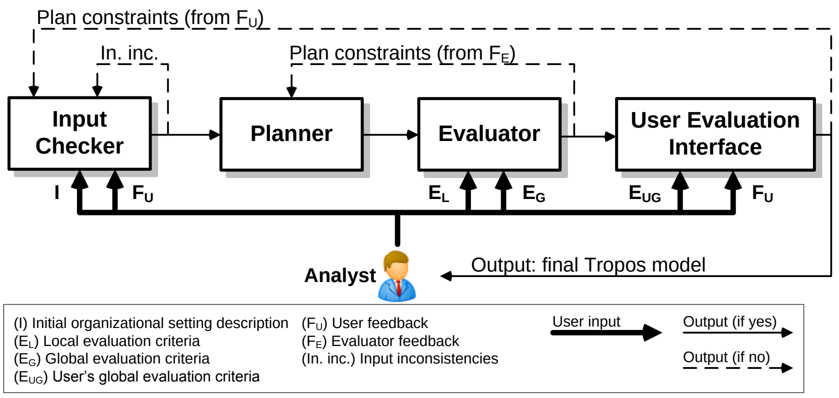

Plan constraints (from F_U)
In. inc.
Plan constraints (from F_E)
Input Checker
Planner
Evaluator
User Evaluation Interface
I
F_U
E_L
E_G
E_UG
F_U
Analyst
Output: final Tropos model
(I) Initial organizational setting description
(E_L) Local evaluation criteria
(E_G) Global evaluation criteria
(E_UG) User's global evaluation criteria
(F_U) User feedback
(F_E) Evaluator feedback
(In. inc.) Input inconsistencies
User input
Output (if yes)
Output (if no)

Figure 4.1: Requirement analysis process: a general schema.

As a first step, the **Input checker** analyzes the organizational setting description, detects inconsistencies, and proposes possible improvements, when then are approved, or rejected, or modified by the designer. After the input is checked, the first possible alternative is generated by the **Planner**. The alternative then is assessed by the **Evaluator** with respect to a number of criteria. The final output of the evaluation process needs to be approved by a human designer. This is done thank to the **User evaluation interface** which presented selected alternative to the designer together with the summarized evaluation results. The designer can make feedback to the alternative, which might initiate a new iteration. The process is completed when the designer is satisfied with the alternative.

4.2 STS: Self-Reconfiguration at Runtime

At runtime, the system organizational structure is not static but continuously evolving. One of the specific properties of an STS is the dynamic organizational objectives[34], because objectives can have different (subjective) interpretations and may vary over time. Thus, the design-time support is not sufficient to provide a comprehensive support for socio-technical systems. Runtime violation of requirement is recognized as an open problem and has been explored since several years. There is hence a clear need for a mechanism which complements the design-time techniques we have presented. In which, the STS knows about the changes in its operation environment then perform self-configuring according to the changes with or without the human administrator.

Currently, there are two different but complementary approaches can drive the self-reconfiguration process [14, 7]:

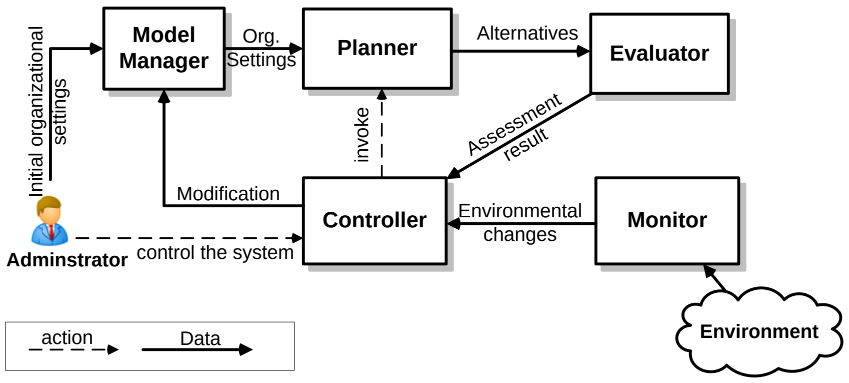


Figure 4.2: Basic settings of a centralized runtime self-reconfigurable STS.

- *Centralized self-configuration*: the self-configurable STS has a centralize knowledge of the various agents and therefore the reconfiguration process is controlled centrally.
- *Decentralized self-configuration*: this approach present self-reconfiguration form the local perspective of an individual agent. Each agent commits to achieve its own goals at best, without having a complete knowledge of the STS. This approach cannot achieve the same level of optimality a centralized approach guarantees, but it the only available solution whenever the agents do not want to disclose their internal structure to a centralized supervisor.

We focus on the centralized approach whose general architecture, depicted in figure 4.2, should include following component:

- *Model Manager*: manages the organizational settings for the system. These settings include the initial organizational settings, the effect to this model according to environmental changes over time.
- *Planner*: generates organizational alternatives using stored organizational settings and environmental effect in the *Model Manager*.
- *Evaluator*: assesses generated alternatives to support decision in selecting alternatives.
- *Monitor*: monitors the environment for changes and reports them to the *Controller*.
- *Controller*: is the coordinator of the STS. The *Controller* invokes the *Planner* for generating alternatives, and then passes them to the *Evaluator*. The *Controller* is also in charge of selected active alternatives

(with/or without the interactive of user), and applies the selected alternative for the STS. More important, the *Controller* receives notification from the *Monitor*, performs appropriate modification to the organizational settings and invoke, if necessary, the self-configuration process.

In this work, we aim at constructing a framework supporting solution evaluation for an STS. Particularly, this framework plays the role of the three components: *Model Manager, Planner* and *Evaluator*. The framework should be generic enough to be reused in many fields of the STSs. In the following chapters, we give a detail discussion about our proposed framework.

Chapter 5

Framework Architecture

In this chapter, we present the architecture of our proposed framework.

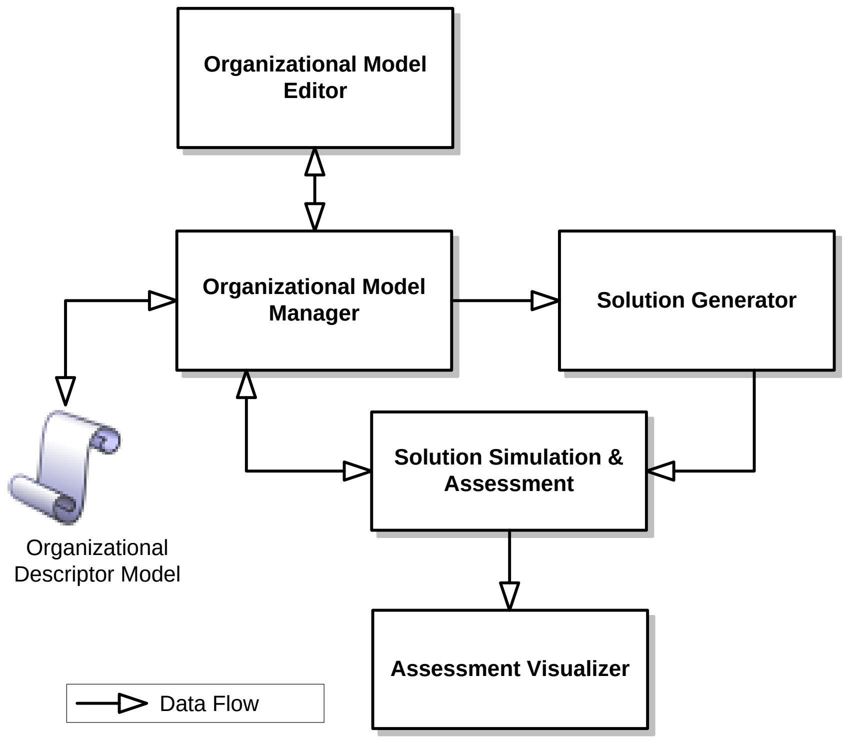

Organizational Model Editor
Organizational Model Manager
Solution Generator
Solution Simulation & Assessment
Organizational Descriptor Model
Assessment Visualizer
Data Flow

Figure 5.1: The general architecture of proposed framework

The general architecture of our proposed framework is illustrated in figure 5.1, which consists of following components:

- *Organizational Model Manager*: maintains the organizational descriptor model including the organizational model, evaluation model and event model.

- *Organizational Model Editor* This component provides a GUI that allows users to edit the descriptor model in a graphical way.
- *Solution Generator*: is in charge of generating a solution space. A solution is a goal-to-actor assignment scheme according to goals' decompositions and actors' capabilities. In this fashion, a solution is also called as configuration of the organizational model.
- *Solution Simulation & Assessment*: performs the vital task of the framework. This component takes each of solutions fed by Configuration generator, and carries out assortment. The output then is transferred to Assessment Visualization to support end-user to make decision on which solution shall be applied.
- *Assessment Visualizer*: takes the simulation result and displays it in a user-friendly look such as charts, diagram, and so on.

5.1 Organizational Model Manager and Editor

The *Organizational Model Manager* (OMM) holds the information loaded from the *organizational descriptor model* (ODM). This information is used to feed the *Configuration Generator* and the *Solution Assessment*. The OMM is also in charge of applying feedbacks to the organizational descriptor during assessment. All functions provided by the OMM can be easily accessed and reused the by the application. On the other hand, the OMM also provides an end-user interface to manipulate the ODM at runtime.

The ODM is the data model organizing data structure necessary for other parts of the framework. The ODM contains the information about the organizational model including list of goals, actors and dependency relationship between them. The ODM also maintains a list of registered evaluators and their configuration which is used to assess solutions for STS. Moreover, the ODM holds a description of the event model used by the the event-based simulator.

The detail of ODM will be discussed in chapter 6.

5.2 Solution Generator

The *Solution Generator* accesses the ODM managed in OMM and generates the solutions space which in turned used by the *Solution Assessment*. From the *potential organizational model*, the *Solution Generator* tries to generate solutions by assigning goals to appropriate actors according to their capabilities. Since a goal can be provided by many actors, there are many ways to do the assignment. For example, if we have 3 goals and 3 actors which can provide each of these goals, then we have $3! = 6$ possible combinations. The goal-to-actor assignment hence is a combinatorial problem which causes a terrible performance and computation issues if the number of goals and actors is just greater than 20.

Therefore, it is impossible to take one-by-one in the solution space and perform the evaluation to find which one is the best. The solution generator hence shows its important contribution to the success of the framework by filtering the solution space. Only considered-optimal solutions are fed to the *Solution Assessment* for evaluation.

Instead of building the generator from scratch, we employ a general AI planner . The general AI planner takes an input describing the planning problem in the Problem Domain Definition Language (PDDL). The PDDL is an attempt to standardize planning domain and problem description languages. It was developed mainly to make the 1998/2000 International Planning Competitions possible. It was first developed by Drew McDermott in 1998 and later evolved with each International Planning Competitions. The latest version of this language is PDDL3.1

In the figure 5.2, it is the general architecture of the *Solution Generator* component, which consists of four components:

- *PDDL Generator* extracts the *potential organizational model* from the ODM and creates a corresponding PDDL problem file.
- *Post PDDL processor* performs a late processing on the generated PDDL file. This component provides a mechanism to modify the PDDL file without affecting the ODM by maintaining a list of post-PDDL commands. A post-PDDL command is an introduction to add/remove a goal/fact from the generated PDDL script. Thank to this mechanism, the event simulator (later discussed in section 8.1.3) can embed state of the world of the broken solution into the re-planning process.
- *AI Planner* acts as a proxy between our framework and the external generic PDDL planner. The PDDL planner should be picked from the

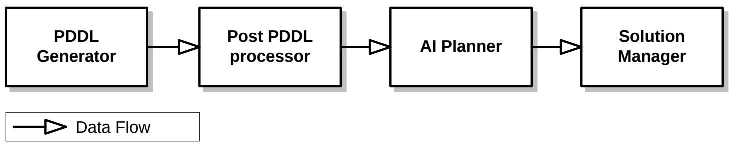


Figure 5.2: General architecture of Solution Generator.

list of participants of the International Competition Planning[1].

- *Solution Manager* collects and stored generated solutions for further assessment.

5.3 Solution Simulation, Assessment and Visualization

This component takes a solution from the *Solution Manager* to evaluate. The result then is displayed to end-user via a graphical ways such as tables, charts. There are two kinds of assessment considered in this framework: *scalar quantitative assessment* and *event-based quantitative assessment*. The quantitative assessment acts as a function of solution and returns a number representing the measurement of this solution. The returned measurement could be a set of numbers which explains how the final value is computed.

For example, consider Cost/Benefit evaluator which computes the benefit over cost quotient of a solution. This evaluator returns not only the quotient value, but also the values of benefit and cost.

The figure 5.3 illustrates the architecture of the *Solution Assessment* and *Visualization*, including following sub components:

- *Solution Evaluator* acts as a front-end proxy that receives assessment request and forwards to appropriate scalar evaluator or the event-base simulator.

- *Evaluator Registry* maintains a list of simple evaluators. Other components query the *Evaluator Registry* for an instance of a specific evaluator to carry out assessment.

[1]http://ipc.informatik.uni-freiburg.de/

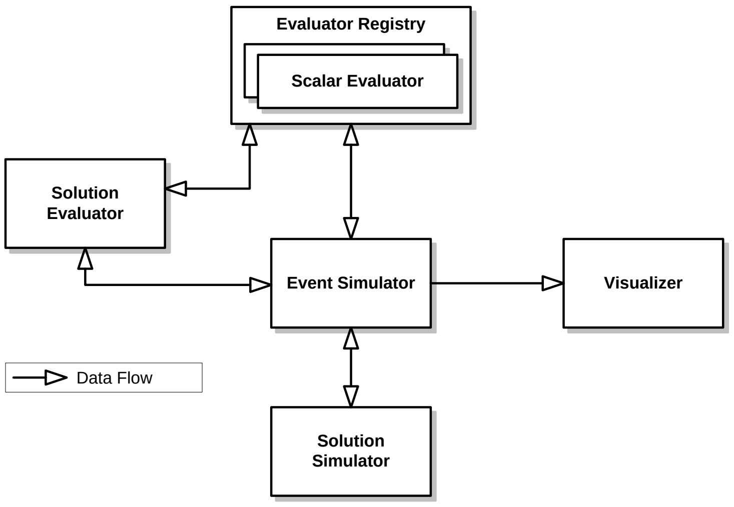


Figure 5.3: The architecture of the Solution Assessment and Visualization

- *Event Simulator* is in charge of simulating a solution in context of events. Given a solution with its corresponding ODM, the *Event Simulator* passes this solution to the *Solution Simulator*. For each time circle of the *Solution Simulator*, the *Event Simulator* checks the precondition of all active events in the Event Model of the ODM. If a precondition of an event is satisfied, this event is considered as happen and its post-conditions are executed.

