ViPen: A Model Supporting Knowledge Provenance for Exploratory Service Composition

Weilong Ding[1,2], Jing Wang[1], Yanbo Han[1]
[1] *Institute of Computing Technology, Chinese Academy of Sciences*
Beijing, 100190, P.R. China
[2]*Graduate University of Chinese Academy of Sciences*
Beijing, 100190, P.R. China
{dingweilong,wangjing,yhan}@software.ict.ac.cn

Abstract

During the current service composition, the intermediate products such as data and related process fragments are neglected in many cases when the deviation occurred from predefined composition. The lack of adequate provenance support makes it not convenient when building the composition in an exploratory fashion. In this paper, we present ViPen model, and illustrate its operations to enhance the provenance for flexible deviation in service composition. Meanwhile, a partially ordered relation "derivation" is formed which guarantees the new derived processes can reuse previous knowledge effectively. A case study of one bioinformatics experiment is explained to verify our work.

Keywords: flexibility, provenance, service composition, workflow

1. Introduction

Service composition has become increasingly popular in both business and scientific domains. Due to the dynamic nature of user requirements and various environments, the composition logic usually cannot be precisely specified beforehand, or needs to be adjusted at runtime in an exploratory manner. Exploratory service composition [1-2] and flexible workflow [3-4] are paid great attention in academic these years. Flexibility in service composition or workflow is the ability to deal with both foreseen and unpredictable changes, which has different extensional concepts. The flexibility by derivation is one type, which means process instances deviate from the execution path prescribed by the original process at runtime without altering its process model [3].

The deviations of service composition are focused by many workarounds recently, but there are still two drawbacks currently. On the one hand, deviation at runtime implies numerous intermediate products generated, which may be valuable knowledge but neglected to manage in a systematical way. The knowledge in this paper is the previous experiences like the data and process fragments gained during the service composition, and is ponderable for reuse. For the flexible services composition, knowledge management has to expend manually, but it is not only exhausting, but also error-prone. On the other hand, current provenance systems (e.g., Vistrails[5]) only emphasize the intermediate data by technical details in fine granularity, such as locating tracks in version-control perspectives. But it may be hard for non-IT-professional users to comprehend what is in provenance in business perspective during the service composition. In a word, it is difficult to manage intermediate products under exploratory context, and not convenient to reuse knowledge in coarser granularity. For example, scientists build workflows by service composition for experiments, in which they may not foresee all the cases at runtime. So the on-the-fly deviations from the predefined template are inevitable. It will be expensive to build process radically without adequate provenance support, for its time-consuming services and complex composition logic. And it is not convenient to reuse previous knowledge only by technical details as versions.

In order to manage the intermediate products in provenance and reuse them when the processes evolved at run time, ViPen (Vinca Process Provenance) model is proposed in this paper. There are two advantages in our work:

First, both intermediate data and process fragments are in provenance in an incremental manner. It is economical in space but also readable for business meanings.

Second, by the hierarchical organization of knowledge, partially ordered relation *derivation* is formed. The previous fragments during composition can be reconstructed by the knowledge in provenance in a precise way.

This paper is organized as follows. Section 2 describes the ViPen model for exploratory service composition. Section 3 elaborates the provenance-related operations in ViPen. Section 4 presents a case study by a practical protein analysis experiment. Section 5 shows the related work, followed by conclusion in Section 6.

2. ViPen model for service composition

ViPen is a provenance model for exploratory service composition. The concepts of ViPen are defined in this section.

2.1. ViPen model

To depict exploratory service composition with provenance support, ViPen (Vinca Process Provenance) model is proposed. Unlike the concepts of BPEL or XPDL in BPM for workflow, whose process structure must be fully predefined before running, ViPen supports incomplete structure and dynamical improvement during the run time. Vipen is a bipartite graph which contains two types of element: *activity* and *transition*.

