Distributed Workflow and View Definition Languages

Nico Kerschbaumer

Department of Informatics-Systems, University of Klagenfurt,
A-9020 Klagenfurt, Austria,
{eder,nico}@isys.uni-klu.ac.at,
WWW: http://www.uni-klu.ac.at/tewi/inf/isys/

1 Introduction

Interorganizational cooperations between geographically distributed business partners are nowadays crucial to stay competitive in a very fast paced and changing market. A lot of research has been conducted in the recent years to develop approaches to support communication with SOA-based protocols between business partners. Choregraphies are used to define the communication protocol between interacting business partners. Such protocols are fully decentralized, that means there is no one that runs this multi partner spanning interorganizational workflow. Still, cooperation between maybe competitive companies is a very sensitive matter, because on one hand they have to work together to fullfil a common business goal that is profitable for both, while on the other hand they have to keep business secrets private to stay competitive. A very promising solution that allows a good balance between openness which is needed for cooperation and privacy to protect business secrets is the concept of workflow views. A workflow view is a workflow itself which contains parts of an internal workflow and is created by applying operations onto the original workflow like for example hiding of certain activity steps or aggregating a set of steps to a new step. All partners that need to cooperate provide views on their private internal processes and then solely communicate through their public views [1, 2]. In this report we introduce two languages that allow the specification of interorganizational workflows that communicate through views:

Distributed Workflow Definition Language(DWDL) : The DWDL allows the specification of workflows and is described in detail in section 2.

Workflow View Definition Language(WfVDL) The WfVDL allows to specify view generation operations based on a DWDL document and is discussed in detail in section 3.

The languages are based on the Workflow Definition Language(WDL) which was introduced in a prototypical workflow management system called *Phanta Rei* [3]. The prototype was further developed and released as a commercial WfMS called *@Enterprise* which also supports the specification of workflows with WDL [4]. Our languages follow the restrictions of full blocked workflows. This means

every split is followed by a corresponding join. Full-Blocked Workflows are less expressive than par-blocked or non-blocked workflows. For example they do not allow jumps. There are cases where a workflow can not be expressed using a full-blocked definition. In [5] it could be shown that such workflows that can not be translated to a full-blocked form get invalid as soon as data-flow comes into play, because of unforeseeable concurrency problems. This means for our approach that is used for control-flow and data-flow full-blocked workflows are a good choice with the right expressiveness. A workflow is a DAG $G = (\mathbb{S}, \mathbb{T}, \mathbb{A}, \mathbb{V}, \mathbb{R}, \mathbb{D})$ composed of a set of steps $\mathbb{S}$, transitions $\mathbb{T}$, activities $\mathbb{A}$, variables $\mathbb{V}$, roles $\mathbb{R}$, and datatypes $\mathbb{D}$. Activities are reusable entities that are executed in order to execute an activity-step. Variables are visible for the corresponding workflow instance. Steps can access data from variables by using in, out or inout parameters that are defined in the correspondincg activities. Each variable has a type that is defined as an XML-Type in some XML-Schema document. Roles can be either internal or external. Internal roles are used to define which role may start a specific step, while external roles are used to indicate who should receive data and from whom data should be received. Data that is exchanged between workflows is realized by sending and receiving steps. These special steps are executed by the system and can send a set of variables to a remote party (sending step) or receive data from a remote party (receiving step). A receiving step can only have out data (This means data is written to the variables of the workflow instance), a sending step can only have input variables since all data is read from the workflow and send as it is.

2 Distributed Workflow Definition Language(DWDL)

Workflows are defined by a custom distributed workflow definition language (DWDL) that is based on full-blocked workflows and asynchronous communication patterns for remote calls. The language is restricted to the main concepts that are needed to define distributed workflows with regard to a good readability. We expect that mappings to standard workflow languages are achievable with reasonable effort. The language contains the following concepts: Workflow parameters, variable declarations, activity declarations, interfaces, transformations, tickets, labels, local steps, receiving steps, calling steps, control steps and roles / actors. Listing 1.1 shows the DWDL in EBNF notation. A DWDL document is similar in structure to an ordinary procedural program and contains the following parts:

Workflow Header Every workflow has a name and a set of optional arguments. Arguments are stored data variables which are passed into the workflow.

Declaration Part The declaration part consists of three units: name space imports, data declaration and workflow declaration. The first unit allows one to import XML-schema namespaces which provide the possible data types for variables. In the data declaration part variables which are used in the workflow have to be declared. A variable can be of any XML-type included in a imported namespace. The workflow declaration part serves for

the specification of the process information and contains used activities with its corresponding parameters and data types.

Workflow Body The body is the main part of a workflow specification. Between the keywords *begin* and *end* the control and the data flow of the process are defined.

The following paragraphs describe the main concepts of the DWDL followed by an EBNF representation in listing .

Workflow A workflow definition in DWDL consists of a header and a body. The header includes namespace imports of XML schemas, variable declarations based on the XML types of the imported schemata and activity definitions which can be reused in any step. The workflow body contains a number of steps.

