Modelling Local and Global Behaviour: Petri Nets and Event Coordination

Ekkart Kindler

Informatics and Mathematical Modelling
Technical University of Denmark
eki@imm.dtu.dk

Abstract. Today, it is possible to generate major parts of a software system from models. Most of the generated code, however, concerns the structural parts of the software; the parts that concern the functionality or behaviour of a system are still programmed manually. In order to overcome this problem, we are developing the concept of *coordination diagrams* that define the global behaviour on top of structural software models. Basically, these diagrams define how the local behaviour of an element is coordinated with the behaviour of the elements it is connected to. The exact concepts of these coordination diagrams and their notation is still under development, but there exists a first prototype for experimenting and for fine-tuning its features. We call it the *Event Coordination Notation* (*ECNO*).
For experimenting with the ECNO, we implemented also a simple modelling notation for the local behaviour, which is based on Petri nets. In this paper, we briefly discuss the general idea of the ECNO and then present *ECNO nets* that define the local behaviour of elements. They are implemented as a Petri net type for the ePNK tool, together with a code generator that produces code that can be used in the ECNO framework and runtime environment. This way, all the behaviour of a system can be modelled – and code can be generated that easily integrates with the structural models and existing software.
Keywords: Model-based Software Engineering, Local and global behaviour modelling, Event coordination, Code generation, ECNO nets.

1 Introduction

Software models and the automatic generation of code from these models are becoming more and more popular in modern software development – as suggested by the success of one of the major approaches, the Model Driven Architecture (MDA) [1]. In many cases, however, code generation concerns the structural parts or the standard parts of the software only; as soon as the actual functionality and behaviour is concerned, major parts of the software are still programmed manually. As pointed out in previous work [2], the reason is not so much that there are no modelling notations for behaviour or that it would be difficult to generate code from these models. The actual problem is to integrate the behaviour models or the code generated from them with the code generated

from the structural models and with pre-existing parts of the software. One of the main reasons for that problem is that, ultimately, the only mechanism for integrating different parts of the software is invocation – of functions, of methods, or of services.

Based on some earlier ideas [3, 4, 2], we set out to develop a concept that allows us to identify *events* in which different partners or parts of the software could or should participate; the notation allows us to define how the execution of these events should be *coordinated*. The overall behaviour of the system would then be a result of this coordination and synchronization of events combined with the local behaviour of the different parts. In a way, this is similar to aspect orientation with its join points and point cuts [5], but with a different focus. The development of this notation is still in progress; in order to gain some experience, though, we have implemented a first prototype, which consists of a modelling part and an execution runtime environment, which we call *Event Coordination Notation* (*ECNO*). From this prototype, we hope to learn more about which constructs do help to adequately model and coordinate behaviour and to collect efficiency and performance results for larger systems and for more complex coordinations. This way, we intend to fine-tune ECNO's constructs and notations and to strike a balance between obtaining an adequate notation on a high level of abstraction, with sufficient expressive power and universality on the one-hand side, and efficient and performant execution on the other side.

As stated above, the research on ECNO is still in progress. Its major concern is the integration of the *local behaviour* of elements by coordinating the execution of their events. In ECNO, the focus is on the coordination, i. e. on *global behaviour*. The local behaviour would still be programmed manually based on an API which is part of the ECNO framework (see [6] for more details). On the one hand side, the possibility of programming the local behaviour in a traditional way, makes it possible to integrate ECNO with classical software development approaches. We consider this possibility as a major feature of ECNO – in particular easing the gradual transition from programmed software to modelled software. One the other hand, programming is very tedious and not very much in the spirit of model-based software engineering (MBSE). Therefore, we started a side-line that is concerned with modelling the local behaviour and generating code from that. We use an extended version of Place/Transition nets (P/T-nets) [7, 8] for that purpose – mostly, because of our own background in Petri nets and because we have a generic tool, the ePNK [9], which allows us to easily define new net types of Petri nets and which is integrated with the eclipse platform that provides all the necessary infrastructure and technology for code generation from models and software development in general [10].

The main contribution of this paper is on the extended version of Petri nets (Sect. 3) for modelling local behaviour: ECNO nets. But, in order to understand the local behaviour models we need to see the context in which this local behaviour is coordinated. Therefore, we start with explaining the idea and concepts of ECNO (Sect. 2).

