I.J. Information Engineering and Electronic Business, 2014, 4, 1-11
Published Online August 2014 in MECS (http://www.mecs-press.org/)
DOI: 10.5815/ijieeb.2014.04.01

A Knowledge-based PSEE with the Ability of Project Monitoring

Shih-Chien Chou
Department of Computer Science and Information Engineering, National Dong Hwa University, Taiwan
scchou@mail.ndhu.edu.tw

Chiao-Wei Li
Department of Computer Science and Information Engineering, National Dong Hwa University, Taiwan
610121027@ems.ndhu.edu.tw

Abstract— Process-centered software engineering environments (PSEEs) facilitate managing software projects. According to the change of enactment environments and the increment of software development complexity, PSEE features should be enhanced. We designed a knowledge-based PSEE named *KPSEE*. It offers the features: (1) maximizing the degree of process parallelism, (2) enhancing process flexibility, (3) managing product consistency, (4) integrating PSEEs, (5) keeping pace with significant process change, (6) preventing technique leakage, and (7) offering project monitoring ability.

Index Terms— PSEE; Knowledge-Based; Product-Driven; Project Monitoring; Maximize the Degree of Process Parallelism

I. Introduction

Process-centered software engineering environments (PSEEs) facilitate managing software projects. PSEE research is affected by "processes are also software" [1]. More than a decade later, the research is still valuable although it is not so hot as before. According to the change of enactment environments (e.g., from single machine to the network) and the increment of software development complexity (e.g., from waterfall to the Agile models [2]), PSEE features should be enhanced.

A PSEE is composed of a process language and an enactment environment. The language implements software processes into *process programs* for the environment to enact. A process program is primary composed of *activities*. An activity is assigned to *role*(*s*) played by *software developers*. When the *condition* of the activity is true, the roles produce *output product*(*s*) (e.g., specification) by referring to *input product*(*s*) using *tool*(*s*). For example, *designers* (i.e., roles) produce a *sub-design document* (i.e., output product) by referring to *sub-specifications* (i.e., input products) using *Rational Rose* (i.e., tool). Attaching a condition to an activity is necessary to implement the selection and repetition constructs. The article in [3] mentioned the important PSEE features: (1) enactment support, (2) software team (organization) distribution, (3) consistency management, (4) process flexibility (i.e., dynamic changing process program during enactment), (5) process evolution, and (6) keeping pace with significant change. Although traditional PSEEs offers one or more features mentioned above, they generally suffer from shortcomings below.

- Their process languages look like programming languages, which may limit the degree of process parallelism. Since the activities of certain processes are difficult to predict (e.g., the Agile models), limiting the degree of parallelism induces trouble when modeling and enacting the processes.
- If multiple organizations cooperate for a software project and they use different PSEEs, coordinators [4-5] are needed. Since the functions of different PSEEs are heterogeneous, the ability of a coordinator may be limited by the PSEEs.
- Traditional PSEEs offer limited functions to handle exceptions. They generally provide functions for process evolution [6-8]. Handling process evolution may stop software project execution, which results in time delay.
- If the cooperating organizations are mutually untrusted, technique leakage may occur because software development technique may be embedded in software products. Transferring a product developed by an organization to an untrusted one may result in technique leakage.
- Traditional PSEEs release the load of project managers by remembering when to do what activities using which tools. However, an executing project should be monitored. Traditional PSEEs did not offer monitoring functions. If a PSEE facilitates project monitoring, the PSEE will be more valuable.

To overcome the shortcomings, we develop a new PSEE. Since the PSEE is knowledge-based, we name it *KPSEE* (knowledge-based PSEE). KPSEE offers the features mentioned in [2] and the following enhanced features:

- KPSEE maximizes the degree of parallelism. The degree of parallelism is maximized if an activity is enacted immediately when the condition of the activity is true and the resources required by the activity (such as input products and roles) are available. KPSEE enacts activities in this manner. Therefore, it maximizes the degree of parallelism.

- KPSEE enhances process flexibility. Process flexibility allows dynamic process program change [2, 9]. KPSEE enhances the flexibility by allowing dynamically adding, removing, and changing all process components at anytime during enactment. Here process components include everything in a process program, such as roles and activities. KPSEE offers this feature by allowing unstructured statements and providing solid exception handling functions. With unstructured statements, KPSEE process statements can be placed without order. This simplifies the addition of process components. As to solid exception handling functions, it supports the changing and removing of process components.
- KPSEE is an *integrator* instead of a *coordinator*, and offers simple interface for the integration. The interface is the KPSEE process language. Process programs in other PSEEs should first be translated into KPSEE process programs. Since KPSEE statements are unstructured, placing the translated process programs together results in an integrated KPSEE process program. KPSEE enacts an integrated program without the intervention of other PSEEs. Therefore, other PSEEs will not limit the function of KPSEE.
- KPSEE process language offers statements to trigger KPSEE for the handling of exceptions and evolutions without stopping the executing projects. Since the characteristic of exception and evolution are similar in this article, we let "exception" to include both "evolution" and "exception" in the rest of the text.
- KPSEE introduces the *information flow control* [10-11] concept to control the access of software products, which prevents technique leakage.
- KPSEE uses rules to facilitate partial project monitoring (it is impossible to facilitate full project monitoring). KPSEE offers default rules for the monitoring. If necessary, rules can be added, removed, or changed by the project managers.

The kernel of KPSEE is a knowledge base. Surrounding the base are functions to enact process programs. It also offers a sub-system to facilitate monitoring software projects. In the rest of this paper, section II discusses related work, section III presents KPSEE in details, section IV proves that KPSEE offers the features we mentioned, and section V is the conclusion.

II. Related Work

Generally, existing PSEEs adopt programming language constructs (e.g., sequences, selections, and repetitions) to develop process languages. Therefore, a process language looks like a programming language. For example, CSPL [12] adopts the Ada95 programming language for its process language. All the features of Ada95 are offered by CSPL. However, most features mentioned in section I are not provided.

In the early days, PSEEs are generally centralized. Centralization limits the distribution of software developers. Under this consideration, decentralized PSEEs are attractive. Oz [13] is decentralized. It is structured by homogeneous and independent sub-environments. Each sub-environment is provided to a development organization. Multiple organizations cooperate for a software project using the "submit protocol". OPERA [14] offers a kernel and an intermediate language. All process programs enacted in OPERA should first be translated into the intermediate language and then enacted by the kernel. Since the kernel can be distributed, the intermediate language can be considered integration interface.

