一种基于扩展任务结构的工作流实例迁移方法*

周明天+, 王敏毅, 姚绍文

(电子科技大学 计算机科学与工程学院,四川 成都 610054)

A Workflow Instance Migration Approach Based on the Extended-Task-Structures

ZHOU Ming-Tian+, WANG Min-Yi, YAO Shao-Wen

(College of Computer Science and Engineering, University of Electronic Science and Technology of China, Chengdu 610054, China)

+ Corresponding author: Phn: 86-28-83203300, E-mail: mtzhou@uestc.edu.cn

http://www.uestc.edu.cn

Received 2001-09-20; Accepted 2002-08-02

Zhou MT, Wang MY, Yao SW. A workflow instance migration approach based on the extended-task-structures. *Journal of Software*, 2003,14(4):757~763.

Abstract: Workflow instance migration is a typical and important problem in the research of workflow evolution. Underlying the definitions of the extended-task-structures based workflow, some rules, conditions and an algorithm for workflow instance migration are presented, which are suited to generic dynamic changes. By the comparisons with other similar research works, the advantages of the approach in terms of applicability, universality, correctness and practicality are illustrated.

Key words: workflow; workflow evolution; adaptive workflow; dynamic change; instance migration

摘 要: 工作流实例迁移是工作流演化研究中典型而重要的问题.在基于扩展任务结构的工作流定义的基础上,提出了适用于一般动态变化处理的实例迁移规则、条件和算法.与同类研究工作比较,该方法在适用范围、通用性、正确性和可实现能力等方面具有一定的先进性.

关键词: 工作流;工作流演化;自适应工作流;动态变化;实例迁移

中图法分类号: TP311 **文献标识码**: A

Flexible and adaptive workflow is one of main issues in the research of advanced workflow[1,2]. Furthermore,

* Supported by the Science Foundation of Electronics of China under Grant No.51415010101DZ0233 (电子科学基金)

ZHOU Ming-Tian was born in 1939. He is a professor and doctoral supervisor of the College of Computer Science and Engineering, University of Electronic Science and Technology of China. His current research interests include computer networking information system, open distributed processing system, computer system and software, and computer supported collaboration work. **WANG Min-Yi** was born in 1973. He is a Ph.D. candidate at the College of Computer Science and Engineering, University of Electrical Science and Technique of China. His research interests are distributed object technology, computer supported collaboration work, intelligent agents. **YAO Shao-Wen** was born in 1966. He is an associate professor and also a Ph.D. candidate of College of Computer Science and Engineering, University of Electrical Science and Technique of China. His current research is focused on protocol engineering, Web-based knowledge representation and colored Petri-nets (CPN) modeling.

the evolution of workflow instances, i.e. a running workflow instance changing its own schema on the fly, is important for adaptive workflow to support dynamic changes. There are some deficiencies on the applicability, universality and correctness in recently proposed methods[3~9]. In this paper, we discuss and address an important problem, namely instance migration, in the context of workflow evolution. In other words, our focus is how a running instance migrates its old schema to a new one in response to dynamic changes of schema. Based on the formal workflow definitions, we present some rules, conditions and an algorithm for workflow instance migration. Further analysis and comparisons illustrate the advantages of our approach.

1 Workflow Schema and Instance

We adopt a popular workflow model, i.e. Task Structures[3,9], and extend it to support data flow.

Definition 1. A workflow schema, also called workflow model or task model, describes some tasks, data and relationship of dependency and links among them. Formally, it can be denoted as: $W=(V,T,Y,C,E,V_0,V_1,V_d,E_d)$, where: $V=T\cup Y\cup C$ is a set of all nodes in the model; T is a set of tasks; Y is a set of synchronizers, which deal with synchronization of concurrent routing; C is a set of conditional nodes, which make routing decision according to bound conditions; $U=T\cup C$ is defined as a set of executable nodes and $X=T\cup Y$ as a set of terminal nodes; A relation $E\subseteq V\times V$, called control flow, models the sequential order of nodes; $V_0\subseteq U$ is a set of initial executable nodes, and $V_1\subseteq X$ a set of terminal nodes; V_d is a set of data elements in the model; the data flow $E_d\subseteq U\times V_d\times\{read,write\}$ indicates the links between executable nodes and data.

