Reasoning on Models Combining Objects and Aspects

Ella Roubtsova

Open University of the Netherlands
ella.roubtsova@ou.nl

Abstract. Modelling techniques are instruments for reality reflection. Precision of reality reflection demands coexistence of different abstraction types like objects and aspects in one model. Experiments with extension of modelling techniques aimed to accommodate combinations of objects and aspects in one specification have resulted in aspect-oriented extensions of many conventional modelling semantics. It was found that one of semantics called Protocol Modelling possess a very practical property of local reasoning on objects and aspects about behaviour of the whole model. In this paper the local reasoning property is defined in the reasoning logic and this property is demonstrated with a case study in the Protocol Modelling approach. Then the same case study is presented in aspect-oriented extensions of modelling approaches based on the semantics of contracts, sequence diagrams, workflows and state machines. The case study shows that the extensions of conventional semantics do not possess the local reasoning property. The semantic difference between Protocol Modelling and the listed modelling semantics is discussed and the useful semantic elements are recommended for new aspect-oriented languages and middleware.

Keywords: Local Reasoning, Aspects, Protocol Models, Contracts, Sequence Diagrams, Workflows, State Machines.

1 Introduction

Modelling abstractions were created to mirror systems and reflect the stepwise way of collecting domain knowledge during requirements engineering. Using objects for system decomposition is a very common practice and it is well known that separation of objects often causes crosscutting abstractions scattered through system specification.

In order to implement a crosscutting abstraction a modular unit called *aspect* was designed [7]. An aspect contains an *advice* in the form of a code presenting a concern and *pointcut designators* being the instructions on where, when and how to invoke the advice. The well defined places in the structure of a program or a model where an advice should be attached were named *join points*. Programming community had already accepted a join point model that used method calls as join points and inserted advice before, after or around a method call [7]. This join point model was implemented as various extensions of programming languages.

B. Shishkov (Ed.): BMSD 2011, LNBIP 109, pp. 1–18, 2012.
© Springer-Verlag Berlin Heidelberg 2012

Such extensions gave a new task to compilers: to produce the code with aspects woven in necessary places at the compilation time. This way an aspect is localized only at the design time. In the code it remains scattered through the code.

Another branch of aspect-oriented programming developed middleware for run-time aspect weaving without producing the code of the complete program. The weaving program in the middleware registers aspects and their pointcut designators. At run time the weaving program is intercepting the method invocations and inserting aspects before, after or around specified method invocations. The weaving programs implement sequential composition of method calls and returns of the base program and the method calls and returns of aspects.

However, it was found that the aspect-oriented programming techniques built on the existing aspect definition allow producing so-named invasive aspects [12]. Invasive aspects change the values of variables in other aspects and objects. In the case of invasive aspects no guarantee can be given about preserving behaviour of the base program after adding aspects and it is impossible to keep the reasoning control over the evolving program.

At this point the modelling community decided to investigate the problem and find the semantics that prevents constructing invasive aspects and guarantees safe modelling and system construction. The idea was to recommend such semantics for new aspect-oriented languages and weaving middleware.

The experiments were made in combining objects and aspects in different modelling techniques. These experiments have shown that practical use of modelling semantics combining different abstractions demands the convenient way to reason on models. The most attractive reasoning is the local reasoning on abstractions about behaviour of the whole system. Local reasoning makes the reasoning simple and allows building scalable models of systems and at the end the working systems.

The goal of this paper is to define local reasoning in reasoning logic and demonstrate its presence and absence in different modelling semantics. In the correspondence with the goal, section 2 reminds the reasoning logic and defines the local reasoning in this logic. Section 3 presents models of the same case study in the Protocol Modelling approach that possess the property of local reasoning and in other aspect-oriented modelling approaches that do not have such a property. The case study demonstrates the semantic elements that make the local reasoning impossible. Section 4 summarizes the semantic elements of Protocol Modelling that enable localization of reasoning.

2 Reasoning Logic and Local Reasoning

Let us consider a reasoning logic for state-transition systems: $S = (s_0, S, T)$, where s_0 is an initial state, S is a set of states and T is a set of transitions of type (s_i, s_j) and $s_0, s_i, s_j \in S$.

Let a state $s \in S$ be defined on a set of variables V used to store data, a set E variables used to temporally store events received from the environment and IE variables used to temporary store internal events generated inside the

system $S = V \cup E \cup IE$; $s = (v_1, ..., v_n, e_1, ..., e_m, ie_1, ...ie_k); n, m, k \in N$. (An event, an operation call or return can be stored in a data structure).

Let AP be a set of Atomic Propositions $\phi \in AP$ about values of variables of a system $V \cup E \cup IE$. The examples of atomic propositions are *"Amount = 2000"*, *"Event=Open"*, *"Password=Saved Password"*, etc.

A reasoning logic about a system is traditionally defined on a Kripke structure [1,21]: $\mathcal{M} = (M, R, \mu)$, where (M, R) is a reachability graph of this system. A node $m \subseteq M$ of a reachability graph is a state of the whole system. R is a set of relations on states giving possible transitions, and $\mu : M \rightarrow 2^{AP}$ is a function which assigns true values of propositions to each node of this reachability graph.

Function μ states that a predicate ψ is true in state s corresponding to node n of a reachability graph, iff and only iff the state described by the proposition is a sub-set of the set presented by the node of the reachability graph $\mu(\phi) \subseteq n$.

