Towards Choreography Model Transformation via Graph Transformation

Fenglin Han, Surya Bahadur Kathayat, Hien Le, Rolv Bræk and Peter Herrmann
Department of Telematics
Norwegian University of Science and Technology
Trondheim, Norway
Email: simon.han,surya,hiennam,rolv.braek,herrmann@item.ntnu.no

***Abstract*—We present a Model-Driven method to develop collaborative systems. In our method, we use UML collaborations to capture the requirements and architecture of such a system. The system behavior is specified by two choreography models: an abstract flow-global and a more detailed flow-localized choreography. These choreography models are both described by UML activity diagrams. A graph-based transformation approach carrying out the transformation from the flow-global to the flow-local choreography is the core contribution of this paper. Our approach is illustrated using a case study of the European Rail Traffic Management System (ERTMS).**

***Keywords*-Choreography Model; Model Transformation; Graph Transformation; Collaborative Service**

I. INTRODUCTION

Model-Driven Development (MDD) is an approach supporting the software development process by creating models on different levels of abstraction and platform independence. First, one develops more abstract models specifying the pure functionality of a particular solution or an application domain but hiding aspects of the later realization. These models can be transformed into models incorporating more implementation details. Based on the refined models, application code can be generated ranging from system skeletons to complete, deployable products for different platforms. MDD is considered effective when the transformation from the abstract to the detailed models can be done with a high degree of automation. This makes it easy to keep consistency between the two model levels. In addition, the developer can utilize the comprehensibility and the generality of the platform-independent more abstract model as well as the fine-grained semantics and the mature structuring mechanisms of the more detailed one.

In this paper we discuss a model-driven development method for distributed, reactive and collaborative services. A collaborative service is defined as a partial system functionality in which two or more components collaborate to achieve a common goal [4], [11]. UML 2.x collaborations are used to describe the structure of participants cooperating with each other, while UML activities specify the corresponding behavior. According to the objectives of MDD, we consider the behavior models (called choreography models in the following) at two levels of detail and use graph-based model transformation to derive detailed implementable models from more global abstract ones.

Figure 1 delineates the overall MDD approach:

- A flow-global choreography model (label 1 in Figure 1) seeks to specify the desired global behavior in terms as close to the problem domain as possible. It is intended to be understandable by end-users and experts of a specific domain. Thus, the flow-global choreography focuses on the global interaction while the details and resolutions of coordination problems that may occur at the level of a distributed realization are not modeled. We express flow-global choreographies by a special kind of UML 2.x activities [7].
- A flow-localized choreography (label 5 in Figure 1) is used to define global behavior in sufficient detail that coordination problems arising in a distributed system can be resolved. Further, the level of detail shall allow extensive analysis, synthesis of the behavior in the distributed system parts, and automatic generation of the application code. In particular, we apply the system engineering approach SPACE [8] and the corresponding tool-set Arctis [9] that also uses UML 2.x activities to model behavior. The models created in Arctis can fully automatically transformed to Java code running on several platforms [11].
- Flow-global choreographies can be transformed to flow-localized choreographies using graph transformation techniques. The localization policies are implemented by applying graph transformation rules (label 3 in Figure 1) to a host-graph representing a flow-global choreography model (label 2) to derive a post-graph of a flow-localized choreography

978-1-4244-9698-3/11/$26.00 ©2011 IEEE

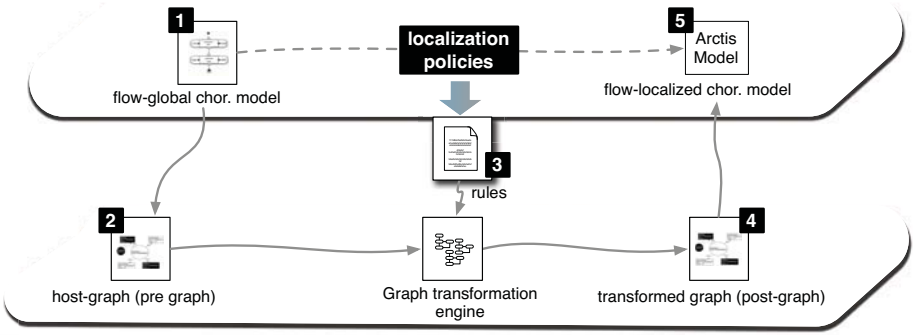

Figure 1. Overall Approach.

