A Game of Attribute Decomposition for Software Architecture Design

Jiamou Liu and Ziheng Wei

School of Computer and Mathematical Sciences
Auckland University of Technology, New Zealand
{jiamou.liu, bys7090}@aut.ac.nz

Abstract. Attribute-driven software architecture design aims to provide decision support by taking into account the quality attributes of softwares. A central question in this process is: *what architecture design best fulfills the desirable software requirements?* To answer this question, a system designer needs to make tradeoffs among several potentially conflicting quality attributes. Such decisions are normally ad-hoc and rely heavily on experiences. We propose a mathematical approach to tackle this problem. Game theory naturally provides the basic language: players represent requirements, and strategies involve setting up coalitions among the players. In this way we propose a novel model, called *decomposition game (DG)*, for attribute-driven design. We present its solution concept based on the notion of cohesion and expansion-freedom and prove that a solution always exists. We then investigate the computational complexity of obtaining a solution. The game model and the algorithms may serve as a general framework for providing useful guidance for software architecture design. We present our results through running examples and a case study on a real-life software project.

Key words: Software architecture, coalition game, decomposition game

1 Introduction

Computational game theory studies the algorithmic nature of conflicting entities and establishes *equilibria*: a state of balance that minimises the negative effects among players. The field has attracted much attention in the recent 10-15 years due to applications in multi-agent systems, electronic markets and social networks [9,10,11]. In this paper, we investigate the problem of software architecture design from a game theory perspective. In particular, we provide a novel model, called *decomposition game*, which captures interactions among software requirements and derives a software architecture through equilibria.

The architecture of a software system lays out its basic composition. It has been a common belief that architecture design heavily influences the quality attributes such as performance, reliability, usability and security of a software system [2]. A major objective of architecture design is therefore the assurance of non-functional requirements through compositional decisions. In other words, we need to answer the following question: *what architecture best fulfills the desirable software requirements?* There is, however, usually no "perfect" architecture that fulfills every requirement. For example, performance and security are both key non-functional requirements, which may demand fast response time to the users, and the application of a sophisticated encryption algorithm, respectively. These two requirements are in intrinsic conflict, as a strong focus of one will negatively impact the fulfilment of the other. A main task of the software architect, therefore, is to balance such “interactions” among requirements, and decide on appropriate tradeoffs among such conflicting requirements.

While it is a common practice to decide on software architecture designs through the designers’ experiences and intuition, formal approaches for architecture design are desirable as they facilitate standardisation and automation of this process, providing rigorous guidelines, allowing automatic analysis and verifications [6]. Notable formal methods in software architecture include a large number of formal architecture description languages (ADL), which are useful tools in communicating and modeling architectures. However, as argued by [14], industry adoptions of ADL are rare due to limitations in usability and formality. Other algorithmic methods for software architecture design include employing hierarchical clustering algorithms to decompose components based on their common attributes [8], as well as quantifying tradeoffs between requirements [1].

In this paper, we propose to use computational game theory as a mathematical foundation for conceptualising software architecture designs from requirements. Our motivation comes from the following two lines of research:

(1). Attribute driven design (ADD) : ADD is a systematic method for software architecture design. The method was invented by Bass, Klein and Bachmann in [4] and subsequently updated and improved through a sequence of works [3,13]. The goal is to assist designers to analyse quality attribute tradeoffs and provide design suggestions and guidance. Inputs to ADD are functional and non-functional requirements, as well as design constraints; outputs to ADD are conceptual architectures which outline coarse-grained system compositions. The method involves a sequence of well-defined steps that recursively decompose a system to components, subcomponents, and so on. These steps are not algorithmic: they are meant to be followed by system designers based on their experience and understanding of design principles. As mentioned by the authors in [4], an ongoing effort is to investigate rigorous approaches in producing conceptual architectures from requirements, hence enabling automated design recommendation under the ADD framework. To this end, we initiate a game-theoretic study to formulate the interactions among software requirements so that a conceptual architecture can be obtained in an algorithmic way.

(2). Coalition game theory : A coalition game is one where players exercise collaborative strategies, and competition takes place among coalitions of players rather than individuals. In ADD, we can imagine each requirement is “handled” by a player, whose goal is to set up a coalition with others to maximise the collective payoff. The set of coalitions then defines components in a system decomposition which entails a software architecture. This fits into the language of coalition games. However, the usual axioms in coalition games specify superadditivity and monotonicity, that is, the combination of two coalitions is always

more beneficial than each separate coalition, and the payoff increases as a coalition grows in size. Such assumptions are not suitable in this context as combination of two conflicting requirements may result in a lower payoff. Hence a new game model is necessary to reflect the conflicting nature of requirements. In this respect, we propose that our model also enriches the theory of coalition games.

Our contribution. In Section 2, we provide a formal framework which, following the ADD paradigm [4], recursively decomposes a system into sub-systems; the final decomposition reveals design elements in a software architecture. The basis of the framework is an algorithmic realisation of ADD. A crucial task in this algorithmic realisation is *system decomposition*, which derives a rational decomposition of an attribute primitive.

In Section 3, we propose a *decomposition game* to capture system decomposition. The game takes into account *interactions* between requirements, which express the positive (enhancement) or negative (canceling) effects they act on each other. A *solution concept* (equilibrium) defines a rational decomposition, which is based on the notions of *cohesion* and *expansion-freedom*. We demonstrate that any such game has a solution, and a solution may not be unique.

In Section 4, we study algorithms that compute solutions for the decomposition game. Finding cohesive coalitions with maximal payoff turns out to be NP-hard (Thm. 12). Hence we propose a relaxed notion of k*-cohesion* for $k \geq 1$, and present a polynomial time algorithm for finding a k-cohesive solution of the game (Thm. 15). To demonstrate the practical significance our the framework, we implement the framework and perform a case study on a real-world Cafeteria Ordering System in Section 5.

2 Algorithmic Attribute Driven Design (ADD) Process

ADD is a general framework for transforming software requirements into a *conceptual software architecture*. Pioneers of this approach introduced it through several well-formed, but informally-defined concepts and steps [4,13]. A natural question arises whether it can be made more algorithmic, which provides unbiased, mathematically-grounded outputs. To answer this question, one would first need to translate the original informal descriptions to a mathematical language.

2.1 Software Requirements and Constraints

Functional requirements. Functional requirements are specifications of what tasks the system perform (e.g. “the system must notify the user once a new email arrives”).A functional requirement does not stand alone; often, it acts with other functional requirements to express certain combined functionality (e.g. “the user should log in before making a booking”). Thus, a functionality may depend on other functionalities. We use a partial ordering $(\mathsf{F}, \prec)$ to denote the functional requirements where each $r \in \mathsf{F}$ is a functional requirement, and $r_1 \prec r_2$ denotes that r_1 depends on r_2. Note that $\prec$ is a transitive relation.

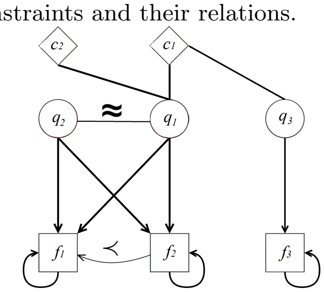
Non-functional requirements. Non-functional requirements specify the desired quality attributes; ADD uses *general scenarios* and *scenarios* as their standard representations. A general scenario is a high-level description on what it means to achieve a non-functional requirement [4]. For example, the general scenario "*A failure occurs and the system notifies the user; the system continues to perform in a degraded manner*" refers to the availability attribute. There has been an effort to document all common general scenarios; a rather full list is given in [3]. Note that a general scenario is vaguely-phrased and is meant to serve as a template for more concrete "instantiations" of quality attributes. Such "instantiations" are called scenarios. More abstractly, we use a pair $(\mathsf{S}, \approx)$ to denote the non-functional requirements where S is a set of scenarios and $\approx$ is an equivalence relation on S, denoting the *general scenario relation*: $q_1 \approx q_2$ means that q_1 and q_2 instantiates the same general scenario.

Design constraints. Design constraints are factors that must be taken into account and enforce certain design outcomes. A design constraint may affect both functional and non-functional requirements. More abstractly, we use a collection of sets $\mathsf{C} \subseteq 2^{\mathsf{F} \cup \mathsf{S}}$ to denote the set of design constraints, where each set $c \in \mathsf{C}$ is a design constraint. Intuitively, if two requirements r_1, r_2 belong to the same $c \in \mathsf{C}$, then they are constrained by the same design constraint c.

