ISSN: 2278 – 1323
International Journal of Advanced Research in Computer Engineering & Technology (IJARCET)
Volume 1, Issue 8, October 2012

Visual Composition and Automatic Code Generation for Heterogeneous Components Coordination with Reo

Herve Kabamba Mbikayi

Abstract— **With the expansion of new technologies such as web-services, most of tools and systems have migrated toward an adaption to this actual context. However legacy components and applications still constitute major part of companies' system infrastructures. Their use in the actual context is of a particular interest for companies that possess critical infrastructures in production and that do not want to be involved in costly activities in terms of money and time. In this paper, we present a tool that enables the automatic generation of code for the composition and coordination of legacy heterogeneous components with the Reo coordination language. Reo is a channel-based exogenous coordination model that enables the coordination of entities in an effective way. The benefit it offers by its independence from the implementation of the components or their properties and their interactions makes it a good choice for coordination. In this paper, we present an approach based on the integration of the CORBA middleware as a new layer part of the Extensible Coordination Tools (ECT) framework which is a java implementation of the REO language to extend its features to the coordination of heterogeneous components.**

Index Terms—**Heterogeneous components, coordination, Reo, CORBA**

I. INTRODUCTION

Heterogeneous composition and coordination can be described as the processes of gluing concurrent components from different programming languages taking into account the relations and interactions among them to satisfy an expected goal. Coordination models and languages are a way of dealing with the growing complexity of concurrent systems by separating the active parts performing the computations, such as components or services, from those parts that glue these pieces of software together, we talk about component or service connectors.

Reo is a channel-based exogenous coordination model wherein complex coordinators, called connectors, are compositionally built out of simpler ones. So the inductive approach to increasing complexity is used here. As its main feature, Reo facilitates compositional construction of circuits which can be seen as communication mediums that coordinate the interactions among the entities that are part of the composed system. Using the Reo visual language, one can simply design and check connectors as well as runtime engines. Moreover components interactions and composition can be abstracted and designed graphically using Reo connectors. The final results are auto-generated codes gluing all those components and executing the designed circuit that abstracts the behavior of the interactions among the components to reach a certain goal. While it is a coordination model, Reo really doesn't emphasize on the computational entities whose activities it coordinates. These objects can be represented as modules of sequential code, processes, threads, and even software components. The separation of the computation from the behavior of the specific composed system leads to formal ways to address coordination issues by abstracting the interactions between the different components involved in the computation. Their behavior in the computation is however modeled using constraint automatons which abstracts the states of the composed system thus offering a mathematical representation of the interactions among all parties.

Being a powerful framework implementing the Reo language and integrating a set of tools for coordination of components, including model verification and even behavior simulation, the Extensible Coordination Tools (ECT) has been from its beginning subject to many extensions to support new features and concepts that can deal with the Reo language. This latter one is implemented using the Java language thus enabling only the composition of component developed in Java.

Although the best choice would be to develop and compose components using the same environment to avoid interoperability issues, in most of the cases it would be common to deal with objects from different environments due to the open choice individuals, companies or organizations have in deciding which environment to use.

The Reo semantics at a higher level can easily model the coordination of those objects independently of any restriction on the environment that should be used. However, at a lower level composing components based on different development platforms to build the final application appears to be a real issue that still needs to be solved. During the past days, there has been a work tackling part of this problem, addressing it as services orchestration. An approach on how web-services (ws) which are in some way independent of the environment can be orchestrated with Reo has been proposed in [1]. Although that work tried to open the door to many

Manuscript received Oct 18, 2012.
Herve Kabamba Mbikayi, *Leiden Institute of Advanced Computer Science, Leiden University., Leiden, The Netherlands, Phone/ Mobile:+31684906277*

technologies such as CORBA, SOAP; some questions raise in the way the application is modeled using ws which are only based on message passing and any use of external technologies such as CORBA or SOAP is handled by the ws deployed infrastructure.

Over the time, there has been a huge move toward a new way of developing applications by building systems from different components. This process of gluing such objects is called composition. With the expansion of the new technologies such as ws, many companies have been migrating their infrastructures toward service oriented ones. However, all enterprises have legacy applications and databases, and still want to continue to use them while adding or migrating to a new set of applications that utilize the Internet, e-commerce, and other new technologies.

Rewriting legacy applications running on old systems to connect them to this new functionality is usually expensive and time-consuming. Moreover, already purchased systems may be too critical or costly to replace. Legacy applications are still being developed in the market and their components present a strong need for adequate coordination and composition to build powerful systems.

