Towards Automatic Problem Decomposition: An Ontology-based Approach*

Zhi Jin*
Academy of Mathematics and System Science
Beijing, 100080, China
zhijin@amss.ac.cn

Lin Liu
School of Software, Tsinghua University
Beijing, 100080, China
linliu@tsinghua.edu.cn

ABSTRACT

In this paper, we propose a conceptual description schema based on the Problem Frames (PF) approach, which treats environment as a first-class concept. Requirements are defined as problem descriptions in terms of the environment model. Thus, knowledge about the environment can be used to facilitate the derivation of software specifications from requirements. Heuristic rules that help structuring software problems are given. Major idea of the proposed approach is illustrated with a simple real world example.

Categories and Subject Descriptors

D.2.1 [**Requirements/Specifications**]: Elicitation methods, Languages, Methodologies, Tools

General Terms

Design, Languages, Theory

Keywords

Problem Frames, Requirements Engineering, Problem Decomposition, Environment Ontology

1. INTRODUCTION

In the field of requirements engineering, the idea of taking the environment into account can be traced back to Parnas et al's Four-Variable Model [1], which uses four sets of environment related variables to describe system behaviors. Jackson and Zave's well-known work proposed a more general framework [2, 3, 4], which explicitly distinguishes three sets of statements: $\mathcal{K}$ (the domain knowledge), $\mathcal{R}$ (the requirements) and $\mathcal{S}$ (the specification). Furthermore, in [5], the authors discussed in detail this relationship and formalized this relationship as $\mathcal{K}, \mathcal{S} \vdash \mathcal{R}$, to claim that a proof can be given that an implementation of $\mathcal{S}$, introduced into a problem domain satisfying $\mathcal{K}$, will guarantee satisfaction of the requirements $\mathcal{R}$.

PF approach [6] gives a systematical structure to the environment ($\mathcal{E}$) which the problems are situated in and describe what is there and what effects ($\mathcal{R}$) the desired software ($\mathcal{S}$) located there have to achieve. Several work [7, 8] have been done for extending some aspects of this approach. PF approach allows people with knowledge in specific problem domain to drive the RE process and to derive specifications from original requirements statements.

However, at present, this derivation is an empirical, tedious, and subjective process, which heavily depends on the analysts' experiences. The main arguments in PF approach is that there exist some basic problem frames and a real world problem can be decomposed to be finally fitted to them. But we could not assume that any original real world problem description can happen to be fitted to a particular basic problem frame or a combination of basic problem frames. For many real world problems, in order to let them be fitted to these basic problem frames, some intermediate or designed or virtual domains have to be introduced in. That might be a difficult task for those inexperienced analysts.

This paper argues that explicitly modelling and representing the environments can help the automated execution of PF approach. Based on these knowledge, some heuristics could be defined for conducting the automated problem decomposition. This paper is organized as follows. Section 2 presents a small ontology for modelling the environments. Section 3 shows how to represent the software problem based on the ontology. Section 4 discusses the principle for problem decomposition. Section 5 gives several heuristics for problem refinement and decomposition. Section 6 gives a meaningful example for illustrating the approach proposed in this paper. Finally, Section 7 briefly comments some related work and indicates our future work.

2. ENVIRONMENT ONTOLOGY

The main constituents of our environment ontology are the categories of environment concepts, the associations between environment concepts, the types of the interactions between software and its environment entities and the properties of environment entities.

*Partly supported by the Natural Science Foundation of China (No.60233010, No.60496324 and No.60503030), the National Key Research and Development Program of China (Grant No. 2002CB312004), the Knowledge Innovation Program of the Chinese Academy of Sciences, MADIS of the Chinese Academy of Sciences.

*Dr. Jin insisted her name be first.

IWAAPF'06, May 23, 2006, Shanghai, China.

2.1 Top-Level Concepts

After examining many software applications in different application domains as well as summarizing the results in many books on system analysis and modelling, e.g. [2, 6, 9], we extract a set of concept categories. Table 1 gives the concept hierarchy and summarizes the meanings of these concepts.

