Data-Aware Design and Verification of Service Compositions with Reo and mCRL2*

N. Kokash†
CWI, P.O. Box 94079
1090 GB Amsterdam
The Netherlands
Natallia.Kokash@cwi.nl

C. Krause
CWI, P.O. Box 94079
1090 GB Amsterdam
The Netherlands
Christian.Krause@cwi.nl

E.P. de Vink
Formal Methods Group
TU Eindhoven
The Netherlands
evink@win.tue.nl

ABSTRACT

Service-based systems can be modeled as stand-alone services coordinated by external connectors. Reo is a channel-based coordination language with well-defined semantics that enables a compositional construction of complex connectors from a set of primitive channels. It has been successfully applied in the area of web service composition specification as well as in business process modeling. In this paper, we present a mapping from Reo to `mCRL2`, a specification language based on the process algebra ACP, extended with data and time. The mapping enables verification of Reo process models and service compositions using the `mCRL2` model checking facilities. The supporting Eclipse Coordination Tools suite provides a user-friendly environment for the modeling and verification process.

Keywords

Service-based systems, Reo, `mCRL2`, verification

1. INTRODUCTION

With the advancement of service-oriented computing and web service technology, organizations increasingly assemble their business processes and information systems from services provided by third parties. A new challenge in software engineering has emerged, namely, the question of reliable and adaptable web service composition, particularly addressing service coordination, data transformation, and analysis of non-functional properties.

Reo [1] is a model for exogenous coordination of software components, wherein complex connectors are constructed out of simple primitives called channels. Reo connectors serve to manage communication and coordination among separate components. They specify whether message exchange is synchronous or asynchronous, reorder, duplicate and synchronize messages, etc. Reo has been applied to service composition [15], mash-up building [16] and business process modeling [4, 13]. Moreover, Reo has been extended to capture the notions of time, quality of service, resources, as well as probabilistic and stochastic behavior. Several semantical models of Reo serve different purposes, in particular, its constraint automata semantics [6] is used for formal verification.

The Eclipse Coordination Tools (ECT) [3] is a Reo-based framework for verifiable design of component and service-based software. The framework consists of a set of integrated tools that are implemented as plug-ins for the Eclipse[1] platform. ECT provides functionality for converting high-level modeling languages such UML, BPMN and BPEL to Reo, for editing and animation of Reo models, synthesis of automata-based semantical models from Reo, annotation of Reo and automata with QoS constraints and verifying these models using dedicated model checking tools such as Vereofy [5].

Due to its high expressivity and extensibility, Reo is also suited for specifying complex protocols concerning data manipulation and exchange, such as composing new input messages from multiple output messages of other services, data replication, transformation, cloning and filtering. This caters for focused control flow and data flow models of service compositions. For analysis of these models, e.g. ensuring absence of deadlocks and livelocks, and confirming data consistencies, we need a model checking tool that is able to deal with generic data types as well as user-defined functions for data transformation. In this paper, we propose to rely on the extensive, industrial strength `mCRL2` toolset for this purpose not only for its support of data, but also for its multi-actions which lead to efficient encodings.

The `mCRL2` toolset [8] consists of a behavioral specification language and a model checking tool suite, extending the process algebra ACP with data and time. A fundamental concept in `mCRL2` is the process. Processes perform actions and can be composed using algebraic operators. A system usually consists of several processes running in parallel. A process can carry data via parameters. The state of a process is a specific combination of its control and parameter values. Every specification has a corresponding labeled transition system (LTS) containing all reachable states, along with the possible transitions between them. Most tools in the `mCRL2` toolset operate on so-called linear processes, a normal form

*Supported by NWO GLANCE project WoMaLaPaDiA, SYANCO and IST COMPAS FP7-ICT-2007-1 contract number 215175.
†Corresponding author

SAC 2010 March 22–26, 2010, Sierre, Switzerland

[1] `http://www.eclipse.org`

of mCRL2 processes. This facilitates the verification of properties, specified as formulae in the μ-calculus, for systems with finite as well as infinite state spaces.

In this paper, we integrate mCRL2 model checking facilities in the ECT by mapping Reo connectors to mCRL2 processes. The mapping is automatic and fully compositional: each Reo channel and each Reo node corresponds to a simple process; channels are connected through nodes by synchronizing process actions corresponding to channel/node ends. Larger connectors are built by a suitable synchronization of actions in a parallel composition of a node and its channels connecting to the part of the connector already modeled. We present mapping rules for all basic Reo channels including filters and transformers, and also discuss data-aware analysis of Reo process models with complex data transformation and typecasting.

The remainder is organized as follows. In Section 2, we explain the basics of Reo. In Section 3, we review the mCRL2 specification language and model checking tools. Section 4 presents the mapping of Reo to the mCRL2 specification language. Section 5 describes the corresponding ECT plug-in. In Section 6, we briefly discuss the application of Reo and mCRL2 for modeling and verification of a generic service-based business process. Section 7 treats related work. Finally, in Section 8, we conclude and outline future work.

2. REO AND CONSTRAINT AUTOMATA

A connector in Reo, also called a circuit or a network, is a graph-like structure where the edges are communication channels which are connected using nodes. A channel always has two ends, which can be a source or a sink. Source ends accept data into, sink ends dispense data out of their channel. Reo allows channels with ends which are both sinks or both sources.

