On the Expressiveness of Event-Based Coordination Media

Mirko Viroli Andrea Omicini Alessandro Ricci

Abstract—In the context of component-based systems, coordination can be thought of as an event-based service provided by a coordination medium. However, this interpretation is not endorsed by some researches in the field of coordination and in particular by existing attempts to define a notion of expressiveness for coordination models. So, in this paper we elaborate on providing a new characterisation of expressiveness for coordination models, which takes into account the ability of a coordination medium to support (possibly complex) event-based coordination mechanisms, capturing and generalising features of both data-driven and control-driven coordination models. Based on this characterisation, a formalism for denoting coordination media is developed and is proved to be fully expressive. Examples are provided that show its ability to represent some of the main features and mechanisms of well-known models, such as Linda primitives, event notification, and expiring data.

Index Terms—Expressiveness of coordination models, tuple-based coordination, event notification, formal models, interactive computing

I. INTRODUCTION

Coordination models have been proposed as a successful means for harnessing the complexity of interactions in component-based systems [1]. In particular, coordination models provide the proper abstractions to rule and govern the interactions between their coordinated entities, by defining the conceptual locus — called the coordination medium — where dependencies among entities can be embedded and engineered, promoting a conceptual separation of algorithmic and interactive aspects [2].

As argued in [1], component-based systems are by nature event-driven, that is, their components interact with each other by means of events posting and receiving. These interactions act not only cause an exchange of information, but also drive the triggering of activities within the whole component-based system. As a result, applying coordination models to component-based systems generally calls for a rather new viewpoint on coordination, in terms of an event-based service provided to the system components by a coordination medium.

One of the issues that needs to be reconsidered in this context is that of a coordination model expressiveness. We claim that existing characterisations [3], [4], [5] do not provide for a suitable tool to compare the expressiveness of coordination models in the context of event-based scenarios.

In Section II we start from the idea of characterising a coordination model in terms of a coordination medium re-

This work has been partially supported by MIUR, and by Nokia Research Center, Burlington, MA, USA.

M. Viroli (presenter), A. Omicini, and A. Ricci are with DEIS, Università degli Studi di Bologna, via Rasi e Spinelli 176, 47023 Cesena (FC), Italy – email: {mviroli,aomicini,aricci}@deis.unibo.it

ceiving input communication events and reactively sending output communication events, thus capturing the common behaviour of a wide set of data-driven and control-driven coordination models. Based on this ontology, in Section III we provide a new framework for reasoning about the expressiveness of coordination models, which takes as a foundation Interactive Computing as defined by Wegner [6]. The expressiveness of a coordination model is characterised in terms of the histories of input and output communication events that its medium can exhibit. Accordingly, full-expressiveness of a formalism for defining coordination media is defined in terms of Wegner’s Sequential Interaction Machines (SIM) [7] — that is, in short, as the ability of exhibiting any computable interaction history. This definition accounts for the ability of a coordination framework to specify any kind of complex event-based coordination service, improving existing notions of expressiveness based on the concept of program transformation.

In order to deepen our investigation, a core formalism for coordination media called σRST is developed in Section IV, which is inspired by ReSpecT [8], a language for programming the behaviour of tuple centres [9]. In Section V some examples are provided to show the ability of σRST to encode common mechanisms and features of existing coordination models, such as Linda primitives, event-notification abilities similar to that of JavaSpaces [10] or IWIM channels [11], and the concept of expiring data in shared dataspaces [12]. Finally, in Section VI, σRST is proven to be fully expressive, thus stating its power as a framework for defining complex event-based coordination services.

II. EXTERNAL CHARACTERISATION

In this paper we take the general viewpoint of a coordination model expressed in terms of coordination media aggregating coordinated entities [2]. In particular, following the approaches in [13], [14], we focus on considering one medium interacting with several entities by means of an event space. To this end, the coordination medium is seen as an interactive component / software abstraction whose dynamics is described in terms of the reception of input (communication) events from the coordinated entities, of changes to the medium state, and of the transmission of output (communication) events to the coordinated entities.

The following syntactic conventions are adopted. Given a set \( X \) we generally let meta-variable \( x \) range over \( X \), \( \hat{x} \) range over the set \( \hat{X} \) of finite sequences over \( X \), \( \overline{x} \) over the set \( \overline{X} \) of multisets over \( X \), and \( x_\bot \) over the set \( X_\bot = X \cup \{ \perp \} \), where \( \perp \) is (or \( \bot \) for short) is typically considered an exception value over \( X \). Union of multisets \( \overline{X} \) and \( \overline{Y} \) is denoted by \( \overline{X} \cup \overline{Y} \), concatenation of sequences \( \hat{x} \) and \( \overline{x} \) by \( \hat{x} \cdot \overline{x} \).

