A Simple Solution for Information Sharing in Hybrid Web Service Composition

Brahmananda Sapkota, Camlon H. Asuncion, Maria-Eugenia Iacob, Marten van Sinderen
Centre for Telematics and Information Technology
University of Twente
The Netherlands
{b.sapkota, c.h.asuncion, m.e.iacob, m.j.vansinderen}@utwente.nl

***Abstract*—In hybrid service composition approaches, processes are used to describe the core part of the composition logic whereas the rules are used to specify decision making constraints and conditions. These rules are exposed as services and are used in the processes whenever a certain decision has to be made. To evaluate these rules, we need a mechanism 1) to share data between the main process and the rule service; 2) to decide upon which service to invoke based on the result received from the rule service. In this paper, we propose a tuple space based solution for supporting these requirements. We also include descriptions of an application scenario to motivate our work, a prototype of the proposed solution and we demonstrate how our solution helps in achieving process flexibility with minimal maintenance costs in the context of changing requirements.**

I. Introduction

In modern enterprises, business processes are used to describe ways of achieving business goals by specifying a sequence of activities that need to be performed in order to achieve them. The sequence of activities can be specified imperatively by means of control and data flows, in languages such as Business Process Execution Language (BPEL) [1]. Business processes described in such a manner are supported by means of software systems. This facilitates the separation of business process logic from business applications, consequently allowing changes into business processes with minimal impact on business applications. Additionally, this also allows business managers and application developers to focus on their own areas of expertise independently from each other.

However, there are several challenges that a business enterprise should deal with. The requirements, business rules, processes as well as the offered service functionality may change. Furthermore, even the business goals may change as a result of changes in the external environment or in other processes [2]. These changes can be planned or unplanned. The planned changes are known a priori, at design time, and can be dealt with. In contrast, the unplanned changes are not known at design time and can only be handled at run time. It can also happen that changes of requirements, rules processes or services influence or demand changes of other aspects as well (e.g., implementation). Such effects are not desirable and should be minimised as much as possible. In addition, new functionality may need to be added, some existing functionality may need to be removed or modified from already existing service. This is particularly the case when changes in the environment or the requirements are introduced. It might even be required to provide variations of the same service due to variations in requirements coming from different users of that type of service. This results in service evolution. Providing support to such an evolution requires a flexible service composition approach which allows us to configure composed services to satisfy requirements of specific users [3]. This brings us to the main research question addressed in this research, namely: *How to define a run-time service composition approach that can capture and incorporate new and evolving business requirements also after a serviced composition is specified by means of a business process at design time?*

Following an idea we proposed in our earlier work [4], this paper argues that, in order to deal with the challenges discussed above, we need to use a hybrid service composition approach by combining complementary features of business process and business rules approaches. Hence, we must specify the stable parts of the service composition (i.e., the service orchestration) in terms of a business process, while the variable parts of the service composition must be separated from the main process specification, to increase maintainability and reuse of rules, which is otherwise not possible. These variable parts are represented as decision activities in the business process and are specified separately as business rules. Rules are stored persistently and are encapsulated in so-called rule services. Thus, a decision activity is seen as an invocation of a decision service, which, in turn, is accessing rule services [5] in order to take a decision with respect to a particular run-time composition.

To deal with some of the challenges discussed above, in our earlier work [4], we defined a hybrid service composition approach by combining complementary features of business process and business rules approaches. We separate business rules from the main process to increase maintainability and reuse of rules, which is otherwise not possible in existing approaches. The separated rules are stored persistently and accessed through so called decision services. Decision services are used as per the SOA [5] principles to access rules. The decision services are invoked whenever some rules have to be evaluated at some decision points in the main process. The

use of decision service provides support for 1) modification of the processes and rules independently; 2) reuse of rules and decision logics; 3) increasing the flexibility of the composition process.

The decision service should evaluate the decision logic in a given context of a particular composition instance. Because the decision service invokes rule services to evaluate rules, data may need to be shared between the main process and the decision service. An important limitation of the approach proposed in [4] is that it does not clearly specify a mechanism for 1) passing required data items to the decision service, and for 2) interpreting the results received from the decision service. One of the possibilities, of course, is to pass data as parameters to address issue 1), but then we will need to know a priori what types of data are used. Issue 2) can be addressed by endowing the decision points with 'some' logic in order to interpret the data. But such a solution will require changes in implementation every time the requirements or the business logic will change. Considering the above drawbacks, we choose to use tuple spaces to support loosely coupled communication between participating services and to provide persistence storage of data. Because of decoupling, decision logic per instance can be handled efficiently by using a tuple space. This is required since the actual control flow decisions are often based on the context of a composition instance.

The rest of the paper is structures as follows. Background information is discussed in Section II. A solution approach for information sharing is presented in Section III. Implementation of the proposed solution is detailed in Section IV. Lessons learned during the implementation are discussed in Section V. Similar existing works are presented in Section VI, and finally Section VIII concludes this paper by mentioning possible future directions.

II. Background and Motivation

In this section, we discuss the background information based on which the solution presented in this paper is proposed.

A. Hybrid Composition

In the service oriented paradigm, services can be combined to create other complex value-added services. A service composition method prescribing the composition logic specifies how these services should be composed. Three different types of approaches are discussed in literature [4], [6], [7], [8], [9] which are either a) process-based, b) rule-based or c) a combination of these two, i.e., a hybrid approach. In process-based approaches, the composition logic is described in terms of business processes or workflows that specify the invocation order of individual services. The invocation order can either be sequential, parallel, choice or join. The logic required to determine such ordering is usually specified in terms of conditional expressions and is built-in the process itself. In rule-based approaches, the entire composition logic is described in terms of rules. Both these approaches have their own limitations and advantages. The former is quite inflexible but is more tractable whereas the latter approach is more flexible and less tractable. In a hybrid composition approach, both process-based and rule-based approaches are used to exploit their positive features in an attempt to make composition logic more flexible. In this paper, processes are used to describe the core and stable part of the composition logic whereas the rules are used to specify decision-making constraints and conditions, which are required in the process. The processes and rules are managed separately, but are used together during execution time.

Thus, the hybrid composition approach provides a mechanism to define composition logic in such a way that the process and rules can be managed separately. This allows for updates of core processes as well as rules, when necessary and is suitable to describe a composition logic that needs to adapt to frequently changing requirements. Since the processes and rules can be changed independently, the underlying implementation code can be updated with minimal effort.