The *activity* is the executable task in the service composition. *Activity = <activityID, type, state>*, where *type = {service, start, end, orSplit,orJoin, andSplit, andJoin}* and *state = {init, running, suspend, executed}*. When type is *service*, *acitivity* can be some kinds of service such as local Java service, standard Web Service, Soaplab service, Biomoby service and Biomart service etc. The *state* of activity reflects the status in the run time. Note that, the I/O parameters of the services can be configured as data association from parameters of previous services. The *transition* is a mapping from activity to activity. *Transition=<tranID, fromAct, toAct>*, which expresses one activity can be triggered after another activity is finished.

With activity and transitions, composition can be organized as workflow in three-levels in ViPen: *template*, *instance* and *track*, as Fig. 1 showed.

Template

Template=<templateProfile, activities, transitions>

A *template* describes the common backbone structure shared by some process instances. The *templateProfile* is the basic information like the *templateID* and *name*. The *activities* and *transitions* are set of activities and transitions respectively. Templates are defined by the process designer from scratch or from the executed tracks, which will be discussed later.

Instance

Instance=<instanceID, name, templateID>

The *instance* describes an exploratory procedure for certain goals through workflows, and regarded as a package of some *tracks*. Note that, *templateID* is either a pre-defined template identifier (i.e. a certain value of *templateID* in *template*) or NULL. If the former, the structures of all the included tracks of this instance should comply with the given template; otherwise the tracks are constructed from scratch without any template constrains.

Track

Track=<trackProfile, status, LocalAct, LocalTran, directDeriv>

The *Track* is the ultimate executable workflow unit, which contains the parameterized initiated activities. The *trackProfile* contains *trackID*, *name*, *createTime* and *instanceID*. A track must belong to one certain instance, so *instanceID* can not be null. The *status = {init, running, suspend, complete}*, ,i.e., the track in the *init* state when created, in *running* state when being executing, in *suspend* state when parameters need to be assign or error occurs, and in *complete* state when the execution finishes. *LocalAct* and *LocalTran* are the activities and transitions respectively belonged to this track. The *directDeriv* locates the direct original tracks in *derivation* relation, and that will be demonstrated in next section.

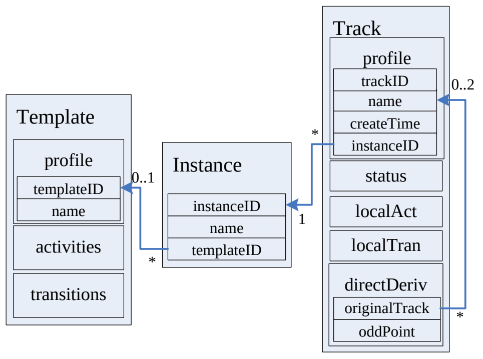


Fig. 1 ViPen Model

Note that, the structure of neither template nor track needs to be complete. These tracks will suspend during runtime at activities with no post-transitions except for the end activity. The absence of process structure could be filled incrementally on-the-fly.

2.2. Derivation relation between tracks

Tracks in ViPen model are preserved incrementally, which means one track may be relevant to other tracks sharing some process fragments. The relation between them is *derivation*.

Def. Derivation. Binary unidirectional relation *derivation* denoted as *derive(tr1,tr2)* exists between the original track *tr1* and the derived track *tr2*. There is at least one identical activity between *tr1* and *tr2*. One process fragments are identical with another if only the same activities and transitions (the same identifiers) in *executed* state are included. That is, two tracks are in the *derivation* relation if they share identical fragments, in which the early (by *createTime*) track is original track and the other is the derived one. Furthermore, the two tracks *tr1* and *tr2* are in *direct derivation*, denoted as *direct_dervie(tr1, tr2)*, if $\exists$ *activity a,b. a* $\in$ *tr1.localAct, b* $\in$ *tr2.localAct* $\exists$ *transition t, t* $\in$ *tr2.localTran, t.fromAct=a, t.toAct =b*. The *direcDeriv* in track points to the direct original tracks. *direcDeriv = < originalTrack, oddPoint>*, where the *originalTrack* is either null or the identifier of an existing track. If the former, all its fragments are preserved in *localAct* and *localTran*. Otherwise the track is the direct derived from *originalTrack*, and *oddPoint* refers to the activity in *originalTrack* where the derivation starts. The shared fragments are preserved only in original tracks and the derived tracks can reuse them.