Namespace Imports A namespace imports an internal or external XML schema which includes data types and roles.
Example: `Namespace sns=http://someurl.com/aDoc.xsd`

Variables Variables are used to store data and can be of any XML type. A variable is declared with the keyword `VAR`, the name of the variable and the namespace followed by a ":" and the XML-Type of the variable.
Example: `VAR $myVariable sns:BigDecimal`

Activities An activity declaration is used to declare the input and output data of an activity and also to implement the activity in the body of the declaration. The declaration contains the name of the activity and their parameters. A parameter can be `in`, *out* or `inout`. Whereas *in* means that the parameter is read by the activity but not changed, *out* means the parameter is not read but the activity but written and `inout` means the variable is read and after some processing written back to the workflow instance. Example: `Activity PrepareOrder(in sns:OrderDocument) ...`

Steps We support three different kinds of steps. Control steps , activity steps (every step that uses some activity) and communication steps that can be either send or receive. Control and activity steps are executed by internal roles while communication steps are executed directly by system.

1. Control Step: A control step is a XOR-split, XOR-join, PAR-split or PAR-join, i.e. conditional- and parallel splits.
2. Choice Step: Is a deferred choice as described in the workflow patterns [6].
3. Activity Step: An activity step is an internal step that uses a predefined activity which can be reused and defines the formal parameters.
4. Communication Step: Communication steps are either of type *sending* or *receiving* and are used to communicate with external partners. A sending steps sends data to the partner that has the external role and a receiving step receives data from an external partner with a certain role.

Roles We differ between internal roles and external roles. Internal steps are executed by internal roles while our communication steps use external roles for the interorganizational communication between business partners. An external role is used in a send step, i.e. we transfer data to an external partner as a role that can take part in workflow or in a receive step in which we expect data from a certain role .

Formal Parameters Activities define the formal parameters which consist of a type based on a namespace import and a mode(in,out,inout).

Actual Parameter Steps have actual parameters which are filled with variables.

Listing 1.1. EBNF of the DWDL

```
Workflow :=       "Workflow" NAME "(" ARG{"," ARG} ")"
"{"
                  {IMPORTS}          {VARDEF}          {TICKETDEF}
                  {ROLEDEF}          {IFDEF}           {ACTDEF}
                  {WFDEF}
"}"
ARG = VAR_NAME;

IMPORTS = "NAMESPACE" NAME "=" XSD;

VARDEF = "VAR" VAR_NAME VAR_TYPE;
VAR_NAME = "\$"NAME;

TICKETDEF = "CORRELATION" TICKET_NAME VAR_TYPE;
TICKET_NAME = "*"NAME;

ROLEDEF = INTR|EXTR "ROLE" ROLE_NAME ROLE_TYPE;
INTR = "intR:";
EXTR = "extR:";
ROLE_NAME = NAME;

IFDEF = "INTERFACE" IF_MODE PROTOCOL IF_NAME

ACTDEF = "ACTIVITY" ACT_NAME "(" [FORM_PARAM{"," FORM_PARAM}] ")";

FORM_PARAM = MODE VAR_TYPE;

MODE = "IN"|"OUT"|"INOUT";

IF_MODE = "SEND"|RECEIVE"

WFDEF = { STEPS };

STEPS = {NORMALSTEP | CONDITIONAL}

NORMALSTEP = (LABEL":" INTERNAL|RECSTEP|CALLSTEP);

INTERNAL =        "INTERNAL" ROLE_NAME ACT_NAME "(" { VAR_NAME} |
VAR_NAME {"," VAR_NAME} ")";

RECSTEP ="RECEIVE" INTRROLE_NAME ACT_NAME "("
                   ACTUAL_PARAM "/" VAR_NAME {"," ACTUAL_PARAM "/" VAR_NAME}
                   "FROM" EXTRROLE_NAME "RECEIVE VIA" IF_NAME
                   {"REQUIRE" TICKET_NAME "==" TICKET_NAME} ;

CALLSTEP = "SEND" ACT_NAME "("
                    VAR_NAME {"," VAR_NAME} ")"
                    "TO" EXTRROLE_NAME "SEND VIA" IF_NAME
           ["CORRELATE WITH" TICKET_NAME [INSTANTIATE AS TransformationEXPR];

CONDITIONAL = XOR | PAR;

XOR = "IF" "(" XOR-COND ")" "THEN" {STEPS} "ELSE" {STEPS} "ENDIF";
PAR =    "ANDSPLIT" { "BRANCH" BRANCH_NAME {STEPS} } "ANDJOIN";
CHOICE = "CHOICE"   { "BRANCH" BRANCH_NAME {STEPS} } "ENDCHOICE";
ACT_NAME = VAR_TYPE = ROLE_TYPE = LABEL = ACTUAL_PARAM = NAME = SIGN{SIGN};
SIGN = "A..Z" | "a..z" | "0..9" | "-";
```

2.1 Control Flow Specification

As explained before, activities can be reused in workflow steps which are then executed from the workflow engine. DWDL offers a variety of control structures to specify the order of execution of steps. In the following we describe the supported control flow constructs and explain which combination of constructs are forbidden in a distributed workflow.