- *Solution Simulator* simulates the execution of a given solution. The *Solution Simulator* is based on a time-based simulation algorithm (see algorithm 8.4) which executes solution's actions when their preconditions are matched.

- *Visualizer* gets the evaluation result and renders it to end-users. The result will be displayed in a table if the assessment result is from a scalar evaluator. Otherwise, the *Visualizer* will create a tree structure. The root will be the given solution, called solution node. When events happen while simulating this solution, a child node is created, called event node. If the solution is replanned, a special child node called branches is created, and for each new solution, it creates a child node for branches node. And the same logic is applied new solution nodes. The *Visualizer* labels solution node with the measurement value of a selected scalar evaluator, event node with the time when this event happens.

Chapter 6

Organizational Descriptor Model

The ODM includes *i*) the *potential organizational model* (see definition 1) describing the list of goals, actors and all possible capabilities of each actor as well as all possible dependencies among actors; *ii*) the *evaluation model* in which users can specify which evaluators should be used in assessment. This model also contains the evaluators' configuration that might be used to control the evaluators' behavior at runtime. Users can add evaluators from a predefined list and their custom evaluators; *iii*) the *event simulation model* allows users to define a list of events being used during the event-based simulation which will be shown up in the next section.

6.1 Potential Organizational Model

The *potential organizational model* maintains a list of goals, actors, dependency relationships that potentially generates organizational model. We formally define the potential organizational model as follows:

Definition 1 *The potential organizational model $\mathcal{O}$ is a tuple of $\langle$ A, G, Dc, Dp, Cap, $\mathcal{P}^d$, $\mathcal{P}^v\rangle$, in which*

- *A: is a set of actors,*

- *G: is a set of goals,*
- *$Dc \subseteq \{AND, OR\} \times G \times 2^G$: is a set of all possible decompositions for each goal in G. A $dc = \left\langle [AND|OR], g, 2^G \right\rangle \in Dc$ is an AND/OR decomposition of goal g into other sub-goals. Goals are recursively decomposed until operations which can be assigned to human actors of software components.*
- *$Dp \subseteq A \times A \times G$: is a set of dependency relationships between actors. A dependency relationship $dp = \langle a_1, a_2, g \rangle \in Dp$ presents a potential delegation which actor a_1 can delegate the satisfaction of goal g to a_2.*
- *$Cap \subseteq A \times G$: is a set of actors' capabilities which describes goals provided by each actor.*
- *$\mathcal{P}^d$: is a set of property descriptors. A property descriptor is triplet $\langle name, d\_type, def\_value \rangle$ which defines a property* name *of type* d_type *with default value* def_value. *Each property descriptor is associated with an actor, goal or capability.*
- *$\mathcal{P}^v : \{A \cup G \cup Cap\} \times \mathcal{P}^d \rightarrow value$: is a function mapping an actor, a goal, or a capability and a property descriptor to a scalar value. This value could be a* text, number, *or even reference to another object.*

The figure 6.1 is the class diagram of the *potential organizational model*. In this diagram, we focus on the relationship between entities; hence only important properties are displayed. The main entities in the diagram are `Actor` and `Goal` which represent actor and goal in the organizational model. The capability of each actor is captured in `Capability` showing which goals can be provided by this actor. In order to adapt with various domains, the properties of actor, goal and capability are constructed dynamically by inheriting these entities from a so-called `CustomizableObject`. Each `CustomizableObject` belongs to an `ObjectClass` which holds a list of `PropertyDescriptor`. Each `PropertyDescriptor` has name, data type and also the default value for a property. The `PropertyDescriptor` also specifies the PDDL function used to concert this property to the PDDL script. A `CustomizableObject` instance maintains its specific properties' value in a list of `PropertyValue`. For the ease of use of innocent users, all possible properties of actor, goal and capability are predefined by experts for specific application domain.

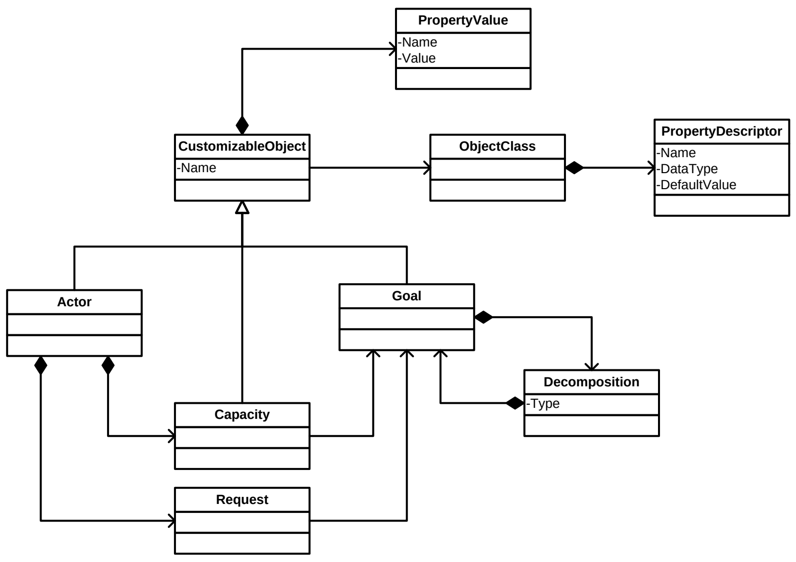


Figure 6.1: Potential organizational model class diagram

Beside, the `Decomposition` entity captures all possible decompositions of a goal. In the *potential organizational model*, a goal is decomposed until it reaches to particular operations which can be performed by human actors or software components.

6.2 Evaluation model

The next part of the ODM is the *evaluation model*. An evaluator is a tuple $\langle name, executor, parameters \rangle$. The evaluator *name* invokes the *executor* with *parameter* to carry out the assessment on a given solution.To this extend the evaluation model are defined as follows.

Definition 2 *An evaluation mode is a tuple* $\langle EV, EP \rangle$

- *EV: is a set of evaluators.*
- $EP \subset \{A \cup G \cup Cap\} \times \mathcal{P}^d$*: is a set of properties associated with actors, goals and capabilities. The properties are used by the evaluators to perform computation.*

The class diagram of the evaluator model is depicted in figure 6.2. In this model, users are able to declare which evaluator will be used to do the quan-

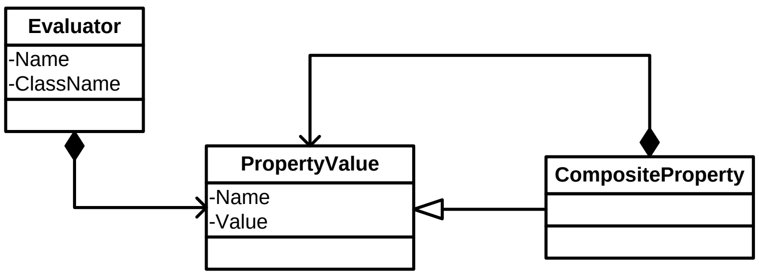


Figure 6.2: Evaluation model class diagram

titative evaluation. Each `Evaluator` contains the class name that implements the evaluator, and a list of options which will be passed to the implementation at runtime to configure its behavior. The implementation, certainly, has to implement a predefined interface called `IEvaluator`(see figure 8.1) in order to be understood by the system. More discussion about quantitative evaluations and others is provided in section 8.

6.3 Event Model

The final part of the ODM describes the event model which is used to carry out the event-based simulation. This part consists of list `EventSet` entities. Each `EventSet` has a name to distinguish itself with others; a Boolean property called `Enabled` to let the simulator know whenever to use this event set; and a list of event definitions. The event definition employs the idea of *Event Calculus* in which each event should be modeled as a triplet ⟨*precondition, post-condition, parameters*⟩.

The *event precondition* shows whenever the event should happen. It can be the absolute time while simulating a solution, a relative time correlated with other events i.e., after an action has been executed, or an event happens. Or it can the combination of primitive conditions.

On the other hand, the *event post-conditio*n describes the effect of this event to the model. This effect is a list of reaction. There are many types of reaction, e.g., a reaction that modifies the ODM at runtime (more particularly, the potential organizational model part) by changing actors/goals properties, introducing new goals, actors as well as capabilities. Or another reaction refines the solution-generating process. The framework supports not only a list of built-in actions (listed in table 8.2), but also custom reactions. The custom reaction is a special reaction which allows users to embed their own actions into the framework without rebuilding the source code.

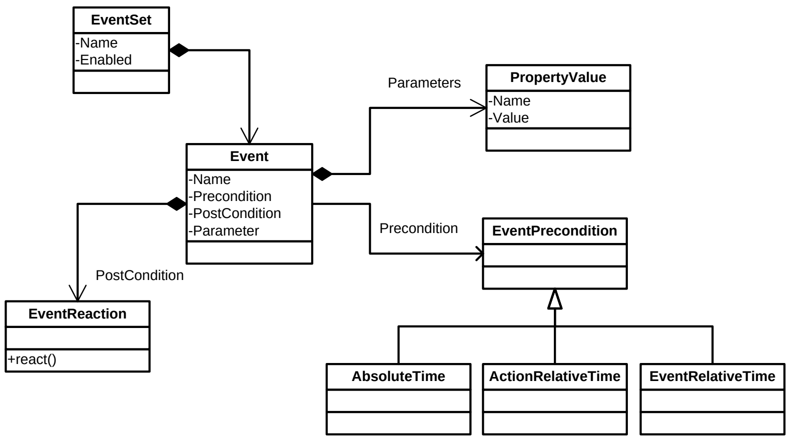


Figure 6.3: Event model class diagram

The *event parameters* are a list of values used by the reactions defined in the post-condition. These values are fed by user before the simulation. Or, they can be fed by the simulation engine base on the simulated environment at the moment events happen.

The event precondition and post-condition are depicted as `EventPrecondition` entity and `EventReaction` entity in the class diagram (see figure 6.3).

As modeling event in this fashion, user is able to define many events with different effects by combining reactions (predefined and custom) in different order. In section 8.1.2, we will discuss how events are used in our framework to carry out assessment.

Chapter 7

Socio-Network Planning Problem

7.1 Organizational level planning

The organization planning problem is to assign goals to role in the organizational level settings. We inherits work of [8] in organizational level planning, but we modifies action declaration as well as introduce some new functions in order to embed preliminary assessment criteria into the PDDL script.

7.1.1 Formalized input from ODM

Table 7.1 lists the predicates used to formalize the ODM. These predicates take variables of the following type: *goal*, *actor*, *agent* and *role* which correspond to the basic concepts of Tropos modeling notation. In our framework, we adopted predicates suffixed by (*) from [8] to formalize the ODM.

An *actor* is specialized into *agent* and *role*. However, at this level there is no need to distinguish between *agent* and *role*. Thus we use *actor* to specify input settings at this level.

The predefined ways of goal decomposition are presented using $and/or\_subgoal_n$ predicates. To capture the meaning of *goal conflict* and *goal prerequisite*, we uses predicate `conflict` and `require`. The goal A and goal B are considered to be conflict if the satisfaction of goal A negates the satisfaction of goal B, or

Goal property and dependency predicates
$and\_subgoal_n(g, g_1, g_2, ..., g_n : goal)$ (*)
$or\_subgoal_n(g, g_1, g_2, ..., g_n : goal)$ (*)
$conflict(g_1, g_2)$ (*)
$require(g_1, g_2)$
$means\_end(g_1, g_2)$ (*)
Actor property predicates
$request(a : actor, g : goal)$ (*)
$can\_provide(a : actor, g : goal)$ (*)
$can\_depend\_on(a_1, a_2 : actor)$ (*)
$can\_depen\_on\_g(a_1, a_2 : actor, g : goal)$ (*)
Actor property functions
$work\_sat(a : actor, g : goal) : int$
$work\_effort(a : actor, g : goal) : int$
$work\_time(a : actor, g : goal) : int$
$work\_max\_effort(a : actor, g : goal) : int$
Misc. functions
$total\_sat : int$
$total\_effort : int$

(*): predicates adopted from [9].

Table 7.1: PDDL predicate in organizational planning domain.

if goal A is satisfied then goal B is not able to be satisfied. This is probably a consequence of the sharing resources between A and B. The satisfaction of A consumes some not reusable resources that required by goal B. Hence there is no resource available for goal B. On the other hand, if the goal A is *prerequisite* of goal B, then before satisfying goal B, we should satisfy goal A.

Actor capabilities are described with `can_provide` predicate and `work_sat /work_effort/ work_time/work_max_effort` functions. While the former indicates that an actor has enough capability to satisfy a specific goal, the later predicates describe this capability more detail. Although many actors can provide a same goal, there are many differences between them about the time, the effort the actor has to pay, and the level of satisfaction of the goal. The last function, `work_max_effort`, determines the maximum efforts this actor could spend for fulfilling goals. It means that when an actor fulfills a goal, its availability is decreased by a certain amount. And it could not fulfill any goal when its availability moves to zero. This happens frequently in the real since the availability of an actor is not infinitive.

Possible dependencies among actors are specified the help of `can_depend_on` and `can_depend_on_g` predicates, which mean that an actor can delegate to another actor the fulfillment of any goal, or a specific goal, respectively.