In this paper, we propose a way to compose and coordinate the interactions of legacy components from different platforms using the graphical language Reo. We propose an approach based on the integration of the Common Object Broker Architecture (CORBA) middle-ware to handle the platforms dependability of components. We tackle this issue on a practical way presenting a case study as proof of concept that captures the given problem and propose a way to deal with it as solution.

The rest of the paper presents all needed steps toward the automatic generation of Java objects for the coordination of the heterogeneous components, including the interfaces they will use to invoke operations on the ORB server given an IDL file, how each of them behaves given a constraint automata and how their interactions are coordinated given a Reo connector.

In sec. 2, we present the Reo language to give some background, in sec. 3 we introduce the Common Object Broker Architecture (CORBA). The section 4 describes the full architecture of our platform as well as the steps involved in its implementation. In Sec 5, we present a case study to familiarize the reader with our tool and in section 6, we draw final conclusions and our future work is described.

II. REO

Reo is a channel-based exogenous coordination model wherein complex coordinators, called connectors, are compositionally built out of simpler ones [2]. The simple connector is called a channel. User can define channels with any behavior. Figure 1 shows the basic set of Reo channels. Channels can be considered as primitives, user-defined entities that have two ends that can be either sources or sinks. A source end on its role can only accept data in, while the sink end dispenses data out of its channels. The Reo Coordination Language can be used to construct negotiation protocols as it provides formal semantics to abstract objects and their interactions.

We give below a description of the basic channels in Reo.

- Synchronous: Atomically fetches an item on its source end and dispatches it on its sink end.
- Lossy synchronous: Atomically fetches an item on its source end and, non-deterministically, either dispatches it on its sink end or loses it.
- $FIFO_1$: an asynchronous channel with one buffer cell which is empty if no data item is shown in the box. If a data element d is contained in the buffer of a $FIFO_1$ channel then d is shown inside the box in its graphical representation.
- Synchronous Drain: a synchronous channel that has two sources and no sink. It accepts data simultaneously if it is available on both sides. Due to its lack of sink all data it accepts is lost.
- Synchronous Drain: an asynchronous channel that has two sources and no sink. Contrarily to the Synchronous Drain, it accepts data from both sides but never simultaneously but all accepted data is lost.
- Filter (P): Atomically fetches an item on its source end and dispatches it on its sink end, if this item satisfies the filter constraint P; loses the item otherwise.
- Synchronous Spout (p): has two sink ends. The pair of write and read operations on its two ends can succeed only simultaneously. Each sink of this channel acts as an unbounded source of non-deterministic data items that match with the pattern p. The data items taken out of the two sinks of this channel are not related to each other.
- Asynchronous Spout (p): is dual to Asynchronous Drain and has two sinks through it produces non-deterministic data items. The channel guarantees that two operations.
- Timer channel: with early expiration allows the timer to produce its time-out signal through its sink end and resets itself when it consumes a special "expire" value through its source [3].
- P-producer: requires synchronous writing and reading at its source and sink end. When an arbitrary data item is written at its source end, it non-deterministically generates a data item $d \in P$ which is then taken at its sink end [4].

ISSN: 2278 – 1323
International Journal of Advanced Research in Computer Engineering & Technology (IJARCET)
Volume 1, Issue 8, October 2012

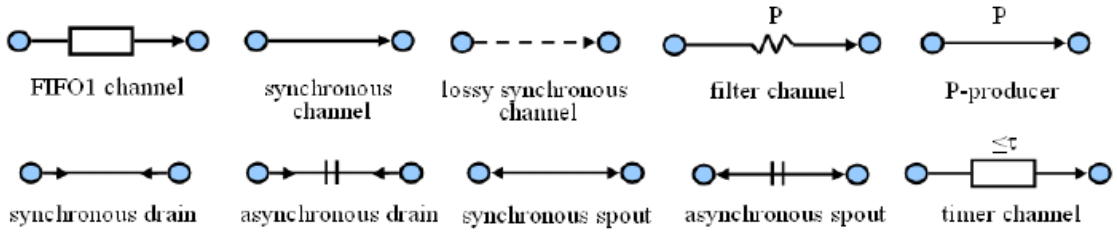

Figure 1: Graphical representation of basic channels

The behavior of each connector in Reo imposes a specific coordination pattern without explicit knowledge of the components [5], [6]. This is the particularity owned by Reo that enables the separation of the computation from the behavior of the composed system. The components, on the other hand, perform write and read operations through the connectors, without prior knowledge of the type or the topology of connectors. This creates a clear independence between Reo and the components' code, as well as between the components themselves. Channels are composed by joining them into nodes to form a complex structure representing a connector which is a formal concept on which the Reo language lies. A system is considered by Reo as a set of components instances that interact through connectors but, emphasis is given on the communication between the component instances which takes place through the connectors.