For simplicity, we just investigate the acyclic model and ignore the compensating tasks. We implement a special implied task t_{sys0} for every schema, which is able to initialize the data elements and instantiate the schema.

Similar as instantiation of class to object, the procedure that the engine of WfMS launches a workflow schema is called instantiation too, and a running workflow is called an instance. In the runtime, there are several possible states for a task of an instance, written:

StateOf_W: $W.V\rightarrow$ { *NOT_ACTIVATE*, *ACTIVATED*, *RUNNING*, *COMPLETED* }.

Definition 2. A workflow instance $I_W=(id,W,S_W)$ is the snapshot of a running workflow, reifying an instant status of workflow, where: *id* is the identifier of the instance, owing to a schema may initialize several concurrent instances at the same time; W is the corresponding schema, which is changeable during runtime; S_W is the status of running instance, which is a quintuplet: $S_W=(V_c,T_e,U_s,Y_w,D_v)$, where

V_c is a sequence of executed nodes: $\forall_{v\text{ in }V_c}(v\in V\wedge\text{StateOf}_W(v)=COMPLETED)$;

T_e is a set of scheduled and executing nodes: $\forall_{t\in T_e}(t\in T\wedge\text{StateOf}_W(t)=RUNNING)$;

U_s is a set of activated and ready for scheduling nodes: $\forall_{t\in U_s}(t\in U\wedge\text{StateOf}_W(pred_U(t))=RUNNING)$[2];

Y_w is a set of synchronizers waiting for synchronization:
$\forall_{y\in Y_w}(y\in Y\wedge\exists_{t\in PRED_T(y)}(\text{StateOf}_W(t)=COMPLETED)\wedge\exists_{t\in PRED_T(y)}(\text{StateOf}_W(t)\neq COMPLETED))$[2];

D_v is a set of data elements' value, which is instantiated from V_d.

Some criteria to ensure the consistency of workflow schema are listed as following:

(1) Structure Correctness

i) *The Task Structure of schema should be connected, i.e., there should be a path between any two nodes*

ignoring the direction of control edges: $\forall_{v_1,v_2\in V}\exists_{\bar{p}=(e_1,e_2,\ldots,e_n)}(e_1=(v_1,v_{e_1})\wedge e_n=(v_{e_{n-1}},v_2))$ [3];

ii) *Every node should be on a path from an initial node to a terminating node:*

$\forall_{v\in V}\exists_{v_0\in V_0,v_1\in V_1}(v_0\xrightarrow{\bar{p}_1}v\xrightarrow{\bar{p}_2}v_1)$ [3];

iii) *From any reachable status, it is possible to reach a terminal status:*

$\forall_{S_W}(S_{W,begin}\xrightarrow{*}S_W)\Rightarrow S_W\xrightarrow{*}S_{W,end}$ [4], where: $S_{w,begin}$ and $S_{w,end}$ are initial and terminal state.

(2) Data Correctness

i) *Any data element read by an executable node must be initialized by the system or written by some preceding task:*

$\forall_{d\in V_d}\exists_{t\in U}((t,d,read)\in E_d)\Rightarrow\exists_{t'\in T}((t'<_{\tilde{W}}t)\wedge(t',d,write)\in E_d)\vee(t_{sys0},d,write)\in E_d$ [5];

ii) *The relationship of partial order between two executable nodes, which are conflicted in accessing some data element, is certain:* $\forall_{t,t'\in U}\exists_{d\in V_d}(t\ddagger_{\mathrm{d}}t')\Rightarrow(t<_{\overline{W}}t'\vee t'<_{\overline{W}}t)$ [6].

2 Instance Migration

2.1 Dynamic change

Definition 3. A dynamic change is the evolution of a schema occurring in the runtime of its instances, which transfers the schema from W to W', written: $\delta_W=(W,W')$. To an instance with the status S_W, the change migrates it to the status $S_{W'}$ with the new schema W', written: $\delta_S=(S_W,S_{W'},\delta_W)$ or $S_W\xrightarrow{\delta_W}S_{W'}$.

