The Event Coordination Notation: Behaviour Modelling Beyond Mickey Mouse

Jesper Jepsen[1] and Ekkart Kindler[2](✉)

[1] Alumnus of DTU Compute, Kgs. Lyngby, Denmark
jepsen.jesper@gmail.com

[2] DTU Compute, Technical University of Denmark, Kgs. Lyngby, Denmark
ekki@dtu.dk

Abstract. The *Event Coordination Notation* (*ECNO*) allows modelling the desired behaviour of a software system on top of any object-oriented software. Together with existing technologies from *Model-based Software Engineering* (*MBSE*) for automatically generating the software for the structural parts, ECNO allows generating fully functional software from a combination of class diagrams and ECNO models. What is more, software generated from *ECNO* models, integrates with existing software and software generated by other technologies.

ECNO started out from some challenges in behaviour modelling and some requirements on behaviour modelling approaches, which we pointed out in a paper presented at the second BMFA workshop [1]; the integration with pre-existing software was but one of these requirements.

Different ideas and concepts of ECNO have been presented before – mostly with neat and small examples, which exhibit one special aspect of ECNO or another; and it would be fair to call them "Mickey Mouse examples".

In this paper, we give a concise overview of the motivation, ideas, and concepts of ECNO. More importantly, we discuss a larger system, which was completely generated from the underlying models: a workflow management system. This way, we demonstrate that ECNO can be used for modelling software beyond the typical Mickey Mouse examples. This example demonstrates that the essence of workflow management – including its behaviour – can be captured in ECNO: in a sense, it is a domain model of workflow management, from which a fully functioning workflow engine can be generated.

Keywords: Workflow engine · Meta-modelling · Behaviour modelling · Event Coordination · Code generation

1 Introduction

Long before the advent of *Model-based Software Engineering* (*MBSE*) and one of its main driving forces, the *Model-driven Architecture* (*MDA*) [2], there was an endeavor to better understand and distill the nature of communication and interaction in concurrent systems – with pioneering work of Petri [3], Hoare [4,5], Harel [6], and Milner [7] developing modelling notations for behaviour and identifying the fundamental concepts of communication and coordination, which are still valid to date. Today, there exist a plethora of modelling notations for modelling the behaviour of distributed, concurrent, or cooperating systems based on theses concepts.

With the advent of Model-based Software Engineering, models received even more attention – with the promise that a software system (or at least major parts of it) could be automatically generated from these models. Using technologies like the Eclipse Modeling Framework (EMF) [8] can save a lot of programming, making software development significantly faster and the resulting software more reliable. Most of the automatically generated code, however, concerns the structural parts of the software or standard functionality, but not the actual behaviour.

In view of the fact that notations for modelling behaviour have been out there for a quite a long time, it might appear a bit surprising that the use of behaviour models lags a bit behind in Model-based Software Engineering. There are different reasons for that. One reason is that the structural models, such as class diagrams, typically, lack a natural mechanism for "hooking" in behaviour on a higher level of abstraction. The only mechanism they provide for "hooking in" behaviour is method invocation – which is quite different from the communication mechanisms proposed by Hoare [4] and Milner [7]. Other reasons why behaviour modelling lags behind were pointed out in our contribution to the second BMFA workshop [1]: e.g. lack of mechanisms for integration with existing software which would allow for a smooth transition from programming software to modelling it.

Starting from these issues and challenges, we gradually developed a notation for modelling behaviour, which could overcome these problems: the *Event Coordination Notation* (*ECNO*) [9]. *ECNO* is a modelling notation that allows modelling the behaviour of a system on top of structural models (such as class diagrams) on a high level of abstraction based on some of the basic communication mechanisms proposed by Hoare and Milner. Still, the software can be generated from these models fully automatically. With the publication of the ECNO technical report [9], which covers the motivation, philosophy as well as all the details of ECNO's modelling concepts and notations and with the publication of the ECNO Tool, ECNO has reached a major milestone, which we report on in this paper.

The basic mechanism of ECNO for integrating behaviour models with structural models are *events*[1]; and events are a first class modelling concept in ECNO. The *life-cycle* of an object, basically, defines when an object can participate in which kind of events. We call this the *local behaviour* of the object; it, roughly, corresponds to what Harel and Marelly [10] call *intra-object behaviour*.

[1] In Milner's terminology, our events would be called *actions*, and in Hoare's terminology events would be channels or channel names.

ECNO provides different ways for defining the local behaviour of objects; the default way for modelling the life-cycle of objects in ECNO, however, is a simple form of Petri nets, which we call *ECNO nets*.

The more interesting part of ECNO, however, is the coordination of the behaviour of different objects that need to join in on the execution of events. ECNO provides *coordination diagrams* for defining which partners need to participate in an event, in a given situation. We call this the *global behavior*, whereas Harel and Marelly [10] would call it inter-object behaviour. The mechanisms used in coordination diagrams are similar to the communication mechanisms of Hoare and Milner, but – as we will see later – coordination diagrams are more general in that many partners can be required, and more than one event might be jointly executed. Moreover, the partners that are required to participate in an event, can depend on the current situation and the underlying object structure. We call the particular combination of objects and events that meet the requirements of the coordination diagram and are executed together an *interaction*.

Altogether, ECNO allows modelling the desired behaviour of software systems on a high level of abstraction (on top of structural software models), from which fully functioning software can be generated fully automatically. To achieve this, ECNO uses a carefully balanced and adjusted set of concepts and, on the technical sided, was designed so that it integrates with different MBSE and object-oriented technologies.

In this paper, we give an overview of the concepts of ECNO and provide some motivation and philosophical background. We informally introduce the main idea and concepts of ECNO by a simple, but complete, example in Sect. 2. In Sect. 3, we give a more systematic account of ECNO's basic concepts and introduce some of ECNO's more advanced concepts.

Up to now, our published papers on ECNO used neat and small examples particularly tuned to explain some concepts of ECNO. And it is fair to call them "Mickey Mouse examples". In order to demonstrate that ECNO reaches "beyond Mickey Mouse", this paper shows that ECNO can be used for modelling the concepts of workflow management, including their behaviour, from which a workflow engine can be generated fully automatically. This model, is discussed in Sect. 4; it was developed in a 5-month master's project [11]. Even though, we consider this a "beyond Mickey Mouse example", we do not call it case study; the reason is that ECNO actually evolved from an ad-hoc notation that we used to capture the essential concepts of business process models, which we called *AMFIBIA* for "*A Meta-Model for Integrating Business Process Aspects*" [12]. Though ECNO has significant extensions and a much more thought through and balanced combination of concepts as compared to AMFIBIA, a real case study for evaluating ECNO would need to come from a domain different from workflow management.

Most concepts of ECNO are actually not new, considered in isolation. The novelty of ECNO is the combination of concepts and its integration with MBSE technologies. In Sect. 5, we discuss the contributions of ECNO and relate them to existing work and concepts.

Naturally, this paper cannot cover all the details of ECNO. We refer to the ECNO technical report [9] and some earlier publications [13–16] for more details on ECNO – in particular concerning some more technical aspects and the ECNO Tool itself. The ECNO Tool and its documentation are available from the ECNO Home page: http://www2.compute.dtu.dk/~ekki/projects/ECNO/index.shtml.

2 ECNO: An Example

In this section, we give a brief and informal overview of the concepts and notations of ECNO by formalizing the behaviour of *Petri nets* by means of ECNO [16]; to be precise, we formalize the semantics of *P/T-systems* [17]. One reason for choosing Petri nets as our example here is that the modelled workflow engine will use *workflow nets* [18,19] for modelling the control aspect of business processes. Since workflow nets are a restricted class of Petri nets, we can re-use the understanding of this Petri net semantics later on when discussing the model of the workflow engine in Sect. 4.2.

Another reason for choosing Petri nets as our example is that it is neat and concise and allows us discussing the most important concepts of ECNO. Even though the example is neat, it might twist your mind a bit. The reason is that on the one-hand side, we use ECNO models for defining the semantics of Petri nets, which means that Petri nets occur on the instance level; but, ECNO itself also uses another version of Petri nets called ECNO nets for modelling the behaviour, which means that Petri nets occur also on the modelling level. The Petri nets used on these two level should not be confused with each other.

2.1 Petri Nets

Figure 1 shows a simple example of a Petri net, which models the mutual exclusion of two processes by a semaphor: there are two agents or processes, which cyclically run through the phases *idle*, *pending* (*pend*), and *critical* (*crit*). As indicated by the name, the two processes should never be in their critical section at the same time. This is achieved by each agent acquiring the *semaphor* (*sem*) when entering the critical section. The semaphor is returned again when the agent exits the critical section. In a Petri net, the possible states are represented by *places* which are graphically shown as circles or ellipses. A black dot, called a *token*, on a place indicates that the agent currently is in this state. In P/T-systems, it is possible that there is more than one token on a place, but this situation does not occur in our example. Figure 1 shows that, initially, both processes are idle (represented by the tokens on places *idle1* and *idle2*) and that the semaphor is available (represented by the token on place *sem*). The distribution of tokens on the places of the Petri net represents the current state of the system; it is called the *marking* of the Petri net.

The possible state changes are defined by the *transitions* of the Petri net, which are graphically represented by squares. The *arcs* from a place to a transition indicate on which places there needs to be a token for the transition to

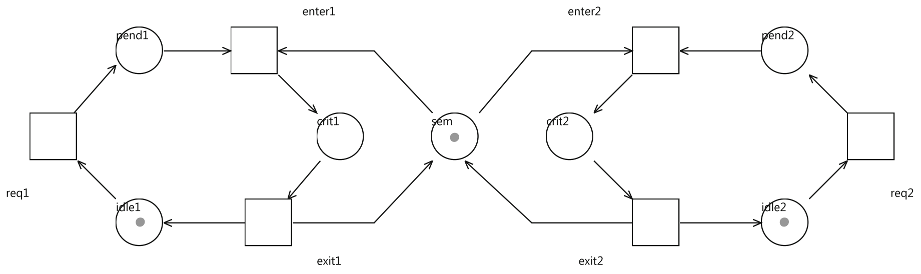


Fig. 1. A Petri net modelling mutual exclusion

be *enabled*. In the marking of the Petri net of Fig. 1, for example, the transition *req1* is enabled, since there is a token on the place *idle1*, which is the only place with an arc to transition *req1*. Likewise, transition *req2* is enabled since there is a token on *idle2*. All the other transitions are not enabled in this marking. An enabled transition can *fire*: if and when a transition fires, it *removes* one token from each place from which an arc is pointing to the transition; at the same time the transition *adds* one token to each place to which it has an arc pointing to.