On the Expressiveness of Event-Based Coordination Media

Mirko Viroli Andrea Omicini Alessandro Ricci

Abstract—In the context of component-based systems, coordination can be thought of as an event-based service provided by a coordination medium. However, this interpretation is not endorsed by some researches in the field of coordination and in particular by existing attempts to define a notion of expressiveness for coordination models. So, in this paper we elaborate on providing a new characterisation of expressiveness for coordination models, which takes into account the ability of a coordination medium to support (possibly complex) event-based coordination mechanisms, capturing and generalising features of both data-driven and control-driven coordination models. Based on this characterisation, a formalism for denoting coordination media is developed and is proved to be fully expressive. Examples are provided that show its ability to represent some of the main features and mechanisms of well-known models, such as Linda primitives, event notification, and expiring data.

Index Terms—Expressiveness of coordination models, tuple-based coordination, event notification, formal models, interactive computing

I. INTRODUCTION

Coordination models have been proposed as a successful means for harnessing the complexity of interactions in component-based systems [1]. In particular, coordination models provide the proper abstractions to rule and govern the interactions between their coordinated entities, by defining the conceptual locus — called the coordination medium — where dependencies among entities can be embedded and engineered, promoting a conceptual separation of algorithmic and interactive aspects [2].

As argued in [1], component-based systems are by nature event-driven, that is, their components interact with each other by means of events posting and receiving. These interactions act not only cause an exchange of information, but also drive the triggering of activities within the whole component-based system. As a result, applying coordination models to component-based systems generally calls for a rather new viewpoint on coordination, in terms of an event-based service provided to the system components by a coordination medium.

One of the issues that needs to be reconsidered in this context is that of a coordination model expressiveness. We claim that existing characterisations [3], [4], [5] do not provide for a suitable tool to compare the expressiveness of coordination models in the context of event-based scenarios.

In Section II we start from the idea of characterising a coordination model in terms of a coordination medium re-

This work has been partially supported by MIUR, and by Nokia Research Center, Burlington, MA, USA.

M. Viroli (presenter), A. Omicini, and A. Ricci are with DEIS, Università degli Studi di Bologna, via Rasi e Spinelli 176, 47023 Cesena (FC), Italy – email: {mviroli,aomicini,aricci}@deis.unibo.it

ceiving input communication events and reactively sending output communication events, thus capturing the common behaviour of a wide set of data-driven and control-driven coordination models. Based on this ontology, in Section III we provide a new framework for reasoning about the expressiveness of coordination models, which takes as a foundation Interactive Computing as defined by Wegner [6]. The expressiveness of a coordination model is characterised in terms of the histories of input and output communication events that its medium can exhibit. Accordingly, full-expressiveness of a formalism for defining coordination media is defined in terms of Wegner’s Sequential Interaction Machines (SIM) [7] — that is, in short, as the ability of exhibiting any computable interaction history. This definition accounts for the ability of a coordination framework to specify any kind of complex event-based coordination service, improving existing notions of expressiveness based on the concept of program transformation.

In order to deepen our investigation, a core formalism for coordination media called σRST is developed in Section IV, which is inspired by ReSpecT [8], a language for programming the behaviour of tuple centres [9]. In Section V some examples are provided to show the ability of σRST to encode common mechanisms and features of existing coordination models, such as Linda primitives, event-notification abilities similar to that of JavaSpaces [10] or IWIM channels [11], and the concept of expiring data in shared dataspaces [12]. Finally, in Section VI, σRST is proven to be fully expressive, thus stating its power as a framework for defining complex event-based coordination services.

II. EXTERNAL CHARACTERISATION

In this paper we take the general viewpoint of a coordination model expressed in terms of coordination media aggregating coordinated entities [2]. In particular, following the approaches in [13], [14], we focus on considering one medium interacting with several entities by means of an event space. To this end, the coordination medium is seen as an interactive component / software abstraction whose dynamics is described in terms of the reception of input (communication) events from the coordinated entities, of changes to the medium state, and of the transmission of output (communication) events to the coordinated entities.

The following syntactic conventions are adopted. Given a set \( X \) we generally let meta-variable \( x \) range over \( X \), \( \hat{x} \) range over the set \( \hat{X} \) of finite sequences over \( X \), \( \overline{x} \) over the set \( \overline{X} \) of multisets over \( X \), and \( x_\bot \) over the set \( X_\bot = X \cup \{ \perp \} \), where \( \perp \) is (or \( \bot \) for short) is typically considered an exception value over \( X \). Union of multisets \( \overline{X} \) and \( \overline{Y} \) is denoted by \( \overline{X} \cup \overline{Y} \), concatenation of sequences \( \hat{x} \) and \( \overline{x} \) by \( \hat{x} \cdot \overline{x} \).
by \( \check{\tau}; \check{z}' \), void multiset and empty sequence by \( \bullet \). As a syntactic facility, we also assume \( \perp \in \mathcal{P} = \check{\tau} \).