Reo models a connector as a graph of nodes and edges [2], [7]. There are three types of nodes namely sink, source and mixed. A mixed node contains both source and non-deterministically selects a data item available at one of its sink channel ends and replicates the data item to all of its source channel ends when all of them can accept the data item. A source node contains only source channel ends. If a component writes a data item to a source node, the node replicates the data item to all of its source channel ends when all of them can accept the data item. A sink node contains only sink channel ends. When a component tries to take a data item from a sink node, the node non-deterministically selects a data item available at one of its sink channel ends. Reo enforces a channel based coordination model that defines how complex coordinators called connectors can be built out of simpler ones. It also enables one to clearly designate the order of events that take place in a negotiation.

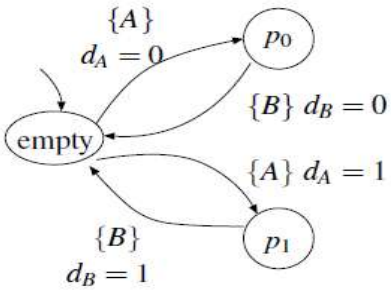

Figure 2: Constraint automaton for FIFO1

To describe the behavior of Reo connectors, various formalisms have been proposed, each of them adapted to a particular approach and solution, among them connector coloring which marks the presence or absence of data flow by graphically coloring the connector [8]; and constraint automaton (CA) which can abstract the internal configuration of a connector represented by its states, are the most used. In some cases having knowledge of existent relations between them gives a deep understanding on the applicability of the designing of tools that implement the Reo language. An investigation on the existent relations between CA and Connector Coloring semantics has been done and some operators to abstract the transformation from one semantic to the other one have been proposed in [11].

In this paper, we use constraint automata to formalize the behavior of the components and to compile Reo circuits to Java code in an automatic way.

In regards to its finite states, CA can be considered as finite states machine. It abstracts the behavior of a Reo circuit by using its states to represent the configurations of its corresponding circuit, while its transitions are used to represent what is going to happen on triggered event (constraints) or simply its maximally-parallel stepwise behavior. Figure 2 depicts a constraint automaton for a $FIFO_1$ channel with source node A and sink node B, assuming that the data domain consists of two data items 0 and 1 and that, the initial state (empty) stands for the configuration where the buffer is empty, while the states p_0 and p_1 represent the configurations where the buffer is filled with one of the data items [4].

I. COMMON OBJECT REQUEST BROKER (CORBA)

The Common Object Request Broker Architecture (CORBA) is an open standard for distributed object computing defined by the Object Management Group (OMG). CORBA is a built-in middle-ware that allows a client to invoke methods on remote objects at the server independently of the language the objects have been written in, and their location. Those interactions between client and server are supported by object request brokers (ORBs) on both the client and server sides, using the IIOP which stands for Internet Inter-ORB Protocol.

A CORBA based system is a collection of objects that isolates the requests of services (clients) from the providers of services (servers) by a well defined encapsulating interface [12].

CORBA objects can be located both client and server on the same location or either distributed on a remote server, without affecting their implementation or use. The ORBs are responsible for the management of this flexibility. CORBA defines the notion of object references to facilitate

communication between objects. Those references are called Interoperable Object References (IOR). When a component of an application wants to access a CORBA object, it first obtains an IOR for that object. Using the IOR, the component can invoke methods on the server [10].

The capabilities of CORBA objects are defined using the Interface Definition Language (IDL). Operations defined on the interface accept input parameters and return values; and can raise exceptions. The implementation languages supported by CORBA include C, C++, Java, Ada95, COBOL as well as some scripting languages such as Perl, Python, Javascript. These features enable CORBA to be an independent framework capable of supporting many heterogeneous components. It works on many plate-forms such as UNIX and real-time embedded systems. Moreover, the communication protocols used by CORBA for ORB communication include TCP/IP, IPX/SPX, ATM, etc.

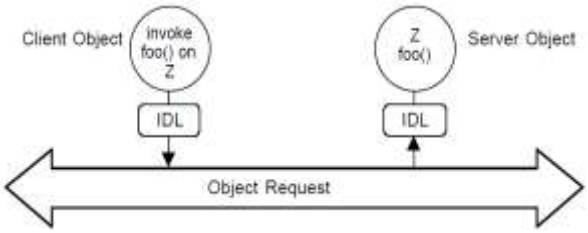

Figure 3: Client invoking a method at server

A server object implemented in Java can be accessed by a C++ component. The C++ object just needs to invoke the particular methods on the server as though they are local C++ method calls. These interactions are handled by the ORB that acts as a mediator. It provides object location and transparent access to object by facilitating clients' invocations of method on the server. A client can connect or bind to a server object statically if the server interface is known at build time.