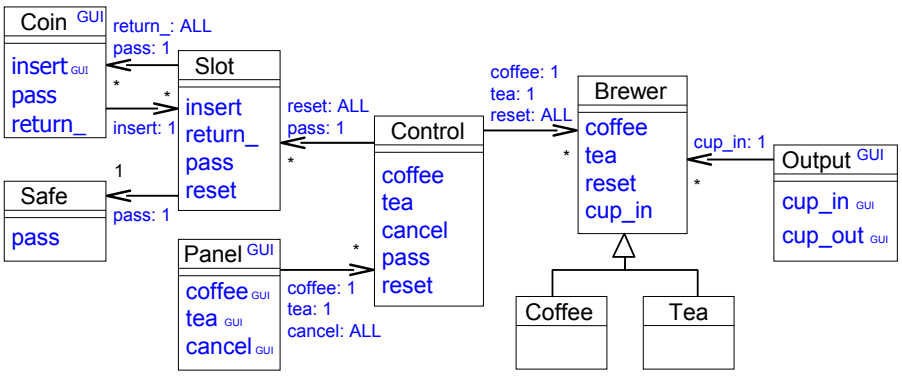

Fig. 1. A class and coordination diagram

2 The Event Coordination Notation

In this section, we discuss the main ideas and main concepts of ECNO. We start with an example in Sect. 2.1 and summarize the concepts in Sect. 2.2.

2.1 Example

The concepts of ECNO will be explained by using a simple example: the eternal coffee (and tea) vending machine. Figure 1 shows a UML class diagram[1] with some extensions concerning *events* and their *coordination*. Therefore, we call it *coordination diagram*.

Before explaining the extensions, let us have a brief look at it as a UML class diagram. The diagram shows the different possible *elements*[2] that are part of the system: A coin can be close to the slot, which is represented by the reference from Coin to Slot, or a coins can be in the slot, which is represented by the reference from Slot to Coin. There is a Safe to which the coin will be passed, once a coffee or tea is dispensed. There is a Panel for the user to interact with the vending machine. This panel is connected to controllers, which is represented by the reference from Panel to Control. The controllers are connected to the brewers, which can be either coffee or tea brewers. At last, there is an output device for the beverage, which is connected to the brewers. Note that Fig. 1 is a kind of class diagram. Therefore, there can be different configurations, which could be defined as an *instance* of this class diagram (in UML this would be an object diagram). In our example, we assume that there are (initially) three coins (not inserted yet to the slot), and that there are two coffee brewers and one tea brewer; for

[1] For the experts, this actually is an EMOF diagram [11] or an Ecore diagram [10]; but this is not relevant here.

[2] In order to point out that our objects are not objects in the traditional sense of object orientation, we call them *elements*.

all other classes, we assume that there is exactly one instance. Remember that, in these instances, the references of the class diagram are represented by *links* from one element to another (where the link's type is the reference).

Now, let us explain the extensions concerning the coordination of events between the different elements: First of all, there are some events mentioned in the operations section of the class diagram, like insert, pass and return_ [3] for Coin. They define in which *events* the different elements could be involved or could participate. The actual definition of these events will be discussed later (see Fig. 2). More importantly, the references between the different elements are annotated with events and an additional quantifier, which can be 1 or ALL. These annotations define the coordination of events and which events and elements needed to be executed together. We call such a combination of elements and events an *interaction*. The meaning of these annotations is as follows: Let us assume that some element is involved in the execution of some event and that, in the coordination diagrams, the type of this element has a reference that is labelled with that event. Then, some elements at the other end of the respective link also need to participate: to be more precise, if the event is quantified by 1, exactly one of these elements must participate; if the event is quantified by ALL, all the elements at the other end of these links need to participate.

In our example, let us assume that there are two coins inserted to the slot. This would be represented by two links from the slot to a coin – one to each of the coins. If the slot does a return_ event, the annotation return_:ALL at the reference from Slot to Coin means, that both coins must participate in the execution of the event return_.

Note that, normally, this is required for all the references that are annotated with the event. In our example, there are two references from Slot that are annotated by pass – one to Coin and one to Safe. Therefore, when a slot participates in a pass event, exactly one coin and exactly one safe will need to participate in this interaction – this way, the coin will be passed from the slot to the safe, which will be discussed later in Sect. 3.1.

In some cases, we do not want all these references to be considered. To this end, the ECNO provides the concept of *coordination sets*, which allows us to define which references should be followed together. But, we do not discuss the details here since this is not relevant in this paper (see [6] for some more details).

Up to now, we have used events basically as names. In general, events can have parameters, which can be used to share information between the different partners that are participating in the event and the interaction. Figure 2 shows the declaration of all events of our vending machine along with the *event parameters*. On a first glance, this looks very much like method declarations. In contrast to methods, events do not have behaviour of their own, and they do not belong to a particular element (or are not owned by them). Therefore, event parameters can be contributed in many different ways, and by different elements. It is not defined in advance, who will provide and who will use the parameters.