2.2 Formalizing Petri Nets

Next, we formalize the syntax and semantics of Petri nets in a software engineering way by providing models [16]; actually, we formalize the *abstract syntax* of Petri nets only. The abstract syntax is defined by a class diagram[2], which represents the concepts of Petri nets and their relation among each other. The behavior is defined by a *coordination diagram* on top of the class diagram later.

Figure 2 shows the class diagram formalizing the concepts or the abstract syntax of Petri nets. We omit some constraints, though. The main concepts of Petri nets are Places and Transitions, and Arcs connecting them.

Next, we discuss how to define the behaviour of Petri nets (their semantics) by some ECNO models. In ECNO, the behaviour is modelled in two parts: the *local behaviour* or the life-cycle for each *element* (in ECNO, objects that have an explicit life-cycle are called elements); and the *global behaviour* which defines how to coordinate the local behaviours of the different elements with each other by so-called *coordination diagrams*.

We start with the discussion of ECNO coordination diagrams for modelling the behaviour of Petri nets, with some informal hints to the local behaviour, which we model later. Figure 3 shows the coordination diagram that defines the global behaviour of Petri nets. It shows the main elements of the class diagram from Fig. 2 again – with some additions. The main concept of ECNO that enable us to coordinate behaviour are *events* which can have different types (*event types*): for our Petri net example, the relevant event types are fire, add and

[2] Actually, it is an Ecore model, which is a kind of lightweight version of UML class diagrams supported by EMF [8].

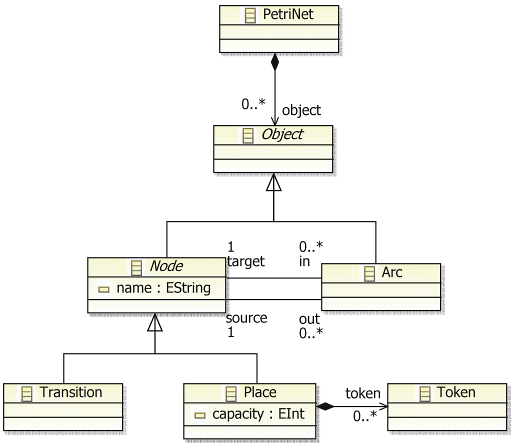


Fig. 2. Abstract syntax of Petri nets: Class diagram

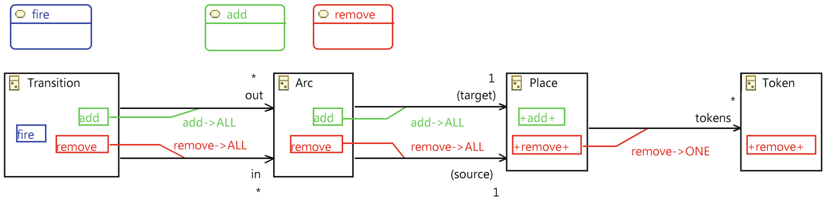


Fig. 3. Coordination diagram: Global behaviour of Petri nets

remove, which correspond to firing a transition, and the corresponding removal and addition of tokens from and to places. The event types are explicitly defined in the coordination diagram, shown as rounded rectangles. The rest of the coordination diagram defines how different elements coordinate their behaviour via events by annotating the underlying class diagram with *coordination annotations*, which we explain below.

Since the semantics of Petri nets is about firing transitions, we start explaining the coordination diagram at the transition. The *element type* Transition is associated with three event types: fire, add, and remove. Technically, this can be seen by the boxes inside the element types with the respective labels referring to the respective event types, which are called *coordination sets*. We will see later in Fig. 4 that the local behaviour of the element type Transition requires that three events of event types fire, add, and remove must be executed together for a Transition element. The *coordination annotations* attached to the references *out* and *in* respectively, require that all the arcs starting at the Transition (*out*) need to participate in an add event, and that all the arcs ending at the Transition (*in*) need to participate in a remove event whenever a Transition participates in such an event.

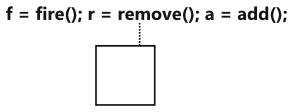


Fig. 4. Local behaviour of a Transition

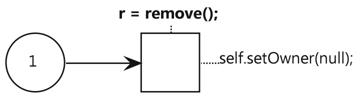


Fig. 5. Local behaviour of a Token

As we will see later, when defining the local behaviors of the different element types, an Arc can always participate in add and remove events. The relevance of the Arc is that the coordination annotation attached to the reference *target* requires that the Place to which the arc points to participates in the add event, too. Likewise, the coordination annotation attached to the reference *source* requires the Place at which the arc starts to participate in the remove event, too. This way, the coordination diagram guarantees that every place with an arc to a transition is involved in a remove event, and every place with an arc from a transition is involved in an add event – initially triggered by a transition participating in a fire event.

The local behaviour of a Place when it participates in an add event adds a new token to this place. The place does not require other associated elements to participate in the add event. Therefore, the coordination set for add is not attached to any coordination annotation. Note that the label of that coordination set is enclosed between two plus signs, which indicates a subtlety of ECNO, which we will discuss later in Sect. 3.5: *counting events*.

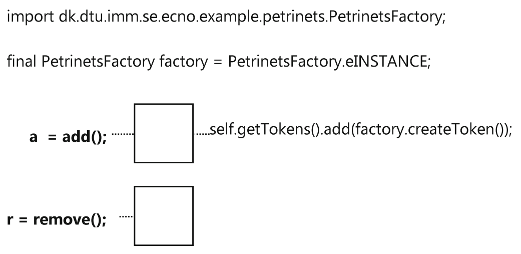
The local behaviour of a Place when participating in a remove event does nothing, but the coordination annotation requires that one of the Place's tokens participates in the remove event. This participating Token will then take care of removing itself from the place.

In ECNO, the local behaviour of an element defines when the element can participate in an event – and what effect that will have on the element. ECNO uses a simple form of Petri nets for that purpose again[3], which we call *ECNO nets*; the ECNO nets (model level) for the different element types of Petri nets (instance level) are shown in Figs. 4, 5, 6 and 7. Note that most of these ECNO nets are very degenerated Petri nets (transitions not connected to any places, which means that they are enabled anytime).

Figure 4 shows the local behaviour of the Transition. It shows that a transition element can join a fire event any time (from the transition's point of view), but it requires the remove and add events to be part of that interaction too – this way, in combination with the global behaviour and the local behaviour for the other element types, making sure that the respective tokens are ready for removal and also removing and adding the respective tokens when executed.

Figure 7 shows the local behaviour of the Arc. There are two transitions, which can be executed anytime, which means that the Arc can participate in the events add and remove anytime. Since the transitions are independent of

[3] Note that these Petri nets are used on the modelling level now.



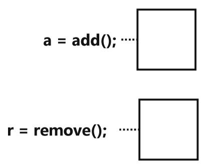
```
import dk.dtu.imm.se.ecno.example.petrinets.PetrinetsFactory;

final PetrinetsFactory factory = PetrinetsFactory.eINSTANCE;
```

```
a = add();
self.getTokens().add(factory.createToken());
```

```
r = remove();
```

Fig. 6. Local behaviour of a Place



```
a = add();
```

```
r = remove();
```

Fig. 7. Local behaviour of an Arc

each other, an arc can even participate in both events in parallel. Actually, the participation of an Arc in any of these events does not have any local effect on the Arc at all. The Arc is a mediator only which propagates the respective events from the transition to the respective places as defined by the coordination diagram in Fig. 3.

Figure 6 shows the local behaviour of the Place. Again, there are two transitions in this ECNO net, which are enabled all the time. The first transition is bound to the add event. Note that there is an *action*, a Java code snippet, attached to this transition. This action is executed when the add event is executed: it creates a new token and adds it to the place itself (`self`) by using the API which is automatically generated from the underlying class diagrams.

Figure 5 shows the local behaviour of the Token. This is the only ECNO net in our example where the firing of the transition is restricted. Actually the single token on the place makes sure that a token can be removed only once in its life-time – the very semantics of a token in Petri nets. The action (Java code snippet) attached to the transition actually removes the token from the place (its owner) when the Token participates in a remove event.

Together, the models for the local behaviour (Figs. 4, 5, 6 and 7) and for the global behaviour (Fig. 3) define the semantics of P/T-systems. Starting from a fire event on a Transition element, the transition will also be required to participate in an add and remove event, which then will require the connected Arcs, Places and Tokens to participate. This combination of elements and events is called an *interaction*. Figure 8 graphically shows an example of one interaction that is possible in that Petri net in the given marking. The interaction is shown as an octagon containing all instances of events; the dashed lines show which events are associated with which elements. Executing the interaction shown in Fig. 8 corresponds to firing transition t1 with the topmost token on place p1. Note, that there would be one other interaction possible in this situation (which is not shown here).

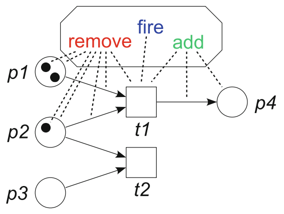


Fig. 8. Interaction: t1

3 ECNO: Concepts

In this section, we give a more systematic overview of the terms and concepts of ECNO. We start with the core concepts, which we have seen in the example already. In the end, we give an overview of ECNO's more advanced concepts in order to give a more complete picture of ECNO.

3.1 Object-Oriented Modelling

ECNO is based on object-oriented models, in particular, *class diagrams* and *object diagrams*. In order to present a consistent overall picture, we briefly rephrase the needed concepts from object-orientation. ECNO and its tool are independent from any specific object-oriented technology. As a default however, the ECNO Tool uses the Eclipse Modeling Framework (EMF) [8] as its underlying technology. Therefore, we use EMF's terminology instead of UML's [20].