A coordination medium operational semantics can be given in terms of the transition system \((C, \rightarrow_{C}, Ie \times \mathcal{Oe})\), where \( \rightarrow_{C} \subseteq C \times (Ie \times \mathcal{Oe}) \times C \), \( Ie \) is the set of input events, and \( \mathcal{Oe} \) is the set of output events. In particular, elements \([id, v]^I\) denote input events sent from a coordinated entity to the medium, where \( id \in Id \) is the identifier of the sending coordinated entity, and \( v \in V \) is a generic value representing the event content. Similarly, elements \([id, v]^O\) denote output events, where \( id \) is the receiving entity and \( v \) is the event content. In particular, by the notation \( c \xrightarrow{\text{ie, } v \in V} C \) — which is a form equivalent to \( \langle c, \text{ie, } v \rangle, c' \rangle \in \rightarrow_{C} \) — we mean that the coordination medium in state \( c \) atomically (i) receives input event \( \text{ie} \), (ii) moves to state \( c' \), and (iii) sends the multiset of output events \( \mathcal{Oe} \).

Being inspired by the general semantic framework presented in [13], this schema makes it possible to represent the interactions between coordinated entities and coordination medium as they occur in a wide set of existing coordination models. For instance, in tuple-based models such as Linda (which is analysed in detail in Section A), the medium can receive an input event and reactively send: (i) no output events (e.g., when receiving a request for an unsatisfied \( \text{in} \)), (ii) one output event (e.g., due to a satisfied \( \text{rdp} \) immediately providing a reply), or (iii) a number of output events to other entities (e.g., when an \( \text{out} \) causes many replies to previously requested \( \text{rd} \)). Typical aspects of control-oriented coordination models [11], [15] can be represented as well. As an example, event-notification can be easily represented (see Section B): in JavaSpaces [10] for instance, after a \text{write} primitive a message can be notified to all the coordinated entities that was registered through a \text{notify}.

We model a medium as an intrinsically reactive abstraction, in that changes in the medium state and delivery of output events occur only at the reception of some input event. However, several known coordination models such as JavaSpaces and T Spaces [16] promote the idea of medium proactiveness [14], that is, of media including inner activities able both to alter the medium state and send output events independently from the reception of input events. One of the most notable examples of inner activity is JavaSpaces tuple expiration, where internal events representing time-passing are conceptually raised which can cause some expiring tuple (specifying a \text{lease time} [10]) to be dropped from the dataspace. In principle, it would have been possible to tackle the issue of proactiveness by extending the characterisation of a medium interactive behaviour. For instance, in the transition system describing the medium behaviour, “proactive actions” can be added that represent internal events causing a state change and output events to be sent — analogously to the approach in [14]. Despite this technique would be interesting and deserves further studies, in this paper we stick to the more natural and simple idea of reactive medium, showing how mechanisms related to proactiveness can be anyway captured by our notion of expressiveness. For instance, in Section C time-passing is modelled by means of an external coordinated entity sending input events representing time tics — namely, supposing to move a medium inner activity outside the medium itself.

III. The Expressiveness Issue

To the best of our knowledge, two main approaches have been developed that define a notion of expressiveness for coordination models, which are both based on the framework of Turing Machine (TM) formalism. They basically encode the coordinated system evolution (or at least, a subpart of it) into a TM-like program transformation.

The traditional framework for this kind of approach is that of modular embedding [17], where the expressiveness of two concurrent languages is compared by trying to define a mapping between programs written in one language into programs written in the other, respecting given observation criteria — such as compositionality, non-deterministic choices invariance, and termination invariance. Applications of this framework to coordination languages have been studied by several works of Busi, Gorrieri, and Zavattaro [3], [18], [19], and by Brogi and Jacquet [4].

For instance, in [3], Busi et al define the expressiveness of Linda in terms of which computations can be carried on by a coordinated system composed by a tuple-space and a number of coordinated entities that feature (i) parallel composition, (ii) Linda primitives as synchronous communication acts, and (iii) recursion. In this framework, Linda has been proved to be Turing-equivalent. This characterisation of expressiveness is basically meant to emphasise that the existing primitives of Linda are expressive enough to make a coordinated system as a whole realising any computation, provided that a portion of the computation burden is charged upon coordinated entities — e.g., the ability of repeatedly executing a Linda operation. In [18], [19], the expressiveness of other coordination mechanisms – such as notification and \text{copy-collect} primitive – is also studied, by providing several encodings of various languages including such mechanisms. Brogi and Jacquet [4] extended this general approach so as to capture the expressiveness of multiset-rewriting models such as Gamma [20], and models based on communication transactions such as Shared Prolog [21].

A different notion of expressiveness is adopted in [5], which considers the Turing equivalence of the computations occurring within the coordination medium. So, instead of focussing on the computations executed by the whole coordinated system adopting a given coordination model as in [3], in [5] the focus is put on the computations executed within the medium alone, as defined by the coordination model. This characterisation makes it possible to measure the ability of a coordination model to support distribution of the coordination burden between coordinated entities and coordination medium [9]. An example of full-expressiveness in this acceptance is that of ReSpeCT tuple centres [8], a coordination model extending standard tuple spaces. In tuple centres, depending on a behaviour specification, input and output events can be intercepted
which trigger internal computations called reactions. The execution of reactions generally produces changes on the dataspace, so that the overall medium behaviour is altered possibly implementing high-level coordination services [9]. In [5], Denti et al. proved that the ReSpecT logic language is expressive enough to model any computation occurring in a tuple centre – and so, to embody any computable coordination policy within the medium. In particular, for any computable function $f$ on multisets of tuples, there exists a ReSpecT program for a tuple centre so that when a generic input event $\text{out}(\text{Start})$ reaches the tuple centre, its dataspace moves from $s$ to $f(s)$.