Alternatively, the client can use dynamic binding to ascertain the server interface and construct a call to the server. The Figure 3 illustrates a client invoking a server method.

II. COMPOSING SYSTEM WITH HETEROGENEOUS COMPONENTS

Visual composition of heterogeneous components in Reo involves (i) designing a circuit to coordinate the different interactions between the entities, (ii) deploying and running the circuit; (iii) connect the different components to the circuit. The automatic generation of code for orchestrating ws with Reo using ECT, was addressed in [1], and the concept of Proxy was introduced as an intermediate agent between the circuit and the ws that relays data items from one side to the other, and vice-versa. Contrarily to the problem they addressed, the question that raises here is how to go from a designed circuit diagram abstracting the interactions among the components to the automatic generation of executable code, as well as how to connect legacy heterogeneous components to it for a proper coordination?

We propose a tool that extends this approach by integrating the Common Object Broker Architecture (CORBA) as part of the system architecture, and allows the automatic generation of code for composite systems based on heterogeneous components. Generating Java code from circuit

One feature supported by ECT is the visual designing of circuit. Users can simply drag-and-drop controls from their screens using it to design any circuit that meets their expectations. The abstracted description of this circuit will then be saved as an XML file that the Reo compiler will parse to compute the constraint automaton that captures the expected behavior of the designed circuit. Based on this latter one computed, the compiler generates a Java class which when executed will run the state machine. These functionalities are already supported by the ECT framework. At runtime when an instance of this class is executed, according to the behavior of the constraint automaton it will either await for writing or taking requests to coordinate on its nodes.

Data items are taken or written on synchronization points that refer to the data structures used to store them [1]. To execute a transition based on the constraint automaton description, the running automaton instance first checks the synchronization point involved in a particular transition to see if a particular request at that point satisfies the predetermined condition for that transition, by invoking a routine that does the jobs. If the condition is satisfied, the transition fires and the automaton changes state according to its specifications.

The execution of the constraint automaton can be considered as the runtime coordination based on the pre-designed circuit that Reo uses to coordinate interactions among system composing entities. However generated and executed code for the state machine are Java environment specific thus raising an issue on how to connect heterogeneous components to the interfaces of the executed automaton known as synchronization points. A simple example could be i.e. a concurrent C++ component needing to take some data items from its synchronization points connected to the circuit and a Python component needing to dispense data through its sink channels. Clearly an issue appears at this level while a solution to it has the benefit to expand the Reo language to the coordination of platform dependent components.

We tackle this problem under two approaches dividing it into two questions to solve. (i) How to break the dependency between the components? (ii) How to connect those components to their synchronization points to the executed circuit?

An answer to the second question needs first to solve the first problem. For this, we propose to use an architecture based on CORBA as part of coordination system to break the dependency of components. For the second problem we use an approach based on proxy as an intermediate agent between the Reo circuit and the composing components [1]. This proxy must be automatically generated based on the IDL file describing the exposed interfaces between the involved components and the CORBA server.

A. Breaking components dependency

Next, we discuss how CORBA is integrated in our tool to

break the dependencies of components to their respective environments. CORBA is a middle-ware that uses the IIOP protocol to enables components from different environments to invoke methods on a particular server. However, being an independent and complete infrastructure, its integration results in adding a new layer to the ECT framework, and raises a question on how to model this integration for the automatic generation of proxies to handle heterogeneous components coordination and composition. The CORBA server must first expose at least two public interfaces or methods through an IDL file which describe the actions that will be performed by the components. We have abstracted them into two principal activities that are "writing" and "taking". Suppose that a component C_1 wants to write a data item to its sink end and that another component C_2 wants to take data from its source end. Since the automatic generation of proxies is based on the information of the IDL file which describes the exposed interfaces. C_1 will call the "write" method on the CORBA server and will pass a formatted string containing the information on the synchronization point's name, the data to write, the location and port of the CORBA server to it.

By invoking the write action (method) on the CORBA server, the object representing the interface between the proxy and the CORBA server side is called as a method taking as arguments, the formatted string described above and encapsulating the payloads that needs to be written to synchronization points then, the corresponding transition in the executed constraint automaton will fire as described by the behavior file abstracting the coordinator.