The two last functions `total_sat` and `total_effort` are used to drive the planning process. Basically, the AI planner tries to optimize the solution such that the new solution is better than the old one. This "better" concept is depended on which criterion is used. Normally it is the length of the solution: *shorter is better*. But in most case, it makes no sense. Thus we can ask the planner to maximize the `total_sat`, or minimize the `total_effort`, or maximize the quotient of `total_sat` and `total_effort`.

7.1.2 Planning problem

As discussed in section 5.2, the task of building a socio-technical model can be framed as a planning problem: selecting a suitable set of delegations and assignment of goals to actors corresponds to selecting a plan that satisfies actors' and organizational goals. In general, AI planning is about automatically determining a list of actions needed to achieve a certain goal where an action is a transition rule form one state of the world to another. Actions are described in terms of preconditions and effect as discussed in section 4.2.1, which are in our case encoded in predicates/functions listed in table 7.1. In the domain of socio-technical system design, a plan (or solution) comprises the following actions, which are detailed in table 7.2:

$\mathbf{AND\_DECOMPOSE(a, g, g_1, \ldots, g_n)}$
precondition: $request(a, g) \land and\_decompose(g, g_1, \ldots, g_n)$ **effect**: $\neg request(a, g) \land request(a, g_1) \land \ldots \land request(a, g_n)$
$\mathbf{OR\_DECOMPOSE(a, g, g_1, \ldots, g_n)}$
precondition: $request(a, g) \land or\_decompose(g, g_1, \ldots, g_n)$ **effect**: $\neg request(a, g) \land request(a, g_1) \land \ldots \land request(a, g_n)$
$\mathbf{PASSES(a_1, a_2, g)}$
precondition: $request(a_1, g) \land (can\_depend\_on(a_1, a_2) \lor can\_depend\_on\_g(a_1, a_2, g))$ **effect**: $\neg request(a_1, g) \land request(a_2, g)$
$\mathbf{SATISFIES(a_1, g)}$
precondition: $request(a, g) \land can\_prodive(a, g) \land \neg satisfied(g)$ **effect**: $\neg request(a, g) \land satisfied(g)$

Table 7.2: Organizational planning actions' logic

- `AND/OR_DECOMPOSES`$(a : actor, g, g_1, , g_n : goal)$: actor a decomposes goal g into n AND/OR subgoals;
- `PASSES`$(a_1, a_2 : actor, g : goal)$: actor a_1 delegates goal g to actor a_2;
- `SATISFIES`$(a : actor, g : goal)$: goal g is assigned to and is satisfied by actor a.

7.2 Instance level planning

The organizational level planning, however, considers only the relation between roles and goals which is not able to directly apply in the real. Practically, *a role has as many instances as there are agents playing this role, and a goal has as many instances as there are agents who want this goal to be satisfied* [40]. There are some certain constraints can be expresses and considered only at the instance level, where a socio-technical model reflects the actual dependencies among agents playing roles and goal assignment to these agents [8]. They could be legal regulations (e.g., two activities cannot be performed by the same agent) or optimization concerns (e.g., overlapping assignments of goals to agents should be avoided). The work that decides which agents play which roles is called *instantiation*. And the process accomplishing instantiation is called *instance level planning*.

The instantiation comes up with some issues need to be clarified. They are how to instantiate and formalized roles and goals in the organizational model. The discussion about these questions is as follows.

Question 1. How to instantiate roles? Roles are instantiated to *agents* at instance-level. Answering this question is quite simple since we usually know the number of agents and their capabilities before planning. ■

Question 2. How to formalize agents? Agents can be described either as instances of roles, or separated objects. In the former case, declaring an agent is instance of a role (or agent can plan a role) automatically infers that this agent inherits all capabilities of this role. These capabilities might be different among agents of the same role. In the later case, agents are declared as separated entities which have their own capabilities. The planer (or user) has to classify agents to roles according to their abilities. Notice that in the first option, the planner/user might also have to assign agents to roles since an agent can plan more than a role. But, the nature of decision between two options is different. ■

Question 3. How to instantiate goals? The answer for this question is not as simple as the one with agents. At organizational level, a role requests a goal. But at instance level, there many agents play the same role, and it leads the fact that many agents might request many goals. This problem is known as goal instantiation problem. For the sake of clarity, goals at instance level are renamed as *desire*. Hereafter, the term *goal* refers to organizational level goal, and term *desire* refers to a goal at instance level.

At the organizational level, an actor $a$requests a goal g. The number of tasks of goal g are depended on the number of agents of actor a. Basically, we identify three different cases for the goal instantiation listed in table 7.3.

- *one-to-one*: the number of desires equal to the number of agents. For instance, the actor *Student* requests the goal *pass the Math exams*. Suppose that there are 100 students, then there are 100 desires to pass the exams.
- *many-to-one*: every agents of a role share the same desire e.g., in a fire fighting, every firemen have an ultimate goal which is to stop the fire.
- *many-to-many*: a group of agents share a same desire, but there are more than one group in the same roles. And the desires among groups are different. For example, there are 20 students in class, and they are divided into 4 groups. Each group shares a desire of accomplishing course's project. But different groups have different projects.

Case	**Description**
one-to-one	one agent yields one desire.
many-to-one	every agents share a same desire.
many-to-many	different groups of agents in the same role share different desires.

Table 7.3: The different cases of instantiating goal to desire.

	Static request	Non-static request
Static goal	1	1
Non-static goal	1	n

Table 7.4: The number of desires regards to the static/non-static characteristic of goal and request.

To cope with the two first situations, in the ODM, the goal has an extra property saying that goal is *static* or *non-static*. And the same property is also applied for a goal request of an actor. The static goal at organizational level will introduce only one desire at instance level under any circumstance. On the other hand, non-static goal might yield one or more desires with respect to the static or non-static request. The table 7.4 shows how we applies *static* and *non-static* notions in order to deal with goal instantiation problem. The third situation is the most complex one. Users have to specify desire for each specific group. ■

Question 4. How to formalize desires? This question has the similar meaning with question 1. And then we can apply the same reasoning logic. Desire can be formalized either base on goal, or as independent entities. ■

According to [8], there are two options for instantiation, which can start either from an organizational level solution, or from a formalized initial organizational setting. The former option needs predicate to describe an organizational level solution an, meanwhile, the later options does not. Both approaches also need additional predicates to model the agents and their properties. From this point of view, we called these two approaches as *direct* and *indirect instance level planning.*

7.2.1 Indirect instance level planning

This approach, as discussed, requires a solution from an organizational level planning. The task is to assign a certain number of agents to pre-planned roles as well as assigns specific goals of each role to agents playing this role. Table 7.5 lists predicates for instantiation.

Predicates capturing organizational solution
$and/or_decomposed(r : role, g, g1, \ldots, gn : goal)$
$passed(r1, r2 : role, g : goal)$
$assigned(r : role, g : goal)$
Predicate instantiating organizational solution
$can\_play(ag : agent, r : role)$
$desire_from(d : desire, g : goal)$
$desire\_assignment\_conflict\_agent(d1, d2 : desire, ag : agent)$
$desire\_assignment\_conflict(d1, d2 : desire)$
$role\_assignment\_conflict_agent(r1, r2 : role, ag : agent)$
$role\_assignment\_conflict(r1, r2 : role)$

Table 7.5: Instance level predicate (indirect approach).

The organizational-level solution is captured by the predicates `assigned` which specifies the goal-to-role assignment and the `and/or_decomposed` shows the ways goals should be decomposed according to the given solution. In that way, every instances of a same goal have the same way of decomposition. The predicate `desire_from(d,g)` binds each desire to its corresponding goal in the organizational-level solution. This binding is needed for the planner to decompose desires in future.

The table 7.5 implies the answer to question 2 previously mentioned. In this approach, agent is described as potential instance of a role by the predicate `can_play`. There two constraints are considered. The first one is the conflict in desire assignment which means there two desires could not be assigned to a same agent according to the regulation. The predicate `desire_assignment_conflict_agent(t1, t2, a)` means it is not allowed to assign both desires $d1$, $d2$ to the same agent a. More generally, the predicate `desire_assignment_confict(d1, d2)` is used to described the conflict for any agent. Similarly, two versions of `role_assignment_conflict` are employed to specify the prohibition of assigning roles $r1$, $r2$ to an/any agent. To this extend an instantiated plan/solution consists of the following actions:

- `TAKE_ON_ROLE`$(ag : agent, r : role)$: agent ag is assigned to role r.
- `AND/OR_DECOMPOSES`$(ag : agent, d, d_1, , d_n : desire)$: agent ag decomposes desire d into n AND/OR sub-desires;
- `PASSES`$(ag_1, ag_2 : agent, d : desire)$: agent ag_1 delegates desire d to agent ag_2;
- `SATISFIES`$(ag : agent, d : desire)$: agent ag is assigned to satisfied desire d.

7.2.2 Direct instance level planning

The direct instance-level planning, in contrast with the indirect one, constructs the social network directly from the formalized ODM with extra information about agents.

In this approach, the search space is bigger due to dependency relations and capabilities are linear increased proportionally with the number of agents. This number is usually larger many times in comparison with that of roles. There are two strategies, as discussed, to formalize agents/desires: describing agents/desires based on either roles/goals, or independent entities. Obviously, the former requires less formalization predicates then the later. But the planning complexity is higher in the former since the planner has to infer instances' properties and relations based on their corresponding class.

Formalizing agents/desires based on roles/goals. In this strategy, we reuse all predicates formalizing actors/goals mentioned in section 7.1 for describing roles. For agents, we employ predicates discussed in section 7.1.1 (except ones used for capture organizational-level solution).

The constructed solution includes actions mentioned in section 7.1.1, but they, certainly, have a different logic inside. The planner needs more extra actions to perform the planning. All necessary actions with their meaning are described in table NN.

Formalizing agents/desires as independent entities. In this strategy, each agent is described separately. In some points of view, this kind of formalization is a degeneration of an organizational planning in which each role has only one instance, and there are plenty of different roles that have the same capabilities. In this fashion, the instantiation formalization becomes as simple as that of organizational level, except the former has more predicates than the later. In this setting, we can reuse all predicates as well as actions describing in section 7.1

Institutively, formalization done in the first strategy requires less number of predicates in the problem file than that with the second one. In contrast, the first one consumes more CPU resources than the second since all information for instances needed for planning have to be inferred at planning time.

There two formalizing strategies for agents and goals can be mixed. It means that we can use the first strategy for formalizing agents and the second one for goals, or vice versa. This mixed strategy is employed in our framework to deal with the instantiation problem.

Chapter 8

Solution Assessment

Solution Assessment, as aforementioned, plays a critical role to the success of our framework. A good and informative assessment will support designers at design time as well as users at runtime when deciding an appropriate configuration for their STS. For automatically self-reconfigurable STS, the Solution Assessment becomes more important because the STS has to make decision by itself about which configuration should be applied.

There are two types of assessment: *qualitative* and *quantitative*. While the former suits only for human, the later is applicable for automatic self-reconfiguration. In our framework, we mainly support the later assessment. However, it is not difficult to extend the framework to accept qualitative assessment.

From the technical point of view, quantitative evaluators could be felt into two groups: *early evaluators* and *late evaluators*. The formers are used by the AI planner to drive the planning process. Only solution that is better than the previous one, in sense of applying a particular evaluator, is considered. This kind of evaluators needs to be hard-code in the PDDL script. In the meanwhile, the later evaluators, instead of hard-code script, are java classes. These evaluators are used to evaluate generated solutions.

The early evaluators can accessed properties of organizational objects (actors, actor's capabilities and goals) by encoded functions in the PDDL script. As discussed in section 6, each objects' property descriptor has a corresponding PDDL predicate. This predicate is used to encode the property values. The early evaluators need to integrated into PDDL actions' preconditions and effects. Therefore, there is a commitment of properties' name in the PDDL script and the ODM.

The late evaluators are also divided into two categories: *simple evaluator* and simulating-based evaluator. The *simple evaluator* takes the input solution, performs analysis on this solution and computes the final measurement. During analyzing process, the evaluator can access the ODM to retrieve information associated with goals or actors e.g., an *effort*, *time* which an actor have to pay to accomplish a task. The *simulating-based evaluator*, on the other hand, simulates the input solution and analyzes the simulation process to calculate the assessment value. A simple example of this kind of evaluator is the *Execution Time* evaluator. This evaluator simulates a given solution and measure the time needed for complete the whole solution. Notice that, the execution time of a solution is different from its length. A longer solution might complete earlier than a shorter one. This behavior happens because solution's actions can be performed simultaneously. When precondition of an action is satisfied, it is immediately executed without waiting for the prior action in the solution.

For the sake of clarity, our framework supports three different types of evaluators:

- *Early evaluators* which are hard-coded in the PDDL script,
- *Simple evaluators* which are scalar functions taking a solution and computing the measurement,
- *Simulating-based evaluators* which are also scalar functions, but they takes the simulation result returned by the event-based simulation of a given solution.

The design of this framework accepts runtime addition of late evaluators. Extra evaluators are imported to the framework by registering with the *Evaluator Registry*. While registering an evaluator, users are able to declare parameters using by this evaluator. Each parameter has a name, data type and default value. The default value can be changed later. Currently, four primitive types of parameter are accepted: *Integer, String, Date time* and *Boolean*. Besides, the *composite parameter* type allows user to declare structural values.

The simple evaluator should implement interface `IEvaluator` (figure 8.1). This interface provides two methods: one for computing assessment value and one for combining two assess value. In method `evaluate()`, we can access to other services provided by the framework (context), ODM (model), the parameters of this evaluator declared in registry entry (item) and list of actions to be evaluated (actions). This method returns an `EvaluatorResult` which contains the assessment value as well as all necessary values to recalculate it.

```
interface IEvaluator
    function evaluate(context: IJasimContext, model:ODM, entry: EvaluatorEntry, actions: list of PddlAction): EvaluatorResult;
    function combine(res1, res2: EvaluatorResult): EvaluatorResult;
end
```

Figure 8.1: IEvaluator interface.