All variety of the reasoning statements is inductively defined as a set of predicates built from atomic propositions called state formulas $\phi \in TP$ about states in the reachability graph. We will analyse reasoning in different behaviour modelling semantics and to cover them all will use a superset CTL* of computational tree logic (CTL):

$$\psi ::= true \mid false \mid \phi \mid \neg\psi \mid \psi_1 \wedge \psi_2 \mid \psi_1 \vee \psi_2 \mid \psi_1 AU \psi_2 \mid \psi_1 EU \psi_2.$$

The interpretation of satisfaction relations in the reasoning logic has the reachability graph semantics:

1. predicate ϕ is satisfied in all nodes m of the reachability graph where predicate $\phi = true$.
2. predicate $\neg\psi$ is satisfied in all nodes where predicate $\psi = false$.
3. predicate $\psi_1 \vee \psi_2$ is satisfied in all nodes where ϕ_1 or ψ_2 is satisfied.
4. predicate $\psi_1 \wedge \psi_2$ is satisfied in all nodes where both ϕ_1 and ψ_2 are satisfied.
5. predicate $\psi_1 AU \psi_2$ is satisfied in node m if *for every path* $m_0, m_1, ...$ of the reachability graph starting from node $m = m_0$ there is node m_i such that for nodes $m_0, ..., m_{i-1}$ predicate ψ_1 is *true* and for m_i predicate ψ_2 is *true*.
6. predicate $\psi_1 EU \psi_2$ is satisfied in node m if *for some path* $m_0, m_1, ...$ of the reachability graph starting from node $m = m_0$ there is node m_i such that for node $m_0, ..., m_{i-1}$ predicate ψ_1 is *true* and for m_i predicate ψ_2 is *true*.

There are two groups of reasoning statements: state predicates and path predicates.

- State predicates can be formulated about one state, about a set of states and all states. The state predicates are expressed using variables. The two special forms of state variables are: STATE that presents the state from the semantic point of view and EVENT that express the fact that an event of a given data structure has been sent or received or an operation presented as a data structure has been called or returned.
- Path predicates can be formulated about states that follow each other in an existing path, about states that follow each other in a set of existing paths.

The reasoning structure presented before is applied both to the whole program or model and to an abstraction.

Definition 1. Local Reasoning Statement on an Abstraction. *A reasoning statement is local to an abstraction if it is formulated in terms of the states of this abstraction and events or operation calls (returns) accepted by the abstraction and describes the states or paths of the abstraction.*

Definition 2. Local Reasoning Property of a System of Abstractions. *A model/ system possesses the property of local reasoning if in any state any reasoning statement about the model/system is a conjunction of finite number of* local reasoning statements *of abstractions of this model/system and there are no other reasoning statements about this model/system.*

In another words, if a model/system possess the property of local reasoning, then the analysis of every system property is reduced to analysis of a conjunction of properties of a finite number of system abstractions.

3 Reasoning in Different Modelling Semantics

3.1 Case Study

The chosen case study is deliberately simple. It is designed to show the difference in composition of abstractions and the different reasoning possibilities in modelling semantics.

Let us consider a customer and a bank account. A customer can be registered. A registered customer can open an account and leave the bank. A customer can be frozen and released from freezing. The customer with the status "frozen" cannot leave the bank and open an account. An active account can be operated. An active account can be closed. An account can be also frozen and released. If an account is frozen then closing, depositing and withdrawing are impossible.

3.2 Protocol Modelling - Modelling with Local Reasoning

To date only one aspect-oriented modelling semantics, namely Protocol Modelling, has proven the possession of property of local reasoning [14]. Let us present the case study in Protocol Modelling and show what local reasoning means in practice. Figure 1 shows the case study in Protocol Modelling semantics.

Structure. A protocol model of a system is a composition of protocol machines *Customer*, *Account*, *Freezing* and *Freeze Control*. Protocol machines are partial descriptions of behaviour classes. For example, the behaviour class *Account* is described by three protocol machines: *Account*, *Freezing* and *Freeze Control*. Behaviour of class *Customer* is described by three protocol machines *Customer*, *Freezing* and *Freeze Control*. *Freezing* and *Freeze Control* are aspects woven into both behaviour classes *Account* and *Customer*. In order to generate own

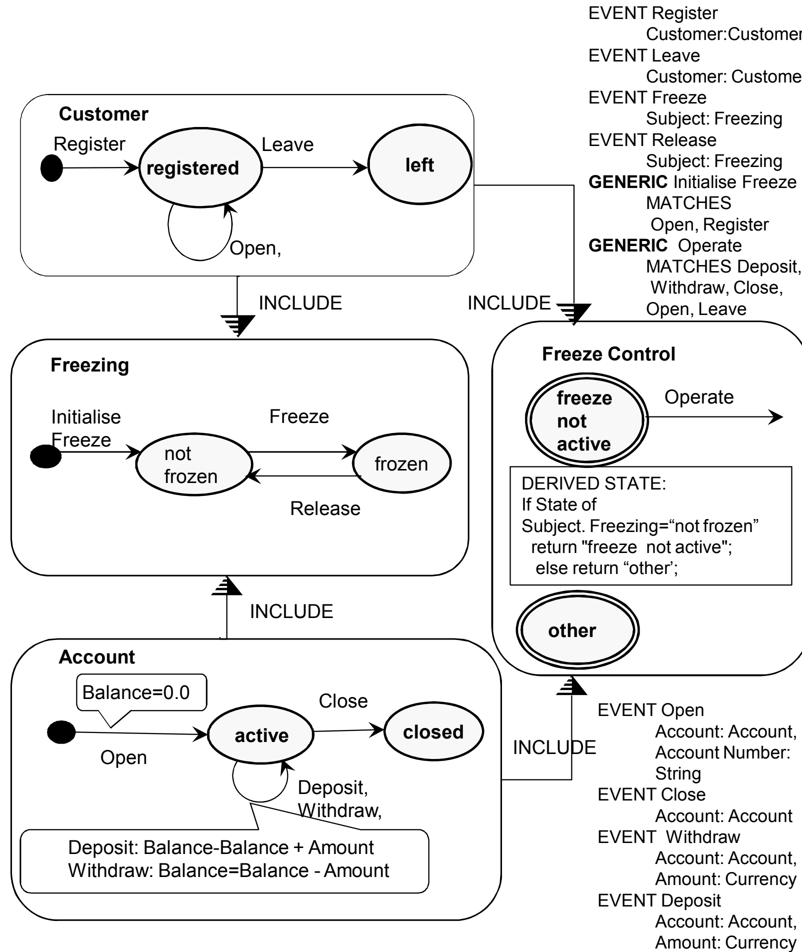

Fig. 1. Protocol Model

instances of *Freezing* and *Freeze Control* for each object of different behaviour classes, behaviours *Freezing* and *Freeze Control* are included into those objects. The INCLUDE-relation, depicted as a half-dashed triangle, gives to Protocol Models the expressiveness of multiple inheritance. Each object has its object identifier and behaviours of aspects are instantiated with instantiation of objects.