Table 1: Categories of Environment Concepts

Environment Entity	*entities which a system will interact with*
Atomic Entity	*an individual entity*
Causal Entity	*an individual entity whose properties include predictable causal relationships among its shared phenomena*
Autonomous Entity	*an individual entity which usually consists of people*
Symbolic Entity	*an individual entity which is a physical representation of data*
Aggregate Entity	*a collective entity*
Event	*individual taking place at some particular point of time*
Physical Event	*event initiated by an environment entity*
State Change Event	*event due to a state change of a causal entity*
Signal Event	*event due to a command etc.*
Call/Return Event	*event due to a value query.*
Time Event	*a time point, a time period, etc.*
State Machine	*a direct graph for characterizing the dynamic property of a causal entity*
State	*circumstances of an entity at any given time*
Transition	*a state change in a state machine*
Attribute	*a named variable for characterizing a static property of an entity*
Value	*intangible individual entity that exists outside time and space, and is not subject to change*
Interaction	*observable shared phenomenon between a system and one of its environment entity*
Event Interaction	*they interact with an event*
State Interaction	*they interact with a state*
Value Interaction	*they interact with a value*

2.2 Associations

Besides classifying things, an ontology provides the associations between the ontological categories. These associations form a general conceptualization of the related domain. Some of the associations between environment-related concepts are listed in Table 2.

2.3 Categories of Interaction

Interactions are supposed to be observable and their contents might be different. We argue that distinguishing the contents of interactions may bring back the richer semantics of the interactions. By summarizing the known interactions between the software and its environment entities. Eight different notations have been adapted to depict different interaction contents. These notations are listed in Table 3, in which *ee* means an environment entity, ε is an anonymous environment entity, *event* means an event, *state* means a state and *attr.val* means the value of an attribute.

Table 2: Formation of Associations

Association	formation
has_static_prop	AtomicEntity→Attribute↦Value
has_dynamic_prop	CausalEntity→StateMachine
has_part	AggerateEntity→EnvironmentEntity
can_cause	AutonomousEntity→PhysicalEvent
	CausalEntity→PhysicalEvent
	Transition→PhysicalEvent
has_state	StateMachine→State
has_in_event	StateMachine→Event
has_out_event	StateMachine→PhysicalEvent
has_transition	StateMachine→Transition
source_from	Transition→State
sink_to	Transition→State
be_triggered_by	Transition→Event

Table 3: The Meaning of Interactions

notation	meaning
ee $\uparrow$ *event*	**ee** *causes* **event** *to happen*
ee $\downarrow$ *event*	**ee** *accepts* **event** *that caused by others*
$\varepsilon \uparrow$ *event*	ε *causes* **event** *to happen*
ee : *state*	**ee** *is in* **state**
ee $\twoheadrightarrow$ *attr.val*	**ee** *sends out* **attr.val**
ee $\twoheadleftarrow$ *attr.val*	**ee** *receives* **attr.val**
ee $\rightsquigarrow$ *attr.val*	**ee** *edits and sends out* **attr.val**
ee $\curvearrowright$ *attr.val*	**ee** *sends multiply out* **attr.val***s*
ee $\curvearrowleft$ *attr.val*	**ee** *receives multiply* **attr.val***s*

2.4 Properties between Environment Entities

The static part of environment entities have some properties. An entity is *multiple* if it is allowed to have more than one instances, *changeable* if it has changeable attribute value, and *dividable* if it contains sub-structures. There are also some properties referring to multiple entities or phenomena which are listed in Table 4.

Table 4: Meanings of Environment Property

Property	Meaning
1to1Map(Env,Env)	*an one-to-one mapping exists between two environment entities*
paDepend(Env,Env)	*an environment entity partly depends on another*
paDepend(Phe,Phe)	*a phenomenon partly depends on another one*
identical(Env,Env)	*two environment entities are identical*
subEnv(Env,Env)	*the first environment entity is a part of the second one*
model(Env,Env)	*the first environment entity is a model of the second one*
instance(Env,Env)	*the first environment entity is an instance of the second one*
modeOf(Env,Env)	*the first environment entity is an execution mode of the second one*[1]

3. PROBLEM REPRESENTATION

3.1 Grounding the Environment Entities

For modelling the real world problems, one of the most important steps is to ground formal representations onto re-

[1]Different execution modes of an environment entity have to obey some stipulations of consistency and precedence. That is out of the scope of this paper.