The second method in this interface is used to combine two separated results into one. It is extremely useful in simulating-based evaluator. During event-considered simulation, if an event happens and causes the simulator to re-plan, the original solution is not completely executed, and is substituted by a new one. Therefore, to assess the original solution, we should evaluate the executed part of the original solution, and the substitution.

8.1 Simulation Engine

The simulating-based evaluators carry our assessment by analyzing the execution of a given solution. This is done thank to the *simulation engine*. There are two kind of simulation supported by the simulation engine: *plain simulation* and *event-based simulation*. The *plain simulation* takes a solution and simulates the whole solution, meanwhile the *event-based simulation* take into account events which may happen during the simulation process. The effects of an event may cause simulating solution invalid, and the solution needs to be replanned at the time it gets corrupted. Simulating solution in this manner shows the resilience of the original solution regard to a given set of events. This assessment point of view is important since these events can happen in the real, and the selected solution should be good enough to resist the environmental changes. The simulation engine provides two simulators called *plain simulator* and *event-based simulator* to support the plain simulation and event-simulation, respectively. These simulators are described in the following sections.

8.1.1 Plain Simulator

The objective of the simulation is to create an environment in which all actions of a solution are executed as they are in the real. That is each action requires a period of time to accomplish, and consumes some resources. At a certain time, one agent can only perform one action, but actors can carry out actions in parallel unless there are dependencies among jobs. A good simulator should have an internal optimizer so that there are as much as possible actions are performs at a same moment to accomplish the solution in the shortest time.

Each action has a particular execution time. The unit of time is not important in the simulation; it thus can be minute, hour, day or whatever. Instead, we emphasize on the relative between them i.e., the goal g_1 is five times longer that goal g_2 to be accomplished. The table 8.1 shows the execution time for supported actions in a solution. Particularly, the `SATISFIES` action's duration depends on what goal is satisfied as well as who is doing the goal, meanwhile, other actions are assumed to be accomplished in a constant time. For other actions not supported, the simulator simply drops them out.

Action	**Exec. Time**
`AND/OR_DECOMPOSE`	1
`PASSES`	1
`SATISFIES`	vary

Table 8.1: Supported actions by the solution simulator and their execution time.

The heart of the simulator is the algorithm illustrated in figure 8.2. This algorithm is based on the idea of the JASON simulator [4] in which the simulation time are divided into slots. In each timeslot, actions whose precondition is matched are executed. The simulation stops when no more action is executed.

The simulation algorithm starts by a call to `InitializeAgentsList()` (line 7). This procedure scans through the solution for the list of actors and their corresponding actions. It also assigns the initial goals for these actors based on the initial requests. When a goal is assigned to an actor, its corresponding actions will be put in the action queue of this actor. In line 8 to 9, it resets the virtual clock to 0 and sets the termination condition to false. The virtual clock is used to measure the duration of an action as aforementioned. The simulation actually starts in loops until its termination condition is matched in the While loop found in line 10. In this loop, the simulator builds a list of active actions (line 11). In a clock circle, there only one action is active for an actor. The active action for an actor is selected from its action queue, and should be in ready-to-execute status which means the execution precondition for action is satisfied. For action `AND_DECOMPOSE`, `OR_DECOMPOSE`, the precondition is the to-be decomposed goal is the active goal of an appropriate actor. For action `PASSES`, the precondition is the delegator actor is available and free. Alternatively, action `PASSES` is always true, and each actor has its own goal queue to keep a list of goals to be fulfilled. For action `SATISFIES`, the precondition is more complex. It is the actor who will satisfy this goal is

```
function simulate(actions: list of PddlAction)
var
    clock: integer; terminated: boolean;
    activeActions: list of PddlAction;

begin
    initializeAgentsList(actions);
    clock = 0;
    terminated = false;
    while terminated = false do
        activeActions = getListOfActiveActions();
        if activeActions is not empty then
            slot = create new time slot for the current clock;
            for each action: Action in activeActions do
                slot.add(action);
                performAction(action, clock);
        else
            terminated = true;
    return clock;
end
```

Figure 8.2: Solution simulation algorithm.

available and free, all prerequisite goals are fulfilled, all resources required by this goal are available and ready to use.

If the active action list is not empty, the simulator creates a new time slot which holds a list of actions and actors who are in charge of doing them in a specific time circle. After being added to the time slot, each of active actions is started executing (line 14). Base on action's type, the simulator performs different behaviors as following:

- `AND/OR_DECOMPOSE`: the simulator looks up the ODM for the sub goals and then replaces the top goal by its sub goals in the actor's goal queue.
- `PASSES`: the simulator remove the goal in the goal queue of the actor who delegates and add this goal to the delegate's goal queue.
- `SATISFIES`: the simulator looks up the model the simulator does nothing as assuming the actor is doing something to archive the goal.

8.1.2 Simulation event

As discussed, a simulation event has three part: *precondition*, *post-condition* and *parameters*. The event *precondition* express whenever an event happens, and its effects are described in *post-condition*. Meanwhile, the *parameters* are values used by the reactions composing the *post-condition*.

Currently, there three preconditions are supported:

- `AbsoluteTime` is triggered when the simulator's clock reachs to a given time value,
- `ActionRelativeTime` is triggered in a given time circle after a specific action happens,
- `EventRelativeTime` is triggered in a given time circle after a specific event happens.

At runtime, when an event reaction is executing, beside parameters that could be valued before simulating, it may also need other contextual information at the moment the event happens e.g., the goals, actors affected by this action. The reaction can get this information thank to the `Parameters` section of the event definition and the magic of the simulator which will be detailed in section 8.1.3.

`ModifyObjectReaction`

Description modifies dynamic property of an object.

Parameters

Action	: $\{Assign, Increase\}$
NewValue	: map to a variable (*) holding the new value.
Object	: map to a variable (*) points to the modified object.
Property	: the `PropertyDescriptor` describing the modified property.

`ModifyPDDLReaction`

Description modifies the generated PDDL script

Parameters

Action	: $\{Add/Remove\_Fact, Add/Remove\_Goal\}$
Predicate	: the PDDL predicate being injected to (removed from) the PDDL script.
PredicateArg	: arguments of the predicate.
Negative	: indicate whether the predicate is negated.

`ForAllReaction`

Description repeats a list of reactions where the parameters receive new values in each loop according to a *ForAll-domain*. The ForAll-domain provides a list of value based on the input parameters. The following ForAll-domains are currently supported in the framework:

- `ProvidedGoalDomain`: goals provided by a given actor.
- `RequestGoalDomain`: goals requested by a given actor.
- `AssignedGoalDomain`: goals assigned to a given actor.
- `SatisfyGoalDomain`: goals assigned to a given actor, but not yet satisfied.
- `RequestorDomain`: actors who request a given goal.
- `ProviderDomain`: actors who provide a given goal.

`ReplanReaction`

Description forces the event simulator to re-plan.

Table 8.2: Built-in event reactions

Example 8.1. Let consider a solution in which an actor *Alice* is assigned goal *Shutdown_electricity*. But she fails in 5 units of time after trying to accomplish the goal. And *Alice* does not has the ability to fulfill this goal any longer. Unluckily, this goal still needs to be fulfilled. So, we need to replan to find a new solution in the context that *Alice* no longer provides this goal. The definition for the event describing the situation is illustrated in figure 8.3. When the simulator encounter the action `(SATISFIES Alice Shutdown_electricity)`, this event is triggered, the parameter *Actor* and *Goal* are respectively set to `Alice` and `Shutdown_electricity`. Afterward, the post condition is applied which removes the fact `(CAN_PROVIDE Alice Shutdown_electricity` from the new PDDL script before re-planning. ■

```
Event: SatisfactionFailure
Precondition:
    ActionRelativeTime
        Action       = SATISFIES
        Actor        = ?
        Goal         = Shutdown_electricity
        Time         = 5
Parameters:
        Actor        = ?
        Goal         = ?
Post-Condition:
    ModifyPDDLReaction
        Action       = REMOVE_FACT
        Predicate    = CAN_PROVIDE
        PredicateArg = {Actor} {Goal}
    Replan
```

Figure 8.3: Satisfaction Failure event

8.1.3 Event-based simulator

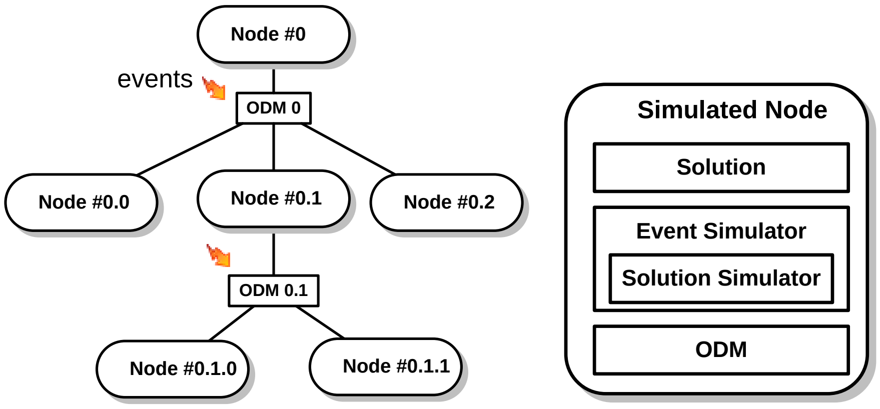
In this section, we discuss about the *event-based simulator* (hereafter, referred to as *event simulator*) which is built on the plain simulator as previously represented. The event simulator hooks into the plain simulator for checking whether events happen on every time circle. The event simulation algorithm is listed in figure 8.4

The algorithm starts by creating a structure called *simulation node* (figure 8.5(b)) has references to the corresponding solution, the PDDL script that

```
function event_simulate(model: ODM, sol: Solution)
var
    node, root, child: SimulationNode;
    Q: Queue of SimulationNode;
    new_solutions: list of Solution;

begin
    root = new SimulationNode(solution);
    Q.enqueue(root);
    while Q is not empty do
        node = Q.dequeue();
        node.createPlainSimulator(); {create a plain simulator for the solution associated with the processing node. The created simulator is accessed through Simulator property of node}
        Hook event processor into simulator;
        node.Simulator.simulate(node.Solution);
        if Need replan then
            Capture state of world into the model's post-PDDL command list;
            new_solutions = generateSolutions(model);
            if new_solutions is not empty then
                for each s: Solution in new_solutions do
                    child = new SimulationNode(s);
                    node.addChild(child);
                    Q.enqueue(child);
            else
                node.isDanglingNode = true; {This node needs to replan, but no solution found. Then it is considered as dangling node.}
        else
            node.isCompleteNode = true;
    return root;
end
```

Figure 8.4: The event simulation algorithm.



(a) Simulation Tree (b) Simulation Node

Figure 8.5: The simulation tree structure returned by the event simulator (a) and the structure of a simulation node (b).

generates this solution, the ODM that generates the PDDL script and the plain simulator that simulates this solution. The created node is put in a queue `nodes` (line 9).

In the While loop (line 10), the event simulator picks up an *simulation node*, and invokes the plain simulator to simulate the original solution. When an event precondition is satisfied, event simulator bind each unassigned parameter in `Parameters` section (c.f. section **??**) with appropriate value extracted from the solution's action interrupted by the event. The binding is done by using name matching. Then, all reactions in the event's post condition are applied.

If one of the reactions causes the solution fails to continue, the event simulator captures the current state of world and passes to the *Solution Generator* for re-planning. New generated solutions could be filtered to improve the performance as well as to prevent the solution explosion. The filter criteria are based on a simple evaluator, by which, only "good enough" solutions are returned. By default, the filter does nothing. Users have to choose an appropriate simple evaluator and a threshold value. A solution is considered as good enough if the returned value of the filter evaluator is greater than or equals the threshold value. Afterward, the event simulator creates new simulation nodes corresponding to new generated solutions, line 17. These nodes are enqueued, line 22, and later processed by the event simulator.

Another point to be considered is that the AI planner does not always find and return a solution. Because the most of planner chooses the local search approach, hence there is no guarantee that the search will stop with a solution. And it is not enough evidence to conclude that the problem is unsolvable. To deal with such situations, we set the time limit for the planner. We assume that if the AI planner does not find any solution within a period of time, then the problem is unsolvable.

If a *simulation node* needs to re-plan but there is no solution found, we call this node as dead-end node or *dangling node*. The *dangling node* means if we follow this solution, and the given set of events happens, we will never archive all the top goals. On the other hand, we called a simulation node as leaf node if the associated solution is successfully completed.

At the end of simulation progress, the event simulator returns root node of the simulation tree (as depicted in figure 8.5(a)) to the caller which in turned passes it to the event-based evaluators, and to the *Visualizer* to display the final result on the screen.

8.2 Simple Evaluators

8.2.1 Overall Benefit/Cost Quotient

The benefit/cost quotient (BCQ) is widely adopted in many fields, especially economic. The objective is to gain benefit per each unit of cost as much as possible. In our organizational planning problem, the BCQ could be the satisfaction degree of all goals over the effort (or consumed time) need to pay to archive goals. In some situations, users may want to focus only on the cost (or benefit) come from goal satisfaction.

Normally, each goal has its own threshold level of satisfaction. A goal is considered as satisfied if it is fulfilled with a certain level of satisfaction at least the threshold value. In practice, there are many actors are able to satisfy a given goal, but their satisfaction abilities of this goal are different.

Example 8.2. Considering an example where two students Alice and Bob want to satisfy the goal *"passing the Math examination"*. Alice accomplishes the exam in 1 hour with grade 8 (10 is max) while Bob needs 2 hours with grade 9. Obliviously, the later student satisfies the goal with a higher level than the former. But both of them pass the exams since their grades are greater than 5. Alice, however, needs less effort to complete exams than Bob. Then we can say that the benefit and cost of actor (student) Alice for completing this goal are 8 and 1, respectively; and those of actor Bob are 9 and 2.