Figure 1 shows a sequential execution order of workflow steps and is specified by simply listing the corresponding step one after another, each separated by a semicolon.

```
InternalRole1 Task1;
InternalRole2 Task2;
```

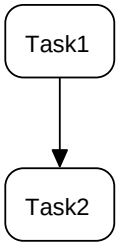

Fig. 1. Workflow Sequence

Figure 2 shows an exclusive choice, i.e. the well known if-then-else construct. The evaluated condition is a boolean expression based on variables used in the workflow. To fulfil the full-blocked structure every split, i.e. *IF* has to be associated with its corresponding join, i.e. *ENDIF*.

```
InternalRole1 Task1;
IF(<condition>)
THEN
    InternalRole2 Task2;
ELSE
    InternalRole2 Task3;
ENDIF
InternalRole3 Task4;
```

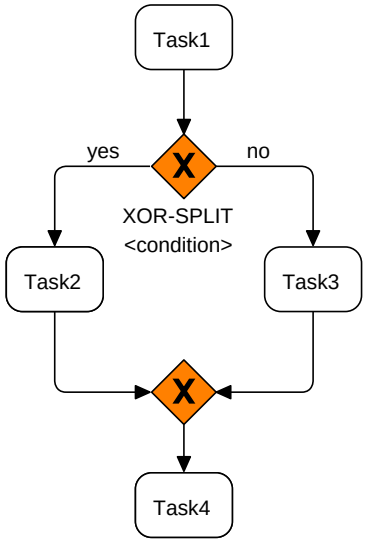

Fig. 2. Workflow with XOR-Split

Figure 3 shows an and-split node which can help to reduce the execution time of a workflow by performing workflow steps in parallel. After the split node steps are executed in parallel for all appearing branches. The and-join node synchronizes all pathes, that means that the execution of the workflow can only continue after all parallel branches have finished.

```
InternalRole1 Task1;
ANDSPLIT
    BRANCH A
     InternalRole2 Task2;
    BRANCH B
     InternalRole2 Task3;
ANDJOIN
InternalRole3 Task4;
```

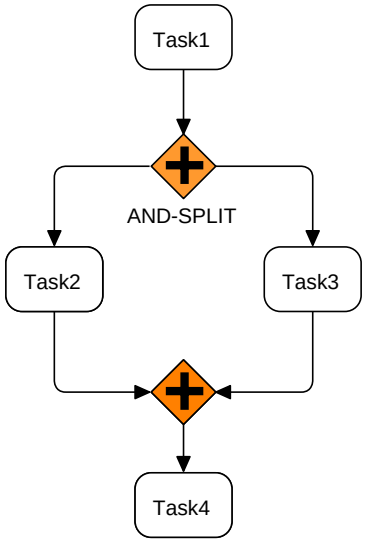

Fig. 3. Workflow with AND-Split

Figure 4 shows a choice node which has the following semantics and constraints. The choice constructs waits for a message of an external business partner and executes a path depending to which step the message is send. To support this behavior each branch following a choice node needs to have a receive step before other steps can occur. There is no synchronization needed because only a single path will be picked and executed.

```
InternalRole1 Task1;
CHOICE
    BRANCH A
     InternalRole2 Task2;
    BRANCH B
     InternalRole2 Task3;
ENDCHOICE
InternalRole3 Task4;
```

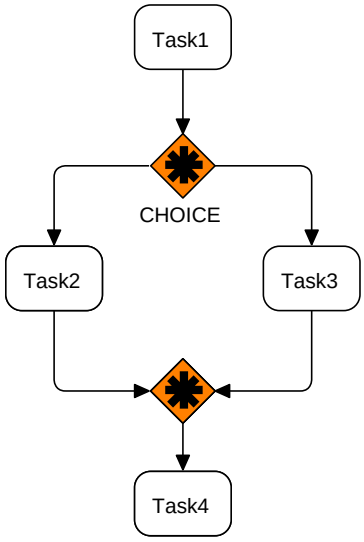

Fig. 4. Workflow with Choice

Figure 5 shows how send and receive activities between two interacting business partners are defined.

```
InternalRole1 Task1;
SEND Task2 to PartnerB;
RECEIVE InternalRole2 Task3
from PartnerB;
InternalRole3 Task4;
```

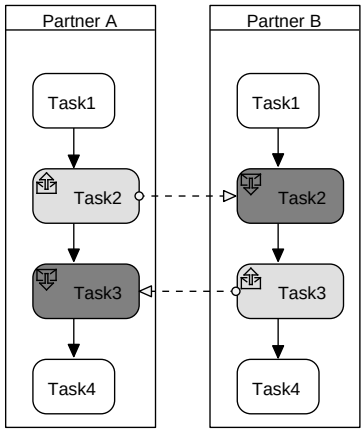

Fig. 5. Workflows with Send and Receive Activities

Besides the constraint of a full-blocked workflow model we also need to consider a problem that can occur in an interorganizational scenario. If we take a look at the example of Figure 6, we see two interacting business with an exclusive choice in their processes. The path which partner B must pick depends on the decision in the of partner A, therefore we forbid such interactions because deadlocks can occur very easy and frequently.