Events. A protocol machine has its own alphabet of recognized events. Event types *Open, Close, Deposit, Withdraw, Register, Leave, Freeze, Release* are presented as data structures in Figure 1. An event instance contains values of the attributes. Event alphabets of protocol machines can have a not empty intersection.

For example, the intersection of alphabets of protocol machines *Account* and *Customer* is event *Open.* It is used for synchronization of an instance of a *Customer* and an instance of an *Account.*

State. A protocol machine has its local state. The intersection of local states of protocol machines is always empty. The local state of a protocol machine is presented as a set of attributes and special enumerated attribute STATE. For example, the protocol machine *Account* has attributes *Account Number* and *Balance* and the attribute STATE with values $@new|open|closed$.

Behaviour Semantics of Event Refusal. A protocol machine presents a system that communicates with its environment. Events are presented to the model by the environment. Being in a suitable state, a protocol machine accepts the presented event, otherwise it *refuses* the event. The states accepting an *event* have an outgoing arc labeled by this event. A transition is depicted as an arc connecting two states *(state, event, state).* The behaviour of a protocol machine is a set of sequences of accepted events. The sequences of transitions of a protocol machine can be combined into a computation tree and their properties can be described with path predicates.

CSP Parallel Composition. Behaviour of a protocol model is a composition of behaviours of its protocol machines. The composition operator for protocol machines is a variant of the parallel composition operator defined by Hoare [9] in his process algebra Communication of Sequential Processes (CSP). This operator was extended by McNeile and Simons [16] for machines and events with data. Protocol machines use the CSP parallel composition algorithm to form more complex protocol machines. This is the description of the CSP parallel composition algorithm:

- A protocol model handles one event at a time and reaches a well defined quiescent state before handling the next event;
- If all machines of the protocol model, having an event in their alphabet, accept the event, the protocol model accepts it;
- If at least one of protocol machines, having this event in its alphabet, refuses the event, the composition of machines refuses it.

Derived States. A protocol machine can have a state function to derive its states from states of other protocol machines. Derived states are states that are calculated from the state of other protocol machines. For example, the protocol machine *Freeze Control* derives its state from the state of machine *Freezing* using its state function. The state function associated with the protocol machine results in state *'freeze not active'* or state *'other'* (Figure 1).

The derived states should not be topologically connected with other states. The arc of *Freeze Control* labeled with *Operate* does not need the right-end node. The arc means that *Freeze Control* accepts event *Operate.* The output state is

defined by the transitions of stored state protocol machines labeled with event *Operate* or events matching with it.

Protocol Modelling distinguishes protocol machines with derived state from protocol machines with stored states in order to simplify modelling.

Protocol Machines with derived state can be seen as spectative aspects. They observe the state of other protocol machines, calculate state from them and allow or forbid some traces of the system [17].

Execution and Reasoning. The Protocol Model is directly executed in the Modelscope tool [15] that provides a generic interface for execution. All possible events are visible at any step of the execution. All states may be made visible during the execution. As any state variables and any attribute is local to a protocol machine, there is a local reasoning predicate about every state change. Let us go through a sequence of model execution and reasoning.

1. For a new *Customers* the only available event is *Register*.
   The reasoning is local to the object *Customer*:

   $$((Customer = @new)EU(Exists\,Until)(Event = Register)).$$

2. If event *Register* takes place, then *Customer* transits into state *'registered'* and instances of two protocol machines *Freezing* and *Freeze Control* are created for the *Customer*.
   Freezing is instantiated in the state *'not frozen'* and *Freeze Control* derives its state 'freeze not active' from *Freezing* .
   Reasoning statements are local to *Customer* and *Freezing*:

   $$((Event = Register)EU(Customer = registered)),$$

   $$((Freezing = @new)EU(Event = InitialiseFreeze)),$$
   $$((Event = InitialiseFreeze)EU(Freezing = not\,frozen)),$$
   $$State\,Function:\ FreezeControl(Freezing = not\,frozen) = freeze\,not\,active,$$
   $$Generic:\ Initialize\,Freeze\,MATCHES\,Register.$$

   After application of the State Function and substitution of the Generic each of three reasoning statements as well as the conjunction of these local reasoning statements is the true reasoning statement about the behaviour of the whole model at this step.
3. Next, event *Customer.Freeze* becomes possible thanks to the *Freezing* aspect:

   $$((Freezing = notfrozen)EU(Event = Freeze)).$$

4. Than event *Open* becomes possible thanks to *Customer* and *Freeze Control*:

   $$((Customer = registered)EU(Event = Open));$$

   $$((FreezeControl = freeze\,not\,active)EU(Event = Operate));$$
   $$Generic:\ Operate\,MATCHES\,Open.$$

5. We also can reason that event *Register* for a chosen *Customer* is impossible because the local statements on *Customer*

$$((Customer = registered)\neg EU(not\, exists)(Event = Register)).$$

We can continue the execution and reasoning. Any state change can be explained by a conjunction of local reasoning statements on a limited number of abstractions. The composed model does not have states and paths properties of which cannot be described as a conjunction of local properties of a final number of composed protocol machines.

If a large number of instances of abstractions is involved in a reasoning, then a Protocol Machine with a derived state is created. It derives its states from all instances and reasoning remains local to this Protocol Machine with the derived states. For example, if event *Leave* for *Customer* would be possible only if all corresponding *Accounts* are closed, then we would add a protocol machine *Close Control* with the derived state *'All Accounts of Customer are closed'* and allow acceptance of event *Leave* only in this state. Modelscope provides SELECT functions [15] to select instances and derive states from the selected instances of different abstractions.