ality, which is referred to as 'designation' [3, 4, 6]. However, the designations in many available methods are rather random and sometimes lack of shared understanding. We have shown in our previous work [10] that a shared understanding is extremely important and efficient in requirements engineering. With the environment ontology, the grounding process for making designations can be guided and will be easier and immune from ambiguity. In fact, the concepts in the environment ontology can serve as the classes of those in the reality. The designation process could be: (1)for each real environment concept, choose an appropriate concept category; (2)instantiate this pre-defined concept category by filling its association slots; (3)perform cross reference checking for guaranteeing the term declaration completeness. The term declaration process and the consistency and completeness checking algorithms could be found in [11, 12].

For illustrating this process, we use Chemical Reactor Controller, adapted from [8], as an example. From the problem statements as well as the problem diagram, we can easily build the models of its environment entities. Table 5 lists the environment entities and Table 6 shows the dynamic models of those causal entities.

Table 5: Environment Entities of Controller

Catalyst: CausalEntity
  be_featured_as:{StateReactive}
  has_dynamic_property:{CatalystStateMachine}

Operator: AutonomousEntity
  be_featured_as:{EventActive}
  can_issue:{Open,Close}

Alarm: CausalEntity
  be_featured_as:{StateReactive}
  has_dynamic_property:{AlarmStateMachine}

CoolingSystem: CausalEntity
  be_feature_as:{StateReactive}
  has_static_property:{waterLevel↦ RealInterval}
  has_dynamic_property:{CoolingSystemStateMachine}

GearBox: CausalEntity
  be_featured_as: {EventActive, StoppableStateActive}
  has_static_property: {oilLevel↦ RealInterval}
  has_dynamic_property: {GearBoxStateMachine}

3.2 Extending the Problem Representation

Instead of representing only the interactions between the desired software and its environment entities, we also explicitly represent the relationships between these interactions and group all of the information related to the desired software into four parts. Concretely, a quadruplet $< Env, Int, Sce, Con >$ has been designed for representing the desired software. In which, *Env* is the set of environment entities which the desired software will interact with or depend on. *Int* is the set of observable interactions between the desired software and its environment entities. *Sce* contains the scenario clauses each of which is formed as 'cause side→effect side' and indicate the direct causal order of the 'cause side' interactions and 'effect side' interactions. *Con* is the set of property clauses. Each property clause is an assertion on a property of an environment entity or a relationship among two environment entities. We call such a kind of representation the external manifestation [13] of the desired software.

Table 6: Dynamic Models of Environment Entities

CatalystStateMachine: StateMachine
  has_state:{open,closed}
  has_in_event: {OpenCatalyst,CloseCatalyst}
  has_transition:{
    open⇓CloseCatalyst→closed,
    closed⇓OpenCatalyst→open
    open⇓OpenCatalyst→open
    closed⇓CloseCatalyst→closed}

CoolingSystemStateMachine: StateMachine
  has_state:{rising,falling}
  has_in_event:{IncreaseWater,DecreaseWater}
  has_transition: {
    rising⇓IncreaseWater→rising,
    rising⇓DecreaseWater→falling
    falling⇓DecreaseWater→falling
    falling⇓IncreaseWater→rising}

AlarmStateMachine: StateMachine
  has_state:{ringing,dumb}
  has_in_event:{RingBell}
  has_transition{dumb⇓RingBell→ringing,
    ringing⇓1min→dumb}

GearBoxStateMachine: StateMachine
  has_state: {value(oilLevel)> δ, value(oilLevel)≤ δ}
  has_out_event:{RequestService}
  has_transition:{value(oilLevel)> δ →
    value(oilLevel)≤ δ⇑RequestService}

We have to give more words to scenario clauses. In a scenario clause, each cause side and effect side could be an atomic interaction or a composite interaction via $\vee$ and $\wedge$. An atomic interaction could be an event interaction, a state interaction or a value interaction. It could also be a value expression constructed from value interaction and/or constants (such as 0) to which functions (such as +) and predicates (such as >) are applied. For example,

$$GearBox \twoheadrightarrow oilLevel.value(oilLevel) < \delta$$

Back to the above example, the external manifestation of *Operation Machine* can be figured out as shown in Table 7.