However, this notion of expressiveness does not take into account event-based abilities of the medium, e.g. ReSpecT is fully expressive according to [5] but is unable to support JavaSpaces-like notification – at least, unless it is properly extended as studied in [22]. In general, defining the notion of expressiveness in terms of TM-like computations seems to be neither the only possible chance nor probably the most reasonable one for coordination models. On the one hand, coordination concerns managing interaction in multi-component systems, on the other hand, in [6] Wegner defines a concept of interaction-based computing machine called Sequential Interaction Machine (SIM), which is shown to be intrinsically more powerful than TM as a tool for representing interactive behaviours [7]. So, in this paper we define a notion of expressiveness for coordination models by mapping coordination media upon SIMs, instead of TMs as in [5], by taking as a foundation Wegner’s idea of computation as a sequence of interactions.

A SIM is a machine that at each computation step accepts an input value and replies one output value, keeping track of previous inputs. The expressiveness of a SIM is characterised by all the interaction histories it computes, which are sequences of couples $\langle i_k, o_k \rangle$, where $i_k$ is the $k^{th}$ input value provided and $o_k$ the corresponding output value returned. As argued in [7], a SIM behaviour can be simulated by a TM, that is, by a computable function from (finite) sequences of input values to output values, associating to an history of inputs $i_0; i_1; \ldots; i_n$ the output $o_n$ to be produced at time $n$.

Accordingly, given a coordination medium expressed in terms of the transition system $(C, \rightarrow_C, Ie \times Oe)$ and an initial state $c_0 \in C$, we characterise its expressiveness as the set of the interaction histories $\langle i_{e_0}, o_{e_0} \rangle; \langle i_{e_1}, o_{e_1} \rangle; \ldots; \langle i_{e_n}, o_{e_n} \rangle$ it can exhibit starting from $c_0$, that is, those obtained by transition sequences of the kind:

$$c_0 \xleftarrow{i_{e_0}} c_1 \xleftarrow{i_{e_1}} c_2 \ldots \xleftarrow{i_{e_n}} c_n \xrightarrow{o_{e_n}} c_{n+1}$$

Then, analogously to the treatment of ReSpecT tuple centres in [5], a definition of full-expressiveness can be given on a generic framework (or formalism) for denoting coordination media. We consider a coordination framework $(C, \rightarrow_{C}, Ie \times Oe)$ whose behaviour is parametric on a specification $\sigma$, and an initial state $c_0 \in C$. This framework is said to be fully expressive if for any computable function $f \in Ie \mapsto Oe$ from sequences of inputs events to multisets of outputs events, it exists an actual specification $\sigma(f)$ so that, for any input sequence $i_{e_0}; i_{e_1}; \ldots; i_{e_n}$, the following evolution is admissible:

$$c_0 \xleftarrow{i_{e_0}} f(\sigma_{e_0}) c_1 \xleftarrow{i_{e_1}} f(\sigma_{e_1}) c_2 \ldots \xleftarrow{i_{e_n}} f(\sigma_{e_n}) c_{n+1}$$

that is, the medium obtained through specification $\sigma(f)$ can exhibit any history that can be computed by $f$. As a result, a coordination framework satisfying this property is able to model any event-based coordination service that adheres to the characterisation introduced in Section II.

IV. THE COORDINATION FRAMEWORK $\sigma$RST

In this section we define the $\sigma$RST formalism for specifying coordination media with the goal of providing for a very simple, yet fully expressive framework for the specification of event-based coordination services.

This is inspired to the ReSpecT logic language for programming tuple centres [8], borrowing a similar mechanism of reactions execution. A ReSpecT program is given in terms of a set of reaction specifications, which are meant to alter the behaviour of a standard Linda-like tuple space. When either $(i)$ an input event comes in the tuple centre, $(ii)$ an output event is sent out, or $(iii)$ an internal event occurs in the centre (due to the affection of the dataspace) a number of specification tuples may match that trigger reactions. These are executed in a transactional way (in the sense that any failure would result in rolling back the overall execution), each time producing a change in the dataspace possibly causing new reactions to be triggered. As an overall effect of this management of reactions, any computation can be performed when an input event is received by the medium.

The main difference between $\sigma$RST and ReSpecT is in the mechanism of reaction triggering. Instead of intercepting external and internal events as in ReSpecT, in $\sigma$RST reactions are fired by means of a newly defined primitive, thus becoming a first-class concept of the language. Then, $\sigma$RST provides for a primitive to explicitly send an output event to a given coordinated entity, which is necessary in order to reach the desired expressiveness in the definition of event-based coordination mechanisms [23].

