MODEM MODAF Migration

Providing an ontological foundation

Chris Partridge
February 2011

Business Object Reference Ontology

Contents

Preface 3
Executive Summary 4
Real World Analysis Overview 6
  Background 6
  State Type Succession pattern – real world analysis 12
  Interaction Diagrams – real world analysis 20
  Summary 29
IDEAS Detailed Technical Analysis 30
  Introduction 30
  State Type Succession pattern –real world analysis 30
  Interaction pattern – real world analysis 49
  Summary 58
Appendices 61
  Appendix A – MODAF UML Behaviour Scope 62
  Appendix B – State diagram as a mathematical structure – example definition 67
  Appendix C – State machines as a formal structure in the UML Superstructure Specification 68

© 2011 BORO Solutions Limited. All Rights Reserved.

Preface

This report on the MODEM project is in three sections:

1. An executive summary that explains the motivation for the MODEM work.
2. An introduction to the real world analysis that was done as part of the MODEM work, which gives a deeper understanding of the ideas that underlie it and provides examples of their use.
3. A detailed technical IDEAS analysis explaining the IDEAS MODEM model.

Each of the sections builds upon the previous section and is aimed at a different audience. The first section is aimed at management who need to understand the basis for the MODEM work. The second section is aimed at users who need to understand the issues that the MODEM work raises without delving into the technical details of the IDEAS model. The third and final section provides the detailed IDEAS analysis for the technical experts.

© 2011 BORO Solutions Limited. All Rights Reserved.

Executive Summary

Coalition operations are going to be a feature in the defence landscape for the foreseeable future. Effective and efficient coalition operations require collaboration at all levels. The IDEAS Group is a multinational project that aims to significantly improve collaboration at the level of military enterprise architectures through the development of a data exchange format. The purpose is to allow seamless sharing of architectures between the partner nations regardless of which modelling tool or repository they use.

Prior to the start of the project, the partner nations used a variety of architectural frameworks, tools and repositories and this made sharing of architectures a difficult manual task. The group’s goal is to develop a standard foundation upon which each partner nation builds its architectural framework, which will enable seamless sharing.

The group recognised that its most difficult challenge was to develop a format that worked at the level of meaning; so that when deployed it gave assurance of a common understanding of the data exchanged. With this requirement in mind, they developed the tool and repository agnostic IDEAS Foundation and issued it in April 2009.

The next stage in the development is for the member nations to migrate their frameworks onto the IDEAS Foundation. The US has now completed the migration of their DODAF metamodel onto the IDEAS Foundation – producing DM2 (DODAF Metamodel 2) – and is consolidating the revised model. Sweden has initiated the MODEM (MODAF Ontological Data Exchange Model) migration project. Its goal is to migrate the MODAF metamodel (M3) onto an IDEAS Foundation. When this is complete, the US DODAF and MODEM architectural framework will have a common foundation.

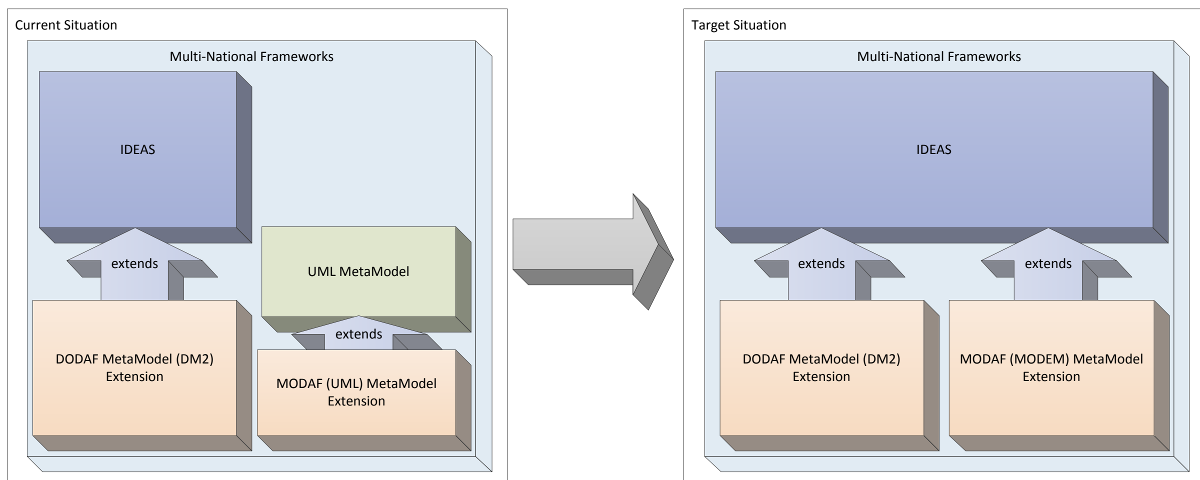


Figure 1 - Migrating to a common ontological framework

This will establish a foundation to take the nations to the next level of unification – a core for a unified defence enterprise architecture (EA) model. Clearly, there are many benefits to this. Each nation would have greatly reduced maintenance costs as framework development and change management would be shared between the nations. A single framework with a single metamodel also represents a more attractive market to EA tool vendors and so would increase choice, competition and quality.

© 2011 BORO Solutions Limited. All Rights Reserved.

BORO Solutions

Currently the MODAF metamodel is built upon the UML foundation. There are several issues with this. It limits the vendors that can easily support MODAF to UML vendors. Furthermore UML is not an ideal foundation for EA models. It originated from, and so was designed for, a technical programming environment rather than enterprise architecture. One result is that it has a number of technical features more suited to programming and systems development, some of which can cause EA interoperability problems. In addition, it was not designed to work at the level of meaning and so is unsuitable as a foundation in this project.

The migration from UML to MODEM will resolve these issues. NATO nations (and NC3A) have expressed very forcefully the need for a non-UML based NAF (NATO Architecture Framework) foundation. The migration will enable MODAF to meet the requirement from NATO to have a NAF model without UML dependencies.

There is a significant investment in MODAF, both directly in the MODAF metamodel and users' models and indirectly in the investment in UML. The MODEM migration aims to harvest and build upon this investment. The MODEM migration aims to:

- harvest the relevant features of UML and the MODAF metamodel and migrate them to MODEM,
- winnow out the irrelevant technical features – particularly the constraints that were stovepiping the UML metamodel and the MODAF metamodel built upon it,
- provide a clearer picture of the enterprise – one which reveals the common underlying business patterns across what previously appeared as very different areas, and
- provide a migration path for the existing MODAF models.

This enables the partners using MODAF to take advantage of the significant historic investment made in:

- the UML foundation without having to also take on the burden of irrelevant features and constraints,
- the UML-based MODAF model, while also providing access to the improved features of the new foundation, and
- the users' existing MODAF models.

And to do this while moving to a more flexible foundation that provides a basis for significantly improved collaboration at the level of military enterprise architectures through the seamless sharing of architectures between the partner nations regardless of which modelling tool or repository they use.

© 2011 BORO Solutions Limited. All Rights Reserved.

Real World Analysis Overview

Background

In the late 20th Century, enterprises grew to unprecedented levels of both size and complexity. It was recognised (early on by John Zachman) that this was creating a situation where the enterprise's architecture needed to be engineered rather than accidental. One of the challenges was devising engineering tools as none existed. John Zachman developed the first of these - his Zachman Framework – and the discipline of Enterprise Architecture was born. Since then a number of other frameworks have been developed and it is now recognised that a framework is an essential tool for enterprise architecture.

In the last few years, people have started to recognise that in order to develop a common understanding these frameworks need to be informed by ontology, particularly a top ontology. John Zachman has been vocal about this and started rethinking his framework in the light of this requirement. What ontology brings is a way of enabling the framework to present a clear picture of the enterprise – a real world semantics – where the enterprise models actually accurately reflect the real enterprise. And this clear picture provides a common reference point that is a solid basis for a common understanding.

Some years ago, the multi-national IDEAS Group realised the importance of ontology both for enterprise architecture frameworks in general and also specifically for enterprise architecture (EA) interoperability (a key requirement for coalition collaboration). They realised that if the coalition forces used the same top ontology and so shared a real world semantics, they could greatly simplify the exchange of enterprise architectures thereby improving collaboration and coordination. Hence, they devised and published a suitable top ontology – The IDEAS Foundation – to act as a common foundation for each nation's architectural framework.

The US has migrated their architectural framework - DODAF – onto the IDEAS Foundation, producing DM2 (DODAF Metamodel 2). SweAF has taken on the task of starting the migration of the MODAF Meta-Model to the IDEAS Foundation, with the goals of providing a stable baseline for MODAF and aligning with what has been done with DM2. This migration is shown graphically in Figure 2.

© 2011 BORO Solutions Limited. All Rights Reserved.

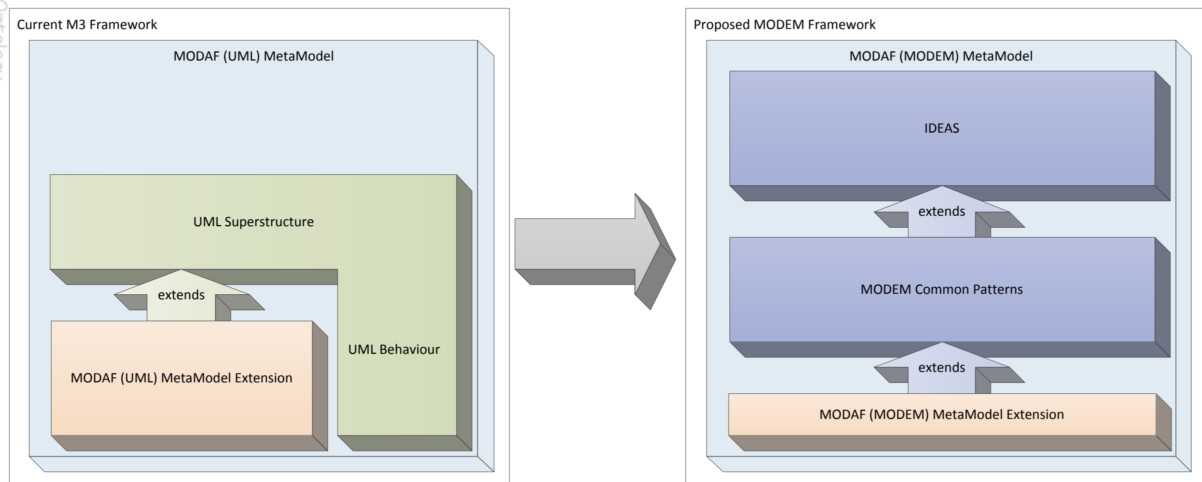


Figure 2 - Migrating MODAF to an ontological foundation

Real world semantics

As noted earlier, a key benefit of the migration is the provision of a real world semantics. From an EA perspective, the boxes and lines in their models need to have a clear connection to the real world – a real world semantics. If the EA model has boxes with the text 'cat' and 'mat' and a line joining the boxes with them with the text 'sat on the' – then there needs to be some confidence that people will interpret this as being about a cat sitting on a mat. There should be no worries that the users might interpret 'cat' as a dog or 'sat on the' as slept under – or that different users might interpret the model in different ways. In the case of simple examples using cats and mats, there is little reason for concern. But once the enterprise become bigger, there is a need for a semantic framework to bring a level of rigour.

In UML, as in many other modelling frameworks, the overarching framework was not designed with these semantic requirements in mind – though the examples often pay lip service it. The UML structures grew from a programming language base and are strongly influenced by this and the desire to be able to automatically generate program code from the models – which are not core requirements for EA. This lead to a focus on formal structure and a corresponding lack of focus on the real world semantics. Appendices B and C give a flavour of this. Appendix B contains a formal description of a state machine and Appendix C contains descriptions of the UML state machine. The result is often a framework that can hinder rather than help the uncovering of the real world semantics - there are a number of examples of this in the analysis section.

In modelling situations, such as EA in general and MODAF in particular, it is vital that the real world semantics are clear. Hence, an important aim of this project is to provide MODAF with a semantic framework –the IDEAS Foundation.

The introduction of real world semantics to MODAF will help to improve the semantic quality of the data exchange of enterprise architectures. It will lead to the removal of implementation constraints and so give a clearer real world semantics – as well as a more flexible structure. This in turn provides a better picture of the common underlying business patterns. Which leads to simpler, more accurate, models. Which leads to a better common understanding.

© 2011 BORO Solutions Limited. All Rights Reserved.

The Approach

Devising structures to support UML's specific goals created architectural pressures that hid the underlying real world semantics. A basic task of the analysis was to reconstruct the hidden real world semantics.

It used the BORO Method and this enabled the:

- harvesting of the relevant features of UML and migrate them to MODEM,
- winnowing out of the irrelevant technical features – particularly the constraints that were stove piping the UML metamodel,
- provision of a migration path for the existing MODAF models, and
- provision of clearer simpler picture of the enterprise – one which reveals the common underlying business patterns across what previously appeared as very different areas.

This enables the IDEAS partners to take advantage of the significant historic investment made in the UML and MODAF metamodels without also having to take on the burden of irrelevant features and constraints.

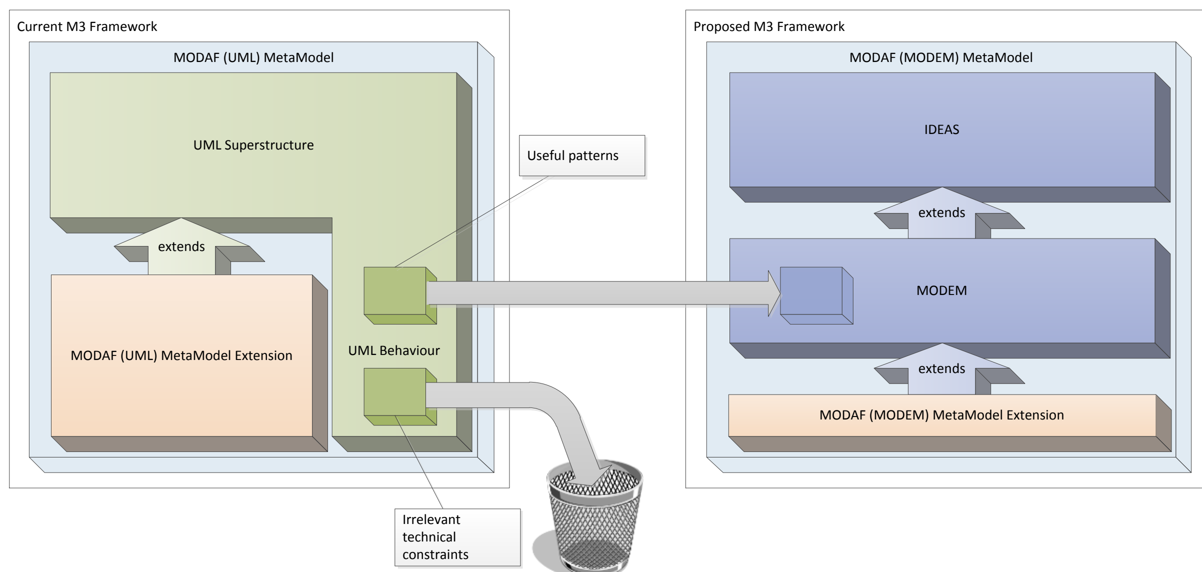


Figure 3 - Harvesting and winnowing the patterns

It also provides the vendors with the specification of a migration that would allow existing MODAF users to migrate to the MODEM Foundation and take advantage of the new functionality at little or no cost.