Basically, ECNO assumes that there are *classes* and *references* between them along with their *multiplicity*. Later, we also use *inheritance* on classes. Classes, can have *attributes*, but attributes are not specifically exploited by ECNO. In some of our class diagrams, we use *compositions*, which are a special case of references. These are typically relevant when a bunch of objects is serialised to a file. Like attributes, compositions are not directly exploited in ECNO, however. Classes and references are defined in the scope of a *Package*.

We have seen an example of a class diagram or a package in Fig. 2, which actually is an EMF *Ecore diagram* already.

An *object diagram* is an instance of a class diagram, which shows a specific situation of a system. An instance of a class is an *object* and an instance of a reference is a *link* – or vice versa, the type of a link of an object is a reference of the object's class.

3.2 ECNO: Basic Concepts

Next, we discuss the basic concepts of ECNO, which we have seen in the example already.

Elements and Element Types. ECNO aims at modelling the behaviour of a system on top of a class diagram. To this end, ECNO extends the notion of objects and classes of object-orientation. Note that an ECNO model does not need to define behaviour for all classes of the underlying class diagram. Not all classes have ECNO behaviour, some might be used as data only. In order to make this difference explicit, we call objects that have behaviour in ECNO *elements*, and we call the classes for which ECNO defines behaviour *element types*.

Life-Cycle (Local Behaviour). In a sense, an ECNO element is an object with an explicitly defined *life-cycle*. The life-cycle of the element is defined by a model which defines the *local behaviour* for a specific element type. So, an ECNO element type consists of a class from object orientation and a local behaviour; and there are some more concepts, which we discuss later. In this paper, the local behaviour is defined by a special version of Petri nets, which we call *ECNO nets*. In our example, the element types were defined by the class diagram of Fig. 2 and the ECNO nets of Figs. 4, 5, 6 and 7, by using the same name for the classes and the ECNO nets.

Events and Event Types. The life-cycle of an element defines when (i.e. in which situations) the element could participate in some *event*, and how the state of the element changes, when the element participates in that *event*. To be precise, we distinguish between *event types*, and an instance or occurrence of an event at a specific time at runtime when an interaction is executed. The *event types* are defined in ECNO's coordination diagrams, where they are represented as rounded rectangles. The ECNO nets refer to these event types for defining the local behaviour. For simplicity, we call an instance of an event type just an *event* in the rest of this paper. The relation between an event and an event type is similar to the relation between objects and classes. Still the nature of events is fundamentally different from the nature of objects: An event is inherently volatile and – at least conceptually – has no duration; when the execution of the interaction is finished, all its events "evaporate"; only their effects – defined by the life-cycles of the elements participating in the events – stay.

Coordination and Interactions. An *interaction* is the joint participation of some elements in some events. Conceptually an interaction is executed instantaneously[4]. What constitutes a combination of elements and events that make up a legal interaction is defined by the local behaviour of the elements as well as the *coordination annotations*. As explained already, the local behaviour defines, whether an element could participate in an event at a given time; it can also require that some events need to be executed together (see Fig. 4). The coordination annotations define which combinations of elements and events are valid. Basically, each coordination annotation formulates a requirement of the following nature: if an element of some type participates in an event of some type, one or all elements to which there exists a link of a certain reference need to participate in that event too. We have seen in Fig. 3 that, for a Transition to participate in an add event, all Arcs which go out from this transition must participate in the add event too. Together, these requirements might require that many elements participate in a valid interaction – once one element participates in some event. In practice, the possible interactions will be computed starting from one element participating in an event of some type, until all the requirements of all coordination annotations of the involved elements are met. The interaction of

[4] In practise, executing an interaction takes time. We will see later that the instantaneous execution of interactions is mimicked by executing them transactionally in the sense of the ACID principle [21]; in particular, interactions are executed "atomically" and "in isolation".

our example shown in Fig. 8 was computed starting from the transition t1 and a fire event. Note that, in general, there might be more than one possible interaction for a given element and event. And it might happen, that there is no such combination at all – in which case the overall behaviour would not allow the element to participate in that event at that time.

The coordination annotations are defined in an ECNO *coordination diagram*, which is based on an underlying class diagram. Note that coordination diagrams are also used to define the event types. The defined event types can then be used in the coordination annotations as well as in the ECNO nets for the local behaviour.

Element State and Situations. As discussed above, the coordination annotations together with the local behaviour for the elements define which interactions are possible in any given situation. The local behaviour and, in particular, the *actions* (Java snippets attached to the transition of the ECNO net, cf. Fig. 6) define what each element would do and how its state would change when the interaction is actually executed. Note that the *state* of an element consists of two parts: the state of the underlying object (i.e. all its attributes and links) as well as the state of its life-cycle (the marking of the ECNO net in our case). A *situation* consists of the state of all objects (e.g. represented as an object diagram) plus the state of the life-cycle for each object.

Above, we have roughly sketched which interactions would be valid in a given situation. A formal definition for the fragment of ECNO that we have discussed so far, can be found in Chap. 4 of the ECNO technical report [9].

Controllers and GUI. One question, however, was not answered yet: when will a possible interactions be computed and executed? Actually, the ECNO models do not define that at all. The ECNO models specify which interactions could be executed (are valid) in a given situation – and the ECNO execution engine will make sure that only valid interactions are executed. It is left to some *controllers* on top of the ECNO engine to decide when valid interactions are computed and then scheduled for execution. Typically, the execution of interactions is triggered by the user by clicking on some button in some Graphical User Interface (GUI); and the ECNO Tool comes with some predefined controllers and a default GUI for that purpose. These controllers are automatically instantiated for new elements, when the ECNO engine becomes aware of them. These controllers can also be programmed manually and registered with the ECNO engine, which then can compute and execute interactions on elements as they see fit. To this end, ECNO comes with a programming framework for implementing own controllers and for configuring them for an ECNO application. For details, see Sect. 5.5 of the ECNO technical report [9].

In our simple example, some element types and some event types, are specifically marked as *GUI types*. From this information, the ECNO code generator generates a simple GUI with standard controllers, where the user interactively can trigger enabled interactions on the GUI elements.

3.3 ECNO: Event Synchronisation and Parameters

Next we discuss some more advanced concepts that concern events, in particular, the synchronization of different events and *event parameters*, a concept which did not occur in our example yet.

Life-Cycle: Choices. As discussed above, each element has a life-cycle or a local behaviour, which in our example are defined by ECNO nets. But, there could be other formalisms for the local behaviour of elements; actually, the local behaviour can also be programmed. Basically, the *local behaviour* associated with an element, defines in any given situation, in which *choices* the element could participate. In ECNO nets, the possible choices are defined by the transitions of the ECNO net; these choices define which events would be involved in the choice by the *event binding* associated with the respective transitions. Moreover the choice defines the element's state changes when the choice is taken (executed); in ECNO nets, that would be defined by the change of the marking of the ECNO net by firing the transition, as well as by the *action* associated with the transition, which could change some of the attributes and links of the underlying object, as defined by some Java code snippet. In ECNO nets, transitions can also have an additional *condition*, which can refer to the parameters of the events (see below) and the attributes of the object in order to define additional pre-conditions for firing the respective choice. The interfaces for the local behaviour of an element as well as for choices form the backbone in the ECNO framework and allow the ECNO engine to compute valid interactions and execute them independently from a specific modelling notation for the local behaviour.

Synchronizing Different Events. Typically, the transitions of an ECNO net are associated with exactly one event type. But, it is possible that an event binding for a transition refers to more than one event type (in the ECNO net of Fig. 4, the binding refers to three event types fire, remove, and add). In that case, the same element would be required to participate in two or more events at the same time within the same interaction. This way, it is possible that an element participating in one event requires the element to participate in some other events too, which is then also propagated to other elements as defined by the coordination annotations. Basically, this corresponds to the synchronization of two or more different events.

Event Parameters. In general, event types can have parameters, which are defined for each event type with a name and a data type. The local behaviour can assign values to these parameters in the event bindings, and the parameters of the involved events can be used in the condition and action associated with the event. In contrast to methods of classes, an event is not owned by an element. The relation of all participants to an event is completely symmetric: there is no "caller" or "callee" of an event; there are only participants in an event. In principle, any participant of an interaction can contribute a value to an event it is participating in. This could result in different elements contributing different values. The default behaviour of ECNO's event parameters is that an interaction is valid only, if all values contributed to the same parameter are actually the same (in the sense of Java's `equal()`). We call this kind of event parameter *exclusive* parameters. And the ECNO execution engine will make sure that valid interactions meet this condition.

In some cases, however, we would like to allow all participants that engage in an event to contribute a value; and the contributed values should not be required to be equal. We call this kind of event parameter a *collective* parameter. In that case, when accessed in a condition or action, the value of a parameter would return a collection of all the values contributed by the different partners. This can, for example, be used, to get hold of all the partners involved in the same event, by each partner assigning itself to this parameter.

The parameters are assigned to the events in the event bindings, by providing an expression for the respective parameter. This expression could refer to attributes of the element or the underlying object (`self`) and also refer to other event parameters. The parameters of an event `e` can be accessed by `e.parameter`, where `parameter` is the name of the parameter. In order to refer to an event, event bindings are represented as an assignment, assigning the bound event to a name: In the binding shown in Fig. 4, we could refer to the instance of the fire event by `f`, the variable the event is assigned to. When using an expression for assigning a parameter to an event, which refers to other parameters, there is one complication: There could be an assignment of a parameter that depends on an other parameter. In such cases, the ECNO engine makes sure that these assignments are done in the order respecting the dependencies – in case of cyclic dependencies, the interaction would be considered invalid. The value of an event parameter, can be accessed in event bindings, in conditions and in actions, by referring to the respective event and the name of the parameter as discussed above.

Actually, it is this completely symmetric way of dealing with contributing values to an event parameters, which helps us doing away with the invocation based way of coordinating behaviour. Parameters are not passed in a specific direction; they are just shared among different participants. It is perfectly possible that different parameters of the same event are contributed by different partners of an interaction.

3.4 ECNO: Inheritance

ECNO also has a concept of inheritance. Actually, there are different forms of inheritance in ECNO: There is inheritance on element types, which we call *behaviour inheritance*. And there is inheritance on event types, which even comes in two flavours: *specialisation* and *extension*.