Table 7: External Manifestation of Operation Machine

OperationMachine
  Environments:{
    Catalyst, Operator, CoolingSystem, Alarm, Gearbox}
  Interactions:{
    Operator↑{Open,Close}
    Catalyst↓{OpenCatalyst, CloseCatalyst}
    CoolingSystem↓{IncreaseWater, DecreaseWater}
    Alarm↓{RingBell}
    Catalyst:{open,closed}
    CoolingSystem:{rising,falling}
    Alarm:{ringing,silent}
    GearBox↠{oilLevel.value(oilLevel)}}
  Scenarios:{
    Operator↑open⇒Catalyst:open,
    Operator↑close⇒Catalyst:closed,
    GearBox↠oilLevel.value(oilLevel)< δ ⇒Alarm:ringing}

4. PROBLEM DECOMPOSITION AS ENVIRONMENT PARTITION

It is commonly recognized that the key to managing problem size and complexity is problem decomposition, i.e. breaking down a given large and complex problem into a number of smaller and simpler subproblems. Different requirements modelling approaches have different decomposition strategies. Continuing our discussion on '*Operation Machine*' with its following three scenario clauses:

$$\text{Operator} \uparrow \text{Open} \Rightarrow \text{Catalyst:open} \quad (1)$$

$$\text{Operator} \uparrow \text{Close} \Rightarrow \text{Catalyst:closed} \quad (2)$$

$$\text{GearBox} \twoheadrightarrow \text{oilLevel.value(oilLevel)} < \delta \Rightarrow \text{Alarm:ringing} \quad (3)$$

Obviously, these clauses are not thoroughly independent. For example, clause(1) and clause(2) share two environment entities, *Operator* and *Catalyst*. These two clauses are talking about how *Operator* causes the state changes of *Catalyst*. We say that they are related with each other. On the contrary, clause(3) is not related with any of them. It talks about another thing, i.e. how *GearBox* causes the state change of *Alarm*, concerning neither *Operator* nor *Catalyst*. We say that clause(3) is related with neither clause(1) nor clause(2). That is to say that two scenario clauses are relevant when they share at least one environment entity. With the kind of relevance, an relevancy graph can be constructed, in which each node is a clause and each edge connecting two clauses means that the two clauses are relevant. We call this graph the **clause relevancy graph**.

For an illustration, we can draw the clause relevancy graph of the above example as shown in Figure 1. The clause relevant graph has two sub-graphs. This indicates the need of decomposing software problems. In other words, in terms of the clause relevant graph, two independent sub-problems can be recognized. One responds to *Operator*'s commands by issuing corresponding control signals to *Catalyst*. The other monitors *GearBox* and issues corresponding signal to *Alarm* if the oil level is below a predefined value.

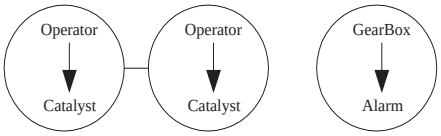

Figure 1: Clause Relevant Graph

This suggests that a possible process for decomposing a software problem is to build the clause relevancy graph, and then each disconnected sub-graph can give the partition of an original software problem. However, the clause relevance originates from the shared environment entities of the scenario clauses. On the other hand, the environment entities involved in same scenario clause are also related. Thus, the partition of the clause relevancy graph becomes the clustering of the environment entities. Environment entities in same cluster are related due to some scenario clauses. Once the set of the environment entities form clusters, the original problem then can be decomposed into several sub-problems. Each sub-problem interacts with one subset of the environment entities and satisfies the scenario corresponding to the environment entities. Based on these observations, we get to the following partition rule.

Partition the set of environment entities: if there exists an equivalent relation among the set of environment entities of a software machine, the machine could be decomposed into several sub-machines, so that these sub-machines don't share any environment entities. In this case, each sub-machine corresponding to one equivalent class satisfies the scenarios on the environment entities in this equivalent class. We call this rule the **environment-based partition rule**.

For an illustration, we apply this rule to above mentioned "Chemical Reactor Controller" problem. Three sub-problems can be identified by clustering the environment entities in terms of the scenario clauses. That derives three sub-problems and the external manifestations of these sub-problem machines can be easily derived as shown in Table 8. They are a commanded behavior problem *Control Catalyst*, a required behavior problem *Regulating Cooling System*[2], and an information display problem *Raise Alarm* according to the Problem Frame approach.