Furthermore this ensures that MODEM provides a truly EA tool-agnostic representation of MODAF. The elimination of the UML-dependency should help to increase the set of EA tool vendors that base their EA approach on a common metamodel – providing the IDEAS partners with greater choice.

Use of IDEAS Foundation

The analysis process used the IDEAS Foundation to reconstruct the missing real world semantics starting from the existing UML metamodel. In order to maximise the benefits of the IDEAS Foundation, great care was taken to ensure that the whole of the IDEAS

© 2011 BORO Solutions Limited. All Rights Reserved.

Foundation was used without any modification, and that where a pattern existed in the foundation it was used rather than re-invented at a lower level.

Also a clear distinction between the IDEAS Foundation and the MODEM extension was maintained. Where important common business patterns were discovered, these were marked as candidates to be promoted to the foundation in a future version of the foundation.

Scope of the report

Two aspects of the MODAF metamodel use native UML without any additional MODAF structure – state machines and interactions. In other words, no new stereotypes are defined in M3 for these areas. This presents a challenge to the migration project in that the semantics of those UML aspects must be fully analysed to identify the required functionality to be met by new IDEAS patterns for interactions and state transitions. Both these aspects fall within the UML behaviour model[1]. This report describes the migration of these two aspects.

This report describes the analysis for the behaviour section of M3 and UML and illustrates the harvesting, winnowing and simplifying that has taken place. There is significant scope for this in the behaviour section of the UML metamodel as its constraints lead to a particularly stovepiped architecture. Here are a couple of examples that highlight these issues.

The scope of UML behaviour can be traced to MODAF views. State Machine diagrams are used in OV6b and SV10b and Interaction diagrams in OV6c and SV10c, where OV deals with nodes and SV deals with resources. Appendix A contains an overview of these diagrams.

State Machine and Interaction diagrams can be viewed as complementary. Where one wants to look at the behaviour of a single object, state machines are used. Where one wants to look at behaviour across objects, interaction diagrams are used.

Process Detail

The analysis has clarified the main features of the real world semantics for ‘behaviour’ – the relationships between objects over time. It has mined the UML Behaviour model, stripping away the particular implementation decisions that UML made to reveal the underlying structure. The resulting patterns are not only a clearer picture of the real world but they also show the underlying simple straight-forward structures and so are easier for users to work with. Taking away the particular implementation constraints has resulted in a structure that gives a closer fit to EA user requirements and is more flexible than the original UML patterns.

The BORO analysis worked from the bottom up. It started by developing a clear picture of the individual objects that the UML structures were describing (the UML structures are several type layers above the individuals). Once this was established, it worked up the type layers. There is a detailed record of this analysis in the Worked Examples report. Given how

[1] They are specified in Chapters 14 ‘Interactions’ and 15 ‘State Machines’ in UML Superstructure Specification, v2.3.

© 2011 BORO Solutions Limited. All Rights Reserved.

key the formal structure is, extensive automated validation was performed on the MODEM model and examples.

It identified two core behaviour patterns that underlie the two UML diagrams:

- A pattern that deals with an object's state successions, which is handled by UML State Machines.
- A pattern that deals with the exchanges between the different objects participating in an interaction, which is handled by UML Interaction messages.

In UML, these two diagrams are in separate stovepipes with no overlap. The types of element in one diagram cannot appear in the other. One of the identified requirements was to break down this stovepipe and allow elements to appear in both diagrams. The analysis not only did this but also identified that the patterns associated with state machines are at the heart of the interaction diagram.

UML Interoperability Issues

UML's lack of a real world semantics (and so the benefits of moving to a MODEM Foundation) can be illustrated by the kind of semantic interoperability issues that are endemic within UML where different nations may take different views of the same domain.

UML State Machines and Interaction diagrams tend to assume that there is only a single view of the enterprise and so do not allow any other alternate views. However, in coalitions with multiple partners there are likely to be multiple views. The following examples illustrate the issue.

This example is shown graphically in Figure 4. In UML one can choose which states to include in a state machine. If Nation 1 chooses to include three states in a state machine and Nation 2 chooses to only include two of these states, then it is impossible in UML to combine the two models. There is no real world semantics reason why one should not be able to do this, so it is possible within MODEM.

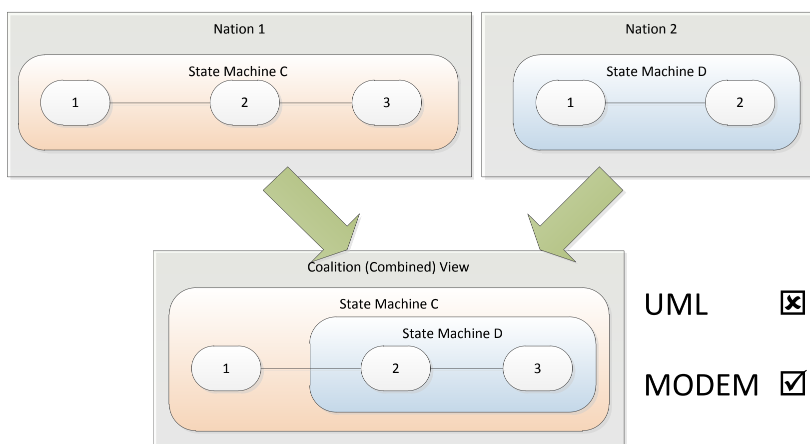


Figure 4 - Interoperability issues - partial views

This example is shown graphically in Figure 5. In UML one cannot inherit either states or state machines. But if Nation 1 chooses to include three states in a state machine and Nation 2 chooses model sub-states of two of these, this cannot be described directly in

© 2011 BORO Solutions Limited. All Rights Reserved.

UML. There is no real world semantics reason why one should not be able to do this, so it is possible within MODEM.

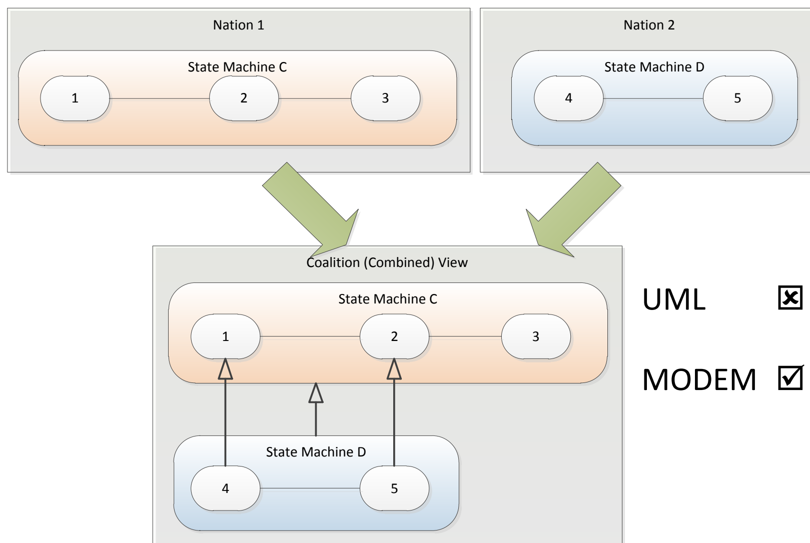


Figure 5 - Interoperability issues - inheritance

This example is shown graphically in Figure 6. In UML one can describe exactly the same orthogonal regions in a single state machine or multiple state machines. If Nation 1 describes the two regions in two state machines and Nation 2 describes them in one state machine, then it is impossible in UML to combine the two models. There is no real world semantics reason why one should not be able to do this, so it is possible within MODEM.

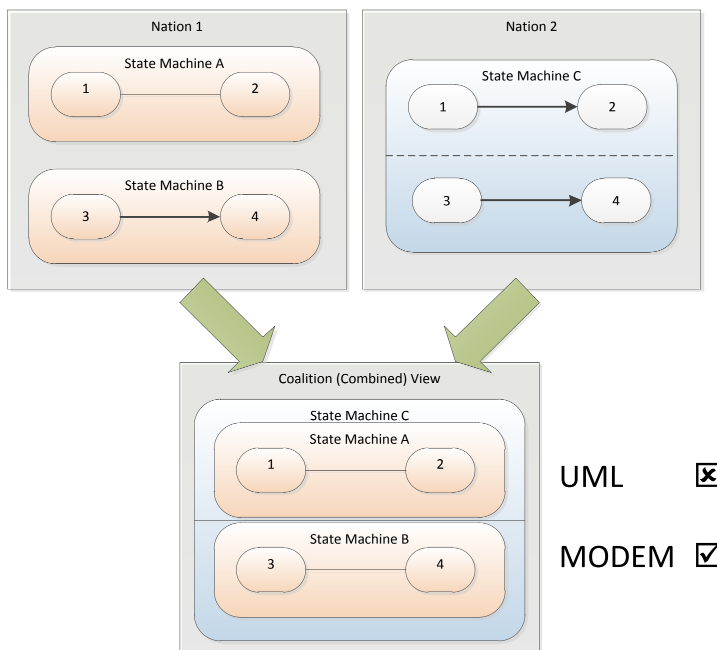


Figure 6 - Interoperability issue example - regions

While UML's constraints might make sense in a programming situation, they hinder interoperability between different nations' models.

© 2011 BORO Solutions Limited. All Rights Reserved.

State Type Succession pattern – real world analysis

This section focuses on the real world semantics for state machines.

UML State Machines - Examples

We use a number of examples to help guide the analysis and illustrate the results. The UML specification contains a number of examples. Figure 7 shows the one used as a basis for the analysis.

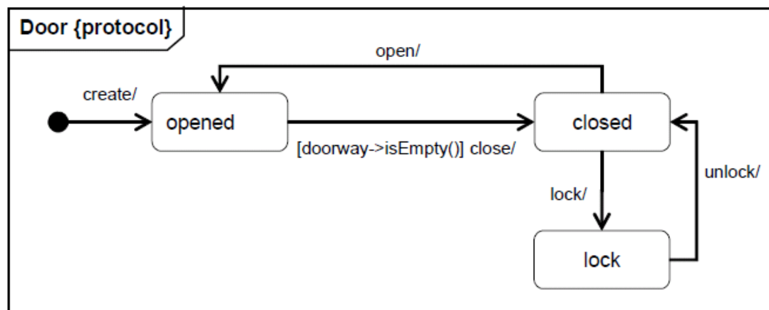


Figure 7 – "Figure 15.12 - Protocol state machine" (p. 552 - UML Superstructure Specification, v2.3)

The Figure 7 example is extended in Figure 8 to show multiple (concurrent) 'orthogonal regions' and 'hierarchically nested states'. 'Door Open-Closed-Locked' and 'Door Alarmed' are both 'orthogonal regions' of the Door State Machine. 'Alarm Level One' and Alarm Level Two' are sub-states of 'Door Alarmed' (in its submachine).

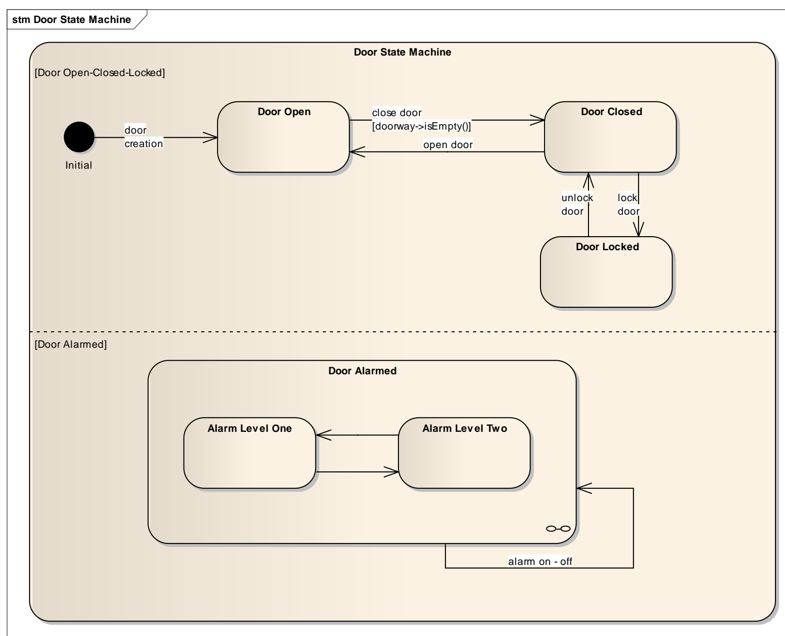


Figure 8 - Extended Example

Real World Semantics for a 'State Successions' pattern

The starting point for the analysis is the 'state succession pattern'. The first task in unbundling this is to establish from a real world semantics perspective, what a state is, what it is that has a succession.

© 2011 BORO Solutions Limited. All Rights Reserved.

A real world state

From the IDEAS perspective, this is well-established. A state of X is a temporal slice of X. For example, a door is opened and then closed. While it is open, the door is in a 'door open' state – this is a temporal slice of the whole four-dimensional extent of the door – as shown diagrammatically below.