Derived Functionalities. The enforcement of certain quality attributes may lead to additional functionalities. For example, to ensure availability, it may be necessary to add extra functionalities to detect failure and automatically bypass failed modules. Hence we introduce a *derivation relation* $\hookrightarrow \subseteq \mathsf{S} \times \mathsf{F}$ such that $r \hookrightarrow s$ means the functional requirement s is derived from the scenario r.

2.2 Attribute Primitives

Fig. 1. Example 1: The requirements, constraints and their relations.



The intentional outcome of ADD describes the *design elements*, i.e., subsystems, components or connectors. It is important to note that the goal of ADD is not the complete automation of the design process, but rather, to provide useful guidance. Thus, the conceptual view reveals only the organisational structure but not the concrete design.

An attribute primitive is a set of design elements that collaboratively perform certain functionalities and meet one or more quality requirements; it is also the minimal combination with respect to these goals [4]. Examples of attribute primitives include data router, firewall, virtual machine, interpreter and so on. ADD prescribes a list of attribute primitives together with descriptions of their properties and side effects (such as in [3]). Hence, ADD essentially can be viewed as assigning the right attribute primitives to the right requirement combinations. Note also that an attribute primitive may be broken down further.

Definition 1 (Attribute primitive). *An* attribute primitive *is a tuple*

$$\mathcal{A} = (\mathsf{F}, \mathsf{S}, \mathsf{C}, \prec, \approx, \hookrightarrow)$$

where F *is a set of functional requirements,* S *is a set of scenarios,* $\mathsf{C} \subseteq 2^{\mathsf{F} \cup \mathsf{S}}$ *is a set of design constraints,* $\prec$ *is the dependency relation on* F*,* $\approx$ *is the general scenario relation of* S*, and* $\hookrightarrow \subseteq \mathsf{S} \times \mathsf{F}$ *is a derivation relation.*

Let $\mathcal{A} = (\mathsf{F}, \mathsf{S}, \mathsf{C}, \prec, \approx, \hookrightarrow)$ be an attribute primitive. We also need to define:

- A *requirement* of $\mathcal{A}$ is an element in the set $\mathsf{R} := \mathsf{F} \cup \mathsf{S}$.
- For $r \in \mathsf{F}$, the *dependency set* of r is the set $f(r) := \{r' \in \mathsf{F} \mid r \preceq r'\}$.
- For $r \in \mathsf{S}$, the *general scenario* of r is the set $g(r) := \{r' \in \mathsf{S} \mid r \approx r'\}$, i.e., the $\approx$-equivalence class of r.
- For $r \in \mathsf{R}$, the *constraints* of r is the set $c(r) := \{t \in \mathsf{C} \mid r \in t\}$.
- For $r \in \mathsf{S}$, the *derived set* of r is $d(r) := \{s \in \mathsf{F} \mid r \hookrightarrow s\}$, and for $s \in \mathsf{F}$, let $d^{-1}(s) := \{r \in \mathsf{S} \mid r \hookrightarrow s\}$

Definition 2 (Design element). *A* design element *of $\mathcal{A}$ is a subset $D \subseteq \mathsf{R}$. An* decomposition *of $\mathcal{A}$ is a sequence of design elements $\boldsymbol{D} = (D_1, D_2, \dots D_k)$ where $k \geq 1$, $\bigcup_{1 \leq i \leq k} D_k = \mathsf{R}$, and each $D_i \cap D_j = \varnothing$ for any $i \neq j$.*

Example 1. Fig. 1 shows an attribute primitive $\mathcal{A} = (\mathsf{F}, \mathsf{S}, \mathsf{C}, \prec, \approx, \hookrightarrow)$

- $\mathsf{F} = \{f_1, f_2, f_3\}$ and $\mathsf{S} = \{q_1, q_2, q_3\}$ are the requirements
- $\mathsf{C} = \{c_1, c_2\}$ where $c_1 = \{q_1, q_3\}, c_2 = \{q_1\}$
- $f_1 \prec f_2$, $q_1 \approx q_2$, $q_1 \hookrightarrow f_1$, $q_1 \hookrightarrow f_2$,$q_2 \hookrightarrow f_1$, $q_2 \hookrightarrow f_2$, $q_3 \hookrightarrow f_3$.

2.3 The ADD Procedure

Essentially ADD provides a means for system decomposition: The entire system is treated as an attribute primitive, which is the input. At each step, the procedure decomposes an attribute primitive $\mathcal{A}$ by identifying a decomposition $(D_1, D_2, \dots, D_k)$. The process then maps each resulting design element D_i to an attribute primitive $\mathcal{A}_i = (\mathsf{F}_i, \mathsf{S}_i, \mathsf{C}_i, \prec_i, \approx_i, \hookrightarrow_i)$, which contains all elements in D_i and may require some further requirements and constraints. Hence we require that $D_i \subseteq \mathsf{F}_i \cup \mathsf{S}_i$ and $\prec_i$, $\approx_i$, C_i, $\hookrightarrow_i$ are consistent with $\prec$, $\approx$, C and $\hookrightarrow$ on D_i, resp.; in this case we say that $\mathcal{A}_i$ is *consistent* with D_i. Thus the attribute primitive $\mathcal{A}$ is decomposed into k attribute primitives $\mathcal{A}_1, \mathcal{A}_2, \dots, \mathcal{A}_k$. On each $\mathcal{A}_i$ where $1 \leq i \leq k$, the designer may choose to either terminate the process, or start a new step recursively to further decompose $\mathcal{A}_i$. See Procedure 1.

Procedure 1 $\mathsf{ADD}(\mathcal{A})$ (General Plan)

```
1: (D_1, D_2, ..., D_k) ← Decompose(A) // compute a rational decomposition of A
2: for 1 ≤ i ≤ k do
3:     A_i ← an primitive attribute consistent with D_i
4:     if A_i needs further decomposition then
5:         ADD(A_i)
```

We point out that the ADD procedure, as presented by its original proponents, involves numerous additional stages other than the ones described above [13]. The reason we choose this over-simplified description is that we believe these are the steps that could be rigorously presented, and they abstractly capture in a way most of the steps mentioned in the original informal description.

The $\mathsf{Decompose}(\mathcal{A})$ operation produces a rational decomposition $(D_1, \dots, D_k)$ of the input attribute primitive $\mathcal{A}$ that satisfies the requirements of $\mathcal{A}$. We also note that $\mathsf{Decompose}(\mathcal{A})$ amounts to a crucial step in the ADD process, as the decomposition determines to a large extend how well the quality attributes are met. This step is also a challenging one as interactions among quality attributes create potential conflicts. Thus, in the next section, we define a game model which allows us to automate the $\mathsf{Decompose}(\mathcal{A})$ operation.

3 Decomposition Games

3.1 Requirement Relevance

Let $\mathcal{A} = (\mathsf{F}, \mathsf{S}, \mathsf{C}, \prec, \approx, \hookrightarrow)$ be an attribute primitive. Relevance between requirements are determined by the relations $\prec, \approx, \hookrightarrow$ and the constraint set C. In the following the *Jaccard index* $J(S_1, S_2)$ measures the similarity between two sets S_1, S_2 with $J(S_1, S_2) = \frac{|S_1 \cap S_2|}{|S_1 \cup S_2|}$.