[3] Since return is a key word of our target language Java, we use an additional underscore in the event name return_.

```
insert(Coin coin, Slot slot);      coffee();
pass(Coin coin, Slot slot);        tea();
return_(Slot slot);                cup_in();
reset();                           cup_out();
cancel();
```

Fig. 2. Event declarations

And it is not clear in which direction the values will be propagated. The execution engine of ECNO, however, guarantees that all elements participating in an interaction have the same parameters for the same event – if two partners contribute inconsistent values, the interaction would not be possible. Likewise, the interaction would not be possible if some partner needs[4] a parameter, but there is no partner that provides it.

A minor extension to class diagrams are the GUI annotations. These annotations indicate which elements and events are relevant to the user – and actually can be triggered by the user. As the name GUI indicates, this is mostly relevant for the GUI part of execution engine for generating buttons and user dialogs. In our case, there will be only buttons (see Fig. 7 in Sect. 4). Actually, it will not always be the GUI that triggers events; events can be triggered also programmatically; to this end, some elements would be registered with some specific controllers.

2.2 Summary of concepts

Altogether, ECNO extends class diagrams by the explicit definition of *events* and in which way different elements need to participate when an event is executed. This is defined by *coordination diagrams*, which are an extension of class diagrams. The basic mechanism for defining these coordination requirements is annotating references with an *event* and a *quantification*. Each of these references defines a bilateral coordination. In combination, however, they might require that many different elements participate in an *interaction*: First, there might be different references for the same event, which require different other elements to participate. Second, the other elements that are required to participate might have references with annotations, which require further elements to participate; in this way, establishing a chain of required elements. Third, an event annotation with quantification ALL requires that all the elements at the other end of the respective links participate. As we will see later, there is a fourth possibility: the local behaviour of an element can require synchronization of two different events. This way, cooperation diagrams define the *global behaviour* of a system based on the *local behaviour* of its elements[5].

[4] We will see later in the Petri nets defining the local behaviour what that means.

[5] Harel and Marelly called the global behaviour inter-object behaviour and the local behaviour intra-object behaviour [12] – in a different setting though.

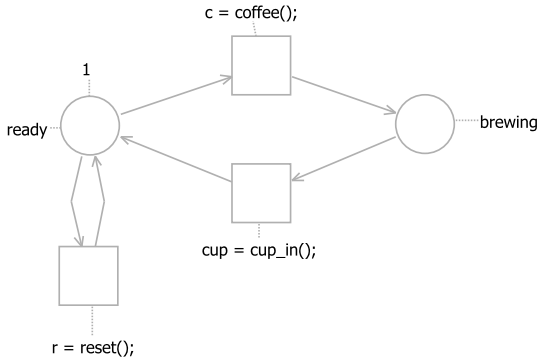

Fig. 3. Local behaviour of the coffee brewer

In the ECNO, there exist some more concepts (e. g. collective parameters of events) and we have some ideas of additional constructs, which are not discussed here (see [6] for a bit more information).

Figure 1 does not say anything about the local behaviour of the elements. The definition of the local behaviour will answer the following questions: when can an event be executed by an element, what is the local effect, and which (different) events need to be executed together (synchronized). ECNO provides an API for defining, or more precisely, for programming local behaviour for every element. But, in the context of this paper, we will use Petri nets for defining the local behaviour. These extended Petri nets will be discussed in the next section.

3 Modelling local behaviour with Petri nets

The *local behaviour* of an element defines when and under which (local) conditions an event or a combination of events can be executed locally, and it defines what happens locally when the event is executed as part of the global interaction.

3.1 Examples

Not surprisingly, this local behaviour can be defined by a slightly extended version of P/T-nets. We will discuss the main concepts by the help of some examples first. The concepts will then be clarified and explained in general in Sect. 3.2.

We start with a simple ECNO net that models the coffee brewer. It is shown in Fig. 3. Except for the annotations associated with the transitions, this is a conventional P/T-net. The transition annotations relate a transition of the Petri net to an[6] event; we call this annotation an *event binding*. After the event coffee, which represents the user pressing the coffee button, the coffee is brewed, which

[6] We will see later, that one transition can actually be bound to more than one event – this way enforcing the synchronization of different events.