The researches in [6-8] manage process evolution or process change. The general problem for the researches is time-consuming because process programs may be suspended when handling evolution. As to product consistency management, we develop a technique to achieve that [15]. The technique manages the dependencies among software products. When a product is changed, those directly or indirectly dependent on it will be identified and changed accordingly. ADAMS [16] applies the fine-grained concept to manage software artifacts, including all kind of software documents such as specifications and design documents. It is a SCM (software configuration management) model rather than a PSEE. Finer granularity offers the primary features of: (1) evolving one part of a product will not affect other parts and (2) reducing the possibility that more than one developer intends to develop the same artifact. As a SCM model, ADAMS keeps traceability among artifacts. Therefore, it manages software consistency. SPACE [17] is a domain independent environment. It applies meta-models to manage software process as well as artifacts. The use of meta-models allows semantic process/artifact-oriented collaboration. SPACE offers good collaboration among software organizations and keeps product traceability. Therefore, consistency management in SPACE is of no problem.

The researches in [18-19] use process agents [18] or deviation rules [19] to detect and handle the deviation of software processes. In general, software process deviation almost always happens during a software project. Therefore, deviation handling (or process evolution handling) is necessary. However, some deviation, such as that in the Agile models, may be out of control. We are not sure whether the researches in [18-19] can handle the deviation. The model-driven approach [20-22] use meta-models to manage process variability. For example, MoDErNE [20] reuses existing process models and applies rules to customize the reused models. During process modeling, reused process models appear in a process as *modeling tasks* or *editors*. During process enactment, if a modeling task or editor is encountered, the associated rules customize the process model. The approach solves the variability of process. However, if exceptions occur after customization, MoDErNE cannot solve them using meta models. This reduces the power of exception handling in model-driven approach.

Since preventing technique leakage is an important feature of KPSEE, we also survey this type of PSEEs. However, we cannot identify a PSEE that offers the

feature, except our previous research [23]. The research embedded an information flow control model in a PSEE, which is similar to that of KPSEE. The major difference is that an information flow control model is "embedded" in a PSEE in our previous research, but KPSEE and the information flow control model are "fused" together.

III. KPSEE

KPSEE decides whether an activity can be enacted by checking the status of input products. Therefore, it is also *product-driven*. As described in section I, if the input product set *IPds* of an activity *Act* are available and the condition *Cond* of the activity is true, then *Act* is enactable. When the role set *Rles* required to enact *Act* are available, *Act* is enacted immediately. The following rule depicts the kernel concept of KPSEE, in which *avl*(*PD*) is the set of available products, *avl*(*RLE*) is the set of available roles, and *enact*(*Act*) means enacting *Act*.

$$(IPds \subseteq avl(PD)) \wedge (Rles \subseteq avl(RLE)) \wedge (Cond = TRUE) \Rightarrow enact(Act)$$

The rule does not mention tools because we assume that software tools are available for software development organizations.

To prevent technique leakage, the kernel concept should be adjusted. Suppose a software product is developed by role(s) and a role is in an organization. Moreover, an organization trusts zero or more others. With the assumptions, when an activity requires one or more input products, the roles enacting the activity should be in the organizations that can access all the input products. The organizations that can access a product belong to the set "$ORG \cup (\cap_i TORG_i)$", in which *ORG* is the set of organizations that developed the product, $TORG_i$ is the set of organizations trusted by Org_i, and $Org_i \in ORG$. Based on this, if the input product set of an activity is *IPD* = {IPd_i | IPd_i is a product} and the organizations that develop an input product IPd_i is ORG_i = {Org_i | Org_i is an organization}, then the roles that can enact the activity should be in the role set *AURLE* = {Rle_k | Rle_k is a role in an organization of the set $(\cap_i ORG_i) \cup (\cap_j TOrg_{i,j})$ in which $TOrg_{i,j}$ is the set of organizations trusted by the organization Org_i and $Org_i \in (\cap_i ORG_i)$ }. Sometimes, roles in *AURLE* are not enough to enact an activity. To solve this problem, a set of authorized organizations *AO* trusted by every organization should be available. According to the description above, we need the following information for technique leakage prevention:

- An organization is associated with a list containing the organizations it trusts. That is, an organization Org_i should be associated with a $TOrg_{i,j}$.
- A product Pd_i is associated with a list of organizations that produced it. That is, Pd_i should be associated with an organization set *ORG*.
- An authorized list *AO* containing organizations trusted by every organization should be offered.

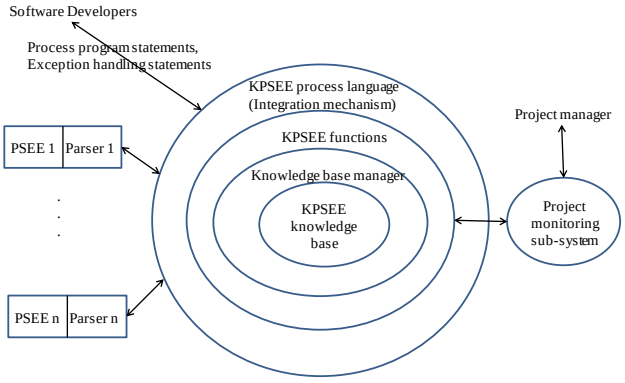

Fig. 1. The architecture of KPSEE

Having described the kernel concept of KPSEE, we describe KPSEE. Fig. 1 shows that KPSEE can be accessed directly by software developers. It can also integrate process programs from different PSEEs (i.e., KPSEE process language acts as an integration mechanism). To integrate PSEEs, every PSEE should offer a parser to translate their process programs into KPSEE process programs. The component "KPSEE functions" in the figure includes a parser, the functions to enact KPSEE process programs and prevent technique leakage, and an exception handler. The component "KPSEE knowledge base" records the status of products and activities, and the relationships among products, roles, activities, organizations, and tools. Information in the knowledge base is managed by the "Knowledge base manager". KPSEE also offers a "Project monitoring sub-system" to facilitate project monitoring. Information needed by the sub-system is offered by the KPSEE functions and the project manager. The following sub-sections describe the KPSEE components.