When C_2 wants to take data, it calls the "take" method on the CORBA server that accordingly, will access the interface of C_2 connected to the proxy through the CORBA server and take the data resulting in firing a transition on the executed state machine.

The component involved is the "CCU" in Figure 4 and has a full knowledge of the executed constraint automaton by monitoring its transitions. The communication between the components is ensured by CORBA and their connections to the circuit are managed by the CORBA proxy. We discuss some details about it in the below section.

B. Proxy generation

In the above section we discussed on how environment dependent components can communicate in Reo, in this section we discuss on how these components can be connected to the Reo circuit for their coordination.

Suppose that we have a reo circuit RC and that we want to connect our composing components to their synchronization points in RC. This approach raises an unacceptable issue due to the limitation of Reo. Reo is implemented in Java and how can a C++ component be plugged into its synchronization points in RC? Clearly to avoid recoding tasks on either the Reo language implementation framework or components' sides, an additional layer can be added between the both sides to handle these synchronizations. We call it Proxy. A proxy has two interfaces, one connected to RC and the other one connected to the components through CORBA technologies.

The proxy needs to have full knowledge on the behavior of the components to ensure a proper synchronization with Reo circuit it is connected to. The processes involved in the generation of the proxy from the architecture of the proxy generator shown in Figure 4, are the same with those proposed in [1]. Only input data for the generator are modeled differently to produce a runtime heterogeneous coordinator. However name of the components are changed to fit the description of our tool. More details on it can be obtained in [1].

Automatic generation of proxies: The proxy generator needs two files to generate the proxy. An IDL file containing technical information about the CORBA server and the exposed interfaces is required as well as a file abstracting the configuration of the circuit. Based on this latest one, the generator will first compute a constraint automaton for the circuit. Going through all steps described in Figure 4, proxy's codes will be automatically generated. More details about this process can also be found in [1].

Putting everything together: after generating the proxy, we need to combine it to the compiled Reo circuit that will coordinate the components interactions. In Figure 4, the component named "Composing code" is responsible for that. It will produce a Java class that will create a synchronization point for each boundary node of the circuit and pass them to an instance of the constraint automaton code generated by the Reo Compiler based on the XML document describing the circuit. It will then create an instance of the class Proxy generated by the Proxy Compiler on input of a IDL file and CBS file, and pass to this instance, the appropriate synchronization points that was created above. Both the circuit and the proxy sharing the same synchronization points create the link between the heterogeneous components and the coordinator circuit.

III. CASE STUDY

To familiarize the reader with our tool, we consider a practical case study as an example that shows the expressiveness of Reo as well as the huge need of such tool to coordinate interactions between environment dependent components. Figure 5 shows an ECT designed representation of the circuit for coordinating the interactions of four components: Partner1, Partner2, BCKLChecker, RateRetr. This scenario is a common example of the hands-to-hands money transfer proposed by a lot of financial companies which involves a customer sending or receiving, some cash amount to or from his partner around the world through companies' partner network.

The objects named Partner1, Partner2, are respectively C++ and Java components representing the external interface of three different offices located around the world that interact through a secure channels such as IPSec to exchange information about the transaction their register. RateRetr is a Java procedure that connects to XE.com servers to retrieve the current rate of a given currency when a transaction needs to be registered and sent. BCKList is a Python component that checks in the European Union public blacklist to see if the name of the customer is present before any transaction can be registered and sent for the customer.

Figure 6 gives a description of the different services. From

Figure 5, we can many different channels have been used to compose our coordinator. Each of them is appropriate to the behavior we want our connector to have. The FIFO channels are used to store information on their buffer before it can be delivered to the appropriate nodes. The FIFO channel is used i.e. in the case when Partner1 want to register a new transaction; it first asks for the rate, and since the rate should be sent back to both Partner1 and Partner2 (history), the information is put on a buffer of the FIFO channel. We also used the Filter channels in order to deliver the correct information to the source nodes. An example in Figure 5, when Partner1 waits for the answer from the BCKList to send the information about the customer, when this information is delivered by BCKList it is written on a node that is shared by both partner1 and Partner2. The Filter channels allow one to take only the data if a certain condition is fulfilled (i.e. if the data is sent to Partner2).

We have implemented this example using a server in which we ran the CORBA instances, the coordinator and the proxy. However, the components have been deployed remotely on the network.

The order in which the actions must be performed is managed by a sequencer which enforces it on the coordinator according to desired behavior.