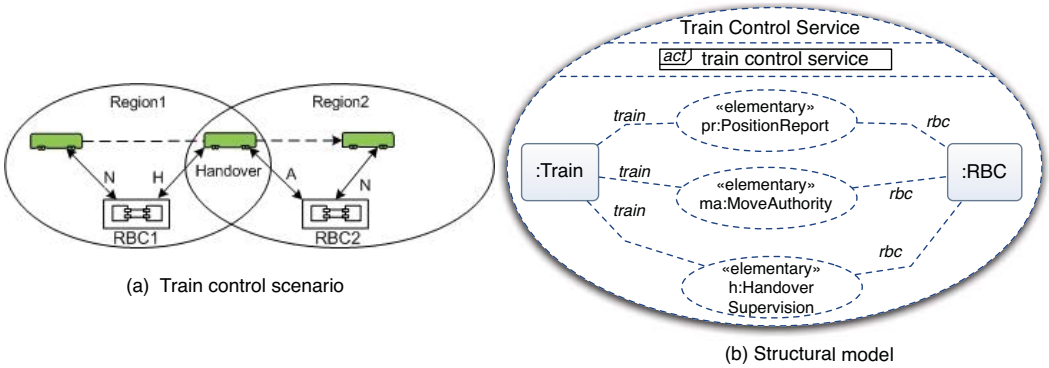

Figure 2. Structure Model of the Train Control System.

specification (label 4).

The focus of this paper is on the model transformation using a graph transformation engine. This provides for the following advantages: First, the flow-global models, which can be re-used in different system scenarios, are stored in domain-specific repositories [7]. In a similar way, we can predefine graph transformation rules for certain refinements which are stored in repositories as well and can be used for different kinds of transformations. Second, it is not necessary that the full flow-global model must already be available in order to be transformed to a flow-localized model. This is due to the possibility to transform partial flow-global choreography models.

We introduce the structure and choreography models by a train control system scenario in Section II. A survey of the task for the model transformation is provided in Section III. Model transformation using the graph transformation approach is presented in Section IV. Section V discusses the related work and is followed by concluding remarks in Section VI.

II. ARCHITECTURE AND CHOREOGRAPHY

In this section, we discuss the related models contributing to our MDD approach:

- The collaboration model that defines service participants as roles and sub-services as collaboration uses.
- The flow-global choreography model specifying the high-level global behavior including the ordering and causality among sub-services and service roles in a composite service.
- The flow-localized choreography model defining detailed behavior so that application code can be generated.

We present a summary of these models using an example of the European Rail Traffic Management System (ERTMS).

A. Collaboration

UML collaborations are used to specify the structure (i.e., roles and interactions) of distributed collaborative entities and collaboration uses representing sub-collaborations. Figure 2 (a) shows an example of a train control service which is described as following: A train must always be supervised by a radio block center (*RBC*). The *RBC*'s responsibility is to monitor and control all train movements in a particular region. Guided by its current *RBC*, the train keeps on moving and sends its position reports to the *RBC* and the *RBC* validates