Channels can be joined together using nodes. There are three types of nodes: source, sink and mixed nodes, depending on whether all coinciding channel ends are source ends, sink ends or a combination of both, respectively. Components can write data to source nodes of connectors and read data from sink nodes. Data items that entered the circuit can be rearranged and transformed according to relational constraints imposed by Reo channels. Some basic examples of Reo channels are shown in Figure 1. A *synchronous channel* (Sync) •⟶• has a source end and a sink end and no buffer. It accepts a data item through its source if it can instantly dispense it through the sink. A *lossy synchronous channel* (LossySync) •- - - -• is similar to a synchronous channel except that it always accepts all data items through its source end. The data item is transferred if the data item can be dispensed through the sink end, otherwise the data item is lost. A *synchronous drain* (SyncDrain) •⟶⟵• has two source ends and no sink end. It accepts a data item through one of its ends exactly if a data item is also available to be accepted simultaneously through the other end as well. An *asynchronous drain* (AsyncDrain) •⟶||⟵• accepts data items through its source ends and loses them, but never simultaneously. A FIFO1 channel (FIFO1) •⟶□⟶• represents an asynchronous channel with one buffer cell.

Other channels, particularly useful for modeling service coordination and business process are filter, transformer and timer channels. For a *filter channel* •⟶^^⟶• , its pattern $P \subseteq Data$ specifies the type of data items that can be transmitted through the channel. Any value $d \in P$ is accepted through its source end if its sink end can simultaneously dispense d; all data items $d \notin P$ are always accepted through the source end but are immediately lost. A transformer channel •⟶▷⟶• applies a user-defined function to a data item consumed from its source. Timer channels •⟶○⟶• have been introduced in [2] to deal with time-aware coordination. For example, the source end of a *t-timer with off and reset option* channel accepts any input value $d \in Data$ and returns at its sink end a timeout signal after a delay of t time units. The *off-option* allows the timer to be stopped before the expiration of its delay, when a special 'off' value is consumed through its source end, while the *reset-option* allows the timer to be reset before the expiration of its delay, when a special 'reset' value is consumed through its source end. For the sake of exposition we discuss in the present paper time-less connectors only.

Semantics of Reo can be given in terms of constraint automata. Constraint automata have transitions whose labels capture the synchronization and data flow constraints between ports by describing the sets of ports that fire together. Figure 1 depicts the constraint automata for the basic Reo channels. Note that only synchronization constraints are shown in the picture. A detailed discussion of data constraints and the formal definition of constraint automata semantics for Reo can be found in [6].

Reo allows an open-ended set of user-defined channels with arbitrary input/output behavior, but it fixes the semantics of nodes. A source node acts as a synchronous replicator. A sink node acts as a non-deterministic merger. A mixed node combines the behavior of the other two nodes and consumes an item from one of its selected sink ends and replicates it to all of its source ends. Nodes do no buffering. This forces synchrony and exclusion constraints to propagate through the connector, and causes the channels of the connector to synchronize their actions in atomic steps. Figure 2 shows Reo circuits for replication, merging and routing of data along with their constraint automata. We often use a node ⊗ as shorthand for the router circuit drawn in the figure. Another useful element for modeling data flow using Reo circuits is the so-called *join* node, denoted by ⊕, which represents a component that accepts data items from all connected sink ends and creates a tuple of them which is then send to one of the connected source ends.

3. mCRL2

The mCRL2 language provides a number of built-in data types such as Booleans, natural and positive numbers, integers, and reals. All standard arithmetic operations are provided. Algebraic data type definition mechanisms in mCRL2 allow users to declare new sorts including constructors and function definitions.

One of the reasons to choose the mCRL2 toolset for verifying Reo connectors is its ability to deal with composite data types such as lists, sets, and bags, as well as with user-defined sorts. E.g., an enumerated type in mCRL2 is declared in the following form:

$$\begin{aligned} \mathrm{S} = \textbf{struct}\ & c_1(p_1^1{:}S_1^1, ..., p_1^{k1}{:}S_1^{k1})?r_1 \\ & \quad | \quad ... \quad | \\ & c_n(p_n^1{:}S_n^1, ..., p_n^{kn}{:}S_n^{kn})?r_n \end{aligned}$$

This defines the sort S together with the constructor functions $c_i : S_i^1 \times ... \times S_i^{ki} \to S$, projection functions $p_i^j : S \to S_i^j$, and recognizer functions $r_i : S \to Bool$. For more details,

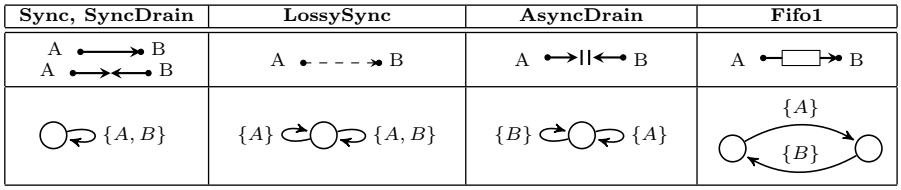


Figure 1: Graphical notation and constraint automata for basic Reo channels

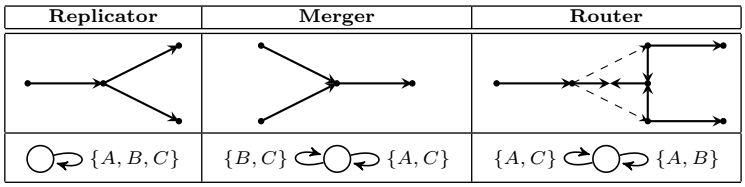


Figure 2: Graphical notation and constraint automata for simple connectors

see the mCRL2 web site.[2]