Therefore, the efficiency (BCQ) of Alice ($8/1 = 8$) is theoretically better than Bob ($9/2 = 4.5$). Nevertheless, the comparison of BCQ does not make sense in this case because the final grade, of course, is more important regardless how much effort each student has to pay. ■

To this extend, in order to evaluate the BCQ of a solution, the BCQ evaluator need to know the benefit and cost for each action in a solution. In this evaluator, we consider only satisfaction action. Therefore, the benefit of the action $SATISFIES(a, g)$ determines how well the actor a fulfills the goal g. Similarly, the cost of this action is how much effort the actor a has paid for fulfilling goal g. Hence, the domain of the effort and benefit could be $\{low, medium, high, veryhigh\}$, and {*average, good, verygood, excellent*}. Therefore, the registered entry of BCQ in the event model of ODM as a tuple $\langle BCQ, \{\langle Cap, BENEFIT\rangle, \langle Cap, EFFORT\rangle\}\rangle$. We employ a *normalize function*, **norm**, which maps these enumerate values to numeric as of formula 8.1.

$$\mathbf{norm}(effort) = \begin{cases} 25, & low, \\ 50, & medium, \\ 75, & hight, \\ 100, & veryhight \end{cases}, \mathbf{norm}(benefit) = \begin{cases} 25, & average, \\ 50, & good, \\ 75, & verygood, \\ 100, & excellent \end{cases} \tag{8.1}$$

The BCQ of a solution is computed as follow:

$$BCQ(S) = \frac{\sum_{act \in S} \mathbf{norm}(Bef(act))}{\sum_{act \in Act} \mathbf{norm}(Cost(act))} \tag{8.2}$$

Where:

- $Bef(act) = \begin{cases} \mathcal{P}^v(\text{cap}, \texttt{BENEFIT}), & act = SATISFIES(a_i, g_k) \wedge \\ & cap = \langle a_i, g_k\rangle \in Cap \\ 0, & \text{otherwise} \end{cases}$
- $Cost(act) = \begin{cases} \mathcal{P}^v(\text{cap}, \texttt{EFFORT}), & act = SATISFIES(a_i, g_k) \wedge \\ & cap = \langle a_i, g_k\rangle \in Cap \\ 0, & \text{otherwise} \end{cases}$

The figure 8.6 presents the algorithm of the BCQ evaluator.

This algorithm is worth for some comments. Line 8 and 9 find the `PropertyDescriptor` of `BENEFIT` and `COST` which are attached to actors' `Capability` as mentioned above. In the FOR loop (line 10), only `SATISFIES` actions are considered. The algorithm extracts the actor, goal from the action; and looks for the corresponding capability. Then, it extracts the cost and benefit from this capability to accumulate.

This benefit/cost model could be extended to accept many types of cost as well as many kinds of benefit. Then cost and benefit is not only a scalar value,

```
function CBQ(model: ODM, actions: list of PddlAction)
var
    $benefitProp, costProp$: PropertyDescriptor;
    $act$: Actor; $goal$: Goal; $cap$: Capability;
    $cost, benefit$: float;

begin
    $benefitProp$ = $model$.getDescriptor(Capability, BENEFIT);
    $costProp$ = $model$.getDescriptor(Capability, COST);
    for each $a$: PddlAction in actions do
        if $a$.Functor = SATISFIES then
            $act$ = $model$.getActorByName($a$.getArgument(0));
            $goal$ = $model$.getGoalByName($a$.getArgument(1));
            $cap$ = $act$.findCapability($goal$);
            if $cap$ is not null then
                $cost$ += norm($benefitProp$.getValue($cap$));
                $benefit$ += norm($costProp$.getValue($cap$));
    return $benefit/cost$;
end
```

Figure 8.6: The Benefit/Cost quotient algorithm.

but a vector. In this case, each element in a vector has different contribution factor as it has different important level. More important, elements in the cost vector are usually belong to different domain or measurement unit. For instance, a cost vector consists of two element $\langle effort, duration\rangle$, the *effort* could be one of $\{low, medium, high, veryhigh\}$, and *duration* could be the number of minutes to complete the task. Therefore, we need a *normalize* function that maps these heterogenous values to a standard domain. The following is an alternative of formula 8.2, but it accepts cost/benefit vector.

$$\overrightarrow{BCQ}(Act) = \frac{\sum_{act \in Act} \overrightarrow{Bef}(act)}{\sum_{act \in Act} \overrightarrow{Cost}(act)} \tag{8.3}$$

or in the scalar form

$$\overline{BCQ}(S) = \frac{\sum_{act \in S} \sum_{i=1}^{n} \alpha_i \cdot \mathbf{norm}(Bef_i(act))}{\sum_{act \in S} \sum_{i=1}^{m} \beta_i \cdot \mathbf{norm}(Cost_i(act))} \tag{8.4}$$

where:

- α_i, β_i: the contribution factor of the i^{th} element of benefit and cost vectors, respectively.
- **norm**: is the normalize function since the costs values are heterogenous.
- $Bef_i(act), Cost_i(act)$: the i^{th} element of benefit and cost vectors, respectively.

8.2.2 Actor criticality analysis

The solution could be considered as a social network. In this network, an actor is a super node consisting of many goal nodes. Goal delegations among actors create links for the network. The criticality of an actor measures how a social network will be affected in case the actor has been removed or has left the network. This notion is highly connected to that of *resilience* of network [9]. In practice, a social network might be collapsed if its highest-degree nodes are removed. Therefore the study of criticality of actors in a solution is quite important to the self-reconfigurable STS, since this highly vulnerable can happen in the real life.

In our framework, we adopt the idea of criticality analysis in [9] which concisely summarized as follow.

Leaf goals satisfaction dimension All the leaf goals assigned to an actor will be unsatisfied when that actor is removed. Let an integer number $w(g)$ is the weight of goal g. $w(g)$ is intuitively the measure of importance of g for the system defined by a human designer. The criticality of actor a in a solution/configuration S, according to leaf goals satisfaction dimension, is defined as:

$$cr_g(a, S) = \frac{\sum_{SATISFIES(a,g) \in S} \omega(g)}{\sum_{SATISFIES(x,g) \in S} \omega(g)} \tag{8.5}$$

Where x is an actor and g is a goal

Dependency dimension All the in- and outgoing dependencies for goals, together with the actor, are removed when actor is removed. This means that a number of delegation chains become broken and the goals delegated along these chains cannot reach the actor at which they will be satisfied. Hence, the fraction of "lost" dependencies (ingoing or outgoing) when actor a is removed from the socio network constructed in accordance with solution $S$is:

$$cr_{in}(a, S) = \frac{\sum_{PASSES(a',a,g) \in S} \omega(g)}{\sum_{PASSES(x,y,g) \in S} \omega(g)},$$

$$cr_{out}(a, S) = \frac{\sum_{PASSES(a,a',g) \in S} \omega(g)}{\sum_{PASSES(x,y,g) \in S} \omega(g)},$$

$$cr_{dep}(a, S) = cr_{in}(a, S) + cr_{out}(a, S) \tag{8.6}$$

Where *a', x, y* are actors and g is a goal

Actor criticality with respect to a set of goals It is also important to quantify the impact of an actor removal on the top-level goals of a STS, or, in general, on any predefined set of non-leaf goals. Let $G_{dir\_aff}(a, S)$ is the set of goals directly affected by the removal of actor a in solution

S:

$$\begin{aligned} G_{dir\_aff}(a, S) = \{ & g : goal.SATISFIES(a, g) \in S \vee \\ & \exists a' : actor.(PASSES(a', a, g) \in S) \vee \\ & \exists a'' : actor.(PASSES(a, a'', g) \in S) \} \end{aligned}$$

Let $G_{aff}(a, S)$ is the set of goals affected by the removal of actor a in solution S, corresponding to the set $G_{dir\_aff}(a, S)$. $G_{aff}(a, S)$ is constructed by goal reasoning which allows to infer the (un)satisfiability to top goals by propagating though a goal graph the (un)satisfaction evidence [9].

Let G_r is the set of "reference" goals, i.e. top-level or any pedefined subset of system goals with respect to which criticality of an actor in a solution will be evaluated [9]. Then the criticality of a in S corresponding to $G_r$is defined as follows:

$$cr(a, S, G_r) = \frac{\sum_{g \in G_r \wedge G_{aff}(a,S)} \omega(g)}{\sum_{g \in G_r} \omega(g)} \tag{8.7}$$

Based on the above three dimensions, we introduce the concept of *overall actor criticality* for a specific actor $a$as:

$$cr(a, S) = \omega_1 cr_g(a, S) + \omega_2 cr_{dep}(a, S) + \omega_3 cr(a, S, G_r) \tag{8.8}$$

where $\omega_i, i = \overline{1, 3}$ are the contribution factors of each type of criticality measurements. Since there are many actors with different criticality in one solution, we are thus interested in the variance analysis of actor criticality in solutions. The variance ratio is computed as:

$$\Delta cr(S) = \frac{1}{N} \sum_{a \in S} (cr(a, S) - \overline{cr})^2 \tag{8.9}$$

where

$$\overline{cr} = \frac{1}{N} \sum_{a \in S} cr(a, S)$$

We consider this variance ratio (or standard deviation) as another metric for evaluating solutions. The lower the variance ratio of actor's criticality in a solution, the more *resilience* the solution is.

Another approach for criticality analysis Thank to the event-based simulator, we can explore another expansion to evaluate the criticality of an actor. We start the simulation on the given solution then trigger an event that removes an actor to see how the solution is adapted. If the solution fails and can not be replanned successfully, then the actor is high-criticality in solution; otherwise it is not. We can also remove some particular parameters (e.g, the capability of satisfying certain goal) of an actor instead of removing the actor itself. The simulation can also be run on several solutions to see how resilient they are with the same event.

By this faction, we might identify the criticality of the actor in a specific period of time during the execution of the given solution.

8.3 Simulating-base Evaluators

8.3.1 Solution execution time

The solution execution time is somehow an interesting criterion. In some situation when the time to archive the top goal is the most important regardless of cost and benefit obtaining from the goal.

As aforementioned, actions in a solution can be performed in parallel. Hence, there is no correlation between solution length and solution execution time. Instead, an accurate way is to simulate the given solution. Depend of which simulator is applied, there are two calculation for the solution execution time. If solution simulator is used, the execution time is the value returned by the simulator. If event simulator is employed, the simulation tree is returned. Let consider the simulation tree as a direct weighted graph where the weight of a connection is the simulation time of the target node. If the target is a dangling node, the weight is set to infinitive. To this extend the execution time is the simulation time at the root node plus the cost of the shortest path from root to a leaf node.

In figure 8.7, it is the algorithm calculating the execution time of a solution base on the simulation tree returned by the event simulator. The input of the algorithm is a simulation node. The idea is to employ recursive method. The algorithm result is first set to the simulation time of the input node. If the simulation node has any child, it recursively calls itself for each of children to find the smallest value. Finally, it adds this value the initial result and returns the sum value to the caller.

```
function executionTime(node: SimulationNode)
var
    $time, child\_time, temp$: integer;

begin
    $time$ = $node$.Solution.getSimulationTime(); {get the simulation time of the associated solution.}
    if $node$.Children is not empty then
        $child\_time$ = MAX_INT;
        for each $child$: SimulationNode in $node$.Children do
            temp = executionTime($child$);
            if $temp < child\_time$ then
                $child\_time = temp$;
        $time$ += $child\_time$;
    return $time$;
end
```

Figure 8.7: The solution execution time algorithm.

8.4 Early Evaluators

8.4.1 Local Gain of Benefit/Cost Quotient

The *Local Gain of Benefit/Cost Quotient* (LG-BCQ) evaluator computes the gain of each action generated during the planning process. Each time the AI planner tries a move (or action), it calculates the LG-BCQ value for this move as guided in the PDDL script. This value then is accumulated and used as a optimizing metric for the planner. Therefore, the planer always tries to generate a solution with the maximal accumulated LG-BCQ.

Unlike the overall BCQ (section 8.2.1) which only measures cost and benefit for goal satisfaction, LG-BCQ takes into account other actions as well. Eventually, there three different kinds of action are considered as follows:

- *Goal satisfaction*: $\mathbf{cs}_{ik}$ and $\mathbf{bs}_{ik}$ respectively denotes the cost and benefit of satisfying goal g_k by actor a_i. These values are the same of that used in the overall BCQ.
- *Goal decomposition*: $\mathbf{cr}_{ik}$, $\mathbf{br}_{ik}$ denotes the cost and benefit of decomposing goal g_k by actor a_i.
- *Goal delegation*: $\mathbf{cd}_{ik}$, $\mathbf{bd}_{ik}$ denotes the cost and benefit of delegating

goal g_k by actor a_i.

The LQ-BCQ then is computed as of formula 8.10:

$$\begin{aligned} LQ - BCQ(S) = & \sum_{SATISFIES(a_i,g_k)\in S} \frac{\mathbf{bs}_{ik}}{\mathbf{cs}_{ik}} + \\ & \sum_{DECOMPOSES(a_i,g_k,g_{k1},\ldots,g_{kn})\in S} \frac{\mathbf{br}_{ik}}{\mathbf{cr}_{ik}} + \\ & \sum_{PASSES(a_i,a_j,g_k)\in S} \frac{\mathbf{bd}_{ik}}{\mathbf{cd}_{ik}} \end{aligned} \tag{8.10}$$

where

- $\mathbf{bs}_{ik} = \frac{1}{n} \cdot \sum_{j=1}^{n} \mathbf{norm}(Bef_j(SATISFIES(a_i, g_k)))$
- $\mathbf{cs}_{ik} = \frac{1}{n} \cdot \sum_{j=1}^{n} \mathbf{norm}(Cost_j(SATISFIES(a_i, g_k)))$
- $\mathbf{br}_{ik}$, $\mathbf{cr}_{ik}$, $\mathbf{bd}_{ik}$, $\mathbf{cd}_{ik}$ are assigned values by domain experts. These values should also be normalized like that of SATISFIES action.