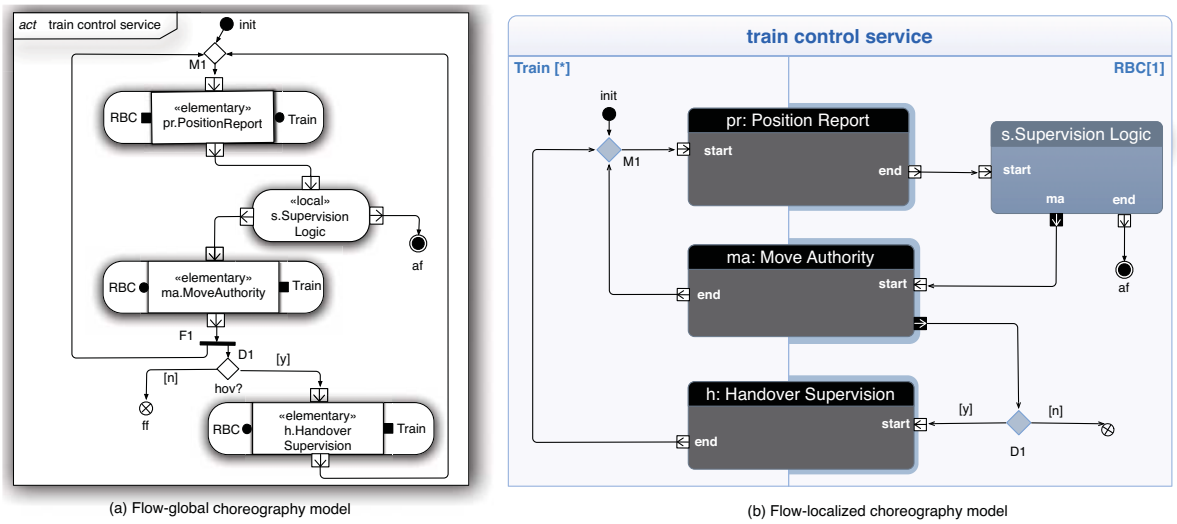

Figure 3. Flow-Global model and Flow-Localized model (derived as Arctis model).

the received position information of the train. Moreover, the *RBC* issues successive movement authorities (*MA*) to the train, which specifies a safe distance the train may travel. In addition, the train may also travel across several regions covered by different *RBC*s. This means, the supervision of the train movement can be handled by more than one *RBC*. When the train crosses a border between two regions, the *RBC* of the current region will hand over the control to another one by executing a handover process. The train control service is modelled as a UML collaboration as shown in Figure 2 (b).

The *TrainControlService* system has two main participants, *Train* and *RBC*, represented as roles. In Figure 2 (b), the *Train* and the *RBC* participate in three collaborative sub-services: *PositionReport*, *MoveAuthority*, and *HandoverSupervision* which perform the following activities:

1) *PositionReport* reports the current train position to the RBC.
2) *MoveAuthority* sends the safe travel distance from the RBC to the train.
3) *HandoverSupervision* transfers the train control supervision to the new RBC if the train travels to a different region.

B. Flow-global choreography models

The flow-global choreography shown in Figure 3 (a) is defined by UML activities connecting actions (in particular, call behavior actions, i.e., actions including own behavioral models) by flows that are not assigned to any particular role in the collaboration. Actions may either specify the behavior of a collaboration or a local activity. Collaborative actions contain references to their participants in the form of roles. Further, we indicate initiating and terminating roles by dots respective squares. Note that the pins are not localized to the roles in this model type.

Flows may contain intermediate control nodes (e.g. start, stop, choice, merge, fork and join) defining the ordering and causality among the actions. Like the pins, the control nodes are not assigned to any particular component, too.

Thus, the flow-global choreography model abstracts from several design issues that need to be addressed when transforming it to a flow-localized choreography model. We believe that this is the right level of abstraction to discuss the intended behavior with end-users and other stake holders since global choreography models specify the global action order without describing detailed interactions as in interaction diagrams. They hide details needed in a distributed realization such as the location of decisions and other control nodes, and coordination details ensuring that the global ordering is satisfied. This pure focus on the functional behavior of a system allows to gain a better understanding of the system behavior and to find potential development errors early. Further, the constriction on functionality reduces the state space produced by model checkers verifying certain system properties which eases the use of these automatic analysis tools also for more complex systems. All-in-all, a flow-global choreography model is a useful first step in the formalization of the requirements of a system.

Figure 3 (a) depicts the flow-global choreography behavior of the train control service. A train on its journey reports its current position in intervals to the *RBC* which is responsible for the region the train operates

in. This operation is specified by the collaboration *PositionReport*. Thereafter, the *RBC* validates the received position information of the train via the local activity *SupervisionLogic*. If the information about the location of the train is correct, the *RBC* issues successive movement authorities (MA) to the train which is modeled by the collaboration *MoveAuthority*. Finally, if the train crosses the border between two regions, the collaboration *HandoverSupervision* is invoked.

C. Flow-localized choreography models

As already mentioned, the flow-localized choreographies are modeled in Arctis [9], our tool-set to develop component based collaborative systems which specifies behavior using UML activities as well. Also here, actions representing the behavior of a collaboration or a local activity are connected by flows and intermediate control nodes. In contrast to the activities introduced above, however, all nodes and pins are each localized to a role specifying a participating distributed entity. The roles are represented in an activity by partitions. Flows that cross partition boundaries thereby imply communication and transfer delays.