Table 8: Decomposition of 'Control Machine'

Problems	Their External Manifestation
Control Catalyst	ControlCatalyst Environment:{Catalyst, Operator} Interactions:{Operator↑{Open,Close} Catalyst↓ {OpenCatalyst, CloseCatalyst} Catalyst:{open,closed}} Scenarios:{ Operator↑Open⇒Catalyst:open, Operator↑Close⇒Catalyst:closed}}
Cooling System	ReguleteCoolingSystem Environment:{CoolingSystem} Interactions:{ CoolingSystem↓ {IncreaseWater,DecreaseWater} CoolingSystem:{rising,falling}}
Raise Alarm	RaiseAlarm Environment:{Alarm, Gearbox} Interactions:{Alarm:{ringing,silent}, GearBox↠{oilLevel.value(oilLevel)}} Scenarios: {GearBox↠oilLevel.value(oilLevel)< δ ⇒Alarm:ringing}

5. PROJECTION HEURISTICS

However, the decomposition depends on the structure of the scenario clauses. In other words, the decomposability of the software problem depends on the layout of the scenario structure. How to make a scenario structure decomposable becomes another key issue. We argue that introducing some intermediate symbolic environment entities is the strategy for making a software problem becoming decomposable. This strategy is indeed the problem projection. This section will illustrate some heuristics for introducing intermediate environment entities. Before describing

[2]'Cooling System' is not involved in any scenario. That indicates that information on how to regulate the flow of the 'Cooling System' might be missed. The discussion on this issue is considered by the completeness concern which is out of the scope of this paper.

these heuristics, let's name some terms referring to the sub-structures of a scenario clause which are listed in Table 9.

Table 9: Sub-Structures of a Scenario Clause

Terms	Refers To
controller	the environment entity appears in an event interaction in the cause side of a scenario clause
guard	the environment entity appears in a state interaction in the cause side of a scenario clause
valueSender	the environment entity appears in a value interaction in the cause side of a scenario clause
controllee	the environment entity appears in an event interaction in the effect side of a scenario clause
effector	the environment entity appears in a state interaction in the effect side of a scenario clause
valueReceiver	the environment entity appears in a value interaction in the effect side of a scenario clause

The following are some common used projection strategies for making problem more decomposable:

Projection with a shared active environment (non-interacting): There are a set of scenario clauses, which are relevant just due to a shared environment entity *ee* as a controller. If the set of the rest environment entities is partitionable and forms n independent entity clusters, make n copies of *ee* and let each copy instantiate *ee* in only one cluster. That makes the clusters unrelated.

Projection with a shared passive environment: There are a set of scenario clauses, which are relevant just due to a shared environment entity *ee* as a value receiver. If the set of the rest environment entities is partitionable and forms n independent entity clusters, and if *ee* is an aggregative passive entity and each sub-structure is controlled by exact one entity cluster, divide *ee* into n sub-entities corresponding to the sub-structures and let them instantiate *ee* in the corresponding cluster. That makes the clusters unrelated. We call this heuristic rule the **non-interacting value receiver projection rule**.

Projection with a shared non-passive environment: There are a set of scenario clauses, which are relevant just due to a shared environment entity *ee* as a controllee. If the set of the rest environment entities is partitionable and forms n independent entity clusters, make n copies of *ee* and let each copy instantiate *ee* in only one cluster. In this case *ee* has n execution modes. As these execution modes must certainly interact, some consistency and precedence conditions should be further considered at the same time[3].

Projection by introducing a symbolic environment (the producer/consumer pattern): A typical situation which can use this heuristic is when the input values are changeable and editable. For this situation, an environment entity edits an attribute value or several attribute values, which may be used by another environment entity. Just like one producer produces the value(s) that will be consumed by consumer(s). When the value changes, a model environment entity is introduced in for separating the changeability so that the usage of values may not be effected. And then this problem can be split into two subproblems: one builds the model and the other uses it.