The LQ-BCQ is an early evaluator, it thus is hard-coded in the PDDL script which has limitation of function invocation. Therefore, in our implementation all costs value are first normalized before they are scripted.

8.4.2 Actor Budget Constraint

The *Actor Budget Constraint*(ABC) evaluators prevents the situations in which an actor is assigned too much that exceed actor's limitation. Concretely, suppose that each actor has a budget to satisfy goals. And when this actor fulfills a goal, the budget is decreased a certain amount. Once this budget is too low to fulfill any goal, this actor is exhausted and it should not be assigned goals any more. The ABC objective is to avoid such a situation.

The implementation of ABC is quite simple. At the beginning, each actor has its own value of maximum budget. For each of consumed energy actions (`AND/OR_DECOMPOSE, PASSES, SATISFIES`) we append a precondition saying that actor's budget should be greater than the required cost of this action. And in the action's effect, we decrease actor's budget by amount of action's cost. The action's cost can be retrieved as described in 8.4.1.

To this extend, this early evaluator can be seen as an implementation of the *local optimization* in Bryl et al approach [9], but at the planning process.

Chapter 9

Implementation

This chapter gives a glance to the implementation of the framework as well as the employed technologies. We use Java as programming language to develop the framework due to its popularity in the research community. The framework is developed base on the Eclipse Modeling Framework Project. And for the AI planner, we adopt LPG-td which won the best planner prize in the International Planning Competition 2004[1].

9.1 Eclipse Modeling Framework

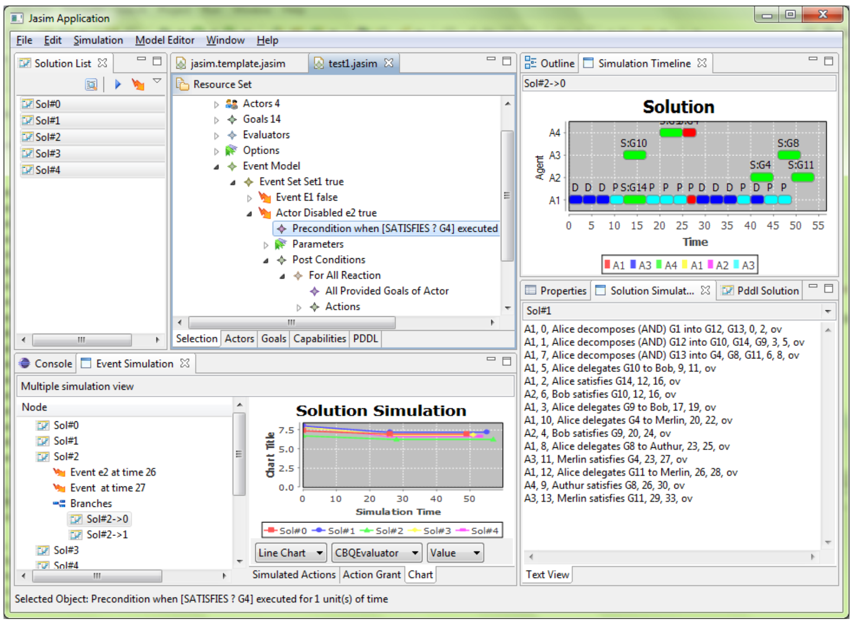
Eclipse Modeling Framework project (EMF) [35, 17] is a modeling framework and code generation facility for building tools and other application based on a structured data model.

EMF provides tools to define the data model which saved in XMI format, and provides tool for generate runtime support Java classes. The generated classes include APIs allow user to construct viewer, editor for the data model. They also include a basic editor for the data model.

The interesting point (also the strength) of EMF is to support code regenerate. It means that users are able to customize generated code. When they need to update to model, EMF can regenerate the code without affecting to customized code. In this way, EMF provides a mechanism for developing model-based applications.

Generated code can run as ether a rich-client program (RCP), or an plugin

[1]http://www.informatik.uni-freiburg.de/hoffmann/ipc-4/.



```
A1, 0, Alice decomposes (AND) G1 into G12, G13, 0, 2, ov
A1, 1, Alice decomposes (AND) G12 into G10, G14, G9, 3, 5, ov
A1, 7, Alice decomposes (AND) G13 into G4, G8, G11, 6, 8, ov
A1, 5, Alice delegates G10 to Bob, 9, 11, ov
A1, 2, Alice satisfies G14, 12, 16, ov
A2, 6, Bob satisfies G10, 12, 16, ov
A1, 3, Alice delegates G9 to Bob, 17, 19, ov
A1, 10, Alice delegates G4 to Merlin, 20, 22, ov
A2, 4, Bob satisfies G9, 20, 24, ov
A1, 8, Alice delegates G8 to Authur, 23, 25, ov
A3, 11, Merlin satisfies G4, 23, 27, ov
A1, 12, Alice delegates G11 to Merlin, 26, 28, ov
A4, 9, Authur satisfies G8, 26, 30, ov
A3, 13, Merlin satisfies G11, 29, 33, ov
```

Selected Object: Precondition when [SATISFIES ? G4] executed for 1 unit(s) of time

Figure 9.1: The MDI and dockable interface of an RCP.

of the Eclipse IDE. It inherits a powerful plug-in infrastructure of Eclipse IDE that allows user to customize the application by plug-in. It also provides an elegant user interface supporting Multiple-Document Interface (MDI) and dockable interface which the MDI editor sits in the center and other views are placed around (see figure 9.1).

9.2 AI Planner

As aforementioned, our framework relies on an off-the-shelf AI planner tool. We adopted LPG-td [28], a fully automated system for solving planning problems, which supports PDDL 2.2 specification [19]. The LPG-td is a fast planner with an easy, simple command interface, which supports both target platforms: Windows and Linux. However, as PDDL is a standard for the International Planning Competition, it is not difficult to move to other planners.

9.3 Jasim Application

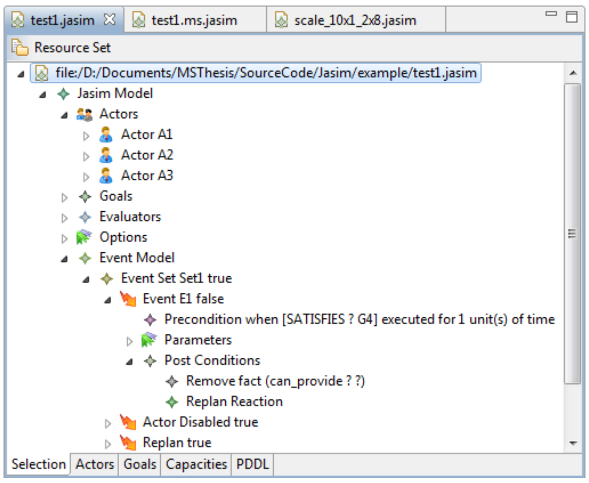
This section discusses about the Jasim Application which is built on top of the framework. Although Jasim itself is not an self-reconfigurable STS, it provides a GUI supporting designers in developing their STS. Jasim developers employ the EMF to develop an elegant, user-friendly, flexible and extensible tool. Most of functions of Jasim can be reused in developing other applications in a simplest way thank to the powerful Eclipse Plugin Architecture. The major features of Jasim are: ODM editor, ODM solution generator, and ODM solution simulation.

9.3.1 ODM Editor

The ODM editor is an elegant MDI editor which allows users to open many ODM documents at the same time. There are two kinds of ODM documents:

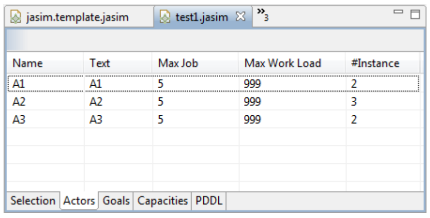
- *Template document* is a variant of the ODM. The *template document* contains *i*) sets of property descriptors for actor, goal and capability. *ii*) list of available evaluators *iii*) list of available custom reactions which can be used in define events *iv*) list of predefine events. These information can be shared between different ODM in a same domain. For example, the models of fire fighting are vary in different scenarios e.g., fire in buildings, fire in forests. But, the properties of actors, goals and capability in these scenario are somewhat similar as well as the event types and evaluators.
- *Normal document* contains information of the potential organizational model and the event model for simulation.

In figure 9.2(a), it is the ODM editor. Each of ODM document has fives subtabs. The *Selection* tab is tree-like editor for the whole content of the ODM. When a object is selected, its properties are shown in the *Properties* view in order to edit. The next three tabs, *Actors, Goals*, and *Capabilities*, are different view aspects of the ODM (see figure 9.2(b)). They are tabular editors whose columns are dynamically constructed based on the number property descriptors of the corresponding object. These editors provide users an easier way to edit objects' properties. The final *PDDL* tab shows the PDDL code reflect to the editing model.



```
test1.jasim | test1.ms.jasim | scale_10x1_2x8.jasim
Resource Set
file:/D:/Documents/MSThesis/SourceCode/Jasim/example/test1.jasim
  Jasim Model
    Actors
      Actor A1
      Actor A2
      Actor A3
    Goals
    Evaluators
    Options
    Event Model
      Event Set Set1 true
        Event E1 false
          Precondition when [SATISFIES ? G4] executed for 1 unit(s) of time
          Parameters
          Post Conditions
            Remove fact (can_provide ? ?)
            Replan Reaction
        Actor Disabled true
        Replan true
Selection | Actors | Goals | Capacities | PDDL
```

(a) Selection tab



jasim.template.jasim | test1.jasim

Name	Text	Max Job	Max Work Load	#Instance
A1	A1	5	999	2
A2	A2	5	999	3
A3	A3	5	999	2

Selection | Actors | Goals | Capacities | PDDL

(b) Actors tab

Figure 9.2: Jasim MDI editor of ODM document

9.3.2 ODM Solution Simulation

The feature consists of different views supporting the solutions simulations as follows:

- *Solution List* (figure 9.3(a)) displays a list of generated solutions. From this view, users can view the solution detail, simulate selected solutions with/without events.

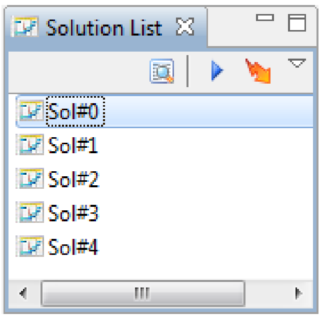
- *PDDL Solution* (figure 9.3(b)) displays the detail of selected solutions as well as the PDDL code generating this solution. This information is extremely useful for the event simulation feature. It allows users to view PDDL script of each simulation node.

- *Solution Simulation* (figure 9.3(c)) displays the execution of a solution. Each execute action has information about agent who is in charge of, the detail action and the starting time as well as ended time.

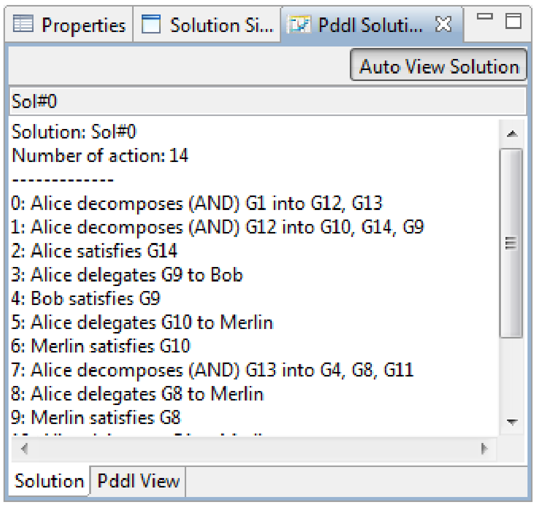
- *Simulation Timeline* (figure 9.3(d)) display the same information as *Solution Simulation* view, but in a visual way. In this view, each actor occupies a swimlane which shows actions performed by this actor. Each action is presented by a round rectangle with different color, and a label on top e.g., D for DECOMPOSE action and P for PASSES. The rectangle's length determines the duration of this action in virtual time. If a action is interrupt by a replan action, it is displayed with a red color.

- *Event Simulation* (figure 9.4) displays the result of the event simulator. The simulation tree is shown of the left hand side, meanwhile the detail of selected simulation node is displayed on the right hand side. A summarized chart for the solutions' adaptation is also illustrated on the this side. In this chart, user is able to select different chart types and scalar evaluator evaluating simulation tree.

  When user selects a simulation tree node, the corresponding solution and solution's execution are shown in the *PDDL Solution* and *Simulation Timeline* views, respectively.