Join Points. A join point in Protocol Modelling is a set of events that can be seen identical to each other at the abstraction level of a particular protocol machine. For example, the Freezing abstraction does not see the difference between events *Open* and *Register* and defines join point *GENERIC Initialise Freeze* that matches each of these events. The *Freeze Control* abstraction does not separate events *Deposit, Withdraw, Leave and Close* and defines *GENERIC Operate* that matches each of those events.

The proof presented in [14] shows that the CSP parallel composition of protocol machines guarantees preservation of ordering of traces of aspects and objects in the whole specification. This property is called *observational consistency* [6]. In combination with localization of state and the prohibition for protocol machines to change state of each other, the observational consistency guarantees the property of local reasoning of protocol models.

Small and deterministic protocol machines are verified or tested by direct execution. Any new functionality, even the crosscutting one, is localized in a new protocol machine and synchronized with existing protocol machines. New protocol machines cannot cause any damage to behaviour of other protocol machines except possible forbidding of some traces. But this is directly identified as the conjunction of the reasoning statements local to this new forbidding protocol machine and the local reasoning statements of protocol machines allowing this trace. For example, the conjunction of reasoning statements of the *Customer* and the *Freeze Control* in state 'other':

$$((Customer = registered)EU(Event = Open))AND$$
$$((FreezeControl = other)\neg EU(Event = Operate));$$
$$GENERIC:\ Operate\, MATCHES\, Open.$$

results is the forbidding statement

$$((FreezeControl = other)\neg EU(Event = Open)).$$

3.3 Visual Contract Language

In this section we model our case study in the contract-based semantics called Visual Contract language (VCL) [2]. VCL explores the declarative way of aspect specification based on composition of sets of operations, attributes, classes and packages. In the VCL specification (Figure 2) classes *Customer* and *Account* are defined as sets of attributes and operations. Operations are depicted as hexagons.

A contract for a class is a set of its operations. Each operation has a corresponding diagram which specifies the operation name, its input (?) and output (!), the pre-conditions in the left hand side field and the post-conditions in the right hand side field. For example, the input of operation *Open* is an *Account Number? String* and the output is the *a! Account.* The post-condition is *Balance=0 AND Account Number=Account Number.* The empty pre-condition field means that there are no restrictions on the values of variables for this operation. (If we define variable STATE for the *Account*, than the precondition will be *(STATE≠Closed).*

Classes can have relations. For example, *'Customer opens Account'.*

Aspects are specified as classes. Figure 2 shows aspect *Freezing.* The functionality of *Freeze Control* is specified inside *Freezing* as *Freeze Control* functionality does not contain any own operations.

Classes can be combined into packages. A class and a package may have invariants that specify the types, or relations that are not changed during the object life cycle. A package may have operations that are specified as join interfaces *(JI).* Packages can join on join interfaces. Figure 2 shows how aspect *Freezing* is woven into objects *Account* and *Customer.* Package *AccountJI1* contains join interface *Open* used for weaving operation *Account.Freezing.Initialise Freezing.* Package *AccountJI2* contains join interface *Withdraw, Deposit, Close.* Each of these operations is used for weaving of *Account.Freezing.Get State Freezing.*

There are some general features of the contract semantics used by VCL that should be mentioned.

- The units of behaviour in contracts are operations. An operation itself has a body that may change both the state of its own object and the state of other objects. Pre- and post-conditions are even able to specify the changes of variables of other objects. Such operations cannot be elements of local behaviour of any object or aspect.
- A contract does not define what happens with an operation call if the pre-condition is not satisfied [18]: "If a precondition is violated, the effect of the section of code becomes undefined and thus may or may not carry out its intended work." Operation calls are not refused. Usually an operation call is kept (somewhere in a stack) waiting for the preconditions to become true. The absence of the refuse semantics in contracts makes the CSP synchronization impossible.