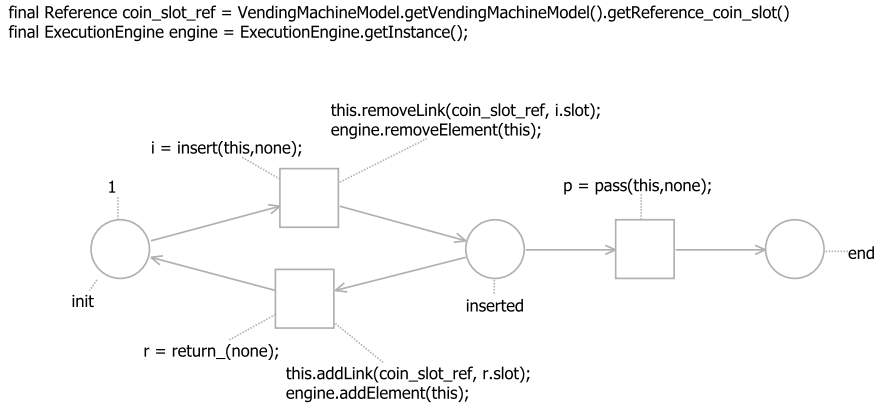

Fig. 4. Local behaviour of the coin

will be dispensed, when a cup is inserted (event cup_in). The reset event is possible only when the coffee machine is in the initial state (no coffee is being made). In this example, the notation for event bindings might appear a bit verbose, and it is not clear why, the event is assigned to some "variable". This will become clear in the next example, when referring to the events' parameters in conditions or actions. In the coffee brewer, these assignments do not have any particular meaning.

The behaviour of the coin is more interesting. It is shown in Fig. 4. First of all, the event bindings and the involved events have parameters. Let us consider the one with the insert event: as we have seen earlier, insert has two parameters, the coin and the slot. The annotation refers to these two parameters. The first one, this, assigns the coin itself as the first parameter (coin) to this event. The second parameter is none, which is the keyword indicating that in this instance, the coin does not assign a parameter to the event insert (it could do that in other instances though). The other annotation of this transition is the *action*, which will be executed when all partners of an interaction are found. In this case, the coin does two things: First, it deletes the link to the slot (since it is inserted now) and it also removes itself from the engine so as not to be visible at the GUI anymore. The first line removes the link, which is "addressed" by the coin_slot_ref attribute (which is a constant which will be discussed later). The slot from which it is removed is denoted by i.slot, where i is the variable to which the insert event was assigned, and slot is one of its parameters, which is assigned to the event by the slot. This is why we needed to give a name to an event in the event binding.

Once the coin is inserted, the Petri net for the coin allows two things to happen: either the coin can be passed to the slot by the transition that is bound to event pass, or the coin is returned by the transition bound to event return_. In the case of pass, the coin assigns itself (this) as the coin parameter; and there

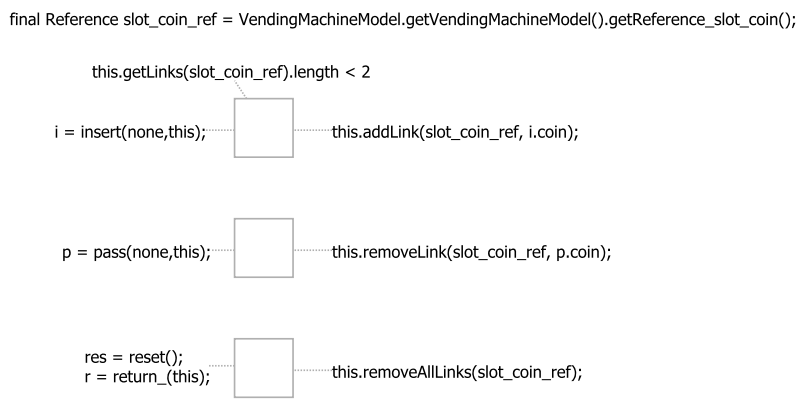

Fig. 5. Local behaviour of the slot

is no action. In the case of a return_, no parameter is assigned to the event (none), but the action will add a link from the coin to the slot again, where the slot is coming from the parameter r.slot of the event return_. Moreover, the coin registers itself with the engine again, so as to be visible in the GUI again.

Another extension that we see in this example is the declaration of attributes at the top of Fig. 4. These declarations follow the syntax of the Java language. Therefore, the additional keyword final actually defines a constant. In the declarations of Fig. 4, the first line uses the ECNO mechanism to get access to the reference feature from Coin to Slot in the model – this is very similar to the mechanisms of the Eclipse Modeling Framework (EMF) [10] and we do not go into details here. This feature is assigned to the attribute coin_slot_ref, which is then used in the two actions to add, resp. remove a link of that kind. The second line gets access to the execution engine of the ECNO framework (for adding and removing the coin from and to the GUI).