IV. IMPLEMENTATION OVERVIEW

When a new transaction must be registered and sent to Partner2, Partner1 first needs the actual rate of the currency used for the transaction. It writes on it sink end some data that "RateRetr" needs in order to decide if it should get the rate for Partner1. For this, Partner1 invokes the "*RequestCurrency*" method passing as parameter the information needed by "RateRetr" on the CORBA server, this method will then forward to the proxy the data that needed to be written on the synchronization point of the circuit, and this latter one will then fire a transition. Based on the ordering defined by the sequencer, "RateRetr" will then execute the "*CheckReq*" method on the CORBA server; this method will ask the proxy to take the data present on the on the endpoint corresponding to the synchronization point for "RateRetr" and return the value to "RateRetr". This latter one will get the rate from "xe.com" and execute the *"SendRate*" method on the CORBA server that will ask the proxy to write the value to the synchronization point with the circuit. Through the channels the data is forwarded to the shared node, Partner1 will execute the "*GetCurrency"* method on the CORBA server that will ask the proxy to take it and forward it back to Partner1.

We observe that two main operations are modeled for this purpose "Write" and "Take" whatever the methods called on the CORBA server, they will either write or take the data in the same way at their synchronization points through the proxy. For the others methods, the same process are done according to their signatures and based on the specification of circuit.

Note that the proxy is relaying messages from one side to the other, thus since the components are connected to the proxy, to avoid synchronization problems, the proxy must execute a simulation of the same CA that will follow each of its transition. The executed state machine is known as the "coordinator" or circuit; in other term the proxy will monitor the executed state machine for its transitions and will do the same.

V. RELATED WORK

In [13], coordination problem of concurrent objects have been addressed and a model of coordination separating coordinators from objects to be coordinated based on Coordination Groups has been proposed. However, although this model at a higher level allows a good abstraction of the coordination, practically it lacks an implementation framework to enable the composition of applications to final codes as Reo does.

In [14], a concrete experiment in composing and coordinating heterogeneous components from different "protocol aware" framework such as CORBA, Jini and the Coordination Language Facility (CLF) has been made. An approach on how the CLF could be used as a coordinator for negotiation and to perform distributed transactions to those external frameworks has been described. This approach takes the advantage of the linearizability offered by those external frameworks to combine operations in a linearizable manner on the whole set of components which results in enforcing the serializability of the transactions.

David Safranek proposed in [15], a prototype of visual specification language called Visual Coordination Diagram for high-level design with heterogeneous coordination models. At a higher level this approach separates the behavioral aspect from the coordination. It can be mixed with other coordination models toward the formation of a new specification. The need of tools to implement at a lower level the automatic generation of codes from the visual specification is still an open challenge.

Others works on the subject, addressed the composition of the heterogeneous components such as [15], but do not put emphasis on the model used to coordinate the interactions among the composing components. Most of the works define knowledge at higher level for coordination, but lacks concrete implementation of tools in order to make concrete use of them. Reo being a powerful coordination language, at a higher level have been implemented as a framework for automatic code generation resulting from the visual composition. This has influenced our choice in choosing this coordination model to extend it to heterogeneity feature that is addressed in our work.

VI. CONCLUSION

In this paper we presented a tool which is an extension of the ECT framework for composing and coordinating heterogeneous components with the Reo language. Reo as a coordination language offers a simple way to compose the behavior of components of a system. Addressing the composition issue of heterogeneous components in Reo, we proposed an approach toward the automatic generation of code, by integrating in the architecture of the framework the

CORBA middle-ware. Taking as input a description of the behavior of the components, a constraint automaton is computed and the corresponding code generated. A solution based on the concept of proxy introduced in [1] is used but specific to our case to relay messages from the CORBA side to the connector side, represented by the executed automaton.

In our future work, we would like to extend this approach to support the exchange of objects, serialization and other concepts related to the object oriented programming.