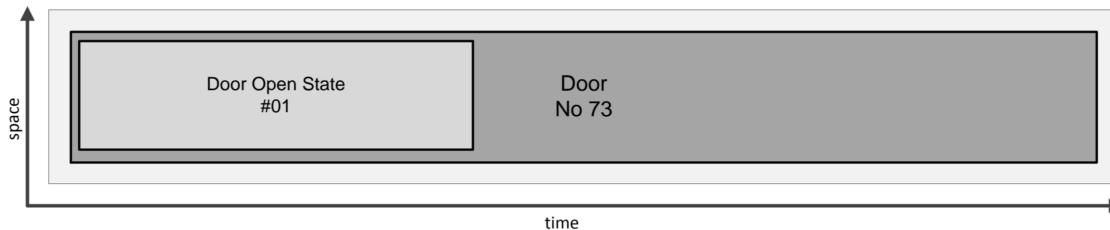


Figure 9 - A door open state space-time map

This slicing works much in the same way one would take a slice out of the middle of a long sausage. The object is sliced at its start boundary and end boundary – and everything in between is in the temporal slice. The limiting case is where one takes a slice off the beginning or the end – then only one real slice is needed, the other being notional.

Not every temporal part is a temporal slice. A simple example would be the fusion of two separates temporal slices. For example, as shown in Figure 10, a fusion of a door open and a door locked temporal slice is not itself a temporal slice. There are two indicators of this; firstly, one cannot mark out the state with a slice at the start and another at the end boundary – it needs four slices. Secondly, there is a temporal slice in its middle (marked in the diagram) that is not part of it but is part of the door. When we look at the succession pattern, it will become clear why this can cause a problem.

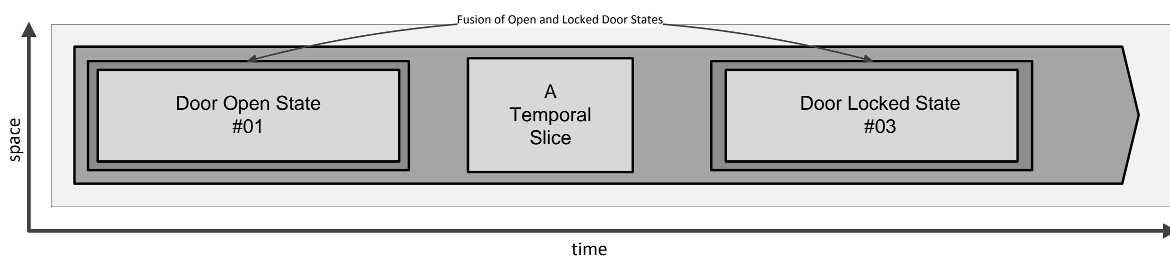


Figure 10 - Example of a non-slice temporal part

The example in Figure 10 might lead one to think that the state (temporal slice) must be connected; in other words one can draw a line from any one part to any other part without leaving the state. But it turns out that this requirement is too strong, as there can be perfectly valid scattered temporal slices, so long as it inherits its scattering from the thing it is a slice of.

To see this, consider the following example:

Manchester United and Wimbledon play a football match in two halves, with a short interval. It seems reasonable to assume that the interval is not part of the match. Then the football match is scattered, as it has two temporally disconnected halves. (The halves are not connected as one cannot draw a line though space and time from one half to the other

© 2011 BORO Solutions Limited. All Rights Reserved.

without leaving the extension of the football match – just as one cannot draw such a line on the space-time map in Figure 11.)

Assume that Manchester United played well for part of the match; that they started playing well after about 10 minutes from the start and stopped playing well about 15 minutes before the end. This gives us a 'Manchester United playing well' state of a football match, shown in Figure 11. It is a temporal slice of the football match, with a clear start and end slice but it, like the football match, is scattered – that is, it is not connected. However, because the slice inherits the scattering from the football match, it does not introduce a gap in the slice relative to the whole being sliced. So states can be scattered, so long as they inherit the scattering from the whole of which they are a state.

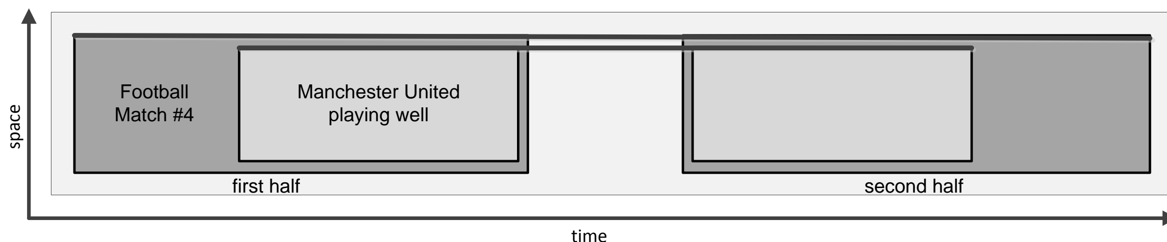


Figure 11 - A scattered state of a scattered football match

A real world state succession

However, central to the operations of a UML State Machine are the transitions between a set of (UML) states. From a state perspective, this is what we call a state succession. Consider a case where a door is opened, closed and then locked. There is a clear succession (transition) from a door open to a door closed and then to a door locked state – as shown below as a space-time map.

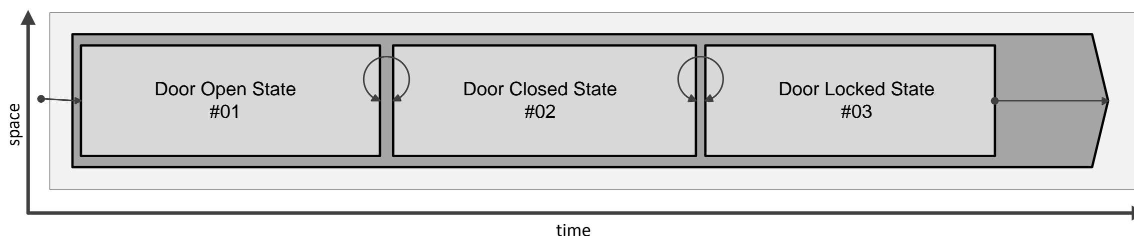


Figure 12 - Open-Closed-Locked Space-Time Map

One can see in the space-time map that the states form a chain or line with an initial state followed by a number of state successions (or transitions) and then a final state. (Arrows in the space-time map mark the initial and final states in the space-time map.)

The states do not have to immediately succeed one into the other, as in Figure 12. If we consider just the open and locked states, we get a succession that happens after a period of time – see Figure 13. This is valid and it is often useful to have different views.

© 2011 BORO Solutions Limited. All Rights Reserved.

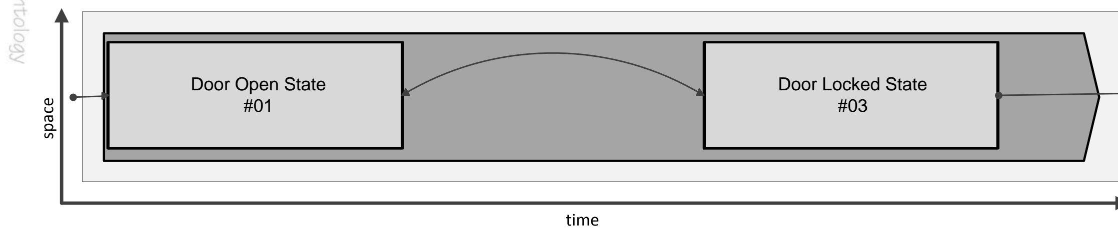


Figure 13 - Open-Locked Space-Time Map

Non-causal successions

As these examples show, the succession relation is not causal. The first half of a football match does not cause the second half. Nevertheless there is a dependency between the successions, it is logically impossible for there to be a second half of a football match without there being a first half – it is necessary that there is a first half before there can be a second half.

Set of Successions Relative to Set of States

The last two examples also illustrate the requirement (constraint) that the relevant successions are determined by the collections of states under consideration. So in Figure 13, which excludes the Door Closed State, the succession from Door Open #01 to Door Locked #03 is included. But in Figure 12, which includes the Door Closed State it is not. This shows how being a succession is relative to a set of states. Within that set of states, the succession picks out the next state in the collection – and which state is next depends upon which states are in the collection. This requirement excludes the succession from Door Open #01 to Door Locked #03 (shown in Figure 13) as it does not pick out the next state in the collection – even though both states at the ends of the succession are in the collection. Also, a succession can be associated with more than one set of states – the succession from Door Open #01 to Door Closed #02 is a succession in Figure 13 and Figure 14.

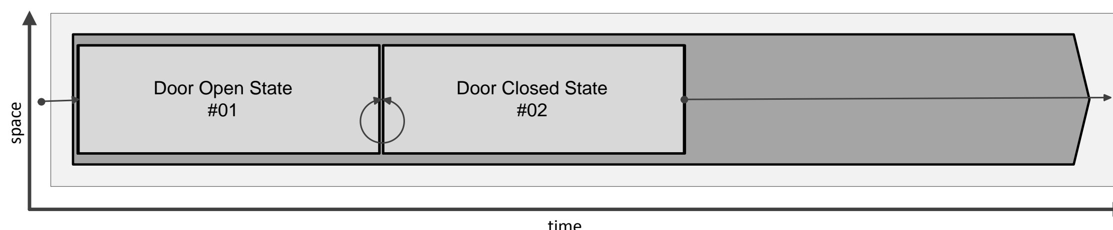


Figure 14 - Open-Closed Space-Time Map

This also shows that states and succession can be in many state succession views – something UML does not cater for.

Disjoint Set of States Requirement

Furthermore, not just any collection of states will have this pattern – the collection of states must be disjoint – in others words, they must not overlap. Otherwise, the successions will not 'work' because the end of one state is before the beginning of the next state.

For example, a door can be both open and alarmed – in other words, the 'door open state' and the 'door alarmed state' can overlap as shown in the space-time map in Figure 15. The

© 2011 BORO Solutions Limited. All Rights Reserved.

door alarmed state cannot succeed the door open state – as it has started before the door open state has ended. Hence the collection {Door Open State #01, Door Alarmed State #11} is not an example of the state succession pattern.

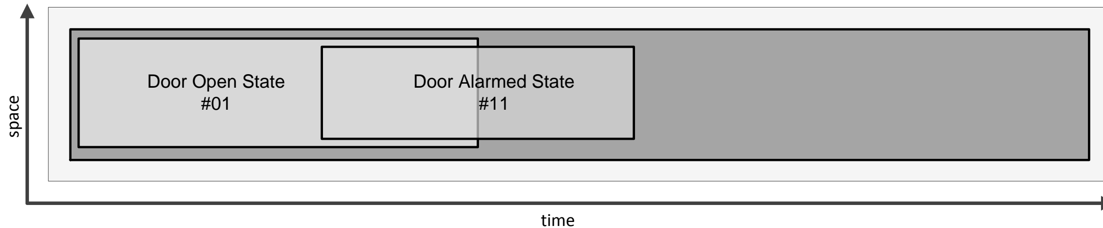


Figure 15 - Overlapping states

One can begin to see the reason why scattered temporal parts cannot be states (scattered relative to the things they are states of). If you look at the example in Figure 10 again, then the fusion of the door open with the door locked temporal slice, and the other temporal slice, are disjoint – but there is no way one can succeed the other, as they interleave each other. So both a non-relatively scattered state and disjointness are required to enable succession.

Real World Semantics for a 'State Type Successions' pattern

We have grounded this state succession pattern at the individual level for an individual door; we now need to take it up a level for doors in general. In the next stage we consider the state successions pattern in general, not just for doors. For this first step, we generalise to door state types rather than individual door states.

Consider doors in general and their open, closed and locked states. We can describe a pattern of successions between these states in a grid – shown in Figure 16.

PREVIOUS STATE	NEXT STATE			
	OPEN DOOR	CLOSE DOOR	LOCKED DOOR	final
initial	✓			
OPEN DOOR		✓		✓
CLOSE DOOR	✓		✓	
LOCKED DOOR		✓		
OPEN-CLOSED-LOCKED DOOR STATES SUCCESSION GRID				

Figure 16 - Example succession grid

The new feature at this level is the state types – we have divided the individual states into types. And we have three types of states of this thing (doors), whose instances are temporal stages of doors. However, for the successions to 'work', there are some additional constraints that need to be satisfied.

Instance-wise Disjoint Union of Set of States Requirement

At the grounding level, we have the constraint that the states of the whole must be disjoint. This translates into a requirement that the union of the three state sub-types (so the union

© 2011 BORO Solutions Limited. All Rights Reserved.

of open, closed and locked states) are disjoint relative to their whole. It turns out that a requirement that state sub-types are just altogether disjoint is too strong.

To see this consider this example. We have a prison door and it has a cell viewing door built into it (see Figure 17 below). Clearly the cell viewing door is an integral part of the prison door and both doors can be open and closed independently.

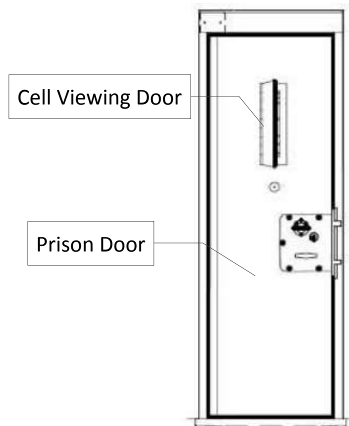


Figure 17 - An example of a door within a door

It is likely that the viewing door will be open when the prison door itself is open, closed or locked. This is shown diagrammatically Figure 18 for the case where the door is open and then closed. This means from a spatio-temporal perspective that the collection of all the door open-closed-locked states (their spatio-temporal extents) overlap, that they are not disjoint.

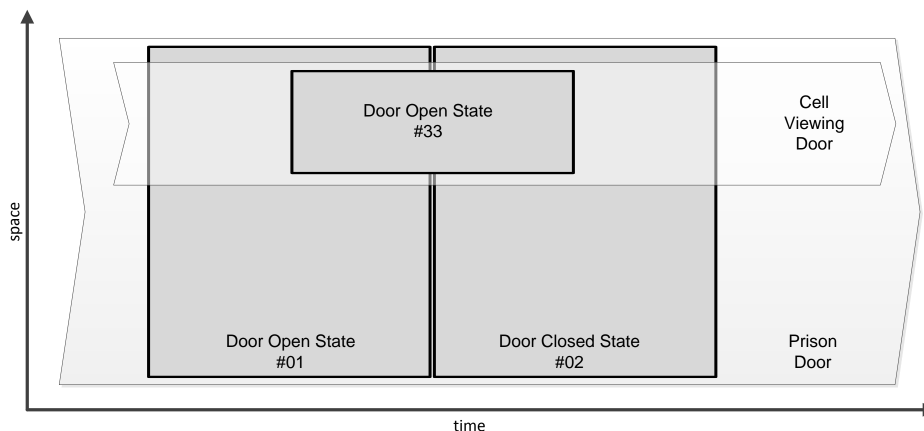


Figure 18 – (Non-Instance-wise) Door Overlapping States

If we made it a requirement that the states were disjoint, we would exclude cases where there is a clear state succession pattern. Instead, we work with the weaker requirement that for each door instance all its states in the collection must be disjoint. This requirement must be met to enable the successions to work. So, in this example, when we consider the prison door (as the 'owner' of the collection of its temporal stages), the cell viewing door states are not considered as they are not states of the prison door. So cases such as the prison door state succession patterns are included.