The most basic notion in mCRL2 is an action. Actions represent atomic events and can be parametrized with data. Parametrized actions are referred to as $a(\overrightarrow{x}), b(\overrightarrow{y}), ...$ with $\overrightarrow{x}, \overrightarrow{y}, ...$ vectors of data parameters. Actions in mCRL2 can synchronize. In fact, mCRL2 has multiway synchronization, rather than handshaking only. Communication functions $\Gamma_C(p)$ govern this. Here, C is a list of communications of the form $a_0 \,|\, ... \,|\, a_n \rightarrow c$, $(n \geq 1)$. Within the execution of p, when offered in parallel, the actions $a_0, ..., a_n$ synchronize yielding the action c. A special τ (tau) construct is used to refer to an internal unobservable action. As we shall use frequently below, mCRL2 allows multiactions. E.g., the expression $a|b$ prescribes the simultaneous execution of the actions a and b. The constant process δ denotes inaction or deadlock and does not display any behavior.

Processes are defined by process expressions, which are compositions of actions using a number of operators. The basic operators in mCRL2 are the following:

- *alternative composition*, written as $p + q$, which represents a non-deterministic choice between processes p and q;
- *sequential composition*, written $p \cdot q$, which means that q is executed after p assuming that p terminates;
- *conditional*, written as $c \rightarrow p \diamond q$, where c is a data expression that evaluates to true or false;
- *summation* $\Sigma_{d:D}\, p$, where p is a process expression in which data variable d may occur, used to quantify over data types;
- *parallel composition* $p \parallel q$, which interleaves and synchronizes the actions of p with actions of q;
- *allow* $\nabla_V(p)$, where only actions in p from the set V are allowed to occur;
- *encapsulation* $\partial_H(p)$, where all actions in p from the set H are blocked;
- *renaming* $\rho_R(p)$, where R is a set of renamings of the form $a \mapsto b$, meaning that every occurrence of action a of p is replaced by action b;
- *hiding* $\tau_I(p)$, renaming all actions in I of p into τ.

The mCRL2 toolset collects a number of tools that allow users to verify software models specified in the mCRL2 language. The toolset includes a tool for converting mCRL2 specifications into linear form (a compact symbolic representation of the corresponding LTS), a tool for generating explicit LTSs from linear process specifications, and tools for optimizing and visualizing these models. A detailed overview of all available tools can be found on the mCRL2 web site.

For symbolic model checking, system properties are specified as formulae in a variant of the modal μ-calculus extended with regular expressions and data. Together with the model description in mCRL2, a formula is transformed into a parametrized boolean equation system (PBES) and fed into the PBES solver. Analysis at the level of LTSs is possible too by means of the *ltsconvert* utility. This tool can minimize a given LTS and compare LTSs in different formats modulo strong bisimilarity, branching bisimilarity, strong simulation equivalence, trace equivalence and weak trace equivalence.

4. ENCODING REO CONNECTORS

The purpose of mapping Reo to mCRL2 is to enable verification of service compositions and business processes modeled in Reo. When converting Reo specifications to a format suitable for model checking, it is beneficial to exploit the compositionality of the model. The mCRL2 modelling language supports this. Below we provide rules for mapping Reo primitives (channels and nodes) to mCRL2 processes and explain how the mCRL2 specification for a general Reo connector with data is composed from the mCRL2 specification of subconnectors.

4.1 Mapping of basic Reo channels

The mapping of basic Reo channels is guided by the constraint automata semantics of Reo. For defining a process for a Reo channel, we introduce two actions that represent data flow on its ends. For defining a process for a node, we introduce one action for each channel end connected to the node. Processes corresponding to channels and nodes are defined using the aforementioned mCRL2 operators according to the semantics of channel and node types discussed in Section 2. Thus, a synchronous channel with ends A and B

[2] http://www.mcrl2.org

can be represented as a recursive process which executes a multiaction $A|B$:

$$Sync = A|B \,.\, Sync$$

The data-agnostic definition of the synchronous drain channel is similar. A synchronous lossy channel with the source end A and sink end B can be defined as

$$LossySync = (A + A|B) \,.\, LossySync$$

while an asynchronous drain channel with source ends A and B is represented as

$$AsyncDrain = (A + B) \,.\, AsyncDrain$$

As an empty *Fifo* channel first accepts data through its source end A and then asynchronously dispenses it through its sink end B, its corresponding `mCRL2` process can be defined as

$$Fifo1 = A.B \,.\, Fifo1$$

For a full $Fifo1$, we introduce an additional process

$$FullFifo1 = B \,.\, Fifo1$$

which requires data flow in B to be observed first. Alternatively, a $Fifo1$ can be modeled as a parametrized process

$$Fifo1(b : Bool) = (\neg b \rightarrow A \diamond B) \,.\, Fifo1(\neg b)$$

This definition introduces a process with two states: in case the buffer is empty, data can be read through end A which leads to the change of the state to full; otherwise, the data item has to be dispensed through end B which leads to the change of the state to empty. The process needs to be initialized as $Fifo(false)$ for an empty $Fifo1$ channel and $Fifo(true)$ for a full Fifo1 channel. The LTS generated by the `mCRL2` tools for the aforementioned processes correspond to the constraint automata shown in Figure 1.

A process for a node with one sink end X and two (or more) source ends Y and Z, which acts as a replicator, can be described as

$$ReplicatorNode = X|Y|Z \,.\, ReplicatorNode$$

while a process for a node with two (or more) source ends X and Y and a sink end Z which works as a non-deterministic merger can be represented as a process with a non-deterministic choice between two multiactions:

$$MergeNode = (X|Z + Y|Z) \,.\, MergeNode$$

A process for a router node is defined in a similar way.