In general, the philosophy of the two models is different: while ReSpecT reactions are meant to modify and adapt the default behaviour of a Linda tuple space, $\sigma$RST has no underlying model, so that his behaviour is completely defined by its reaction specification.

In a $\sigma$RST medium, both sets of values $V$ and of coordinated entities identifiers $Id$ are represented in terms of the set of logic tuples $T$, ranged over by variable $t$. Logic tuples are atomic formulae as defined in standard Prolog-like logic programming languages, built over a set of variables $V$, function symbols $F$, predicate symbols $P$, and terms $T$. As in Prolog, while variables start with an uppercase letter, functions, predicates, and terms start with a lowercase letter. A variable substitution $\theta$ in $\Theta = V \mapsto T$ can be applied to a logic tuple $t$ by means of notation $t\theta$. We
\[
\begin{align*}
& \langle t, \tilde{t}, t \tilde{t}, I_{\sigma}, \perp_{Oe}, \perp \rangle \in \epsilon \\
& \langle t, t \tilde{t}, t \tilde{t}, \perp_{Oe}, \perp \rangle \in \epsilon \quad \text{if} \quad [t/t'] \not\perp \\
& \langle t, t \tilde{t}, t \tilde{t}, \perp_{Oe}, \perp \rangle \in \epsilon \quad \text{if} \quad [t/t'] \not\perp \\
& \langle t, t \tilde{t}, t \tilde{t}, \perp_{Oe}, \perp \rangle \in \epsilon \quad \text{if} \quad \exists \tilde{t}' \in T : [t/t'] \not\perp \\
& \langle o(t), T \tilde{t}, I_{\sigma}, \perp_{Oe}, \perp \rangle \in \epsilon \\
& \langle t \mapsto t \tilde{t}, T \tilde{t}, I_{\sigma}, [t/t'] \tilde{t}, \perp \rangle \in \epsilon \\
\end{align*}
\]

denote by \([t/t']\) the most general variable substitution so that \(t[t/t'] = t'[t/t']\); when such a variable substitution is not defined we simply write \([t/t'] = \perp\). Symbol \(I_{\sigma}\) denotes the identity substitution.

A \(\sigma\text{RST}\) medium keeps a dataspace containing a multiset of tuples \(\tilde{T} \in T\), similarly to standard tuple-based coordination models. Then, it is also equipped by a specification \(\sigma \in T \times \tilde{T}\), specifying a (multi)set of reaction declarations. Each declaration \((t, r)\) assigns to a logic tuple \(t\) called reaction goal a reaction body \(r \in R = \delta\), that is, a sequence of operations \(o \in O\) to be executed. A triggering function \(\text{trig}_\sigma(t)\) is automatically defined that associates to a tuple \(t\) the multiset of reactions it triggers, defined as \(\text{trig}_\sigma(t) = \{\delta(t/t') : [t', \delta] \in \sigma, [t/t'] \not\perp\}\). In particular, in this definition, notation \(\delta(t/t')\) means that variable substitution \([t/t']\) has to be applied to any logic tuple specified in any operation \(o\) of the sequence. Operations are of the kind:

\[
o ::= \downarrow t \mid \uparrow t \mid \downarrow t \mid \neg t \mid o \mapsto t \mapsto t'
\]

Informally, \(\downarrow t\) puts \(t\) in the medium dataspace, \(\uparrow t\) removes a tuple matching \(t\) from the dataspace, \(\downarrow t\) reads a tuple matching \(t\) from the dataspace, \(\neg t\) checks for the absence of \(t\) in the dataspace, \(o\) triggers reactions associated to goal \(t\) by current specification \(\sigma\), and \(t \mapsto t'\) sends a message to entity \(t'\) carrying value \(t\).

Formally, the semantics of operations is specified by a relation \(\epsilon \subseteq O \times T \times T \times \Theta \times Oe \times \delta \subseteq R\) called goal execution relation. The occurrence of \((o, T \tilde{t}, \theta, Oe \tilde{t}, \perp)\) in \(\epsilon\) means that the execution of operation \(o\) on the dataspace \(T\) causes (i) the dataspace to move to \(\tilde{T}\), (ii) the variable substitution \(\theta\) to be propagated in next operations, (iii) output message \(Oe \tilde{t}\) to be sent out (unless it be \(\perp_{Oe}\)), and (iv) the reactions \(\tilde{r}\) to be triggered. Rules defining the semantics of operations are shown in Figure 1.