© 2011 BORO Solutions Limited. All Rights Reserved.

This kind of constraint can be difficult to spot as there is a natural assumption that all instances of everyday types (such as doors) are disjoint. However, a little reflection can soon provide one with many counter-examples.

Disjoint Set of State Types

We need to add one further constraint. Consider a case where we have chosen the two state types: Open Door and Unlocked Door (where this is the union of the Open Door and Closed Door states). This does not violate the 'instance-wise disjoint union of set of states' requirement described above as at the individual level, the union of the state sub-types are instance-wise disjoint. However, it does not exhibit the state succession pattern – it does not make sense to talk of an Open Door state transitioning into an Unlocked Door state as it is already in an Unlocked State. The underlying reason is that at the state type level, the state types are not disjoint, they share members – as shown in Figure 19.

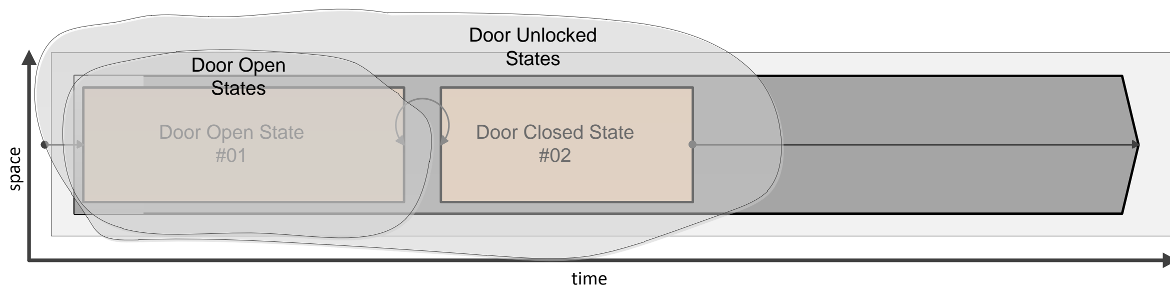


Figure 19 – Non-instance-wise Disjoint Types Space-Time Map

This is easy to see in the Venn diagram format in Figure 20.

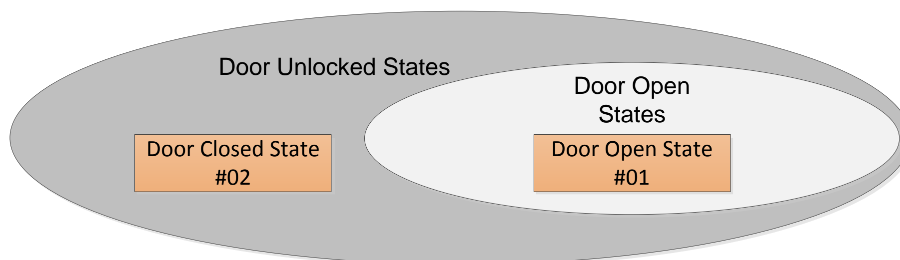


Figure 20 - Non-instance-wise Disjoint Types Venn Diagram

This shows the need for an additional constraint; that the state types chosen must be disjoint – they must have no members in common. It also clearly illustrates that the constraint is a property of the collection of states, rather than the individual states.

Real World Semantics for the general 'DisjointStateTypesSets' pattern

The door's 'state type succession' pattern is one example of the general 'DisjointStateTypesSets' pattern. In general, something is a 'DisjointStateTypesSets' if:

1. It contains a disjoint set of types
2. The union of these types are all temporal stages of some sub-type of 'Individual' (an Individual sub-type) and
3. These temporal stages are instance-wise disjoint relative to the sub-type.

© 2011 BORO Solutions Limited. All Rights Reserved.

Individual sub-type's hierarchy of 'DisjointStateTypesSets'

There is no constraint on how many 'DisjointStateTypesSets' an Individual sub-type can have, provided they meet the criteria for 'DisjointStateTypesSets'. An Individual sub-type will typically have a hierarchy of 'DisjointStateTypesSets'. We can use the door example to illustrate this. The example contains the set {Door Open, Door Closed, Door Locked}. Consider the sub-sets of this:

- {Door Open, Door Closed},
- {Door Open, Door Locked},
- {Door Closed, Door Locked},
- {Door Open},
- {Door Closed}, and
- {Door Locked}.

Each of these is a 'DisjointStateTypesSets' in its own right – these are shown in the Venn diagram in Figure 21. From a pragmatic perspective, there may be situations where it is useful to filter out the states an audience is not interested in. This provides the structure to select for each audience the view that contains what they are interested in.

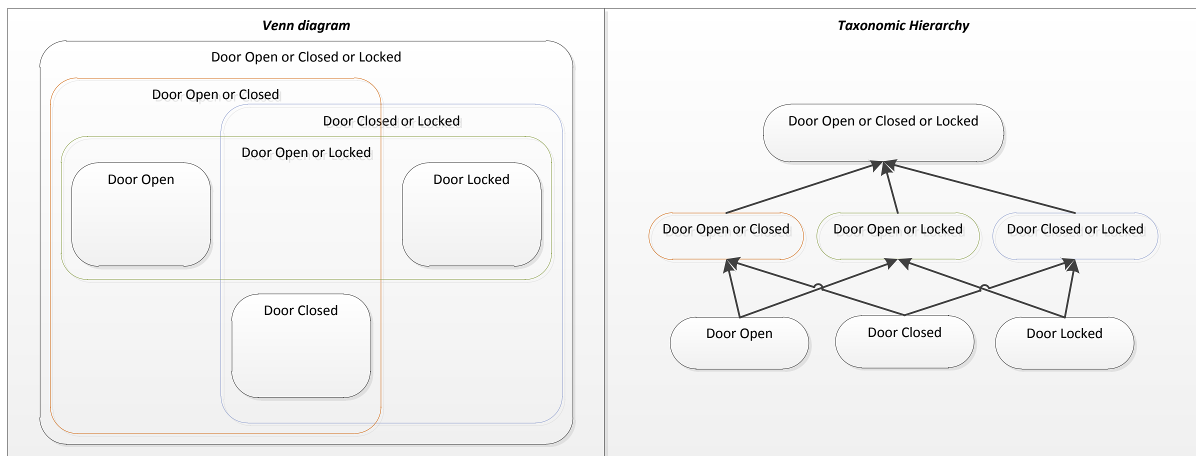


Figure 21 – Example hierarchy of 'DisjointStateTypesSets'

Multiple 'orthogonal 'DisjointStateTypesSets' pattern

In the example above, the hierarchy of 'DisjointStateTypesSets' shared members. It is common to include multiple orthogonal sets – orthogonal[2] in the sense that they do not share members (though they may share instances of their members). As noted earlier, the selected worked example (see Figure 8) contains an instance of multiple 'orthogonal' sets: AlarmedDoorStateTypeSet and OpenCloseLockDoorStateTypeSet.

[2] 'Orthogonal' is defined on p. 563 of the UML Specification in relation to regions as 'Description. A region is an orthogonal part of either a composite state or a state machine. It contains states and transitions.' Though some formal structure is defined elsewhere, there is no further description of the real world semantics for this.

© 2011 BORO Solutions Limited. All Rights Reserved.

As this case illustrates, the union of the two sets may not be a ‘DisjointStateTypesSets’ - Figure 15 shows a Door Open state can overlap a Door Alarmed state. Hence, the behaviour of an individual can be characterised by a number of different state machines. Picking on set of state types does not exclude the overlapping states from being participants in another state succession pattern.

Nested ‘DisjointStateTypesSets’ pattern

Given that any individual sub-type can have ‘DisjointStateTypesSets’, it follows that the state types in one set can be the owner of its own ‘DisjointStateTypesSets’. This is a common behavioural pattern. The worked example provides an instance of this. The AlarmedDoorStateTypeSet owns the Alarmed Door Level One - Level Two State Type Set as shown in state machine diagram format in Figure 8.

The multiple orthogonal ‘DisjointStateTypesSets’ and nested ‘DisjointStateTypesSets’ patterns are examples of structures that are inherited from UML to MODEM.

Inheriting the ‘State Successions’ pattern

As noted at the beginning of this section, different nations may decide to model their state machines at different levels of generalisation. When combined these lead to a requirement for inheritance of the state successions pattern. The requirement can exists with a MODAF model, where the state type succession in an OV6b is sometimes specialised in a SV10b, adding or removing structure. The analysis clarified that this specialisation is a super-sub-type relation between the state types – and shows how additional structure can be added. This is illustrated with this simple extension to the earlier door example.

Consider ‘Fridge Doors’, a sub-type of ‘Doors’ – and assume, as is often the case, that they cannot be locked – in other words, there is no Fridge Door Locked state. Clearly Fridge Doors and the set {Fridge Door Open, Fridge Door Closed} are just a specialisation (in some sense) of the earlier example and exhibit the state type succession pattern – as shown in Figure 22. This is an example of the inheritance pattern in Figure 5.

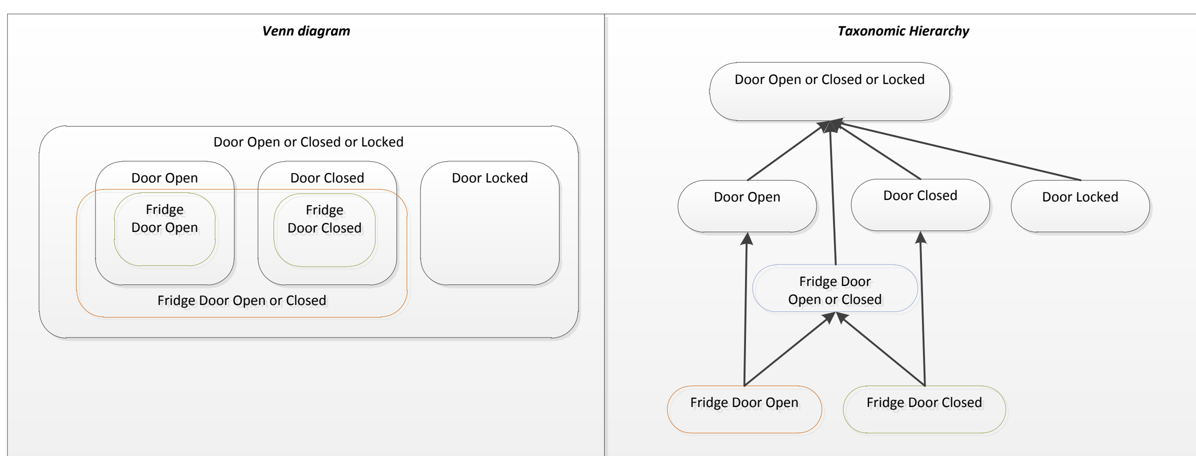


Figure 22 – Example inherited states

Interaction Diagrams – real world analysis

This section focuses on the real world semantics for interaction diagrams.

© 2011 BORO Solutions Limited. All Rights Reserved.

Chosen Example

The chosen interaction diagram example (Figure 23) looks at the role of people in an 'Eat Restaurant Meal' interaction. This Interaction diagram takes a particular view of the interaction through its choice of participants. One could easily take other views of this interaction by choosing different participants. For example, one could have a cutlery and crockery view or a food and wine view or a money view of this example interaction.

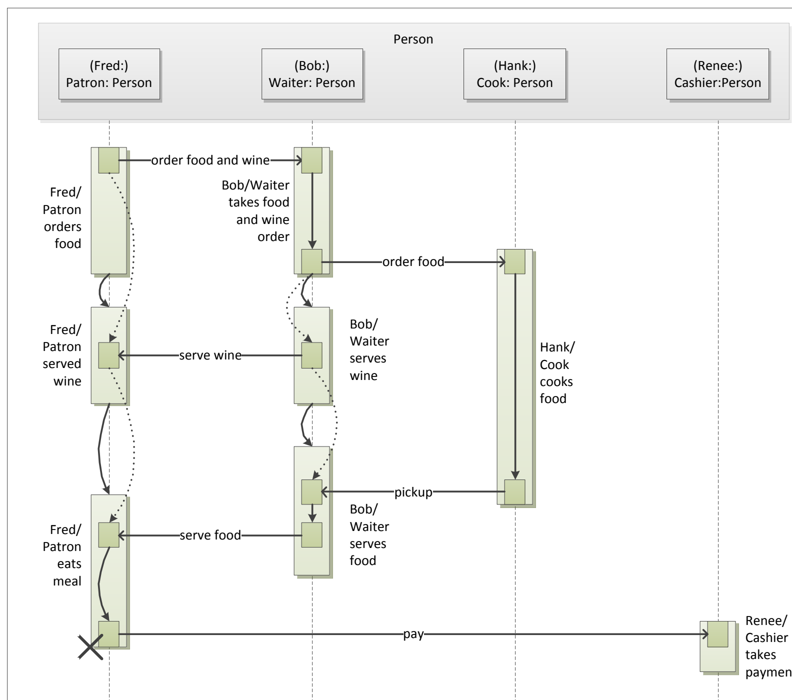


Figure 23 - Chosen Interaction Diagram Example

Two Key Aspects

There are two key aspects of an interaction diagram. Firstly, the interaction has as components a number of participations by other Individuals (called in MODEM 'Interaction Roles' and in UML 'lifelines'). Secondly, there are temporal ordering inter-dependencies between components of the interaction. These aspects can be seen in the chosen example.

The participation aspect

Figure 24 is a space-time map that shows the first aspect – the whole-part relationship between the individual interaction role participations and the overall interaction - for example, the 'Fred: Patron' interaction role is the participation by Fred in 'Fred Eat Restaurant Meal'.

© 2011 BORO Solutions Limited. All Rights Reserved.