4.2 Channel composition

A process for a Reo circuit is formed by synchronizing actions corresponding to joint channel ends at a node in a parallel composition of processes corresponding to channels and nodes of the circuit. For example, a `mCRL2` process for the replicator circuit shown in Figure 2 can be formed from three synchronous channels

$$\begin{aligned} Sync1 &= A|X \,.\, Sync1 \\ Sync2 &= Y|B \,.\, Sync2 \\ Sync3 &= Z|C \,.\, Sync3 \end{aligned}$$

and a replicator node

$$ReplicatorNode = X'|Y'|Z' \,.\, ReplicatorNode$$

applying communication and blocking operators to their parallel composition:

$$\begin{aligned} Replicator = \partial_{\{X,Y,Z,X',Y',Z'\}}\big(\\ \Gamma_{\{X|X'\rightarrow\tau,Y'|Y\rightarrow\tau,Z'|Z\rightarrow\tau\}}\big(\\ Sync1 \parallel Sync2 \parallel Sync3 \parallel ReplicatorNode \,)\,\big) \end{aligned}$$

Here we assume that the sink end X of the channel $Sync1$ is connected to the sink end X' of the node $ReplicatorNode$, while sink ends Y' and Z' of the node are connected to source ends Y and Z of channels $Sync2$ and $Sync3$.

Although it is possible to describe any circuit as a single process using the principle above, state explosion may occur since many nondeterministic unfoldings with unmatched internal synchronization actions will be caught as unfeasible at a late stage. To overcome this problem, we create a process for a Reo circuit consequently synchronizing actions corresponding to channel and node ends for each node separately. The fact that is semantic preserving relies, among others, on the associativity and commutativity of the parallel operator and on distributivity properties. Note that the representation is data oblivious. The `mCRL2` process for a connector can be written as follows:

$$\begin{aligned} P_0 &= \partial_{H_0}\big(\ \Gamma_{C_0}(\,N_0 \parallel \textstyle\prod_{M_0}\,)\,\big) \\ P_i &= \partial_{H_i}\big(\,\Gamma_{C_i}(P_{i-1} \parallel N_i \parallel \textstyle\prod_{M_i}\,)\,\big) \end{aligned}$$

where N_i is a node, $\prod_{M_i}$ denotes the parallel composition of processes from the set M_i defined as a set of processes for channels connected to the node N_i that have not been initialized before, C_i is a set of multiactions for pairs of joint channel/node ends and H_i is a set of atomic actions corresponding to these ends that should synchronize within the subconnector. For example, for the router circuit shown in Figure 2 we first generate a process

$$\begin{aligned} P_0 = \partial_{\{D',D'',F',F'',P',P'',S',S''\}}\big(\\ \Gamma_{\{D'|D''\rightarrow\tau,F'|F''\rightarrow\tau,P'|P''\rightarrow\tau,S'|S''\rightarrow\tau\}}\big(\\ Node1 \parallel Sync1 \parallel LossySync1 \\ \parallel LossySync2 \parallel SyncDrain1\,)\,\big) \end{aligned}$$

which connects the sink end of the channel

$$Sync1 = A|S'' \,.\, Sync1$$

to the source ends of the channels

$$\begin{aligned} LossySync1 &= (D'' + D''|E'') \,.\, LossySync1 \\ LossySync2 &= (F'' + F''|G'') \,.\, LossySync2 \\ SyncDrain1 &= P''|Q'' \,.\, SyncDrain1 \end{aligned}$$

using a node

$$Node1 = S'|D'|F'|P' \,.\, Node1$$

Then, we define a process

$$\begin{aligned} P_1 = \partial_{\{E',E'',H',H'',L',L''\}}\big(\\ \Gamma_{\{E'|E''\rightarrow\tau,H'|H''\rightarrow\tau,L'|L''\rightarrow\tau\}}\big(\\ Node2 \parallel P_0 \parallel Sync2 \parallel Sync3\,)\,\big) \end{aligned}$$

which connects a node

$$Node2 = E'|H'|L' \,.\, Node2$$

and channels

$$\begin{aligned} Sync2 &= H''|B \,.\, Sync2, \\ Sync3 &= L''|M'' \,.\, Sync3 \end{aligned}$$

to the process P_0. This procedure is iterated until all (four) nodes are covered. The nodes can be added in any order. However, as it turns out, by using a depth-first strategy while traversing the connector graph, which leads to the synchronization and hiding of actions in joint nodes, we can significantly reduce the number of states in the intermediate networks, keep the linear process representation concise and speed up the computation involved.

4.3 Reo connectors with data

Applications of Reo to component/service coordination as well as business process modeling pose the question of data-aware analysis of Reo circuits. Indeed, a coordination strategy or process behavior may depend on data types external components operate on or data values returned/accepted by services.