Definition 3 (Relevance). *Two requirements $r_1, r_2 \in \mathsf{R}$ are* relevant *if*

- *$r_1, r_2 \in \mathsf{F}$, and either $d^{-1}(r_1) \cap d^{-1}(r_2) \neq \varnothing$ (derived from some common scenario), or $f(r_1) \cap f(r_2) \neq \varnothing$ (relevant through dependency), or $c(r_1) \cap c(r_2) \neq \varnothing$ (share some common design constraints).*
- *$r_1, r_2 \in \mathsf{S}$, and either $r_1 \approx r_2$ (instantiate the same general scenario), or $d(r_1) \cap d(r_2) \neq \varnothing$ (jointly derives some functionality) or $c(r_1) \cap c(r_2) \neq \varnothing$.*
- *$r_1 \in \mathsf{F}$, $r_2 \in \mathsf{S}$, and either $f(r_1) \cap d(r_2) \neq \varnothing$ (r_1 depends on a requirement that is derived from r_2), or $c(r_1) \cap c(r_2) \neq \varnothing$.*

We define the *relevance index* $\sigma(r_1, r_2)$ of $r_1 \neq r_2 \in \mathsf{R}$ as a real number:

1. if two functional requirements $r_1, r_2 \in \mathsf{F}$ are relevant, then
$$\sigma(r_1, r_2) = \alpha J(d^{-1}(r_1), d^{-1}(r_2)) + \beta J(f(r_1), f(r_2)) + \gamma J(c(r_1), c(r_2));$$
2. if two scenarios $r_1, r_2 \in \mathsf{S}$ are relevant, then
$$\sigma(r_1, r_2) = \beta J(d(r_1), d(r_2)) + \gamma J(c(r_1), c(r_2));$$
3. If $r_1 \in \mathsf{F}$ and $r_2 \in \mathsf{S}$ are relevant, then
$$\sigma(r_1, r_2) = \sigma(r_2, r_1) = \beta J(f(r_1), d(r_2)) + \gamma J(c(r_1), c(r_2));$$
4. otherwise, $\sigma(r_1, r_2) = \lambda$

The constants α, β, γ are positive real numbers, that represent weights on the overlaps in d_1, d_2's generated sets , dependency sets and constraints, respectively. We require $\alpha + \beta + \gamma = 1$. The constant λ is a negative value that represents a "penalty" one pays when two irrelevant requirements get in the same design element. For simplicity, we do not include these constants in expressing the function σ, and all subsequent notions that depend on σ.

Example 2. Continue from $\mathcal{A}$ in Example 1. To emphasise the non-functional requirements we give a larger weight to α, setting $\alpha = 0.5$, $\beta = 0.4$, $\gamma = 0.1$. We also set $\lambda = -0.5$. Then $\sigma(r_1, r_2) = 0.4 \times \frac{2}{2} = 0.4$ for any $(r_1, r_2) \in \{(q_1, q_2), (q_3, f_3)\} \cup (\{q_1, q_2\} \times \{f_1, f_2\})$; $\sigma(q_1, q_3) = 0.1 \times \frac{1}{2} = 0.05$; $\sigma(f_1, f_2) = 0.5 \times \frac{2}{2} + 0.4 \times \frac{2}{2} = 0.9$; and relevance between any other pairs is -0.5. Fig. 2(a) illustrates the (positive) relevance in a weighted graph.

3.2 Decomposition Games

We employ notions from coalition games to define what constitutes a *rational* decomposition. In a coalition game, players cooperate to form coalitions which achieve certain collective payoffs [5].

Definition 4 (Coalition game). *A* coalition game *is a pair (P, ν) where P is a finite set of players, and each subset $D \subseteq P$ is a* coalition*; $\nu : 2^P \to \mathbb{R}$ is a* payoff function *associating every $D \subseteq P$ a real value $\nu(D)$ satisfying $\nu(\varnothing) = 0$.*

This provides the set up for decompositions: Imagine a coalition game consisting of $|\mathsf{R}|$ agents as players, where each agent is in charge of a different requirement. The players form coalitions which correspond to sets of requirements, i.e., design elements. The payoff function would associate with every coalition a numerical value, which is the payoff gained by each member of the coalition. Therefore, an equilibrium of the game amounts to a decomposition with the right balance among all requirements – this would be regarded as a rational decomposition.

It remains to define the payoff function. Naturally, the payoff of a coalition is determined by the *interactions* among its members. Take $r_1, r_2 \in D$. If one of r_1, r_2 is a functional requirement, then their interaction is defined by their relevance index $\sigma(r_1, r_2)$, as higher relevance means a higher level of interaction. Suppose now both r_1, r_2 are scenarios (non-functional). Then the interaction becomes more complicated, as a quality attribute may enhance or defect another quality attribute. In [12, Chapter 14], the authors identified effects acting from one quality attribute to another, which is expressed by a *tradeoff matrix* T:

- T has dimension $m \times m$ where m is the number of general scenarios
- For $i \neq j \in \{1, \ldots, m\}$, the (i, j)-entry $T_{i,j} \in \{-1, 0, 1\}$.

Let $g_1, g_2, \ldots, g_m$ be general scenarios. $T_{i,j} = 1$ (resp. $T_{i,j} = -1$) means g_1 has a positive (resp. negative) effect on g_2, $T_{i,j} = 0$ means no effect. E.g., the tradeoff matrix defined on six common quality attributes is:

	Perfo.	Modif.	Secur.	Avail.	Testa.	Usabi.
Performance	0	−1	0	0	0	−1
Modifiability	−1	0	0	1	1	0
Security	−1	0	0	1	−1	−1
Availability	0	0	0	0	0	0
Testability	0	1	1	1	0	1
Usability	−1	0	0	0	−1	0

Note that the matrix is not necessarily symmetric: the effect from g_1 to g_2 may be different from the effect from g_2 to g_1. For example, an improvement in system performance may not affect security, but increasing security will almost always adversely impact performance. we assume that the matrix T is given prior to ADD; this assumption is reasonable as there is an effective map from any general scenario to the main quality attribute it tries to capture. We use this tradeoff matrix to define the interaction between two scenarios in S.

Definition 5 (Coalitional relevance). *For a coalition $D \subseteq \mathsf{R}$ and $r \in D$, the* coalitional relevance *of r in D is the total relevance from r to all other requirements in D, i.e., $\rho(r, D) = \sum_{s \in D, s \neq r} \sigma(r, s)$.*

Definition 6 (Effect factor). *For scenarios r_1, r_2 in the same coalition D, the* effect factor *from r_1 to r_2 expresses the effect of r_1 towards r_2, i.e.,*

$$\varepsilon(r_1, r_2, D) = \begin{cases} -|\rho(r_1, D)| & \text{if } T(g(r_1), g(r_2)) = -1 \\ 0 & \text{if } T(g(r_1), g(r_2)) = 0 \\ \rho(r_1, D) & \text{if } T(g(r_1), g(r_2)) = 1 \end{cases}$$

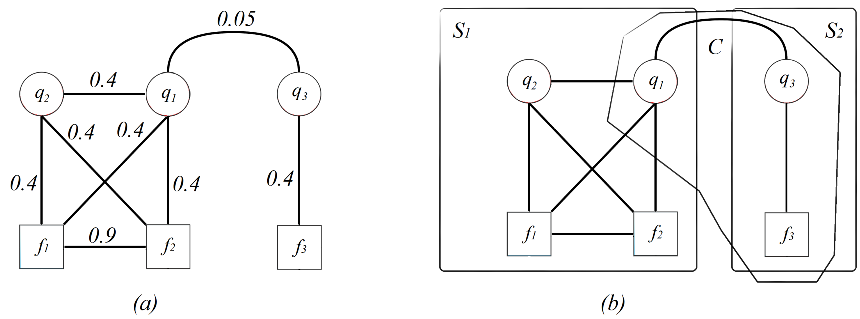
Definition 7 (Interaction). *Let $r_1 \neq r_2 \in \mathsf{R}$ be requirements. The* interaction *between r_1, r_2 is simply the relevance $\sigma(r_1, r_2)$ if one of r_1, r_2 is functional; otherwise (both r_1, r_2 are non-functional), it is the sum of their effect factors, i.e., the* interaction $\nu(r_1, r_2, D) := \begin{cases} \sigma(r_1, r_2) & \text{if } \{r_1, r_2\} \cap \mathcal{F} \neq \varnothing \\ \varepsilon(r_1, r_2, D) + \varepsilon(r_2, r_1, D) & \text{otherwise} \end{cases}$

The coalition utility $\nu(D)$ *of any coalition* $D \subseteq \mathsf{R}$ *is defined as the sum of interactions among all pairs of requirements in the coalition, i.e.,*

$$\nu(D) = \sum_{r_1 \neq r_2 \in D} \nu(r_1, r_2, D)$$

Definition 8 (Decomposition games (DG)). *Let* $\mathcal{A} = (\mathsf{F}, \mathsf{S}, \mathsf{C}, \prec, \approx, \hookrightarrow)$ *be an attribute primitive. The* DG $G_{\mathcal{A}}$ *is the coalition game* $(\mathsf{F} \cup \mathsf{S}, \nu)$ *where* $\nu : 2^{\mathsf{F} \cup \mathsf{S}} \to \mathbb{R}$ *is the coalition utility function.*