2.2 State transformation rules

As the above definition, the dynamic change is essentially the valid state transformation between two schemas. Considering different runtime states of a node, we give the following rules:

$\forall_{v\in W'.V}A(v):\mathrm{StateOf}_{W'}(v)=s$

$A(v):v\in W'.(C\cup Y),s=PENDING$

$A(v):v\notin W.V\wedge v\in W'.T,s=NOT\_ACTIVATE$

$A(v):v\in W.T\wedge\mathrm{StateOf}_W(v)=NOT\_ACTIVATE,s=NOT\_ACTIVATE$

$A(v):v\in W.T\wedge\mathrm{StateOf}_W(v)=ACTIVATED,s=NOT\_ACTIVATE$

$A(v):v\in W.T\wedge\mathrm{StateOf}_W(v)=COMPLETED,s=COMPLETED$

$A(v):v\in W.T\wedge\mathrm{StateOf}_W(v)=RUNNING,s=HOLDING$

Specially, we introduce two temporary states, i.e. *HOLDING* and *PENDING*, for the task and non-task nodes respectively.

State Transformation Rules specify how an instance migrates its status with old schema to the status of new schema. However, the rules don't guarantee the availability of migration. So we will present the conditions on migratable instance.

2.3 Migratable conditions of instance

2.3.1 Migratable state conditions (MSC)

Given a dynamic change δ_W arising in the runtime of an instance I_W, we propose three conditions to verify the correctness of state transformation according to the above rules, described as following:

$$\forall_{t\in W.T}(\mathrm{StateOf}_W(t)=COMPLETED)\Rightarrow$$
$$t\in W'.T\wedge \qquad (MS-i)$$
$$\neg\exists_{t'\in W'.T}((StateOf_{W'}(t')\neq COMPLETED)\wedge(t'<_{\tilde{W}'}t))\wedge \qquad (MS-ii)$$
$$\exists_{\tilde{p},t_0}(t_0\in W'.V_0\wedge t_0\xrightarrow{\tilde{p}}t) \qquad (MS-iii)$$

Theorem. The state transformation $\delta_S=(S_W,S_{W'},\delta_W)$ is correct *iff* the instance migration satisfies the MSC.

Proof. It is obvious that the MSC is necessary to a correct state transformation. So we just prove the sufficiency of the conditions.

Actually, we need only prove that $S_{W'}$ is reachable if the change satisfies the MSC. Therefore we construct a virtual enactment to reach $S_{W'}$. The enactment adopts the same scheduling strategy as the workflow engine, but does not really launch the tasks. Any concurrent routing running in the enactment will stop scheduling the tasks when meeting either the task with the state *NOT_ACTIVATE*, *HOLDING*, or the terminal node.

We suppose that there be a completed task $t_n\in W'.T$ without being "scheduled" after the virtual enactment is over. According to the condition (*MS–iii*), we can find a path $\tilde{p}$ from some initial item t_0 to t_n. Considering the procedure of the enactment, it's deducible that the preceding node t_{n-1} of t_n on the path must not be enacted. From the condition (*MS–ii*): $\mathrm{StateOf}_{W'}(t_{n-1})=COMPLETED$. Similarly, we trace along the path $\tilde{p}$ back to t_0, with $\mathrm{StateOf}_{W'}(t_0)=COMPLETED$ and t_0 is not enacted. Obviously, it is impossible. Thus the hypothesis just made is wrong.

As a result, we say that the state $S_{W'}$ is reachable and the sufficiency of the conditions is proven. □

2.3.2 Migratable data conditions (MDC)

The MSC take into account the control flow and state of task, next we will give the migratable conditions relating to the data flow.

While the correctness criteria of schema and MSC of instance are satisfied, it is allowable either to add or to remove data links, without breaking the consistency of instance state.

First, we review the data correctness condition ii) deeply. Obviously, swapping the order of completed tasks may lead to inconsistency of data results, although such kind of structural change satisfies the correctness criteria of schema and MSC. As a result, the migration should satisfy the following condition to keep the data flow consistent:

$$\neg\exists_{t_1,t_2\in W.T,W'.T}((\mathrm{StateOf}_{W,W'}(t_1)=\mathrm{StateOf}_{W,W'}(t_2)=COMPLETED)\wedge(t_1\ddagger_d t_2)\wedge(t_1<_{\overline{W}}t_2\wedge t_2<_{\overline{W}'}t_1)) \qquad (MD-i)$$

Second, the data elements can influence the execution of the tasks, as well as the routing of instance enactment, in other words, the choice of conditional nodes. A prerequisite to the successful migration is that any referenced data element of the new added conditional nodes is determinable in the instance of new schema, formalized as:

$$\forall_c\forall_d(c\in W'.C\wedge c\notin W.C\wedge(c,d,read)\in W'.E_d)\Rightarrow$$
$$\neg\exists_{t\in W.T}(\mathrm{StateOf}_W(t)=COMPLETED\wedge(t,d,write)\in W'.E_d)\wedge c<_{\tilde{W}'}t) \qquad (MD-ii)$$

Oppositely, the virtual enactment constructed in the migration will be ambiguous when meeting a conditional node not satisfying the condition (*MD–ii*).

