Time and Data-Aware Analysis of Graphical Service Models in Reo

N. Kokash[1,2] & C. Krause[3]
CWI, P.O. Box 94079, 1090 GB Amsterdam
The Netherlands

E.P. de Vink
Technische Universiteit Eindhoven
P.O. Box 513, 5600 MB Eindhoven, The Netherlands

Abstract—**Reo is a graphical channel-based coordination language that enables the modeling of complex behavioral protocols using a small set of channel types with well-defined behavior. Reo has been developed for the coordination of stand-alone components and services, which makes it suitable for the modeling of service-based business processes. The formal semantic models for Reo lay the grounds for computer-aided analysis of different aspects of Reo diagrams, including their animation, simulation and verification of control flow and data flow by means of model checking techniques. In this paper, we discuss the verification of data aware Reo process models using the `mCRL2` model checking toolset including time analysis. We also show how behavior abstraction can be used to minimize Reo process models and generate smaller `mCRL2` specifications. A detailed auction example illustrates our approach to time-aware modeling and verification of data-centric service models.**

Keywords—**formal methods for service-oriented computing; model checking; coordination languages**

I. Introduction

An essential feature of a process modeling notation is a clear definition of its execution semantics, supported by efficient tools for the validation, verification and performance analysis of resulting process models. In the past decades, various modeling notations and tools for workflow and services have been proposed [1], [2]. They vary at different levels, ranging from the particular graphical syntax for process model definitions to the expressiveness and extensibility of functional and organizational aspects of the model. However, general formalisms for verifiable process design, such as process algebras or finite-state machines, are typically too low level. Therefore, encoding high-level modeling notations such as BPMN and UML Activity Diagrams into these formalisms usually yields a complex and hardly understandable description of the original system. Moreover the analysis results are hard to trace back into the original model. The abundance of process definition languages on the one hand, and low-level analysis techniques on the other, suggests that an intermediate between the design of a workflow process and its analysis is required. Petri nets are a prime example of such a model. In various extensions –most notably reset and inhibitor arcs, color, time and hierarchy– they provide solid ground to workflow analysis. However, as Petri nets are token-driven by nature, they are on their own not sufficient for dataflow verification.

Services are autonomous, loosely coupled software components that can be integrated into more complex systems via publicly available interfaces. When designing a service-based system, the developer focuses on a seamless integration of existing services, rather than on the implementation of new functional modules. This integration or composition of services requires a proper synchronization, buffering and ordering of the exchanged messages, as well as a transformation of data formats of the different services. While the routing of messages can be sufficiently modeled with traditional Petri nets, the data manipulation aspect requires more powerful concepts [3], [4].

Conceptually, service-oriented computing is similar to exogenous coordination, which advocates the separation of computation provided by services on the one hand, and their coordination provided by some 'glue code' [5] on the other. The Reo coordination language [6] adheres to this principle and provides concepts and tools to construct such glue code to integrate services given a specification of their behavioral interfaces. A Reo process model is essentially a set of complex connectors (also called circuits) composed of channels which provide the basic form of interaction between two parties by imposing constraints on the data they exchange. The application of Reo to business process and service modeling is discussed more extensively in [7], [8]. The formal semantics for Reo [9], [10] makes it possible to analyze the behavior of a connector using model checking techniques. In our recent work [11], we present a tool for converting Reo to `mCRL2`, a specification language based on the process algebra ACP extended with time and data [12]. Specifications in this language can be analyzed by the model checking tool available in the `mCRL2` toolset itself, or converted into an LTS in various formats and used as input for external model checking tools. The `mCRL2` model checker is capable of dealing with state spaces containing millions of states and transitions and it has proven its power in large scale industrial applications. In particular the support for structured data types and function definitions supported by `mCRL2` are of major importance for data-aware analysis, as required in the setting of service-based systems.

In this paper, firstly, we extend the approach of [11] to

[1]Supported by IST COMPAS FP7–ICT–2007–1, contract 215175
[2]Corresponding author, email `Natallia.Kokash@cwi.nl`
[3]Supported by NWO GLANCE WoMaLaPaDiA and SYANCO

deal with timed process models. For enabling time-aware service interaction, Reo introduces special channels with internal timers. Their semantics is given by timed constraint automata or TCA [13]. Given graphical timed Reo models, we automatically generate equivalent `mCRL2` specifications for subsequent model checking and other analysis with the `mCRL2` toolset. Secondly, we show how the behavior abstraction mechanism in Reo can be used to reduce the size of the generated `mCRL2` specification and improve its processing time. Finally, we discuss a fragment of a classical auction scenario to illustrate our approach to modeling and analysis of timed data-centric service models.

The rest of this paper is organized as follows. In Section II, we summarize the basics of Reo. In Section III, we present the `mCRL2` specification language and briefly describe the translation of Reo to `mCRL2`. In Section IV, we extend our approach with the translation of timed Reo to `mCRL2`. We address model abstraction and `mCRL2` code minimization in our framework in Section V and present tool support in Section VI. In Section VII we apply Reo and `mCRL2` for the analysis of an auction. Sections VIII and IX discuss related work and conclusions.

II. Reo coordination language

Reo is a channel-based coordination language in which components and services are coordinated exogenously by so-called *connectors* [6]. Connectors are essentially graphs where the edges are user-defined communication channels and the nodes implement a fixed routing policy.

Channels in Reo are entities that have exactly two ends, also referred to as *ports*, which can be either *source* or *sink* ends. Source ends accept data into, and sink ends dispense data out of their channel. Reo allows directed channels as well as ones with respectively two source or sink ends. Although channels can be defined by users in Reo, a set of basic channels suffices to implement rather complex coordination protocols. The most basic channel in Reo is the Sync channel, which is a directed channel that accepts a data item through its source end if it can instantly dispense it through its sink end. The LossySync channel behaves similarly except that it always accepts data items through its source end. The data item is transferred if it can be dispensed through the sink end, and is lost otherwise. The SyncDrain has two source ends and accepts data through them only simultaneously, and deletes them immediately. The AsyncDrain channel accepts data items through either of its two source ends, but never from both at the same time. The FIFO is an asynchronous channel with a buffer of capacity one. The basic set of Reo channels also includes ones with data dependent behavior or that perform data manipulation. For instance, the Filter channel loses the data item at its source end if the it does not match a certain pattern, which is defined in terms of a data constraint for a particular instance of this channel.

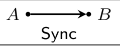
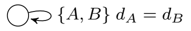
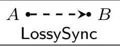
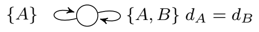
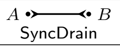
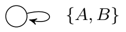
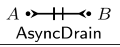
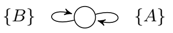
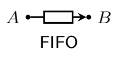
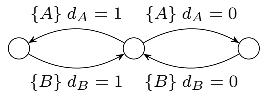
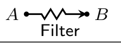
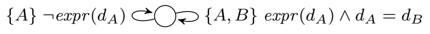
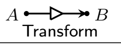
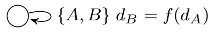
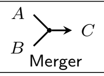
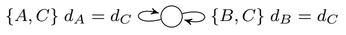
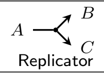
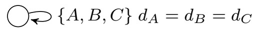
Primitive	Constraint automaton
 Sync	
 LossySync	
 SyncDrain	
 AsyncDrain	
 FIFO	
 Filter	
 Transform	
 Merger	
 Replicator	

Table I
GRAPHICAL NOTATION AND SEMANTICS FOR CHANNELS AND NODES

Furthermore, data manipulation can be implemented using the Transform channel. It applies a user-defined function to the data item received at its source end and yields the result at its sink end.