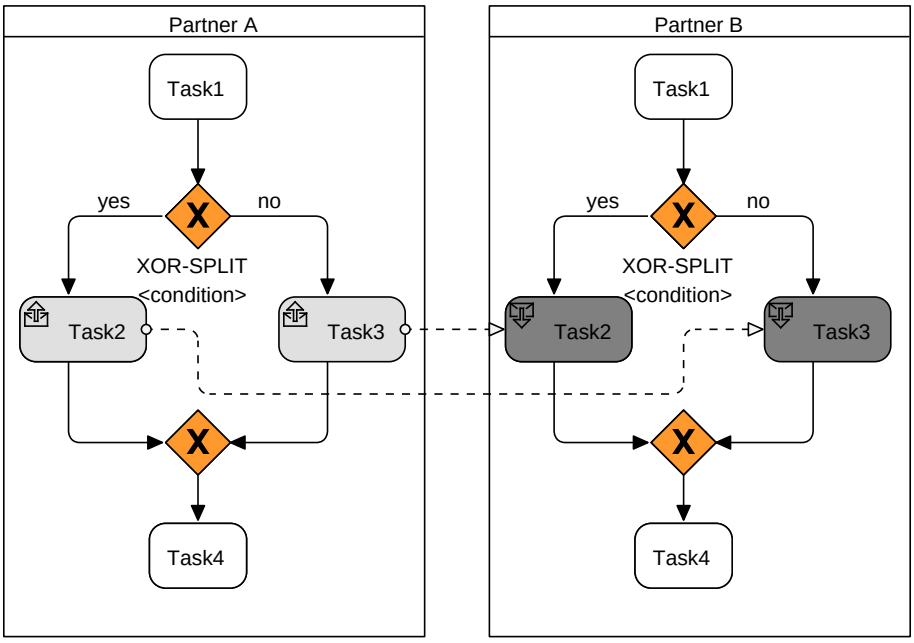

Fig. 6. Choreography with two XOR-Splits

To overcome the problem shown in Figure 6 and still be able to model such processes we utilize a solution presented in the BPMN 2.0 specification [7]. If the initiating partner has an exclusive choice which influences what message is send to which step of an external partner, then the receiving end must model their corresponding process with a choice construct as seen in Figure 7. When we use a choice the process will wait simultaneously in both receiving steps and continue the path that receives the first message while discarding the other path.

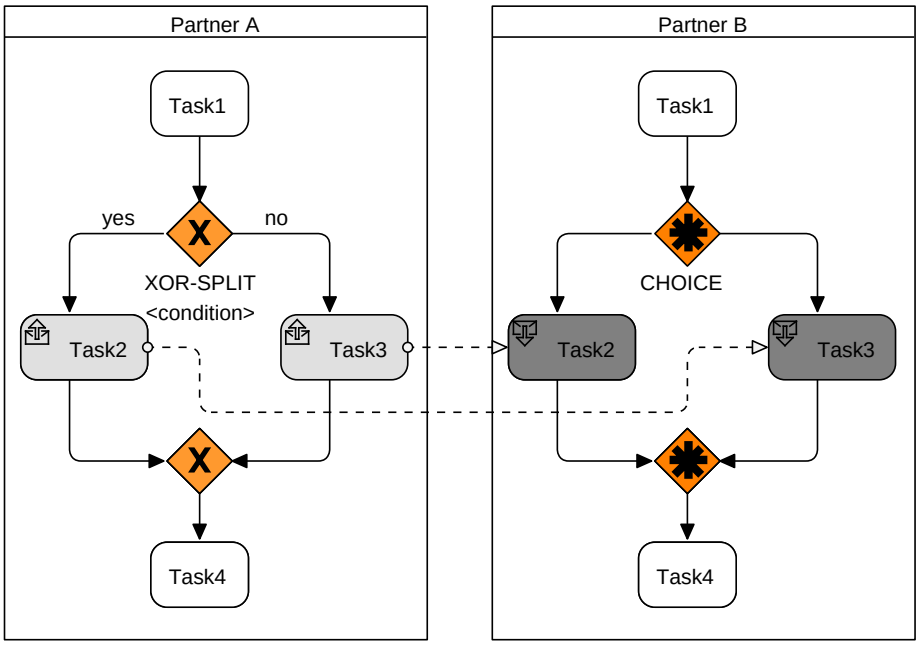

Fig. 7. Choreography with XOR-Split and Choice

Example DWDL Definition A workflow can be expressed in our Workflow definition language. We will show the primary concepts of the language in a small example (see listing 1.2). The example document represents a seller orchestration. In the example listing 1.2 the header includes a namespace import followed by variable declarations based on types of the imported schema. The first step in line 15 is a communication step that receives some order document from some remote party that holds the *Buyer* role. Afterwards an activity step is performed by some internal role called *StockEmployee*. The step is based on the activity CheckStock(...). As declared in the activity definition in line 11. It reads some order document and writes a stock report. Every step in the workflow body needs some label to address the step via a view. This label can be omitted if a referenced Activity is used only once in the workflow. In this case the labelname is equal to the activityname.