Fig. 2. (a) Weights on the edges are relevance (function σ) between requirements in Example 2; the diagram omits the negative weighted pairs. (b) The decomposition $\{S_1, S_2\}$ is a solution with $\nu(S_1) = 2.5$, $\nu(S_2) = 0.4$. The coalition C has $\nu(C) = -1$



Example 3 (Coalition Utility). Continue the setting in Example 2. Let the general scenarios be $g_1 = \{q_1, q_2\}$ and $g_2 = \{q_3\}$. We assume matrix T specifies $T(g_1, g_2) = 1$ and $T(g_2, g_1) = -1$. Consider the coalition $C = \{q_1, q_3, f_3\}$. We have: $\rho(q_1, C) = 0.05 - 0.5 = -0.45$; $\rho(q_3, C) = 0.4 + 0.05 = 0.45$. So $\varepsilon(q_1, q_3, C) = -0.45 \times 1 = -0.45$; $\varepsilon(q_3, q_1, C) = 0.45 \times (-1) = -0.45$. Thus $\nu(q_1, q_3, C) = -0.45 - 0.45 = -0.9$. Therefore, $\nu(C) = \sigma(q_1, f_3) + \sigma(q_3, f_3) + (-0.9) = (-0.5) + 0.4 + (-0.9) = -1$ but $\nu(C \setminus \{q_1\}) = \nu(\{q_3, f_3\}) = \sigma(q_3, f_3) = 0.4$; See Fig. 2(b).

Note that the above matrix T indicates that q_1 will act positively to q_3. Furthermore q_1, q_3 have a positive (0.05) relevance. However, adding q_1 into the coalition of $\{q_3, f_3\}$ drastically decreases the coalition utility.

3.3 Solution Concept

We point out some major differences between decomposition and typical coalition games: Firstly, in coalition game theory, one normally assumes the axioms of superadditivity ($\nu(D_1 \cup D_2) \geq \nu(D_1) + \nu(D_2)$) and monotonicity ($D_1 \subseteq D_2 \Rightarrow \nu(D_1) \leq \nu(D_2)$) which would obviously not hold for decomposition as players may counteract with each other, reducing their combined utility. Secondly, the typical solution concepts in coalition games (such as Pareto optimality, and Shapely value) focus on distribution of payoffs to each individual player assuming a grand coalition consisting of all players. In decomposition such a grand coalition is normally not desirable and the focus is on the overall payoff of each coalition D, rather than the individual requirements. The above differences motivate us to consider a different solution concept of DG $G_{\mathcal{A}}$. At any instance of the game, the players form a decomposition $(D_1, D_2, \dots, D_k)$. We assume the players may perform two collaborative strategies:

1. *Merge strategy*: Two coalitions may choose to merge if they would obtain a higher combined payoff.
2. *Bind strategy*: Players within the same coalition may form a sub-coalition if they would obtain a higher payoff.

Example 4 (A Dilemma). We present an example demonstrating the dynamics of a DG $G_\mathcal{A}$. This example shows a real-world dilemma: as a coalition pursues higher utility through expansion (merging with others), it may be better to choose a "less-aggressive" expansion strategy over the "more-aggressive" counterpart, even though the latter clearly brings a higher payoff. Assume the following situation (which is clearly plausible in an attribute primitive):

– $\mathsf{R} = \{d_1, d_2, d_3, d_4\}$ where $\mathsf{S} = \{d_1, d_4\}$ and $d_1 \not\approx d_4$.
– We set $\sigma(\{d_1, d_2\}) = \sigma(\{d_1, d_3\}) = \sigma(\{d_2, d_3\}) = 0.1$, and $\sigma(\{d_2, d_4\}) = 0.5$.
– The tradeoff matrix indicates $T(g(d_1), g(d_4)) = 0$, $T(g(d_4), g(d_1)) = -1$.
– And, d_1 and d_4 are irrelevant, namely $\sigma(d_1, d_4) = \lambda = -0.7$.

Suppose we start with the decomposition $\{S = \{d_1, d_2\}, \{d_3\}, \{d_4\}\}$. Then $\nu(S) = \rho(d_1, d_2, S) = \nu(d_1, d_2, S) = 0.1$. Coalition S has two merge strategies:

(1) For $S_1 = S \cup \{d_3\}$: $\nu(d_1, d_2, S_1) = \sigma(d_1, d_2) = 0.1$, $\nu(d_1, d_3, S_1) = \sigma(d_1, d_3) = 0.1$, $\nu(d_2, d_3, S_1) = \sigma(d_2, d_3) = 0.1$. Thus $\nu(S_1) = 0.3$.
(2) For $S_2 = S \cup \{d_4\}$: $\nu(d_1, d_4, S_2) = \varepsilon(d_4, d_1, S_2) = -0.7{+}0.5 = -0.2$, $\nu(d_1, d_2, S_2) = \sigma(d_1, d_2) = 0.1$, $\nu(d_2, d_4, S_2) = \sigma(d_2, d_4) = 0.5$. Hence $\nu(S_2) = 0.1{–}0.2{+}0.5 = 0.4$

Merging with $\{d_4\}$ clearly results in a higher payoff for the combined coalition. However, if this merge happens, as $\nu(\{d_2, d_4\}) = 0.5 > \nu(S_2) = 0.4$, d_2 and d_4 would choose to bind together, hence leaving S_2. This would be undesirable if d_1 is a critical non-functional requirement for d_2.

Example 4 shows that a solution concept would be a decomposition where no "expansion" nor "crumbling" occur to any coalition.

Definition 9 (Solution). *Let $\boldsymbol{D} = (D_1, \ldots, D_k)$ be a decomposition of $\mathcal{A}$.*

1. *A coalition $D \subseteq \mathsf{R}$ is* cohesive *if for all $C \subseteq D$, $\nu(C) < \nu(D)$; $\boldsymbol{D}$ is* cohesive *if so is every D_i.*
2. *A coalition D_i is* expansion-free *with respect to $\boldsymbol{D}$ if $\max\{\nu(D_i), \nu(D_j)\} > \nu(D_i \cup D_j)$; $\boldsymbol{D}$ is* expansion-free *if so is every D_i.*

A solution *of a DG is a decomposition that is both cohesive and expansion-free.*

Example 5 (Solution). Continue from Example 3, the utilities for

$S_1 = \{q_1, q_2, f_1, f_2\}$ and $S_2 = \{q_3, f_3\}$ are :

– S_1: $\nu(q_1, q_2, S_1) = 0$, $\nu(q_1, f_1, S_1) = \nu(q_1, f_2, S_1) = \nu(q_2, f_1, S_1) = 0.4$, $\nu(q_1, f_2, S_1) = 0.4$, $\nu(f_1, f_2, S_1) = 0.9$. Thus $\nu(S_1) = 0.4 \times 4 + 0.9 = 2.5$
– S_2: $w_(q_3, f_3, S_2) = 0.4$. Thus $\nu(S_2) = 0.4$

Both S_1 and S_2 are cohesive. Furthermore, we have $\nu(q_1, q_3, \mathsf{R}) = 0.75{–}1.05 = -0.3$ and $\nu(q_2, q_3, \mathsf{R}) = 0.2 - 1.05 = -0.85$. Thus $\nu(\mathsf{R}) = 2.9{–}0.5{\times}6{–}0.85{–}0.3 = -1.45$. Consequently, $\{S_1, S_2\}$ is also expansion-free, and is thus a solution of the game.

A solution of a DG $G_\mathcal{A}$ corresponds to a rational decomposition of the attribute primitive $\mathcal{A}$. As shown by Thm. 10, any attribute primitive admits a solution, and rather expectedly, a solution may not be unique.