Even though inheritance on event types is probably more interesting, we focus on behaviour inheritance in this paper. For details on inheritance on events, we refer to the ECNO technical report [9].

If the underlying classes of two element types have an inheritance relation in the object-oriented model, the corresponding element types in an ECNO model can also have an inheritance relation between them. In ECNO, however, one element type can inherit from at most one other element type, which gives rise to a linear inheritance hierarchy – multiple inheritance on element types is not supported by ECNO.

The behaviour of an element of an element type that inherits from an other element type (and indirectly from more), basically consists in running all the life-cycles of the element's type hierarchy in parallel and synchronizing them on the same events. This way, a sub-type restricts the behaviour of the life-cycle of its super types. In addition, a sub-type can introduce event types not known or not used by its super types; in that case, the sub-type will actually add new behaviour concerning the new event types since the super type will not synchronize on events it does not know. Typically, the top-level element type will define some overall life-cycle; and sub-types will just add some additional constraints on when the sub type can participate in the event, and what happens when the event is executed. Due to the synchronization of all the life-cycles of the element type hierarchy, it could easily happen that some events are blocked completely. Developing a methodology and modelling guidelines that avoid inadvertently blocking some events, is planned for the future.

Generally, we feel that synchronizing the life-cycles of the element type hierarchy provides a more faithful notion of inheritance since the sub-types cannot arbitrarily change the behaviour – the behaviour of the super types is still accounted for in the sub-types, which, for example, is not true in Java, where sub-classes can completely change the behaviour of a method by overriding it.

In some cases, however, sub-types might want to change the behaviour that was defined by the super type. To this end, ECNO also provides a mechanism to partially or completely override the local behaviour of super types. And in the actions, the local behaviour of sub-types has options to determine in which order the actions of the life-cycles on the element's type hierarchy should be executed – the default is starting with the action of the sub-types and continuing all the way up in the type hierarchy. But, we do not discuss the details here (see Chap. 4.2.1 of [9] for more information).

In addition to the local behaviour, sub-types can also add new coordination sets and new coordination annotations for an element type that inherits from another element type. These additional coordination sets and annotations are taken into account for computing valid interactions of course. Whether and to which extend this is needed and would need some extensions, is yet to be seen, since all of our examples make very limited use of this possibility.

3.5 ECNO: More Concepts

In order to get an overview, we give a brief account of some subtle additional concepts of ECNO, which we do not discuss in full detail though.

Coordination Sets and Priorities. In our example, each element type had at most one coordination set for each event type. In general, an element type can have more than one coordination sets for the same event type. For the coordination, this means that one of these coordination sets could be chosen for coordinating an event of that type with other elements. So, only the coordination annotations for one of the coordination sets of that event type needs to be taken into account. This way, the requirements for coordinations with respect to one event type is a disjunction (choice between the different coordination sets) of conjunctions of coordination annotations (all coordination annotations attached to the coordination set).

In some cases, we would like to give one coordination set preference over some others, if both of them would result in viable interactions. To this end, different coordination sets can be given a priority. Then, an interaction in which a coordination set with a higher priority is enabled will be given the preference. Section 6.1 of [9] and [16] show an application of this feature when defining the semantics of so-called Signal-Event nets – as an extension of the semantics for P/T-systems that we discuss here.

Parallel Behaviour. Sometimes, we want to allow an element to participate in two events in parallel. In our Petri net example, the ECNO net for the Place (Fig. 6) allows the place to participate in and add and a remove event at the same time. This way, a transition with a loop to some place can remove and add a token to the same place at the same time (in the same interaction). In ECNO nets, two transitions with there associated events can fire in parallel when the transitions are completely independent of each other or because there is more then token available at the places they have in common. In some cases, such as defining the semantics of Petri nets, it makes sense that the local behaviour of an element exhibits such parallel behaviour too. Therefore, ECNO supports behaviour where more than one choice is allowed to be executed at the same time, which we call *parallel behaviour*.

Counting Events. In the default case of ECNO, if an element participates in an event of the required type already, it will not participate in another event of that type, if another element also requests this: both requests will be joint on a single event. This way, we can be sure that the computation of valid interactions always terminates.

In some cases, however, we want an element to participate as many times in the event as request exists from other elements to do so. In the semantics of Petri nets for example, we would like to add a token as many times as there are arcs from the transition to the place. In order to achieve this, it is possible in ECNO for an element type to declare an event type as *triggering* or *counting event*, graphically indicated by the event type being enclosed between two plus symbols in the respective coordination sets (see element types Place and Token in Fig. 3). This makes sure that elements of that type participates in these events as many times as it is triggered by other elements. In that case, an interaction does not only take care of that an element participates in events of the respective type – it also takes care of it participating in the correct number of times.

In case of cyclic requirements, counting events can, however, result in requesting an unbounded number of participations, and consequently the computation of valid interactions might not terminate anymore (this would be ECNO's counterpart of infinite loops). Therefore, the modeller needs to take great care when making use of counting events for an element type.

3.6 Execution Engine

Above, we have discussed the concepts of ECNO and how they define possible interactions in any given situation. The possible interactions in a given situation are calculated by an ECNO execution engine – typically triggered by the controllers associated with the elements. The controllers will also issue the execution of the interactions calculated by the engine; of course, it might happen that an interaction becomes invalid due to some changes made by other interactions or some other programs running concurrently. Therefore, the interaction might not be valid anymore, when its execution is issued. The execution engine will actually take care of that interactions that became invalid after they were computed are invalidated and not executed at all. In addition, all interactions are executed in a transactional way according to the ACID principle. The ECNO execution engine can also be used to save the complete state of a running ECNO application to a file and later start the ECNO application again from there.

In the latest official release, the state (current situation) of an ECNO application is saved in a file; but it was demonstrated in a masters project that the state of an ECNO application can also be persisted in databases [22].

4 ECNO: Modelling a Workflow Engine

In this section, we discuss the ECNO models of a workflow engine, which is inspired by the ideas of AMFIBIA [12]; which distinguishes the core concepts of business processes, and separates them from the concepts of specific aspects of a business process such as control, information, and organization. Note that we discuss only the most relevant excerpts of these models. This ECNO Workflow Engine and all its models are deployed together with the ECNO Tool, so that you can have a look at the actual models in all their details yourself, and you can play with the generated workflow engine with some example processes (see detailed instructions in the ECNO technical report [9]).

4.1 Core Model

We start with discussing the core concepts of business process models and their behaviour in this section. These concepts are independent from the different aspects of business process models and independent from the formalisms used for modelling the different aspects.

Figure 9 shows the class diagram with the core concepts of our workflow meta-model. The most important concepts are shown in the two rows at the bottom (shaded in light yellow). The two top rows (shaded in grey) show some more technical infrastructure, which allows us structuring and accessing business process models and their different aspects, registering them with the engine, and maintaining the runtime information. In our discussion, we focus on the concepts at the bottom. Actually, the diagram is also split vertically: the left-hand side shows the concepts of the business process models (modelling time); the right-hand side shows the concepts of instances of business processes (runtime). Having meta-models for the modelling concepts for business process models as well as for the runtime information and clearly separating the runtime information from the model was one of the main principles of AMFIBIA already. Note that runtime information can refer to the information of the models, but not the other way round: The process models "do not know" which instances of them are running, but instances "know" the modelling concepts they are an instance of.

At the heart of AMFIBIA [12] and also of our ECNO Workflow Engine are four concepts: Process, Task, Case, and Activity, where Case represents one running instance of a Process (indicated by the reference process) and Activity represents one running instance of a Task (indicated by the reference task). On the modelling side, the main concepts are *processes* and *tasks*: a business *process* model may consist of any number of *tasks*, which, at runtime, are reflected by *cases* and their *activities*.

Note that the core concepts do not yet represent in which order the tasks (actually the corresponding activities) are supposed to be executed. Neither do the core concepts represent who is allowed to initiate or execute activities, or which data are needed for or are produced by the activities. All this is represented by models that represent different aspects of a business process. In the ECNO workflow engine, the three main aspects from AMFIBIA are covered: *control*, *organisation*, and *information*. We discuss the concepts for some of these aspects later in Sect. 4.2.

The core concepts do not mention any of the concrete aspects yet. They just provide the infrastructure so that a Process can consist of different parts that represent the concepts relating to its aspects – in the models as well as at runtime. The respective concepts are shown in the left-most column concerning models, and in the right-most column concerning the runtime information for the running instances: a process model refers to the models for the different aspects of that process; likewise the case and the activity contains the runtime information for the different aspects. Note again, that the runtime information can refer to the models, but not the other way round.

Note also that all the concepts for aspects are interfaces only. This means that specific concrete versions of them need to be defined when defining an aspect.