B. Tuple Space

A tuple space provides a medium and mechanisms to support communication between multiple processes, applications or agents, based on the principle of associative addressing [10].

Communication between applications takes place by writing and reading tuples from a tuple space. A tuple is a sequence of typed fields. A typed field is called an *actual field* if it is valued and it is called a *formal field* if it is non-valued. Currently available tuple space implementations follow Gelernter's Linda model [11]. In Linda, the first field is reserved to contain an actual field which is used as the identifier of that tuple. Therefore, an example of tuple according to Linda model, would be (idvalue, String, 20). One of the interesting characteristics of tuple space is that a minimal set of operations is required for parallel or concurrent communication purposes. Linda, for example, provides `out()`, `rd()`, and `in()` operations to write, read and destructively read tuples to and from a tuple space. The `rd()` and `in()` take a so-called *template* as a parameter to retrieve information written in a tuple space. A template is similar to a tuple and is used to specify information search criteria. For a given template, a set of tuples is returned if the following conditions hold true: 1) number of fields in the tuples are equal to that of given template and 2) each field in the tuples matches a corresponding field in the template by value or by type. Therefore, a template (idvalue, "Age", Integer) would be an example of specifying search criteria to look for tuple stored as (idvalue, String, 20) in the tuple space. However, using this template `rd()` or `in()` operation would not return, for example, a tuple stored as (idvalue, String, 20, Boolean) in the tuple space, because the number of fields in this tuple is not equal to that in the template.

Tuple spaces provide a simple but powerful means to share information between multiple distributed processes or applications. Because the communication takes place between applications by writing and reading tuples from a tuple space, it supports loose coupling in terms of time, location and reference. The decoupling in time is supported because the communicating applications do not need to be available at

the same time. Communicating applications also do not need to be at the same location and hence, the location decoupling. Furthermore, they also do not need to know the access address for communication. The only address they need to know is that of the tuple space. Therefore, applications are decoupled in terms of reference. Due to these inherent support for decoupling, tuple space provides a platform well-suited for the integration of loosely coupled systems, e.g., systems in enterprise applications environments.

C. Motivation

An information sharing approach for service composition is required in situations where a possible future recipient is not known *a priori*. We motivate this problem through the following example scenario taken from a home-care application domain. We acknowledge that the scenario presented below is simple. However, it is still sufficient to motivate and illustrate the work presented in this paper.

A "medicine-reminder-and-dispenser" (MRD) service is constructed by composing the "reminder" and the "dispenser" services to provide medicine reminder functionality to people with memory problem helping them take right the medicine at the right time. The reminder service sends a reminder message to a subscribed user at the predefined times (specified at subscription). The dispenser service enables the release of medicines and also monitors whether the medicines have actually been taken. The MRD service uses the functionality of a reminder service to send the reminder message at the right time. It also uses the functionality of a dispenser service to ensure that the recipient of the message takes the right medicine at the right time. To do so, the MRD service repeats the same reminder if the medicine has not been taken by its recipient within a certain time interval (Δt). If the medicine is taken, the MRD service stops sending the same reminder. At a later point in time, the requirement changes in such a way that if the recipient of the reminder message does not take the medicine even after N repetitions of the reminder, an alarm should be raised to seek external help. This additional requirement demands the extension of original functionality offered by the MRD service with the "alarm" functionality offered by, for example, an "alarm" service. Again, at a later point in time, the requirement changes in such a why that the alarm should be raised only for those recipients who are having life threatening health conditions and should not miss their medicine intake.

This type of changing requirements calls for a flexible service composition solution which is cost-effective and easy to maintain and manage. Because the hybrid service composition approaches such as the one described in [4] are aimed at supporting such flexibility, the MRD service can be described using that approach as depicted in Fig. 1.

As shown in Fig. 1, the constraints and conditions that specify when to use the alarm service are externalised as part of the rules, whereas the core logic is specified as process. The externalised rules are connected to the process through the so-called decision activity via the decision service.

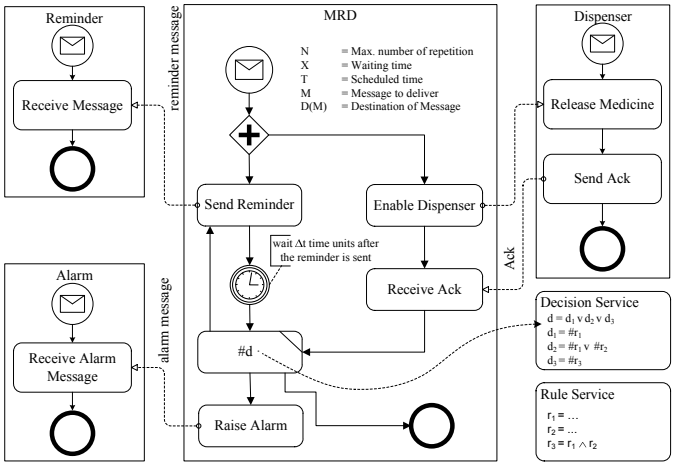

Fig. 1. MRD Service

Information required to make the decision is passed to the decision service which, in turn passes this information to the rule service to ascertain certain constraints and conditions. This would, however, require *a priori* knowledge of the parameter list. In addition, when the decision result is received, the service implementing decision activity needs to further analyse the result to determine which of the alternative services should be invoked. This requires changing the implementation code should the new services have to be added or the existing ones have to be removed. Such a change in implementation code, with each change in requirement, is usually costly and time consuming. Moreover, as the system becomes complex, identifying the right place where new changes have to be adapted becomes a challenging task. Thus, it is necessary to find a mechanism that resolves these issues in a flexible manner.

III. Information Sharing

In this section, we propose an approach for enabling information sharing in hybrid web service composition. Our approach is aimed at supporting flexibility and reducing maintenance costs. The design of the proposed solution is based on the SOA principles and supports loose coupling between participating services. Problems related to information sharing and interpreting decision results as discussed in Section I are addressed by using tuple space.

A. Handling Information Sharing

Information between services is shared by sending them as parameters when invoking the service. In a decision making scenario, this approach is not applicable because the parameter list can be of variable length per decision expression. This would become even worse if new decision expressions are to be introduced, which is usually the case since flexible service composition approaches are aimed at supporting such new additions.