The *derivation* has some properties in nature, which guarantee the track can be constructed with previous knowledge incrementally precisely.

Theorem : derivation is a partially ordered relation.

Proof: The partially ordered relation has three properties: reflexive, anti-symmetric, and transitive, which we should certificate *derivation* holds.

The first two properties *reflexive* and *anti-symmetric* can be concluded intuitively from the definition of *derivation*. A track must be wholly identical with itself. Meanwhile, if *derive (tr1,tr2), derive(tr2,tr1)* can not be deduced because *derivation* is unidirectional relation. Thus both *reflexive* and *anti-symmetric* are held in *derivation*.

To prove the third property, let's assume the identical fragments *pf* and *pf'* are respectively shared in *derive (tr1,tr2)* and *derive(tr2,tr1)*. The intersection between *pf* and *pf'* can not be empty set, because both *pf* and *pf'* contains fragments in track *tr2* and *tr2* is derived from *tr1*. So we define *pf''=pf* $\cap$ *pf'*, and *pf''* is the identical fragment shared between *tr1* and *tr3*. As a result, *tr3* are also derived from *tr1*, and they are in *derivation* (but not *direct derivation*). So *derivation* is transitive. All this leads up to the partially ordered relation.

The provenance is supported in ViPen by the derivation relation, and next we will talk about the modeling approach for ViPen, and especially focus on the provenance related operations.

3. Modeling and usage of ViPen with provenance-related operations

There are more than 30 operations on modeling ViPen, which are classified into five categories by the different objectives (i.e., activity, transition, template, instance and track) of the operations. The operations in each category contains *add, delete, configure* and other specific operations. With ViPen, knowledge is in provenance through a convenient but strict way during exploratory service composition at runtime. In this section, we focus on the provenance-related operations: *add_probe_point*, *merge_track* and some inquiry operations, by which *derivation* is formed or employed.

3.1. Add-probe-point operation

A new track can be derived by *add_probe_point* operation on one track. A *Probe Point* is an executed activity in the original track that marks the location where the derivation starts. The *direcDeriv* in the derived track points to the direct original track, in which the *oddPoint* is the probe point in *originalTrack.*

The operation *add_probe_point(tr1,a)* has two inputs and one output. The inputs are the existing track *tr1* and a local executed activity *a* in *tr1*. The output is a derived track *tr2*. The relation *direct_derive(tr1,tr2)* is formed after the operation, because there is one unique identical process fragment between two tracks, which lies in the fragment of *tr1* from the start activity to direct previous activity of probe point *a*. The semantic of *tr2 = add_probe_point (tr1, a)* implies:

1) *tr2. originalTrack = tr1.trackID*

2) *tr2. oddPoint = a.activityID*

3) *tr2.localAct={oi|* $\forall$*si* $\in$ *reachableAct(a),* $\exists$ *oi=clone(si) } , i=1..n, n=|reachableAct(a)|*

The function *reachableAct(activity)* gets the respective set of the reachable activities including this *activity* itself. The function *clone(activity)* creates new activity in the derived track *tr2* by deep-copying from *activity* in *tr1*, where service inputs are hold as default values, but a new identifier (*trackID*) is assigned, state is set to *init*, outputs values are cleared, and related new transitions are generated. The Fig. 2 shows the procedure that *localAct* and *localTran* in the derived track stem from cloned fragments. Then *localAct* and *localTran* in *tr2* can be modified like reconfiguration or adding new process fragments, which have no influence on the original track.

By this operation, a new track can be derived from only one original track, and form a tree-like structure in provenance in ViPen. Due to the partially ordered *derivation* relation, the derived track can be constructed by back-tracking the original track.