This location information allows to analyze the flow-localized choreography for realization problems like for instance mixed initiatives in which two different physical components concurrently initiate cooperations which due to the transmission lag are not properly detected and may lead to unpredicted erroneous behavior.

In Arctis, the activities are provided with a formal semantics [10] which allows for the application of model checkers to detect design errors [11]. Further, application code for different platforms such as Standard Java, Android and Sun Spots can be automatically generated from the Arctis building blocks. In the train control scenario, this is the code for the train and RBC components.

A screen shot of the Arctis model of the train control system is shown in Figure 3 (b). The collaborative and the local actions are represented as Arctis building blocks (the dark gray respectively blue boxes with pins). Control nodes are localized to components represented by the Activity partitions. The extra gray border of the collaborative actions (*pr*, *ma* and *h*) at the *RBC* part denotes that the service roles bond to this part are multiple-session. This specifies that an *RBC* may cooperate with several trains at the same time.

III. FLOW LOCALIZATION

Given the two choreography models, we outline the overall transformation process. First, we define the causal relationship between sequential collaborative activities. Second, we introduce a localization policy based on the causal properties.

A. Causality relationship

In order to localize the flows and control nodes between actions in a flow-global choreography, we first need to classify the causality among actions that follows directly from the flow-global choreography. As described in [4], [7], the following causal relationships between any two sequential connected actions C_1 and C_2 can be:

- *Strong flows:* The terminating role of C_1 and the initiating role of C_2 belong to the same system component. In this case, the flow between C_1 and C_2 can be executed locally by this component.
- *Non-causal flows:* The initiating role of C_2 belongs to a component that does not participate in C_1. This means that local ordering between actions of C_1 and C_2 cannot be achieved by a local flow. Here, we need communication between different components.
- *Weak flows:* The initiating role of C_2 belongs to the same component as a non-terminating role in C_1. Here, the non-terminating role of C_1 and the initiating role of C_2 can be ordered by a local flow, but one has to be aware that other roles in C_1 may not be finished when C_2 starts, and both collaborations run in parallel for a while.

B. Localization policy

In the simplest case where there are no intermediate control nodes between actions (i.e., direct-flows), a global flow from C_1 to C_2 is localized as follows:

- Localize *strong* flows to the role that initiates C_2 and terminates C_1.
- *Non-causal* flows can only be maintained using send and receive events realizing communication between the role terminating C_1 and the one initiating C_2. In Arctis, this is modeled by flows passing partition borders. We call this procedure *enforced* strong sequencing [7].
- In the case of *weak* flows we have two alternatives. We can either use enforced strong sequencing as well or add an extra streaming pin to C_1 which, however, changes the internal behavior of this action slightly. For this modification we prepared a set of transformation rules for weaving extra behavior into building blocks without effecting the original functional design. This will not be addressed in this paper due to the space limitation.

If there are one or more intermediate control nodes between actions C_1 and C_2, one must consider all possible flow-paths passing through them. This means that each control node can be part of several paths. We use the following notation to represent the paths and path property:

$$Path ::= (sourceNode) \overset{causality}{\rightarrow} (targetNode)$$

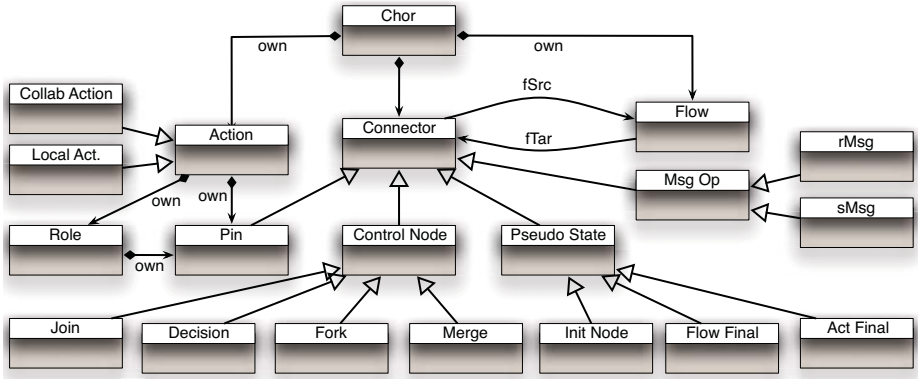

Figure 4. Meta-model for the choreography graph models.