Theorem 10 (Solution Existence). *There exists a solution in any DG $G_{\mathcal{A}}$.*

Proof. We show existence of a solution by construction. Let $(D_1, D_2, \ldots, D_k)$ be a longest sequence such that for any $i = 1, \ldots, k$, D_i is a minimal coalition with maximal utility in $\mathsf{R} \setminus \{D_1, \ldots, D_{i-1}\}$ (i.e., $\forall D \subseteq \mathsf{R} \setminus \{D_1, \ldots, D_{i-1}\} : \nu(D_1) \geq \nu(D)$ and $\forall D \subseteq D_1 : \nu(D_1) > \nu(D)$).

We claim that $\boldsymbol{D} = (D_1, \ldots, D_k)$ is a solution in $G_{\mathcal{A}}$. Indeed, for any $1 \leq i \leq k$, any proper subset of D_i would have payoff strictly smaller than $\nu(D_i)$ by minimality of D_i. Thus $\boldsymbol{D}$ is cohesive. Moreover, if $\nu(D_i \cup D_j) > \min\{\nu(D_i), \nu(D_j)\}$ for some $i \neq j$, then $D_{\min\{i,j\}}$ does not have maximal utility in $\mathsf{R} \setminus \{D_1, \ldots, D_{\min\{i,j\}-1}\}$. Hence $\boldsymbol{D}$ is expansion-free. □

Proposition 11. *The solution of a DG may not be unique.*

Proof. Let $\mathcal{A} = (\mathsf{F}, \mathsf{S}, \mathsf{C}, \prec, \approx, \hookrightarrow)$ be an attribute primitive where $\mathsf{S} = \varnothing$ and $\mathsf{F} = \{d_1, d_2, \ldots, d_6\}$. We may define $\mathsf{C}, \prec, \approx, \hookrightarrow$ in such a way that

- For all $\{i, j\} \subseteq \{1, 2, 3, 4\}$ and $\{i, j\} \subseteq \{4, 5, 6\}$, $i \neq j \Rightarrow \nu(\{d_i, d_j\}) = 0.1$
- For all $i \in \{1, 2, 3\}$, $j \in \{5, 6\}$, $\nu(\{d_i, d_j\}) = -0.1$

Consider $\boldsymbol{C} = \{C_1 = \{d_1, d_2, d_3\}, C_2 = \{d_4, d_5, d_6\}\}$ and $\boldsymbol{D} = \{D_1 = \{d_1, d_2, d_3, d_4\}, D_2 = \{d_5, d_6\}\}$. Note that $\nu(C_1) = 0.3$ and $\nu(C_2) = 0.3$; $\boldsymbol{C}$ is cohesive and $\boldsymbol{C}$ is expansion-free as $\nu(\mathsf{F}) = 0.3 = \nu(C_1)$. Note also that $\nu(D_1) = 0.6$ and $\nu(D_2) = 0.1$; $\boldsymbol{D}$ is cohesive and $\boldsymbol{D}$ is expansion-free as $\nu(D_1) > \nu(\mathsf{F})$ □

4 Solving Decomposition Games

Based on our game model, the operation $\mathsf{Decompose}(\mathcal{A})$ in Procedure 1 is reduced to the following DG problem:

INPUT: An attribute primitive $\mathcal{A} = (\mathsf{F}, \mathsf{S}, \mathsf{C}, \prec, \approx, \hookrightarrow)$

OUTPUT: A solution $\boldsymbol{D} = (D_1, D_2, \ldots, D_k)$ of the game $G_{\mathcal{A}}$

Here, we measure computational complexity with respect to the number of requirements in $\mathsf{F} \cup \mathsf{S}$. The proof of Theorem 10 already implies an algorithm for solving the DG problem: check all subsets of R to identify a minimal set with maximal utility; remove it from R and repeat. However, it is clear that this algorithm takes exponential time. We will demonstrate below that a polynomial-time algorithm for this problem is, unfortunately, unlikely to exist.

We consider the decision problem $\mathsf{DG\_D}$: *Given $\mathcal{A}$ and a number $w > 0$, is there a solution $\boldsymbol{D}$ of $G_{\mathcal{A}}$ in which the highest utility of a coalition reaches w?* Recall that the payoff function ν of $G_{\mathcal{A}}$ is defined assuming constants $\alpha, \beta, \gamma > 0$ and $\lambda < 0$. The theorem below holds assuming $\lambda < -\gamma$.

Theorem 12. *The $\mathsf{DG\_D}$ problem is NP-hard.*

Proof. The proof is via a reduction from the maximal clique problem, which is a well-known NP-hard problem. Given an undirected graph $H = (V, E)$, we construct an attribute primitive $\mathcal{A}$ such that any cohesive coalition in $G_{\mathcal{A}}$ reveals a clique in H. Suppose $V = \{1, 2, \ldots, n\}$. The requirements of $\mathcal{A}$ consist of n^2 scenarios: $\mathsf{R} = \mathsf{S} \coloneqq \{a_{i,i'} \mid 1 \leq i \leq n, 1 \leq i' \leq n\}$. In particular, all requirements are non-functional. We define an edge relation E' on S such that

1. $(i, j) \in E$ iff $(a_{i,i'}, a_{j,j'}) \in E'$ for some $1 \leq i' \leq n$ and $1 \leq j' \leq n$

2. If $(a_{i,i'}, a_{j,j'}) \in E'$ then $(a_{i,i''}, a_{j,j''}) \notin E'$ for any $(i'', j'') \neq (i', j')$.
3. Any $a_{i,i'}$ is attached to at most one edge in E'.

Note that such a relation E' exists as any node $i \in V$ is only connected with at most $n-1$ other nodes in H. Intuitively, a set of requirements $A_i = \{a_{i,1}, \ldots, a_{i,n}\}$ serves as a "meta-node" and corresponds to the node i in H. In constructing $\mathcal{A}$, we may define the general scenarios in such a way that

- $T(g(a_{i,j_1}), g(a_{i,j_2})) = 0$ for any $1 \leq i \leq n$ and $j_1 \neq j_2$.
- $T(g(a_{i_1,j_1}), g(a_{i_2,j_2})) = -1$ for any $(i_1, i_2) \notin E$.
- $T(g(a_{i_1,j_1}), g(a_{i_2,j_2})) = 1$ for any $(a_{i_1,j_1}, a_{i_2,j_2}) \in E'$
- $T(g(a_{i_1,j_1}), g(a_{i_2,j_2})) = 0$ for any $(i_1, i_2) \in E$ but $(a_{i_1,j_1}, a_{i_2,j_2}) \notin E'$

For every $1 \leq i \leq n$ and $1 \leq j < j' \leq n$, put in a constraint $c_i(j, j') = \{a_{i,j}, a_{i,j'}\}$. Thus the relevance between $a_{i,j}$ and $a_{i,j'}$ is

$$\sigma(a_{i,j}, a_{i,j'}) = \frac{|c(a_{i,j}) \cap c(a_{i,j'})|}{|c(a_{i,j}) \cup c(a_{i,j'})|} = \frac{\gamma}{2(n-1)}$$

Furthermore if $i \neq i'$, then for any j, j' we set $\sigma(a_{i,j}, a_{i,j'}) = \lambda$. Suppose $U = \{i_1, \ldots, i_\ell\}$ induces a complete subgraph of H. We define the *meta-clique coalition* of U as

$$D_U = \bigcup_{1 \leq j \leq \ell} A_{i_j}$$

By the above definition, for any $1 \leq s < t \leq \ell$, take j, j' such that $(a_{i_s,j}, a_{i_t,j'}) \in E'$.

$$\begin{aligned} w(i_s, i_t, D_U) &= \varepsilon(a_{i_s,j}, D_U) + \varepsilon(a_{i_t,j'}, D_U) \\ &= \rho(a_{i_s,j}, D_U) + \rho(a_{i_t,j'}, D_U) \\ &= (n-1) \times \frac{\gamma}{2(n-1)} + (n-1) \times \frac{\gamma}{2(n-1)} = \gamma \end{aligned}$$

Thus $\nu(D_U) = \frac{n(n-1)\gamma}{2}$. Taking out any element from D_U results in a strict decrease in utility, and hence D_U is cohesive.