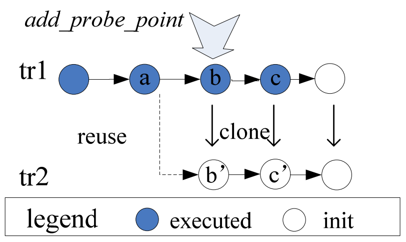


Fig. 2 Original Track and Its Derived Track

3.2. Merge-tracks operation

In order to reuse the knowledge from more than one track, merge operation is introduced. There are always reusable results among some fragments in different tracks especially for the similar goal. In the computation-sensitive context, it is not efficient to repeat these fragments manually every time.

A new track can be derived by *merge_track* operation from two original tracks. Note that, if the original tracks contain identical fragments (e.g., in *derivation* relation), the identical fragment is melted into one. The *direcDeriv* of the track points to the two direct original tracks, where *oddPoint* here refers to two activities in *originalTrack*, which are the *sPoint* and *jPoint* respectively. The *sPoint* and *jPoint* in the original tracks refer to the activities, where the and-split branch of the derived (merged) track starts and ends respectively.

The operation *merge_track(tr1, tr2)* have two inputs and one output. The inputs are two existing track *tr1*, *tr2*. The output is a new generated track *trm*. The relation *derive(tr1, trm)* and *derive(tr2, trm)* are formed after the operation, and they are also the *direct derivation* relation if $jPoint \in tr_i.localAct$, i=1,2. For concise description, we assume these direct derivations in the following. There are identical process fragments between *tr1* and *trm*, which lies in the fragment in *tr1* from start activity to the direct previous *sPoint* activity, and the fragment from *sPoint* to *jPoint*. Similarly, there are identical fragment between *tr2* and *trm*, which are the fragment in *tr2* from *sPoint* to *jPoint*. The semantic of *trm = merge_track(tr1, tr2)* implies:

1) *trm. direcDeriv [i-1]. originalTrack* $=tr_i.trackID$, *i=1,2*

2)*trm. direcDeriv [i-1]. oddPoint[0] = split Service* (tr_i) , *trm. direcDeriv [i-1]. oddPoint[1] = joinService*(tr_i), *i=1,2*

3)*trm.localAct={oi|* $\forall si \in$ *reachableAct (next(joinService(tr1))),* $\exists$ *oi=clone(si)* , *i=1..n, n=|reachableAct(next(joinService(tr1)))}|* $\cup$ *{newAndSplitAct(), newAndSplitAct()}*

The functions *splitService(track)* and *joinService (track)* return the first and last non-identical service activity in executed state respectively in certain track. The functions *newAndSplitAct()* and *newAndSplitAct()* generate and-control activity(i.e., *andSplit* and *andJoin* type). Other functions have been discussed in section 3.1. Fig. 3 shows the semantic of the mergence procedure. Similarly, only the partial fragments (i.e., the local activities and transitions) are preserved in a track in ViPen. The process designer can also modify the *localAct* and *localTran* on demand.

By this *merge_tracks* operation, a new track can be derived from two original tracks, and the new tack and its original tracks form a graph-like structure in provenance. Due to the partially ordered *derivation* relation, the new track can be constructed by back-tracking the original tracks.

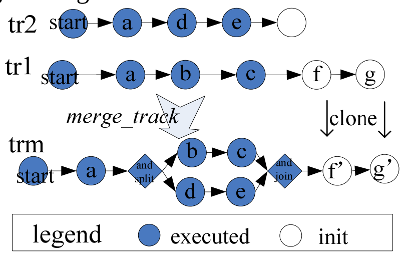


Fig. 3 Mergence of Two Tracks

3.3. Inquiry-track operations

Only the partial fragments *localAct* and *localTran* are persisted in one track by ViPen model. The track structure could be deduced by back-tracking from the original tracks through *derivation* relation. So the operation *getTrack* is introduced.