where *sourceNode* and *targetNode* are either pseudo nodes (such as an initial node or an activity final node which are represented by $\star$) or collaboration role identifiers (represented by collaborationId.roleId). The arrows show the flow direction between those collaborative activities and contains a mark for the causality relation on top. In the flow-global choreography model in Figure 3 (a), there are in total eight paths:

- $P_0 : \star \overset{na}{\rightarrow} pr.Train$;
- $P_1 : pr.RBC \overset{strong}{\rightarrow} s.RBC$;
- $P_2 : s.RBC \overset{strong}{\rightarrow} ma.RBC$;
- $P_3 : ma.Train \overset{strong}{\rightarrow} pr.Train$;
- $P_4 : ma.Train \overset{na}{\rightarrow} \star$;
- $P_5 : ma.Train \overset{weak}{\rightarrow} h.RBC$;
- $P_6 : h.Train \overset{strong}{\rightarrow} pr.Train$;
- $P_7 : s.RBC \overset{na}{\rightarrow} \star$;

The abbreviation *na* means that there is no causality relationship available since the start or end of the flow is a pseudo node. P_0, P_4 and P_7 are such dangling paths in which the pseudo state and the control nodes along the path will be localized to the activity linked to the path. For instance, since the initial node occurs only in P_0, it will be localized to the role *Train*. In order to localize the remaining control nodes M_1, D_1 and F_1 and paths P_1, P_2, P_3, P_5 and P_6, we need to consider the path properties that each control node is involved in:

- M_1: $P_6(strong)$, $P_3(strong)$;
- F_1: $P_3(strong)$, $P_5(weak)$;
- D_1: $P_5(weak)$.

M_1 fulfills *strong* causality for both involved paths and is therefore localized to *Train*. In contrast, F_1 and D_1 contain *weak causal* paths such that we need to find suitable breaking points along the involved paths, i.e., P_5 containing both F_1 and D_1. To achieve that, we assign a breaking priority level to all the nodes on the path. The priority level is defined by the combination of causality properties in Table I. Here, the columns and rows represent the causality property of a path through a node. Altogether, we define seven breaking priority levels of which 1 refers to the lowest and 7 to the highest priority.

Table I
Localization priority order and policy matrix for control node.

property	strong	weak	non-causal	all
strong	1	2	3	-
weak	-	5	6	-
non-causal	-	-	7	-
all	-	-	-	4

In the *TrainControlSystem* specification, D_1 is involved in $P_5(weak)$ and has priority level 5 according to Table I. Similarly, F_1 is involved in $P_3(strong)$ and $P_5(weak)$ and has priority level 2. According to our policy, D_1 is selected as breaking node as it has the higher breaking priority level. In this case D_1 is broken at the incoming edge, i.e., if we decide to resolve the weak flow by enforced strong sequencing, the communication is provided at the edge between F_1 and D_1. However, in the succeeding refinement steps, we decide to add a streaming pin to $ma.MovementAuthority$ on the side of the *RBC* from which the edge to D_1 begins. Then, the fork F_1 has only one outgoing edge remaining and will therefore be removed. After finishing the various transformation steps, the constituted flow-localized graph model is transferred to the Arctis specification shown in Figure 3 (b).

IV. Graph-based Model Transformation

This section discusses in which way graph transformation techniques can be used for model transformation from the flow-global choreography model to the flow-localized model. In the following, we describe the definition of the graph models, the graph transformation rules and implementation aspects.