Projection by introducing a symbolic environment (the database pattern): A typical situation for this heuristic is that one environment entity produce continuously or periodically many values of the same attribute and one other environment entity will obtain a function value on the set of values. In this case, as we know, we normally use a database to remember these values for later use. There are many situations that a database environment entity may be useful, such as: remembering past events and states of the active environments; summarizing the past; supporting inferences about the private phenomena of the environment entity; anticipating lengthy calculations, etc.

Projection by introducing a symbolic environment (the transformation pattern): This rule can be used when the value sent out by the environment entity in the cause side functionally related with the value received by the environment entity in the effect side. Normally, in this situation, both the cause environment entity and the effect environment entity are symbolic environment entities.

[3]We will leave these concerns our other papers for keeping this paper more concentrated.

6. CASE STUDY

Let's illustrate another small problem: Telemarketing [14].

> A society needs a *Telemarketing System* for selling lottery tickets to its *Supporters*. This society has different kind of *lottery campaigns*. The available campaigns may be updated by the *Campaign Designer*. The telemarketer communicate with the supporters via *telephones*. All of these connections are made by this system. If one supporter is willing to buy a special kind of lottery, the ticket placement should be recorded for producing a ticket order. The ticket orders will be sent to *Order Processor*.

The external manifestation of the 'Telemarketing' system is shown in Table 10. Figure 2 shows the order of scenarios clauses.

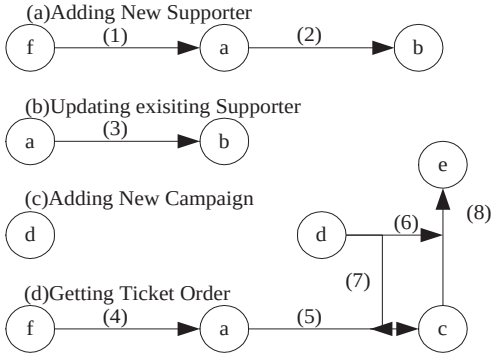

Figure 2: Scenarios of Telemarketing

Four heuristics can be used for enabling the problem decomposition. They are (1) interacting controllee projection rule for duplicating '*Phone*', (2) non-interaction controller projection rule for duplicating '*Telemarketer*', (3) separation rule in database pattern for introducing '*Supporter Database*' and '*Campaign Database*', and (4) separation rule in producer/comsumer pattern for duplicating

Table 10: External Manifestation of Telemarketing

Environments: {Campaign Designer(CD), Supporter(S), Phone(P), Order Processor(OP), Telemarketer(T)}
Interations: {
a.Telemarketing System(TS) connects a P for getting through a S
b.A S inputs or updates his/her 'supporter details' to TS
c.A S inputs 'ticket placement' to TS based on a 'campaign details'
d:CD inputs or updates a 'campaign details' to TS
e:TS sends a 'ticket order' to OP
f:T inputs a phone number}
Scenarios{
Adding new supporter(see Figure 2(a))
Updating existing supporter(see Figure 2(b))
Adding new campaign(see Figure 2(c))
Getting ticket order(see Figure 2(d))}
Constraints: {
multiple(P), multiple(S), changeable(S),
changeable('campaign details'),
multiple('campaign details'),
paDepend('ticket order','campaign details'),
paDepend('ticket order','supporter details')}

'*Supporter*', '*Supporter Database*' and '*Campaign Database*'. The refined problem diagram as shown in Figure 3.

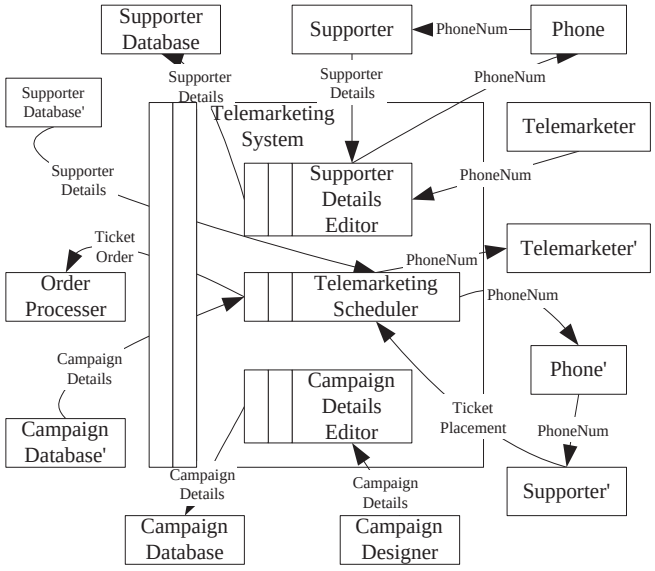

Figure 3: Decomposition of the Telemarketing System

By using the environment-based partition rule to the refined problem diagram, three sub-problems: Supporter Details Editor, Campaign Details Editor and Telemarketing Scheduler can be identified. These three sub-problems are depicted in Table 11, Table 12, and Table 13.