Data can be added to mCRL2 specifications of Reo networks by defining parametrized actions for all channel and node ends. Let us assume that all channels operate on a global sort (type) *Data*. Then, every Reo channel must be able to accept any value of this sort. Such a channel can be specified in mCRL2 with the help of the summation operator, e.g.,

$$Sync = \Sigma_{d:Data}\ A(d)|B(d)\ .\ Sync$$

The data-aware model of a synchronous drain channel differs from a synchronous channel since the former channel does not pose constraints on values accepted by its source ends, while for the latter the data on its ends is the same. This leads to the following code:

$$SyncDrain = \Sigma_{d_1,d_2:Data}\ A(d_1)|B(d_2)\ .\ SyncDrain$$

Similarly, with only one data parameter needed,

$$AsyncDrain = \Sigma_{d:Data}\ (\ A(d) + B(d)\ )\ .\ AsyncDrain$$

For defining a *Fifo1* channel, we introduce a new sort

$$DataFifo = \textbf{struct}\ empty?isEmpty|full(e : Data)?isFull$$

which allows us to specify whether the channel buffer is empty or full, and if it is full, store the value of the sort *Data* in it:

$$Fifo1(f : DataFifo) = \Sigma_{d:Data}\ (\ isEmpty(f) \rightarrow A(d)\ .\ Fifo1(full(d)) \diamond B(e(f))\ .\ Fifo1(empty)\ )$$

Replication, merge and router nodes are defined as previously with the only difference that all actions corresponding to node ends are parametrized. Additionally, in many cases we want to join several data elements to a single tuple. This can be accomplished with the help of a join node:

$$JoinNode = \Sigma_{d_1,d_2:Data}\ (\ X(d_1)|Y(d_2)|Z(tuple(d_1,d_2))\ )\ .\ JoinNode$$

However, this introduces a problem as apparently all other channels involved should be able to operate with tuples. Therefore, given a set of elementary data types $DT_1, ..., DT_n$, we define *Data* as follows:

$$Data = \textbf{struct}\ D_1(e_1 : DT_1)\ |\ ...\ |\ D_n(e_1 : DT_n)\ |\ tuple(e_1 : Data, e_2 : Data)$$

Note that this recursive definition allows us to form tuples from elements of any basic type as well as their binary tuples, thus, forming tree-like structures. Note, this data type is suitable for circuits with join nodes that contain two source ends only. In the general case, for every join node with k source ends a $tuple_k(e_1 : Data, ..., e_k : Data)$ must be added to the definition. Elementary types are defined by external components/services coordinated by the circuit.

Filter and transformer channels can be used to modify data flow in the circuit. A filter channel represents a lossy synchronous channel with a filter expression $expr(d)$ yielding true or false:

$$Filter = \Sigma_{d:Data}\ (\ expr(d) \rightarrow A(d)|B(d) \diamond A(d)\ )\ .\ Filter$$

A transformer channel in the simplest case works as a synchronous channel that performs some operation f on a data item consumed through its source end:

$$Transformer = \Sigma_{d:Data}\ A(d)|B(f(d)\ .\ Transformer$$

with f: $Data \rightarrow Data$. For partially defined functions the transformer channel works as a filter channel. Other variants of transformers can be defined as well, e.g., channels that implement functions $g_{ij} : DT_i \rightarrow DT_j$ for transforming data items of one elementary data type to another as well as to define projections of data tuples appear to be useful.

5. TOOL SUPPORT

Using the conversion plug-in of the Eclipse Coordination Tools, an mCRL2 script can be obtained automatically for any Reo circuit by selecting the mCRL2 tab in the channel/connector property view of the ECT (see Figure 3). In this view, a user can switch between several options such as *network scope*, which means that the generated mCRL2 code will define processes for components that provide data for connector input ports and consume data from their output ports, or the *with data* option, which indicates whether encoding is data-aware. Alternatively *intentional encoding* can be selected, to switch between mappings based on constraint automata or on intentional automata-based.[3] Also, for fine-grained inspection, one can choose the *user-defined tau* option. Contrary to the other options, all synchronized actions corresponding to mixed circuit nodes will not be renamed to the internal, unobservable action τ, treated by mCRL2 in a special way.

Sorts of data generated/accepted by components or services coordinated by Reo can be defined in the field *Datatype*. For a correct processing, these definitions must be valid mCRL2 definitions. The generated specification can be model checked by the *lps2pbes* tool from the mCRL2 toolset which is invoked by pressing *Check* button. A formula for verification is specified in the field *Formula* provided by the plug-in. For finite data domains, one can visualize the circuit mapping by generating an LTS which can be obtained by pressing a *Show LTS* button. The LTS can be exported as well. Other verification tools, such as the one provided by the CADP toolbox[4], can then be applied as well.

6. CASE STUDY

We briefly discuss a web-service example which illustrates the formal process modeling and verification using Reo and mCRL2.

[3] mCRL2 encodings based on the intentional automata semantics for Reo are reported in a forthcoming paper.