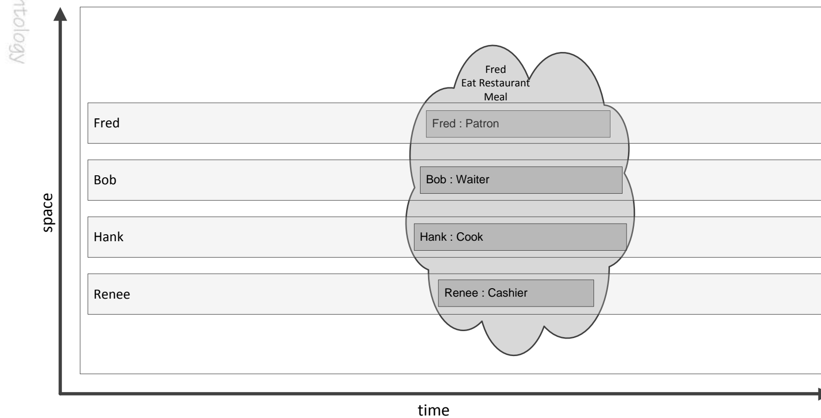


Figure 24 - Component participation space-time map

Figure 25 abstracts the same structure from the interaction diagram – though this is at a higher type level – working at the level of Patron (IndividualType) rather than Fred (Individual).

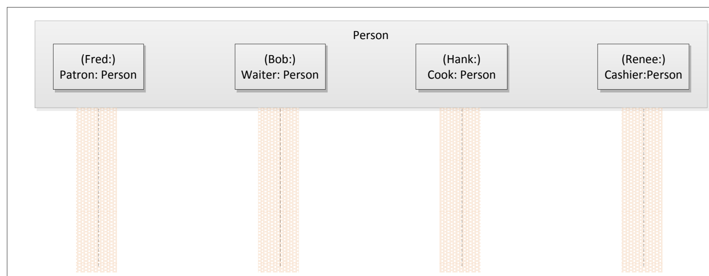


Figure 25 - UML Notation for lifeline participations

The temporal ordering aspect

The second aspect can be seen in Figure 26, which abstracts the temporal orderings from the Interaction diagram. This produces a directed graph in which the interaction components are nodes and dependencies are arrows. As the figure shows there is a close inter-connection between the two aspects; the component nodes are not just components of the interaction, but also components of the participations. This also provides the basis for partitioning the temporal ordering relations into those within and those between the interaction role participations – shown in different colours in the figure.

© 2011 BORO Solutions Limited. All Rights Reserved.

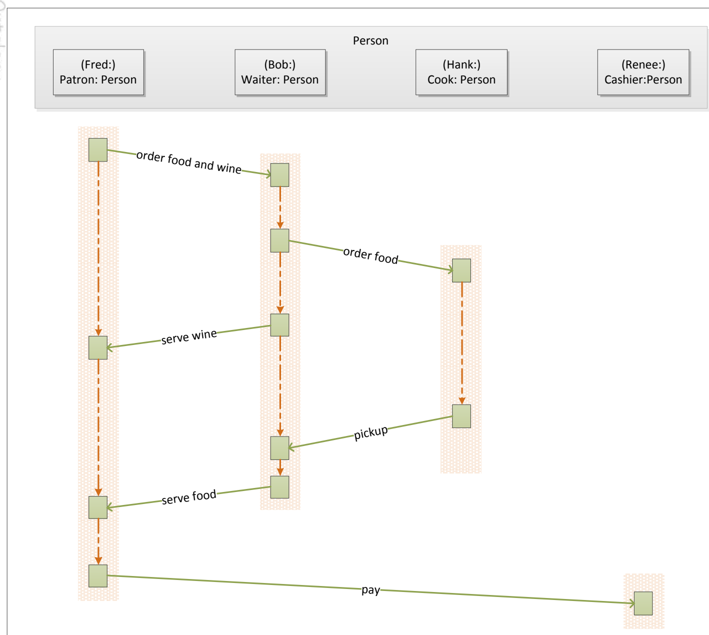


Figure 26 - Interaction component dependencies

UML makes the same distinction. The dependencies across the interaction roles (lifelines) as 'Messages', whereas the dependencies within the roles (lifelines) are based upon a 'GeneralOrderings'.

Non-circularity constraint

An essential feature of temporal orderings is that they are not circular. One cannot follow a string of temporal orderings and return to where one started.

Figure 27 illustrates a circular ordering within an interaction diagram. If one starts at '1' then follows the orderings '2', '3' and '4' one then returns to '1'. A telltale clue here is that the lines in the diagram cross.

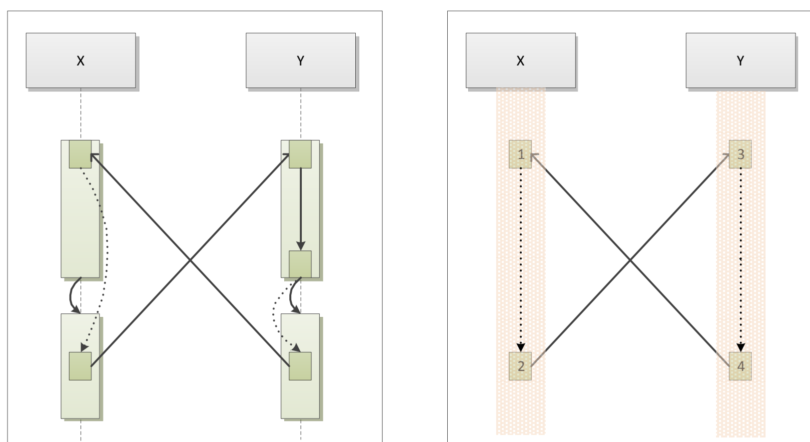


Figure 27 - Circular temporal dependencies

© 2011 BORO Solutions Limited. All Rights Reserved.

Temporal Ordering within an Interaction Role

The ordering within the roles (lifelines) arises from patterns within the lifelines that are not shown in Figure 26. Figure 28 shows the finer detail. In UML, each lifeline (Interaction Role) has two-levels of components, with interaction dependency components at the second level.

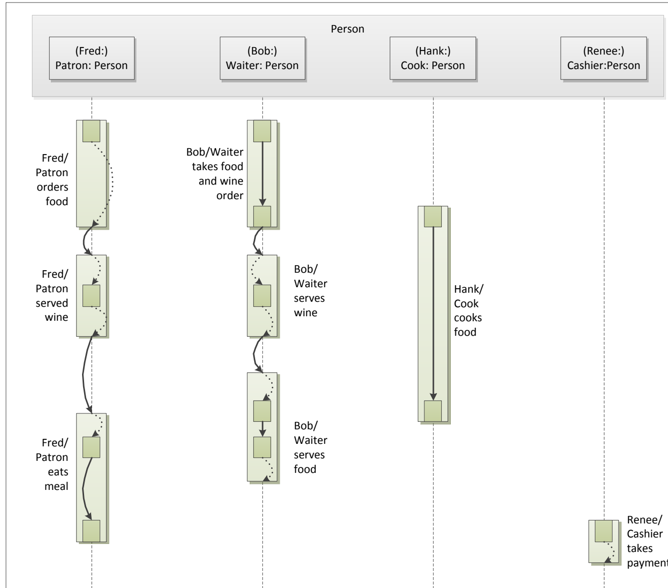


Figure 28 - Interaction Roles and their components

These components exhibit the same state succession pattern that underlies state machines. The components at the first level ('ExecutionSpecifications' in UML-speak) divide the lifeline into sets of disjoint state types. For example, Figure 29 shows the Waiter role/lifeline's three components as disjoint successions – firstly as an abstraction from an interaction diagram and then as a space-time map.

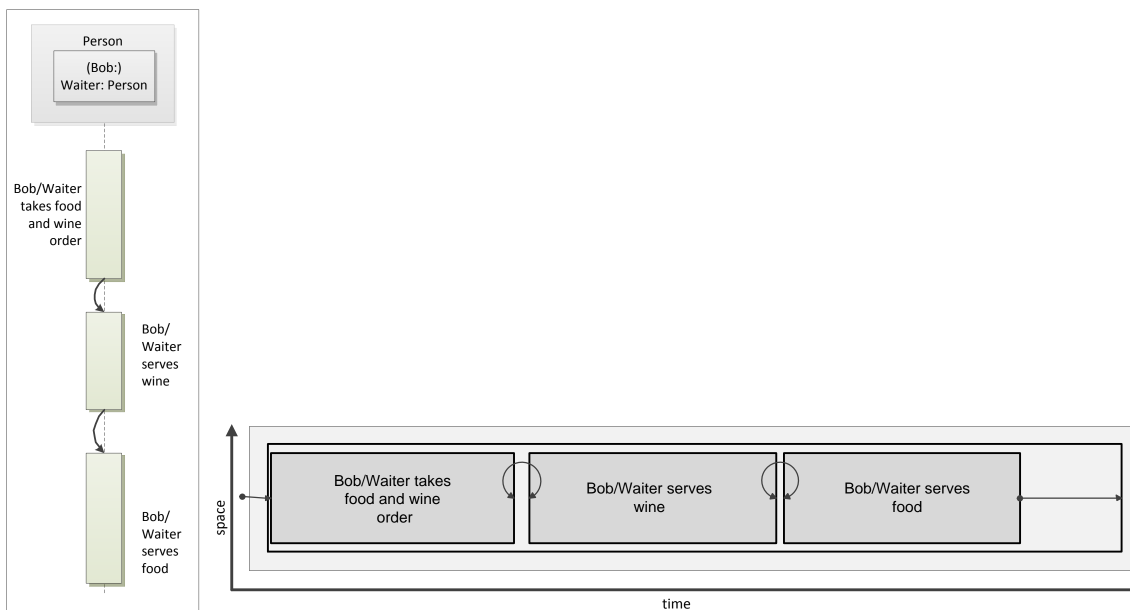


Figure 29 - Lifeline components as disjoint state types

© 2011 BORO Solutions Limited. All Rights Reserved.

This, in effect, means the lifelife in a UML Interaction diagram is a State Machine. This point can be made explicit notationally by using the UML State Machine notation to represent the lifeline – this alternative notation is shown in Figure 30.

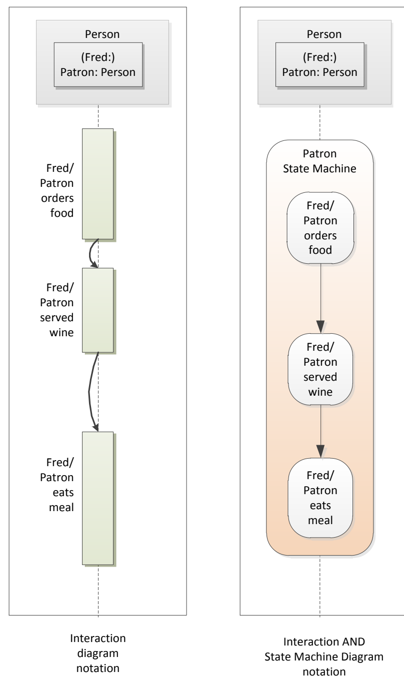


Figure 30 - Combined Interaction and State Machine UML Notation

However, not any state succession is allowed in an Interaction. In the general pattern, there can be orthogonal regions – but not in an interaction diagram. Furthermore, in the general pattern a state type can be succeeded by more than one other type. For instance, in the door example, the door closed could be succeeded by either a door open state or a door locked state. In the interaction diagrams state succession, one state type is always followed by another state type – there is no variety. This leads to a linear chain of state types – and so a linear chain of states in the instances.

Temporal Ordering across Interaction Roles

In UML, ExecutionSpecifications can contain a second level of events, OccurrenceSpecifications. Where these are MessageOccurrenceSpecifications, they are the send or receive end of a message. Figure 31 provides an example; the yellow MessageOccurrenceSpecification is a send message end and the red MessageOccurrenceSpecification is a receive message end.

© 2011 BORO Solutions Limited. All Rights Reserved.

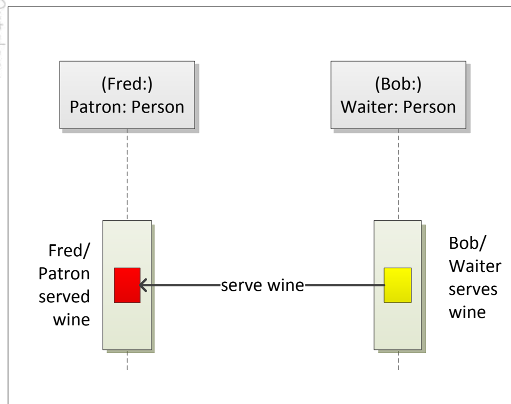


Figure 31 - An Example 'Message'

From a real world semantic perspective, these are a different kind of ordering from the state successions within the interaction role. Earlier we noted that state successions are not causal; the first half of the football match does not cause the second half. Similarly, the waiter serving the wine does not cause the waiter to serve the food. However, the temporal ordering of the send and receive messages is causal. The sending – the waiter serving the wine – causes the receiving – the patron served the wine. In the MODEM model these are called state interactions.

Strong and weak temporal orderings

This semantic difference between state interactions and successions leads to a structural difference. If one looks closely at the example, one can see that the waiter serving the wine must start before the patron receiving the wine – a cause cannot start before its effect. However, in this and other cases, it is likely that the patron starts receiving the wine before the waiter has finished serving it. Similarly, the waiter serving the wine must finish before the patron receiving the wine finishes. This is shown graphically in Figure 32.

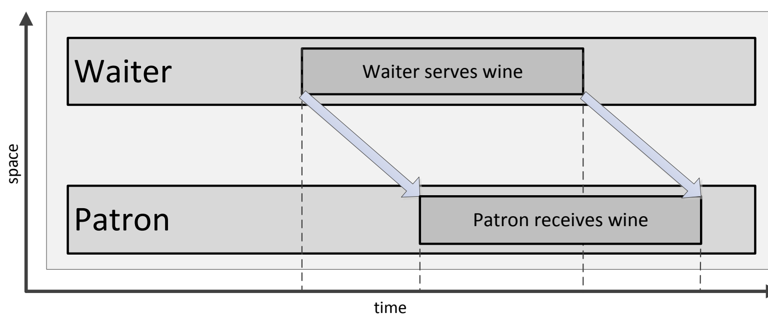


Figure 32 - start-end temporal ordering

This kind of temporal ordering is weaker than the strong total temporal ordering of state successions. As noted earlier, one state must end before the next state can start.

Unnecessary UML Layering

The UML Specification makes a clear and absolute distinction between the two layers in an interaction role (lifeline). The lifeline is divided into ExecutionSpecifications and these are divided into OccurenceSpecifications. The state interaction orderings hang off the

© 2011 BORO Solutions Limited. All Rights Reserved.