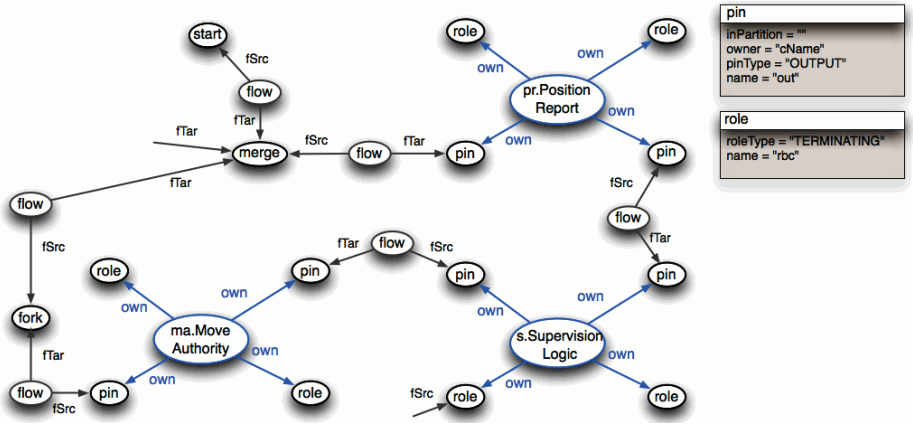

Figure 5. Graph model of the part of choreography graph model of Train control scenario

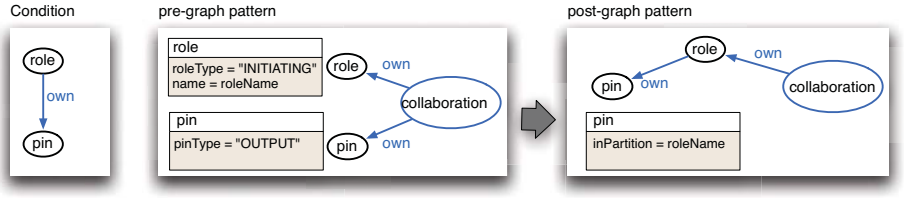

Figure 6. Pin localization rule graphs.

A. Graph model definition

The meta-model for the choreography graph model (also called *type graph* in the following) is shown in Figure 4 in terms of UML class diagrams. A choreography type graph depicted by the class *Chor* has three main entities: *Actions*, *Connectors*, and *Flows*. An action can be either a *local activity* performed by only one specific role or a *collaborative activity* carried out by the cooperation of at least two roles. Connectors represent the mechanism to connect Actions, i.e., how collaborations connect to other collaborations or local activities. There are three types of connectors: *Pin*, *Control Node* and *Pseudo State*. Pins are connection points which are associated with either Roles (in the flow-localized form) or Actions (in the flow-global form). Control Nodes include join, fork, merge and decision nodes. Pseudo Nodes include initial nodes, activity final nodes and flow final nodes. Connectors can also be message operators (*MSg Op*), i.e., send message actions (*sMsg*) or receive message actions (*rMsg*).

Moreover, there are three types of graph edges: *own* specifies that one node is owned by another which is described by aggregations in the meta-model. Further, *fSrc* models the source node of the flow while *fTar* specifies the flow target.

Based on this type graph, two kinds of graph models can be defined corresponding to the flow-global and flow-localized choreography models. Figure 5 illustrates a flow-global graph representation of the train control system in Figure 3 (a) without the Handover activity. Note that there are two dangling edges in the graph model: *fTar* to a merge node and *fSrc* to a role node. These edges will eventually be connected to the Handover activity.

B. Graph models of the transformation rules

Transformation rules can be visualized and are mainly composed of two graph parts: the pre-pattern subgraph, expressing what to replace, and the post-pattern subgraph describing the replacement. A transformation condition can also be used to define conditions or constraints describing how and under which conditions the graph production can be applied (such as a negative application condition NAC introduced in [16]). In the following, we introduce the graph model transformation rules and the policies through some representative rules: the pin location rule and the direct flow localization rule.

1) Graph models of the pin localization rule: Figure 6 depicts the graph transformation rule for pin allocation. Pins in the pre-pattern of the graph model are owned by collaboration nodes as shown on the left side of Figure 6. They are localized, i.e., connected to roles in the post-pattern of the graph as depicted on the center and right sides of Figure 6. Note that attributes and their values attached to the graph nodes can be used, checked,