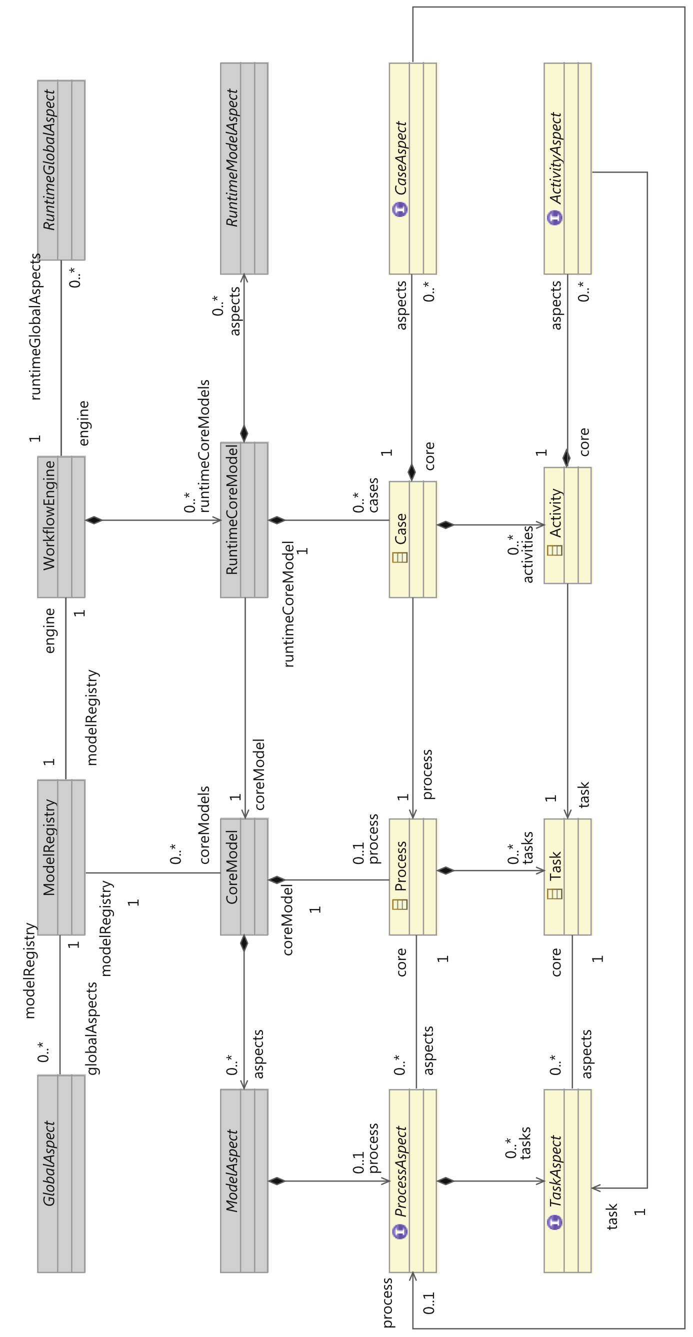


Fig. 9. BPM: Core concepts (Color figure online)

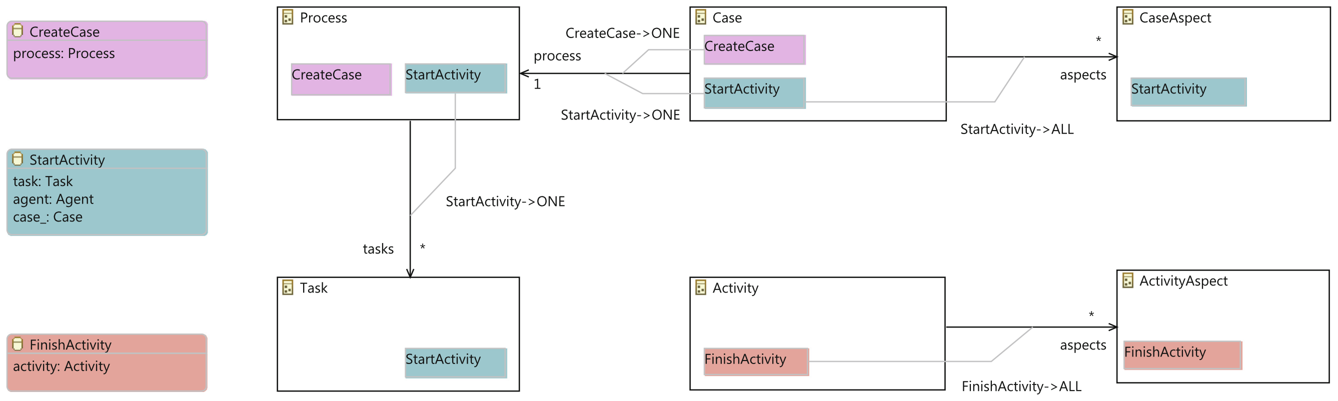


Fig. 10. BPM: Coordination diagram (core)

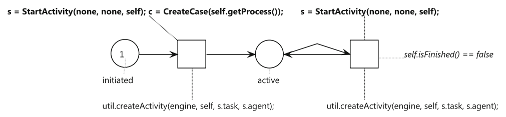

s = StartActivity(none, none, self); c = CreateCase(self.getProcess());
s = StartActivity(none, none, self);
1
initiated
active
self.isFinished() == false
util.createActivity(engine, self, s.task, s.agent);
util.createActivity(engine, self, s.task, s.agent);

Fig. 11. BPM: local behaviour of Case

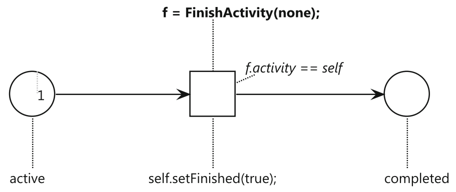


Fig. 12. BPM: local behaviour of Activity

Now, let us have a brief look at the behaviour concerning the core concepts. Figure 10 shows the coordination diagram for the core elements. This diagram defines the event type CreateCase for creating a new instance of a process, which is a Case for the respective process; moreover, the diagram defines the event types StartActivity and FinishActivity, which represent starting and finishing an activity within a case. Note that there is a slight asymmetry between the event types StartActivity and FinishActivity: the event StartActivity is triggered from a Case, whereas FinishActivity is triggered from the Activity itself. The reason is that, when starting an activity, the activity does not exist yet; therefore, it needs to be created from somewhere else, the Case. The Activity can, however, take responsibility for its own termination. Note that the CreateCase is dealt with in a slightly different way: the Case, actually, seems to trigger itself. This is a minor hack: the workflow engine always keeps one fresh instance of a case for each process ready, which is activated when the first activity of the case is started; the StartActivity will then by synchronized with the CreateCase event, which in turn will activate this case and create another fresh instance of a case for that process for the next case to be started. Conceptually, this reflects the fact that starting the first activity of a case, actually, starts the case.

The coordination diagram of Fig. 10 also shows the necessary coordinations for these event types, most of which are straight-forward. The most important coordinations concern the runtime part: the StartActivity and FinishActivity require all the aspects of the respective concept to participate in that event. This way, the coordination makes sure that activities are started and finished only when all aspects are ready for that. With the three aspects that we cover here, an activity can be started only when the control allows to start it (control aspect), all the needed data are ready (information aspect), and there is an agent available who is allowed to perform this activity (organisation aspect).

Most of the life-cycles of the core concepts are trivial – meaning that all events are possible anytime (there are some minor twists, though, which we do not discuss here). The most interesting local behaviours are the ones for Case and Activity, which are shown in Figs. 11 and 12, respectively.

We start discussing the life-cycle of the Case. Remember that there is always one fresh case, which is ready for being activated by starting one of its initial activities. The case is activated by a CreateCase event, which actually is jointly executed together (synchronized) with the StartActivity event that starts the first activity of the case at the same time. After that, the Case can participate in further StartActivity events without synchronizing it with another CreateCase event. In a running case, StartActivity events can happen as long as the case is not finished, which is represented by an additional condition.

Figure 12 shows the life-cycle of the Activity. This is almost trivial, making sure that every activity can finish only once, which is similar to the local behaviour of tokens in the ECNO semantics of Petri nets.

4.2 Models for Aspects and Formalisms

Next, we discuss some of the models relating to the different aspects of business processes. Note that we restrict this discussion to the control aspect and a part of the organisation aspect. The ECNO workflow engine covers the information aspect too, but we do not discuss this aspect here.

We start with the discussion of the control aspect as well as one formalism for modelling the control aspect of business processes: Petri nets or actually workflow nets. Note that AMFIBIA set out to separate the concepts of an aspect from the realization of these concepts in a concrete formalism. Anyway, we discuss the general concepts of the control aspect together with a concrete modelling formalism here.

Figure 13 shows the general concepts of the control aspect as well as how these concepts can be captured by the Petri nets formalism – actually by *workflow nets* [19,23]. The classes in the top row (in light yellow) represent the concepts from the core model again (as see in Fig. 9 already). The classes in the two rows below (in light blue) represent the general concepts of the control aspect, and the classes below that (in magenta) show the concepts of Petri nets implementing the concepts of the control aspect. Like before, the classes on the left-hand side represent the modelling concepts, whereas the classes on the right-hand side represent the runtime concepts.

The class TaskC represents the control aspect of a task (it implements TaskAspect), which in Petri nets is realized as a Transition. On the runtime side, the class ActivityC represents the control aspect of an activity (implementing ActivityAspect), which in turn refers to the control aspect of the case CaseC. The most important part of the control aspect is that a case has a concept of a State, which determines which activities are possible to be started in the current situation. In Petri nets, the State is realized as a Marking, which is represented by a set of tokens associated with some places of the Petri net. The model for Petri nets here, roughly, resembles the model of Petri nets that we had seen in Sect. 2.1. The most important difference is that tokens are not contained in places anymore, but are part of the marking; instead, each token refers to the place it belongs to. The reason for detaching tokens from places in this model is that tokens represent runtime information, which the actual model should not know about. This leaves the question of how the initial marking of the Petri net is represented in the model itself. To this end, we exploit a speciality of workflow nets: they always start in a specific marking, exactly one token on a so-called start place. So, the model does not need to represent the initial marking; we just need to represent the `start` place and the `finish` place of the net. This is reflected by the references from the PetriNet to the start and finish place. Transitions that are enabled when a single token is added to the start place correspond to the initial tasks of a process, which implicitly start the process (as discussed above).