Similarly as the MSC, the sufficiency of the MDC can be proven by constructing a virtual enactment. Limited by the text space, further detail is omitted here.

2.4 Instance migration algorithm

We apply the MSC and MDC to the implementation of our approach, described as the following algorithm:

Algorithm: Instance Migration

Description: A dynamic change of schema $\delta_W=(W,W')$ arises when the running instance I_W of the schema W holds the state S_W. The instance need migrate to $I_{W'}$ based on the new schema W'.

Procedure:

1. Initialize some elements of $I_{W'}$ and temporary data of the algorithm:

1.1 Initialize $I_{W'}$: $\forall_{t\in W'.T}:\text{StateOf}_{W'}(t)\leftarrow NOT\_ACTIVATE$; $I_{W'}.S_{W'}.V_c\leftarrow\phi$;

$\forall_{v\in W'.(C\cup Y)}:\text{StateOf}_{W'}(v)\leftarrow PENDING$;

1.2 Migrate the data elements of I_W to $I_{W'}$: $I_{W'}.S_{W'}.D_v\leftarrow I_W.S_W.D_v$;

1.3 $\forall_t(t\in I_W.S_W.T_e\wedge t\in W'.T):\text{StateOf}_{W'}(t)\leftarrow HOLDING; H\leftarrow H\cup\{t\}$;

1.4 Create a temporary task set:

if (ClassOf($I_W.S_W.V_c[i]$)=*TASK*) then $M\leftarrow I_W.S_W.V_c[i]$, i = 0, Length($I_W.S_W.V_c$) [1];

2. Mark the states of some tasks in $I_{W'}$ according to M, $\forall_{t\in M}$:

if ($t\in W'.T$) then StateOf$_{W'}(t)\leftarrow COMPLETED$; else migration not allowable, algorithm halts.

3. Reach the new state $S_{W'}$ in term of the rules of virtual enactment:

3.1 Select the node $u\in U$ from W', which satisfies: StateOf$_{W'}(u)$=*COMPLETED* or *PENDING*, if such node doesn't exist, go to 4;

3.2 if (ClassOf(u)=*CONDITION*) then

$\forall_d((u,d,read)\in W'.V_d)$: Mark(d,$read_u$); decide a routing; go to 3.1 to continue scheduling;

3.3 if ($\exists_d$($(u,d,write)\in W'.V_d\wedge$Marked($d$,$read_{u'}$))), then migration not allowable, algorithm halts.

3.4 $\forall_{t'}(t'$ **in** $I_{W'}.V_c\wedge u\ddagger t')$: if ($u<_{\overline{W}}t'$), then migration not allowable, algorithm halts.

3.5 AddTail($I_{W'}.V_c$, u), $M\leftarrow M\backslash\{u\}$, go to 3.1 to continue scheduling;

4. if ($M\neq\varnothing$) then migration not allowable, algorithm halts.

5. $\forall_{t\in H}$: if (StateOf$_{W'}$ (pred$_U(t)$) = *COMPLETED*) then

StateOf$_{W'}$(t)$\leftarrow$*RUNNING*; $I_{W'}.S_{W'}.T_e\leftarrow I_{W'}.S_{W'}.T_e\cup\{t\}$;

else StateOf $_{W'}$(t)$\leftarrow$*NOT_ACTIVATE*;

6. Migration successful, end of algorithm.

The implementation of the algorithm is driven from the proof in 2.3. In other words, we construct a virtual enactment, where the MSC and MDC can be verified in turn. In more detail, step 2 checks the *MS–i*; the enactment of new schema W' with the selection in step 3.1 follows the *MS–ii* implicitly, because every preceding node t' of u in $I_{W'}.V_c$ must satisfy: StateOf$_{W'}(t)$=COMPLETED; in the step 3.3, the *MD–ii* is checked implicitly, i.e., we can not find a data element written by u, which has been marked; since in step 3.4 $t'<_{\overline{W'}}u$, so if ($u<_{\overline{W}}t'$) the *MD–i* is not satisfied and the algorithm should halt; in the step 4, ($M\neq\varnothing$) shows that the virtual enactment can't cover all the nodes with COMPLETED state in I_W and the *MS–iii* is not satisfied.