To share information between the service implementing the decision point activity and the decision service, we use a

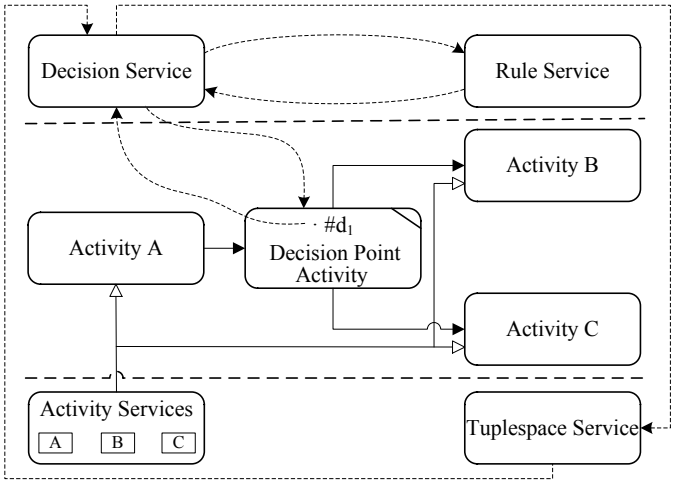

Fig. 2. Information Sharing

tuple space as a common communication medium between these services as shown in Fig. 2. This approach is chosen because the decision service can provide a standard generic interface definition for invocation thereby eliminating the need of defining new interface definition every time a new decision expression has to be added. In the figure, we used different line styles to differentiate between design and implementation parts. The dotted lines are used to indicate the execution time interaction. Arrows with a solid head are used to connect activities of the process. Arrows with an empty head are used to indicate the services that implement the corresponding activities in the process. The *Decision Service*, *Tuple Space Service* and the *Rule Service* are part of the design but not of the process, dotted lines with solid arrow head are used to indicate the interaction between the process and these services during execution time.

The *Decision Point Activity*, represented as a rectangle with a line connecting its top and right boundaries, is used for invoking the *Decision Service* in order to obtain a certain decision result. It refers to the decision deployed in the *Decision Service*. It is used at places, in the process, where a certain decision has to be made. This activity is realised by the *Decision Activity Service*. The *Decision Activity Service* handles the data and the control flow of the decisions. The *Decision Service* allows deployment of decision expressions and acts as a gateway to a *Rule Service* where specific constraints and conditions are defined and deployed as rules. In addition, the *Decision Service* evaluates the requested decision expression and returns the result to the *Decision Activity Service*. The idea behind creating the *Decision Activity Service* is to separate the activities related to the control and data flow from the decision making logic, which is done by *Decision Service*. This allows both services to be encapsulated, modularized and reusable. The *Tuple Space Service* manages the tuple space by exposing tuple space operations as a Web service. The tuple space is used as a medium to share information between interacting services by reading and writing tuples. The *Activity Services* represents various services that realise and are invoked by the various activities of the process.

B. Decision Expression

Decision expressions are used to specify the decision making logic. We define it in terms of rule expressions, i.e., a decision expression consists of a logical expression referring to rules which are deployed in the *Rule Service*. If rules $r_1, r_2, ..., r_n$ are used, for example, to make a certain decision, these rules are grouped using logical connectors in a decision expression d. To increase modularity and reusability, we allow decision expressions to refer to other previously defined decision expressions as well. If decision expressions $d_1, d_2, ..., d_n$ are already defined, for example, these decisions are allowed to be used in a new decision expression d'. Thus, a decision expression can be a combination of rules and/or existing decision expressions. The logical connectors used in the decision expression are AND, OR, XOR and NOT operators.

C. Indexing the Endpoint

The invocation of *Decision Service* results in a decision value. The *Decision Activity Service*, which realises the *Decision Point Activity* is required to identify the endpoint of the next services to be invoked based on the decision value d received from the *Decision Service*. This is the same problem as, for example, opening a particular BPMN [12] gateway. In order to resolve this problem, the *Decision Activity Service* maintains a simple index of endpoints of the *Activity Service(s)* realising the activities which are directly connected to the *Decision Point Activity*. Assuming that the decision value is the name of the next activity to perform, the entries of the index consists of (key, value) pair where key represents the name of the activity whereas the value represents the endpoint of the corresponding *Activity Service*. When the *Activity Service(s)* are implemented, their endpoints are stored in the tuple space by writing a tuple (tupleid, EndPoint). In this tuple, the field tupleid represents the name of the corresponding activity whereas the *EndPoint* represents the endpoint of the *Activity Service* of type URI. The *Decision Activity Service* uses the template (activityName, URI) to read the *EndPoint* of the *Activity Service*, which realises the activity *activityName* from the tuple space. This information is then stored in the endpoint index. Having this index in place, the endpoint of the next service to invoke can be identified by performing a lookup using the decision value as the key.

D. Interaction Between Services

The interaction sequence between the various services is illustrated in Fig. 3. This figure mainly illustrates the information sharing between the *Decision Activity Service* and the *Decision Service*. The interaction begins with the *Decision Activity Service* receiving an input data *i* that will be needed for decision making purposes. Since the decisions are made by the *Decision Service*, this data should flow to the *Decision Service*. To handle this data flow, the *Decision Activity Service*

creates an unique ID (i.e., the U_k) and stores the received data, *i*, into the tuple space, by writing a tuple (U_k), *i*). The unique ID is needed to retrieve the data later on.