A. KPSEE Process Language

KPSEE process language does not use traditional constructs such as selections and repetitions. Its statements are unstructured (i.e., without order). As long as the required resources of an activity are available and its condition is true, the activity is enacted immediately. This maximizes the degree of process parallelism. Statements can be added, changed, or removed anytime during process enactment. This enhances process flexibility and allows software processes with unpredictable activities, such as the Agile processes, to be easily implemented and enacted. KPSEE process language offers the following simple statements.

- *+Role*(*Rle*, *PdRle*, *SD*, *IP*, *Org*), *-Role*(*Rle*, *SD*, *Org*), **Role*(SD_1, SD_2). The former two statements respectively add and remove a role, in which *Rle* is a role, *PdRle* indicates the products that can be used by the role, *SD* is a software developer playing the role, *IP* is the IP address assigned to the *SD*, and *Org* is the organization of the role. KPSEE must know the IP addresses to inform the required roles for activity enactment. It should also know the role's organization to prevent technique leakage. The **Role* statement

replaces SD_1 by SD_2. It handles SD departure. A departed SD should be replaced by another to maintain the products produced by the departed one.

Table 1. Relationships between roles, tools, and products

Pd	*PType*	*Rle*	*PdRle*	*Tool*	*PdTl*
Requirement	1	Customer	{1, 2}	Word editor	{1, 2, 3}
Specification	2	Domain expert	{1, 2}	UML tool	{1, 2, 3}
Design document	3	Analyst	{1, 2, 3}	Programming Language	{4}
Source code	4	Designer	{2, 3}	Testing tool	{4, 5, 6, 7}
Test case	5	Programmer	{3, 4}		
Test report	6	Tester	{4, 5, 6}		
Released product	7	Project manager	{1, 2, 3, 4, 5, 6, 7}		

PdRle facilitates monitoring activities. For example, requesting a programmer to use requirements for system analysis is infeasible. In addition to the relationships between roles and products, those between products and tools should also be monitored. For example, using Microsoft Word to implement a program is infeasible. We use Table 1 to show the relationships among roles, tools, and products, in which *PType* is a product type, *Tool* is a tool, and *PdTl* indicates the products that can be operated by a tool. To allow more flexibility, the contents of Table 1 can be changed by a project manager.

- +*Organization*(*Org*, *Rle*, *TOrg*), -*Organization*(*Org*), **Organization*(*Org*, *TOrg*). The former two statements respectively add and remove an organization *Org*. When adding *Org*, the roles in it (i.e., *Rle*) and the organizations trusted by it (i.e., *TOrg*) should be described. The **Organization* statement changes the organizations trusted by *Org*.
- +*AOrg*(*Org*, *Rle*), -*AOrg*(*Org*), **AOrg*(*Org*, *Rle*). The former two statements respectively add and remove an authorized organization. When adding an authorized organization, roles in the organization (i.e., *Rle*) should be presented. The **AOrg* statement changes the roles in an authorized organization.
- +*Product*(*Pd*, *PType*, *Org*), -*Product*(*Pd*), **Product*(*Pd*): The statements respectively add, remove, and change a product. *PType* is the type of the product (see Table 1). *Org* is a set of organizations that produced the product. The +*Product* statement can add initially available products such as user requirements. Adding initially available product is necessary because KPSEE is product-driven. If no available products exist, no activity will be enacted.
- +*Variable*(*Var*, *Val*), -*Variable*(*Var*), **Variable*(*Var*, *Val*): The statements respectively add, remove, and change variables used in a process program, in which *Var* is the variable set and *Val* is the corresponding value set for *Var*. The statements are necessary to implement the selection and repetition constructs.
- +*Tool*(*TlName*, *PdTl*), -*Tool*(*TlName*), **Tool*(*TlName*, *PdTl*). The statements respectively add, remove, and change a tool, in which *TlName* is a tool name, and *PdTl* is shown in Table 1.
- +*Activity*(*ActID*, *IPd*, *OPd*, *Cond*, *Action*, *Rle*, *Tool*, *Schl*, *Budget*, *HouAct*), -*Activity*(*ActID*), **Activity*(*ActID*, *newIPd*, *newOPd*, *newCond*, *newAction*, *newRle*, *newTool*, *newSchl*, *newBudget*, *newHouAct*): The former two statements respectively add and remove an activity. The **Activity* statement changes the contents of an activity. In the statements, *ActID* is the identity of an activity to differentiate activities, *IPd* is the set of input products, *OPd* is the set of output products, *Cond* is the condition to trigger the activity, *Action* is the action of the activity, *Rle* is the set of roles to take the action, *Tool* is the set of tools used in the activity, *Schl* is the schedule of the activity, *Budget* is the budget of the activity, and *HouAct* is the housekeeping actions after the activity finishes (e.g., if an activity is in a loop, the housekeeping actions may contain a loop counter decrement statement). The **Activity* statement adds a word "*new*" before parameter names means the contents of the activity is changed.
- +*ScheduleBudget*(*TolSchl*, *TolBudget*), **ScheduleBudget*(*TolSchl*, *TolBudget*). They respectively indicate and change the schedule and budget of a project.

B. KPSEE Knowledge Base and its Manager

KPSEE knowledge base *KPKB* (KPSEE knowledge base) is defined below, in which "$\rightarrow$" is a "depend on" relationship:

Definition 1. *KPKB* = (*PD*, *ACT*, *ROLE*, *TOOL*, *ORG*, *AO*, *PDDEP*, *PDACT*, *PDROLE*), in which:

- *PD* is the set of software products. It is defined below.

  PD = {(*Name*, *Status*, *PType*, *Org*)$_i$ | *Name*$_i$ and *Status*$_i$ are the name and status of the ith product, respectively. *PType*$_i$ is shown in Table 1. *Org*$_i$ is the organization set that produced the product. Product status may be "A" (available), "U" (unavailable), and "D" (removed).}

- *ACT* is the set of activities, which is defined below.