Figure 5 shows the local behaviour of the slot. This is a rather degenerated net. As a P/T-net, all transitions would be enabled all the time since their presets are empty. Due to the event bindings, however, the local behaviour becomes a bit more interesting. We start with explaining the bottom transition: This transition, actually, has two events bound to it: reset and return_. This implies that reset and return_ must be executed together; this way, all coins will be returned during a reset. The slot assigns itself as a parameter to the return_ event. Moreover, the slot deletes all the links to the coins it contains (i. e. it returns the coins).

The transition bound to the pass event is even simpler. When it happens, the link to the coin that is passed (accessible via the parameter p.coin) is removed (since it is removed from the slot and passed to the safe).

At last, let us discuss the top transition of Fig. 5. It is bound to the insert event, where the slot assigns itself to the event's slot parameter. In the action, it sets a link to that coin. This transition uses another concept: the *condition* which

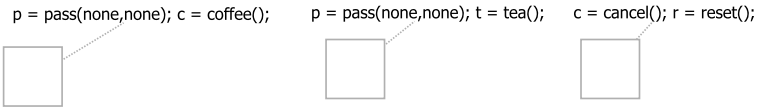

Fig. 6. Local behaviour of the control

is shown above the transition. This condition guarantees that an insert event can happen only when there are less than two coins in the slot. The condition refers to the attributes (the references to the coin) in this example, but it could in principle also refer to the parameters of the involved events.

The Petri net models for the other elements are similar. The last one that we discuss here is the one for the control. This net is shown in Fig. 6. The first transition guarantees that the event coffee (pressing the coffee button on the panel) must go together with an event pass (passing a coin from the slot to the safe). This way, it is indirectly guaranteed that there is a coin inserted. Therefore, the coffee button on the panel is enabled only when there is a coin inserted to the slot and when the coffee brewer is ready. The second transition does the same for event tea. The last event synchronizes the cancel event (which is triggered by pressing the cancel button on the panel) with the reset event: this way, it is indirectly guaranteed that all coins that are currently inserted to the slot are returned (see discussion of the local behaviour of the slot).

3.2 Concepts

As we have seen in Sect. 2.2, the main concepts of ECNO are the explicit definition of events and the coordination annotations of references that define how to coordinate the execution of events. At runtime, the combination of all the elements and events that meet these requirements will form an interaction.

The main concept for modelling the local behaviour of an element is to define when the element can participate in an event and what happens when the event is executed. In our Petri net notation, the event binding for transitions define when an event or a combination of some events is possible. To this end, the transitions of the Petri net refer to the respective events, and at the same time provide the parameter that this element would contribute to the event.

In our example, these parameters where expressions with values of the element (mostly this in our example). But, actually this can be more involved. For example, if we have some event event(Integer x, Integer y), it would be possible that an event binding looks as follows: e = event(none,e.x+1). The meaning of this is that the element takes the first parameter of the event increments it and assigns that value to the second parameter. If there were more events in the binding, we could also use a parameter of one event and assign it as a parameter to another event. Actually, there is no restriction in which way this could be done – if there are cyclic dependencies in these assignments in some interaction, the assignments will not be possible; the interactions will not be complete and,

therefore, will not be executable. It could also be that two partners would provide a value to the same parameter of the same event. The result will also be an illegal interaction, if the provided parameter are not the same (equal in Java terms).

On the one hand, this mechanism of parameter passing provides much expressive power and flexibility. Within the same interaction, data can be passed in opposite directions. On the other hand, such power requires great care in using this mechanism in order not to preclude desired executions by unintended cyclic dependencies. This, however, is a question of methodology and analysis functions that can check that no cycles of that nature would occur (which however are yet to be developed). The execution engine of ECNO copes with these situations – though computing all the parameters in complex situations might be quite computation intensive. Making this more efficient is ongoing research.

In the extended Petri nets, transitions have two more extensions: conditions and actions. Conditions are expressions that may refer to local attributes of the element (and whatever the element can access from there); and they may refer to the parameters of the events in which it participates. Conceptually, the condition is evaluated when all necessary parameters of the events are available. If, in a given combination of events, the condition evaluates to false, the complete interaction is not executable. Only if the conditions of all participating elements evaluate to true, the interaction is executable; its execution amounts to executing all the actions of all participating elements (in an arbitrary order). An action may refer to and access and change the local attributes and the parameters. The parameters or rather the objects they refer to can be changed – a parameter object itself cannot not be replaced, which follows the principle of parameters in the Java language.

3.3 Discussion

Altogether, the ECNO together with ECNO nets for the local behaviour allow to fully model the behaviour of a system. The generated code together with the ECNO runtime environment implement that behaviour (see Sect. 4).

Since the ECNO is still under development, there are still some issues open that need to be adjusted in order to strike a balance between usability of the language (expressing behaviour on a high level of abstraction) and performance. This is ongoing research and requires some larger case studies.