Listing 1.2. DWDL definition of the Seller Orchestration

```
1  #Workflow Header
2  Namespace sns = http://example.com/businessdocument.xsd
3
4  #Variable definitions
5  VAR $OrderDocument sns:OrderDocument
6  VAR $OrderConfrimation sns:OrderConfirmation
7  VAR $PickupReply sns:PickupReplyMessage
8  VAR $DeliveryReply sns:DeliveryReplyMessage
9
10 #Activity Declarations
11 Activity CheckStock(in sns:OrderDocument, out sns:StockReport) {...}
12 Activity PrepareOrder(in sns:OrderDocument) {...}
13
14 #Workflow Body
```

```
15 RECEIVE ReceiveOrder($OrderDocument) from BuyerRole
16 StockEmployee CheckStock($OrderDocument, $OrderConfrimation)
17 SEND SendConfimration($OrderConfirmation,$OrderDocument) to BuyerRole
18 StockEmployee PrepareOrder($OrderConfrimation)
19 SEND RequestPackagePickup($OrderConfrimation) to ShipperRole
20 RECEIVE ReceivePickupConfirmation($PickupReply) from ShipperRole
21 RECEIVE ReceiveDeliveryConfirmation($DeliveryReply) from ShipperRole
```

3 Workflow View Definition Language(WfVDL)

The view definition language is used to create views on Workflows. A view is always created for one remote role. A view can contain the whole or a subset of steps of the original workflow. The view can:

- Expose steps that should be included in the view
- Rename steps
- Rename variables
- Rename roles
- Define transformations for send and receive steps

This means the view can expose only wanted aspects of the workflow. In addition the view can realize interoperability between different partners by allowing the definition of transformations on the view level. The expose statement can be applied on all steps. If an expose statement is used for a non-communication step this is for informational purposes only. The exposition of communication steps allows the definition of interfaces.

Workflow View Definition Document A WfVDL document is based on a workflow specified in a DWDL document and created for a certain role. It consists of a header that is similar to that of a DWDL document and a body. The body contains a series of operation which define how the view will be generated.
Example: `Create View aView.vdl based on aWorkflow.dwdl for Seller {...}`

Rename Operation The rename operation allows us to rename steps and roles for the generated view.

Expose Operation The expose operation allows the exposing of internal workflow steps of a DWDL document. Partners are only allowed to call exposed communication steps, however it is possible to include other internal steps for information purposes into the view which allows partners to track the status of the interorganizational process. Furthermore we support data transformation in an expose statement which can transform the internal representation of data to a form an external can process and vice versa. Other methods supported in an expose statement are map and aggregate, the first one is used to map the internal parameters to external view parameters and the latter one allows the aggregation of steps.

Listing 1.3. EBNF of the WfVDL

```
VIEW := VIEW_HEADER
                    "Create View" (VIEW_ID|URI) ["name=" VIEW_NAME]
                    "based on" WF_ID
                    "for" EXTERNAL_ROLE VIEW_BODY ;

VIEW_HEADER := {{RENAMES}{NAMESPACE_IMPORTS}{IF_DECLARATION}};
RENAMES := {ROLE_RENAMES|VAR_RENAMES};
NAMESPACE_IMPORTS := "import" {ABBREV ":" URI ";"};
ROLE_RENAME := "rename role" INTERNAL_ROLE "to" INTERNAL_ROLE;
VAR_RENAMES := "rename var" VARIABLE "to" VARIABLE;

VARIABLE="$"VARNAME;

VIEW_BODY := "{" {STATEMENT ";"} "}";

STATEMENT := "Expose" STEP_LABEL
                              ["as" EXTERNAL_NAME]
                              [EXPOSE_BODY]

EXPOSE_BODY := "{" {EXP_OPERATION} "}";

EXP_OPERATION := {MAP_STATEMENT}{CHANGE_STATEMENT};

MAP_STATEMENT : = "map" (SEND_MAP|REC_MAP)
REC_MAP := VDLPOS "to" WDLPOS | TRANSFORM_REC to WDLPOS
SEND_MAP := WDLPOS "to" VDLPOS | TRANSFORM_SEND to VDLPOS

WDLPOS := "WDL" NUMBER
VDLPOS := "VDL" NUMBER

TRANSFORM_SEND := "transform("WDLPOS {"," WDLPOS } "," EXPR "," XML_TYPE)

TRANSFORM_REC := "transform("VDLPOS{"," VDLPOS} "," EXPR "," XML_TYPE)
CHANGE_STATEMENT := {"change" IF_NAME "to" IF_NAME| "change" ROLE_NAME "to" ROLE_NAME}

EXPR := XPath|XQuery|XSLT|URI;

VIEW_NAME := ORCHESTRATION_LABEL := PARAMETER_NAME := STEP_LABEL := EXTERNAL_NAME
:= ROLE_NAME := INTERFACE_NAME := ABBREV := WORKFLOW_NAME := WF_ID := SIGN {SIGN};

SIGN = "A..Z" | "a..z" | "0..9" | "_";
```