  ACT = {(*ActID*, *IPd*, *OPd*, *Cond*, *Action*, *Rle*, *Tool*, *Schl*, *Budget*, *HouAct*, *Status*)$_i$ | *ActID*$_i$, *IPd*$_i$, *OPd*$_i$, *Cond*$_i$, *Tool*$_i$, *Shcl*$_i$, *Budget*$_i$, and *Status*$_i$ are respectively the identity, the set of input products, the set of output products, the condition, the schedule, the budget, and the

status of the i[th] activity. $Action_i$ is the action of the activity. Rle_i is the set of roles to take the action. $Tool_i$ is the set of tools used in the activity. IPd_i and OPd_i are subsets of *PD*. Activity status may be "E" (enacting), "W" (wait for enactment), "F" (finish enactment), and "D" (removed).}

- *ORG* is the set of organizations, which is defined below:

ORG = {(Org, Rle, $TOrg$)$_i$ | Org_i is the i[th] organization, Rle_i is the set of roles in Org_i, and $TOrg_i$ is the set of organizations trusted by Org_i}

- *AO* is the set of authorized organizations, which is defined below:

AO = {($AOrg$, Rle)$_i$ | $AOrg_i$ is the name of the i[th] authorized organization and Rle_i is the set of roles in $AOrg_i$}

- *ROLE* is the set of roles. It is defined below:

$ROLE$ = {($RName$, $SDName$, $PdRle$, IP, Org)$_i$ | $RName_i$ is the name of the i[th] role, $SDName_i$ is the developer's name playing the role, $PdRle$ is shown in Table 1, IP_i is the IP address to access the role, and Org_i is the organization containing the role.}

- F. *TOOL* is the set of tools. It is defined below:

$TOOL$ = {($TlName$, $PdTl$)$_i$ | $TlName_i$ is the name of the i[th] tool. $PdTl_i$ is shown in Table 1}

- G. *PDDEP* is the set of product dependencies. After finishing an activity, every output product depends on every input product. *PDDEP* is defined below:

$$PDDEP = PD \rightarrow 2^{PD}$$

- H. *PDACT* is the relationships between products and activities. If a product is developed by an activity, there is a relationship between the product and the activity. *PDACT* is defined below:

$PDACT$ = {($Pd \rightarrow Act$)$_i$ | Pd_i is developed by the activity Act_i. $Pd_i \in PD$ and $Act_i \in ACT$}

- *PDROLE* is the relationships between products and roles. If a product was developed by one or more roles, the relationships are established between the product and the roles. *PDROLE* is defined below:

$$PDROLE = PD \rightarrow 2^{ROLE}$$

PDDEP facilitate handling the ripple effects induced by changing or removing a product. *PDACT* and *PDROLE* facilitate correcting a product. That is, the original developers should re-enact the original activity to correct a product if necessary.

The knowledge base KPKB should be associated with a set of functions to manage the knowledge. We collectively call the functions the *KPKB knowledge base manager*. To simplify describing the manager, *we use the component of a definition as a function to retrieve the component*. For example, $Status(Pd_i)$ retrieves the status of the product Pd_i. Important KPKB management functions are listed below.

- *getActID*(Pd_i). This function returns the identity of the activity that developed the product Pd_i. According to Definition 1, the function returns $ActID_i$ in which $Pd_i \rightarrow ActID_i$.
- *getSDSet*(Pd_i). This function returns the *IP set* of the software developers that developed the product Pd_i (the software developers played proper roles to develop the product). According to Definition 1, the function returns the set $\{IP(Rle_i) \mid (Pd_i \rightarrow Rle) \wedge (Rle_i \in Rle)\}$.
- *getDepPdSet*(Pd_i). This function identifies the products affected by removing or changing Pd_i. Therefore, it returns those directly or indirectly dependent on Pd_i. According to Definition 1, the function returns the set *depPdSet* defined as: $\{Pd_j \mid (Pd_j \rightarrow Pd_i) \vee (\exists Pd_k \in depPdSet\ ,\ Pd_j \rightarrow Pd_k)$. *depPdSet* is recursively defined to identify the products indirectly dependent on Pd_i.

The KPKB manager also offers functions for the statement of +Activity, +Product, and so on. The functions insert to KPKB the information obtained from the parser. As to the functions that implement -Activity, *Activity, -Role, *Role, -Product, and *Product, they handle exceptions. We describe them in the next sub-section.

C. KPSEE Functions

A parser for KPSEE process language is the basic function. After parsing a statement, the parser invokes functions to take proper actions. For example, after parsing the +Activity statement, the parser invokes the KPKB manager function to insert the activity information to KPKB. In addition to the parser, KPSEE offers a *proactive* function to enact process programs and *reactive* ones to handle exceptions. The proactive function identifies activities with true conditions and available input products (the activities are enactable). For an enactable activity, the proactive function informs the required roles. To prevent technique leakage, only roles in the organizations that can access all the input products are informed. An idle role being informed should react. After the reacted roles are enough, the activity is enacted immediately. After an activity is finished, the data structure of KPKB is adjusted.

To inform roles, two approaches can be applied. Firstly, roles in the authorized organization list and those that can access all the input products are informed simultaneously. Secondly, roles that can access all the input products are informed first. If the reacted roles are not enough after a time period, roles in the authorized organization list are informed. The second approach takes authorized organizations as valuable resources and should be used only when necessary. We accept the second approach. The execution logic of the proactive function is shown in Algorithm 1.

Algorithm 1. Execution logic of the proactive function (it enacts process programs).

function *enactProcess*():
  if *avl*(*IPd*(*ActID*$_i$)) $\wedge$ *Cond*(*ActID*$_i$) then
    insertPd(*OPd*(*ActID*$_i$), "U"); // Insert output products with status "U".
    inform(*Rle*(*ActID*$_i$), ($\cap_j$ *Org*(*IPd*(*ActID*$_i$))$_j$)) $\cup$
      ($\cap_k$ *TOrg*($\cap_j$ *Org*(*IPd*(*ActID*$_i$))$_j$)$_k$); // Inform the required roles in the organizations that are allowed to access all the input products.
      setTimeout(*time*); // Set timeout counter.
  end if;
  if *enough*(*Rle*(*ActID*$_i$)) then // Enough roles causes activity enactment.
    enact(*subRle*, *ActID*$_i$) in which *subRle* $\subseteq$ *(Rle(ActID*$_i$*));*
  else if *Timeout*() $\wedge \neg enough(Rle(ActID_i))$ then
    inform(*Rle*(*ActID*$_i$), *Rle*(*AO*)); // If the reacted roles are not enough, inform roles in the authorized organization list.
    if *enough*(*Rle*(*ActID*$_i$)) then
      enact(*subRle*, *ActID*$_i$) in which *subRle* $\subseteq$ (*Rle*(*ActID*$_i$));
    end if;
  end if;
  if *finish*(*ActID*$_i$) then // If *finish* is true, the activity is finished.
    setStatus(*OPd*(*ActID*$_i$), "A"); // The output products become available.
    setStatus(*ActID*$_i$, "F"); // The activity has been finished.
    $\forall OPd_i \in OPd(ActID_i)$ do
      $\forall IPd_i \in IPd(ActID_i), Opd_i \rightarrow IPd_i$;
      $\forall Rle_i \in subRle, Opd_i \rightarrow Rle_i$;
      $Opd_i \rightarrow ActID_i$;
    end do;
    HouAct(*ActID*$_i$); // Do the housekeeping actions of the activity.
  end if;

There are built-in functions offered by KPSEE, such as *avl* and *setTimeout*. Perhaps the most important built-in function is *inform*, which informs the roles required by an activity when its condition is true and input products are available. Parameters of the function include: (1) the roles required by *ActID*$_i$ and (2) the roles' organizations. The set *subRole* in Algorithm 1 is a subset of the reacted roles informed by the proactive function. Roles in *subRole* are selected to enact the activity.