Technically, the ECNO nets define the local behaviour of the elements or parts of a system only. To some Petri net people, this might look a bit strange, since Petri nets are typically used to describe the overall behaviour or rather the global behaviour of a system. In fact, this is more a question on how these nets are used. The approach used here could be used for coordinating behaviour in a more global way: to do that, we would have one or more elements that are global and coordinate other elements by the mechanisms proposed here. That is why our approach is technically local, but the Petri nets could still be used for coordinating behaviour globally.

4 ECNO engine and generated code

Though the ECNO is still under development, the current prototype fully supports the concepts discussed in our example, and there is a code generator that fully automatically generates the code for the elements from the ECNO nets. An experimental ECNO runtime environment as well as the code generator for ECNO nets are deployed as an extension to Eclipse (Galileo or Helios) via the ePNK update site; You will find the information on how to install this experimental ECNO release, the example discussed in this paper, and some explanations on how to generate the code from ECNO nets at `http://www2.imm.dtu.dk/~eki/projects/ECNO/`.

In this paper, we give a brief overview of the resulting software and of how the generated code could be used. The generated code and the ECNO runtime environment can be run as a stand-alone Java application (under JRE 1.6 or higher). When the "VendingMachineInstance" is started as a Java application, a GUI will start, which looks like the ones in Fig. 7. The left one shows the GUI immediately after startup; the middle one shows the GUI after pressing the "insert" buttons on two coins (the last coin cannot be inserted anymore); the right one shows the situation after pressing the "coffee" button (note that since two coins were inserted and there are two independent coffee brewers, you could order the next coffee right away).

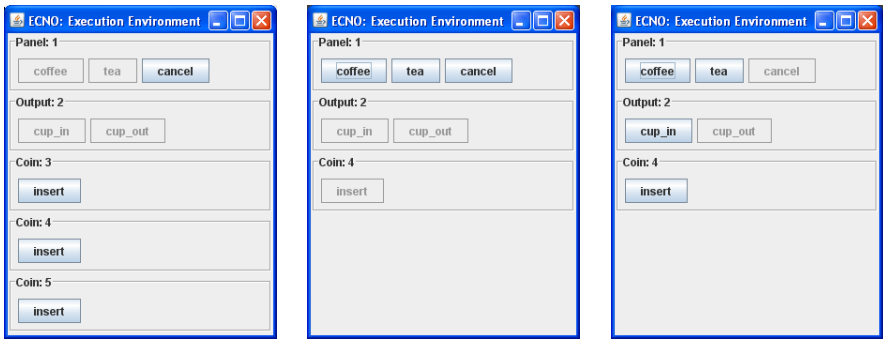

Fig. 7. Three screenshots: initial, after inserting two coins, after ordering a coffee

Note that a code generator for the coordination diagrams is not yet deployed. Therefore, the model from Fig. 1 is "programmed" in class "VendingMachineModel", in a style that is similar to the automatically generated Package and Factory classes for Ecore models of the Eclipse Modeling Framework (EMF) [10]. Based on that "programmed model", the class "VendingMachineInstance" sets up the concrete configuration of a coffee machine.

In this paper, we cannot discuss the details of the generated code since this would require a more detailed discussion of the ECNO runtime engine, its architecture, and the ECNO programming API (see [6] for some more details).

Basically, each element has a method that, for a given event, defines all the possible *choices*; in turn, a choice defines how to calculate the parameters for the events it participates in (which might depend on parameters of other events), a condition defining whether the choices is enabled, and a method that finally executes the action of the element. If you install the experimental prototype, there will be an example project which includes the generated code.

As motivated earlier, the generated code should integrate with other existing code, which could be legacy code, manually written code, or code generated from other models. This integration is possible along different lines: First, the generated code can be used by other parts of the software in the classical object oriented way[7] without taking its ECNO specific extensions into account – and without compromising the ECNO interactions. Note that, in order to be a bit more agile, the current prototype uses its own philosophy to generate the object oriented code from models. In principle however, ECNO could use any other technology for automatically generating code from class diagrams; once the prototype of ECNO stabilizes, we intend to use the EMF technology with its vast amount of related technologies and tools. Second, the code snippets in the ECNO nets (and in general any other implementations of the local behaviour) could use and invoke any other part of the software. Third, other parts of the software could use the API of the ECNO runtime to find out about enabled interactions and initiate their execution (see [6] for details). At last, existing software could be integrated with the ECNO by using adapter elements in the coordination diagrams, which delegate their behaviour to other parts of the software. Clearly, the first two "lines of integration" would be the most convenient ways; the last one is the most tedious one, since the adapters would require manual implementation – but it should not be too difficult.