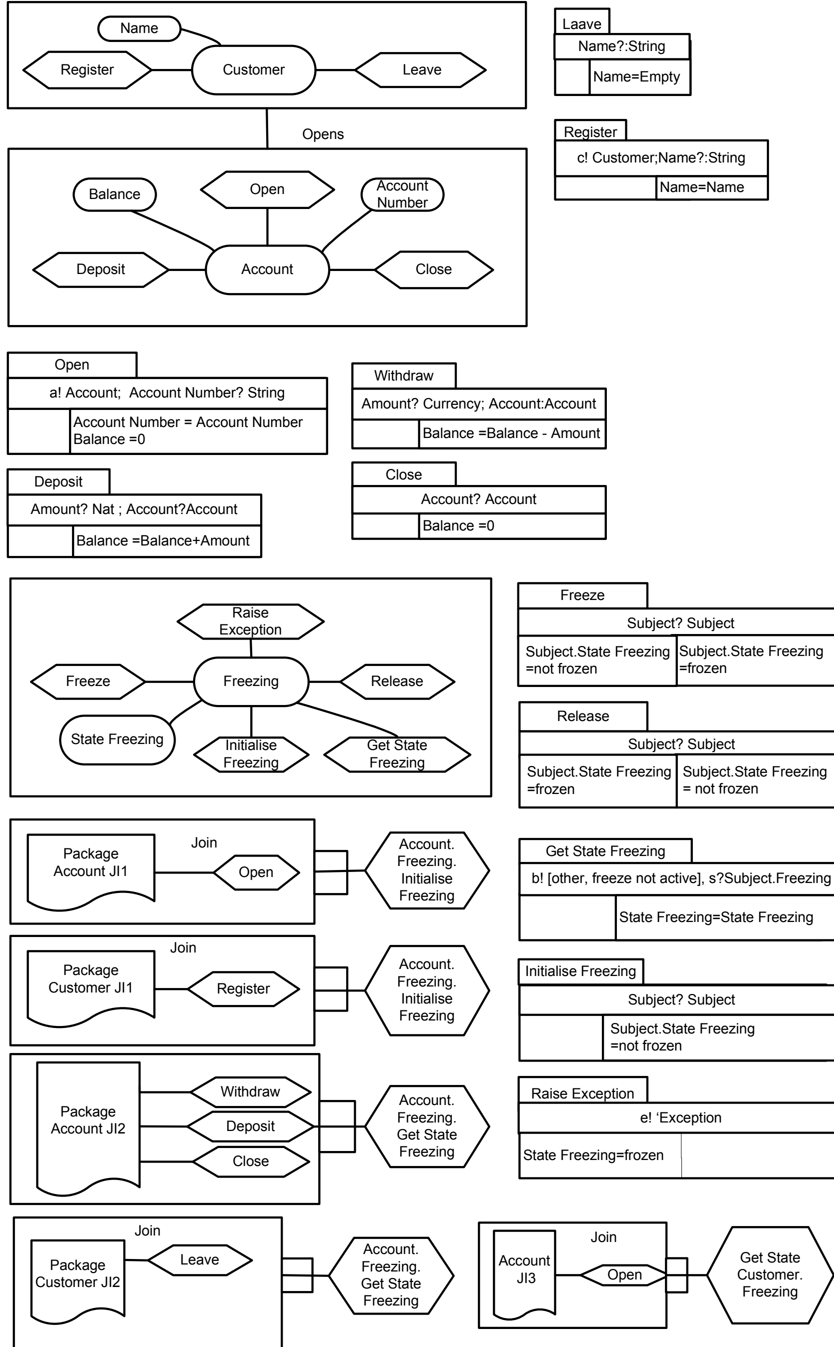

Fig. 2. VCL Model

From carefully specified contacts, having the Z-semantics, it is possible to generate a computation graph that shows the behaviour of the system but only if events happen when they are expected to happen.

- Operations are called one after another even all of them have true preconditions. The operation calls form sequences that can be sequentially composed or inserted between an operation call and return. For example, after *Register(Customer Name)*, *Initialise.Customer.Freezing* can be called and then operation *Register* can proceed to the completion. This is the aspect-oriented 'around invoke' technique. Because of the sequential way of weaving, the behaviour of the whole model (and its computation graph) will always contain states that cannot be composed from the states of the abstractions. It is impossible to reason about such new states using reasoning statements defined on the states of abstractions.

  For example, if $((Customer.Freezing = other)\neg EU(Event = Open))$ and event *Open* is called, it will not be refused immediately as the state of *Customer.Freezing* has to be checked. There will be a state after *Customer.Open* before call *Customer.Get State Freezing* where

  $$((Customer.Freezing = other)\neg EU(Event = Open))\, is\, false.$$

- In order to check a state of another package, the abstraction has to call operation *Get State*. This technique does not allow abstraction A to have derived states corresponding to the states of abstraction B. During the time interval between the call of *Get State* and its return the state of abstraction B can be changed. Therefore, quantification on states and using derived states as join points is impossible.

We can now summarize, that three semantic elements: (1) using operations that can change the state of other objects, (2) absence of operation synchronization and (3) sequential composition of operations, - produce in the whole model extra states and paths that cannot be described as composition of states of local abstractions. The whole model needs global reasoning and complete reachability graph has to be analyzed using theorem proving techniques.

3.4 Sequence Diagrams with Joint Point Diagrams

A set of sequence diagrams with conventional semantics is aimed to present only a part of possible sequences of system behaviour. Sequence diagrams illustrate behaviour of programs and therefore use operation calls and returns as elements of behaviour. The composition techniques of sequence diagrams are restricted to sequential composition, alternatives, cycles and insertion of sequences of operations calls and returns.

For Aspect-Oriented Modelling (AOM) the conventional sequence diagrams were extended with Join Point Designation Diagrams (JPDDs) [23]. Conventional sequence diagrams usually specify sequences to the completion of a use case. Sequence diagrams with JPDDs specify sequences of base objects as well as fragments of sequences of repeated aspects and join points.