Channels can be joined together using nodes. A node can be a source, a sink or a mixed node, depending on whether all coinciding channel ends are source ends, sink ends or a combination of both. Source and sink nodes together form the boundary nodes of a connector, allowing interaction with its environment. Source nodes act as synchronous replicators, sink nodes as mergers. Mixed nodes combine both behaviors by atomically consuming a data item from one sink end at the time and replicating it to all source ends.

The Reo channels introduced above can be formally modeled using constraint automata [9]. The transitions in a constraint automaton are labeled with sets of ports that fire synchronously and with data constraints on these ports. Figure I depicts the graphical notation and the constraint automata semantics[1] for the basic channels and of the Merger and Replicator primitives, which can be used to construct nodes compositionally. The constraint automaton for the FIFO is shown with respect to the data domain $Data = \{0, 1\}$.

The application of Reo to business process modeling resembles those of Petri nets. Intuitively, a FIFO channel corresponds to a place with capacity one in a classical Petri net. The notion of a Petri net transition is generalized in Reo and can be composed of multiple synchronous channels. The constraint automata-based semantics for Reo

[1]Observe that constraint automata do not reflect properly the semantics of the LossySync channel suggesting that it can decide whether to lose or transfer data non-deterministically. This problem is known as the problem of context-dependency modeling in Reo. Several formalisms have been proposed to solve it. The detailed discussion of this issue is out of the scope of this paper.

is compositional, meaning that the behavior of a complex Reo circuit can be obtained from the semantics of its constituent parts using a product operator. This puts Reo in line with hierarchical Petri nets, suitable for modeling complex workflows as the models can be easily decomposed and their parts analyzed separately. Furthermore, a hiding operator can be used for abstracting from internal behavior. This abstraction mechanism can be used to turn a connector with observable data flow at its boundary ports into a component, which subsequently can be used for the design of more complex systems.

III. Model checking Reo with mCRL2

A strength of Reo as a workflow modeling language is its being amenable to formal verification. Currently, the most feature-complete and efficient way to analyze Reo process models is to generate a `mCRL2` specification and to use the corresponding verification facilities. In this section, we briefly describe `mCRL2` and its application to model checking Reo.

`mCRL2` is a specification language based on the process algebra ACP. The basic notion in `mCRL2` is the action. Actions represent atomic events and can be parametrized with data. Actions in `mCRL2` can be synchronized using the synchronization operator $|$. Synchronized actions are called multiactions. Processes are defined by process expressions, which are compositions of actions and multiactions using a number of operators. The basic operators include (i) *deadlock* or *inaction* δ, (ii) *alternative composition* $p+q$, (iii) *sequential composition* $p \cdot q$, (iv) *conditional* operator or *if-then-else* construct $c \to p \diamond q$ where c is a boolean expression, (v) *summation* $\Sigma_{d:D} p$ used to quantify over a data domain D, (vi) *at* operator $a@t$ indicating that multiaction a happens at time t, (vii) *parallel composition* $p \parallel q$, (viii) *encapsulation* $\partial_H(p)$, where H is a set of action names that are not allowed to occur, (ix) *renaming* operator $\rho_R(p)$, where R is a set of renamings of the form $a \to b$ and (x) *communication* operator $\Gamma_C(p)$, where C is a set of communications of the form $a_0|...|a_n \mapsto c$, which means that every group of actions $a_0|...|a_n$ within a multiaction is replaced by c.

The `mCRL2` language provides a number of built-in datatypes (e.g., boolean, natural, integer) with all standard arithmetic operations predefined. Moreover, a datatype definition mechanism allows users to declare new types or sorts. An arbitrary structured type in `mCRL2` can be declared by a construct of the form

$$\textbf{sort}\, S = \textbf{struct}\ c_1(p_1^1{:}S_1^1,\ \ldots,\ p_1^{k1}{:}S_1^{k1})?r_1 \mid \ldots \mid \\ c_n(p_n^1{:}S_n^1, \ldots, p_n^{kn}{:}S_n^{kn})?r_n\ ;$$

It defines the type S together with constructors $c_i\colon S_i^1 \times \ldots \times S_i^{ki} \to S$, projections $p_i^j\colon S \to S_i^j$, and type recognition functions $r_i\colon S \to Bool$.

The `mCRL2` toolset includes a tool for converting `mCRL2` specifications into linear form (a compact symbolic representation of the corresponding LTS to speed up subsequent manipulations), a tool for generating explicit LTSs from linear process specifications (LPS), tools for optimizing and visualizing these LTSs, and many other useful facilities. A detailed overview can be found at the `mCRL2` web site.

Sync	$= \Sigma_{d:Data}\ A(d)\vert B(d) \cdot \mathsf{Sync}$
LossySync	$= \Sigma_{d:Data}\ (A(d)\vert B(d) + A(d)) \cdot \mathsf{LossySync}$
SyncDrain	$= \Sigma_{d_1,d_2:Data}\ A(d_1)\vert B(d_2) \cdot \mathsf{SyncDrain}$
AsyncDrain	$= \Sigma_{d:Data}\ (A(d) + B(d)) \cdot \mathsf{AsyncDrain}$
FIFO$(f : DataFIFO)$	$= \Sigma_{d:Data}(isEmpty(f) \to A(d) \cdot \mathsf{FIFO}(full(d))$ $\diamond\ B(e(f)) \cdot \mathsf{FIFO}(empty))$
Filter	$= \Sigma_{d:Data}\ (expr(d) \to A(d)\vert B(d) \diamond A(d)) \cdot \mathsf{Filter}$
Transform	$= \Sigma_{d:Data}\ A(d)\vert B(f(d)) \cdot \mathsf{Transform}$
Merger	$= \Sigma_{d:Data}\ (A(d)\vert C(d) + B(d)\vert C(d)) \cdot \mathsf{Merger}$
Replicator	$= \Sigma_{d:Data}\ A(d)\vert B(d)\vert C(d) \cdot \mathsf{Replicator}$
XOR	$= \Sigma_{d:Data}\ (A(d)\vert B(d) + A(d)\vert C(d)) \cdot \mathsf{XOR}$
Join	$= \Sigma_{d_1,d_2:Data}\ A(d_1)\vert B(d_2)\vert C(tuple(d_1, d_2)) \cdot \mathsf{Join}$

Table II
mCRL2 ENCODING FOR CHANNELS AND NODES

For model checking, system properties are specified as formulae in a variant of the modal μ-calculus extended with regular expressions, data and time. In combination with an LPS such a formula is transformed into a parametrized boolean equation system (PBES) and can be solved with the appropriate tools from the toolset. Analysis at the level of LTS, in particular, deadlock detection or checking of the presence or absence of certain actions, is also possible.