5 Related work

The ideas for ECNO have developed over some years; they started out in the field of Business Process Modelling, where we used events and their synchronization for identifying and formalizing the basic concepts of business process models and their execution: AMFIBIA [13, 3]. It turned out that these ideas are much more general and do not only apply in the area of business processes. This generalisation resulted in MoDowA [4]. MoDowA, however, was tightly coupled to aspects, and event coordination was possible only for very specific types of relations. Therefore, the quest for distilling the basic coordination primitive was still on. In [2], we pointed out some first ideas for such an event coordination notation, which we call ECNO now.

Actually, none of the concepts of ECNO are particularly new or original. For example, Petri nets [7, 8] have been made exactly for the purpose of modelling

[7] This would require the ECNO engine to notify the controllers that are triggering possible interactions about changes in the possible interactions, which is not yet implemented.

behaviour. And many different mechanisms have been proposed for coordinating different Petri nets. For example, the box calculus and M-nets [14, 15] use synchronization of transitions, for coordinating different Petri nets. Also Renew [16, 17] uses executable Petri nets and has a concept of synchronous channels for synchronizing Petri nets.

The idea of events and the way they are synchronized dates back even further (in the earlier work, they would rather be called actions, which should however not be confused with our concept of action). ECNO's synchronization mechanism resembles process algebras like CSP [18], CSS [19], or the π-calculus [20]. But, we are a bit more explicit with whom to synchronize and with how many partners. Actually, there can be arbitrarily long chains or networks of required participants in our approach. One approach that allows to define such interactions (via connectors) is BIP [21]; but ECNO embeds a bit smoother with class diagrams and allows for and is tuned to dynamically changing structures. And it works together with classical programming and method invocation. Other parts of the software could call methods of our elements as they please; we just need to make sure that after such calls the possible interactions are updated. And, our actions can call methods of other parts of the software, and via an API other parts of the software can initiate and hook into interactions.

The ideas of the ECNO are an extension of our MoDowA approach (Modelling Domains with Aspects) [4], which has some similarities with the Theme approach [22]. In MoDowA, the interactions were restricted and implicitly defined in two special kinds of relations. Therefore, we introduced a separate concept and notation on top of references for making interactions explicit [6]; some of these ideas came up during the work on the master thesis [23]. Technically, ECNO is independent from aspect oriented modelling. Still, it was inspired by aspect orientation and is close in spirit to aspect oriented modelling (see [24, 25] for an overview) or subject orientation [26]. Moreover, ECNO could serve as an underlying technology for easily implementing aspect oriented models. In a way, events are join points and the interactions are point cuts as, for example, in AspectJ [5] in aspect oriented programming. There are two main differences though: in aspect oriented programming, the join points are defined in the final program; in that sense, the join points are artifacts and not concepts of the domain model. By contrast, our events are concepts of the domain! The other difference is the more symmetric interpretation and participation in an interaction. The participants in an interaction are not only invoked by another element; they can actively contribute parameters, and even prevent an interaction from happening (if no other partners are available). A more subtle difference is that interactions in our approach are attached to objects (actually, we called them elements) and not to lines in a program, and interactions can only be defined along existing links between the elements. This is a restriction – but a deliberate one: it provides more structure and avoids clutter and unexpected interactions between completely unrelated elements.

One of the main objectives of ECNO and coordination diagrams is that the coordination of events should relate to the structural models (a UML class

diagram). The rationale is the smooth extension of used technologies and easing the integration of behaviour models with structural ones and with pre-existing software.

To sum up, all the individual concepts of ECNO existed before – we did not invent them. The main contribution is the combination of these concepts, and this way, making them more usable in practical software development in general – and in model-based software engineering in particular.

6 Conclusion

In this paper, we have briefly discussed an event coordination notation, which we call ECNO. This notation allows defining global behaviour of a system by coordinating local behaviour: this global coordination is defined on top of structural diagrams. The focus of this paper is on a specific modelling notation for the local behaviour which we called ECNO nets (for a discussion of the ECNO engine and its API, we refer to [6]). From these models, the code for the complete system including its behaviour can be generated. The example shows that the coordination mechanisms of ECNO for defining global behaviour together with the mechanisms for defining local behaviour are powerful enough to completely define a system and generate code for it.

The most interesting research on ECNO, however, is yet to come: Scalability, performance, and adequateness of the modelling notation still need investigation; the constructs need to be adjusted, so as to strike a balance between these different objectives. The prototype implementation discussed in this paper, lays the foundation for these investigations.