In the operation *getTrack(trackID)*, the input is a certain track identifier and the output is its whole structure. The concise pseudo code for this operation is showed here. The function *findTrackInDB* gets the track object in the persistence storage. The *fragment* is the set of *activity* and *transition*. The function *reachFrag(activity)* returns the reachable process fragment from this *activity* and including itself. Function *toTran(activity)={tran|* $\exists$ *transition tran, tran.toAct= activity }.*

```
getTrack(String trackId)
begin
  Track tr=findTrackInDB(trackId)
  return wholeTrack(tr,null);
end

wholeTrack(Track tr, oddPoint op)
begin
fragment=tr.localAct∪tr.localTran
if op!=null
  fragment=fragment-reachFrag(op)-toTran(op)
end if
if  tr.directDeriv!=null
  for each track t in directDeriv
    for for each oddpoint odp in track
      fragment=fragment∪wholeTrack(t,odp)
    end for
  end for
end if
return fragment
end
```

Meanwhile, the original or derived tracks could be retrieved by operation *getOrigin(track)* and *getDeriv (track)*. These two operations are implemented essentially by back-tracking or forward-tracking the *directDeriv* of a certain track. The hierarchical organization of tracks in one instance provides a concise and flexible means to preserve the exploratory behaviors during the run-time. By the relation *derivation*, the tracks in provenance form the hierarchical inheritance. On the one hand, the incremental fashion reduces the space for provenance of data and process fragments than the whole preservation; On the other hand, reusable intermediate products as knowledge can save the time in exploratory service composition.

3.4. Exploratory service composition with ViPen

With the ViPen model, service composition could be achieved in an exploratory way. New track can be created by both from scratch and reusing existing knowledge. The former manner may suffer the non-efficiency in building service composition under computing-sensitive context. In the user-steering exploratory manner [1], the executed fragments may be valuable knowledge for intelligent decision support. At first, the process designer could only build incomplete structures for templates or tracks at design time, and then progressively fulfill the tracks during the run time, such as adding and reconfiguring activities or even deriving new tracks via related operations. Tracks are organized in a business perspective: the tracks (including their derived ones) for certain goals could be assigned in certain instance on user’s demand. The knowledge could be employed conveniently in a comprehensive way because users just need to indicate which fragments are reused in business perspective. The provenance support for late-modeling approach increases the flexibility of service composition without the loss of business implications.

4. Implementation and case study

Vinca Service-Browser[1] IDE is an online service composition system developed by our team to support ViPen model. The activities in composition are service of several types including standard Web Service, Soaplab service and local beanshell java service currently. Provenance is supported in the run time environment as Fig. 4 showed. Fig. 5 is the IDE screenshot. The services and compositions (by template, instance and tracks) are listed in directories in the left panel. The service recommendation [1-2] at bottom panel is another key contribution of our team. In edit panel, scientists can build their service composition as workflow by drag and drop with corresponding GUI.

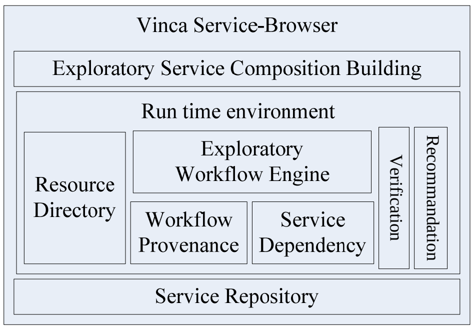


Fig. 4 Architecture of Vinca Service-Browser

In this section, we build an exploratory service composition for one practical bioinformatics-experiment as a case study with Vinca Service-Browser. The sample workflow can also be found in myExperiment[2] community. The community sharing thousands of scientific workflows is initiated by ^{my}Grid [6] project in Europe for scientific experiments. This workflow performs a generic protein sequence analysis whose goal is to draw phylogenetic tree of a certain protein. The input can be analyzed along with a list of known protein identifiers chosen by the scientists to perform a homology search, followed by a multiple sequence alignment and finally a phylogenetic analysis. The scientists may determine the next step after review the returned results from previous services, and modify the process on the fly. It is a typical exploratory experiment and emphasis reuse of previous knowledge in the computing-sensitive context.