Now take any coalition $D \subseteq \mathsf{R}$ that contains two requirements $a_{i,i'}$, $a_{j,j'}$ such that $(i, j) \notin E$. Let $s = |A_i \cap D|$ and $t = |A_j \cap D|$. Note also that $\sigma(a_{j,j'}, a_{i,i''}) = \lambda$ for any $a_{i,i''} \in A_i \cap D$. Therefore we have

$$\nu(D) - \nu(D \setminus \{a_{j,j'}\}) \leq \gamma + 2w(a_{j,j'}, a_{i,i'}, D) \times s \leq \gamma + 2\lambda + \gamma = 2(\lambda + \gamma) < 0$$

The last inequality above is by assumption that $\lambda < -\gamma$. Thus D is not cohesive.

By the above argument, a coalition $D \subseteq \mathsf{R}$ is cohesive in $G_{\mathcal{A}}$ iff D is the meta-clique coalition D_U for some clique U in H. Furthermore, a decomposition $\boldsymbol{D} = (D_1, D_2, \ldots, D_k)$ is a solution in $G_{\mathcal{A}}$ iff V can be partitioned into sets $U_1, \ldots, U_k$ where each U_i is a clique, and $D_i = D_{U_i}$ for all $1 \leq i \leq k$. In particular, H has a clique with ℓ nodes if and only if $G_{\mathcal{A}}$ has a solution that contains a coalition whose utility reaches $\frac{\ell(\ell-1)\gamma}{2}$. This finishes the reduction. $\square$

Theorem 12 shows that, in a sense, identifying a "best" solutions in a DG $G_{\mathcal{A}}$ is hard. The main difficulty comes from the fact that one would examine all subsets of players to find an optimal cohesive coalition. This calls for a relaxed notion of a solution that is computationally feasible. To this end we introduce the notion of *k-cohesive coalitions*. Fix $k \in \mathbb{N}$ and enforce this rule: Binding can only take place on k or less players. That is, a coalition C is *k-cohesive* whenever $\nu(C)$ is greater than the utility of any subsets with at most k players.

Definition 13. *Fix $k \in \mathbb{N}$. In a DG $G_{\mathcal{A}} = (\mathsf{F} \cup \mathsf{S}, \nu)$, we say a coalition $D \subset \mathsf{F} \cup \mathsf{S}$ is* k-cohesive *if $\nu(D') < \nu(D)$ for all $D' \subset D$ with $|D'| \leq k$. An decomposition $\boldsymbol{D}$ is* k-cohesive *if every coalition in $\boldsymbol{D}$ is k-cohesive; if $\boldsymbol{D}$ is also expansion-free, then it is a* k-cohesive solution *of the game $G_{\mathcal{A}}$.*

Remark. In a sense, the value k in the above definition indicates a level of *expected cohesion* in the decomposition process. A higher value of k implies less restricted binding within any coalition, which results in higher "sensitivity" of design elements to conflicts. In a software tool which performs ADD based on DG, the level k may be used as an additional parameter.

Procedure 2 $\mathsf{DGame}(\mathcal{A}, k)$

INPUT: Attribute primitive $\mathcal{A}$, $k > 0$
OUTPUT: Attribute Decomposition $\boldsymbol{D}$

$\boldsymbol{D} \leftarrow \mathsf{Cohesives}(\mathcal{A}, k)$
$\mathsf{Combine} \leftarrow \mathsf{true}$
while $\mathsf{Combine}$ **do**
  $\mathsf{Combine} \leftarrow \mathsf{false}$
  for $(D, D') \in \boldsymbol{D}^2$, $D \neq D'$ **do**
    if $\nu(D' \cup D) > \nu(D)$ and $\nu(D' \cup D) > \nu(D')$ **then**
      $D \leftarrow D' \cup D$ and remove D' from $\boldsymbol{D}$
      $\mathsf{Combine} \leftarrow \mathsf{true}$
return $\boldsymbol{D}$

Procedure 3 $\mathsf{Cohesive}(\mathcal{A}, k)$

INPUT: Attribute primitive $\mathcal{A}$, $k > 0$
OUTPUT: Attribute Decomposition $\boldsymbol{D}$

$\boldsymbol{D} \leftarrow [\,]$, $R \leftarrow \mathsf{F} \cup \mathsf{S}$
while $|R| > 0$ **do**
  $S \leftarrow \mathsf{max}(R, k)$ // compute a maximally k-cohesive coalition
  $R \leftarrow R \setminus S$
  $\boldsymbol{D} \leftarrow [\boldsymbol{D}, S]$
return $\boldsymbol{D}$

Let R be a set of requirements. A coalition D is called *maximally k-cohesive* in R if $|D| \leq k$, D is k-cohesive and $\nu(D) \geq \nu(D')$ for any $D' \subseteq R$. Suppose the operation $\mathsf{max}(R, k)$ computes a maximally k-cohesive set in R. The algorithm $\mathsf{DGame}(\mathcal{A}, k)$ (Proc. 2), which uses $\mathsf{Cohesive}(\mathcal{A}, k)$ (Proc. 3) as a subroutine, computes a k-cohesive solution of $G_{\mathcal{A}}$. Note that the $\mathsf{Cohesive}(\mathcal{A}, k)$ operation maintains a list $\boldsymbol{D}$, which when returned, denotes a decomposition. Note also that the returned $\boldsymbol{D} = (D_1, \ldots, D_m)$ satisfies the following condition:

$$\forall 1 \leq i \leq m : D_i \text{ is maximally } k\text{-cohesive in } D_i \cup \cdots \cup D_m$$

We call this $\boldsymbol{D}$ a *maximally k-cohesive decomposition.*

Lemma 14. *Suppose $\boldsymbol{D}$ is a maximally k-cohesive decomposition. Take any $1 \leq i < j \leq n$. If $\nu(D_i \cup D_j) > \max\{\nu(D_i), \nu(D_j)\}$ then $D_i \cup D_j$ is k-cohesive.*

Proof. Let $S_i = \bigcup_{i \leq j \leq m} D_j$ for any $i = 1, \ldots, m$. Suppose $\nu(D_i \cup D_j) > \max\{\nu(D_i), \nu(D_j)\}$ for $1 \leq i < j \leq m$. By assumption D_i is maximally k-cohesive in S_i. For any finite set $U \subseteq D_i \cup D_j \subseteq S_i$ such that $|U| \leq k$, we have $\nu(U) \leq \nu(D_i) < \nu(D_i \cup D_j)$. Hence $D_i \cup D_j$ is also k-cohesive. □

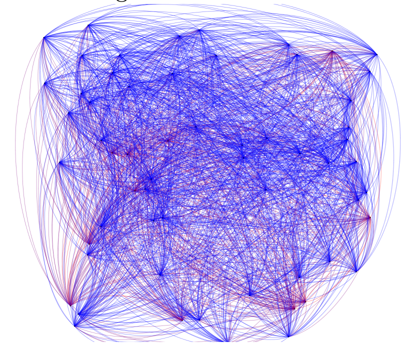
Theorem 15. *Given an attribute primitive $\mathcal{A}$, the $\mathsf{DGame}(\mathcal{A}, k)$ algorithm computes a k-cohesive solution of the decomposition game $G_{\mathcal{A}}$ in time $O(n^k)$, where n is the number of requirements in $\mathcal{A}$.*

Proof. The $\mathsf{DGame}(\mathcal{A}, k)$ algorithm calls $\mathsf{Cohesive}(\mathcal{A}, k)$ to produce a maximally k-cohesive decomposition $\boldsymbol{D}$, and then performs several iterations to "combine" the coalitions in $\boldsymbol{D}$. By Lemma 14, the decomposition $\boldsymbol{D}$ after each iteration is k-cohesive. There is a point when for all $D, D' \in \boldsymbol{D}$ we have $\nu(D \cup D') \leq \max\{\nu(D), \nu(D')\}$. At this moment, the **while**-loop will terminate and $\boldsymbol{D}$ is expansion-free. The time complexity is justified as there are $O(n^k)$ subsets of $\mathsf{F} \cup \mathsf{S}$ with size $\leq k$. Thus computing a maximally k-cohesive decomposition takes time $O(n^k)$. □

5 Case Study: Cafeteria Ordering System

To demonstrate applicability of our game model in real-world, we build a DG for a cafeteria ordering system (COS). A COS permits employees of a company to order meals from the company cafeteria online and is a module of a larger cafeteria management system. The requirements of the project have been produced through a systematic requirement engineering process and is well-documented (See full details from [12, Appendix C]). Since COS is a subsystem within a larger system, the requirements also incorporate interfaces with other subsystems of the overall system. The initial attribute primitive has 60 requirements with $|\mathsf{S}| = 11$, $|\mathsf{F}| = 49$ and 7 design constraints. Non-functional requirements conflict with each other, e.g., the general scenario USE conflicts with the general scenario PER. Also the requirements exhibit some complex relationships, e.g. $\mathsf{SEC1} \hookrightarrow \mathsf{Order.Pay.Deduct}$.