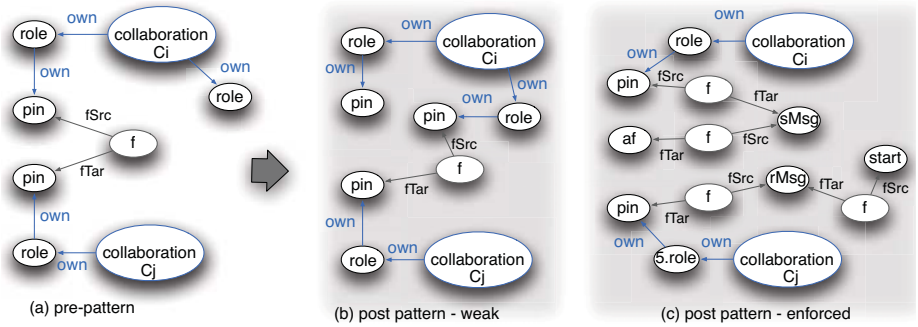

Figure 7. Direct flow localization rule.

or modified. For example, the attributes *roleType* and *pinType* are checked in the nodes *role* and *pin* in the pre-pattern of the graph model. Similarly, the attribute *inPartition* of the pin node is modified (or assigned a value) in the post-pattern of the graph.

2) Graph models of direct flow localization rules: Figure 7 shows the pre-pattern and post pattern of the rules which localize the direct flow having *weak* causality and *non-causal* causality properties. Figure 7 (a) depicts the pre-pattern graph model of the direct flow in which pin allocation rules have been performed. If the flow is *weak* causal, the corresponding post-pattern is shown in Figure 7 (b). In this case, a new output streaming pin is added to a collaborative activity *Ci*. An example of this type is P_5 in the case study. In the case that a direct flow-path is *non-causal*, the flow-path is resolved using send and receive message nodes as shown by the post-pattern graph model in Figure 7 (c).

C. Implementation

An Eclipse plug-in has been developed to create graph models of flow-global choreography models and to import the post-graph (as a result of transformation) into Arctis. As graph transformation engine, we use the Attributed Graph Grammar System (AGG) [16]. AGG offers high flexibility in creating the visual definition of graph models. Further, it provides Java APIs facilitating its integration to the Arctis tool which is also Java-based. AGG has also a facility to enable the verification and correctness of models during transformation.

The AGG graph transformation engine takes the graph model of a flow-global choreography as well as the rules introduced in Section IV-B as inputs and produces the post-graph model which corresponds to a flow-localized choreography.

V. Related Work

A comparison of approaches and tools that use graph transformation techniques for model transformations is provided in [2]. In [3], Kerkouchea et al. propose an approach for transforming UML state-chart and collaboration diagrams to Colored Petri nets using graph transformation techniques. In contrast, the authors of [1], [2] suggest to map activity diagrams into communicating sequential processes (CSP).

Unlike these approaches, both our abstract and detailed models are based on UML activity diagrams. Moreover, our approach envisages collaborative building blocks encapsulating the interaction between different components. UML activities can be defined hierarchically by means of call behavior actions. We use AGG as our graph transformation engine and our post graphs can be directly imported to the Arctis tool for further analysis, synthesis and code generation.

In contrast to [6], [15], we currently miss the formal proof that our graph-based transformation is correctness-preserving. Due to the formal semantics of Arctis [10], however, which can also be used for the flow-global choreography models, the correctness verification can be provided as temporal logic-based refinement proofs [13] which is intended to be done for the close future.

Ideally, one can choose any graph transformation tool to validate or implement our approach. Some candidates other than AGG are ATOM3 [12], VTMS [14] and C-SAW [17]. As mentioned above, we chose AGG due to its high flexibility and its Java-compliance.

VI. Concluding Remarks

In this paper, we presented a Model-Driven Development approach to support the engineering process of distributed collaborative services. The global behavior and distributed realization are captured by two different types of choreography models: flow-global and flow-localized, which are specified using UML activity diagrams. The transformation between these two choreography models are performed with the support of graph transformation techniques. The approach is used within the EU-funded project CESAR for the cost-effective development of safety-relevant embedded systems [5]. As

future work, we plan to refine the model transformation policies and test them with larger and more complex system models. As mentioned above, we will further prove the correctness of the graph transformation rules formally.

Graph transformation is also useful for automatic collaborative building block refinement. We currently developed a set of graph transformation rules to detect and correct mixed initiatives in distributed systems. As mentioned earlier, in this often occurring but not corrected class of errors, two system components may concurrently start cooperations which might lead to unpredictable system errors. Altogether, we are convinced that graph transformation is a highly promising extension to MDD which, essentially, consists of the stepwise refinement of graphical notations. Here, the flexibility of this rule-based approach may facilitate the refinement steps significantly.