OccurenceSpecifications. This levelling is superfluous structure. There is no reason why any leaf state should not be a send/receive node. This can be seen clearly in the example. In Figure 33, the two cases (highlighted) where an ExecutionSpecification contains a single Message OccurenceSpecification, the OccurenceSpecification has been removed and the send/receive link transferred to the ExecutionSpecification.

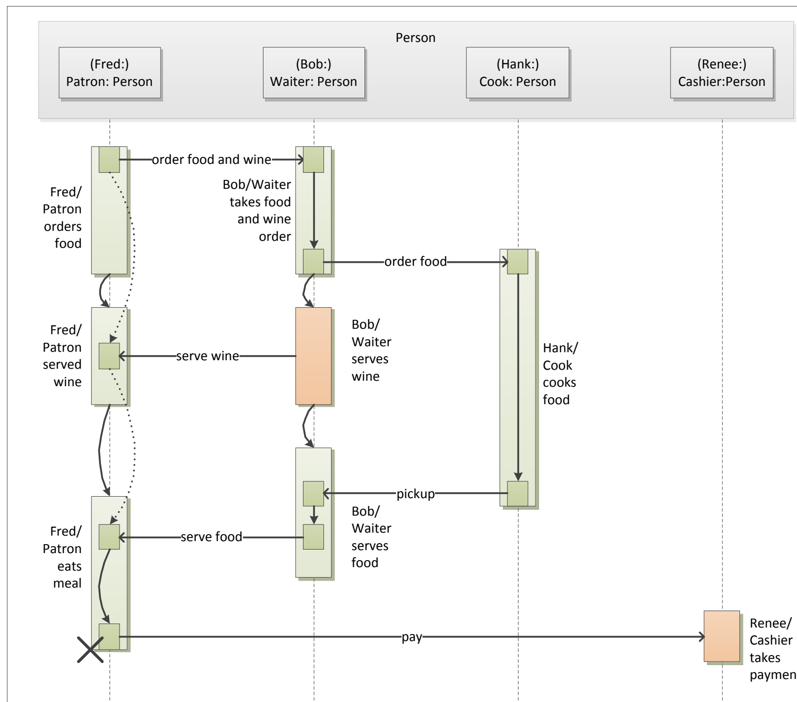


Figure 33 - Removing unnecessary levelling

This makes things clearer. It conveys the same information, and does this with fewer objects. It is also a more faithful representation of what happens. From real world semantics perspective, there seems to be no appreciable difference between the ExecutionSpecification and OccurenceSpecification. They seem to have been introduced merely to fit in with the UML rules. There is no reason to import this constraint into MODEM.

Taking advantage of the embedded state machine pattern

As noted earlier, the analysis shows that the UML Interaction diagram uses a version of the state succession pattern – though this connection is not recognised in any way in the UML specification, where State Machines and Interaction diagrams are treated separately – effectively stove piping them. The version used is quite restricted. This raises the question of why these restrictions are in place and whether they are real.

One of the restrictions is that the interaction lifeline state machines do not cater for orthogonal regions. We can illustrate what one might look like using the chosen example – shown again in Figure 34.

© 2011 BORO Solutions Limited. All Rights Reserved.

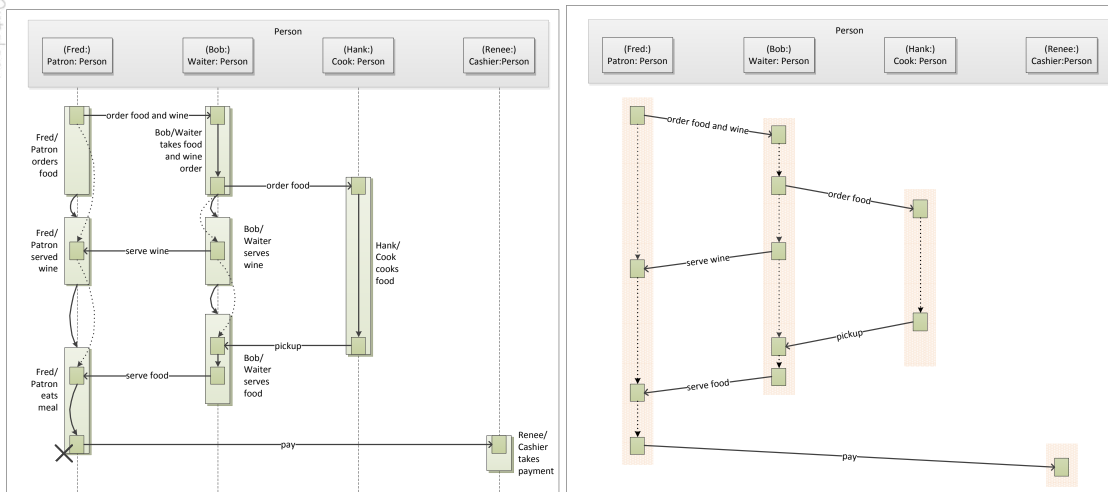


Figure 34 - UML restricted ordering

Let's relax the dependency between the waiter passing the food order to the cook and the waiter serving the wine by introducing a waiter handles food and wine order process which has the waiter passing the food order to the cook and the waiter serving the wine as states in separate regions. The result is shown in Figure 35.

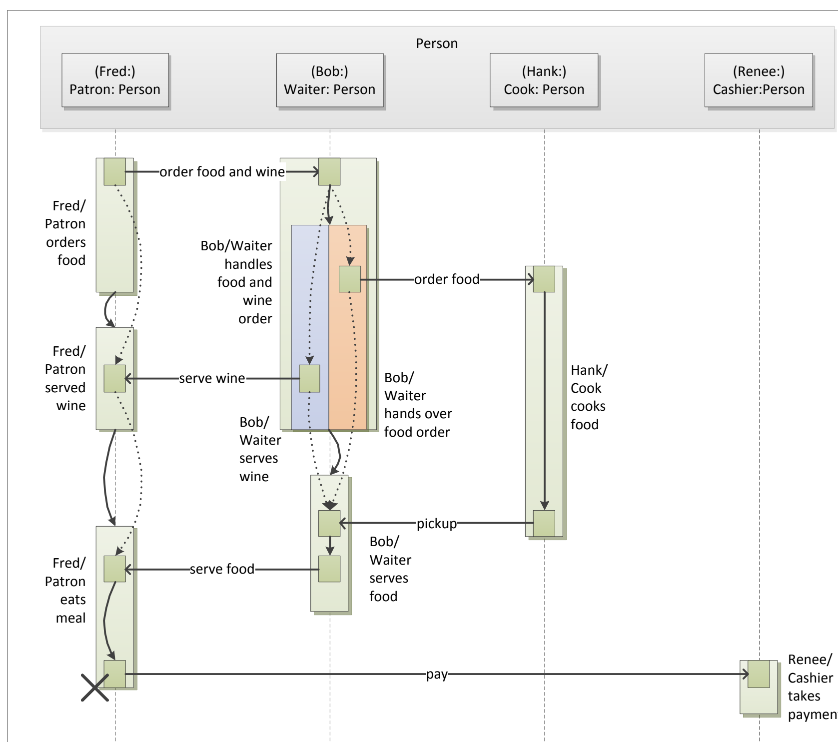


Figure 35 - Interaction diagram with multiple regions

Figure 36 shows the temporal ordering abstracted from the interaction example, with the changes marked in red.

© 2011 BORO Solutions Limited. All Rights Reserved.

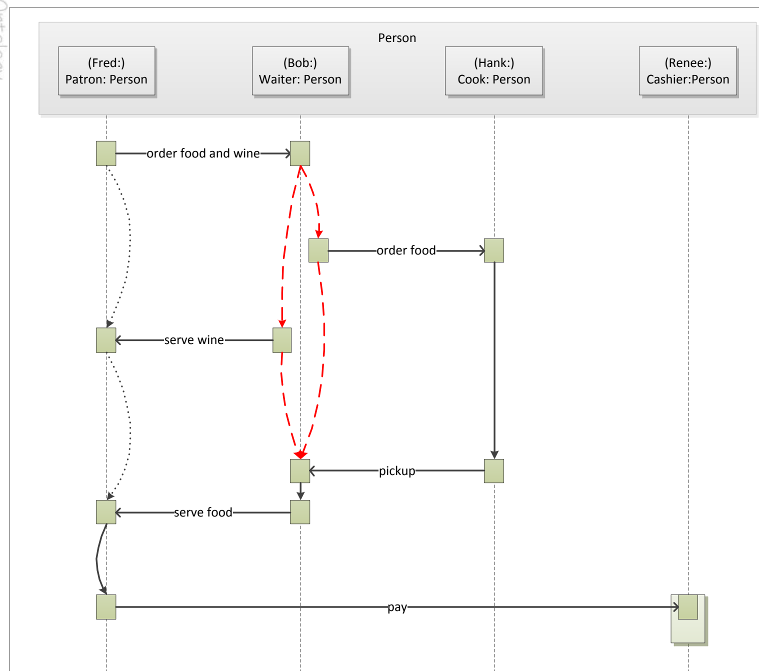


Figure 36 - Multiple region temporal ordering

What this illustrates is that recognising the underlying state succession pattern in the interaction diagram enables new functionality to be added at no extra cost.

Summary

The analysis has identified the real world semantics behind the UML structures creating the basis for a common understanding. The UML structure is divided into rigid siloes. The real world analysis has shown that these silos are not a reflection of the underlying enterprise. These constraints in original structure need not be migrated to MODEM – making it more flexible. The real world semantics makes the underlying business patterns clearer – and so makes the identification of common patterns simpler. A salient example of this is the appearance of the state machine's state succession pattern at the centre of the Interactions. The use of these common patterns makes the model simpler and easier to understand.

© 2011 BORO Solutions Limited. All Rights Reserved.

IDEAS Detailed Technical Analysis

Introduction

This is the third section of the report, which presents the detailed analysis. This translates the points raised in the second section into IDEAS diagrams. Some of these diagrams contain a large amount of information. They are included in the report but may be better viewed in the Worked Examples HTML report.

This section is divided into two parts, one dealing with the state succession pattern, the other dealing with the interaction pattern.

State Type Succession pattern –real world analysis

The analysis starts by giving some context by looking at what UML State Machines are.

UML State Machines

UML State Machines are based upon Harel StateCharts with some additions. The original Harel State Machine focused on the formal structure and relied on an implicit, intuitive real world semantics. The UML State Machines developed the formal structure to meet additional requirements, such as hierarchical state machines and orthogonal regions. Their focus seems to have been more on providing an executable formal structure than developing the real world semantics. One result of this is that the current formal structure of the UML specification for state machines does not map easily onto a real world semantics.

However, careful analysis using the (BORO-based) IDEAS Foundation has recovered the underlying intentions and reconstructed a formal structure with a clear real world semantics.

UML State Machines specification

The UML Superstructure specification contains the specification of state machines (and associated apparatus: regions, transitions and states). A StateMachine contains Regions, which in turn contain Transitions, which in turn link States, which are a sub-type of Vertices (see Figure 37).

© 2011 BORO Solutions Limited. All Rights Reserved.

Package BehaviorStateMachines

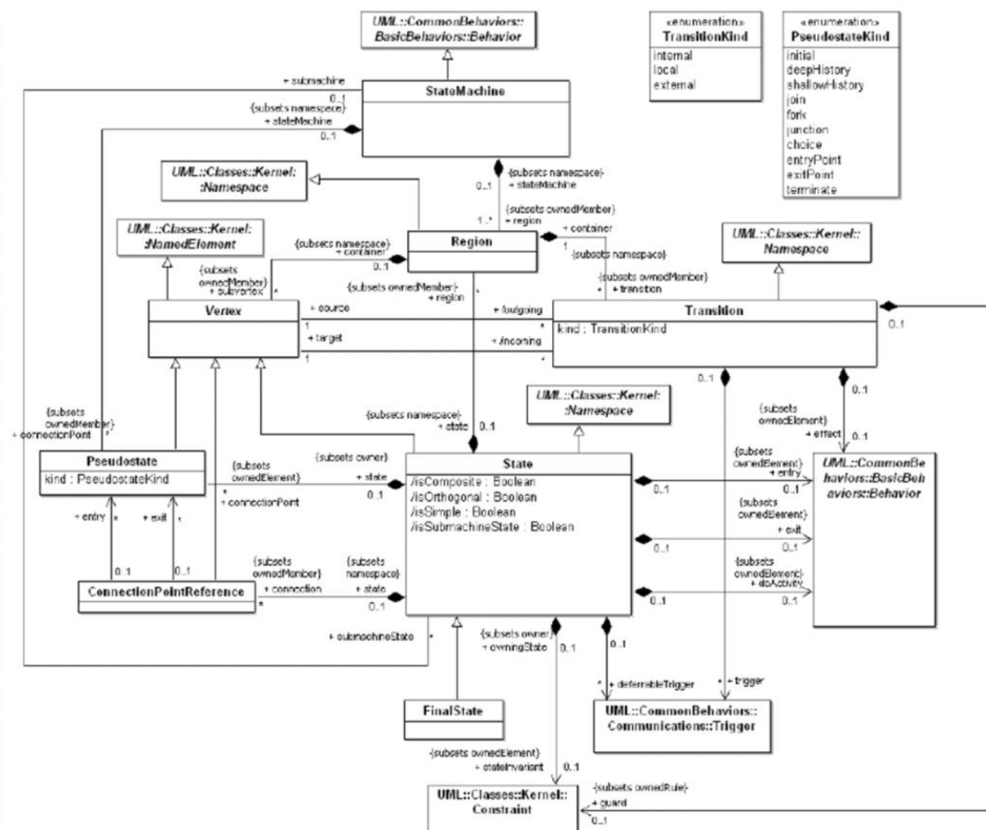


Figure 15.2 - State Machines

Figure 37 – “Figure 15.2 - State Machines” (UML Superstructure Specification, v2.3)

The specification’s diagram is a bit too busy for easy reading, so Figure 38 abstracts this down to the key elements – and adds two missing key elements: StateMachine’s super-type Behaviour and its association with BehaviouredClassifier.

© 2011 BORO Solutions Limited. All Rights Reserved.