There are some kinds of services most frequently used about protein sequence analysis: *BLAST* and *Clustal*. The *BLAST* services enable researchers to compare a queried sequence with a library or database

[1] The trial version is available at http://sigsit.ict.ac.cn
[2] http://www.myexperiment.org/

of sequences, and identify library sequences that resemble the queried sequence above a certain threshold. The *Clustal* services are widely used as multiple sequence alignment programs, by which similar characters are aligned in successive columns. Each type has some different algorithm in implementation for specific situation. Fig. 5 also shows the first steps in building the sample workflow in our IDE. In this experiment, scientists attempt to build the workflow tracks from the skeletons by a template, and then employ two services *Blastp* and *ClustalW* which are the respective implementations of *BLAST* and *Clustal* approaches.

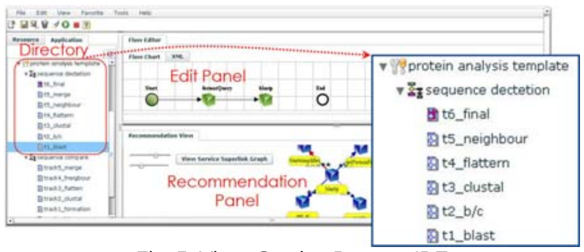


Fig. 5 Vinca Service-Browser IDE

When the incomplete process is executed, the track suspends at *ClustalW* service (in pink color) for unmatched input format, like topmost track in Fig. 6. Considering the *Blastp* is a remote service provided by DDBJ[3], scientists desire to reuse the already gained results to update the track. So by *add_probe_point* at *ClustalW* activity, a derived track is generated, like the middle track in Fig. 6. After modifying the local activities (in blue color), several beanshell services are added before *ClustalW* service in the derived track, and all the services can be executed (in green color), showed by the bottom track in Fig. 6.

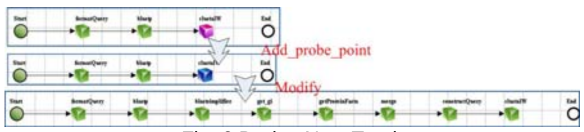


Fig. 6 Derive New Track

Scientists can then draw the phylogenetic tree by two approaches (i.e., *flatternImage* or *fneighbour* algorithm) in independent tracks, showed by the topmost two tracks in Fig. 7. And then a new merged track for comparison is generated by *merge_track* these two tracks. The merged track is showed by the bottom track in Fig. 7. Finally the track is executed to the end after adding a transition between the and-join activity and the end activity as Fig. 8 showed. All the intermediate tracks are organized in the left directory, and the knowledge (i.e., data and process fragments within them visually) is in provenance. Due to *derivation* relation, the visual structure of one track in IDE can be constructed precisely from partly preserved fragments in provenance, which is not aware by the users. Note that, all the tracks (including those intermediate incomplete ones) can be saved as new template at any time.

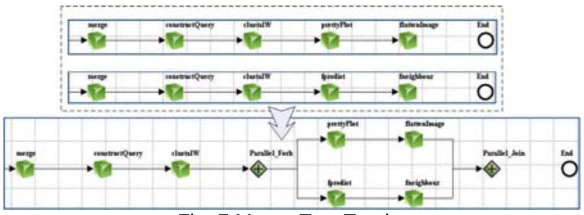
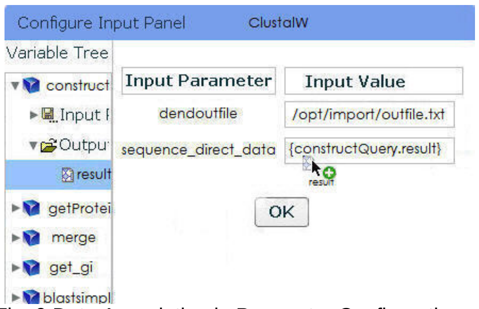


Fig. 7 Merge Two Tracks

Fig. 8 The Final Track for the Experiment

Provenance for both data and process fragments is supported conveniently in exploratory service composition. The data is preserved within the tracks, and even as data association. The scientists can configure data associations by right-click certain service in Vinca Service-Browser, and drag the names or values (e.g., Fig. 9) from the parameters of previous services. The data association is constructed and data flow will be generated at runtime.