Aggregate Operation We can aggregate a set of internal steps and represent them in the view as a single step to hide private process details or to keep the global process clean of unnecessary information. An aggregate can include internal steps as well as communication steps with the following restriction. A communication step may only be part of an aggregate if the role for which the view is generated is neither part of the *send to* clause or *receive from* clause of the communication step that will be aggregated.
Example: `Aggregate (ReadData, ProcessData, WriteData) as Processing`

Transformation A transformation allows us to transform internal or externally received XML documents to another XML document underlying a different schema. The operation takes as input a set of parameter position, an expression and the resulting datatype. The expression can be any XQuery statement and the result can be mapped accordingly with our map operation. The position of parameters are labeled with $WDL_0 \dots WDL_n$ for internal and $VDL_0 \dots VDL_n$ for external parameters.
Example transforming the data of the two internal parameters *WDL0* and *WDL2* to a variable called *$aVariable* of type *Integer* which is then used as an input to the external parameter in the view *VDL0*:
`transform(WDL0,WDL2,EXPR,Integer) to VDL0/aNameSpace:Integer:$aVariable`

Map Operation The map operation is used to provide a mapping from internal parameters to external view parameters or from variables that are the result of a transformation to a parameter.
Example mapping the first internal to the first external parameter:
`map WDL0 to VDL0`
Example mapping two internal parameters via a transformation to an external view parameter that is filled by a variable of type BigInteger which is defined in an imported namespace:

```
map transform(WDL0,WDL1,EXPR,aNameSpace:BigInteger)
to VDL0/aNameSpace:BigInteger:$Var
```

Example View Definition In listing 1.4 an example view definition is shown. It creates the view of the sellers orchestration from listing 1.2 for a specific german buyer that holds the role *Buyer*. In order to setup a cooperation we will expose the *ReceiveOrder* step as *EmpfangeBestellung* and the *SendConfirmation* step as *SendeBestaetigung* to the Buyer. All other steps should not be visible in the view for the Buyer. In addition the buyer uses another XML-Schema this means data needs to be transformed. In detail the *SendConfirmation* step of the seller in line 17 of listing 1.2 sends the original order document and a confirmation document. The buyer can only cope with one single confirmation document that contains elements of both. The described view is defined in listing 1.4. A view definition needs to declare all namespaces that are used in variables which are defined in the view (see line 2). The declaration of namespaces and variables is analogues to the their definition in workflows. The body of a view definition contains a sequence of expose statements. An expose statement has the form: Expose followed by the label of the step in the workflow. With the optional as statement the step can be renamed. The expose statement is followed by the expose body that is surrounded by curly brackets. Within the block a number of map statements can be used. Line 9 expresses that the variable *$BestellDokument* that is at the first position in the view should be transformed by the expression *EXPR* and afterwards be send to the first parameter in the receiving workflow. Analogues to the transformation of a receiving step line 13 expresses that the first two parameters (WDLO,WDL1) from the step in the workflow should be transformed to one single variable called *$EmpfangsBestaetigung* that is at the first position of the step in the view. In the latter two examples *EXP* stands for any XQuery expression.

Listing 1.4. View Definition for Buyer

```
1  #View Header
2  Namesapce Buyer:http://Buyer.xsd;
3  VAR $BestellDokument Buyer:OrderDocument
4  VAR $EmpfangsBestaetigung Buyer:OrderDocument
5
6  #View Body
7  Create View BuyerView based on SellerWorkflow for Buyer {
8      Expose ReceiveOrder as EmpfangeBestellung {
9              map transform($BestellDokument at VDL0 via EXPR) to WDL0;
10     }
11
12     Expose SendConfirmation as SendeBestaetigung {
13             map transform(WDL0, WDL1 via EXPR1) to $EmpfangsBestaetigung at VDL0;
14     }
15 }
```

The given view definition creates a view. The view is on the one hand some kind of interface definition that allows the buyer to setup its workflow/view in order to cooperate on the other hand it is used to actually run the transformation in the SmartViewProxy. The resulting workflow document is the following:

Listing 1.5. Resulting WDL definition for Buyer

```
#Workflow Header
Namespace Buyer=http://Buyer.xsd;

Variable definitions
VAR $BestellDokument Buyer:OrderDocument
VAR $EmpfangsBestaetigung Buyer:OrderDocument

Workflow Body
RECEIVE EmpfangeBestellung($Bestelldokument) from BuyerRole
SEND SendeBestaetigung($EmpfangsBestaetigung) to BuyerRole
```

4 Conclusion and Future Work

We introduced two new languages, the DWDL and WfVDL. The DWDL allows us to specify full-blocked workflows which can span multiple business partners. The communication is realized by executable views on DWDL documents which are created with a view definition language WfVDL.