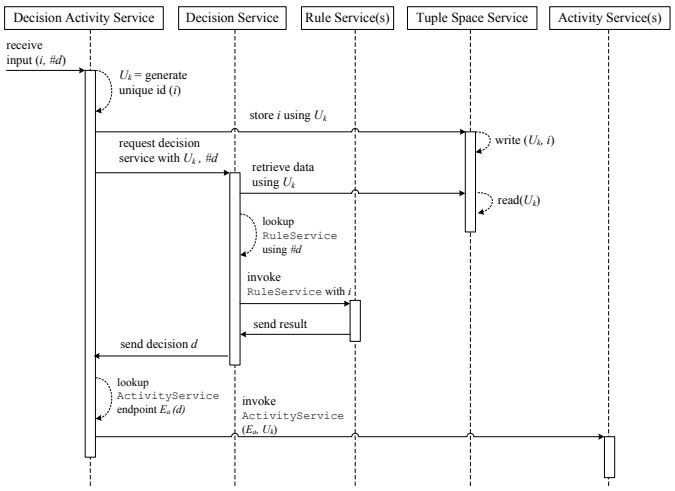

Fig. 3. The order of interaction between various services

After storing the data into the tuple space, the *Decision Activity Service* passes the information (endpoint reference to the *Tuple Space Service*, U_k, decision expression reference *#d*) and the control flow further to the *Decision Service* whose main responsibility is to a) invoke *Rule Service(s)* to evaluate rules used in the decision expression and b) make decision based on the information returned from the *Rule Service*. Upon receipt of the control flow, the *Decision Service* retrieves data *i* which is necessary for decision making from the tuple space using the tuple ID (U_k). The *Decision Service* may perform additional decision processing such as further processing the results when several rules have to be evaluated and to arrive at a correct decision *d* before returning the results back to the *Decision Activity Service*. When the decision result is received, the *Decision Activity Service* determines the correct entry point (E_a) of the next service to invoke based on the received decision result *d*, and then invokes the correct *Activity Service(s)*.

IV. Implementation

In this section, we describe the implementation details of the proposed solution. We first provide a general overview of the implementation solution. We then apply the solution to the application scenario described earlier in Section II-C. Finally, we show how the solution adapts when changes are made to the application scenario.

A. Implementation Overview

To evaluate the usability and feasibility of the proposed approach, a prototype is implemented in Java. We use Axis2 as the Web service engine and Tomcat 6 as the standard container for deploying services. We implement the *Tuple Space Service* using JavaSpaces technology — a simple yet powerful tool for a scalable, high-performance, fault-tolerant, and elegant coordination of distributed processes [13]. In JavaSpaces, a tuple specified as a class that implements the `net.jini.core.entry.Entry` class. For our purpose, we implement the `Entry` class using the `TupleEntry` class as shown in Listing 1. It takes two variables: `id` to indicate the tuple id (denoting U_k in Fig. 3) and a `java.util.Object` called `content` (denoting `i` in Fig. 3). The tuple id U_k is used to uniquely identify tuples from the tuple space. We use Java to generate an immutable and universally unique identifier (UUID) string for generating this unique id.

```
import net.jini.core.entry.Entry;

public class TupleEntry implements Entry{
    public String id = null;
    public Object content = null;
    public String getId(){
        return id;}
    public void setId(String id){
        this.id = id;}
    public Object getContent(){
        return content;}
    public void setContent(Object content){
        this.content = content;}
}
```

Listing 1. The `TupleEntry` implementation of JavaSpaces `Entry`.

We design the *Tuple Space Service* to implement four of the several methods available from `net.jini.space.JavaSpace`: `write()`, `take()`, and `read()`. An example of the `write()` implementation is shown Listing 2. A `TupleEntry` object called `entry` is created to store the `id` and `content` of the tuple passed as a `write` object. For simplicity, we implement the JavaSpace object, `space`, as a singleton so that there is only one space that can be accessed at any given time (although JavaSpaces allows the creation of several spaces). The `write()` method can then be invoked passing the entry object as the tuple. Again, to keep the implementation simple, we do not use a transaction manager (which gives transaction atomicity for a group of space operations); hence, a `null` value as the second argument. The `Lease.FOREVER` code indicates the amount of time that the entry will be stored in the space before it is removed (in this case, indefinite).

```
public WriteResponse write(Write write) {
    ...
    TupleEntry entry = new TupleEntry();
    entry.setId(write.getTupleID());
    entry.setContent(write.getContent());
    ...
    JavaSpace space
        = SpaceSingleton.getInstance();
    space.write(entry, null, Lease.FOREVER);
    ...
}
```

Listing 2. An implementation of the `write()` method of the `Tuple Space Service`.

We implement the *Decision Service* to maintain the decision expression and for decision-making purposes. The decision expressions are stored, for the purpose of this prototype, in a `decision-expression.xml` file. A sample decision expression is given in Listing 3 which specifies that `ruleName`

should be invoked using its endpoint based on the given decision reference.

```
<?xml version="1.0" encoding="UTF-8"?>
<decision-expression xmlns:xsi="http://www.w3.org/2001/
     XMLSchema-instance">
   <decision name="http://example.org/DecisionService/deci-
        sion#1">
     <rule name="ruleName" endpoint="http://localhost:8080/
          axis2/services/RuleService"/>
   </decision>
</decision-expression>
```

Listing 3. An example decision expression.

The *Decision Service* evaluates the decision expression referred by (#d) which is supplied together with data items when the *Decision Service* is invoked. Although the decision reference (#d) can take any string for identification purposes, we format it to take a form of a namespace in our implementation. We use the Java Architecture for XML Binding (JAXB) for serializing XML [14] as Java objects. In Listing 3, there is only one decision reference and one rule as its child element. Thus, the *Decision Service* invokes the ruleName rule service using its endpoint. However, it should be noted that a decision expression file may have several decisions with several rule services combined logically in different ways.

We use Java Expert System Shell (Jess) [15] as the technology for implementing executable rules. The main advantage of using Jess is that it is tightly integrated with Java so that Jess rules can simply be manipulated as regular Java objects. A typical *Rule Service* encapsulates an atomic Jess rule exposed as a Web service.

To support identification of the services that need to be invoked after the decision has been made, we define a simple index and store it as an activity-services.xml file. A sample decision index is given in Listing 4, which specifies that given a name serviceA, its endpoint can be returned.

```
<?xml version="1.0" encoding="UTF-8"?>
<activity-services xmlns:xsi="http://www.w3.org/2001/
     XMLSchema-instance">
    <service name="serviceA">
       "http://localhost:8080/axis2/services/ServiceA"
    </service>
</activity-services>
```

Listing 4. An example service index.

This index is maintained by the *Decision Activity Service* and used when necessary to determine the correct endpoint of the correct *Activity Service(s)*.

B. Implementing the Application Scenario

We now describe how the generic solution described in the previous section can be applied to the application scenario. Fig. 4 shows the invocation order of the different services in the MRD application scenario. The interaction begins with *Remind Service* invoking the *Decision Activity Service* and passing the MRD Repeat Request input data and the decision reference decision#1. The *Decision Activity Service*, thereafter, creates a unique id at runtime, U_k, for MRD Request and passes these to the *Tuple Space Service* for space storage using the write() operation. In this case, the generated id is "74f4cf40-6ca0-449d-b015-1cd1414727a4", and the decision reference is "http://dyscotec.is.utwente.nl/Decision-Service/decision#1". The *Decision Activity Service* now passes the control to the *Decision Service*.