The reactive functions handle exceptions. Exceptions may be caused by changing user requirements or software processes, verification failure, the departure of software developers, and changing the trust relationships among organizations. Changing user requirements or software processes may result in the addition, change, or removing of activities or products. Verification failure may result in correcting products. The departure of software developers may result in replacing software developers. And, changing the trust relationships among organizations may result in changing the list of organizations that can access a product. The addition of activities, products, roles, and organizations and the handling of authorized organizations are not described here because they are KPKB manager functions. The handling of the statement -Organization is not described because it only causes the deletion of an organization. The statement *Organization causes the change of trustable organizations of an organization. The change will affect the organizations that can access a product. The organizations that can access a product are dynamically identified during process program enactment (i.e., the *Organization statement will affect the set "$\cap_k$ *TOrg*($\cap_j$ *Org*(*IPd*(*ActID*$_i$))$_j$)$_k$" in Algorithm 1). In other words, the *Organization statement will not affect other data structure in KPKB. Therefore, it is not described. The important exception handling functions are described below.

- Change a product. This function implements the statement *Product. Changing a product may result in changing the products directly or indirectly dependent on the changed one, which is a ripple effect.

Algorithm 2. Change a product.

function *chgPd*(*Pd*$_i$):
  ActID$_i$ = *getActID*(*Pd*$_i$); // The KPKB function *getActID* identifies the activity that produced *Pd*$_i$.
  setStatus(*Pd*$_i$, "U"); // The function *setStatus* sets the status of a product or an activity.
  if (*status*(*ActID*$_i$) = "E") then
  inform(*Rle*(*ActID*$_i$), "Stop enactment", *ActID*$_i$);
  end if; // If the activity producing *Pd*$_i$ is being enacted, inform the roles enacting *ActID*$_i$ to stop enactment. The *inform* statement is overloaded.
  setStatus(*ActID*$_i$, "D"); // Changing *Pd*$_i$ means the activity producing it becomes incorrect and should be removed. Software developers should redesign the activity and re-enact it to produce the correct *Pd*$_i$.
  // The following statements handle ripple effects.
  affPdSet = *getDepPdSet*(*Pd*$_i$); // The KPKB function *getDepPdSet* identifies the products directly or indirectly dependent on *Pd*$_i$.
  $\forall$ *Pd*$_j \in$ *affPdSet* do
    ActID$_j$ = *getActID*(*Pd*$_j$);
    setStatus(*Pd*$_j$, "U");
    if (*status*(*ActID*$_j$) = "E") then
      inform(*Rle*(*ActID*$_j$), "Stop enactment", *ActID*$_j$);
    end if;
    setStatus(*ActID*$_j$, "D");
  end do;

- Remove a product. This function implements the statement -Product. Removing a product may also result in ripple effects.

Algorithm 3. Remove a product.
function *rmvPd*(Pd_i):
$ActID_i$ = *getActID*(Pd_i);
setStatus(Pd_i, "D");
if (*status*($ActID_i$) = "E") then
inform(*Rle*($ActID_i$), "Stop enactment", $ActID_i$);
end if;
setStatus($ActID_i$, "D"); // When a product is removed, the activity producing it should also be removed.
// The following statements handle ripple effects.
affPdSet = *getDepPdSet*(Pd_i);
$\forall Pd_j \in affPdSet$ do
$ActID_j$ = *getActID*(Pd_j);
setStatus(Pd_j, "D");
if (*status*($ActID_j$) = "E") then
inform(*Rle*($ActID_j$), "Stop enactment", $ActID_j$);
end if;
setStatus($ActID_j$, "D");
end do; // Pd_j may depend on Pd_i and others. In this case, Pd_j and the activity developing Pd_j become incorrect and should be removed. The activity should be re-designed then re-enacted to product correct Pd_j.

- Change an activity. This function implements the statement *Activity. The actions of changing an activity with different status will be different.

Algorithm 4. Change an activity.
function *chgAct*($ActID_i$, *newIPd*, *newOPd*, *newCond*, *newAction*, *newRle*, *newTool*, *newHouAct*):
setStatus($ActID_i$, "D"); // Remove the activity to be changed.
addAct($ActID_i$, *newIPd*, *newOPd*, *newCond*, *newAction*, *newRle*, *newTool*, *newHouAct*); // Add the activity that have been redesigned.
if (*status*($ActID_i$) = "W") then // The activity is waiting for enactment.
// Do nothing.
else if (*status*($ActID_i$) = "E") then // The activity is enacting.
inform(*Rle*($ActID_i$), "Stop enactment", $ActID_i$);
// Inform the roles enacting the changed activity to stop the enactment.
setStatus($ActID_i$, "W"); // Wait for re-enactment.
else if (*status*($ActID_i$) = "F") then // The activity has been finished.
pdSet = *OPd*($ActID_i$);
$\forall pd_i \in pdSet$,*chgPd*(pd_i);
setStatus($ActID_i$, "W");
end if; // If $ActID_i$ finished, the produced products should be changed. The *chgPd* function (Algorithm 2) can be invoked.

- Remove an activity. This function implements the statement -Activity. The actions of removing an activity with different status will be different.

Algorithm 5. Remove an activity.
function *rmvAct*($ActID_i$)
setStatus($ActID_i$, "D"); // Remove the activity.
if (*status*($ActID_i$) = "W") then // The activity is waiting for enactment.
// Do nothing.
else if (*status*($ActID_i$) = "E") then // The activity is being enacted.
inform(*Rle*($ActID_i$), "Stop enactment", $ActID_i$);
else if (*status*($ActID_i$) = "F") then // The activity has been finished.
pdSet = *OPd*($ActID_i$);
$\forall pd_i \in pdSet$,*rmvPd*(pd_i);
end if; // If $ActID_i$ finished, the produced products should be removed. The *rmvPd* function (Algorithm 3) can be invoked.