7. CONCLUSIONS

In summary, from the viewpoint of environment-based approach, **partition** is in essence grouping the set of environments into several independent parts and each environment appears in exactly one partition. This independency implies

Table 11: Supporter Details Editor

Supporter Details Editor
Environment:{Supporter, Phone, Telemarketer, SupporterDatabase}
Interactions:{
Telemarketer↠{Supporter.phoneNum}
Phone↓{Dial(phoneNum)}
Supporter↬{Supporter.Details}
SupporterDatabase↞{Supporter.Details}}
Scenarios:{
Supporter↬Supporter.Details⇒ SupporterDatabase↞Supporter.Details,
Telemarketer↠Supporter.PhoneNum⇒ Phone↓Dial(PhoneNum),
Phone↓Dial(PhoneNum)⇒ Supporter↬Supporter.Details}
Constraints:{changeable(Supporter), changeable(SupporterDatabase), multiple(Supporter)}

Table 12: Campaign Details Editor

Campaign Details Editor
Environment:{CompaignDesigner, CampaignDatabase}
Interactions:{
CompaignDesigner↬{CampaignDetails}
CampaignDatabase↞{CampaignDetails}}
Scenarios:{
CompaignDesigner↬Campaign.Details⇒ CampaignDatabase↞Campaign.Details}
Constraints:{changeable(CompaignDesigner), changeable(CampaignDatabase), multiple(CompaignDesigner)}

that no relation of interest exists between any two environments of different parts and no requirements constraint exists for any two environments of different parts. Relating to the problem, partition means that the problem can be divided into several un-interacting sub-problems.

In contrast, **projection**, as another way to logically decompose system viewpoints, is treated differently comparing to conventional modelling approaches [15, 7]. In these approaches, this is due to the absence of explicit modelling of "environment", such that the only way to differentiate the switch of context is to use concepts such as viewpoints, or perspectives. we treat projection by introducing some heuristics to make the orthogonal and coherent problem separated into independent parts so that the **partition** rule can be used to do the further decomposition. In the proposed approach, projection includes following steps, first, we decompose a holistic problem into physically unique subproblems by observing their interaction with the environment entities. Then, we may compose (i.e. cluster) these subproblems to form different composite problems, among which overlapping or ignorance of parts may happen. It also leads to the change of system-to-environment boundary, which implies switches of the perspectives accordingly. For instance, a "read attribute value" event can be viewed differently in different subproblems. To the least, there are perspectives such as the sender entity's view, the receiver entity's view, and a user of the entity's view.

We think one of the most important enhancement to the PF approach by the proposed approach is explicitly modelling the environment entities and distinguishing different

Table 13: Telemarketing Scheduler

```
Telemarketing Scheduler
  Environment:{Telemarketer',Phone',
    Supporter',SupporterDatabase',
    OrderProcessor,CampaignDatabase'}
  Interactions:{Telemarketer↑{start},
    Phone'↓{dialSupporter.PhoneNum},
    Supporter'↬{Supporter.TicketPlacement},
    SupporterDatabase'↠{Supporter.Details},
    CampaignDatabase'↠{Campaign.Details},
    OrderProcessor↞{Supporter.TicketOrder}}
  Scenarios:
    {Telemarketer↑{start}⇒
      SupporterDatabase'↠{Supporter.Details},
    SupporterDatabase'↠{Supporter.Details}⇒
      Phone'↓{dialSupporter.PhoneNum},
    Phone'↓{dialSupporter.PhoneNum}∧
      CampaignDatabase'↠{Campaign.Details}
      ⇒Supporter'↬{Supporter.TicketPlacement},
    Supporter'↬{Supporter.TicketPlacement}⇒
      OrderProcessor↞{Supporter.TicketOrder}}
```

kinds of interactions. With these explicit represented knowledge, the partition rule and the projection rules can be formatted and become operatable, so that the problem decomposition process could be systematically conducted. Some kind of tool can be developed to make the process automatic or at least too-supported.