The inner dynamics of a \(\sigma\text{RST}\) medium is defined by a labelled transition system \((S, \rightarrow_S, L)\), where \(\rightarrow_S \subseteq S \times L \times S\). The state of the medium is composed by four parts as follows:

\[
S = T \times \tilde{T} \times Oe \times (T \times \tilde{T} \times Oe \times \delta) \perp
\]

The first three parts orderly represent the current dataspace content, the set of triggered reactions waiting to be executed, and the set of pending output events waiting to be sent out. Like ReSpecT, \(\sigma\text{RST}\) supports a transactional semantics for the execution of reactions, so that its effects on the medium dynamics are applied only in the case of successful termination, otherwise they are rolled back. So, each time a new reaction body \(r\) is executed, the fourth component stores transient information about the effect of current transaction, namely: (i) a copy of the dataspace on which modifications are actually made, (ii) the reactions triggered by \(r\), (iii) the output events produced by \(r\), and (iv) the remaining body to be executed. If a transaction execution fails the fourth component is simply ignored, otherwise its content is applied to the persistent part made of the former three components of \(S\). Labels of the transition system are of six kinds:

\[
L ::= \text{input} | \text{fire} | \text{exec} | \text{fail} | \text{commit} | \text{output}(Oe)
\]

and the corresponding rules are shown in Figure 2. When no reactions nor output events are pending, and no reactions are currently in execution, the medium may accept an input event \(\text{input}(t_{id}, t_{oe})\), which triggers some reactions by the triggering function \(\text{trig}\). Then, one triggered reaction is selected for evaluation in a non-deterministic way \(\text{fire}_{\rightarrow_S}\), causing its operations to be executed sequentially. For each operation, if a corresponding entry exists in the goal execution relation \(\epsilon\), then the operation is successfully executed \(\text{exec}_{\rightarrow_S}\), causing \((i)\) new reactions \(\tilde{r}\) to be triggered, \((ii)\) the dataspace to move to \(\tilde{T}\), \((iii)\) the output events \(Oe \tilde{t}\) to be produced, and \((iv)\) the variable substitution \(\theta\) to be propagated in the following operations. Otherwise, if no corresponding entries occur in \(\epsilon\), the computation of the current reaction fails \(\text{commit}_{\rightarrow_S}\), and its effects on dataspace, triggered reactions, and output events are discarded. When no operations remain to be executed the reaction evaluation commits \(\text{commit}_{\rightarrow_S}\), and its effects on the dataspace are made persistent. Finally, when no reactions are still pending \(\text{output}(Oe)_{\rightarrow_S}\), the generated output events are sent out. This operational semantics can be packed into the characterisation defined in Section II by considering \(S\) as the medium state \(C\), and through the straightforward rule
pace, and delivery of a multiset of output events – as which models an atomic interaction of the medium – including reception of an input event, change of the dataspace, and delivery of a multis set of output events – as (i) one internal input transition, (ii) a sequence of fire, exec, fail, and commit transitions, and finally an (iii) output transition.

V. Encoding Coordination Primitives and Mechanisms

In order to help understanding the behaviour of σRST reaction mechanism, and showing the expressiveness of the formalism as a tool for denoting coordination services, in this section we provide the medium specification corresponding to some well-known features of existing coordination models.

A. Linda primitives

Linda synchronisation primitives are described by providing the specification of a Linda medium, e.g. similarly to the model described in [14].

Input events are couples \([id, \text{linda}(op, t)]\), where \(id\) represents the sending entity, \(op\) is a Linda primitive (in, inp, rd, rdp, or out), and \(t\) is the tuple (or tuple template) requested/inserted. Output events are couples \([id, t]\), specifying the receiving coordinated entity (\(id\)) and the tuple notified (\(t\)), which is a reply for a previously made query (in, inp, rd, or rdp). The complete σRST specification of Linda primitives is shown in Figure 3.

When an input event comes in, reaction \(ev\) is triggered that evaluates it. In particular, each query is handled by two reaction declarations, dealing with the case where the arrived query matches an existing tuple or not, respectively. Evaluation of primitives inp and rdp looks for a tuple \(\text{tup}(t)\) occurring in the space that matches their template, and replies a positive or negative result to the requester. Queries in and rd are handled in a similar way, but if they are not satisfied a tuple of the kind \(\text{qry}(op, t, id)\) is stored in the dataspace, representing a sort of pending request waiting to be evaluated.

When an out\((t)\) operation is performed, a tuple \(\text{tup}(t)\) is written in the dataspace, and by triggering reaction serve the matching queries rd and in are served. This is realised in two steps. First of all, all the matching rd are retrieved from the dataspace, and evaluated by means of reaction ev, reiterating the process by continuously triggering serve. When no more matching rd occurs, reaction serve looks for one matching in and evaluates it. This evaluation policy, in particular, guarantees to serve only one destructive query (in) and all non-destructive queries (rd), following the usual “one-in/many-read” strategy [14].

\[
\begin{align*}
&\frac{c_0 \xrightarrow{\text{input}(ic)} S \ c_1 \quad c_0 \xrightarrow{\text{output}(\pi)} S \ c}{0 < i < n, l \in \{\text{fire, exec, fail, commit}\}}
\end{align*}
\]

which models an atomic interaction of the medium – including reception of an input event, change of the dataspace, and delivery of a multis set of output events – as (i) one internal input transition, (ii) a sequence of fire, exec, fail, and commit transitions, and finally an (iii) output transition.