The *Decision Service* now calls the *Tuple Space Service* to retrieve the MRD Request object using the unique id. Next, the *Decision Service* unmarshalls the decision-expression.xml to retrieve the appropriate rule service endpoints that will be invoked later. For this scenario, two rule services will be used with the names MRDRule and MRDRepeatRule as shown in Listing 5. Thus, the *Decision Service* will first invoke the *MRD Rule Service* using the endpoint from the decision-expression file passing the MRDRequest object retrieved earlier from the tuple space.

```
<decision-expression xmlns:xsi="http://www.w3.org/2001/
     XMLSchema-instance">
   <decision name="http://dyscotec.is.utwente.nl/Decision-
        Service/decision#1">
     <rule name="MRDRule" endpoint="http://localhost:8080/
          axis2/services/MRDRuleService"/>
     <rule name="MRDRepeatRule" endpoint="http://localhost:
          8080/axis2/services/MRDRepeatRuleService"/>
   </decision>
</decision-expression>
```

Listing 5. The decision expression of decision#1.

The *MRD Rule Service* is a Web service that exposes a Jess rule called MRDRule as shown in Listing 6. Its role is to check if an acknowledgement from the dispenser has not yet been received, and that the medicine time is equal to the current time. If so, the rule fires and returns a boolean value of true to send a reminder and enable the medicine dispenser. This response is specified as a simple Java bean called MRD Response as shown in Fig. 4.

```
(assert (ack (fetch ACK-VALUE)))
(bind ?mt (fetch MED-TIME))
(bind ?ct (fetch CUR-TIME))

(if (= ?mt ?ct) then (assert (remind)))

(defrule MRDRule
   ?a <- (ack false)
   ?r <- (remind)
   =>
      (store SEND-REMINDER true)
      (store ENABLE-DISPENSER true)
      (retract ?a)
      (retract ?r))
```

Listing 6. MRDRule exposed as the MRD Rule Service in Jess.

One of the roles of the *Decision Service* is to make further decisions based on the results returned after invoking appropriate rule services. It can either choose to invoke more rule services or to end the execution of the process. In the given scenario, when the *MRD Rule Service* returns a value of true for isSendReminder and isEnableDispenser, then the *Decision Service* informs the *Decision Activity Service* about the need to invoke *Send Reminder Service* (otherwise, if the rule does not fire, the *Decision Service* does not need to

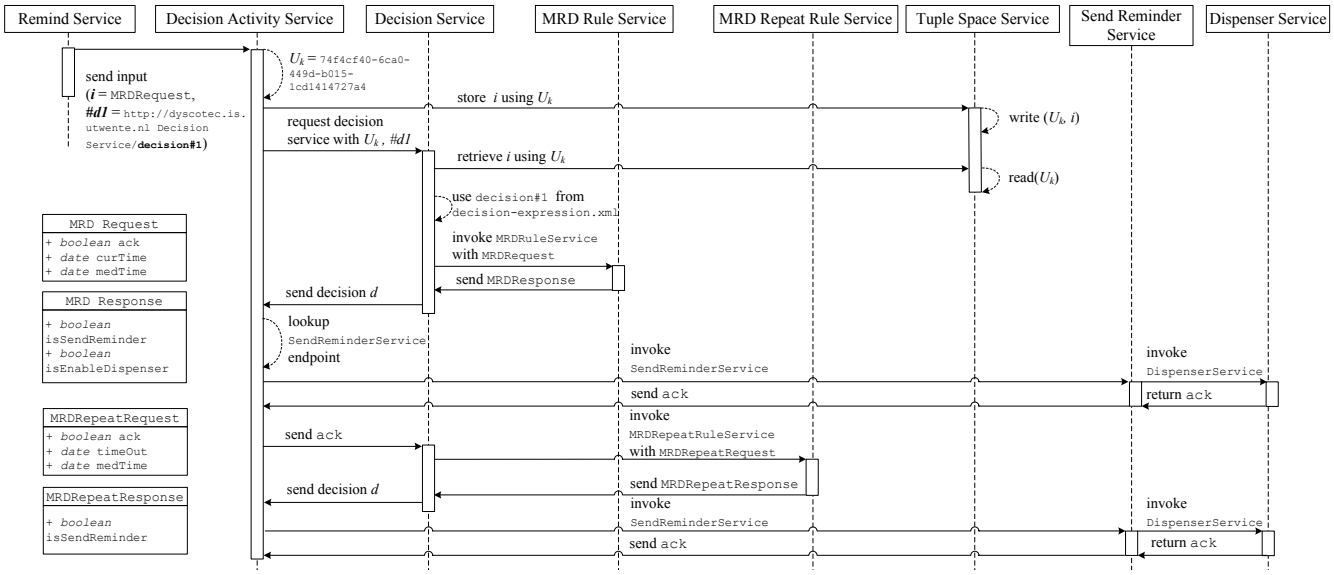

Fig. 4. Interaction between services to realise MRD service

do further actions). To do this, the *Decision Activity Service* looks up the endpoint of the *Send Reminder Service* from the `activity-services.xml` file and thereafter invokes it. The *Send Reminder Service*, thereafter, invokes the *Dispenser Service* to automatically enable the medicine dispenser.

If the *Decision Activity Service* is not able to receive an acknowledgement from the *Send Reminder Service* within a given time frame, then it informs the *Decision Service* about this. The *Decision Service*, in turn, calls the next rule service based on the `decision#1`, i.e., the *MRD Repeat Rule Service*, to remind the patient again of the need to take his medicine. The *MRD Repeat Rule Service* is a Web service that encapsulates the Jess rule `MRDRepeatRule` as shown in Listing 7.

```
(assert (ack (fetch ACK-VALUE)))
(bind ?ct (fetch CUR-TIME))
(bind ?to (fetch TIMEOUT))

(if (= ?ct ?to) then (assert (remind)))

(defrule MRDRepeatRule
   ?a <- (ack false)
   ?r <- (remind)
   =>
      (store SEND-REMINDER true)
      (retract ?a)
      (retract ?r))
```

Listing 7. `MRDRepeatRule` exposed as the *MRD Repeat Rule Service* in Jess.