3 Related Work

Similar as our work, some approaches are proposed in Refs.[4~8] to solve the problems within workflow evolution. In detail, these approaches include: some modification primitives for workflow evolution in Refs.[4,5], an approach based on generic workflow models in Ref.[6], a formal representation of dynamic change based on Petri-Nets in Ref.[8] and a few correctness conditions used to adapt the task model of instance in Refs.[7,8], and so on. compared with these works, our approach has some advantages:

1) Since it's familiar for the tasks in a workflow to interact with each other through some data, our approach supports not only the change of structure/control flow, but also the change of data flow. It's more practical in applications than the methods only considering the structural change, such as Refs.[6,7].

2) Our approach supports generic dynamic changes of workflow, without putting special limitation on the workflow evolution. Some typical structural changes involved in other articles such as: four inheritance types *PT*, *PP*, *PJ*, *PJ*3 in Ref.[6], upsizing and downsizing property of dynamic change in Ref.[7], specific how the changes look like. However, our approach doesn't limit the semantics of changes. As a result, it's general and independent to the concrete

application.

3) Essentially, we regard dynamic change as direct transformation of schema and instance state. Instance migration in our work is based on neither the modification primitives in Refs.[4,5], nor some transitional model/state like [6]. To some extent, the dynamic change shown by us is similar to the atomic transaction in database. Regarding a single operation as a unit of change or introducing some transitional state will easily lead to loss of instance state in migration and even failure of migration.

4) Our approach has broad applicability to kinds of changes. Contrastively, the conditions in Ref.[8] are too strict to evolve some migratable instances; and it's possible for inconsistent migration to be permissive in Ref.[7], because the influence of data flow and tasks on workflow enactment is ignored.

5) The rules and conditions we propose are easy to be validated and implemented. For instance, the conditions of migration in Refs.[4,7] are difficult to be validated; and although the minimal representative defined in Ref.[6] is helpful to transfer the instance state in an idea scenario, it's quite difficult for any workflow to find a proper minimal representative.

On the other hand, the self-learning algorithm for automatic process definition generation in Wowww![10] is also used to improve the flexibility of online process, and the workflow model is alterable. However, the essence and focus of the work are different from ours. The process definitions (i.e., workflow schema in this paper) in Wowww! evolve gradually and become clear in the runtime by gathering information from users. Differently, our approach is used to migrate a running instance from an old schema to a new one, and the schema is always certain before instantiation.

As an adjunctive application of the ROK (Reflective Object Knowledge) model, an approach applied to workflow model and instance evolution is shown in Ref.[3]. Some operations on meta-model of process and some meta-processes made of the operations are introduced in the approach. To some extent it is similar to the modification primitives and their combination. Some issues such as correctness, migration conditions are not involved in the work.

4 Conclusions

On the basis of the formal definitions of the extended-task-structures based workflow model, we present a complete approach to tackle the typical problem in the context of workflow evolution, i.e., the dynamic migration of workflow instance. Our methodology consists of a few state transformation rules, several migratable conditions of instance and an instance migration algorithm. By comparing with some related works, we illustrate the advantages of our approach on applicability, generality, correctness and practicability. The formal definitions and algorithm in the paper have been successfully applied to a flexible WfMS we developed, which supports dynamic changes of workflow. Some issues ignored in the paper, such as: the compensating tasks, the cyclic structure, and the partition of change regions, are our current research focus.

References:

[1] Amit PS, W.M.P. van der Aalst, *et al*. Processes driving the networked economy. IEEE Concurrency, 1999,7(3):18~31.

[2] Shi ML, Yang GX, Xiang Y, Wu SG. WfMS: workflow management system. Chinese Journal of Computers, 1999,22(3):325~334 (in Chinese with English abstract).