Fig. 3. Interactions between requirements in the COS [12]. Blue edges indicate positive interactions and red edges indicate negative interactions.



We demonstrate the complicated interactions among requirements using a complete graph where nodes are all requirements in $\mathsf{R} = \mathsf{S} \cup \mathsf{F}$; see Fig. 3.

The edges are in two colours: (r_1, r_2) gets blue if $w(r_1, r_2, \mathsf{R}) > 0$ and red if $w(r_1, r_2, \mathsf{R}) < 0$. *(For completeness, we include descriptions of constraints and requirements in the APPENDIX.)*

We run the $\mathsf{DGame}(\mathcal{A}, k)$ algorithm to identify a k-cohesive solution for different levels k of expected cohesion. In order to clearly identify sub-components, we give a higher penalty λ between conflicting requirements: $\alpha = 0.4$, $\beta = 0.3$, $\gamma = 0.3$, $\lambda = -1.3$. We choose $k \in \{1, \ldots, 7\}$. As argued above, setting a higher value of k should in principle improve the quality of the output decomposition, although this also means a longer computation time. We implement our algorithm using Java on a laptop with Intel Core i7-3630QM CPU 2.4GHz 8.0GB RAM. The running time for different values of k is: 503 milliseconds for $k = 3$ and approximately 1140 seconds for $k = 6$.

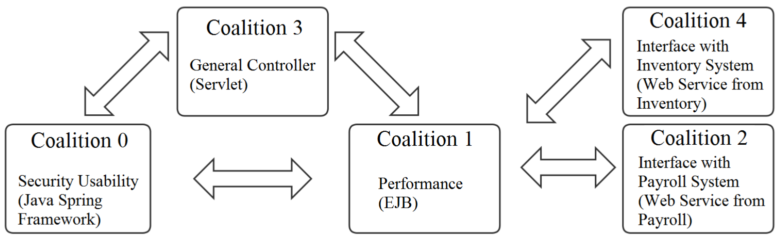
Table 1. Resulting 3- and 6-cohesive solutions, ordered by payoff values.

	3-Cohesive Solution	6-Cohesive Solution
Coalition 0	AVL1 ROB1 SAF1 SEC(1,2,4) USE(1,2) Order.Confirm Order.Menu.Data Order.Deliver.(Select,Location) Order.Pay Order.Place Order.Retrieve Order.Units.Multiple UI2 UI3	AVL1 ROB1 SAF1 SEC(1,2,4) PER(1,2,3) USE(1,2) Order.Confirm Order.Deliver Order.Deliver.(Select,Location) Order.Menu.Date Order.Pay Order.Retrieve Order.Place Order.Units Order.Units.Multiple UI2 UI3
Coalition 1	PER(1,2,3) Order.Units.TooMany Order.Deliver.(Times,Notimes) Order.Place.(Cutoff,Data,Register,No) Order.Pay.(OK,NG) Order.Done.Failure Order.Confirm.(Prompt,Response,More)	Order.pay.(Deliver,Pickup,Deduct) Order.Done.Patron SI2.2 SI2.3
Coalition 2	Order.pay.(Deliver,Pickup,Deduct) Order.Done.Patron SI2.2 SI2.3	Order.Units.TooMany Order.Deliver.(Times,Notimes) Order.Place.(Cutoff,Data,Register,No) Order.Pay.(OK,NG) Order.Done.Failure Order.Confirm.(Prompt,Response,More)
Coalition 3	Order.Menu Order.Unit Order.Done Order.Done.(Menu,Times,Cafeteria) Order.Done.(Store,Inventory) Order.Deliver Order.Menu.Available Order.Confirm.Display Order.Pay.Method SI1.3 SI2.5 CI2	Order.Done Order.Done.(Menu,Times,Cafeteria) Order.Done.(Store,Inventory) SI1.3 SI2.1 SI2.4 SI2.5 CI1 CI2
Coalition 4	SI1.1 SI1.2	Order.Menu.Available SI1.1 SI1.2

Cohesion level $k = 3$. The 3-cohesive solution consists of 5 coalitions. An examination at the requirements in each coalition reveals: *Coalition 0* relates to usability and ensures availability of user interactions; it apparently corresponds to a user interface module. *Coalition 1* is performance-oriented and is separated from the usability requirements; it thus corresponds to a back-end module that handles all the internal operations. *Coalition 2* deals with the payroll system outside COS and defines a controlling interface from COS to payroll. *Coalition 3* consists of several functional requirements that control life cycle of the COS. *Coalition 4* is an interface to access the inventory system outside COS.

It is clear that this solution separates the control, user inputs and computation modules, and fits the MVC (Model-View-Controller) architectural pattern. In addition, there is a design constraint that requires the use of Java and Oracle database engine. So, we instantiate the design elements as in Fig. 4.

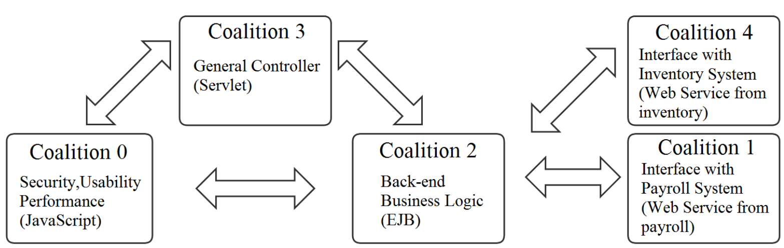
Fig. 4. The 3-cohesive solution. *Coalition 0*: Java Spring framework uses server page as user interface and provides a powerful encryption infrastructure (Spring Crypto Module). Server page is suitable for implementing interactive user interface. *Coalition 1*: Enterprise Java Bean (EJB) is a middleware (residing in the application server) used to communicate between different components. It provides rich features for processing HTTP requests. *Coalition 2*: The COS uses a package solution from corresponding payroll system. *Coalition 3*: A servlet is a controller in Java application server which separates business logic from control. *Coalition 4*: A web service interface outside COS.



Cohesion level $k = 6$**.** The 6-cohesive solution also contains five coalitions, with a similar structure as the 3-cohesive counterpart. There are, nevertheless, several important differences: Firstly, the performance (PER) scenarios now belong to coalitions 0. This means that some performance-related computation is moved to the front-end. This is reasonable as this lightens the computation load of the back-end and thus improving performance and availability. Secondly, the functional requirement Order.Menu.Available is moved to coalition 4, which is the interface between COS and the inventory system. This requirement specifies that the menu should only display those food items that are available in inventory.

Instead of server page, we use scripting to reduce the server's computation load. This can be achieved by changing the front-end to a JavaScript oriented designs. The main difficulty lies in that we need to put extra effort when using JavaScript to communicate with web server (such as AJAX) in order to ensure usability, performance and security. We instantiate design elements as in Fig. 5.

Fig. 5. The 6-cohesive solution. *Coalition 0* uses JavasScript as a front end for user interface. It also takes some computation for sever in order to achieve better performance. *Coalition 1* is an interface for accessing the payroll system. *Coalition 2* ensures the business logic in COS. *Coalition 3* coordinates input from front end (coalition 0) to back end (coalition 1). *Coalition 4* is an interface for accessing the inventory system.



6 Conclusion and Future work

The use of computational games in software architecture design is a novel technique aimed to contribute to this line of direction. We proposed a game-based approach that, not only builds on established software architecture research (ADD), but is also shown – through a case study – to provide reasonable design guidelines to a real world application. We suggest that this framework would be useful in the following:

- Designing a software system that involves a large number of functionalities and quality attributes, which will result in a complicated architecture design
- Designing a software system that hinges on the satisfaction of certain core quality attributes
- Evaluating and analysing the rationale of an architecture design in a formal way; identifying potential risks with a design.