The more interesting part of the models for the control aspect concerns the behaviour at runtime. The corresponding coordination diagram is shown in Fig. 14. Starting an activity requires the control aspect of the case (CaseC) to

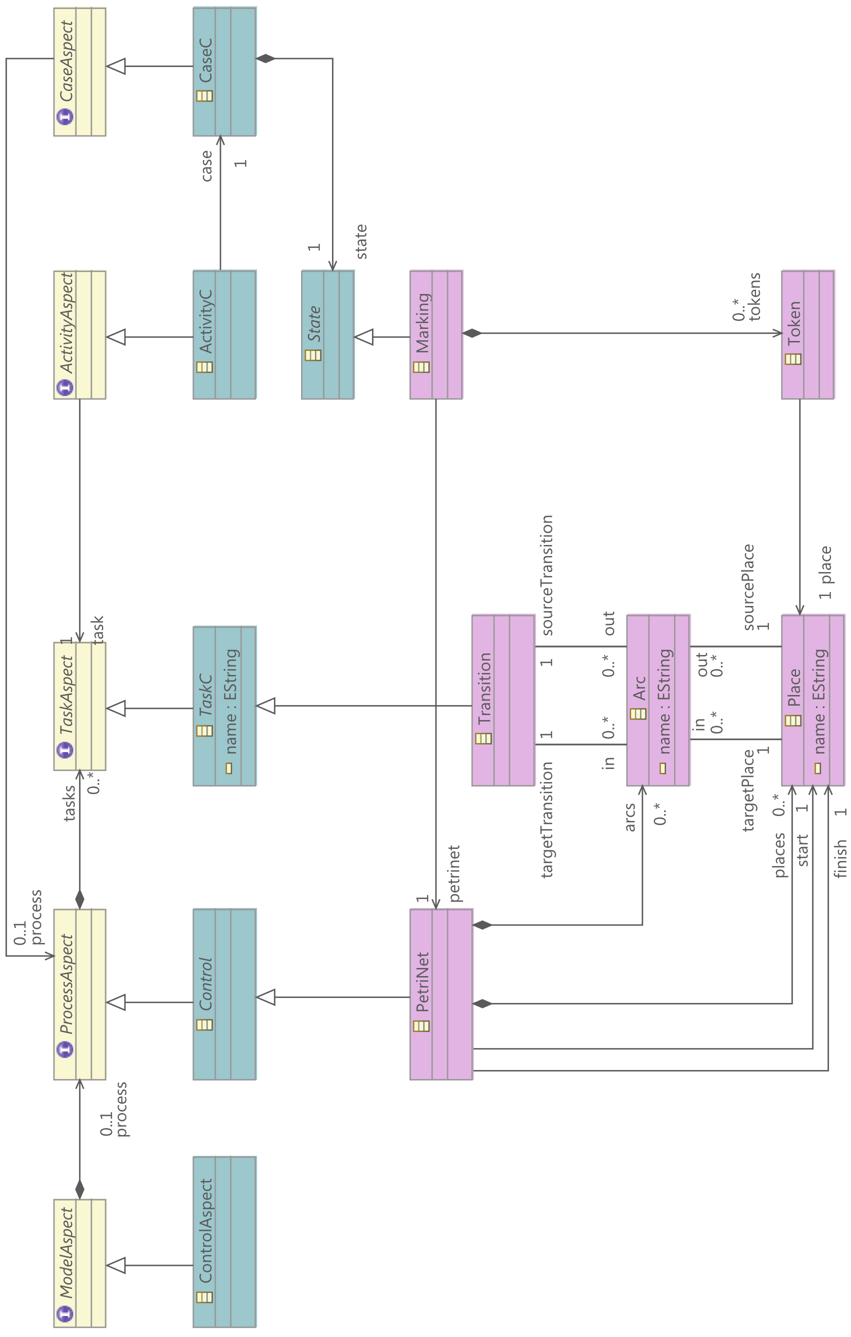


Fig. 13. BPM: Concepts for control and Petri net formalism

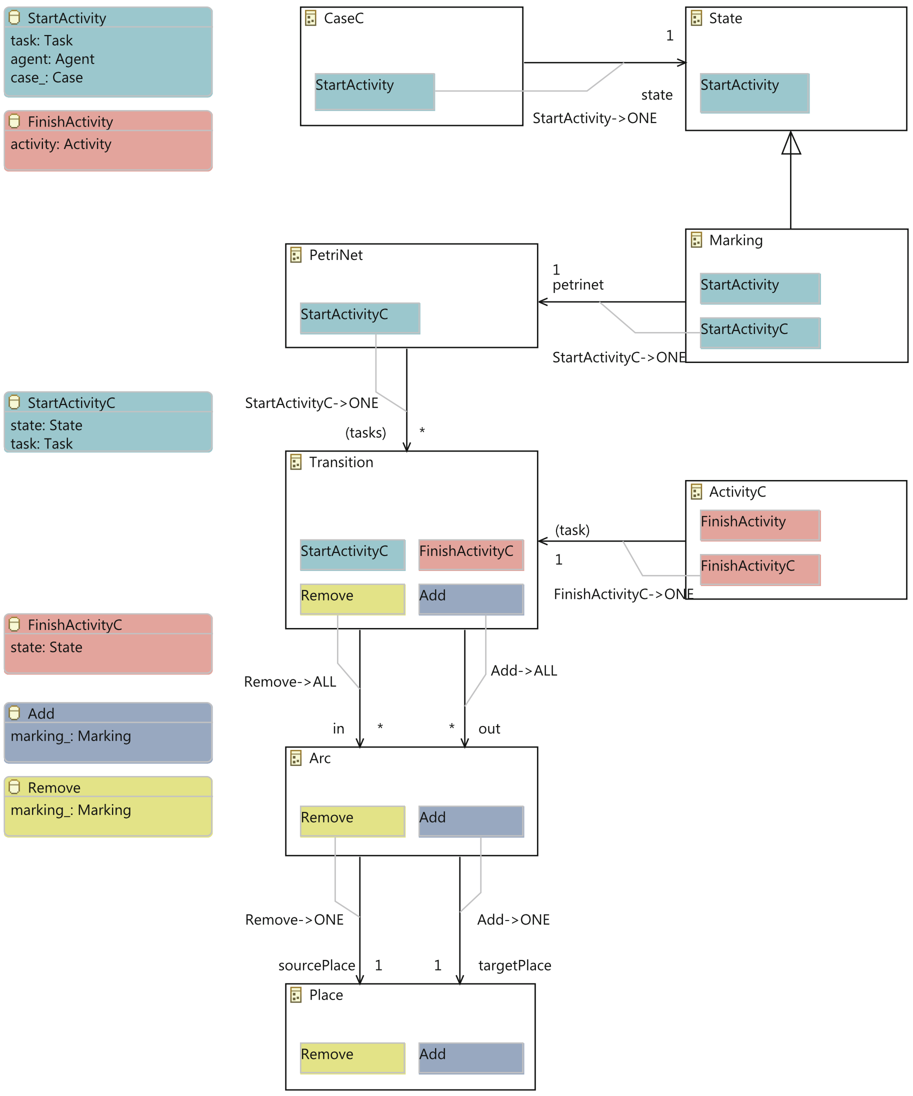


Fig. 14. BPM: Coordination diagram for control aspect and Petri net formalism

coordinate the event StartActivity with its State, which will require a synchronisation with another event StartActivityC; this event represents the control aspect of the event StartActivity. The state, in turn, will need the model for the control aspect – in our case the Petri net – to participate in the StartActivityC event, which will require a Remove event on a Transition to be part of the coordination. The Remove is used and handled in a similar way to the Petri net example that we had discussed in Sect. 2.1. Note that in contrast to the original semantics of Petri nets which fires a transition instantaneously, workflow nets are executed in two steps: Starting the activity removes the tokens from the input places, whereas finishing the activity adds the tokens to the output places. Note that this way, in workflow nets a transition does note fire instantaneously, but take time; this reflects the fact that activities in workflows take time.

The start of an activity needs to be issued from the case since an instance of the activity is created only upon starting it. By contrast, the activity can take care of its own termination. The ActivityC coordinates a FinishActivityC event with the respective transition, which in turn coordinates it with an add event which adds all the tokens to the postset of the transition – similar to the Petri net semantics that we had discussed earlier.

The only interesting local behaviours are the life-cycles of the elements of the Petri net. Since these are similar to the ones discussed in Sect. 2.1, we do not discuss them here. All the other local behaviours are quite simple. Most importantly they synchronize the StartActivity event with the StartActivityC event, and likewise the FinishActivity event with the FinishActivityC event.

Next, let us have a brief look at the organisation aspect. Since the underlying class diagram for this aspect is quite simple, we skip it and discuss the coordination diagram for the organization aspect right away, which is shown in Fig. 15. Basically, the organisation aspect for the case, CaseO, delegates the StartActivity event to one of the (possibly) involved Agents; likewise the organisation aspect of an activity ActivityO delegates the FinishActivity event to the Agent to which this activity was assigned.

The ECNO net for the life-cycle of the Agent is shown in Fig. 16. From the organisation point of view, an agent can start any activity as long as it does so on its own behalf, and the organisation model allows the agent to do so, which is represented by the additional conditions, which are attached to the transitions of the ECNO nets.

4.3 Workflow Engine and GUI

From the models above and a few more models, which are similar, a fully functioning workflow engine can be generated fully automatically. The only part that needed to be implemented manually was the GUI – in particular, the worklist which allows the agents to log in and to select, start and finish work items. The implementation of this GUI, however, is straight-forward. We do not discuss it here (see [9,11] for details), since all behaviour comes from the ECNO models.