The role of the *MRD Repeat Rule Service* is to remind the patient again to take his medicine if no acknowledgement has been received from the dispenser given a certain timeout; i.e., the rule fires, sending an `isSendReminder` of `true`, when the `ack` is `false` and the `medTime` and the `timeOut` are equal. The rule service returns `true` if a reminder is to be sent again. The *Decision Service* also counts the number of times the reminder has been sent. Finally, if the patient has taken his medicine from the dispenser, the *Dispenser Service* returns an acknowledgement to the *Send Reminder Service*, and, thereafter, to the *Decision Activity Service*. The process ends after this.

C. Adapting to Scenario Changes

We now describe the implementation solution when changes to the application scenario are introduced. As previously described, enabling the dispenser may not be enough. The patient, for example, could be in a serious condition that prevents him from taking his medicine on time. This may be a hazardous situation. The functionality of the MRD service can be extended to seek external help (e.g., from a volunteer or a health care professional) in a situation like this.

For this purpose, we introduce a new rule service called *Alarm Rule Service* that is raised when a patient continuously ignores a reminder for some number of times. The *Alarm Rule Service* can be used to invoke the *Alarm Service* that automatically sends an alert (for example, through a text message or an automated phone call) for external assistance. This new rule service can be added in conjunction with the two previously defined rule services (i.e., the *MRD Rule Service* and the *MRD Repeat Rule Service*).

To implement this new rule service easily, a new decision index can be added, hereafter called `decision#2`, to the `decision-expression.xml` as shown in Listing 8. In this new decision index, the previous rules are reused while adding a new rule called the `AlarmRule`.

```
<decision-expression xmlns:xsi="http://www.w3.org/2001/
     XMLSchema-instance">
  <decision name="http://dyscotec.is.utwente.nl/Decision-
       Service/decision#1">
    <rule name="MRDRule" endpoint="http://localhost:8080/
         axis2/services/MRDRuleService"/>
    <rule name="MRDRepeatRule" endpoint="http://localhost:
         8080/axis2/services/MRDRepeatRuleService"/>
  </decision>
```

```
<decision name="http://dyscotec.is.utwente.nl/Decision-
    Service/decision#2">
  <rule name="MRDRule" endpoint="http://localhost:8080/
      axis2/services/MRDRuleService"/>
  <rule name="MRDRepeatRule" endpoint="http://localhost:
      8080/axis2/services/MRDRepeatRuleService"/>
  <rule name="AlarmRule" endpoint="http://localhost:8080/
      axis2/services/AlarmService"/>
 </decision>
</decision-expression>
```

Listing 8. Updated decision expression file.

The *Alarm Service* encapsulates the `AlarmRule` in Listing 9. Its role is to send an external alarm message in case the patient has ignored reminders to take his medicine for a certain number of times.

```
(assert (ack (fetch ACK-VALUE)))
(bind ?c (fetch COUNT))
(bind ?rc (fetch REMINDER-COUNT))

(if (>= ?rc ?c) then (assert (alarm)))

(defrule AlarmRule
   ?a <- (ack false)
   ?r <- (alarm)
   =>
      (store SEND-ALARM true)
      (retract ?a)
      (retract ?r))
```

Listing 9. `AlarmRule` exposed as the `Alarm Service` in Jess.

Using the `Alarm Request` object, the rule fires when the dispenser has not sent an acknowledgement, and the number of times a patient has been reminded is equal to the maximum number of reminders. The service returns an `Alarm Response` object with a boolean value of `isSendAlarm` as shown in Fig. 5.

When the *MRD Request Rule Service* repeatedly sent reminders for a certain maximum number of times, and still the *Dispenser Service* has not returned an acknowledgement, the *Decision Activity Service* informs the *Decision Service* about this. Thereafter, the *Decision Service*, invokes the *Alarm Rule Service* to fire the Jess `AlarmRule` (shown in Listing 9). If the `AlarmReponse` returns a value of `true` for `isSendAlarm`, then the *Decision Service* informs the *Decision Activity Service* that the *Alarm Service* needs to be invoked. The *Decision Activity Service*, thereafter, looks up the endpoint of the *Alarm Service* from the `activity-services.xml` and invokes it. After this step, the process ends.

V. Discussion

Hybrid service composition approaches designed so far focus on control flow aspects while defining a composition logic. However, in these approaches, information sharing between services is equally important because the rules are evaluated outside of the processes. We aim at contributing to this area by providing a tuple-space-based approach for sharing information between rules and processes via decision services. The proposed approach allows sharing of information in a generic way. This means that the *Decision Service* has to provide only one generic access interface because the information is shared by writing and reading it to the tuple space.

The proposed solution is flexible in the sense that new services can be added or old ones can be removed from the already existing service at a minimal cost. The addition or removal can be done by updating the decision expressions and rules, and with minimal changes to the underlying process specification and implementation code. The information sharing can be handled seamlessly because of the use of the tuple space. We tested this feature through the implementation of a prototype.

One of the difficulties we encountered when implementing this solution using an execution language such as BPEL is that the *Decision Service* decides at runtime which *Activity Service* to invoke next based on the current values returned by the *Rule Services*. As described previously, the control flow logic is specified mainly by the `decision-expression.xml`. It is difficult to implement this solution in BPEL since the control flows need to be determined *a priori* during design time.

An additional implementation difficulty when using BPEL is caused by its *Assign* activity. This is especially true when it is used to design the *Decision Service*. An important design requirement for the *Decision Service* is that it must be able to receive and send any data object in order for it to be reusable. With BPEL, however, data objects must be known at design time in order for the *Assign* activity to successfully transform values between BPEL activities (e.g., invoke, reply, receive). Because of these limitations, we based our implementation on the service layer by directly embedding control flows between service invocations. We intend to improve our approach by experimenting with other standard tools and languages, and by trying to address the BPEL related difficulties as well.

As already mentioned, a major challenge in the implementation is making the *Decision Service* operations as generic as possible in terms of receiving and sending data of any type. Following [16], our initial solution has been to treat all messages as a Java `String` type. This ensures interoperability between non-Java implementations. Using JAXB, a Java object is first marshalled into an XML String before Web service `A`, for example, can send it to Web service `B`. Conversely, Web service `B` unmarshalls the XML string into a Java content tree before the object can be used for further business logic processing. However, this brings a significant disadvantage: both Web services must have common knowledge of the actual type of object they are exchanging. This is especially important when upcasting or downcasting Java objects.