It is noted that the framework described here assumes the completion of requirement analysis. In real life requirements are usually identified as the software is implemented (e.g. the agile software development methodology). It would thus be interesting to develop a dynamic version of the game model, which supports architectural design using incremental refinements. Another future work is to develop a mechanism which maps coalitions generated by the algorithm to appropriate attribute primitives. This would then lead to a full automation of the ADD process linking requirements to conceptual architecture designs.

References

1. Alebrahim, A., Hatebur, D., Heisel, M. (2011): A method to derive software architectures from quality requirements. In Software Engineering Conference (APSEC), 2011 18th Asia Pacific (pp. 322-330). IEEE.
2. Bass, L.(2007): Software architecture in practice. Pearson Education India.
3. Bass, L., Klein, M., Moreno, G. (2001): Applicability of general scenarios to the architecture tradeoff analysis method (No. CMU/SEI-2001-TR-014). Carnegie-Melon Univ., Soft. Eng. Inst.
4. Bass, L., Klein, M., and Bachmann, F.(2002):Quality attribute design primitives and the attribute driven design method. Springer.
5. Branzei, R., Dimitrov, D., Tijs, S. (2008): Models in cooperative game theory (Vol. 556). Springer Science & Business Media.
6. Garlan, D. (2003): Formal modeling and analysis of software architecture: Components, connectors, and events. In Formal Methods for Software Architectures (pp. 1–24). Springer.
7. Kazman, R., Klein, M., Barbacci, M., Longstaff, T., Lipson, H., Carriere, J. (1998, August): The architecture tradeoff analysis method. In Engineering of Complex Computer Systems, 1998. ICECCS'98. Proceedings. Fourth IEEE International Conference on (pp. 68-78). IEEE.
8. Lung, C. H., Xu, X., Zaman, M. (2007): Software architecture decomposition using attributes. International Journal of Software Engineering and Knowledge Engineering, 17(05), 599–613.
9. Papadimitriou, C. (2001, July): Algorithms, games, and the internet. In Proceedings of STOC 2001 (pp. 749-753). ACM.

10. Roughgarden, T., Tardos, E., Vazirani, V. V. (2007): Algorithmic game theory (Vol. 1). Cambridge: Cambridge University Press.
11. Shoham, Y., Leyton-Brown, K. (2008): Multiagent systems: Algorithmic, game-theoretic, and logical foundations. Cambridge University Press.
12. Wiegers, K., Beatty, J. (2013): Software requirements. Pearson Education.
13. Wojcik, R., Bachmann, F., Bass, L., Clements, P., Merson, P., Nord, R., Wood, B. (2006): Attribute-Driven Design (ADD), Version 2.0 (No. CMU/SEI-2006-TR-023). Carnegie-Melon Univ., Soft. Eng. Inst.
14. Woods, E., Hilliard, R. (2005): Architecture Description Languages in Practice Session Report. 5th Working IEEE/IFIP Conference on Software Architecture (WICSA'05). p. 243.

APPENDIX: Constraints and Requirements for COS Case Study

Design Constraints

- CO-2: The system shall use the current corporate standard Oracle database engine.
- CO-3: All HTML code shall conform to the HTML 5.0 standard.
- BR-2: Deliveries must be completed between 11:00am and 2:00pm local time.
- BR-3: All meals in a single order must be delivered to the same location.
- BR-8: Meals must be ordered within 14 calendar days of the meal date.
- BR-11: If an order is to be delivered, the patron must pay by payroll deduction.
- BR-33: 256-bit encryption or network transmissions that involve financial information

Functional Requirements

- *Order.Place*: Placing a meal order
  - *.Register*: Confirm that the Patron is registered for payroll reduction.
    - *.No*: If the patron is not registered for payroll deduction, the COS shall give the Patron options to register now and continue placing an order
  - *.Date*: The COS shall prompt the Patron for the meal date (See BR-8)
    - *.Cutoff*: If the meal date is today and is after the cutoff time, inform the Patron that it's too late The Patron can either change the meal date or cancel the order.
- *Order.Deliver*: Delivery or pickup
  - *.Select*: The Patron specifies whether the order is to be picked up or delivered
  - *.Location*: If the order is to be delivered and there are still available delivery times for the meal date, the Patron shall provide a valid delivery location.
  - *.Notimes*: Notify the Patron if there are no available delivery times. The Patron shall either cancel or pick up the order in the cafeteria.
  - *.Times*: Display the remaining available delivery times for the meal date, allowing the Patron to request one of the times shown
- *Order.Menu*: Viewing a menu
  - *.Date*: Display a menu for the date that the Patron specified.
  - *.Available*: The menu for the specified date shall display only those food items for which at least one unit is available in the cafeteria's inventory and which can be delivered
- *Order.Units*: Ordering multiple meals and multiple food items
  - *.Multiple*: Permit the user to order multiple identical meals
  - *.TooMany*: If the Patron orders more units of a menu item than are presently in the cafeteria's inventory, inform the maximum number of units that can order.
- *Order.Confirm*: Confirming an order
  - *.Display*: When the Patron indicates no wish to order any more food items, display the ordered itmes, prices, and the payment amount
  - *.Prompt*: Prompt the Patron to confirm the meal order.
  - *.Response*: The Patron can confirm, edit, or cancel the order.

.More: Let the Patron order additional meals for the same or for a different date.

– *Order.Pay*: Meal order payment

.Method: When the Patron indicates that he is done placing orders, the COS shall ask the user to select a payment method

.Deliver: Se BR-11

.Pickup: If the meal is to be picked up in the cafeteria, the Patron shall choose to pay by payroll deduction or by cash at the time of pickup

.Deduct: If Patron selects payroll deduction, issue a payment request to Payroll

.OK: If the payment request is accepted, display confirmation a message.

.NG: If the payment request is rejected, display the reason for the rejections.

– *Order.Done*: Finishing the process after the Patron confirms the order

.Store: Assign the next available meal order number to the meal and store the meal order

.Inventory: Send a message to the inventory system with the number of units

.Menu: Update the menu for the current order's order date to reflect any items that are now out of stock in the cafeteria inventory

.Times: Updates the remaining available delivery times for the date of this order

.Patron: Send email message to the Patron with the meal order and payment info

.Cafeteria: Send an email message to the Cafeteria Staff with the meal order information

.Failure: If any step of Order.Done fails, roll back the transaction and notify the user

– **User Interfaces**
  - *UI2*: Provide a help link from each displayed webpage to explain how to use that page.
  - *UI3*: The webpages shall permit complete navigation and food item selection

– **Software Interfaces**
  - *SI1.1*: Transmit the quantities of food items ordered to the Cafeteria Inventory System through a programmatic interface.
  - *SI1.2*: Poll the Inventory System to determine whether a requested item is available.
  - *SI1.3*: When the Cafeteria Inventory System notifies the COS that a specific food item is not available, the COS shall remove that food item from the menu for the current date.
  - *CI1*: Send an email or text message to the Patron to confirm order acceptance
  - *CI2*: Send an email or text message to the Patron to report any problems

Non-functional Requirement We have 6 general scenarios: *USE, PER, SEC, SAF, AVL, ROB*. Each of them associates multiple scenarios.

– *USE1*: Allow a Patron to retrieve the previous meal ordered with a single interaction.
– *USE2*: 95% of new users shall be able to order a meal without errors on their first try.
– *PER1*: Accommodate 400 users and up to 100 concurrent users during the peak usage time, with an estimated average session duration of 8 minutes.
– *PER2*: 95% of webpages generated shall download completely within 4 seconds from the time the user requests the page over a 20 Mbps or faster Internet connection.
– *PER3*: Display confirmation messages to users within an average of 3 seconds and a maximum of 6 seconds after the user submits information to the system.
– *SEC1*: All network transactions that involve financial information or personally identifiable info shall be encrypted per BR-33.
– *SEC2*: Users shall be required to log on for all operations except viewing a menu.
– *SEC4*: The system shall permit Patrons to view only orders that they placed.
– *SAF1*: The user shall be able to see all ingredients in any items, with allergic reactions.
– *AVL1*: The COS shall be available at least 98% of the time between 5am and midnight and at least 90% of the time, excluding scheduled maintenance windows.
– *ROB1*: If the connection between the user and the COS is broken prior to a new order being either confirmed or terminated, the COS shall enable the user to recover an incomplete order and continue working on it.