\[
\begin{align*}
&\text{input}(\text{Id, Linda}(X, T)), \quad \text{oev}(X, T, \text{Id}) \\
&\text{ev}(\text{in}, T, \text{Id}), \quad \uparrow \text{tup}(T) ; T \Rightarrow \text{Id} \\
&\text{ev}(\text{in}, T, \text{Id}), \quad \uparrow \text{tup}(T) ; \text{qry}(\text{in}, T, \text{Id}) \\
&\text{ev}(\text{inp}, T, \text{Id}), \quad \uparrow \text{tup}(T) ; T \Rightarrow \text{Id} \\
&\text{ev}(\text{inp}, T, \text{Id}), \quad \uparrow \text{tup}(T) ; \text{null \Rightarrow Id} \\
&\text{ev}(\text{rd}, T, \text{Id}), \quad \uparrow \text{tup}(T) ; T \Rightarrow \text{Id} \\
&\text{ev}(\text{rdp}, T, \text{Id}), \quad \uparrow \text{tup}(T) ; T \Rightarrow \text{Id} \\
&\text{ev}(\text{rdp}, T, \text{Id}), \quad \uparrow \text{tup}(T) ; \text{null \Rightarrow Id} \\
&\text{ev}(\text{out}, T, \text{Id}), \quad \uparrow \text{tup}(T) ; \text{observe}(T) \\
&\text{serve}(T), \quad \uparrow \text{qry}(\text{rd}, T, \text{Id}) ; \text{oev}(\text{rd}, T, \text{Id}) ; \text{observe}(T) \\
&\text{serve}(T), \quad \uparrow \text{qry}(\text{in}, T, \text{Id}) ; \text{oev}(\text{in}, T, \text{Id})
\end{align*}
\]

Fig. 3. Specification of Linda primitives

\[
\begin{align*}
&\text{input}(\text{Id, reg}(T, \text{Id}0s)), \quad \text{qry}(\text{reg}, T, \text{Id}0s) \\
&\text{input}(\text{Id, ntf}(T)), \quad \text{qy}(\text{ntf}, T) ; \text{opre} \\
&\text{pre}, \quad \uparrow \text{qry}(\text{reg}, T, \text{Id}) ; \text{msg}(T, \text{Id}) ; \text{copy}(T, \text{Id}) ; \text{opre} \\
&\text{pre}, \quad \uparrow \text{qry}(\text{reg}, T, \text{Id}) ; \text{send} \\
&\text{send}, \quad \uparrow \text{qry}(\text{ntf}, T) ; \text{msg}(T, \text{Id}) ; T \Rightarrow \text{Id} ; \text{send} \\
&\text{send}, \quad \uparrow \text{qry}(\text{ntf}, T) ; \text{msg}(T, \text{Id}) ; \text{rest} \\
&\text{rest}, \quad \text{msg}(T, \text{Id}) ; \text{copy}(T, \text{Id}) \\
&\text{rest}, \quad \text{msg}(T, \text{Id}) ; \text{copy}(T, \text{Id}) ; \text{qry}(\text{reg}, T, \text{Id}) ; \text{rest} \\
&\text{rest}, \quad \text{msg}(T, \text{Id}) ; \text{copy}(T, \text{Id}) ; \text{qry}(\text{ntf}, T)
\end{align*}
\]

Fig. 4. Specification of the notification mechanism

B. The notification mechanism

As an example of an event-oriented coordination service, we define a simple coordination mechanism implementing basic notification, generalising the one provided by JavaSpaces or IWIM channels. A coordinated entity may register an observer (i.e., another entity or itself) for being notified when a given input event reaches the medium that matches a specified template.

For simplicity, we describe a medium sticking to two operations only: \text{reg}(t, id) and \text{ntf}(t); the former for registering the interest of \(id\) in receiving notification of tuples matching \(t\), the latter for actually notifying tuple \(t\). The corresponding specification is shown in Figure 4.

Information on each registration \text{reg}(t, id) is stored in the dataspace when its input event arrives. Then, when a notification \text{ntf}(t) occurs, it is temporarily stored in the dataspace, and each registration tuple \text{reg}(t, id) is removed and updated by two tuples \text{msg}(t, id) and \text{copy}(t, id), by means of reaction \text{pre}. Then, reaction \text{send} is automatically triggered that uses tuples \text{msg}(t, id) to look for matching registrations and to send the corresponding output events. Each time, the evaluated tuple is dropped from the dataspace so as to keep track of already-sent notifications.

When no more matching \text{msg}(t, id) occur, reaction \text{rest} is triggered that drops unmatched \text{msg}(t, id) and translates any \text{copy}(t, id) into a registration tuple \text{reg}(t, id), so as to restore the original state of registrations.

C. Expiring Data

Our model of coordination media does not take into account proactiveness, that is, a medium ability to spontaneously change its state and send output events. However, we can represent aspects related to proactiveness any-
way, by mapping the medium inner activity upon a sort of special purpose coordinated entity interacting with the medium. From the point of view of the characterisation given in Section II, this means to expose internal events as observable inputs in the transition system similarly to input events, which is an approach for dealing with proactiveness already exploited e.g. in [24].