References

1. Chebbi, I., Dustdar, S., Tata, S.: The view-based approach to dynamic inter-organizational workflow cooperation. Data Knowl. Eng. **56**(2) (2006) 139–173
2. Amirreza, T.N., ed.: Web Service Composition Based Interorganizational Workflows: Modeling and Verification, Saarbrcken, Germany, Sudwestdeutscher Verlag fuer Hochschulschriften AG (5 2009)
3. Johann, E., Groiss, H., Liebhart, W.: The workflow management system panta rhei. In: NATO Advanced Study Institute on Workflow Management Systems (WFMS), Istanbul, Turkey, Springer-Verlag (1 1998)
4. GmbH, G.I.: @enterprise
5. Combi, C., Gambini, M.: Flaws in the flow: The weakness of unstructured business process modeling languages dealing with data. In Meersman, R., Dillon, T.S., Herrero, P., eds.: OTM Conferences (1). Volume 5870 of Lecture Notes in Computer Science., Springer (2009) 42–59
6. Van Der Aalst, W.M.P., Ter Hofstede, A.H.M., Kiepuszewski, B., Barros, A.P.: Workflow patterns. Distrib. Parallel Databases **14**(1) (2003) 5–51
7. OMG - Object Management Group: BPMN 2.0 Specification (August 2009)