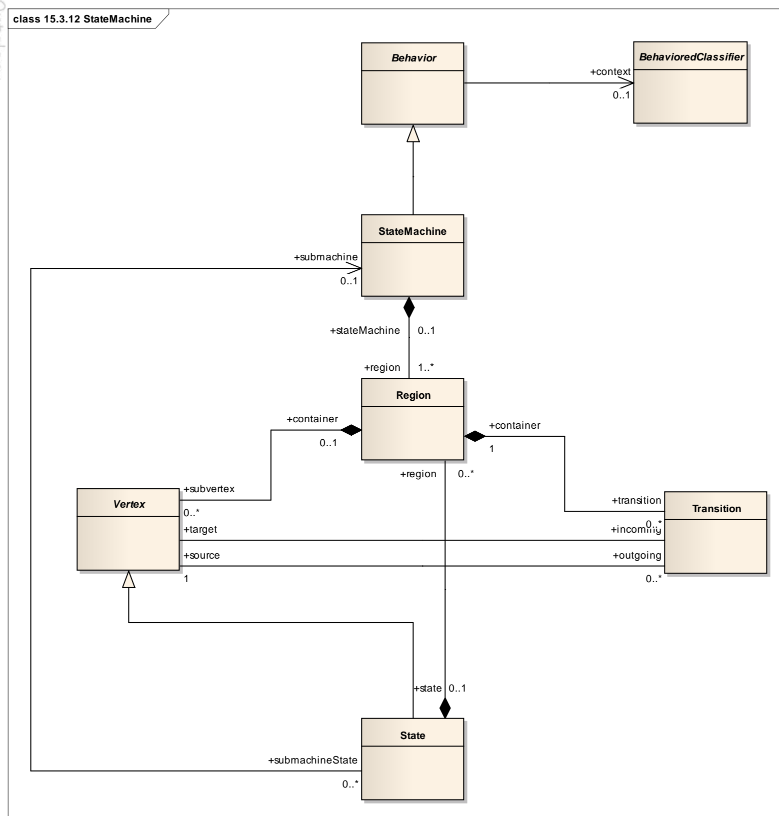


Figure 38 - UML State Machines

The analysis below shows that in some cases this structure obscures the real world semantics, so reconstruction is required to reveal the real world semantics.

Clarifying terminology

While the types of objects used in UML State Machines are similar to the types that are identified in our analysis, they are different in significant ways. This raises the question of what terms to use for the objects found for our analysis. The benefit in using the same term is that its meaning is already known. The problem is that this known meaning will only roughly correspond to the meaning of the object in the analysis, and so potentially mislead. The benefit of using a new term is that it clearly marks that a different sense is intended, but at the cost of not inheriting an already known sense. These costs and benefits have been traded off to arrive at what we hope is a pragmatic decision on the use of terms. So, it seems sensible to use the term 'state' as this is the everyday language term, but in the analysis we use it in the everyday sense (UML uses it for what roughly corresponds to our types of states). However, in other cases, we have used different terms – for example, we use succession for what roughly corresponds to instances of UML transitions. There is a detailed mapping of the terms at the end of the analysis.

The chosen example in IDEAS

The chosen example was a case where a door is opened, closed and then locked – as shown in the earlier state machine in Figure 8 and the space-time map in Figure 12. There is a clear succession (transition) from a door open to a door closed and then to a door locked state.

© 2011 BORO Solutions Limited. All Rights Reserved.

This is captured in the following two IDEAS diagrams. Figure 39 shows in IDEAS format the states as temporal parts of ‘No 73’s Door’. Figure 40 shows the successions.

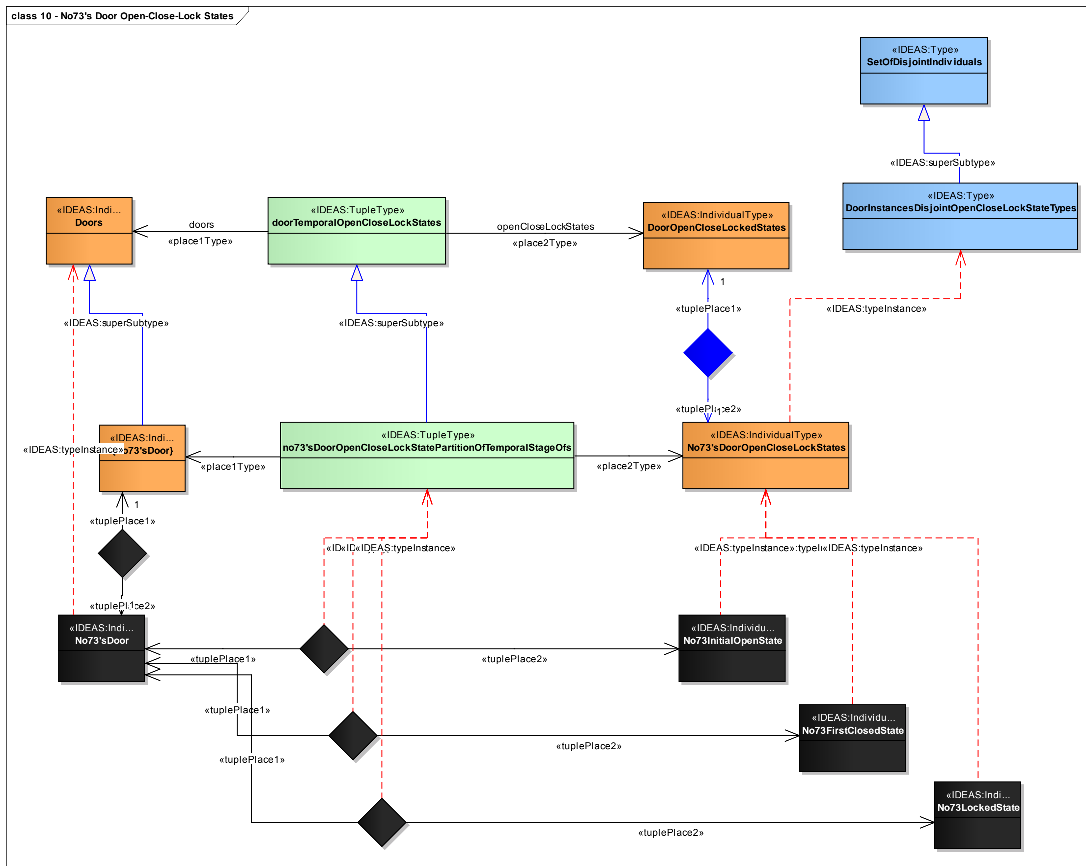


Figure 39 - States as temporal parts of No 73’s Door

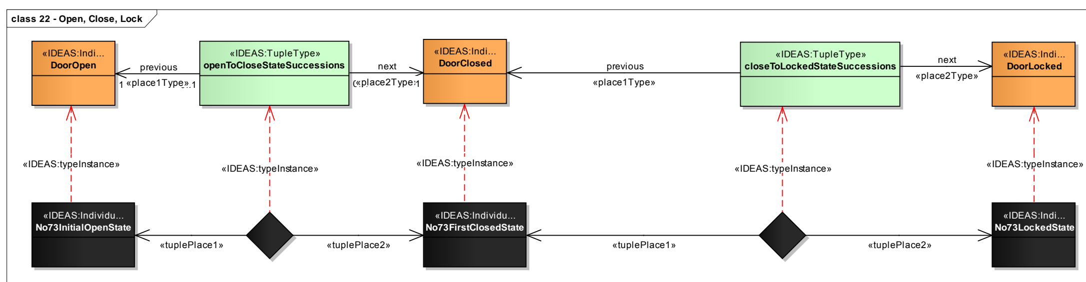


Figure 40 – No 73’s Door successions

One can see both in the earlier space-time map and the IDEAS succession diagram that the states form a chain or line with an initial state followed by a number of state successions (or transitions) and then a final state.

IDEAS ‘State Type Successions’ pattern

This shows the state succession pattern grounded at the individual level for an individual door. We then take it up a level for doors in general. In the earlier analysis, we showed the pattern in the grid in Figure 16 – and reproduced in Figure 41.

© 2011 BORO Solutions Limited. All Rights Reserved.

This grid tells us which types of succession are possible and which aren't. For example, if the door is open, it can only be closed, it cannot be locked – a Door Open, if it is succeeded, must always be succeeded by a Door Closed state, it cannot be succeeded by a Door Locked state. One can easily see this in the matrix, as the the Previous State – Open Door row (circled), the Locked Door column is greyed out, showing this not a feasible type of succession.

PREVIOUS STATE	NEXT STATE			
	OPEN DOOR	CLOSE DOOR	LOCKED DOOR	final
initial	✓			
OPEN DOOR		✓		✓
CLOSE DOOR	✓		✓	
LOCKED DOOR		✓		
OPEN-CLOSED-LOCKED DOOR STATES SUCCESSION GRID				

Figure 41 - Open-Closed-Locked Door States Succession Grid

We represent this grid pattern of successions between the states in the IDEAS diagram in Figure 42. Here the rows and columns of the grid translate into relations between the partitions of the state types giving a different perspective of the same structure.

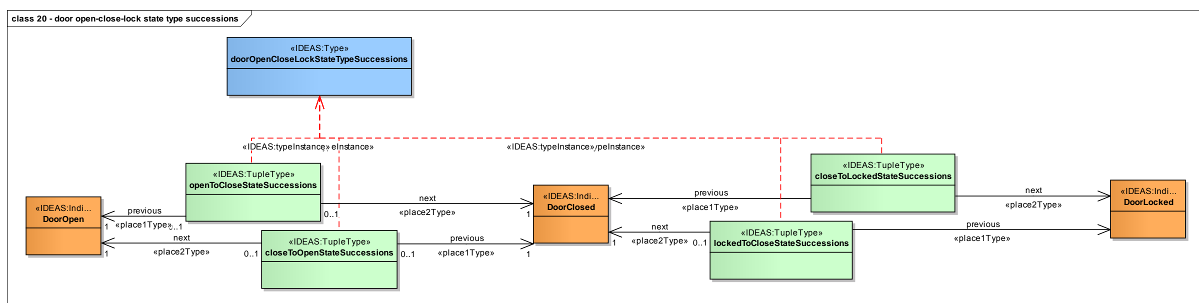


Figure 42 - State successions

Structurally, we have (in IDEAS terms) an Individual Type – a thing (doors) whose instances are individuals with spatio-temporal extent. This thing (doors) has three types of states, whose instances are temporal stages of doors. This is shown in the IDEAS diagram in Figure 43, and in more detail in the worked example.

© 2011 BORO Solutions Limited. All Rights Reserved.

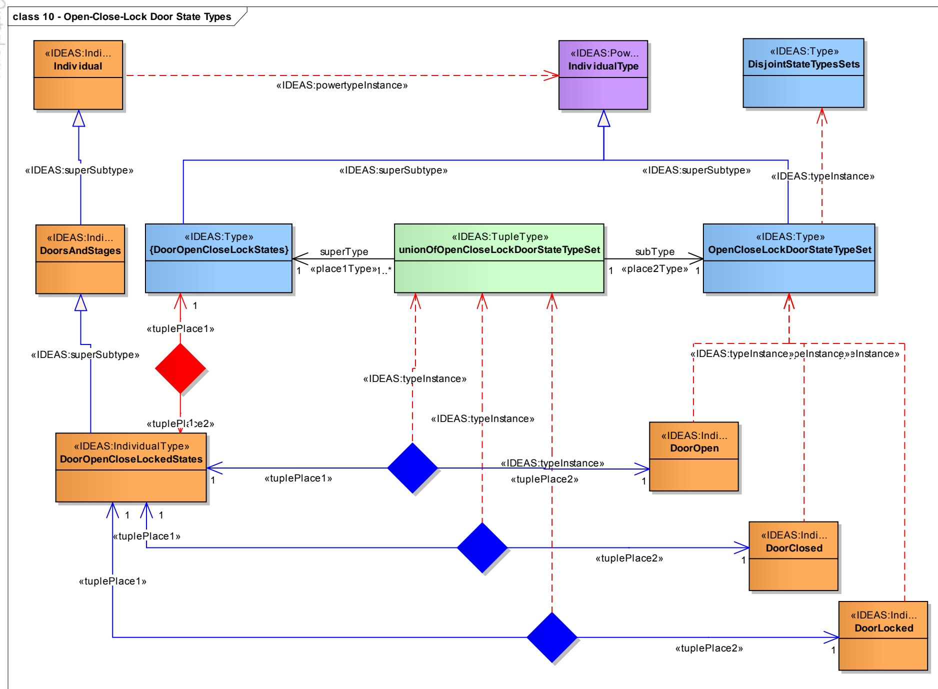


Figure 43 - Example IDEAS State Types Set diagram

IDEAS 'DisjointStateTypesSets'

Earlier we noted that something is a 'DisjointStateTypesSets' if:

1. It contains a disjoint set of types
2. The union of these types are all temporal stages of some sub-type of 'Individual' and
3. These temporal stages are instance-wise disjoint relative to the sub-type.

In practice, when creating an IDEAS diagram one creates the instance of 'DisjointStateTypesSets' with its state types and an instance of 'IndividualTypeSingleton' that contains the singleton of the 'owning' sub-type of 'Individual' and an instance of 'instanceWiseDisjointStateTypeSets' linking the two. This is shown in Figure 44 for the doors example.

© 2011 BORO Solutions Limited. All Rights Reserved.

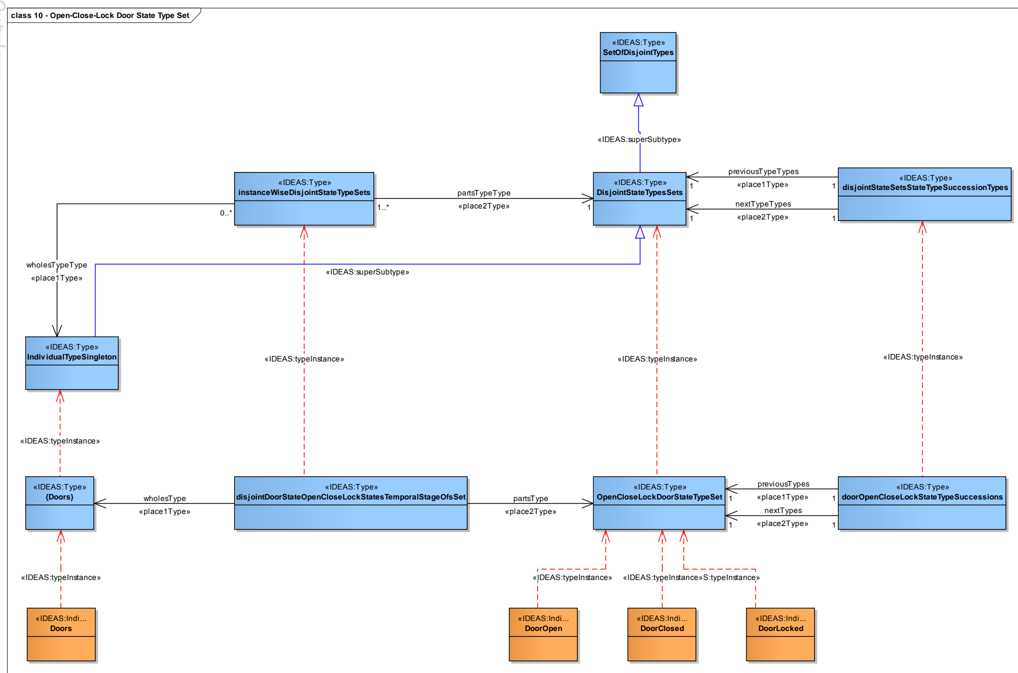


Figure 44 - Example instance of DisjointStateTypesSets

Constraint 1) is easily seen here: 'DisjointStateTypesSets' are a sub-type of 'SetOfDisjointTypes' and so its instances are disjoint types. Constraints 2) and 3) mentioned above are quite complex and they are shown in detail in the worked example. Practical considerations dictate that there is no need for the user to manually enter the lower level details for these last two every time.