- Correct a product. When correcting a product, the original developers that produced the product should re-enact the original activity. The original developers are needed because new ones may be unfamiliar with the product.

Algorithm 6. Correct a product.
function *corrPd*(Pd_i):
$ActID_i$ = *getActID*(Pd_i); // Identify the activity that produced Pd_i.
setStatus(Pd_i, "U"); // Avoid an activity to use the incorrect product.
IPset = *getSDSet*(Pd_i); // The KPKB function *getSDSet* identifies the IP set of the original developers that produced Pd_i.
if *avl*(*IPd*($ActID_i$)) then
loop while (*inform*(*IPset*) = FALSE); // Wait for the software developer responses. The *inform* function is overloaded.
enact(*IPset*, $ActID_i$); // The original developers enact the activity.
if *finish*($ActID_i$) then
setStatus(Pd_i, "A");
setStatus($ActID_i$, "F");
end if;
end if;
// Correcting a product may affect others, which should also be corrected. The correction is achieved by recursively invoking *corrPd*.
affPdSet = *getDepPdSet*(Pd_i);
$\forall Pd_j \in affPdSet$, *corrPd*(Pd_j);

- Replace a software developer. This function implements the statement *Role.

Algorithm 7. Replace a software developer.
function *chgSD*(SD_1, SD_2):
$\forall Rle_i \in ROLE \wedge SDName(Rle_i) = SD_1$,
$setSDName(Rle_i, SD_2)$;

// *ROLE* is defined in Definition 1. The function *setSDName* sets the software developer's name that plays a role.

D. Project Monitoring Support

KPSEE monitoring sub-system in Fig. 1 is a proactive function that monitors a project following the rules described in this sub-section. Violation of the rules will be reported to the project manager for proper handling. The rules are KPSEE default settings. Project managers can add, remove, or change them. The rules are described below (see Table 1 for the meanings of the symbols *PType*, *PdRle*, and *PdTl*):

- The rule to monitor software products in an activity $ActID_i$. For product combination, the product types (i.e., *PType*) of both the input and output products should be the same. For product development, the product types of the input products should be the same, those of the output ones should be the same, and those of the output ones should be one larger than those of the input ones.

Rule 1: $[(PTypeSet(IPd(ActID_i)))\text{-}(PTypeSet(OPd(ActID_i))) = \phi\ ] \vee [(Max(PTypeSet(IPd(ActID_i)))\text{-}Min(PTypeSet(IPd(ActID_i))) = 0) \wedge (Max(PTypeSet(OPd(ActID_i)))\text{-}Min(PTypeSet(OPd(ActID_i))) = 0) \wedge (Max(PTypeSet(OPd(ActID_i)))\text{-}Min(PTypeSet(IPd(ActID_i))) = 1)]$

Contents in the first square brackets mean product combination and those in the second mean product development. The function *PTypeSet* extracts product types from a set of products. It equals to $\cup_j PType(IPd(ActID_i))_j$, in which $PType(IPd(ActID_i))_j$ is the type of the j[th] input product of $ActID_i$.

- The rule to monitor roles in an activity $ActID_i$. When input products are referenced to produce output ones, the required roles can use all the products (the symbol $PdRle(Rle(ActID_i))_j$ is the products that can be used by the j[th] role).
  Rule 2: $(PTypeSet(IPd(ActID_i)) \cup PTypeSet(OPd(ActID_i))) \supseteq (\cup_j PdRle(Rle(ActID_i))_j)$

- The rule to monitor tools in an activity $ActID_i$. When input products are referenced to produce output ones, the required tools can operate on all the products (the symbol $PdTl(Tool(ActID_i))_j$ is the products that can be operated by the j[th] tool).

Rule 3: $(PTypeSet(IPd(ActID_i)) \cup PTypeSet(OPd(ActID_i))) \supseteq (\cup_j PdTl(Tool(ActID_i))_j)$

- Rules to monitor the frequencies of changing or correcting products and activities. Large frequencies reflect the risk of premature project or untrained developers.

Rule 4. $corrCnt(Pd_i)+chgCnt(Pd_i) \le FPd_i$

Rule 5. $corrCnt(ActID_i)+chgCnt(ActID_i) \le FAct_i$

Rule 6. $corrCnt(Pd)+chgCnt(Pd) \le FPd$

Rule 7. $corrCnt(Act)+chgCnt(Act) \le FAct$

Rules 4 and 5 monitor the frequencies of individual product and activity. Rules 6 and 7 monitor those of the entire project. The functions *corrCnt* and *chgCnt* return the counts of correcting and changing products and activities, respectively. They are offered by the "KPSEE functions" component in Fig. 1. The numbers FPd_i, $FAct_i$, *FPd*, and *FAct* are offered by the project manager.

- The rule to monitor the frequency of changing roles (i.e., the departure of software developers). Large frequency reflects the risk of unstable development teams.

Rule 8. $deptCnt(Rle) \le FRle$

The functions *deptCnt* returns the departure frequency of software developers. It is offered by the "KPSEE functions" component. The number *FRle* is offered by the project manager.

- G. Rules to monitor schedule and budget of individual activity and the entire project. Over-schedule and over-budget are possibly the most threatening risk. Violation of the following rule(s) will enforce the project manager to take proper actions.

Rule 9. $Time()\text{-}startTime(ActID_i) \le ActScRate*Schl(ActID_i)$

Rule 10. $Budget(ActID_i)\text{-}usedBudget(ActID_i) \le ActBdRate*Budget(ActID_i)$

Rule 11. $Time()\text{-}startTime(Prj) \le PrjScRate*TolSchl$

Rule 12. $Budget(Prj)\text{-}usedBudget(Prj) \le PrjBdRate*TolBudget\ (Prj)$

The function *Time* gets the current time. The function *startTime* returns the start time of an activity or the entire project. The function *useBudget* returns the budget used by an activity or the entire project. Both *startTime* and *usedBudget* are offered by the "KPSEE functions" component. The functions *TolSchl* and *TolBudget* return the total schedule and budget of the entire project. They are obtained from the *+ScheduleBudget* statement. The numbers *ActScRate*, *ActBdRate*, *PrjScRate*, and *PrjBdRate* are offered by the project manager.