A EBNF of the DWDL in JavaCC

```
PARSER_BEGIN(DWDLParser)
package parser.dwdl;

public class DWDLParser
{
  private static String workflowName = "";

  private static ISymboltable _symboltable = new SymbolTableImpl();

  public DWDLParser() {
        _symboltable = new SymbolTableImpl();
  }

  public static void main(String args []) throws ParseException, FileNotFoundException {
        if (args.length != 2) {
                System.out.println("The source and the output file parameter must exist!");
        } else {
                InputStream stream = (InputStream)new FileInputStream(new File(args[0]));
                DWDLParser parser = new DWDLParser(stream);

        // set the source file as first workflow name
        setWorkflowName(args[0]);

                try {
                        parser.Start();

                        CompilerMessage.printOK(parser.getWorkflowName());
                } catch (TokenMgrError ex) {
                        CompilerMessage.printError((ICompilerError)ex, parser.getWorkflowName());
                } catch (ParseException ex) {
                        CompilerMessage.printError((ICompilerError)ex, parser.getWorkflowName());
                } catch (DWDLException ex) {
                        CompilerMessage.printError((ICompilerError)ex, parser.getWorkflowName());
                }
        }
  }

  private static ISymboltable getSymbolTable() {
        return _symboltable;
  }

  public static String getWorkflowName() {
        return workflowName;
  }

  private static void setWorkflowName(String name) {
        workflowName = name;
  }

  private static void setWorkflowName(IToken workflowToken) {
    setWorkflowName(workflowToken.toString());
  }

}
PARSER_END(DWDLParser)

SKIP : { " " }
SKIP : { "\r" | "\t" | "\n" }

TOKEN :
{
    < WORKFLOW: "WORKFLOW" >
|   < VAR: "VAR" >
|   < ROLE: "ROLE" >
|   < IN: "IN" >
|   < OUT: "OUT" >
|   < INOUT: "INOUT" >
|   < SEND: "SEND" >
|   < RECEIVE: "RECEIVE" >
|   < SEND_VIA: "SEND VIA" >
|   < RECEIVE_VIA: "RECEIVE VIA" >
|   < INTERFACE: "INTERFACE" >
|   < PROTOCOL: "PROTOCOL" >
|   < NAMESPACE: "NAMESPACE" >
|   < INTERNAL: "INTERNAL" >
|   < IF: "IF" >
|   < THEN: "THEN" >
|   < ELSE: "ELSE" >
|   < ENDIF: "ENDIF" >
|   < ANDSPLIT: "ANDSPLIT" >
|   < BRANCH: "BRANCH" >
|   < ANDJOIN: "ANDJOIN" >
|   < CHOICE: "CHOICE" >
|   < ENDCHOICE: "ENDCHOICE" >
|   < REQUIRE: "REQUIRE" >
|   < FROM: "FROM" >
|   < TO: "TO" >
```

```
|    < INTERNAL_ROLE: "INTERNAL ROLE" >
|    < EXTERNAL_ROLE: "EXTERNAL ROLE" >
|    < ACTIVITY: "ACTIVITY" >
|    < LEFT_PARENTHESIS: "(" >
|    < RIGHT_PARENTHESIS: ")" >
|    < LEFT_BRACE: "{" >
|    < RIGHT_BRACE: "}" >
|    < COLON: ":" >
|    < DOT: "." >
|    < SLASH: "/" >
|    < SEMICOLON: ";" >
|    < LOGICAL_EQUAL: "==" >
|    < BUCK: "$" >
|    < STAR: "*" >
|    < COMMA: "," >
|    < EQUAL: "=" >
|        < IDENT: <LETTER> ( <LETTER> | <DIGIT> )* >
|    < BUCK_IDENT: < BUCK > < IDENT > >
|    < STAR_IDENT: < STAR > < IDENT > >
|    < XML_TYPE: < IDENT > ( < COLON > < IDENT > )* >
|    < XSD_NAME: < IDENT > ( < COLON > | < DOT > | < LETTER > | < DIGIT > | < SLASH >
)* ".xsd" >
}

TOKEN:
{
        < LETTER: ["A" - "Z"] | ["a" - "z"] | "_" >
|       < DIGIT: ["0" - "9"] >
}

void Start() throws DWDLException : {}
{
        workflow() <EOF>
}

void workflow() throws DWDLException : { IToken workflowToken; ISymbol sym; }
{
    < WORKFLOW >
    workflowToken=< IDENT >
    {
        // add the workflow symbol to the scope
                sym = new SymbolImpl(workflowToken.toString(), ISymbol.Workflow);
                try {
                        getSymbolTable().addSymbol(sym);
                } catch (DWDLException ex) {
                        throw new DWDLException(ex.errorNumber(), ex.getSymbol(), workflowToken);
                }

            // store workflow name for the parser (error messages)
        setWorkflowName(workflowToken);
    }
  < LEFT_PARENTHESIS >
  < RIGHT_PARENTHESIS >
  < LEFT_BRACE >
  // Variable definition
  (
    imports()
|       variable_definition()
|       role_definition()
|       activity_definition()
  )*
  (
    step()
  )*
  < RIGHT_BRACE >
}

void imports() throws DWDLException : { }
{
    < NAMESPACE >
    < IDENT > // import name
    < EQUAL >
    < XSD_NAME >
    {



    }
    //< SEMICOLON >

}

void variable_definition() throws DWDLException : { IToken name, type;  }
{
    < VAR >
    name = < BUCK_IDENT > // variable name
    type = < XML_TYPE >  // variable type
    {
```

```
    }
}

void role_definition() throws DWDLException : { }
{
        internal_role()
|       external_role()
}

void internal_role() throws DWDLException : { }
{
        < INTERNAL_ROLE >
        < IDENT > // role name
}

void external_role() throws DWDLException : { }
{
        < EXTERNAL_ROLE >
        < IDENT > // role name
}

void activity_definition() throws DWDLException : { }
{
        < ACTIVITY >
        < IDENT > // activity name
    < LEFT_PARENTHESIS >
    [
        form_parameter()
        (
                        < COMMA >
                        form_parameter()
        )*
    ]
    < RIGHT_PARENTHESIS >
 //   < SEMICOLON >
}

void form_parameter() throws DWDLException : { }
{
        mode()
        < XML_TYPE > // type of the parameter
}

void mode() throws DWDLException : { }
{
        < IN >
|       < OUT >
|   < INOUT >
}


void step() throws DWDLException : { }
{
        normal_step()
|       conditional_step()
}

void normal_step() throws DWDLException : { }
{
    < IDENT > // label
        < COLON > // :
        (
                internal_step()
        |       receive_step()
        |       send_step()
        )
}

void internal_step() throws DWDLException : { }
{
    < IDENT > // role name
    < IDENT > // activity name
    < LEFT_PARENTHESIS >
    [
        < BUCK_IDENT > // variable name
        (
                < COMMA >
                        < BUCK_IDENT > // variable name
        )*
        ]
    < RIGHT_PARENTHESIS >
}

void xor() throws DWDLException : { }
{
        < IF >
        xor_condition()
        < THEN >
        (
          step()
```

```
        )*
        < ELSE >
        (
          step()
        )*
        < ENDIF >
}

void conditional_step() throws DWDLException : { }
{
        xor()
|       parallel_split()
|   choice()
}

void xor_condition() throws DWDLException : { }
{
    < LEFT_PARENTHESIS >
// condition
    < RIGHT_PARENTHESIS >
}

void parallel_split() throws DWDLException : { }
{
        < ANDSPLIT >
        (
                < BRANCH >
                < IDENT > // branch name
                (
                        step()
                )*
        )*
        < ANDJOIN >
}

void choice() throws DWDLException : { }
{
        < CHOICE >
        (
                < BRANCH >
                < IDENT > // branch name
                (
                        step()
                )*
        )*
        < ENDCHOICE >
}

void receive_step() throws DWDLException : { }
{
        < RECEIVE >
        < IDENT > // internal name
        < IDENT > // activity name
    < LEFT_PARENTHESIS >
        [
                < IDENT > // parameter
                < SLASH >
                < BUCK_IDENT > // parameter name
                (
                        < IDENT > // parameter
                        < SLASH >
                        < BUCK_IDENT > // parameter name

                )*
        ]
    < RIGHT_PARENTHESIS >
    < FROM >
    < IDENT > // from external name
}

void send_step() throws DWDLException : { }
{
        < SEND >
        < IDENT > // activity name
    < LEFT_PARENTHESIS >
        [
                < BUCK_IDENT > // parameter name
                (
                        < BUCK_IDENT > // parameter name
                )*
        ]
    < RIGHT_PARENTHESIS >
        < TO >
        < IDENT > // external role
}
```