Acknowledgements Since ECNO has developed over several years, many colleagues, students, and anonymous reviewers have contributed to what ECNO is now. They are too many to name them here; but, there is a half-way complete list on the ECNO Home page: `http://www2.imm.dtu.dk/~eki/projects/ECNO/`.

References

1. OMG: MDA guide. `http://www.omg.org/cgi-bin/doc?omg/03-06-01` (2003)
2. 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). (2010) 51–66 (Also published in the ACM electronic libraries).
3. Axenath, B., Kindler, E., Rubin, V.: AMFIBIA: A meta-model for the integration of business process modelling aspects. International Journal on Business Process Integration and Management **2**(2) (2007) 120–131
4. Kindler, E., Schmelter, D.: Aspect-oriented modelling from a different angle: Modelling domains with aspects. In: 12^{th} International Workshop on Aspect-Oriented Modeling. (2008)
5. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An overview of AspectJ. In: Proc. of *ECOOP 2001 – Object-Oriented Programming*, 15^{th} European Conference, Springer (2001) 327–353

6. Kindler, E.: Integrating behaviour in software models: An event coordination notation – concepts and prototype. In: 3rd Workshop on Third Workshop on Behavioural Modelling - Foundations and Application, Proceedings. (2011) Accepted.
7. Reisig, W.: Petri Nets. Volume 4 of EATCS Monographs on Theoretical Computer Science. Springer-Verlag (1985)
8. Murata, T.: Petri nets: Properties, analysis and applications. In: Proceedings of the IEEE. Volume 77. (1989) 541–580
9. Kindler, E.: ePNK: A generic PNML tool - users' and developers' guide : version 0.9.1. Tech. Rep. IMM-Tech. Rep.2011-03, DTU Informatics, Kgs. Lyngby, Denmark (2011)
10. Budinsky, F., Steinberg, D., Merks, E., Ellersick, R., Grose, T.J.: Eclipse Modeling Framework. 2nd edition edn. The Eclipse Series. Addison-Wesley (2006)
11. OMG: Meta Object Facility (MOF) specification, version 1.4.1. Tech. Rep. formal/05-05-05, The Object Management Group, Inc. (2005)
12. Harel, D., Marelly, R.: Come let's play: Scenario-based programming using LSCs and the Play-engine. Springer (2003)
13. Axenath, B., Kindler, E., Rubin, V.: An open and formalism independent metamodel for business processes. In Kindler, E., Nüttgens, M., eds.: Workshop on Business Process Reference Models 2005 (BPRM 2005), Satellite event of the third International Conference on Business Process Management. (2005) 45–59
14. Best, E., Devillers, R., Hall, F.: The box calculus: A new causal algebra with multi-label communication. In Rozenberg, G., ed.: Advances in Petri nets 1992. Volume 609 of LNCS. Springer-Verlag (1992) 21–69
15. Best, E., Fraczak, W., Hopkins, R.P., Klaudel, H., Pelz, E.: M-nets: An algebra of high-level Petri nets, with an application to the semantics of concurrent programming languages. Acta Inf. **35**(10) (1998) 813–857
16. Kummer, O.: A Petri net view on synchronous channels. Petri Net Newsletter **56** (1999) 7–11
17. 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. Volume 3099 of Lecture Notes in Computer Science., Springer (2004) 484–493
18. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall (1985)
19. Milner, R.: Communication and Concurrency. International Series in Computer Science. Prentice Hall (1989)
20. Milner, R., Parrow, J., Walker, D.: A calculus of mobile processes (Parts I & II). Information and Computation **100**(1) (1992) 1–40 & 41–77
21. Basu, A., Bozga, M., Sifakis, J.: Modeling heterogeneous real-time components in BIP. In: Software Enigneering and Formal Methods, Forth IEEE International Conference, IEEE Computer Society (2006) 3–12
22. Clarke, S., Baniassad, E.: Aspect-oriented analysis and design: The Theme approach. Addison-Wesley (2005)
23. Nowak, L.: Aspect-oriented modelling of behaviour – imlementation of an execution engine based on MoDowA. Master's thesis, DTU Informatics (2009)
24. Brichau, J., Haupt, M.: Survey of aspect-oriented languages and execution models. Tech. Rep. AOSD-Europe-VUB-01, AOSD-Europe (2005)
25. Chitchyan, R., Rashid, A., Sawyer, P., Garcia, A., Alarcon, M.P., Bakker, J., Tekinerdogan, B., Siobhán Clarke and, A.J.: Survey of aspect-oriented analysis and design approaches. Tech. Rep. AOSD-Europe-ULANC-9, AOSD-Europe (2005)
26. Harrison, W., Ossher, H.: Subject-oriented programming (a critique of pure objects). In: OOPSLA, ACM (1993) 411–428