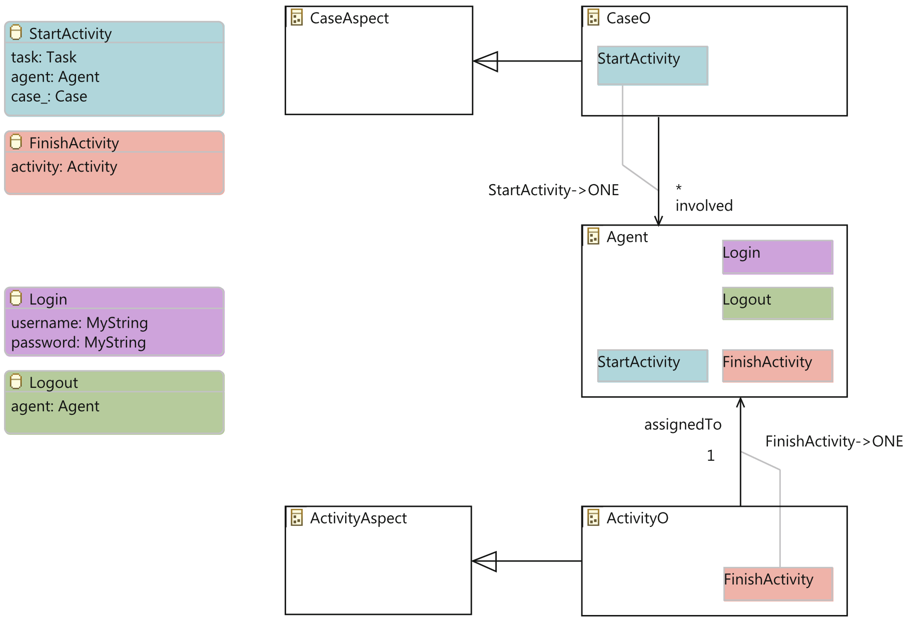


Fig. 15. BPM: Coordination diagram for organisation

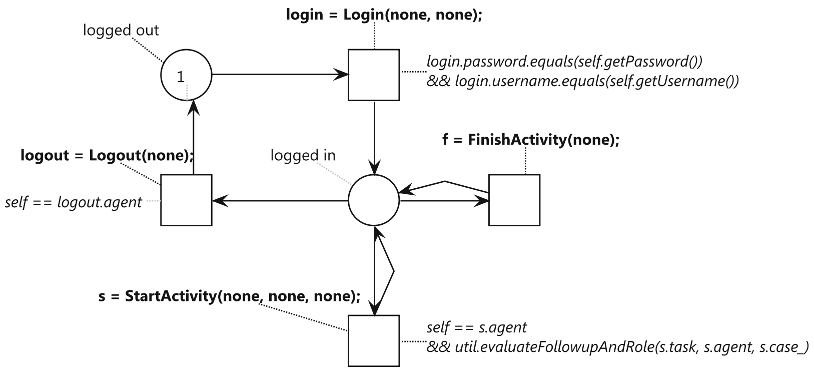

login = Login(none, none);
logged out
1
login.password.equals(self.getPassword())
&& login.username.equals(self.getUsername())
f = FinishActivity(none);
logout = Logout(none);
logged in
self == logout.agent
s = StartActivity(none, none, none);
self == s.agent
&& util.evaluateFollowupAndRole(s.task, s.agent, s.case_)

Fig. 16. BPM: local behaviour of Agent

The workflow engine can then be started with some process models, which cover the control, organisation, and information aspect. Up to now, there is only a simple tree editor for creating and editing such process models, since the focus of this project was on the feasibility and not the usability of its editors. Once the workflow engine is started, the different agents can log on, and via the GUI start cases, activities, and inspect, add, and change the data involved in an activity, and then finish the activity. For lack of space, we cannot discuss the use of the workflow engine and the execution of some example processes here. Some example processes are deployed together with ECNO and the ECNO workflow engine; the installation and the use of the examples are discussed in the ECNO technical report [9].

5 Related Work

The ideas of ECNO have evolved over many years and started out from a meta-model that distilled the essence of business process modelling notations: AMFIBIA [12]. For capturing the behaviour of these concepts, AMFIBIA used a simple ad-hoc notation for the local behaviour and for the coordination of the behaviour of the different elements. The ad-hoc notation that we used in AMFIBIA was later formalized and implemented in a kind of pre-cursor of ECNO, which we called MoDowA [24,25]. The idea of ECNO goes back to the ad-hoc notations of AMFIBIA and MoDowA; ECNO is more general and the too tight integration with aspect-orientation was dropped, so that ECNO – at its core – is not explicitly aspect-oriented anymore. Moreover, inheritance was introduced for elements and for events, which needed some careful tuning; for that reason, there is a complete Chap. 4 that is devoted to the discussion of inheritance in the ECNO technical report [9]. Starting out from the challenges of behaviour modelling [1], we then defined the core concepts of ECNO [13–15] with minor variations, which now seem to converge.

As pointed out in our earlier work [15] already, none of the concepts used in ECNO are particularly new or original; the contribution of ECNO is more in the careful combination of its concepts and, on the technical side, its integration with existing object-oriented technologies.

ECNO's coordination mechanism via events resembles the synchronization of actions in process algebras [4,5,7]. One difference, though, is that ECNO's synchronization is not restricted to bi-lateral synchronizations and that the partners required to participate in an event might dependent on the dynamically changing underlying structure of the system. Also this aspect has been seen before in process algebras like ACP [26], the chemical abstract machine [27], or the Π-calculus [28]. What is new, however, is that, in ECNO's coordination mechanism, different of these synchronization mechanisms work together, combining these coordination requirements transitively, which allows us to define much more complex interactions.

The proposal of behavioral programming [29] exploits the idea of synchronizing events for programming concurrent behaviour (b-threads). The idea of behavioural programming is very similar in spirit to ECNO – with a focus on programming. But, in behavioural programming possible synchronizations on the same event are global and not driven by the dynamic relation to other objects; moreover, behavioural programming does not allow synchronizing different events or transitively combining synchronizations into more complex interactions, nor does it come with an inherent notion of joint atomic execution when synchronizing on events.

Another major concern in the design of ECNO was the clear separation between coordination aspects and computation aspects of a system. Actually, ECNO is about coordination only, but ECNO's concept of actions provides a way to interface with the computational aspects by invoking methods or functions. This idea, however, is not new either: Harel and Pnueli [30] had proposed the distinction between transformational and reactive systems. ECNO takes care of the reactive aspect of the system by defining possible interactions – the transformational aspect is left to the underlying programming language (Java in our case) for the actions by invoking methods.

Another major concern of ECNO is the distinction between local behaviour and global behaviour [1]. Also this idea is not really new: Harel and Marelly [10] distinguish between intra-object behaviour and inter-object behaviour, which correspond to local and global behaviour, respectively. The only difference is the way this behaviour is represented. Concerning the local behaviour, this is mostly a question of syntactic sugar. For inter-object behaviour (global behaviour), Harel and Marelly use a set of Live Sequence Charts (LSCs) [31], which are an extension of Message Sequence Charts [32]. This is a scenario-based and temporal approach, where the focus of inter-object behaviour is the behaviour over time. In ECNO, the coordination annotations focus on the needed partners for a single interaction only: it is about behaviour at a time. Therefore, both approaches have a different focus. It might be interesting to combine both of them; this might in particular be interesting since ECNO does not have a way to define what must happen in a system – it defines what can happen only. LSCs [31] allow to characterize both kinds. But a detailed investigation of such a combination would require further research.

As discussed above, the ideas of ECNO started out from an ad-hoc notation in which aspects were an explicit modelling concept and therefore, ECNO has some relation to aspect-oriented programming [33,34] or aspect-oriented modelling [35,36]. Actually, from the philosophical angle, the original ideas were close to the Theme approach [37] and closer to the idea of subject-oriented programming [38]. Anyway, the explicit notion of aspects was removed in ECNO again. A bit of the original subject-oriented idea survived in one of the two different concepts of inheritance on event types, which we did not discuss here. And by using some specific modelling patterns, ECNO can be used for modelling in an aspect-oriented way: In a way, events of ECNO can be considered to be join points of AspectJ [39]. The difference, though, is that events are an explicit modelling concept [40], whereas join points are formulated on top of a program. This way, events are a concept of the domain, whereas join points are programming artifacts (which of course could have a counter-part in the domain). The coordination annotations of ECNO then correspond to pointcuts. Though stripped of an explicit notion of aspects, ECNO still shares some philosophy with aspect- or subject-orientation: joining events together via coordination annotations into interactions.

The local behaviour of elements could be modelled in many different ways. We could use traditional automata or StateCharts [6]. We mainly use a special form of

Petri nets [3,41], which we call ECNO nets. Initially, the reason for using ECNO nets was mostly a practical one: we could use our own framework for Petri net tools, the ePNK [42], for easily implementing a graphical editor for ECNO nets. And the ePNK is based on EMF [8], which is the object-oriented technology that happens to be the default object-oriented technology of ECNO. But, it turned out to be useful that Petri nets have a natural notion of concurrent or parallel firing of transitions, when it comes to parallel behaviour (see Sect. 3.5). Therefore, simple automata are not sufficient for modelling the local behaviour of elements. Like Petri nets, StateCharts have a notion of parallel behaviour, which makes them an other good candidate for modelling local behaviour, too. Our main concern with StateCharts would be that they might be too powerful: modellers might be tempted to put too much into the local behaviour of elements, since StateCharts allow nested complex states. But, this is up to future evaluation and a question of methodology, which is yet to be worked out in full detail.

At last, ECNO has some similarities with *agent-based software engineering* and *Multi-Agent Systems* (MAS) [43,44]; but, at least in its basic form, ECNO would probably not qualify as an approach towards agent-based software engineering. This, however, depends on which level we look at things: From our point of view, ECNO is more a notation and technique[5] whereas agent-based software engineering is more a way of thinking. Anyway, some of the principles underlying ECNO were proposed by the proponents of agent-based software engineering. The two most important shared principles are: getting rid of the thread-oriented way of thinking, and giving agents control over what they do or to which kind of request they react or – as we would say in ECNO – in which events they participate. In addition, in agent-based software engineering, agents have attitudes and are pro-active and take initiative. Even disregarding the more social notions of initiative and attitude, ECNO elements are not even active – remember that ECNO models describe what can happen in a given situation, but they do not describe what must happen. Therefore, ECNO's elements are technically not agents. But, by adding controllers on top of elements, elements can be turned active. This way, ECNO might be a notation and technique in which agent-based designs or agent-based thinking can be formulated and implemented. But, this is up to others to judge.

Speaking of agents, we should mention another approach, which uses Petri nets for defining local behaviour: Renew [45]. Renew also uses a mechanism for synchronizing different parts of a system with each other following some fixed relations between these parts. But, theses synchronisations need to follow some very specific containment structures following the so-called nets-within-nets paradigm [46]. By contrast, ECNO models can exploit the dynamic structure of the underlying object-oriented model for defining the required partners, which was one of its express goals.

Altogether, ECNO has many different flavours. On a first glance and depending on ones background, ECNO might appear as just another process algebra, just another notation for aspect-oriented modelling, just another agent-based

[5] The methodology part of this technique is yet to be worked out in detail.

approach, just another form of transactions, just another ... – and there might be some truth to that. But, we believe that it is the combination of these different things and a carefully adjusted set of concepts that makes ECNO what it is: A way of clearly separating coordination from computation, and of separating coordination from local behaviour.

6 Conclusion

In this paper, we have given an overview of ECNO and motivated some of its concepts and definitions. Moreover, we have discussed an ECNO model of a workflow engine, which demonstrates that ECNO can be used for "beyond Mickey Mouse examples". From this ECNO model, a complete workflow engine can be generated fully automatically [11]. Together with the other examples [9], this shows that ECNO can be used for a wide range of different applications.