A constraint automaton is essentially an LTS with labels representing two kinds of constraints: synchronization and data constraints. Synchronization constraints represent sets of Reo port names where data flow is observed simultaneously during a transition. Data constraints model conditions on transition enactment and show data assignments on port names. `mCRL2` models for Reo circuits can be generated in the following way: observable events, i.e., data flow on the channel ends, are represented as atomic actions, while data items observed at these ports are modeled as parameters of these actions. Analogously, we introduce a process for every node and actions for all channel ends meeting at the node. The encodings for the basic Reo channels and nodes are depicted in Table II.

In the context of a given connector, we assume a global datatype modeled as the custom sort $Data$ in `mCRL2`. Given such a datatype, we can use the `mCRL2` summation operator to define data dependencies imposed by channels. For the FIFO channel we additionally define the datatype

$$\textbf{sort}\, DataFIFO = \mathbf{struct} \\ empty?isEmpty \mid full(e{:}Data)?isFull$$

which allows us to specify whether the buffer of the channel is empty or full, and if it is full, what value is stored in it.

As in the constraint automata approach, we construct nodes compositionally out of the Merger and the Replicator primitives. A process for a node that behaves like an ExclusiveRouter is defined analogously. The exclusive router is not a primary primitive in Reo, but a circuit composed of basic Reo channels and nodes. However, the exclusive router is frequently used in circuit design. Therefore, we introduce a special notation for it. For dataflow modeling the node Join, comes in handy. This node synchronizes all ends of incoming channels, forms a tuple of data items received and replicates it to the source ends of all outgoing channels.

To handle the data structures formed by the Join node, we need to close our global datatype under tupling. Thus, if a circuit needs to coordinate services operating on domains $\mathcal{D}_1, \ldots, \mathcal{D}_n$ and contains one or more Join nodes with two incoming ends, we define

$$\begin{array}{l}\textbf{sort}\, \mathit{Data} = \mathbf{struct} \\ \quad D_1(e_1{:}\mathcal{D}_1) \mid ... \mid D_n(e_n{:}\mathcal{D}_n) \mid \mathit{tuple}(p_1{:}\mathit{Data}, p_2{:}\mathit{Data})\end{array}$$

More generally, when Join nodes with k incoming ends are present in a circuit, $\mathit{tuple}_k(p_1{:}\mathit{Data}, \ldots, p_k{:}\mathit{Data})$ is added to the definition of the global datatype.

Given process definitions for all channels and nodes, a joint process that models the complete Reo connector is built by forming a parallel composition of these processes and synchronizing actions for coinciding channel/node ends. Optionally, the mCRL2 hiding operator can be employed for abstracting the flow in the internal nodes. Channel/node end synchronization is performed using two the mCRL2 operators communication and encapsulation. For minimization of the intermediate state spaces while generating the mCRL2 specification, we exploit the structure of the circuit and build the process for the whole Reo connector in a stepwise fashion. In our previous work [14], we showed that the operational semantics of the mCRL2 specification obtained is equivalent to the constraint automata semantics of the Reo connector[2].

IV. Translation of Timed Reo

In this section, we extend the translation of Reo to mCRL2 to include timed channels. Timed channels are used in Reo for timing constraints on service interaction or process activity [13]. For instance, a deadline t for the availability of some data can be represented using a channel with a FIFO buffer that loses its data item after t units of time. Another example is a *timer channel* (denoted as •—○→•) that can be seen as an asynchronous blocking channel with internal states: when the timer is switched off, the channel consumes any value of sort Data, starts the timer and generates a special 'timeout' value at its sink end after a predefined amount of time. Its use is illustrated in Section VII.

Often it is useful to influence the behavior of a timed channel. To enable such control, we define channels that react in a special way to specific data inputs. For example, a so-called *timer with off and reset option* allows the timer to be stopped before the expiration of its delay when a special 'off' value is consumed through its source end. Similarly, the 'reset' option allows the timer to be reset to 0 when a special 'reset' value is consumed. The operational model for timed Reo channels is given by timed constraint automata [13], TCA for short. TCA essentially represent constraint automata with clock assignments and timing constraints. A TCA for a *timer with off and reset option* channel is shown in Figure 1(a). In this model, the state s represents a timer which is switched off while the state $\overline{s}$ corresponds to the timer being switched on. For the mapping of such timer channels to mCRL2, we need to capture off, reset and timeout signals and define

$$\begin{array}{l}\textbf{sort}\, \mathit{DataTimer} = \mathbf{struct}\ \mathit{reset}?\mathit{isReset} \mid \mathit{off}?\mathit{isOff} \mid \\ \quad \mathit{timeout}?\mathit{isTimeOut} \mid \mathit{other}(e{:}\mathit{Data})?\mathit{isOther}\end{array}$$

Timer channels also behave differently when switched on or switched off. Taking this into account, the timer with off and reset option can be specified as a parameterized process

$$\begin{array}{l}\mathit{Timer}(\mathit{isOFF}{:}\mathit{Bool}, x{:}\mathit{Real}, t{:}\mathit{Real}) = \\ \quad \mathit{isOFF} \to (\Sigma_{d:\mathit{DataTimer}}\ \mathit{isOther}(d) \to \\ \quad\quad A(d).\mathit{Timer}(\mathit{false}, 0, t)) \diamond \\ \quad ((x < t) \to (\Sigma_{d:\mathit{DataTimer}} \\ \quad\quad \mathit{isReset}(d) \to A(d).\mathit{Timer}(\mathit{false}, 0, t) + \\ \quad\quad \mathit{isOff}(d) \to A(d).\mathit{Timer}(\mathit{true}, x, t)\ + \\ \quad\quad \mathit{tick}@x.\mathit{Timer}(\mathit{false}, x+1, t)) \diamond \\ \quad\ B(\mathit{timeout}).\mathit{Timer}(\mathit{true}, x, t))\end{array}$$

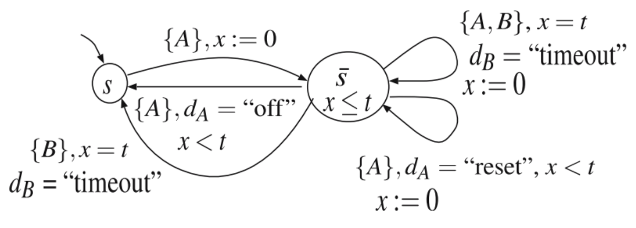
where $\mathit{isOFF}{:}\mathit{Bool}$ indicates whether the timer is off or on, x is the current time, t is the timer delay, A and B are source and sink ends of the channel and the action *tick* occurring at time x represents the progress of time.

Figure 1(b) shows an LTS obtained from the mCRL2 specification for the timer initially off, set to 0 and with a timeout delay of 3 time units. The initial state is labeled by 0. Other states in this LTS correspond to the situation where the timer is on and the time evolves from 1 to 3. In each of these states, the timer can be reset or switched off. The reset action does not change the state of the timer, i.e., it remains switched on, while the 'off' option makes the timer return to the initial state. We can abstract from the current time in this model by hiding the *tick* action, yielding the LTS shown in Figure 1(c).