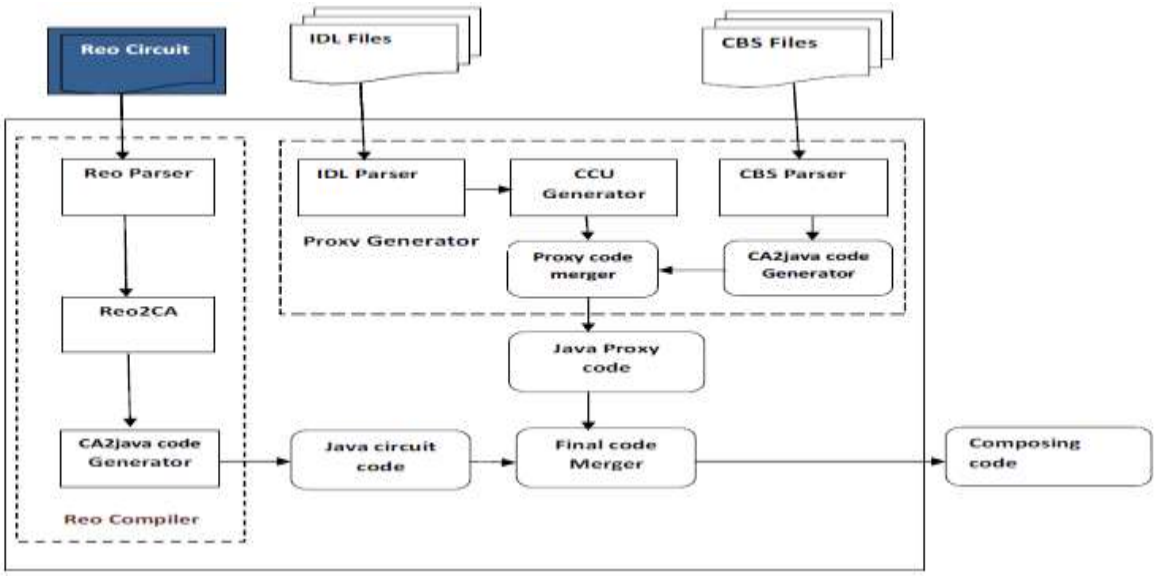

Figure 4: Architecture of the code generation framework

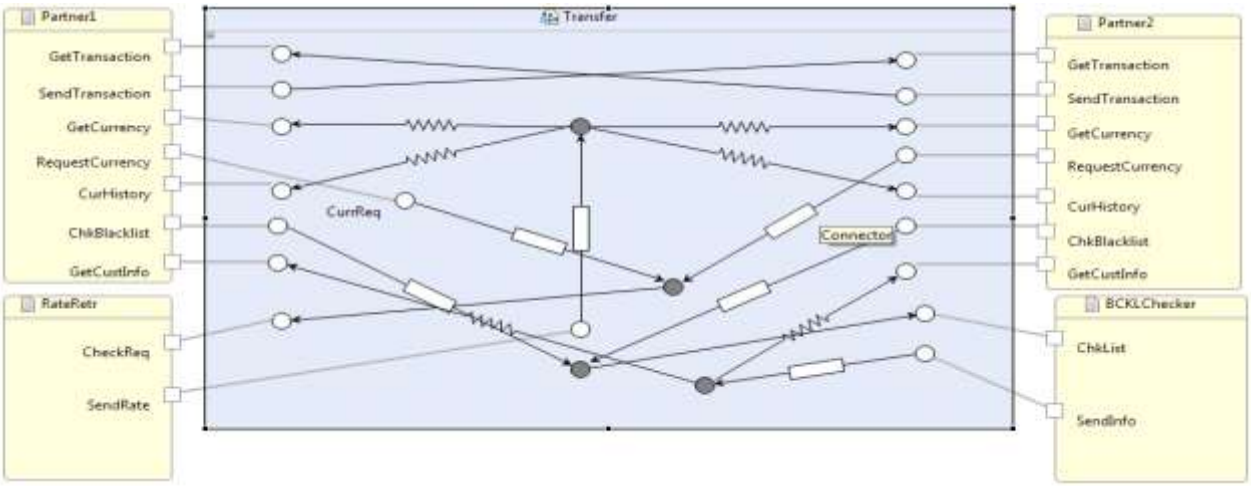

Figure 5: Connector representing the sequential coordination of four heterogeneous components with Reo

Partner1: a C++ component representing the office that sends a new transaction at a given time.
1. It sends a request to the BCKList by calling "ChkBlackList" to ask for verification in the European Union blacklist to see if the name of the customer is present.
2. It waits for the information about the customer from BCKList (a Boolean value)
3. It sends a request to RateRetr by calling "RequestCurrency" for the actual rate of the remote currency.
4. It waits for the Currency information.
5. It sends the transaction to the destination (Partner2) by calling "SendTransaction".

Partner2: a Java component representing the office that receives the transaction at a given time
1. Keeps a history of the rate corresponding to a particular transaction by calling "GetCustInfo". It gets the information about the rate from RateRetr, when it sends back the rate by calling "SendRate" to Partner1.
2. Receives the transaction by calling "GetTransaction"

RateRetr: a Java component that connects to the XE.com system to retrieve the actual rate of a given currency.
1. Gets the request sent by Partner1 for the rate by calling "ChkReq" (it includes information on the currency)
2. Sends the corresponding rate to Partner1 and Partner2 by calling "SendRate"

BCKList: a Perl script component that connects to the EU system to check if a customer is blacklisted.
1. Gets the request sent by Partner1 for the information on the customer by calling "ChkList".
2. Replies to the sender (Partner1) with a Boolean value by calling "SendInfo".