Modelling the workflow engine consisted in providing a domain-specific language (DSL) for workflow models; this DSL was defined by class diagrams concerning the abstract syntax of the DSL; on top of these class diagrams ECNO models defined the actual behaviour (semantics) of this DSL. In a similar way, ECNO was used for defining the semantics for Petri nets – again a meta-model was provided for Petri nets; ECNO models on top of these meta-models defined the semantics of Petri nets. This shows that ECNO can be used to define and implement also the semantics of a DSL. Actually, we believe that the semantics of ECNO can be defined in ECNO itself, which however is yet to be worked out in detail.

ECNO shows that there are mechanisms beyond method invocation for integrating behaviour models with structural models. The implementation of the ECNO framework and tool shows that interactions can be executed in a transactional way, and this way be executed in a multi-threaded or concurrent environment without explicitly thinking about threads or modelling them. Since ECNO is independent of a specific underlying object-oriented technology, it can also be used for integrating software using different technologies.

What is still missing is a coherent methodology with modelling guidelines and best practices for properly using ECNO, which we plan to work out in the future. In order to gain more experience and to work out this methodology, we will need some more examples of realistic size. The currently published version of ECNO and the corresponding ECNO Tool[6], are a good basis for working on some more realistic examples.

References

1. Kindler, E.: Model-based software engineering: the challenges of modelling behaviour. In: Aksit, M., Kindler, E., Roubtsova, E., McNeile, A. (eds.) Proceedings of the Second Workshop on Behavioural Modelling - Foundations and Application (BM-FA 2010), pp. 51–66 (2010) (Also published in the ACM electronic libraries)

[6] see http://www2.compute.dtu.dk/~ekki/projects/ECNO/index.shtml.

2. OMG: MDA guide v1.0.1. (2003). http://www.omg.org/cgi-bin/doc?omg/03-06-01
3. Petri, C.A.: Kommunikation mit Automaten. Technical report Schriften des IIM, Nr. 2, Institut für instrumentelle Mathematik, Bonn (1962)
4. Hoare, C.: Communicating sequential processes. Comm. ACM **21**(8), 666–677 (1978)
5. Hoare, C.: Communicating Sequential Processes. Prentice-Hall, Upper Saddle River (1985)
6. Harel, D.: Statecharts: a visual formalism for computer systems. Sci. Comput. Program. **8**(3), 231–274 (1987)
7. Milner, R.: Communication and Concurrency. International Series in Computer Science. Prentice Hall, Upper Saddle River (1989)
8. Budinsky, F., Steinberg, D., Merks, E., Ellersick, R., Grose, T.J.: Eclipse Modeling Framework. The Eclipse Series, 2nd edn. Addison-Wesley, Reading (2006)
9. Kindler, E.: Coordinating interactions: The Event Coordination Notation. Technical report DTU Compute Technical report 2014–05, DTU Compute, Kgs. Lyngby, Denmark (2014)
10. Harel, D., Marelly, R.: Come Let's Play: Scenario-based Programming Using LSCs and the Play-engine. Springer, Heidelberg (2003)
11. Jepsen, J.: Realizing a workflow engine with the Event Coordination Notation. Master's thesis, Technical University of Denmark, DTU Compute (2013) IMM-M.Sc.-2013-101
12. Axenath, B., Kindler, E., Rubin, V.: AMFIBIA: a meta-model for the integration of business process modelling aspects. Int. J. Bus. Process Integr. Manag. **2**(2), 120–131 (2007)
13. Kindler, E.: Integrating behaviour in software models: an Event Coordination Notation - concepts and prototype. In: Proceedings of the Third Workshop on Behavioural Modelling - Foundations and Application (BM-2011) (2011)
14. Kindler, E.: The Event Coordination Notation: execution engine and programming framework. In: Störrle, H., Botterweck, G., Bourdellès, M., Kolovos, D., Paige, R., Roubtsova, E., Rubin, J., Tolvanen, J.P. (eds.) Fourth Workshop on Behavioural Modelling - Foundations and Application (BM-FA 2012), Joint proceedings of co-located events at ECMFA 2012, pp. 143–157 (2012)
15. Kindler, E.: Modelling local and global behaviour: Petri nets and event coordination. Trans. Petri Nets Other Models Concur. **6**, 71–93 (2012)
16. Kindler, E.: An ECNO semantics for Petri nets. Petri Net Newslett. **81**, 3–16 (2012). Cover Picture Story
17. Reisig, W.: Place/Transition systems. In: Brauer, W., Reisig, W., Rozenberg, G. (eds.) Petri Nets: Central Models and Their Properties. LNCS, vol. 254, pp. 117–141. Springer, Heidelberg (1987)
18. van der Aalst, W.: Exploring the process dimension of workflow management. Computing Science Reports 97/13, Eindhoven University of Technology (1997)
19. van der Aalst, W., van Hee, K.: Workflow Management: Models, Methods, and Systems. Cooperative Information Systems. The MIT Press, Cambridge (2002)
20. OMG: OMG Unified Modeling Language (OMG UML), Superstructure, V2.1.2. Object Management Group, 140 Kendrick Street, Needham, MA 02494, USA (2007) OMG Document number: formal/2007-11-02
21. Gray, J., Reuter, A.: Transaction Processing: Concepts and Techniques. Morgan Kaufmann, San Mateo (1993)
22. Nielsen, H.E.: A database integration for the Event Coordination Notation. Master's thesis, Technical University of Denmark, DTU Compute (2014)

23. van der Aalst, W.: Verification of workflow nets. In: Azéma, P., Balbo, G. (eds.) ICATPN 1997. LNCS, vol. 1248, pp. 407–426. Springer, Heidelberg (1997)
24. Schmelter, D.: Eine Technik zur Entwicklung und Ausführung aspektorientierter Modelle. Master's thesis, Department of Computer Science, Software Engineering Group, University of Paderborn, Paderborn, Germany (2007)
25. Kindler, E., Schmelter, D.: Aspect-oriented modelling from a different angle: modelling domains with aspects. In: 12th International Workshop on Aspect-Oriented Modeling (2008)
26. Bergstra, J.A., Klop, J.W.: Process algebra for synchronous communication. Inf. Control **60**(1–3), 109–137 (1984)
27. Berry, G., Boudol, G.: The chemical abstract machine. In: POPL, pp. 81–94 (1990)
28. Milner, R., Parrow, J., Walker, D.: A calculus of mobile processes (Parts I & II). Inf. Comput. **100**(1), 1–40 & 41–77 (1992)
29. Harel, D., Marron, A., Weiss, G.: Behavioral programming. Commun. ACM **55**(7), 90–100 (2012)
30. Harel, D., Pnueli, A.: On the development of reactive systems. In: Apt, K. (ed.) Logics and Models of Concurrent Systems. Series F: Computer and System Science, vol. 13, pp. 477–498. Springer, Heidelberg (1985)
31. Damm, W., Harel, D.: LSC's: Breathing life into message sequence charts. In: Ciancarini, P., Fantechi, A., Gorrieri, R. (eds.) FMOODS 1999. IFIP, vol. 10, pp. 293–311. Springer, Boston (1999)
32. ITU-T Recommendation Z.120: Message sequence charts (MSC). ITU (1996)
33. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J.M., Irwin, J.: Aspect-oriented programming. In: Moreira, A. (ed.) ECOOP 1997. LNCS, vol. 1743, pp. 220–242. Springer, Heidelberg (1997)
34. Mens, K., Lopes, C., Tekinerdogan, B., Kiczales, G.: Aspect-oriented programming workshop report. In: Bosch, J., Mitchell, S. (eds.) ECOOP 1997. LNCS, vol. 1357, pp. 483–496. Springer, Heidelberg (1998)
35. Brichau, J., Haupt, M.: Survey of aspect-oriented languages and execution models. Technical report AOSD-Europe-VUB-01, AOSD-Europe (2005)
36. Chitchyan, R., Rashid, A., Sawyer, P., Garcia, A., Alarcon, M.P., Bakker, J., Tekinerdogan, B., Clarke, S., Jackson, A.: Survey of aspect-oriented analysis and design approaches. Technical report AOSD-Europe-ULANC-9, AOSD-Europe (2005)
37. Clarke, S., Baniassad, E.: Aspect-Oriented Analysis and Design: The Theme Approach. Addison-Wesley, Reading (2005)
38. Harrison, W., Ossher, H.: Subject-oriented programming (a critique of pure objects). In: OOPSLA, pp. 411–428. ACM (1993)
39. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An overview of AspectJ. In: Lindskov Knudsen, J. (ed.) ECOOP 2001. LNCS, vol. 2072, pp. 327–353. Springer, Heidelberg (2001)
40. Douence, R., Noyé, J.: Towards a concurrent model of event-based aspect-oriented programming. In: European Interactive Workshop on Aspects in Software (EIWAS 2005) (2005)
41. Reisig, W.: Petri Nets. EATCS Monographs on Theoretical Computer Science, vol. 4. Springer, Berlin (1985)
42. Kindler, E.: The ePNK: an extensible Petri net tool for PNML. In: Kristensen, L.M., Petrucci, L. (eds.) PETRI NETS 2011. LNCS, vol. 6709, pp. 318–327. Springer, Heidelberg (2011)
43. Wooldridge, M.: Agent-based software engineering. IEE Proc. Softw. Eng. **144**(1), 26–37 (1997)

44. Jennings, N.R., Sycara, K.P., Wooldridge, M.: A roadmap of agent research and development. Auton. Agent. Multi-Agent Syst. **1**(1), 7–38 (1998)
45. Kummer, O., Wienberg, F., Duvigneau, M., Schumacher, J., Köhler, M., Moldt, D., Rölke, H., Valk, R.: An extensible editor and simulation engine for Petri nets: RENEW. In: Cortadella, J., Reisig, W. (eds.) ICATPN 2004. LNCS, vol. 3099, pp. 484–493. Springer, Heidelberg (2004)
46. Valk, R.: Petri nets as token objects: an introduction to elementary object nets. In: Desel, J., Silva, M. (eds.) ICATPN 1998. LNCS, vol. 1420, pp. 1–25. Springer, Heidelberg (1998)