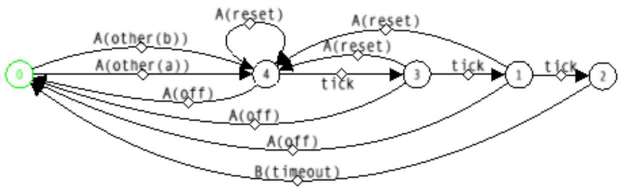
V. Abstraction and mCRL2 Code Minimization

Basic Reo channels together with timed channels are sufficiently expressive to enable the modeling of the major data flow patterns. However, Reo counterparts for some frequently used modeling primitives can be rather complex. For example, Figure 2(a) shows a circuit composed of 17 channels and 12 nodes that behaves as a variable: after an initial value for the variable is set via the port *write*, this value is stored in an internal buffer and can be repeatedly read at the port *read* or updated.

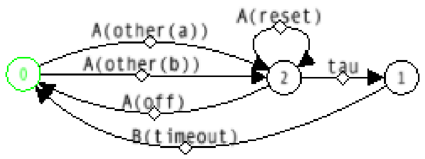
[2]In the same paper, we showed how to address the problem of context-sensitivity in Reo by propagating the information about the presence or absence of requests on the boundary nodes of a connector using parameterized actions.



(a) TCA



(b) LTS with observable time actions



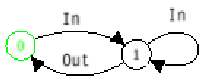
(c) LTS with hidden time actions

Figure 1. Semantics of the timer channel with off and reset options

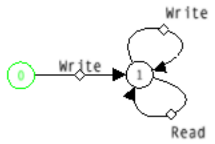
At a first glance, the design of Reo circuits may appear difficult. However, in most cases the compositional semantics of Reo allows for breaking down the task into pieces which are easier to manage. For example, Figure 2(b) shows an equivalent Reo model for the variable connector using two shift lossy FIFO buffers represented as separate components. Each *ShiftLossyFIFO* (see Figure 2(c)) accepts a data item at its *in* port and keeps it in an internal buffer until the data item is consumed at the *out* port. If a new data item arrives before the previous one has been consumed, the circuit loses the old data element and replaces it with the new one. By combining two buffers we obtain the behavior of the variable: a data item at the port *write* is replicated and stored in both *ShiftLossyFIFO* buffers. When a data item is taken from one buffer, a copy of the item from the second buffer is placed into the first, so the same value can be consumed again. Assigning a new value to the variable amounts to replacing the data items in both *ShiftLossyFIFO* subcomponents.

Since the semantics of Reo is compositional, we can construct hierarchical software models using component abstractions for lower-level connectors. Thus, a *shiftLossyFIFO* component is used to build the variable circuit. The variable circuit in its turn can be transformed into a component with two observable ports, *write* and *read*, and used in higher-level process models. In Section VII, we use such a component as part of an auction process.

The translation approach discussed in Section III generates one process for each channel and node. This yields 29 processes in total for the variable circuit. Additionally, 2 processes are created for the writer and reader components. For synchronizing actions corresponding to the ends of connected channels and nodes, `mCRL2` communication and hiding operators are applied 36 times over various stages. Figure 3 shows the data-agnostic LTSs for the *shiftLossyFIFO* and *variable* circuits produced by the `mCRL2` tools. However, it is noticed that the behavior observed at the external ports of these circuits is rather simple and can be expressed more directly. For example, the variable can be described by the `mCRL2` process $Var = Write.(Write + Read).Var$. The LTS for this process is exactly as shown in Figure 3(b).



(a) Shift lossy FIFO



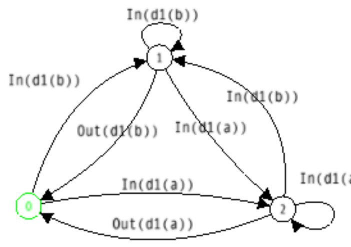
(b) Variable

Figure 3. Data agnostic LTS semantics

Assuming $Data = \{a, b\}$, the data-aware *shiftLossyFIFO* and *variable* circuits are denoted by the LTSs of Figure 4. Each state in these models corresponds to a value of the *shiftLossyFIFO* and *variable* components, and transition labels represent dataflow observed at the external ports. Similarly to the data-agnostic case, the process

$$Var(f{:}DataFIFO) = \\ \Sigma_{d:Data}\, Write(d).\, Var(full(d)) + Read(e(f)).\, Var(f)))$$

represents the behavior for the variable.



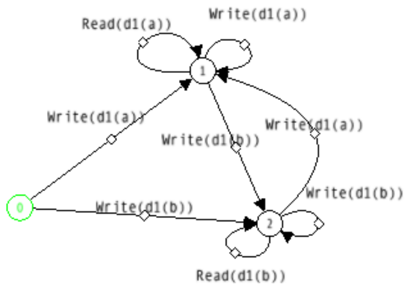
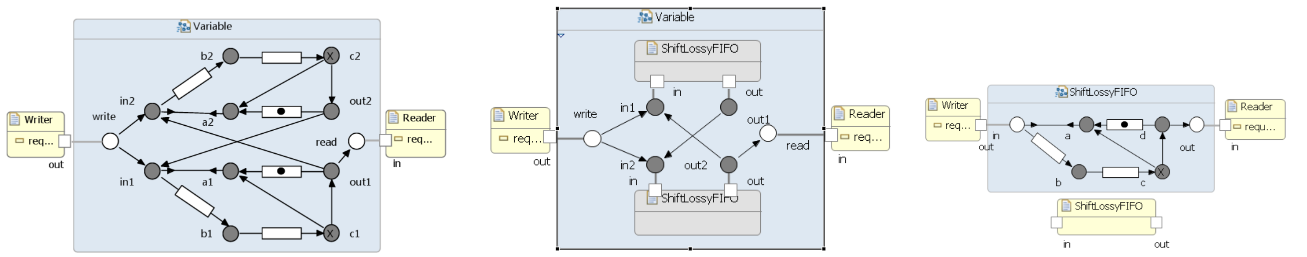


Figure 4. Data-aware LTS for variable (top) and shift-lossy FIFO (bottom)



(a) Plain model (b) Hierarchical model (c) Shift lossy FIFO and its component interface

Figure 2. Reo model of a variable

The converter from Reo to `mCRL2` presented in Section VI generates optimized `mCRL2` code for some predefined frequently used components (e.g., variables). In principle, the abstraction mechanism allows us to mimic the structure of high-level workflow modeling languages by defining components for their basic primitives such as various gateways, compensation pairs, and so on. This abstraction mechanism is also used for the formalization of BPMN and BPEL specifications using Reo [15]. In particular, BPEL variables can be represented in Reo as components with associated constraint automata semantics.

VI. Tool Support

ECT [16], short for Eclipse Coordination Tools, is a framework for modeling, verification and execution of component-based and service-oriented systems. It consists of a set of integrated tools for the Eclipse platform.[3] The framework provides functionality for the conversion of high-level modeling languages, such as UML, BPMN and BPEL to Reo, for editing and animation of Reo models, generation of automata-based semantical models and executable code from Reo and integrations with existing model checking tools.