Currently, the focus of the implementation solution has largely been within the service layer; i.e., control flows are directly embedded between service invocations. Ideally, one should be able to "lift" the specification of the control flow to a more abstract layer, perhaps through the use of a business process modeling language instead of an execution language. We are currently exploring the possibility of using BPMN to specify the control flows. In such a case, BPMN activities, (at a higher layer), would be implemented using more concrete, encapsulated, modularized and reusable Web

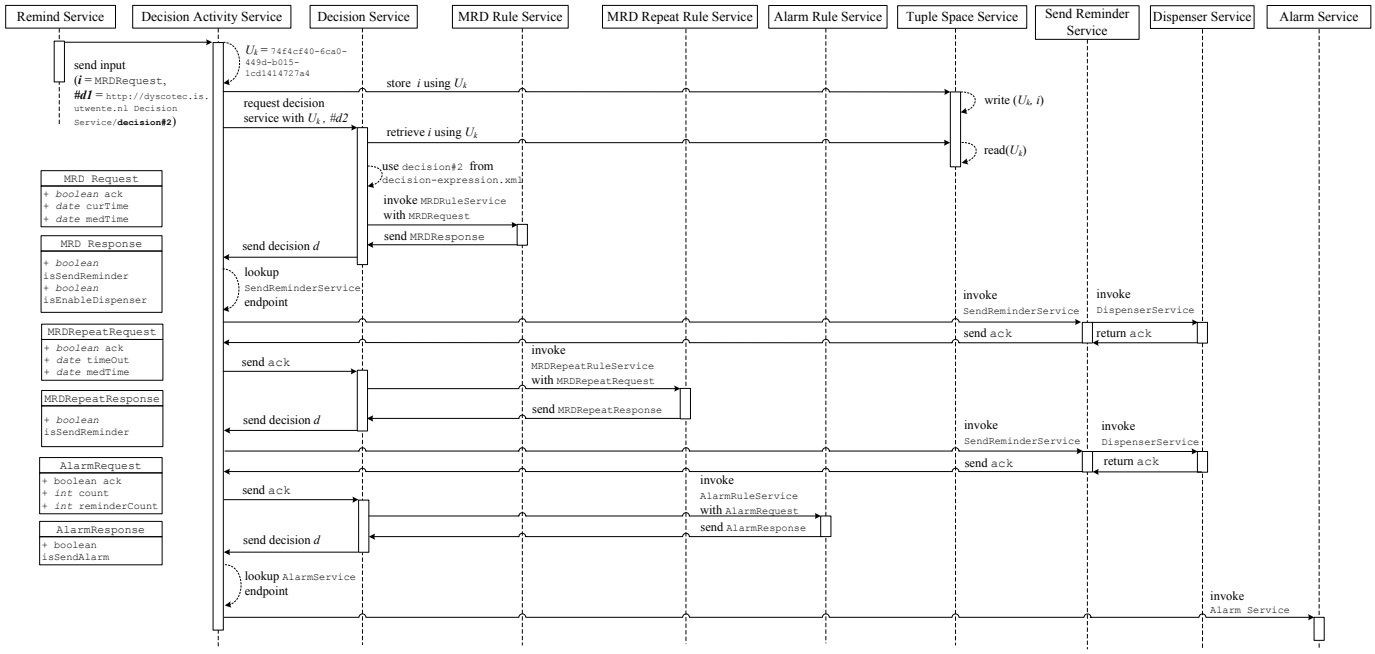

Fig. 5. Interaction between services to realise MRD with Alarm service

services. This is not currently implemented, but, we already see some promising tools/approaches that can achieve this (e.g., BizAgi[1], Mendix[2]).

VI. Related Works

Hybrid service composition approaches, which combine business processes and rules, are aimed at providing an efficient and flexible mechanism for creating application services. Some of the closely related works that are also motivated by the flexibility requirements for service composition are presented in the following.

In [17], an automatic service composition approach for data-providing services is proposed. In their approach, data-providing services are modeled as RDF [18] views over a mediated domain ontology. This ontology is used to describe a relationship between input and output parameters of the services. A query rewriting approach is used to transform user queries into a composition of data providing services. However, the proposed mechanism is mainly targeted to data-providing services, which are special types of services. Data-providing services are considered to have no precondition and do not cause effects after their invocation.

A dynamic web service composition approach that considers operational flow semantics is presented in [19]. In the proposed approach, an abstract composition plan is generated based on the graph flow concept. This abstract plan is then refined by selecting suitable candidate web services and their operations. The flow semantics is used to define a dependency between operations. Unlike our solution, this approach does not clearly specify how information is shared between participating services.

A Tuplespace-based approach for synchronising control flow in workflow management systems is presented in [20]. The basic tuplespace model is extended to handle multiple tuple matching in a single operation and to support join operations to synchronise concurrent threads of control flow. This work focuses mainly on control flow synchronisation whereas information sharing is the main focus of our work.

In [7], a hybrid approach for web service composition is presented. In their approach, business processes are used for describing a core part of the composition logic and the rules are used for defining policy-sensitive aspects of the composition. These rules and processes are kept separately to reduce complexity and to increase adaptability. This approach emphasises a combined usage of business processes and business rules. The information sharing part, which is required for evaluating the rules, is however poorly specified.

A combination of SOA, BPEL and ontologies is considered in [21] to improve maintainability and achieve flexibility of knowledge intensive business processes. The variable tasks are described more abstractly and stored in a task pool during design time whereas their flow is defined during run time. Ontologies and business rules define these abstract parts to support a selection of execution of tasks during run time. Unlike our approach, information sharing is done through a database which would require frequent updates to the database schema if new decisions or rules are required to be added. This would result in increased maintenance costs.

The work presented in [22], propose an approach to customise a business process to a particular case of usage. In their work, authors utilise business rules and workflow patterns to

[1]http://www.bizagi.com/
[2]http://www.mendix.com/

model variable parts of a process flow to support dynamic pattern composition. These workflow patterns are identified and implemented in business rules similarly to the approach proposed in [21]. The rules are kept separately from the process. The focus is on isolating the parts of the process that are likely to change from the rest of the process. A solution to incorporate the isolated process parts and information sharing between them is not clearly defined. In [23], an approach is presented to specify Web service choreography by exploiting business processes and business rules. Their work follows meta-modelling approach and focuses on achieving implementation consistency and behaviour compatibility instead of information sharing between participants.