Figure 6: Description of the services

. ACKNOWLEDGMENT

We are grateful to Sung-Shik Jongsman of the CWI, Amsterdam for his particular support and comments in writing this paper.

REFERENCES

[1] Sung-Shik T. Q. Jongmans, Francesco Santini, Mahdi Sargolzaei, Farhad Arbab and Hamideh Afsarmanesh, "Automatic Code Generation for the Orchestration of Web Services with Reo," *Lecture Notes in Computer Science, Volume 7592/2012, pp. 1-16*, 2012.

[2] Farhad Arab, Christel Baier, Jan Ruttena, Marjan Sirjania, "Modeling Component Connectors in Reo by Constraint Automata," *In Proceedings of FOCLASA, the Foundations of Coordination Languages and Software Architectures, a satellite event of CONCUR, 2003*.

[3] Sung Men, "Connectors as Designs: The Time Dimension," *Sixth International Symposium on Theoretical Aspects of Software Engineering, pp.201-208, 2012*.

[4] Baier C., "Probabilistic models for Reo Connector Circuits," *Journal of Universal Computer Science., 11(10), pp. 1718-1748, 2005*.

[5] Farhad Arbab, "Probabilistic Reo: a channel-based coordination model for component composition," *Journal of Mathematical Structures in Computer Science archive, Volume 14 Issue 3, pp. 329 - 366, June 2004*.

[6] Farhad Arbab, "Coordination for Component Composition," *Proceedings of the International Workshop on Formal Aspects of Component Software, FACS 2005*.

[7] Christian Koehler, Farhad Arbab, and Erik P. de Vink, "Reconfiguring Distributed Reo Connectors," *Lecture Notes in Computer Science, Volume 5486, pp. 221, Berlin Heidelberg, 2009*.

[8] D. Clarke, D. Costa, and F. Arbab, "Connector colouring I: Synchronisation and context dependency," *Science of computer programming vol. 66 issue: 3, pp. 205-225, North-Holland, 2007.*

[9] Vinoski, S., "CORBA: integrating diverse applications within distributed heterogeneous environments," *IEEE Communications Magazine, Vol. 35, Issue:2, pp. 46-55, IONA Technol. Inc., USA, 2007*

[10] Laurent Bussard, "Towards a pragmatic composition model of CORBA services based on AspectJ," *In Workshop on Aspects and Dimensions of Concerns (ECOOP, June 2000.*

[11] Sung-Shik T.Q. Jongmans, Farhad Arab, "Correlating Formal Semantic Models of Reo Connectors: Connector Coloring and Constraint Automata," *ICE, Vol. 59, pp. 84-103, 2011*.

[12] Gagandeep singh, "Design Challenges of Load Balancing-CORBA Architecture," *International Journal of Advanced Research in Computer Science and Software Engineering, Volume 2, Issue 7, July 2012.*

[13] Juan Carlos Cruz, Stphane Ducasse, "A Group Based Approach for Coordinating Active Objects," *IN Proceeding Proceedings of the Third International Conference on Coordination Languages and Models, pp 355 – 370, 1999*

[14] Damián Arregui, Fançois Pacull, Michel Riviere , "Heterogeneous Component Coordination: The CLF Approach," *In Proceedings of the 4th International conference on Enterprise Distributed Object Computing, pp. 194 – 203, 2000*

[15] David Safranek, "Heterogeneous Coordination Models," *Electronic Notes in Theoretical Computer Science 180, pp. 107–121, 2007*

Herve Kabamba Mbikayi
Guest Reasearcher at Leiden Institute of Advanced Computer Science (LIACS), Leiden University, The Netherlands
Research and Teaching Assistant at Institut Superieur de Commerce de Kinshasa, Kinshasa, Dem. Rep. of Congo

He is member of the Leiden embedded System Research Group (LERC) of the Leiden University and PHD student at the same institution. He is in possession of a Bac+ 5 graduate diploma in computer science from the University of Kinshasa since 2007 and had been working in the IT industry for more than 10 years as a Consultant in IT security, Manager and Director of Information and Technology departments.

His more recent works was as Head of IT department at MONEYTRANS Group international, Technical Director at Advanced Center for Information and Technology (ACIT) based in Congo.

He also possess many IT vendors' certifications including Cisco Certified Network Associate (CCNA), Cisco certified Network Professional (CCNP), Cisco certified Pix and ASA, etc.