References

[1] Elmansouri, R., Hamrouche,H., Chaoui, A.: From UML Activity Diagrams to communication sequential processes (CSP) Expressions: A Graph Transformation Approach using Atom3 Tool. International Journal of Computer Science, 8(2), 2011.

[2] Varró, D., Asztalos, M., Bisztray, D., Boronat, A., Dang, D., Geiß, R., Greenyer, J., Pieter, V., Kniemeyer, O., Narayanan, A., Rencis, E., Weinell, E.: Transformation of UML Models to CSP: A Case Study for Graph Transformation Tools. in: Applications of Graph Transformations with Industrial Relevance, pp. 540–565. Springer-Verlag (2011).

[3] Kerkouchea, E., Chaouib, A., Bourennanec, E., Labbanic, O.: A UML and Colored Petri Nets Integrated Modeling and Analysis Approach using Graph Transformation. In: Journal of Object Technology, vol. 9, no. 4, pp. 25ÃŘ43. JOT (2010)

[4] Castejon, H., Braek, R., and von Bochmann, G. (2007). Realizability of collaboration-based service specifications. In *Software Engineering Conference, 2007. APSEC 2007. 14th Asia-Pacific*, pages 73 –80.

[5] CESAR (2010). http://www.cesarproject.eu/ Accessed Jan 2011.

[6] Jayaraman, P., Whittle, J., Elkhodary, A., and Gomaa, H. (2007). Model composition in product lines and feature interaction detection using critical pair analysis. In Engels, G., Opdyke, B., Schmidt, D., and Weil, F., editors, *Model Driven Engineering Languages and Systems*, volume 4735 of *Lecture Notes in Computer Science*, pages 151–165. Springer Berlin / Heidelberg.

[7] Kathayat, S. B. and Bræk, R. (2010). From flow-global choreography to component types. In *System Analysis and Modeling (SAM)*, volume 6598 of *Lecture Notes in Computer Science*. Springer - Verlag.

[8] Kraemer, F. A. (2008). *Engineering Reactive Systems: A Compositional and Model-Driven Method Based on Collaborative Building Blocks*. PhD thesis, Norwegian University of Science and Technology.

[9] Kraemer, F. A., Bræk, R., and Herrmann, P. (2009) Compositional Service Engineering with Arctis. *Telektronikk*, 105(2009)1.

[10] Kraemer, F. A., and Herrmann, P. (2010) Reactive Semantics for Distributed UML Activities. In *Formal Techniques for Distributed Systems*, volume 6117 of *LNCS*, pages 17–31, 2010.

[11] Kraemer, F. A., Slåtten, V., and Herrmann, P. (2009). Tool support for the rapid composition, analysis and implementation of reactive services. *Journal of Systems and Software*, 82(12):2068 – 2080.

[12] Lara, J. and Vangheluwe, H. (2002). Atom3: A tool for multi-formalism and meta-modelling. In Kutsche, R.-D. and Weber, H., editors, *Fundamental Approaches to Software Engineering*, volume 2306 of *Lecture Notes in Computer Science*, pages 174–188. Springer Berlin / Heidelberg.

[13] Lamport, L. (1994). The temporal logic of actions. *ACM Trans. Program. Lang. Syst.*, 16(3):872–923.

[14] Lengyel, L., Levendovszky, T., Mezei, G., and Charaf, H. (2006). Model transformation with a visual control flow language. *International Journal of Computer Science (IJCS)*, 1(1):45–53.

[15] Rafe, V. and Rahmani, A. (2009). Towards automated software model checking using graph transformation systems and bogor. *Journal of Zhejiang University - Science A*, 10:1093–1105. 10.1631/jzus.A0820415.

[16] Taentzer, G. (2004). AGG: A graph transformation environment for modeling and validation of software. In Pfaltz, J. L., Nagl, M., and Böhlen, B., editors, *Applications of Graph Transformations with Industrial Relevance*, volume 3062 of *Lecture Notes in Computer Science*, pages 446–453. Springer Berlin / Heidelberg.

[17] Zhang, J., Lin, Y., and Gray, J. (2005). Generic and domain-specific model refactoring using a model transformation engine. In *Volume II of Research and Practice in Software Engineering*, pages 199–218. Springer.