We developed a converter that generates an `mCRL2` specification from a graphically given Reo circuit, potentially with the timer channels. A screenshot of the tool is shown in Figure 5. The generation of `mCRL2` code can be customized using various options. For instance, the option *with components* incorporates process definitions for the components attached at the boundary of a connector. The option *with data* enables data-aware encoding. Datatypes of components and services coordinated by Reo, as well as data constraints for data dependent channels such as Filter or Transform channels can be defined using the same interface. Note that these are saved as annotations in the Reo model and are propagated to the final `mCRL2` specification. This way Reo circuits can be compiled automatically into `mCRL2` without any manual editing.

The tool further includes an integration with model checking facilities of the `mCRL2` toolset and its state space visualization tools. In particular, we use the `mcrl22lps` tool for the generation of the linear process representation of `mCRL2` code, `lps2lts` and `lpsconvert` for generating and minimizing labeled transitions systems, `lps2pbes` for symbolic model checking of modal μ-calculus formulas, and finally `ltsgraph` for visualization of state spaces. Related verification tools, such as CADP[4] that relies on the same LTS format, can be called from within ECT.

The translation of Reo to `mCRL2` specification requires polynomial time from the number of basic channels in the model. The performance of the dataflow analysis depends on the time period and the complexity of the input domain and filter constraints which affect the total number of states in the final state space. To give you an idea about the toolset performance, the generation of an explicit state space for the timer channel with the delay in 1000000 time units requires around 11 minutes on a standard machine with 4 cores and 8GB of memory, running Linux 2.6.27 and the January 2010 release version of `mCRL2` (revision 201001). Note that the generation of the explicit state space is not needed if the `mCRL2` symbolic model checking utility based on the PBES solver is used.

VII. Case Study

In this section, we illustrate the approach to the verification of timed data-aware process models using Reo and `mCRL2`. As example we choose part of a typical auction process. In this process, a seller opens an auction that runs for a predetermined period. During this time, an auction participant can submit bids, and if the new bid is higher than the starting price and any bid submitted previously, it is accepted as such and stored in the system. After the auction period expires, the participant who submitted the highest bid is the winner and made known to the seller of the item.

The left part of Figure 5 shows a Reo model for this scenario and the generated `mCRL2` specification. In this model, the first writer, on the left, instantiates the auction. The second writer represents a component that models a bidder. The reader component corresponds to the seller and consumes the outcome of the auction. A component *Variable* is used to hold the information about the current state of

[3]http://www.eclipse.org

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

the auction. A timer channel with ports *startAuction* and *timeExpired* is used to control the lifetime of the auction. It accepts any data item of the global sort *Data* and delays its output for a predefined amount of time, i.e., 5 time units. During this period, the value of the variable can be updated with the information regarding the highest submitted bid. This behavior is realized by a synchronous drain with a filter condition connecting ports *readBidInfo* and *newBid*. Since the semantics of the variable does not allow us to read and write to it simultaneously, a FIFO channel is needed to store the item till the next step. When the timeout expires, the timer channel disposes a token through its sink end that together with the output from the variable component enables the synchronous drain to fire and to send the result of the auction to the seller of the item.

Figure 6(a) shows a data-agnostic LTS for the auction circuit with data flow observable at ports *startAuction*, *submitBid*, *announceTheWinner*, *writeBidInfo* and *readBidInfo*. Figure 6(b) is a time-aware version of the model where data flow at the external ports *startAuction*, *submitBid*, *announceTheWinner* is shown along with the *tick* action representing progress of time.

As such, the models are suitable for verifying temporal and timed properties. For example, we can check whether for every auction the seller will eventually receive notification using the μ-calculus formula

$$[\, \mathit{true}^* \cdot \mathit{startAuction}.\mathit{true}^* \,] \\ \langle\, \mathit{true}^* \cdot \mathit{announceTheWinner} \cdot \mathit{true}^* \,\rangle\; \mathit{true}\,.$$

With a slightly modified version of this property

$$[\, \mathit{true}^* \cdot \mathit{startAuction}@0 \cdot \mathit{true}^* \,] \\ \langle \mathit{true}^* \cdot \mathit{announceTheWinner}@5 \cdot \mathit{true}^* \rangle\; \mathit{true}$$

one can verify that the auction runs exactly 5 units of time.

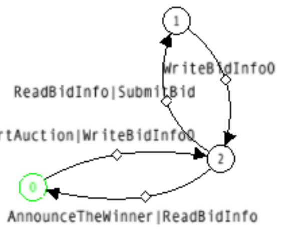
Figure 6(c) shows a data-aware LTS for the circuit. In this model, transitions are labeled with parametrized actions where action parameters represent data values observed on the selected circuit ports. For reasons of presentation, we restrict the input domain of the bidder to tuples $(\mathit{bidderID}, \mathit{bidderPrice})$ satisfying the constraint

$$(\mathit{bidderID}(d) > 0) \wedge (\mathit{bidderPrice}(d) < 2) \wedge \\ (\mathit{bidderPrice}(d) = \mathit{bidderID}(d))\,.$$

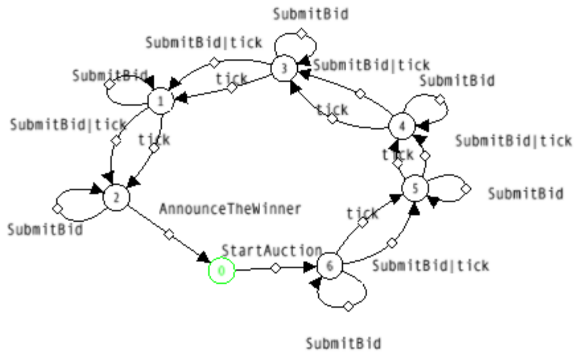
We can update our property with the information about data parameters of the actions observed at the circuit ports:

$$\forall d{:}\mathit{Data}\,. \\ \mathit{val}((\mathit{bidderID}(e_1(d)) > 0) \wedge (\mathit{bidderPrice}(e_1(d)) < 2) \wedge \\ (\mathit{bidderPrice}(e_1(d)) = \mathit{bidderID}(e_1(d)))) \wedge \\ [\, \mathit{true}^* {\cdot} \mathit{startAuction}(d) {\cdot} \mathit{true}^* \,] \\ \langle \mathit{true}^* {\cdot} \mathit{announceTheWinner}(d) {\cdot} \mathit{true}^* \rangle\; \mathit{true}\,.$$

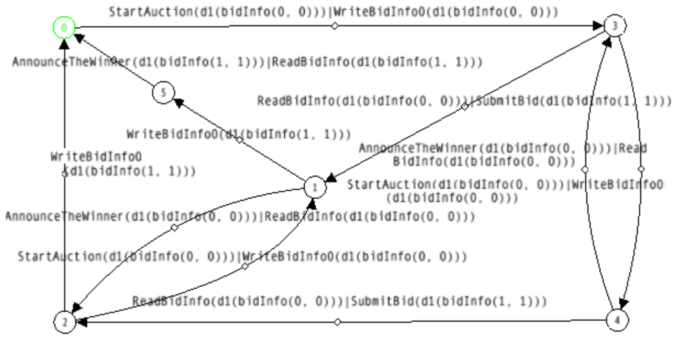
The boolean function $\mathit{val}(\mathit{expr}{:}\mathit{Bool})$, provided in the `mCRL2` property specification language, is used to evaluate a boolean condition that restricts the domain of the data