- H. The rule to monitor the reaction time of an informed role and that to monitor the waiting time of an enactable activity.

Rule 13. $Time()\text{-}informTime(Rle_i) \le ReactTime$

Rule 14. $Time()\text{-}etblTime(ActID_i) \le WaitTime$

The function *informTime* returns the time when the role Rle_i is informed. The function *etblTime* returns the time when the activity $ActID_i$ is enactable. Both the

functions *informTime* and *etblTime* are offered by the "KPSEE functions" component. The numbers *ReactTime* and *WaitTime* are offered by the project manager. The rules facilitate improving project efficiency.

The monitoring rules reveal that the "KPSEE functions" component in Fig. 1 offers many functions for project monitoring. We do not describe them because of their easiness. For example, *informTime* just records the time when a role is informed.

IV. Features

KPSEE offers the features mentioned in section I. Exception handling is solid because of Algorithms 2 through 7. Software development organizations can be distributed because KPSEE is distributed. Enhancing process flexibility is offered because KPSEE allows dynamic adding, removing, and changing process components anytime during process enactment. Integrating PSEEs can be achieved because placing process programs translated from other PSEEs in any order becomes a KPSEE process program. Keeping pace with significant change is obvious because the change as significant as adding, removing, or changing process component at anytime during process enactment are allowed. The other features are proved below.

- Maximize the degree of process parallelism

If an activity is enacted immediately when its condition is true and its input products, required roles, and tools are available, the degree of process parallelism is maximized. According to Algorithm 1, when the input products of an activity is available and its condition is true, KPSEE informs the roles trusted by the input products. As long as the reacted roles are enough, the activity is enacted without waiting. Therefore, KPSEE maximizes the degree of process parallelism.

- Prevent technique leakage

Suppose Rle_i in Org_i is a role that enacts the activity $ActID_i$. Moreover, Org_i cannot access one or more input products of the activity. If this situation occurs, technique leakage happens. However, Algorithm 1 informs roles in the organizations "$(\cap_j Org(IPd(ActID_i))_j)) \cup (\cap_k TOrg(\cap_j Org(IPd(ActID_i))_j)_k)$" or those in *AO* to enact the activity. If an organization cannot access one or more input products of $ActID_i$, it is not in the organization set "$\cap_j Org(IPd(ActID_i))_j$" or *AO*. In other words, roles in the organizations that cannot access one or more input products will not be informed. This prevents technique leakage.

Using roles in trusted or authorized organizations is the basic concept of information flow control to prevent technique leakage. However, the join operation of an information flow control model [24] is not mentioned (the join operation adjusts the subject that can access an object after an information flow). In fact, the join operation in KPSEE is achieved by the statement "$\forall Rle_i \in subRle\ ,\ Opd_i \rightarrow Rle_i$" in Algorithm 1. With the statement, a product depends on roles producing it. When the product should be accessed to enact $ActID_m$, the set "$(\cap_j Org(IPd(ActID_m))_j)) \cup (\cap_k TOrg(\cap_j Org(IPd(ActID_m))_j)_k)$" can correctly identify the organizations that can access the product.

- Manage product consistency

According to Algorithm 1, a product produced by an activity depends on the input products. When a product should be changed, Algorithm 2 forces the activity producing the product to be changed and enacted to change the product. Moreover, the following algorithm segment ensures that the products directly or indirectly depend on the changed product will be changed accordingly.

```
affPdSet = getDepPdSet(Pd_i);
∀ Pd_j ∈ affPdSet do
    ActID_j = getActID(Pd_j);
    setStatus(Pd_j, "U");
    if (status(ActID_j) = "E") then
        inform(Rle(ActID_j), "Stop enactment",
            ActID_j);
    end if;
    setStatus(ActID_j, "D");
end do;
```

When a product should be removed, Algorithm 3 removes the product and the activity producing the product. Moreover, the following algorithm segment removes the products directly or indirectly dependent on the removed product.

```
affPdSet = getDepPdSet(Pd_i);
∀ Pd_j ∈ affPdSet do
    ActID_j = getActID(Pd_j);
    setStatus(Pd_j, "D");
    if (status(ActID_j) = "E") then
        inform(Rle(ActID_j), "Stop enactment",
            ActID_j);
    end if;
    setStatus(ActID_j, "D");
end do;
```

When a product Pd_i should be corrected, Algorithm 6 requires the software developers that developed Pd_i to re-enact the activity to correct Pd_i. The algorithm also corrects the products directly or indirectly dependent on Pd_i through recursively invoking Algorithm 6 as shown below.

```
affPdSet = getDepPdSet(Pd_i);
∀ Pd_j ∈ affPdSet , corrPd(Pd_j);
```

Note that changing or removing activities may also affect products. Algorithm 4 invokes Algorithm 2 (i.e., the function *chgPd*(pd_i);) to handle the change of affected products and Algorithm 5 invokes Algorithm 3 (i.e., the function *rmvPd*(pd_i);) to handle the removing of affected products. The invocations ensure product consistency.

Currently, our knowledge base (which is KPKB) is homogeneously. To improve the performance of KPSEE, we prepare to upgrade the ability of KPKB to offer the ability of storing heterogeneous knowledge [25]. We need this ability because different software tools may create documents in different formats.

V. Conclusion

According to the change of enactment environments and the increment of software development complexity, PSEE features should be enhanced. We designed a knowledge-based PSEE named *KPSEE*. It offers the following features, in which some are enhanced ones.

- It maximizes the degree of process parallelism. When the input products of an activity are available and its condition is true, KPSEE informs the roles trusted by the input products. As long as the reacted roles are enough, the activity is enacted without waiting. Therefore, KPSEE maximizes the degree of process parallelism.
- It enhances process flexibility. KPSEE offers the flexibility by allowing dynamic adding, removing, and changing any components at anytime during process enactment. The flexibility is achieved by allowing unstructured statements and offering strong exception handling functions.
- It manages product consistency. During the deviation of products and activities, KPSEE properly handles the products directly or indirectly dependent on the changed or removed product. Therefore, KPSEE manages product consistency.
- It integrates PSEEs. KPSEE process statements are unstructured. Therefore, when process programs in different PSEEs are translated into KPSEE process statements. They can be placed in any order to become an integrated KPSEE process program. Therefore, KPSEE integrates PSEEs.
- It keeps pace with significant change of a process. KPSEE allows adding, removing, and changing any component of a process program at anytime. With this, no significant change will affect process enactment. That is, KPSEE keeps pace with significant change of a process.
- It prevents technique leakage. KPSEE is fused with an information flow control model. With the model, when an activity can be enacted, KPSEE informs the roles whose organizations are allowed to access all the input products. This prevents technique leakage.
- It offers project monitoring ability. KPSEE offers rules to monitor project related events. For example, it monitors both schedule and budget of activities and the entire project.