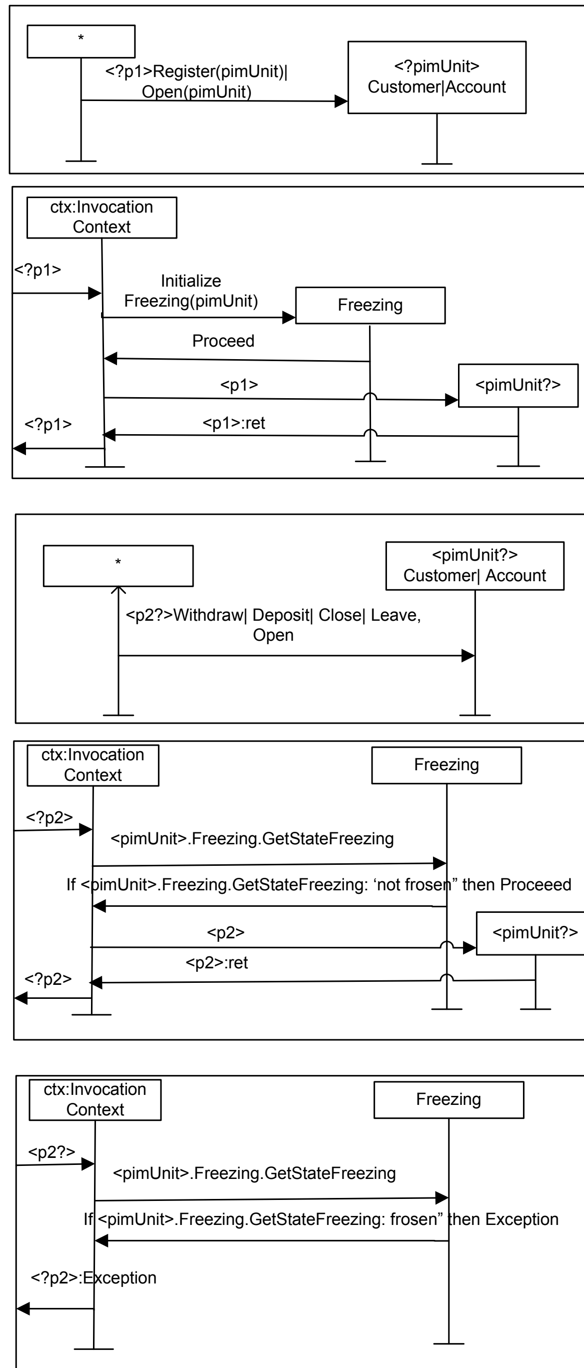

Fig. 3. JPDD diagrams

Figure 3 specifies JPDDs for our case study. The figure does not show the sequences of the base objects *Customer* and *Account* but specifies join points and advice fragments. JPDDs (1) and (3) are modelling means to graphically represent join point queries on *Customer* and *Account*. They use lists of operations that serve as join points. Diagram 2 presents the advice for initializing of aspect *Freezing* and diagrams 4 and 5 specify two different advice traces of the aspect *Freeze Control*.

In reality the set of sequences is infinite and sequence diagrams with JPDDs do not present the complete system behaviour to reason on it. However, even when all possible sequences are specified for a simple model then the sequences are combined into a computation graph using the same sequential composition technique as in contracts. Sequence diagrams use all three semantic elements that make local reasoning impossible.

Several approaches such as Theme [3], GrACE (Graph-based Adaptation, Configuration and Evolution [4], RAM (Reusable Aspect Models) [11] use JPDDs in combination with class diagrams. All approaches use global reasoning techniques [3,4,11].

3.5 Workflows as Aspect-Oriented Notations

Activity and workflow based approaches are aimed to specify complete system behaviour that can be analyzed and verified against required properties. The workflows are often used in AOM approaches as integration means to combine specified aspects. For example, the Theme approach [5] uses an activity diagram as an integration view. There are also AOM approaches that define fragments of workflows and compose these fragments into the complete workflow. An example is the approach called Activity moDel supOrting oRchestration Evolution (ADORE) [19].

Figure 4 renders our case study in ADORE. ADORE specifies a computation graph (a process) combining several basic abstractions. In our case it combines behaviours of *Customer* and *Account* in the workflow. Repeated partial behaviours of abstractions are specified as workflow fragments (or aspects). For example, behaviours *Freezing*, *Freeze Control* and *Leaving* are specified as fragments. Each fragment corresponds to a specific aspect and it is used as a partial point of view on its target. A fragment contains special activities, called predecessors P, successors S and hooks (assimilated as a Proceed in AspectJ). The hook predecessors *(P)* are the immediate predecessors of the first activity in the target block, and the hook successors *(S)* are the immediate successors of the last activity in the block.

The binding or weaving instructions assigning predecessors and successors are specified in a separate file. For example, the fragment $P3; S3$ of *Freeze Control* from Figure 4 can be bound as follows

$$P3 = i : Withdraw; S3 = r : Withdraw,$$

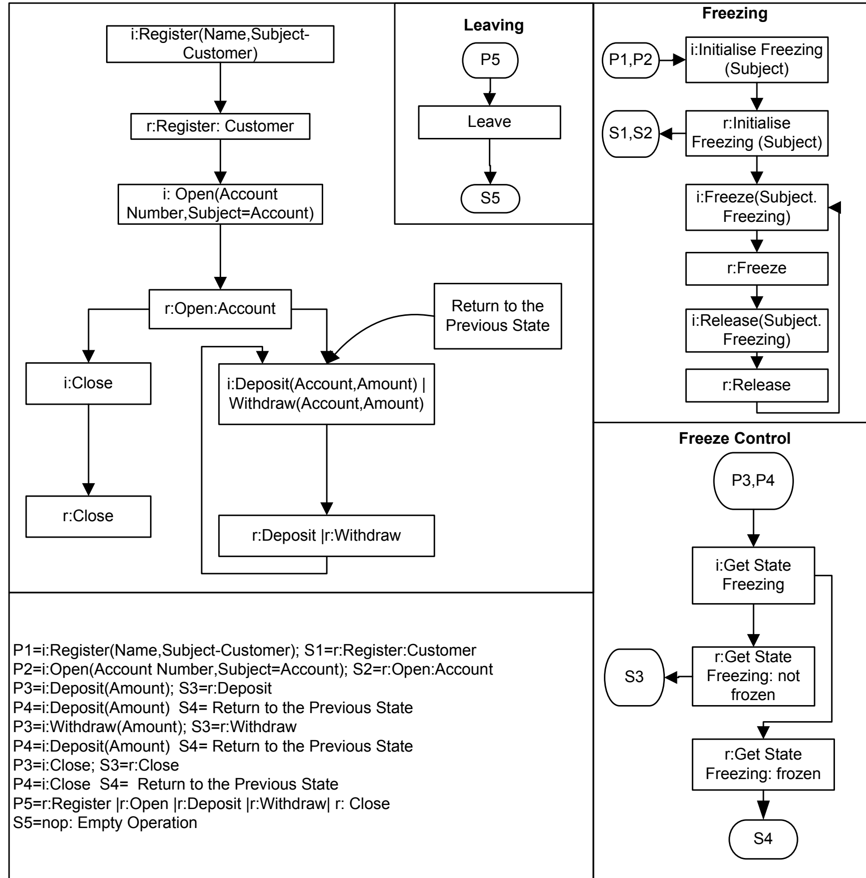

Fig. 4. ADORE Workflow diagram

where i is the invocation and r is the return of operation *Withdraw*. The hook is the sequence of activities

$$hook = i : Get\, State\, Freezing; r : GetStateFreazing = notfrozen.$$

The complete orchestration is generated from fragments according to the binding instructions.