(a) Data-agnostic LTS



(b) LTS with observable timer actions


StartAuction(d1(bidInfo(0, 0)))|WriteBidInfo0(d1(bidInfo(0, 0)))
0
3
AnnounceTheWinner(d1(bidInfo(1, 1)))|ReadBidInfo(d1(bidInfo(1, 1)))
5
ReadBidInfo(d1(bidInfo(0, 0)))|SubmitBid(d1(bidInfo(1, 1)))
WriteBidInfo0(d1(bidInfo(1, 1)))
AnnounceTheWinner(d1(bidInfo(0, 0)))|ReadBidInfo(d1(bidInfo(0, 0)))
WriteBidInfo0
(d1(bidInfo(1, 1)))
1
StartAuction(d1(bidInfo(0, 0)))|WriteBidInfo0
(d1(bidInfo(0, 0)))
AnnounceTheWinner(d1(bidInfo(0, 0)))|ReadBidInfo(d1(bidInfo(0, 0)))
StartAuction(d1(bidInfo(0, 0)))|WriteBidInfo0(d1(bidInfo(0, 0)))
ReadBidInfo(d1(bidInfo(0, 0)))|SubmitBid(d1(bidInfo(1, 1)))
2
4

(c) LTS with data flow

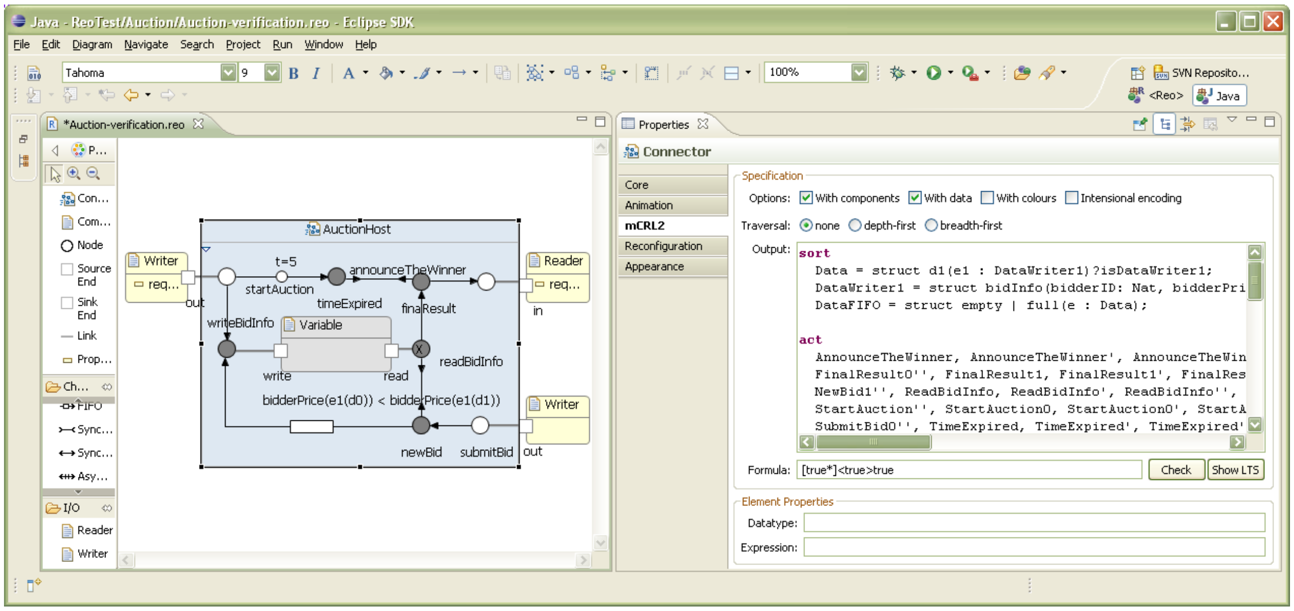
Figure 6. Semantics of the auction process

parameter. Here, e_1 is a projection function used to retrieve necessary information from a data item of sort Data.

Another Reo model for the given scenario is shown in Figure VII. The model illustrates the use of data manipulation operations as supported in our verification framework. The data provided by the first writer is described by the data structure

$$\textbf{sort}\; \mathit{DataWriter1} = \textbf{struct}\; \mathit{auctionInfo}\,(\\ \mathit{auctionID}{:}\mathit{Pos},\; \mathit{auctionOpen}{:}\mathit{Bool},\; \mathit{startPrice}{:}\mathit{Nat}, \\ \mathit{bidderID}{:}\mathit{Nat},\; \mathit{bidderPrice}{:}\mathit{Nat})?\mathit{isAuctionInfo};$$

Here an *auctionID* is a positive number that identifies the auction instance, *auctionOpen* is a boolean variable which is set to true iff the auction is active, and *bidderID* and *bidderPrice* represent the information about the bidder



Specification

Options: [✓ With components] [✓ With data] With colours Intensional encoding

Traversal: [✓ none] depth-first breadth-first

Output:

```
sort
  Data = struct d1(e1 : DataWriter1)?isDataWriter1;
  DataWriter1 = struct bidInfo(bidderID: Nat, bidderPri
  DataFIFO = struct empty | full(e : Data);

act
  AnnounceTheWinner, AnnounceTheWinner', AnnounceTheWin
  FinalResult0'', FinalResult1, FinalResult1', FinalRes
  NewBid1'', ReadBidInfo, ReadBidInfo', ReadBidInfo'',
  StartAuction'', StartAuction0, StartAuction0', StartA
  SubmitBid0'', TimeExpired, TimeExpired', TimeExpired'
```

Formula: [true*]<true>true

Element Properties

Datatype:

Expression:

Figure 5. Reo model of an auction process and its mCRL2 specification

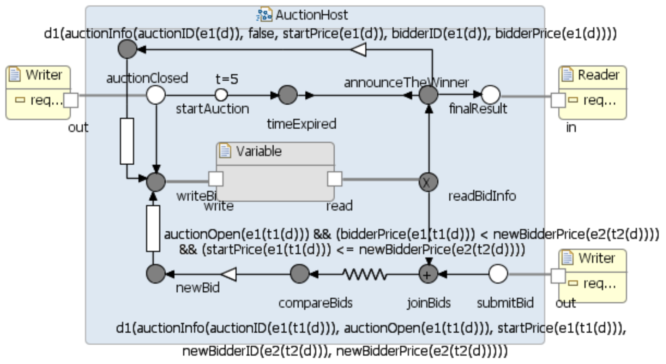

AuctionHost
d1(auctionInfo(auctionID(e1(d)), false, startPrice(e1(d)), bidderID(e1(d)), bidderPrice(e1(d))))
Writer
req...
out
auctionClosed
t=5
startAuction
timeExpired
announceTheWinner
finalResult
Reader
req...
in
Variable
writeBi
write
read
readBidInfo
auctionOpen(e1(t1(d))) && (bidderPrice(e1(t1(d))) < newBidderPrice(e2(t2(d))))
&& (startPrice(e1(t1(d))) <= newBidderPrice(e2(t2(d))))
Writer
req...
out
newBid
compareBids
joinBids
submitBid
d1(auctionInfo(auctionID(e1(t1(d))), auctionOpen(e1(t1(d))), startPrice(e1(t1(d))),
newBidderID(e2(t2(d))), newBidderPrice(e2(t2(d)))))

Figure 7. Reo model of an auction process with data transformations

who submitted the highest bid and the amount offered. The constructor *auctionInfo* and the recognition function *isAuctionInfo* are used to create an object d of the proper type and, given a data item d, check if it is of the type *auctionInfo*, respectively.