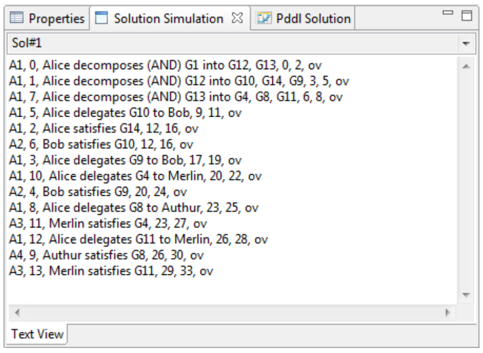


(a) Solution List



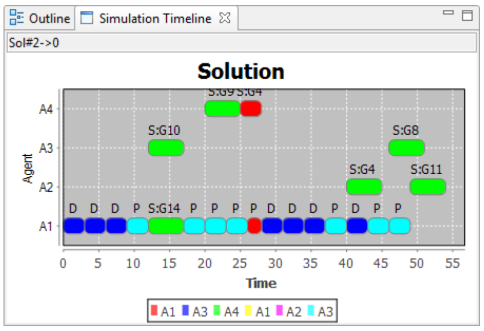
```
Solution: Sol#0
Number of action: 14
-------------
0: Alice decomposes (AND) G1 into G12, G13
1: Alice decomposes (AND) G12 into G10, G14, G9
2: Alice satisfies G14
3: Alice delegates G9 to Bob
4: Bob satisfies G9
5: Alice delegates G10 to Merlin
6: Merlin satisfies G10
7: Alice decomposes (AND) G13 into G4, G8, G11
8: Alice delegates G8 to Merlin
9: Merlin satisfies G8
```

(b) Pddl Solution



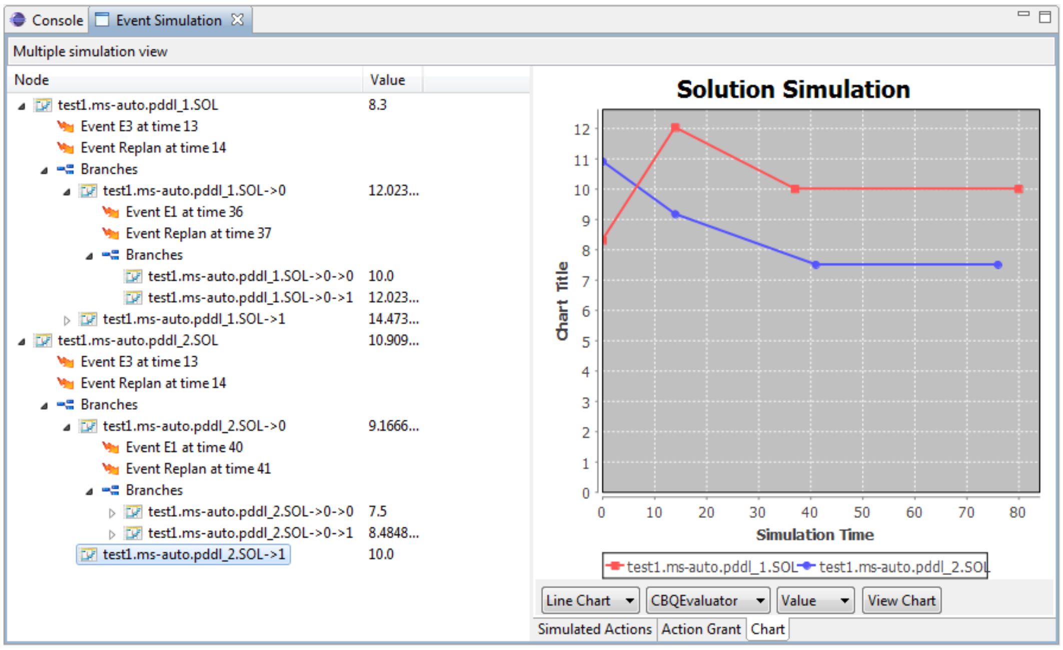
```
A1, 0, Alice decomposes (AND) G1 into G12, G13, 0, 2, ov
A1, 1, Alice decomposes (AND) G12 into G10, G14, G9, 3, 5, ov
A1, 7, Alice decomposes (AND) G13 into G4, G8, G11, 6, 8, ov
A1, 5, Alice delegates G10 to Bob, 9, 11, ov
A1, 2, Alice satisfies G14, 12, 16, ov
A2, 6, Bob satisfies G10, 12, 16, ov
A1, 3, Alice delegates G9 to Bob, 17, 19, ov
A1, 10, Alice delegates G4 to Merlin, 20, 22, ov
A2, 4, Bob satisfies G9, 20, 24, ov
A1, 8, Alice delegates G8 to Authur, 23, 25, ov
A3, 11, Merlin satisfies G4, 23, 27, ov
A1, 12, Alice delegates G11 to Merlin, 26, 28, ov
A4, 9, Authur satisfies G8, 26, 30, ov
A3, 13, Merlin satisfies G11, 29, 33, ov
```

(c) Solution Simulation



(d) Simulation Timeline

Figure 9.3: Jasim's solution simulation's views.


Console | Event Simulation

Multiple simulation view

Node	Value
test1.ms-auto.pddl_1.SOL	8.3
Event E3 at time 13	
Event Replan at time 14	
Branches	
test1.ms-auto.pddl_1.SOL->0	12.023...
Event E1 at time 36	
Event Replan at time 37	
Branches	
test1.ms-auto.pddl_1.SOL->0->0	10.0
test1.ms-auto.pddl_1.SOL->0->1	12.023...
test1.ms-auto.pddl_1.SOL->1	14.473...
test1.ms-auto.pddl_2.SOL	10.909...
Event E3 at time 13	
Event Replan at time 14	
Branches	
test1.ms-auto.pddl_2.SOL->0	9.1666...
Event E1 at time 40	
Event Replan at time 41	
Branches	
test1.ms-auto.pddl_2.SOL->0->0	7.5
test1.ms-auto.pddl_2.SOL->0->1	8.4848...
test1.ms-auto.pddl_2.SOL->1	10.0

Figure 9.4: Event simulation view.

Chapter 10

Experiment

To experimental validate our proposed framework, we run our prototype application (Jasim, cf., 9.3) on a case study of crisis management domain. Particularly, the case study focuses on the *Fire fighting at a building.* In the following sections, we detail the case study constructed based on scenario in [32],

10.1 Case Study: Fire fighting

10.1.1 Scenario

The fire happens in some location of a building. Fire warden of the building performs on-site reactions such as switch off gas/electricity and use fire-extinguishers. If the extinguishers are damaged by the fire, then it cannot be used. After fire trucks arrive to the scene, the fire commander wish to attack the fire. For the preparation, the police cordon area of bystanders. The two teams: Blue team and Green team will attempt scouting the building to locate the seat of fire and locate the place of trapped victim. The Red team launches the main attack to fight with the fire. Victims who are trapped wish to be saved. Any of the three teams can lead victim out. Medical staff then provides medical treatment for the victims.

From the above sample scenario, we derive eight actors and twelve goals (both top goals and leaf goals). The goal model is depicted in figure 10.1. Table 10.1(a) shows the actors' capabilities on satisfying a specific goal. Table 10.1(b) shows the dependency relationship; for example, actor *'A3: Fire commander'*

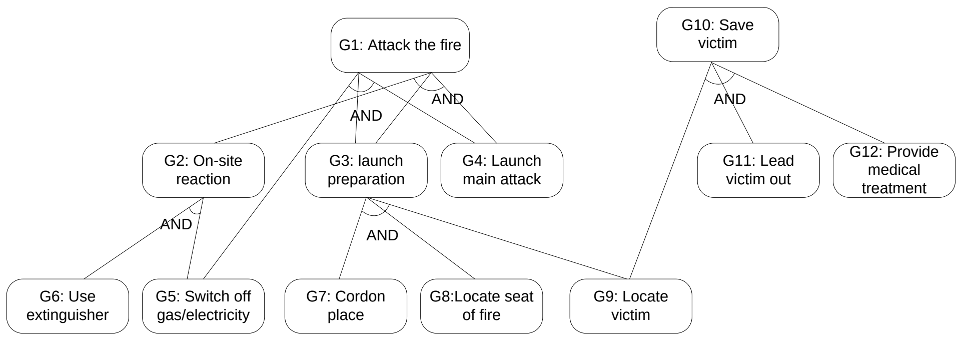


Figure 10.1: The scenario goal model.

can depend on actor *'A8: Medical staff'* to perform the goal *'G12: Provide medical treatment'*.

Table 10.2 shows the precedence constraints between goals; for example, the goal *'G11: Lead victim out'* can only be performed after the goal *'G9: Locate victim'*, and *'G12: Provide medical treatment'* can only be performed after *'G11: Lead victim out'*.

(a) Capabilities

Actor	Goal	Effort	Time	Benefit
A2	G5	25	5	100
A2	G6	25	20	100
A4	G7	25	30	100
A5	G5	25	5	100
A5	G8	25	30	100
A5	G9	25	30	100
A5	G11	50	35	100
A6	G5	25	5	100
A6	G8	25	30	100
A6	G9	25	30	100
A6	G11	50	35	100
A7	G4	75	60	100
A7	G11	50	25	100
A7	G12	25	10	100

(b) Dependency relationship

Actor	Dependum	Dependee
A1	G10	A3
A3	G4	A7
A3	G5	A2, A5, A6
A3	G6	A2
A3	G7	A4
A3	G8	A5, A6
A3	G9	A5, A6
A3	G11	A5, A6, A7
A3	G12	A8

A1:Victim, A2:Fire warden, A3:Fire commander, A4:Police, A5:Blue team, A6:Green team, A7:Red team, A8:Medical staff

Table 10.1: The actors' capabilities (a), and dependency relationship (b)

Goal	Precedence goals
G4	G5, G8
G8	G5
G9	G5
G11	G9
G12	G11

Table 10.2: The precedence constraints between goals

10.1.2 Planning result

Table 10.3 reports eight candidate solutions for the scenario. Solutions are presented in rows, meanwhile columns are organizational leaf goals. Each table cell is the goal-to-actor assignment and the period of time when this goal is fulfilled.

To evaluate these solution, besides the BCQ evaluator (cf. section 8.2.1), we will simulates them over a set of event as follow. Suppose that during the fire fighting, when a solution is in execution for 40 minutes, the available budget (cf. section 8.4.2) of the actor *Green team* is suddenly decreased by 50 (event E1) for some reasons (e.g., some members of *Green team* get injured). And at the minute of 70, the *Blue team*'s budget is decreased by 50 as well (event E2).

To this extend we simulate all candidate solutions within the set of these two events to see their effects to the candidates. The table 10.4 shows the evolution of solutions with respect to events happen. In this table, the first column *Solution* indicates the solution number. The *Initial* determines the initial BCQ assessment value (BCQ for short). Next, column *E1 (t=40)* and *E2 (t=70)* present the solutions' BCQ at the moment events happen. The last column t_{end} shows the time when each solution is accomplished. The graph shown in figure 10.2 is the quality of each solution over time.

According to table 10.4, there five solutions (#1, #3, #4, #6, #7) have the same score as 2.4096. Late on, when event *E1* happens, there two solutions #1 and #6 are affected, their scores are decreased to 2.3256 and 2.2364, respectively. Next, event *E2* arrives, solution #3 suddenly goes to dead-end which means all the top goals are never satisfied. Again, solution #6 is affected and decreased to 2.2599 while the others remain unaffected.

Based on this assessment, if total execution time is the most important, solution #2 may be the best choice. Otherwise, both solution #4 and #7 are more suitable. Solution #6 might be the poor choice since its quality is dropped down and the execution time is prolonged. In the meanwhile, solution #3 is even worst because it leads to a dead-end.

Solution	G4	G5	G6	G7	G8	G9	G11	G12
#0	A7 54–133	A6 18–22	–	A4 21–50	A5 24–53	A5 54–83	A6 84–118	A8 119–128
#1	A7 57–116	A2 18–22	A2 23–42	A4 24–53	A5 27–56	A5 57–86	A5 87–121	A8 122-131
#2	A7 54–113	A2 18–22	–	A4 21–50	A6 24–53	A6 54–83	A5 84–118	A8 119–128
#3	A7 54–113	A2 18–22	–	A4 21–50	A6 24–53	A6 54–83	A5 84–118	A8 119–128
#4	A7 57–116	A2 18–22	A2 23–42	A4 24–53	A5 27–56	A5 57–86	A6 87–121	A8 122-131
#5	A7 51–110	A6 12–16	–	A4 18–47	A5 21–50	A5 51–80	A5 81–115	A8 116–125
#6	A7 57–116	A2 18–22	A2 23–42	A4 24–53	A6 27–56	A5 30–59	A5 60–94	A8 95–104
#7	A7 57–116	A2 18–22	A2 23–42	A4 24–53	A6 27–56	A6 57–86	A5 87–121	A8 122-131

Table 10.3: Eight candidate solutions for the scenario settings.

Solution	Initial BCQ	E1 ($t = 40$)	E2 ($t = 70$)	t_{end}
#0	2.3179	n/a	dead	–
#1	2.4096	2.3256	n/a	179
#2	2.3179	n/a	n/a	128
#3	2.4096	n/a	dead	–
#4	2.4096	n/a	n/a	131
#5	2.3256	2.3392	dead	–
#6	2.4096	2.2364	2.2599	209
#7	2.4096	n/a	n/a	131

n/a: not affected, dead: no available solution

Table 10.4: The evolution of solutions according to events.

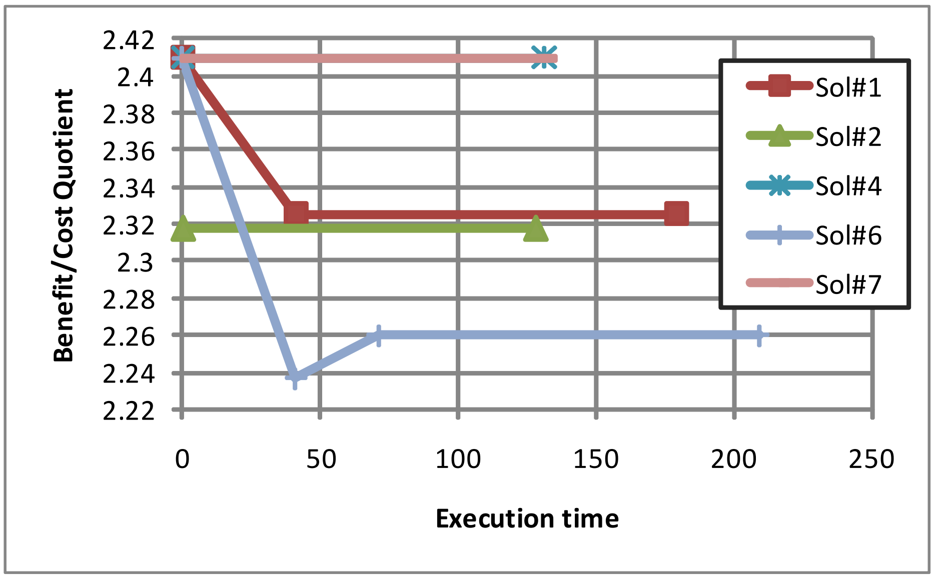


Figure 10.2: Accomplishable solutions' quality according to events happen.

Step	Time(ms)	Percentage(%)
Event-Simulation	80.73181	100%
1. Initialize simulator	0.02089	0.03%
2. Simulate solution	0.853154	1.06%
3. Replan	72.97947	90.40%

Table 10.5: Consumed time for each step in event-based simulation.