In this section we give a flavour of this modelling technique by representing a simple example of data expiration in shared dataspaces, modelling e.g. lease management of JavaSpaces. We assume that the medium keeps track of its internal time, represented as a natural number. This time may increase by one unity when a coordinated entity with special identifier null sends the event tic, representing the inner activity raising an internal event. Also, we suppose that a new kind of input event is allowed to the medium inner activity raising an internal event. Also, we suppose that a new kind of input event is allowed to the medium, and (ii) – for each function \( f \) from \( \overline{I}e \) to \( \overline{O}e \) – a specification \( \sigma(f) \) such that the medium exhibits any interaction history computed by \( f \). First of all, for any such function \( f \) we denote by \( fr \in (T \times T) \rightarrow (T \times T) \) the computable function defined as:

\[
f_T(t_{seq}(\hat{\circ}e), \text{null}) := (t_{seq}(\hat{\circ}e), t_{seq}(\hat{\circ}e)) \iff f(\hat{\circ}e) = T
\]

where \( \hat{\circ}e \) is any sequence built using all the elements of \( \overline{O}e \). Similarly to [3] and [5], we rely on the formalism of Stepherdson and Sturgis’ register machine over sequences [25], which is proven to be Turing-equivalent. In particular, any function \( f_T \) can be computed by a machine with two registers in and out containing logic tuples of the kind \( t_{seq}(\hat{\circ}e) \) and \( t_{seq}(\hat{\circ}e) \), and with a program made of a sequence of instructions of the three kinds:

\[
L: \text{add}(r,t) > Ln \quad L: \text{rem}(r) > Ln \quad L: \text{jmp}(r,v,Lo) > Ln
\]

Since this instruction set provides jumps, each instruction is equipped by a unique label \( L \). Instruction \( \text{add}(r,t) > Ln \) adds tuple \( t \) to the sequence currently stored in register \( r \) (where \( r \) is either term \( \text{in} \) or \( \text{out} \)), and then continues to instruction labelled by \( Ln \) (logically, the next one). Instruction \( \text{rem}(r) > Ln \) removes the first element from the sequence in register \( r \), and then continues to \( Ln \). Finally, instruction \( \text{jmp}(r,v,Lo) > Ln \) continues to \( Ln \) if the first element of the sequence in \( r \) is \( v \), otherwise it jumps to \( Lo \).

We suppose that in the register machine each computation starts at a given initial label \( \text{istart} \) and stops at a given ending label \( \text{istop} \). Input and output values of the computation, then, are the content of registers before and after execution, respectively.

Let any label being represented by a term, and let \( \text{reg}/2 \) be a binary logic function taking a register type \( r \) (\( \text{in} \) or \( \text{out} \)) and its current value \( v \). The initial state \( c_0 \) of the \( \sigma \text{RST} \) medium is \( \text{reg(in,null)|reg(out,null),,},, \), storing the void sequence in each register.

For any \( f_T \), the corresponding register machine can be simulated by a \( \sigma \text{RST} \) medium with the specification shown in Figure 6. The upside part of the specification describes the behaviour of a generic register machine, supposing that reactions go and start – which have to be specified – define the \( \sigma \text{RST} \) program implementing function \( f_T \). The former line intercepts any input, adds its content to the input register – event sender and event content, respectively – and fires execution at the initial label \( \text{istart} \). Execution is handled through five cases. An \( \text{add} \) instruction can be retrieved which \((i)\) adds a value to a register. A \( \text{rem} \) instruction can be retrieved which either \((ii)\) drops an
Fig. 6. Specification implementing a register machine

element to a register if it has any, or otherwise (iii) does nothing. A jmp instruction can be retrieved which (in) continues to labels ln or (v) jumps to Lo, depending on the first element stored into the register. Finally, if the ending label is reached, computation terminates and output register is unloaded producing the intended multiset of output events, i.e. the one yielded by function f.

The downside part of the specification, on the other hand, contains the specification of reactions go and start defining the σRST program implementing fT. A reaction start should be defined that simply triggers go on the starting label. One reaction declaration for goal go has to be specified for any instruction of the program, causing either triggering of reactions add, rem, or jmp, which are specified in the upside part. For the ending label lstop, a reaction declaration for go should finally be specified that triggers out, causing the delivery of output messages.

VII. CONCLUSIONS

Exploiting coordination models in the context of event-based systems raises new issues both in theory and in practice. In this paper we first define a new notion of expressiveness that we believe could be the basis for reasoning on a small yet relevant set of cases.

Indeed, more application cases would be needed in order to deepen its applicability as a tool for specifying coordination services in event-based scenarios. This very issue is likely to be the subject of our future research. Also, it would be interesting to exploit σRST as a formal framework to compare the expressiveness of different existing coordination models and their primitives, so as to better emphasise the relationship among coordination, event-based systems, and Interactive Computing [6].

REFERENCES