As with such an approach there is no guarantee that local properties of aspects are propagated to the complete orchestration, the side effects of separating of concerns and composition in Adore are formulated as rules. The rule violation not always signals a mistake. It may indicate a "bad-smell", like, for example, the non-determinism caused by two conditions evaluated to false at the same time. The "bad-smells" are analyzed by the designer of the orchestration.

All workflow fragments are combined at design or runtime into a computation graph. This means that the behaviour composition technique is the same as in contract-based and sequences based notations.

It is possible to synchronize operation calls and returns in workflows using a synchronization construction. It is also possible to work on the level of events and do not separate calls and returns. However, one semantic feature makes this synchronization different from the CSP parallel composition used in Protocol Modelling. Namely, workflows do not have the semantics of event refusal. At any state several events may happen and the events are kept in bags or stack structures. The event may wait until the model transits to the state where this event is accepted. The computation graph of workflows depends on the state of those stacks or bags and the system has states that cannot be described as conjunction of states of system abstractions. Such a composition semantics does not leave any other possibility for reasoning than the global reachability analysis.

3.6 Aspect-Oriented Extension of State Machines

A UML Behaviour State Machines (BSM) [20] usually presents behaviour of one class. There are several approaches trying to extend BSM to enable several BSMs for one class. Mahoney et al. [13] suggested to exploit the *AND-composition* of several independent (orthogonal) statecharts defined by D.Harel [8]. "The key feature of orthogonal statecharts is that events from every composed statechart are broadcast to all others. Therefore an event can cause transitions in two or more orthogonal statecharts simultaneously" [13].

The ideas proposed by Mahoney et al. were further developed in the approach called High-Level Aspects (HiLA) [10]. HiLA modifies the semantics of BSM allowing classifiers to apply additional or alternative behaviour. Aspects extend the behaviour specified for classes.

The basic static structure usually contains one or more classes. Each base state machine is attached to one of these classes and specifies its behaviour. Figure 5 shows two state machines *Customer* and *Account* that look similar to protocol machines, but have different semantics.

- The first difference is in the semantics of labels on the arcs. A label of a protocol machine presents an *event* but a label of a state machine presents some state information before and after the event that run to completion: [*precondition*] *event*/ [*postcondition*][20].
- The second difference is the absence of event refusal. Events coming from the environment are kept in a queue or a stack of active events [20]. There are complex rules for handling or keeping events in the queue until the state is appropriate for their handling.

The HiLA approach does not change both mentioned semantic features but offers the patterns of aspect weaving. High-level aspects apply to state machines and specify additional or alternative behaviour to be executed at certain "appropriate" points in time of the base machines execution.

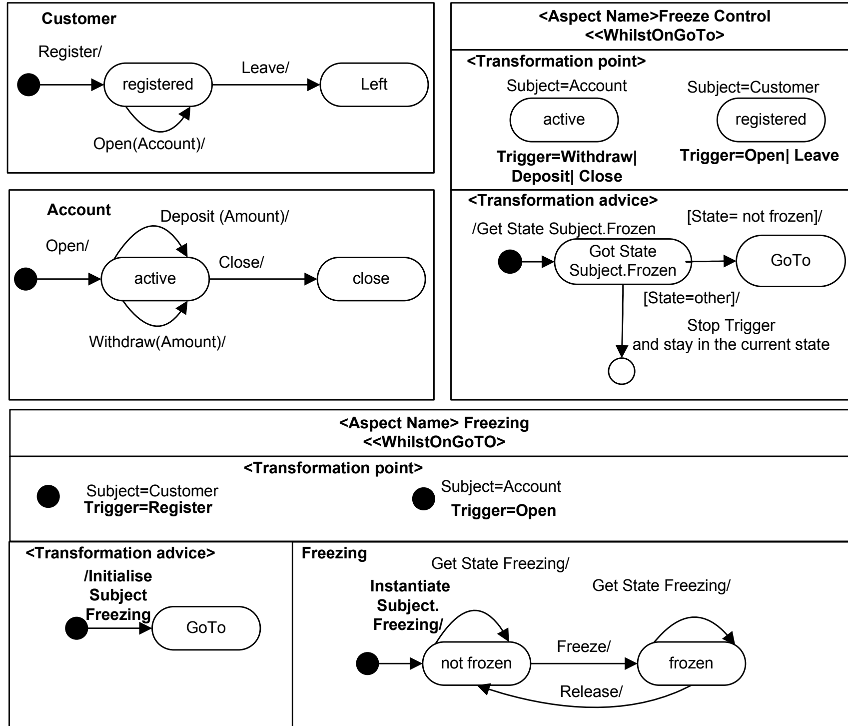

Fig. 5. HiLA State Machines

HiLA introduces patterns for specification of dynamic aspects. Any pattern has an *Aspect Name*, a *Pattern Type*, a *Transformation Point* and a *Transformation Advice*.

In Figure 5 we use pattern of type *"WhilstOnGoTo"* to specify aspects of our case study.

For example, the transformation point of aspect *Freezing* shows that the event *Initilise Freezing* happens in two situations (1) *Whilst Customer* is in the initial state, shown by the black dot, and when event *Register* takes place and becomes the trigger and (2) *Whilst Account* is in the initial state and event *Open* becomes the trigger. A pattern *Whilst* always has a specification of a state and an annotation $Trigger = e$. Conceptually it selects the compound transition from *State* with *Trigger e*, but if this transition does not exist, it is created [10,24]. This means that an aspect is added while an action is in the stack of active actions.

As the authors of the approach indicate [10,24], weaving of aspects into basis BSM results in another UML state machine which is analyzed using the model checking component of Hugo/RT model checking tools. Hugo/RT translates the state machine and the assertions into the input language of a back-end model checker SPIN. SPIN then is used to verify the given properties presented in Linear Temporal Logic for global analysis of the model behaviour.

4 Conclusion

This paper presents a survey of modelling semantics designed to accommodate aspects and objects in one model. The need of scalable models and reasoning control over complex models shows that accommodation of different abstraction types in one model demands more than just instructions on where, when and how to invoke advice of aspects. It is desired that modelling semantics possess the property of local reasoning on abstractions about behaviour of the whole model to reduce the analysis of any whole model property to the analysis of a finite number of local properties of model abstractions.