VII. Conclusions

Our research demonstrates how hybrid service composition allows us to achieve flexibility and reusability of existing services when creating complex composite services to satisfy frequently changing user requirements. This type of approach has the ability to respond to the changing user requirements due to the separation of processes and rules. Current approaches either do not provide adequate support for sharing required information between services or the offered solutions are too restrictive in terms of cost and time required for responding to the change in requirements. This is mainly because the underlying implementation needs to be changed to facilitate information sharing between services when new requirements have to be accommodated.

Extending on our previous work [4], we have proposed an approach for facilitating information sharing using tuple space. By using tuple space, we aim at minimising the maintenance cost due to the addition of new requirements. Information is shared between services by writing and reading tuples in the tuple space.

The proposed solution is implemented in Java and the usability of the proposed approach is illustrated with the MRD application scenario from homecare domain. The current implementation supports AND and OR operators. Providing support for other operators is left as part of our future work. The singleton implementation of JavaSpace is aimed at showing that the proposed solution is feasible. In order to achieve improved scalability and performance, the implementation will be extended to support multiple Tuple Space instances. In the current implementation, the links between services are hard coded. We aim to control this connection via some execution engine, for example, BPEL4WS [1]. We further aim at developing application scenarios from other domains to further test the applicability of the proposed approach.

VIII. Acknowledgements

This material is based upon works jointly supported by the IOP GenCom U-Care project (http://ucare.ewi.utwente.nl) sponsored by the Dutch Ministry of Economic Affairs under contract IGC0816 and by the DySCoTec project sponsored by the CTIT, University of Twente.

References

[1] M. B. Juric, B. Mathew, and P. Sarang, *Business Process Execution Language for Web Services : BPEL and BPEL4WS*. PACKT Publishing, 2004.

[2] M. Hammer and J. Champy, *Re-engineering the Corporation, A Manifesto for Business Revolution*. New York: Harper Business, 1993.

[3] M. van Sinderen and J. P. A. Almeida, "Empowering Enterprises through Next-Generation Enterprise Computing (Editorial)," *Enterprise Information Systems*, vol. 5, no. 1, pp. 1–8, February 2011.

[4] B. Sapkota and M. van Sinderen, "Exploiting Rules and Processes for Increasing Flexibility in Service Composition," in *Proc. of the Fourteenth IEEE International Enterprise Distributed Object Computing Conference Workshops*, 2010.

[5] T. Erl, *Service-Oriented Architecture Concepts, Technology, and Design*. Prentice Hall Professional Technical Reference, 2005.

[6] F. Curbera, R. Khalaf, N. Mukhi, S. Tai, and S. Weerawarana, "The Next Step in Web Services," *Communications of the ACM*, vol. 46, no. 10, pp. 29–34, 2003.

[7] A. Charfi and M. Mezini, "Hybrid Web Service Composition: Business Processes Meet Business Rules," in *Proc. of the 2nd International Conference on Service Oriented Computing (ICSOC)*, 2004, pp. 30–38.

[8] B. Orriens and J. Yang, "A Rule Driven Approach for Developing Adaptive Service Oriented Business Collaboration," in *Proc. of IEEE International Conference on Services Computing*, 2006, pp. 182–189.

[9] C. H. Asuncion, M.-E. Iacob, and M. J. van Sinderen, "Towards a Flexible Service Integration through Separation of Business Rules," in *Proc. of the 14th IEEE International EDOC Enterprise Computing Conference (EDOC 2010)*, 2010, pp. 184–193.

[10] D. Gelernter, "Generative Communication in Linda," *Proc. of ACM Transactions on Programming Languages and Systems*, vol. 7, no. 1, pp. 80–112, January 1985.

[11] D. Gelernter and N. Carriero, "Coordination Languages and their Significance," *Communications of the ACM*, vol. 35, no. 2, pp. 97–108, 1992.

[12] S. A. White, "Introduction to BPMN," Object Management Group (OMG), Tech. Rep., 2004.

[13] E. Freeman, K. Arnold, and S. Hupfer, *JavaSpaces Principles, Patterns, and Practice*, 1st ed. Essex, UK: Addison-Wesley Longman Ltd., 1999.

[14] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau, Eds., *Extensible Markup Language (XML) 1.0 (Fifth Edition)*. W3C Candidate Recommendation, November 2008.

[15] E. F. Hill, *Jess in Action: Java Rule-Based Systems*. Greenwich, CT, USA: Manning Publications Co., 2003.

[16] I. Singh, S. Brydon, G. Murray, V. Ramachandran, T. Violleau, and B. Stearns, *Designing Web Services with the J2EE 1.4 Platform: JAX-RPC, SOAP, and XML Technologies*. Addison-Wesley, Boston, USA, 2004.

[17] M. Barhamgi, D. Benslimane, and B. Medjahed, "A Query Rewriting Approach for Web Service Composition," *IEEE Transactions on Services Computing*, vol. 3, no. 3, pp. 206–222, 2010.

[18] G. Klyne and J. J. Carroll, Eds., *Resource Description Framework: Concepts and Abstract Syntax*. W3C Recommendation, February 2004.

[19] D. A. D'Mello and V. S. Ananthanarayan, "Dynamic Web Service Composition Based on Operation Flow Semantics," *International Journal of Computer Applications*, vol. 1, no. 26, pp. 1–12, 2010.

[20] D. Martin, D. Wutke, and F. Leymann, "Synchronizing Control Flow in a Tuplespace-Based, Distributed Workflow Management System," in *Proc. of the 10th International Conference on Electronic Commerce*, 2008, pp. 19–22.

[21] D. Feldkamp and N. Singh, "Making BPEL flexible," Association for the Advancement of Artificial Intelligence (www.aaai.org), Technical Report SS-08-01, 2008.

[22] T. van Eijndhoven, M.-E. Iacob, and M. L. Ponisio, "Achieving Business Process Flexibility with Business rules," in *Proc. of the 12th International IEEE Enterprise Distributed Object Computing Conference*, 2008, pp. 95–104.

[23] M. Milanović and D. Gašević, "Modeling Service Choreographies with Rule-Enhanced Business Processes," in *Proc. of the 2010 IEEE Enterprise Distributed Object Computing Conference*, 2010, pp. 194–203.