Configure Input Panel ClustalW

Variable Tree

Input Parameter	Input Value
dendoutfile	/opt/import/outfile.txt
sequence_direct_data	{constructQuery.result}

Fig. 9 Data Association in Parameter Configuration of ClustalW Service

5. Related work

We survey the related work systematically into two categories: flexibility and provenance for service composition, which cover the challenges discussed in Section one. To evaluate the benefits of ViPen model on provenance for exploratory service composition, we compare our work with related work. We focus on the rationale, implementation and efficiency in each of them.

[3] http://xml.nig.ac.jp/index.html

Flexibility of service composition or workflow can be regarded as the ability to deal with both foreseen and unforeseen changes, by varying parts of the process, while the essence of workflow is not impacted. Although the concepts of flexibility has some synonyms like adaptability and dynamicity, which may has different focus in certain context [4], the intrinsic nature of these "flexibility" are as much about what should stay the same in a process as what should be allowed to change. Schonenberg et al [3] surveyed the challenges in flexible workflows like the expressive ability and what extend supported. The flexibility by derivation at runtime is our concern. Based on the existing tools, such as ADEPT [7], DECLARE [8], a range of flexibility have been achieved in the control-flow perspective. But the intermediate products are ignored during the variation, which may be the key elements in some fields such as scientific computation. Caeiro-Rodriguez et al [9] and Davidson et al [10] particularly indicate that the provenance is one of the intrinsic challenges and imperative requirements in scientific workflow.

Provenance in service composition or workflows, both for the intermediate data and exploratory process fragments, is essential to allow for process reproducibility, sharing, and knowledge re-use in the process management. Data provenance is emphasized in some practical fields, such as scientific-computing. Taverna[11] was initiated by the ^{my}Grid-Science UK project, and is implemented in a service-oriented architecture, and supports multiple service standards in bioinformatics. Kepler[12] is developed by the University of California at Berkeley, built upon a mature dataflow-oriented architecture (i.e., Ptolemy II) with service support. Each of the two systems captures the details such as who conducted the experiment, which services were used and what results were obtained. They also support modification on the fly. The data obtained by previous services can be reused but only in the current process. Although Taverna can list all the executed processes with details, these processes are all read-only, independent with each other, and can not be shared again. All these intermediate products are cleared after the tool is closed. In a word, only the data provenance for current workflow is partly supported in these two systems. In fact, intermediate data and process fragments are both crucial in rebuilding an experiment in exploratory context.

Meanwhile, process provenance is brought forward in another way by version management. Davidson et al [10, 13] present techniques managing the nested scientific data and the process evolution in the provenance, and Vistrails[14] developed by University of Utah implements this rationale. Vistrails enables multiple-view visualizations by flexible dataflow in an interactive way. The distinguished feature is the action-based provenance mechanism, which records each operation on dataflow. All the process instances form a tree, where node corresponds to a dataflow version and edge between nodes represents a set of change actions applied to the parent node. While in the tree-like fashion, new tracks could be generated from only one parental fragment at one time, which limits the re-usability in a new complex process structure. Zhao et al [15] propose a version preserving directed graph (VPG) model and its operations for provenance. The multiple different versions generated by updating workflow on the fly are organized within a single version graph. In version-control manner, the historical information like process fragments is preserved for reproducing, but the relationship between them is too technical to be employed conveniently for the end-users in a comprehensive perspective.

The ViPen model originated from our previous work in exploratory programming paradigm [1-2, 16], and both data and process fragment are preserved in provenance. Each track is partly stored, and can be built by the knowledge from multiple existing process fragments. On the one hand, it is efficient in storage space and time-saving for users' exploratory service composition. On the other hand, the knowledge like data and process fragments can be reused precisely in a convenient way for service composition on the fly, without loss of business implications.