Input for the bidder component is of type

$$\textbf{sort}\ \mathit{DataWriter2} = \textbf{struct}\ \mathit{bidInfo}\,(\\ \quad \mathit{newBidderID}{:}\mathit{Pos},\ \mathit{newBidderPrice}{:}\mathit{Pos})?\mathit{isBidInfo};$$

where *newBidderID* and *newBidderPrice* are used to identify the bidder and the proposed price.

In the model we do not limit the format in which the output is provided to the reader. Therefore, the global data domain on which the circuit operates is determined by the data types provided by the writers. Since the circuit contains also the join node *joinBids* the global data domain is described as

$$\textbf{sort}\ \mathit{Data} = \textbf{struct}\ d_1(e_1{:}\mathit{DataWriter1})?\mathit{isDataWriter1}\ |\\ \quad d_2(e_2{:}\mathit{DataWriter2})?\mathit{isDataWriter2}\ |\\ \quad \mathit{tuple}(t_1{:}\mathit{Data},\ t_2{:}\mathit{Data})?\mathit{isTuple};$$

When a new bid is submitted, the information about the current state of the auction and the new bid is merged using the join node *joinBids*. The resulting tuple $\mathit{tuple}(d_1, d_2)$, with d_1 a value of sort *DataWriter1* and d_2 a value of sort *DataWriter2*, passes through the filter channel with the condition

$$\mathit{auctionOpen}(e_1(t_1(d))) \wedge\\ (\mathit{bidderPrice}(e_1(t_1(d))) < \mathit{newBidderPrice}(e_2(t_2(d)))) \wedge\\ (\mathit{startPrice}(e_1(t_1(d))) \leq \mathit{newBidderPrice}(e_2(t_2(d))))$$

iff the auction is open, and the price of the new bid is at least the start price and exceeds the previously submitted bid. If the condition holds for a given input, the subsequent transformer channel gets the tuple, and transforms it to the value of sort *DataWriter1* using the combination of the construction and projection functions

$$\mathit{auctionInfo}(\mathit{auctionID}(e_1(t_1(d))),\\ \quad \mathit{auctionOpen}(e_1(t_1(d))),\ \mathit{startPrice}(e_1(t_1(d))),\\ \quad \mathit{newBidderID}(e_2(t_2(d))),\ \mathit{newBidderPrice}(e_2(t_2(d))))$$

Since all actions operate on the sort *Data*, this value then is wrapped using the constructor d_1.

When the auction time expires, its status is changed to ‘closed’ by the transformer channel that implements

$$d_1(\mathit{auctionInfo}(\mathit{auctionID}(e_1(d)), \mathit{false}, \mathit{startPrice}(e_1(d)),\\ \quad \mathit{bidderID}(e_1(d)), \mathit{bidderPrice}(e_1(d))))$$

Similar to the FIFO channel with ports *newBidInfo* and *writeBidInfo*, the FIFO channel with ports *auctionClosed* and *rewriteBidInfo* is used to store the modified data element till the next step, for comsumption by the variable component.

VIII. Related Work

There are various tools for the analysis of workflow models using Petri net based semantics. Among the most well known verification tools for Petri nets are Woflan [17] and CPN Tools [18]. Woflan provides means for the verification of logical correctness of workflow nets, a restricted version of Petri nets for process modeling. CPN Tools represent a toolkit for editing, simulating and analyzing colored Petri nets. Yasper [19] is a modeling framework that supports Petri nets extended with inhibitor and reset arcs. Raedts et al. [20] uses the `mCRL2` toolset for analyzing Yasper models. However, the translation proposed in this work only considers the number of data tokens, but not their content.

The tools mentioned above provide excellent support for process control flow analysis, but do not target at dataflow verification. For dataflow modeling, Hidders et al. [3], [21] extend Petri nets with nested relational calculus, a database query language over composite data types. The authors introduce a set of refinement rules which, being applied hierarchically, guarantee that the final workflow net is sound. Similar to this approach, we deal with concrete data items rather than abstract tokens. However, [3], [21] do not provide integration with model checking tools for verification of the constructed dataflow models. Trčka et al. [4] consider workflow correctness criteria analyzing Petri net-based models extended with *read*, *write* and *destroy* operations to enable mixed control- and dataflow analysis. This work is conceptually close to ours. Reader and writer components in our framework perform the same functions as read and write operations in this work, while the loss of data by lossy or synchronous drain channels is similar to their destruction. The authors provide a set of CTL* properties to identify some common workflow errors. Being based on μ-calculus, the `mCRL2` property specification format subsumes CTL*, and, therefore we believe that corresponding properties can be stated and verified in our framework as well. Additionally, `mCRL2` provides means for further data manipulation not covered by [4].

TINA [22] and Roméo [23] are frameworks for the verification of timed Petri nets. With respect to timed analysis, ECT is not as mature as these tools. However, the compositional semantics of Reo and its extensibility make the toolset capable of analyzing timed data-aware behavior on hierarchical process models. Despite the abundance of workflow management systems and Petri net analyzing tools, at the moment none of them integrates all these facilities.

Kazhamiakin et al. [24], [25] present approaches for the modeling and analysis of data- and time-related properties of web service compositions. Here, BPEL-based service models are represented as state transition systems augmented with data and time constraints and verified using NuSMV and UPPAAL. This is suitable for the verification of service compositions already implemented, but does not support the modeling of choreographies that require data and time-aware coordination. Models and frameworks were developed to check service/process compatibility, including data-aware and time-aware business protocol compatibility in [26], [27], [28]. Our work generalizes these approaches by providing a graphical language that allows designers to build connectors and, thus, create valid service compositions from services that may not be directly compatible. The paper [29] provides an overview of efforts within the Sensoria project on exploiting the COWS process algebra as stepping stone towards further tooling for the design and verification of service architectures. The temporal logic UCTL dedicated to the verification of service-oriented applications, again involving COWS, and its model checker UMC, are discussed in [30]. Exogenous coordination as underlying Reo is also advocated in [31], where a modal logic is proposed specific to connectors. However, at present, data and time are not supported in this approach. Kemper [32] presents a SAT-based approach for bounded model checking of TCA [13]. In this work, the behavior of a TCA is represented by formulae in propositional logic with linear arithmetic to be analyzed by various SAT solvers. Since TCA provide operational semantics for timed Reo, this approach can be used for model checking time properties of Reo connectors. However, at the moment there is no tool for generating TCA from graphical data-aware Reo circuits.