[3] David Edmond, A.H.M. ter Hofstede. A reflective infrastructure for workflow adaptability. Data and Knowledge Engineering, 2000, 34(3):271~304.

[4] Casati F, Ceri S, *et al*. Workflow evolution. Data and Knowledge Engineering, 1998,24(3):211~238.

[5] Reichert M, Dadam P. ADEPTflex: supporting dynamic changes of workflows without loosing control. Journal of Intelligent Information Systems, 1998,10(2):93~129.

[6] W.M.P. van der Aalst. How to handle dynamic change and capture management information. In: Proceedings of the 4th IFCIS International Conference on Cooperative Information Systems (CoopIS99). Edinburgh, Scotland, IEEE Computer Science Press, 1999

[7] Ellis C, Keddara K, Rozenberg G. Dynamic change within workflow systems. In: Comstock N, Ellis C, eds. Proceedings of the Conference on Organizational Computing Systems. Milpitas, CA: ACM SIGOIS, ACM Press, 1995. 10~21.

[8] Mathias W. Formal foundation and conceptual design of dynamic adaptations in a workflow management system. In: Sprague ed. Proceedings of the 34th Hawaii International Conference on System Sciences (HICSS-34). Maui, Hawaii: IEEE Computer Society, 2001.

[9] W.M.P. van der Aalst, A.H.M. ter Hofstede. Verification of workflow task structures: a Petri-net-based approach. Information System, 2000,25(1):43~69.

[10] Shi ML, Yang GX, Xiang Y. A Web-based workflow management system. Journal of Software, 1999,10(11):1148~1155 (in Chinese with English abstract).

附中文参考文献:

[2] 史美林,杨光信,向勇,伍尚广.WfMS:工作流管理系统.计算机学报,1999,22(3):325~334.

[10] 史美林,杨光信,向勇.一种基于 Web 的工作流管理系统.软件学报,1999,10(11):1148~1155.

Appendix:

Some definitions, operators and functions directly used in the text.

1. A function ClassOf: $V \rightarrow \{TASK, CONDITION, SYCHRONIZER\}$ judging the type of a node.
2. Four preceding and succeeding functions of a node: $\mathrm{PRED}_Y(y)$, $y \in Y$, a set of all directly preceding nodes of a synchronizer y; $\mathrm{SUCC}_X(x)$, $x \in T \cup Y$, a set of all directly succeeding nodes of a node x; $\mathrm{succ}_C(c)$, $c \in C$, the directly runtime succeeding node of a conditional node c; $\mathrm{pred}_U(t)$, $t \in U$, the directly runtime preceding node of an executable node t.
3. Path in workflow: a static path $\overline{p} = (e_1, e_2, ..., e_n)$ in W, where: $e_k=(v_{k,i},v_{k,j})$, $v_{k,i}$ is the end point of e_{k-1} and $v_{k,j}$ is the starting point of e_{k+1}; the dynamic path $\tilde{p} = (e_1, e_2, ..., e_n)$ in I_W, an additional condition is that if $\mathrm{ClassOf}(v_i)=CONDITION$ then $v_j = \mathrm{succ}_C(v_i)$. A path p from v_1 to v_2 can be denoted as: $v_1 \xrightarrow{p} v_2$.
4. $S_W \xrightarrow{\sigma} S'_W, \sigma = v_1 v_2 ... v_n$: there is a sequence σ of scheduled nodes which leads from the state S_W to the state $S_{W'}$, $S_W \xrightarrow{*} S'_W$: the state $S_{W'}$ is reachable from the state S_W.
5. A static partial order $(V, <_{\overline{W}})$ is defined as: $t_1 <_{\overline{W}} t_2$ *iff* there is a static path $\overline{p}$ where $t_1 \xrightarrow{\overline{p}} t_2$. Obviously the partial order defines an irreflexive and transitive binary relation $<_{\overline{W}}$. Similarly we can define the dynamic partial order $(V, <_{\tilde{W}})$.
6. Generally, if two executable nodes $v_1, v_2 \in U$ in W satisfy the following condition: $\exists_{d \in V_d}((v_1, d, write) \in E_d \wedge ((v_2, d, write) \in E_d \vee (v_2, d, read) \in E_d))$, then we call that v_1 and v_2 are conflicting to data d, written $v_1 \ddagger_d v_2$. If unnecessarily, we can omit d, written $v_1 \ddagger v_2$.