[4] http://www.inrialpes.fr/vasy/cadp

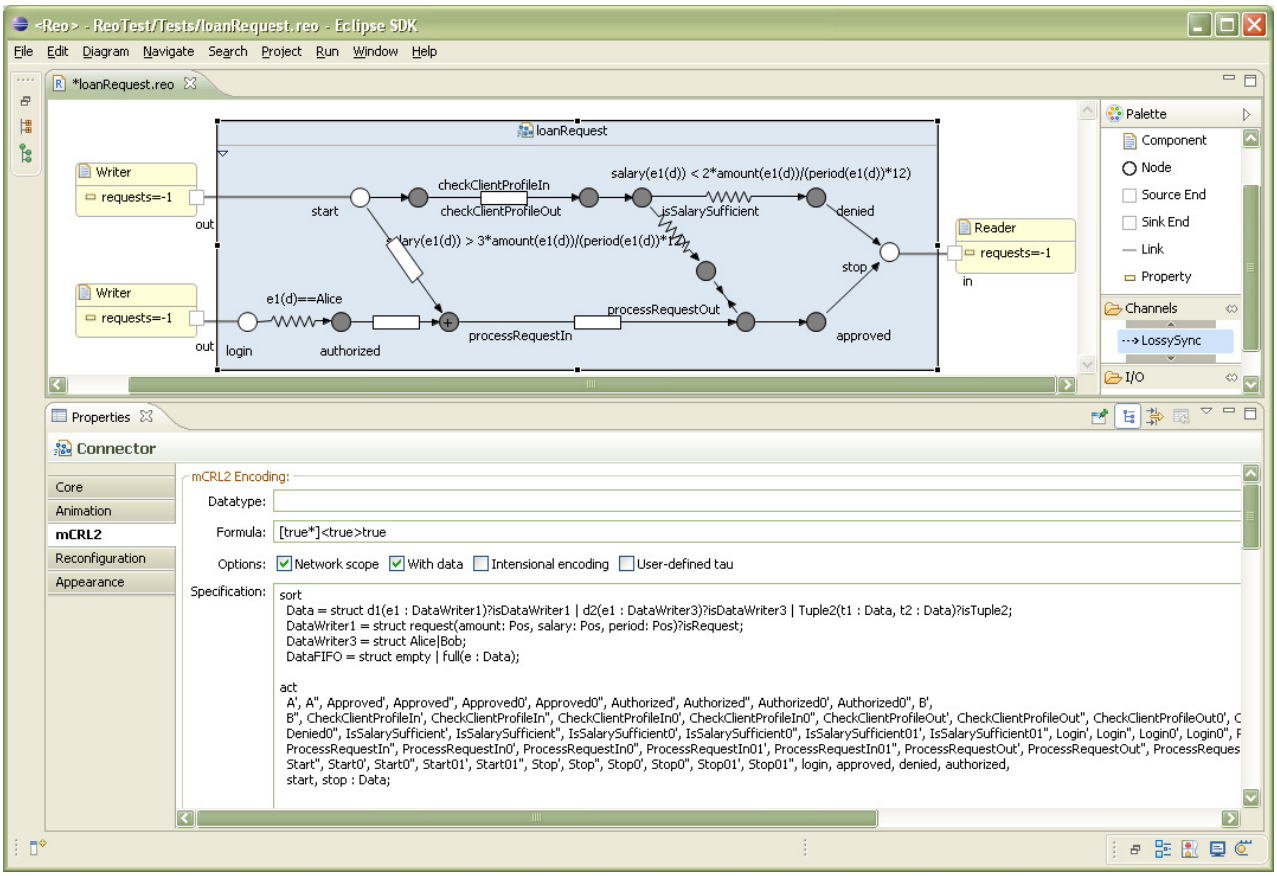


Figure 3: Reo to mCRL2 conversion plug-in

We consider a typical Loan Request scenario where the decision to approve or reject a client's application for a loan depends on the details of the request, say, the client's salary, the required loan amount and the period. A Reo connector for this scheme is shown in the upper part of Figure 3. In this model, a client's request is specified as a data item $request(amount{:}Pos, salary{:}Pos, period{:}Pos)$ provided by a writer component. Fifo1 channels represent basic (atomic) activities that constitute the process. The request is denied if the salary during the loan period is half of the required loan amount, and approved if it is three times bigger than this amount. These conditions are modeled using two filter channels. Another writer represents a bank clerk (Alice or Bob) who, in principle, may process the request. However, the condition of the filter channel $filter(login, authorized)$ allows only Alice to login in the system and process the request.

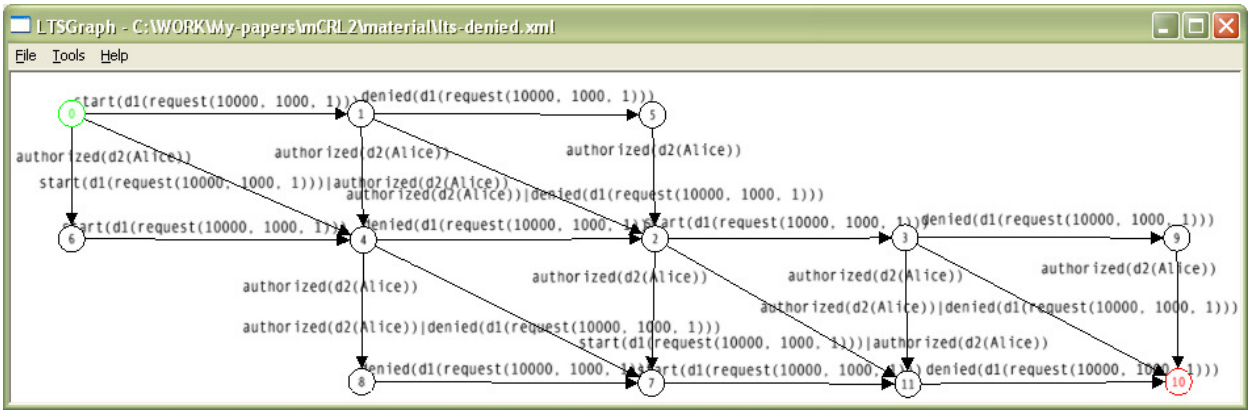
Actually, the process discussed contains a structural error. If the approval condition regarding the loan does not hold, the process deadlocks. Figure 4 shows the reduced LTSs, minimized modulo branching bisimilarity [9], for different instances of the process, namely, for loan requests with amount of 10000, period of one year and client's salaries of 1000, 2000 and 3000. The LTSs for the first and second process instances contain deadlocks (states number 10 and 5, respectively), automatically detectable by our set-up of model checking tools.