In this paper we have given the definition of local reasoning in the reasoning logic. We have applied the definition to show that the models built in many modelling semantics do not possess local reasoning property as they have extra states and paths that cannot be described as conjunction of states and paths of their abstractions.

Using the definition we have shown that Protocol Models possess local reasoning property. Other approaches may use semantic findings of Protocol Modelling as a notation independent basis for combining objects and aspects. The semantic elements that are needed for local reasoning are the following:

- considering events as instances of data structures but not as operations and this way avoiding state transformation defined inside operation bodies;
- using semantics of event refusal allowing synchronization of behaviour abstractions and the CSP parallel composition;
- handing one event at a time until the system stays in a quiescent state;
- allowing abstractions to transform their own state and read but not modify the state of other abstractions;
- allowing abstractions to derive their state from the state of other abstractions.
- mapping events to an alias if events are not differentiated at the level of a particular abstraction.

As further experiments show [22], with involving yet other abstractions into design, a combination of composition techniques might be necessary. The theory of system modelling has developed many composition techniques, but their combinations have not been investigated yet and have not been implemented in suitable modelling and programming tools. As the systems become more complex and use abstractions with different communication techniques, the practical modelling approaches, using combination of composition techniques and providing local reasoning when possible, are yet to be found and implemented.

References

1. Alur, R., Courcoubetis, C., Dill, D.: Model-checking in dense real-time. Information and Computation 104(1), 2–34 (1993)
2. Amálio, N., Kelsen, P.: VCL, a Visual Language for Modelling Software Systems Formally. In: Goel, A.K., Jamnik, M., Narayanan, N.H. (eds.) Diagrams 2010. LNCS, vol. 6170, pp. 282–284. Springer, Heidelberg (2010)

3. Baniassad, E., Clarke, S.: Theme: An Approach for Aspect-Oriented Analysis and Design. In: Proceedings of the 26th International Conference on Software Engineering, ICSE 2004, pp. 158–167. IEEE (2004)
4. Ciraci, S., Havinga, W.K., Akşit, M., Bockisch, C.M., van den Broek, P.M.: A Graph-Based Aspect Interference Detection Approach for UML-Based Aspect-Oriented Models. Technical Report TR-CTIT-09-39, Enschede (September 2009)
5. Clarke, S., Baniassad, E.: Aspect-Oriented Analysis and Design: The Theme Approach. Addison Wesley (2005)
6. Ebert, J., Engels, G.: Observable or invocable behaviour-you have to choose. Technical report. Universität Koblenz, Koblenz, Germany (1994)
7. Filman, R., Elrad, T., Clarke, S., Akşit, M.: Aspect-Oriented Software Development. Addison-Wesley (2004)
8. Harel, D., Gery, E.: Executable Object Modelling with Statecharts. IEEE Computer 30(7), 31–42 (1997)
9. Hoare, C.: Communicating Sequential Processes. Prentice-Hall International (1985)
10. Hölzl, M.M., Knapp, A., Zhang, G.: Modeling the Car Crash Crisis Management System Using HiLA. T. Aspect-Oriented Software Development 7, 234–271 (2010)
11. Kienzle, J., Al Abed, W., Klein, J.: Aspect-oriented Multi-view Modeling. In: Proceedings of the International Conference on Aspect-Oriented Software Development, AOSD 2009, Charlottesville, Virginia, USA, pp. 87–98 (2009)
12. Katz, S.: Aspect Categories and Classes of Temporal Properties. In: Rashid, A., Aksit, M. (eds.) Transactions on AOSD I. LNCS, vol. 3880, pp. 106–134. Springer, Heidelberg (2006)
13. Mahoney, M., Bader, A., Elrad, T., Aldawud, O.: Using Aspects to Abstract and Modularize Statecharts. In: The 5th Aspect-Oriented Modeling Workshop in Conjunction with UML 2004 (2004)
14. McNeile, A., Roubtsova, E.: CSP parallel composition of aspect models. In: AOM 2008: Proceedings of the 2008 AOSD Workshop on Aspect-Oriented Modeling, pp. 13–18 (2008)
15. McNeile, A., Simons, N.: `http://www.metamaxim.com/`
16. McNeile, A., Simons, N.: State Machines as Mixins. Journal of Object Technology 2(6), 85–101 (2003)
17. McNeile, A., Simons, N.: Protocol Modelling. A Modelling Approach that Supports Reusable Behavioural Abstractions. Software and System Modeling 5(1), 91–107 (2006)
18. Meyer, B.: Object-Oriented Software Construction. Prentice Hall (1997)
19. Mosser, S., Blay-Fornarino, M., France, R.: Workflow Design Using Fragment Composition - Crisis Management System Design through ADORE. T. Aspect-Oriented Software Development 7, 200–233 (2010)
20. OMG. Unified Modeling Language: Superstructure version 2.1.1 formal/2007-02-03 (2003)
21. Pnueli, A.: The temporal logic of programs. In: Proc. 18th IEEE Symp. Foundations of Computer Csience (FOCS 1977), Providence, RI, USA, pp. 46–57 (1977)
22. Roubtsova, E., McNeile, A.: Abstractions, Composition and Reasoning. In: AOM 2009: Proceedings of the 13th Workshop on Aspect-Oriented Modeling, Charlottesville, Virginia, USA (2009)
23. Stein, D., Hanenberg, S., Unland, R.: Visualizing Join Point Selections Using Interaction-Based vs. State-Based Notations Exemplified With Help of Business Rules. In: EMISA 2005, pp. 94–107 (2005)
24. Zhang, G., Hölzl, M.: HiLA: High-Level Aspects for UML State Machines. In: Ghosh, S. (ed.) MODELS 2009. LNCS, vol. 6002, pp. 104–118. Springer, Heidelberg (2010)