10.2 Scalability

In the domain of socio-network planning, the scalability of the planning problem and performance are very important. Obviously, the event-base simulators usually takes long time to complete than the simple ones. The table 10.5 shows the time consumed in each step of the event simulation. An event-simulation usually has there major steps: initialize the simulator, simulate the solution and replan when solution gets corrupt as events happen. Depend on the input solution and the events, these steps might happen several time. It is easy to realize that the time for (re)planning dominates times for other steps.

Hence, the performance and scalability of our framework relies heavily on the AI planner. This property depends on two factors. The first one, certainly, is the AI planner itself. We do not discuss detail it here since we can simply choose the fastest planner on the market which supports PDDL standard. The second factor depend on how the problem domain is defined. More specific, it

```
(or_subgoal2 G1 G2 G3)
(and_subgoal3 G2 G4 G5 G6)
(or_subgoal2 G4 G9 G10)
(and_subgoal2 G5 G11 G12)
(and_subgoal2 G3 G7 G8)
(and_subgoal3 G7 G13 G14 G15)
```

(a)

```
(can_depend_on A1 A2)
(can_depend_on A1 A3)
(can_depend_on A2 A4)
(can_depend_on A2 A5)
(can_depend_on A3 A5)
(can_depend_on A3 A6)
```

(b)

Figure 10.3: Elementary goal tree (a), and actors' relationship (b)

is the number of predicates used for describe the problem as well as the actions used by the planner.

Bryl et al [9] has studied the scalability of an off-the-shelf AI planner tool, LPG-td, in their experiment. The planning part of our framework is based on that of [9]. In which, the early evaluators are incorporated into the planning process. Therefore, in this experiment, we aim at studying how the growing complexity of planning problem influences the performance of the approach in comparison to Bryl's.

The planning problem files are conducted in the similar way of Bryl's, which are building from an elementary tree containing 4 decomposition levels, 15 goals ($G_i, i = \overline{1, 15}$, 2 OR and 4 AND decomposition relations (see figure 10.3(a)). All problem files contain 6 actors ($A_i, i = \overline{1, 6}$), organized into three levels with respect to the relations between them as of figure 10.3(b). Overlapping capabilities is also introduced, namely, each leaf goal is satisfied by two actors.

In the experimental result in table 10.6, we study the situation that the planning problem grows in breadth; that is, the number of top goals increase. N_{trees} represents the number of elementary trees in the problem file, N^B_{fact}, t^B_{total} are the number of facts in the problem file and the total planning time of Bryl's approach, respectively. And N^J_{fact}, t^J_{total} are those of our approach. The two last columns, $D_{fact} = \dfrac{N^J_{fact} - N^B_{fact}}{N^B_{fact}}$ shows the difference of number of facts between two approach, and $D_{fact} = \dfrac{t^J_{total} - t^B_{total}}{t^B_{total}}$ is that total time. As shown in the table, our approach has the same scalability of Bryl's. In our approach, since the number of facts in the planning problem file is approximate twice greater than that of Bryl's. Therefore, the difference of total time is not surprisingly around 20% slower.

On the other hand, the table 10.7 reports experimental result of increasing the problem complexity in depth; it means that the level of the elementary goal

N_{trees}	N^B_{fact}	N^J_{fact}	t^B_{total}	t^J_{total}	D_{fact}	D_{total}
1	31	91	0.06	0.08	1.94	0.33
2	56	170	0.53	0.64	2.04	0.21
3	81	249	2.48	2.51	2.07	0.01
4	106	328	3.7	3.73	2.09	0.01
5	131	407	8.52	10.17	2.11	0.19
6	156	486	11.89	15.23	2.12	0.28
7	181	565	15.3	19.33	2.12	0.26
8	206	644	20.25	25.21	2.13	0.24
9	231	723	28.09	31.39	2.13	0.12
10	256	802	40.01	45.21	2.13	0.13
11	281	881	58.39	63.07	2.14	0.08
12	306	960	ERR	ERR	–	–

Table 10.6: Experimental result: increasing number of elementary goal trees.

tree is increased by adding additional goal to leaf goals. The meaning of each column is exactly the same as that of table 10.6. The reported numbers are a little bit suppress since our approach and Bryl's are approximately the same. The reason could be that our approach has a the greater parsing time and the lower searching time compare with that of Bryl's. The reason of greater parsing time is obliviously because of the larger amount of facts. Meanwhile, additional early evaluators helps the planner to cut down the search space; it lead to the smaller searching time.

$Level$	N^B_{fact}	N^J_{fact}	t^B_{total}	t^J_{total}	D_{fact}	D_{total}
13	139	415	6.21	4.15	1.99	-0.33
14	151	451	8.47	5.85	1.99	-0.31
15	163	487	9.34	10.37	1.99	0.11
16	175	523	10.75	9.91	1.99	-0.08
17	187	559	13.95	12.15	1.99	-0.13
18	199	595	13.14	13.71	1.99	0.04
19	211	631	15.76	18.63	1.99	0.18
20	223	667	19.31	19.7	1.99	0.02
21	235	703	23.07	21.12	1.99	-0.08
22	247	739	27.33	25.8	1.99	-0.06
23	259	775	31.51	30.17	1.99	-0.04
24	271	811	ERR	ERR	–	–

Table 10.7: Experimental result: increasing level of elementary goal trees.

Chapter 11

Conclusions and Future Work

Socio-technical systems, which including human actors as an integral parts of the system, has emerged as a promising solution to increase the success rate of a software project. Since the human behaviors are different over time and the continuous evolving characteristic of the organizational structure, modern information systems, particularly socio-technical systems, need to adapt themselves according to the changes in the operation environment. Therefore, runtime self-reconfiguration becomes an important factor to the success of an STS.

Having studied about the self-reconfigurable STSs, we realize that the ability to automatically generate configurations and assess these configurations is essential to support runtime self-configuration. To this extend, this thesis work focuses on constructing a framework for generating and evaluating STS's configurations, which are the organizational model of an STS.

In this work, we have proposed an architecture of such a framework. To generate configuration, we employ AI planning technique in which an off-the-shelf planning tool, LPG-td, is used. The generated configuration is presented as a plan (or solution) containing of a list of actions, in which all system's goals are assigned to human actors and/or software components. To assess generated solution, our framework proposes two kind of evaluators to assess the configuration in both static view and dynamic view. While the static assessment analyzes the whole solution base on the list of actions inside, the dynamic assessment simulates the solution, then carries out assessment based on the simulated execution of the solution. The simulation is analyzing with respect to a set of events in order to evaluate the resilience of solutions.

We have also developed a runtime prototype based on the proposed architecture. The framework prototype are developed based on the Eclipse Modeling Framework and Eclipse Plugin Infrastructure, a popular Java framework, so that our framework can be easy integrated in an STS which is also built on these technologies.

In future, we are planning to fully support instance-level planning, which is currently limited supported, as well as instance-level simulation. Moreover, additional evaluators will be added to provide a wide range assessment e.g., risk analysis.

Bibliography

[1] Yudistira Asnar, Volha Bryl, and Paolo Giorgini. Using risk analysis to evaluate design alternatives. In *AOSE*, pages 140–155, 2006.

[2] G. Baxter and Ian Sommerville. Socio-technical systems: From design methods to systems engineering. Submitted to Int. J. Human Computer Studies., 2009.

[3] Carole Bernon, Marie-Perre Gleizes, Sylvain Peyruqueou, and Gauthier Picard. ADELFE: A methodology for adaptive multi-agent systems engineering. *Engineering Societies in the Agents World III*, 2577:80–81, 2003.

[4] Rafael H. Bordini, Jomi Fred Hubner, and Michael Wooldridge. *Programming multi-agent systems in AgentSpeak using Jason.* Wiley, 2007.

[5] Paolo Bresciani, Anna Perini, Paolo Giorgini, Fausto Giunchiglia, and John Mylopoulos. Tropos: An agent-oriented software development methodology. *Autonomous Agents and Multi-Agent Systems*, 8(3):203–236, 2004.

[6] Vohla Bryl, Paolo Giorgini, and John Mylopoulos. Designing cooperative is: Exploring and evaluating alternatives. In *OTM Conferences (1)*, pages 533–550, 2006.

[7] Volha Bryl and Paolo Giorgini. Self-configuring socio-technical systems: Redesign at runtime. *International Transactions on Systems Science and Applications*, 2(1):31–40, 2006.

[8] Volha Bryl and Paolo Giorgini. Automated design of socio-technical systems: From organizational structure to instance level design. In *Proc. of 23rd IEEE/ACM International Conference on Automated Software Engineering*, 2008.

[9] Volha Bryl, Paolo Giorgini, and John Mylopoulos. Designing socio-technical systems: from stakeholder goals to social networks. *Requir. Eng.*, 14(1):47–70, 2009.

[10] K. M. Carley. Computational organization science: A new frontier. *Proceedings of the National Academy of Science*, 33(3):7314–7316, 2002.

[11] Luca Cernuzzi and Franco Zambonelli. Dealing with adaptive multi-agent organizations in the gaia methodology. *Agent-Oriented Software Engineering VI*, 3950:109–123, 2006.

[12] A.B. Cherns. Principles of sociotechnical design revisited. *Human Relations*, 40(3):154–162, 1987.

[13] L. K. Comfort, M. Hauskrecht, and J. Lin. Dynamic networks: Modeling change inenvironments exposed to risk. In *5th International Conference on Information Systems for Crisis Response and Management*, 2008.

[14] F Dalpiaz, R Ali, Y Asnar, V Bryl, and P Giorgini. Applying tropos to socio-technical system design and runtime configuration. In *In Proceeding of the Italian workshop Dagli OGGETTI agli AGENTI (WOA'08)*, 2008.

[15] Fabiano Dalpiaz, Paolo Giorgini, and John Mylopoulos. Talos: an architecture for self-reconfiguration. Technical report, DISI-08-026, 2008.

[16] Rajdeep K. Dash, Nicholas R. Jennings, and David C. Parkes. Computational-mechanism design: A call to arms. *IEEE Intelligent Systems*, 18(6):40–47, 2003.

[17] Eclipse. Eclipse modeling framework project (emf). http://www.eclipse.org/modeling/emf/.

[18] Stefan Edelkamp and J Hoffmann. Pddl2.2: The language for the classical part of the 4th international planning competition. Technical Report 195, January 2004.

[19] Stefan Edelkamp and Jörg Hoffmann. Pddl2.2: The language for the classical part of the 4th international planning competition. Technical Report 195, January 2004.

[20] F. E. Emery. "designing socio-technical systems for greenfield sites. *Journal of Occupational Behavior*, 1(1):19–27, 1980.

[21] F.E. Emery. Characteristics of socio-technical systems, 1959. London: Tavistock.

[22] M. S. Feather, S. Fickas, A. Van Lamsweerde, and C. Ponsard. Reconciling system requirements and runtime behavior. In *IWSSD '98: Proceedings of the 9th international workshop on Software specification and design*, pages 50–59, Washington, DC, USA, 1998. IEEE Computer Society.

[23] Standish Group. Extreme chaos 2001. Technical report, The Standish International Inc, 2001.

[24] Axel Van Lamsweerde. Divergent views in goal-driven requirements engineering. In *Joint Proceedings of the Sigsoft '96 Workshops – Specifications '96, ACM*, pages 252–256. Press, 1996.

[25] N. G. Leveson, M. Daouk, N. Dulac, and K. Marais. Applying stamp in accident analysis. In *NASA CONFERENCE PUBLICATION*, pages 177–198, 2003.

[26] Lin Liu and Eric Eric Yu. Designing information systems in social context: A goal and scenario modelling approach. *Info. Syst*, 29:187–203, 2003.

[27] A. Majchrzak and B. Borys. Generating testable socio-technical systems theory. *Journal of Engineering and Technology Management*, 18:219–240, 2001.

[28] LPG Home page. Lpg-td planner. http://zeus.ing.unibs.it/lpg/.

[29] Zahid H. Qureshi. A review of accident modelling approaches for complex socio-technical systems. In *SCS '07: Proceedings of the 12th Australian workshop on Safety critical systems and software and safety-related programmable systems*, pages 47–59, Darlinghurst, Australia, Australia, 2007. Australian Computer Society, Inc.

[30] Gnter Ropohl. Philosophy of socio-technical systems. *In Society for Philosophy and Technology*, 4:59–71, 1999.

[31] Walt Scacchi. Socio-technical design. *The Encyclopedia of HumanComputer Interaction*, pages 656–659, 2004.

[32] SPEARS: Scalable Personal Area Services. Deliverable d2.1. disater management - scenatio and use cases. https://doc.freeband.nl/dsweb/Get/Document-31938/Spears

[33] Herbert A. Simon. *The Sciences of the Artificial, 3rd Edition.* MIT Press, 1996.

[34] Ian Sommerville. *Software Engineering 8th Edt*, chapter Socio-Technical system, pages 21–42. Pearson, 2006.

[35] Dave Stenberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework, 2nd Edition.* Addison-Wesley Professional, 2008.

[36] Eric Lansdown Trist, Hugh. Murray, and F.E. Emery. *The social engagement of social science: a Tavistock anthology.* University of Pennsylvania Press, 1990.

[37] Guy H. Walker, Neville A. Stanton, Paul M. Salmon, and Daniel P. Jenkins. A review of sociotechnical systems theory: a classic concept for new command and control paradigms. *Theoretical Issues in Ergonomics Science*, 9(6):479–499, November 2008.

[38] D.S. Weld. Recent advances in ai planning. *AI Magazine*, 20(2):93–123, 1999.

[39] Eric Siu-Kwong Yu. *Modelling strategic relationships for process reengineering.* PhD thesis, Toronto, Ont., Canada, Canada, 1996.

[40] Nicolla Zannone. *A Requirement Engineering Methodology for Trust, Security, and Privacy.* PhD thesis, University of Trento, 2006.