6. Conclusion and future work

In order to build service composition by reusing not only intermediate data but also process fragments under flexible deviation context at runtime, we put forward ViPen model to support provenance for exploratory service composition. It is beneficial for end-users to build workflows from the knowledge in provenance by the comprehensive operations through a convenient way without loss the strict guarantee of partial ordered relation.

At present, the main significance of ViPen is that it provides reusable intermediate products for late-modeling at runtime. Meanwhile, from the historical information in provenance, the services for next steps of one process can be recommended in data perspective. And our further research is on the process recommendation from the pattern perspective, which means frequent patterns of fragments must be mined from the knowledge in provenance.

7. Acknowledgments

This work is financial supported by the National Natural Science Foundation of China under Grant No. 60970132 and the Beijing Natural Science Foundation under Grant No.4092046.

8. References

[1] S. Yan, Y. Han, J. Wang, et al., "A User-Steering Exploratory Service Composition Approach," in *IEEE International Conference on Services Computing(SCC)*, 2008, pp. 309-316.

[2] S. Yan, "End-User Driven Exploratory Service Orchestration," doctoral dissertation, Institute of Computing Technology of Chinese Academy of Sciences, Graduate University of Chinese Academy of Sciences, Beijing, 2009.

[3] H. Schonenberg, R. Mans, N. Russell, et al., "Process flexibility: A survey of contemporary approaches," *Advances in Enterprise Engineering I*, 2008, vol. 10, pp. 16–30.

[4] S. Sadiq, M. Orlowska and W. Sadiq, "Specification and validation of process constraints for flexible workflows," *Information Systems*, 2005, vol. 30, pp. 349-378.

[5] L. Bavoil, S. Callahan, P. Crossno, et al., "Vistrails: Enabling interactive multiple-view visualizations," 2005, pp. 135-142.

[6] M. Addis, J. Ferris, M. Greenwood, et al., "Experiences with e-Science workflow specification and enactment in bioinformatics," in *Proceedings of UK e-Science All Hands Meeting*, 2003, pp. 459-467.

[7] P. Dadam and M. Reichert, "The ADEPT project: a decade of research and development for robust and flexible process support," *Computer Science-Research and Development*, 2009, vol. 23, pp. 81-97.

[8] M. Pesic, H. Schonenberg and W. van der Aalst, "Declare: Full support for loosely-structured processes," in *EDOC*, 2007, pp. 287–298.

[9] M. Caeiro-Rodriguez, T. Priol and Z. Németh, "Dynamicity in scientific workflows," 2008.

[10] S. B. Davidson and J. Freire, "Provenance and scientific workflows: challenges and opportunities," in *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, Vancouver, Canada, 2008, pp. 1345-1350.

[11] T. Oinn, M. Addis, J. Ferris, et al., "Taverna: a tool for the composition and enactment of bioinformatics workflows," *Bioinformatics*, 2004, vol. 20, pp. 3045-3054.

[12] I. Altintas, O. Barney, Z. Cheng, et al., "Accelerating the scientific exploration process with scientific workflows," *Journal of Physics*, 2006, vol. 46, pp. 468-478.

[13] S. Davidson, S. Cohen-Boulakia, A. Eyal, et al., "Provenance in scientific workflow systems," *IEEE Data Engineering Bulletin*, 2007, vol. 32, pp. 44-50.

[14] S. P. Callahan, J. Freire, E. Santos, et al., "VisTrails: visualization meets data management," in *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, Chicago, IL, USA, 2006, pp. 745-747.

[15] X. Zhao and C. Liu, "Version management in the business process change context," in *Business Process Management*. vol. 4714, ed Heidelberg: Springer Berlin, 2007, pp. 198-213.

[16] J. Wang, Y. Han, S. Yan, et al., "VINCA4Science: A Personal Workflow System for e-Science," in *International Conference on Internet Computing in Science and Engineering*, 2008, pp. 444-451.