There is other proposal to model the generic elements. In [16], generic elements of requirements specifications are reviewed, some basic building blocks of which is shared by the environment ontology proposed in this paper, such as, entities, states, transitions, etc. A major difference between the two shouldn't be ignored. That is, despite of these common concepts, the semantics and reasoning behind is different. The proposed environment ontology only concerns the real world entities interacting with the system under design. It assumes the system under design is situated among these entities as a blackbox, who enables the state-transitions and message exchanges. For example, an entity in the Davis requirements model will later become a designed element, within the system. In contrast, entities in these paper will led to the creation of a counterpart in the system interacts with it, but not an implementation of the entity itself. Comparing the two different perspectives yield clearer understanding to the proposed approaches - one is trying to streamline analysis with design and implementation by promoting implementation concepts to capture problem statements. The later aims to spell out the problem without being biased by specific implementation concepts.

Acknowledgment

Thanks are due to Mr. Michael Jackson and the anonymous reviewers for their valuable comments and suggestions.

8. REFERENCES

[1] D.L.Parnas and J.Madey. Functional documentation for computer systems. *Science Computing Program*, 25(1):41–61, Oct. 1995.

[2] M.Jackson. *Software Requirements and Specifications: a Lexicon of Practice, Principles and Prejudices*. Addison-Wesley, 1995.

[3] P.Zave and M.Jackson. Four dark corners of requirements engineering. *ACM Transactions on Software Engineering and Methodology*, 6(1):1–30, January 1997.

[4] M.Jackson. The meaning of requirements. *Annals of Software Engineering*, 3:5–21, 1997.

[5] C.A.Gunter, E.L.Gunter, M.Jackson, and P.Zave. A reference model for requirements and specification. *IEEE Software*, 17(3):37–43, May/June 2000.

[6] M.Jackson. *Problem Frames: Analyzing and Structuring software development problems*. Addison-Wesley, 2001.

[7] Andrés Silva. Requirements, domain and specifications: A viewpoint-based approach to requirements engineering. In *Proceedings of the ICSE'02*, pages 94–104, Orlando, Florida, USA, 2002.

[8] Jon G.Hall, Lucia Rapanotti, and Michael Jackson. Problem frames semantics for softrware development. *Journal of Software and System Modeling*, 40(2):189–198, 2005.

[9] Jeffrey L.Whitten, Lonnie D.Bentley, and Kevin C.Dittman. *System Anaysis ans Design Methods*. McGraw-Hill Companies, fifth edition, 2001.

[10] R.Lu and Z.Jin. *Domain Modelling based Software Engineering*. Kluwer Academic Publishers, 2000.

[11] Z.Jin, D.A.Bell, F.G.Wilkie, and D.G.Leahy. Automated requirements elicitation: Combining a model-driven approach with concept reuse. *International Journal of Software Engineering and Knowledge Engineering*, 13(1):53–82, 2003.

[12] Z.Jin, R.Lu, and D.A.Bell. Automatically multi-paradigm requirements modeling and analyzing: an ontology-based approach. *Science in China (Series F)*, 46(4):279–297, Aug. 2003.

[13] Zhi Jin. Revisiting the meaning of requirements. *Journal of Computer Science and Technology*, 21(1):32–40, 2006.

[14] Leszek A.Maciaszek. *Requirements Analysis and System Design: Developing Information Systems with UML*. Pearson Education Limited, 2001.

[15] A.Finkelstein, J.Kramer, B.Nuseibeh, L.Finkelstein, and M.Goedicke. Viewpoints: a framework for integrating multiple perspectives in system development. *International Journal of Software Engineering and Knowledge Engineering*, 2(1):31–57, 1992.

[16] Alan M.Davis, Kathleen Jordan, and Tsuyoshi Nakajima. Elments underlying the specification of requirements. *Annals of Software Engineering*, 3:63–100, 1997.