7. RELATED WORK

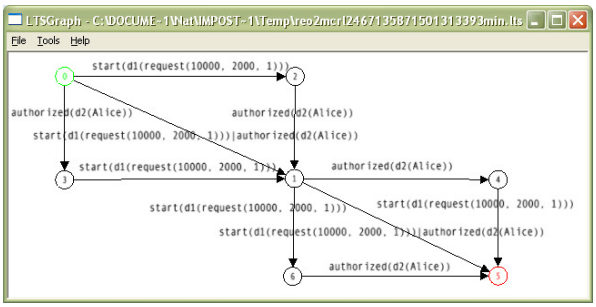
In this section, we compare our framework to other tool supported approaches to verifiable business process design and web service composition modeling.

A brief survey of existing business process design tools can be found in [10]. This work compares a set of academic and commercial products for service-based process modeling, namely, Bind Studio, BPEL Orchestration Server, WebDesigner, eFlow, SWORD, DAML-S, WCPT, SODA, and WebDG. Most of these tools do not provide means for formal process verification, while others (SWORD and DAML-S) rely on Petri net based models. The absence of formal semantics in many business process modeling notation and specification languages poses the need of their transformations into models having a more formal semantical basis such as Reo, proposed here, or Petri nets, process algebras or finite-state machines.

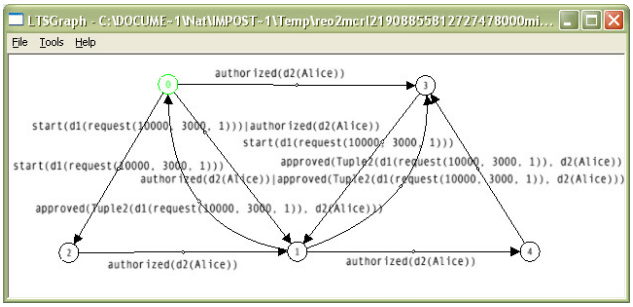
Petri nets are a powerful model for concurrent process specification supported by a number of industrial tools. Multiple extensions of Petri nets have been introduced do deal with various kinds of process analysis. For example, Trčka et al. [18] rely on Petri nets to verify unified control and data flow of business processes. However, to express certain behavioral aspects, e.g., process cancellation and/or compensation, Petri nets are usually extended with additional elements such as inhibitor and reset arcs. As has been pointed out by Raedts et al. [17], this significantly reduces the number of software tools available to analyze such mod-



(a) salary=1000, deadlock state 10



(b) salary=2000, deadlock state 5



(c) salary=3000, no deadlock

Figure 4: LTS for different instances of the Loan Request scenario

els. [17] uses the mCRL2 toolset too for process verification. Our work has been inspired by this paper. However, we do not limit our approach to structural verification of workflow models, but target at time- and data-aware analysis, while the translation of [17] only represents the number of data tokens in Petri nets, but not their content.

Web Service Analysis Tool (WSAT)[5] is a web service composition verification tool which exploits SPIN and NuSMV model checkers. Kazhamiakin and Pistore [11] discuss verification of stateful service compositions under various communication models (synchronous vs. asynchronous). Service behavior represented in BPEL is formalized by means of LTS and analyzed using NuSMV. This work illustrates the importance of proper service coordination (e.g., for correct process cancellation), which can be easily realized using synchronous Reo channels. Eshuis and Wieringa [7] describe verification of processes modeled as UML activity diagrams with events, data, real-time and loops using SPIN and NuSMV model checkers. NuSMV and SPIN appear to be the most popular verification tools in the area of SOA. Their comparison suggests that SPIN can be more flexible than NuSMV in handling large scale systems, but can also be more difficult to use by non-experts. In our future work, we will consider generating input files for SPIN and NuSMV from Reo process models similarly to the mCRL2 scripts. However, the process coordination in SPIN is implemented via shared variables, which conceptually differs from the channel-based coordination in Reo.

While structural verification of process models is well examined, there are not so many tools that provide both control and data-flow analysis. This requires specification languages and model checking tools supporting (algebraic) data types. One of the tools able to deal with composite data structures such as objects, trees and lists is Alloy.[6] The UML2Alloy toolset[7] can be exploited to verify the correctness of UML/OCL models. Khosravi et al. [12] establish a transformation from Reo to Alloy. This work has not been completed as yet, as it currently ignores the actual values of data passed through the channels and various performance issues need to be tackled still.

The tool most closely related to the conversion and integration plug-in presented in this paper is Vereofy [5], a model checking tool developed at the University of Dresden for the analysis of Reo connectors. Vereofy uses two input languages, namely, the Reo Scripting Language (RSL), and a guarded command language called Constraint Automata Reactive Module Language (CARML) which are textual versions of Reo and constraint automata, respectively. Scripts in these languages are automatically generated from graphical Reo models. However, Vereofy provides a format for specifying filter conditions that is less expressive, and, in contrast to our approach, does not allow join nodes in the circuits. Also it expects the user to define a global data domain eligible to all connectors and components in the model. To the best of our knowledge, the toolset presented supports a property specification format that is more expressive than any of the other verification frameworks mentioned.

[5] http://www.cs.ucsb.edu/~su/WSAT

[6] http://alloy.mit.edu/community

[7] http://www.cs.bham.ac.uk/~bxb/UML2Alloy/index.php

Properties of dynamically changing Reo connectors are considered in [14]. Reconfigurations of connectors are modeled using a rule-based graph transformation approach. Similar to the work in this paper, external tools are used for reasoning about the dynamic properties. However, the focus is not on data but on the potentially harmful interactions of reconfiguration and execution.