References

[1] L. Osterweil, "Software Processes Are Software Too", *9'th IEEE International Conference on Software Engineering*, 2-13, New York, 1987

[2] SERENA, "An Introduction to Agile Software Development", available on http://www.serena.com/docs/repository/solutions/intro-to-agile-devel.pdf

[3] R. Matinnejad and R. Ramsin, "An Analytical Review of Process-centered Software Engineering Environments", *IEEE 19'th International Conference and Workshops on Engineering of Computer-based Systems*, pp. 64-73, 2012.

[4] S. -C. Chou, "Using Product Status to Coordinate Heterogeneous Process Environments", *IEICE Trans. Information and Systems*, vol. E86-D, no.1, pp.56-62, Jan. 2003.

[5] S. -C. Chou, "ADPE: Agent-based Decentralized Process Engine", *IEICE Transactions on Information and Systems*, E88-D(3), 603-609, Mar., 2005.

[6] S. C. Bandinelli, A. Fuggetta, and C. Ghezzi, "Software Process Model Evolution in the SPADE Environment," *IEEE Transactions on Software Engineering*, Vol. 19, No. 12, 1128-1144, Dec. 1993.

[7] S. -C. Chou and J.-Y. J. Chen, "Process Program Change Control in a Process Environment", *Software - Practice and Experience*, vol. 30, no. 3, 175-197, 2000.

[8] G. Cugola, "Tolerating Deviations in Process Support Systems via Flexible Enactment of Process Models", *IEEE Transaction on Software Engineering*, vol. 24, no. 11, 982-1001, 1998.

[9] D. Kim, M. Kim, H. Kim, "Dynamic Business Process Management Based on Process Change Patterns", *International Conference on Convergence Information Technology*, pp. 1154-1161, 2007.

[10] W. She, I. –L. Yen, B. Thuraisingham, and E. Bertino, "The SCIFC Model for Information Flow Control in Web Service Composition", *2009 IEEE International Conference on Web Services*, 2009.

[11] A. Myers and B. Liskov, "Complete, Safe Information Flow with Decentralized Labels", *14'th IEEE Symp. Security and Privacy*, pp. 186-197, 1998.

[12] J. Y. J. Chen, "CSPL: An Ada95-like, Unix-based Process Environment," the *IEEE Transactions on Software Engineering*, vol. 23, no. 3, pp. 171 - 184, March 1997.

[13] I. Z. Ben-Shaul and G. E. Kaiser, "A Paradigm for Decentralized Process Modeling and its Realization in the Oz Environment", in *Proceedings of the 16th ICSE*, pp. 179-188, 1994.

[14] C. J. Hagen, "A Generic Kernel for Reliable Process Support", Ph. D. Dissertation of the Swiss Federal Institute of Technology Zurich, 1999.

[15] J.-Y. Chen and S.-C. Chou, "Consistency Management in a Process Environment", *Journal of Systems and Software*, vol. 47, pp. 105-110, 1999.

[16] A.D. Lucia, F. Fasano, R. Oliveto, and G. Tortora, "Fine-grained management of software artefacts: the ADAMS system", *Software Practice and Experience*, vol. 40, no. 11, pp. 1007-1034, 2010.

[17] S. Weber, A. Emrich, J. Broschart, E. Ras, and Ö. Ünalan, "Supporting Software Development Teams with a Semantic Process-and Artifact-oriented Collaboration Environment", *Proc. SOFTEAM'09*, 2009

[18] M.A. Almeida da Silva, R. Bendraou, X. Blanc, and M.P. Gervais, "Early Deviation Detection in Modeling Activities of MDE Processes", *LNCS, vol. 6395*, pp. 303-317, 2010.

[19] M.A. Almeida da Silva, R. Bendraou, J. Robin, and X. Blanc, "Flexible Deviation Handling during Sofware Process Enactment", *Proc. EDOCW'11*, pp. 34-41, 2011.

[20] R. S. P. Maciel, R. A. Comes, A. P. Magalhaes, B. C. Silva, and J. P. B. Queiroz, "Supporting Model-driven Development Using a Process-centered Software

Engineering Environment", *Automated Software Engineerin*g, 20(3), pp. 427-461, 2013.
[21] F.A. Aleixo, M.A. Freire, W.C. dos Santos, and U. Kulesza, "Automating Variabitliy Management, Customization and Deployment of Software Processes: A Model-Driven Approach", *Proc. ICEIS'11*, pp. 372-387, 2011.
[22] R. S. P. Maciel, B. C. Silva, and N. S. Rosa, "An Integrated Approach for Model Driven Process Modeling and Enactment", *Proc. SBES'09*, pp. 104-114, 2009.
[23] S. –C. Chou, W. –C. Hsu, and W. –K. Lo, "DPE/PAC: Decentralized Process Engine with Product Access Control", *Journal of Systems and Software*, 76(3), 207-219, June, 2005.
[24] A. C. Myers, "JFlow: Practical Mostly-Static Information Flow Control", *Proceedings of the 26'th ACM Symposium on Principles of Programming Language*, 228-241, 1999.
[25] M. K. Yusof, A. F. A. Abidin, and M. N. A. Rahman, "Architecture for Accessing Heterogeneous Databases", *International Journal of Information Technology and Computer Science(IJITCS)*, vol 4, no. 1, pp. 25-31, 2012.

Authors' Profiles

Shih-Chien Chou is a Professor in the Department of Computer Science and Information Engineering, National Dong Hwa University, Taiwan. He is major in software engineering, process environment, software reuse, and information flow control.

Chiao-Wei Li is a graduate student in the Department of Computer Science and Information Engineering, National Dong Hwa University, Taiwan.

How to cite this paper: Shih-Chien Chou, Chiao-Wei Li,"A Knowledge-based PSEE with the Ability of Project Monitoring", IJIEEB, vol.6, no.4, pp.1-11, 2014. DOI: 10.5815/ijieeb.2014.04.01