IX. Conclusions

In this paper, we presented the mapping of timed Reo connectors to the process algebra `mCRL2`, and discussed its application to timed data-centric workflow analysis. The `mCRL2` toolset supports efficient full-featured model checking for Reo. Together with other tools from ECT, our work provides a user-friendly environment for graphical modeling of component/service-based systems and business processes, which relieves developers from the need to encode the behavior of their systems in the scripting language `mCRL2` directly. Larger case studies are to be conducted to establish the scalability of our approach, for example, for Reo to become widely accepted by the business process modeling community as a foundation for graphical workflow design. In potential, we feel, it offers an approach complementary to Petri nets for the verification of service-based process models.

References

[1] A. Haller, E. Oren, and S. Petkov, "Survey of workflow management systems," DERI, Tech. Rep. 2005-05-02, 2005.

[2] S. Gorton, C. Montangero, Reiff-Marganiec, S., and L. Semini, “StPowla: SOA, policies and workflows,” in *Proc. ICSOC Workshops*. LNCS 4907, 2007, pp. 351–362.

[3] J. Hidders, N. Kwasnikowska, J. Sroka, J. Tyszkiewicz, and J. den Bussche, “Petri net + nested relational calculus = dataflow,” in *Proc. CoopIS*. LNCS 3760, 2005, pp. 220–237.

[4] N. Trčka, W. van der Aalst, and N. Sidorova, “Analyzing control-flow and data-flow in workflow processes in a unified way,” Technische Universiteit Eindhoven, Tech. Rep., 2008.

[5] F. Arbab, “The IWIM model for coordination of concurrent activities,” in *Proc. COORDINATION’96*, P. Ciancarini and C. Hankin, Eds. LNCS 1061, 1996, pp. 34–56.

[6] ——, “Reo: A channel-based coordination model for component composition,” *Mathematical Structures in Computer Science*, vol. 14, pp. 329–366, 2004.

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

[8] N. Kokash and F. Arbab, “Formal behavioral modeling and compliance analysis for service-oriented systems,” in *Proc. FMCO 2008*, F. de Boer, M. Bonsangue, and E. Madeleine, Eds. LNCS 5751, 2009, pp. 21–41.

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

[10] D. Clarke, D. Costa, and F. Arbab, “Connector coloring I: Synchronization and context dependency,” *Science of Computer Programming*, vol. 66, pp. 205–225, 2007.

[11] N. Kokash, C. Krause, and E. de Vink, “Data-aware design and verification of service composition with Reo and mCRL2,” in *Proc. SAC 2010, Sierre, March 21–26, 2010*. ACM Press, 2010, pp. 2406–2413.

[12] J. Groote et al., “The formal specification language mCRL2,” in *Methods for Modelling Software Systems*, E. Brinksma et al., Ed. IBFI, Schloss Dagstuhl, 2007.

[13] F. Arbab, C. Baier, F. de Boer, and J. Rutten, “Models and temporal logical specifications for timed component connectors,” *Software and Systems Modeling*, vol. 6, pp. 59–82, 2007.

[14] N. Kokash, C. Krause, and E. de Vink, “Verification of context-dependent channel-based service models,” in *Proc. FMCO 2009*, F. d. Boer, M. Bonsangue, S. Hallerstede, and M. Leuschel, Eds. LNCS, 2010, 20pp. To appear.

[15] B. Changizi, N. Kokash, and F. Arbab, “A unified toolset for business process model formalization,” in *Proc. FESCA’10*. ENTCS, to appear.

[16] F. Arbab et al., “Modeling, testing and executing Reo connectors with the Eclipse Coordination Tools,” Tool demo session at FACS ’08, 2008.

[17] W. van der Aalst, “Woflan: a Petri-net-based workflow analyzer,” *Syst. Anal. Model. Simul.*, vol. 35, pp. 345–357, 1999.

[18] A. Ratzer et al., “CPN tools for editing, simulating, and analysing coloured Petri nets,” in *Proc. ICATPN 2003*, W. van der Aalst and E. Best, Eds. LNCS 2679, 2003, pp. 450–462.

[19] K. Hee van, R. Post, and L. Somers, “Yet another smart process editor,” in *Proc. ESM’05*, J. Feliz-Teixeira and A. Carvalho Brito, Eds., 2005, pp. 527–530.

[20] I. Raedts et al., “Transformation of BPMN models for behaviour analysis,” in *Proc. MSVVEIS 2007*, J. Augusto, J. Barjis, and U. Ultes-Nitsche, Eds. INSTICC Press, 2007, pp. 126–137.

[21] J. Hidders, N. Kwasnikowska, J. Sroka, J. Tyszkiewicz, and J. den Bussche, “Dfl: A dataflow language based on Petri nets and nested relational calculus,” *Information Systems*, vol. 33, pp. 261–284, 2008.

[22] B. Berthomieu, P. Ribet, and F. Vernadat, “The tool TINA construction of abstract state spaces for Petri nets and time Petri nets,” *International Journal of Production Research*, vol. 42, 2004.

[23] D. Lime, O. Roux, C. Seidner, and L.-M. Traonouez, *Roméo: A Parametric Model-Checker for Petri Nets with Stopwatches*. LNCS 5505, 2009, pp. 54–57.

[24] R. Kazhamiakin, P. K. Pandya, and M. Pistore, “Timed modelling and analysis in web service compositions,” in *ARES’06*, 2006, pp. 840–846.

[25] R. Kazhamiakin and M. Pistore, “Static verification of control and data in web service compositions,” in *ICWS’06*, 2006, pp. 83–90.

[26] M. J. Ibanez, P. Alvarez, and J. Ezpeleta, “Flow and data compatibility for the correct interaction between web processes,” in *Proc. CIMCA-IAWTIC-ISE*, M. Mohammadian, Ed. IEEE, 2008, pp. 715–721.

[27] N. Guermouche, O. Perrin, and C. Ringeissen, “Timed specification for web services compatibility analysis,” *ENTCS*, vol. 200, pp. 155–170, 2008.

[28] N. Guermouche and C. Godart, “Asynchronous timed web service-aware choreography analysis,” in *Proc. CAiSE*, P. van Eck et al., Ed. LNCS 5565, 2009, pp. 364–378.

[29] L. Bocchi et al., “From architectural to behavioural specification of services,” *ENTCS*, vol. 253, pp. 3–21, 2009.

[30] M. ter Beek et al., “CMC-UMC: A framework for the verification of abstract service oriented properties,” in *Proc. SAC’09*. ACM, 2009, pp. 2111–2117.

[31] M. Barbosa, L. Barbosa, and J. Campos, “A coordination model for interactive components,” in *Proc. FSEN 2009*, F. Arbab and M. Sirjani, Eds. LNCS 5961, 2010, pp. 416–430.

[32] S. Kemper, “SAT-based verification for timed component connectors,” *ENTCS*, vol. 255, pp. 103–118, 2009.