8. CONCLUSIONS

In this paper, we presented a translation of Reo connectors into `mCRL2` specifications. This makes formal verification of Reo connectors possible using the `mCRL2` toolset. The mapping is compositional and is, via a plug-in of the ECT suite, fully automated. Verification of circuits with join nodes, filter, transformer and timer channels that coordinate components and services operating on different data types is supported. In this sense, our solution is more targeted than other approaches to model checking of Reo and analysis of service-based systems.

Future work includes the development of a tool that converts XML schemas used in WSDL to describe message data types into the `mCRL2` data format. In particular, the current experiments reveal that fine-tuning is needed regarding the underlying linear process representations, as this has major impact on the performance of the symbolic model checking algorithms. Together with the BPMN, UML and BPEL to Reo converters available, this then will constitute a tool chain for the analysis of real-world service-based systems.

9. REFERENCES

[1] F. Arbab. Reo: A channel-based coordination model for component composition. *Mathematical Structures in Computer Science*, 14:329–366, 2004.

[2] F. Arbab, C. Baier, F. de Boer, and J. Rutten. Models and temporal logical specifications for timed component connectors. *Software and Systems Modeling*, 6(1):59–82, 2007.

[3] F. Arbab, C. Koehler, Z. Maraikar, Y.-J. Moon, and J. Proença. Modeling, testing and executing Reo connectors with the Eclipse Coordination Tools. In C. Canal and C. Pasareanu, editors, *Proc. FACS 2008*. ENTCS, to appear.

[4] F. Arbab, N. Kokash, and M. Sun. Towards using Reo for compliance-aware business process modelling. In T. Margaria and B. Steffen, editors, *Proc. ISoLA 2008*, pages 108–123. Communications in Computer and Information Science 17, 2008.

[5] C. Baier, T. Blechmann, J. Klein, and S. Klüppelholz. A uniform framework for modeling and verifying components and connectors. In J. Field and V. Thudichum Vasconcelos, editors, *Proc. COORDINATION 2009*, pages 268–287. LNCS 5521, 2009.

[6] C. Baier, M. Sirjani, F. Arbab, and J. Rutten. Modeling component connectors in Reo by constraint automata. *Science of Computer Programming*, 61:75–113, 2006.

[7] R. Eshuis and R. Wieringa. Verification support for workflow design with UML activity graphs. In *Proc. ICSE 2002, Orlando, Florida*, pages 166–176. ACM, 2002.

[8] J. Groote, A. Mathijssen, M. Reniers, Y. Usenko, and M. van Weerdenburg. The formal specification language `mCRL2`. In E. Brinksma, D. Harel, A. Mader, P. Stevens, and R. Wieringa, editors, *Methods for Modelling Software Systems*. IBFI, Schloss Dagstuhl, 2007.

[9] R. van Glabbeek and P. Weijland. Branching time and abstraction in bisimulation semantics. *Journal of the ACM*, 43:555–600, 1996.

[10] S.-M. Huang, Y.-T. Chu, S.-H. Li, and D. Yen. Enhancing conflict detecting mechanism for web services composition: A business process flow model transformation approach. *Science of Computer Programming*, 66(3):205–225, 2007.

[11] R. Kazhamiakin and M. Pistore. A parametric communication model for the verification of BPEL4WS compositions. In M. Bravetti, L. Kloul, and G. Zavattaro, editors, *Proc. WS-FM*, pages 318–332. LNCS 3670, 2005.

[12] R. Khosravi, M. Sirjani, N. Asoudeh, S. Sahebi, and H. Iravanchi. Modeling and analysis of Reo connectors using Alloy. In D. Lea and G. Zavattaro, editors, *Proc. COORDINATION 2008*, pages 169–183. LNCS 5052, 2008.

[13] N. Kokash and F. Arbab. Formal behavioral modeling and compliance analysis for service-oriented systems. In F. de Boer, M. Bonsangue, and E. Madelain, editors, *Proc. FMCO 2008*, pages 21–41. LNCS 5751, 2009.

[14] C. Krause, Z. Maraikar, A. Lazovik, and F. Arbab. Modeling Dynamic Reconfigurations in Reo using High-Level Replacement Systems. *Science of Computer Programming*, 2009. To appear.

[15] A. Lazovik and F. Arbab. Using Reo for service coordination. In B. Krämer, K.-J. Lin, and P. Narasimhan, editors, *Proc. ICSOC 2007*, pages 398–403. LNCS 4749, 2007.

[16] Z. Maraikar, A. Lazovik, and F. Arbab. Building mash-ups for the enterprise with SABRE. In A. Bouguettaya, I. Krüger, and T. Margaria, editors, *Proc. ICSOC 2008*, pages 70–83. LNCS 5364, 2008.

[17] I. Raedts, M. Petkovic, Y. Usenko, J. van der Werf, J. Groote, and L. Somers. Transformation of BPMN models for behaviour analysis. In J. Augusto, J. Barjis, and U. Ultes-Nitsche, editors, *Proc. MSVVEIS 2007*, pages 126–137. INSTICC Press, 2007.

[18] N. Trcka, W. van der Aalst, and N. Sidorova. Analyzing control-flow and data-flow in workflow processes in a unified way. Technical Report CSR–0831, Technische Universiteit Eindhoven, 2008. 21pp.