Taking away the details of the door example give us the pattern – shown in Figure 45.

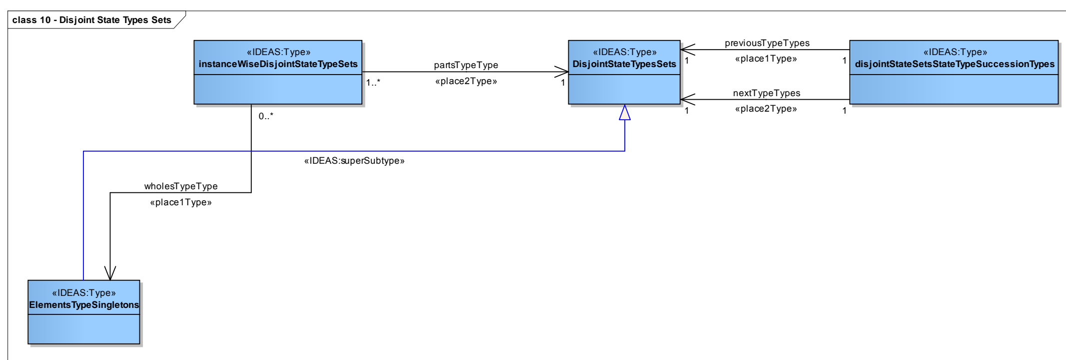


Figure 45 - state type succession pattern

IDEAS 'disjointStateSetsStateTypeSuccessionTypes' pattern

The successions were illustrated by a grid in Figure 16 and IDEAS diagram in Figure 40 and Figure 42. The IDEAS diagrams can be extended to show the finer details of the grid – Figure 46 is an example

© 2011 BORO Solutions Limited. All Rights Reserved.

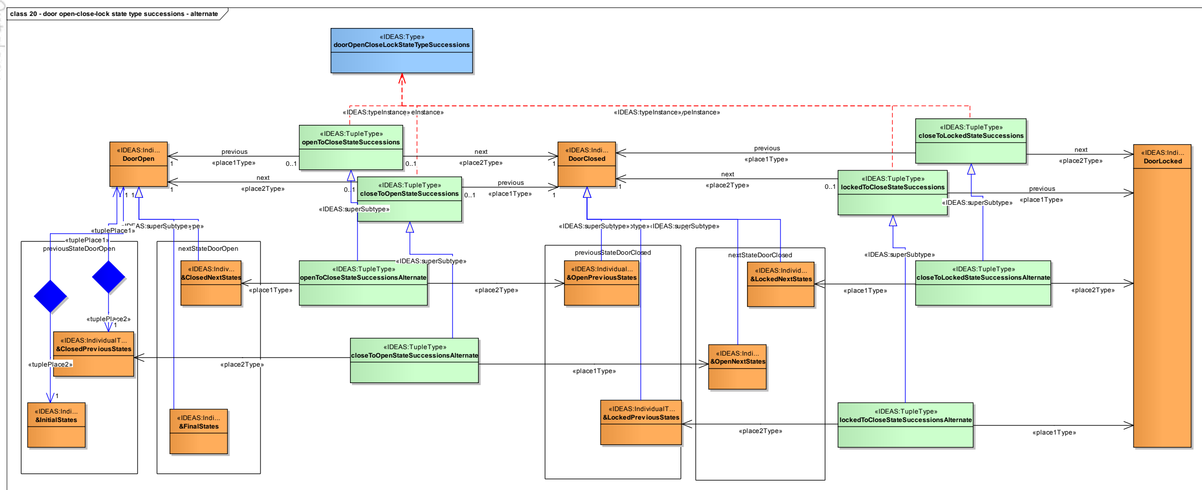


Figure 46 - Finer succession grid

What this diagram does is partition each state type in two ways; by its previous state and by its next state. This, in effect, recreates the grid in the IDEAS diagram. From a row perspective, the cells of the grid partition the state type – from a column perspective, the cells of the grid partition the state type. For example, in Figure 47 the Open Door row is a partition of the Open Door state type by previous state type– highlighted in green in the figure. Similarly, the Open Door column is a partition of the Open Door state type by next state type – highlighted in yellow in the figure.

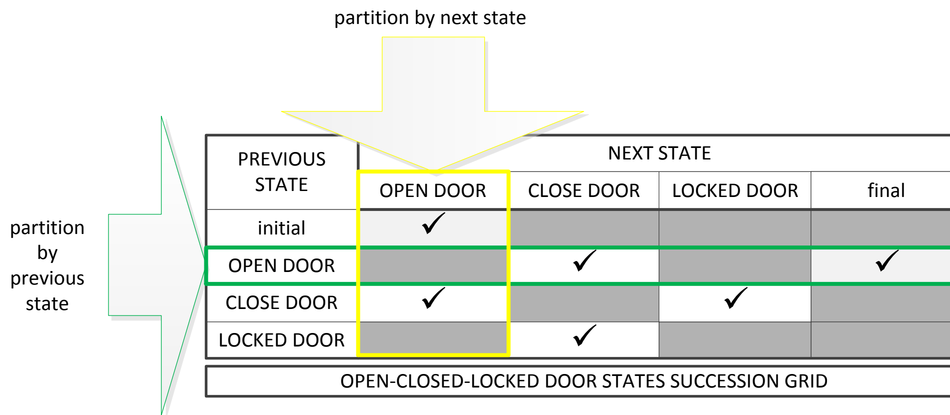


Figure 47 - Grid cells as partitions of the state types.

This example illustrates another feature of the successions: there is an initial state and a final state. At the individual level, we noted earlier that each chain has an initial state and a final state - these are shown using arrows in Figure 13. At the type level, it is possible for different instances to have different state types as initial or final state – in terms of the grid, for more than one row (initial) or column (final) to have ticks. However, in this example, there is only one state type that is 'initial' and one that is 'final'.

IDEAS hierarchy of 'DisjointStateTypesSets'

Figure 21 showed an example hierarchy of 'DisjointStateTypesSets', illustrating that there is no general constraint on how many 'DisjointStateTypesSets' an Individual sub-type can

© 2011 BORO Solutions Limited. All Rights Reserved.

have, provided they meet the criteria for ‘DisjointStateTypesSets’. The example in Figure 21 is translated into IDEAS format in Figure 48.

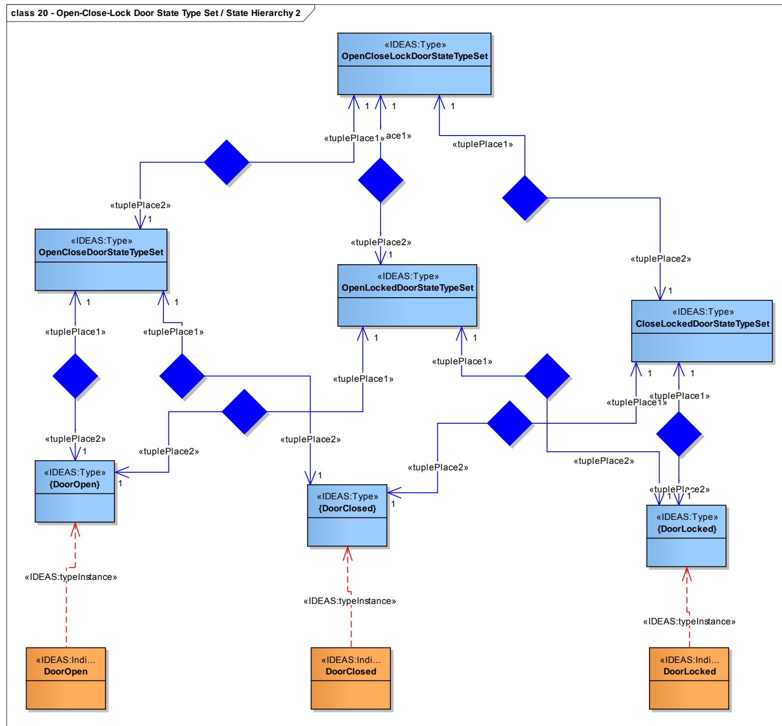


Figure 48 – Part of an individual sub-type’s hierarchy of ‘DisjointStateTypesSets’

IDEAS Multiple ‘orthogonal’ ‘DisjointStateTypesSets’ pattern

UML supports multiple orthogonal regions and this is illustrated in the chosen example (Figure 8) with Door Alarmed and Door Open Close Lock. This UML feature is translated into IDEAS format in Figure 49 as two ‘DisjointStateTypesSets’: AlarmedDoorStateTypeSet and OpenCloseLockDoorStateTypeSet. This shows how the behaviour of an individual can be characterised by a number of different ‘DisjointStateTypesSets’ patterns.

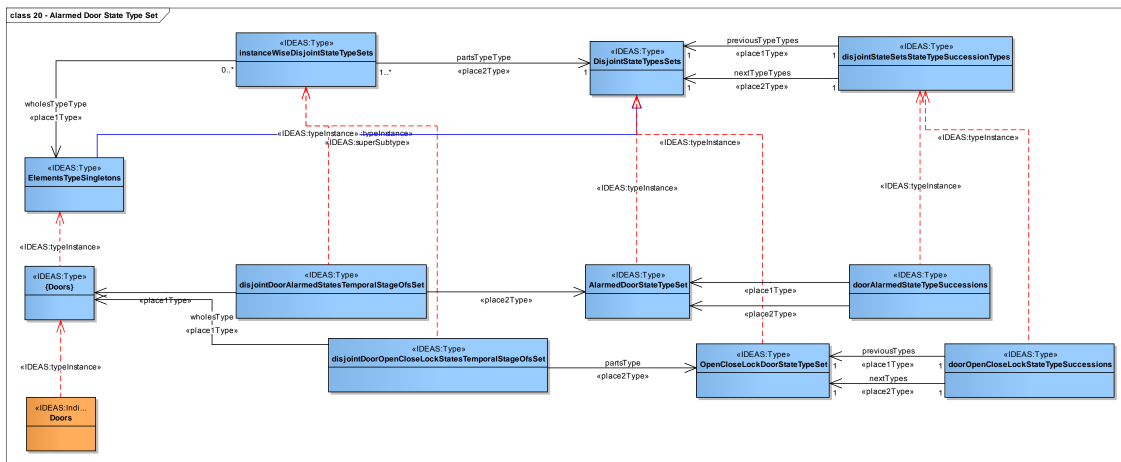


Figure 49 – Multiple ‘orthogonal’ DisjointStateTypesSets

© 2011 BORO Solutions Limited. All Rights Reserved.

IDEAS Nested 'DisjointStateTypesSets' pattern

UML supports the nesting of states in its state machines and this illustrated in the chosen example (Figure 8) with Door Alarmed and its nested Alarm Levels One and Two. This is translated into IDEAS format in Figure 50 as the 'DisjointStateTypesSets' as AlarmedDoorStateTypeSet and its nested Alarmed Door Level One - Level Two State Type Set.

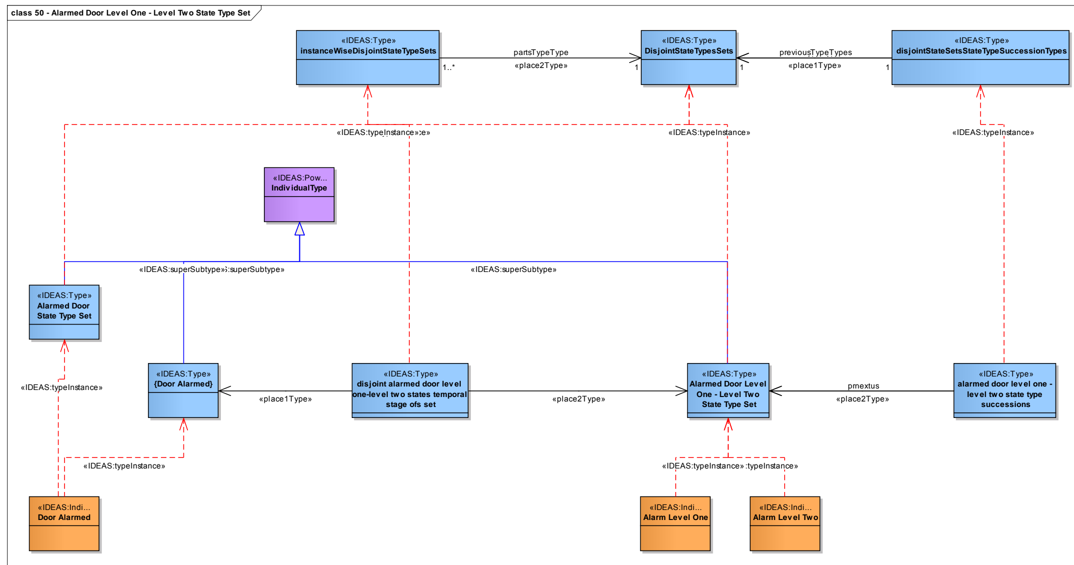


Figure 50 - Example of a nested 'DisjointStateTypesSets' pattern

IDEAS 'State Successions' Inheritance pattern

The analysis described the way 'DisjointStateTypesSets' and so the state succession pattern is inherited using the door example. This was shown graphically in Figure 22. This is translated into IDEAS in Figure 51 and Figure 52. Figure 51 shows the 'Open-Close Fridge Door State Type Set'.

© 2011 BORO Solutions Limited. All Rights Reserved.

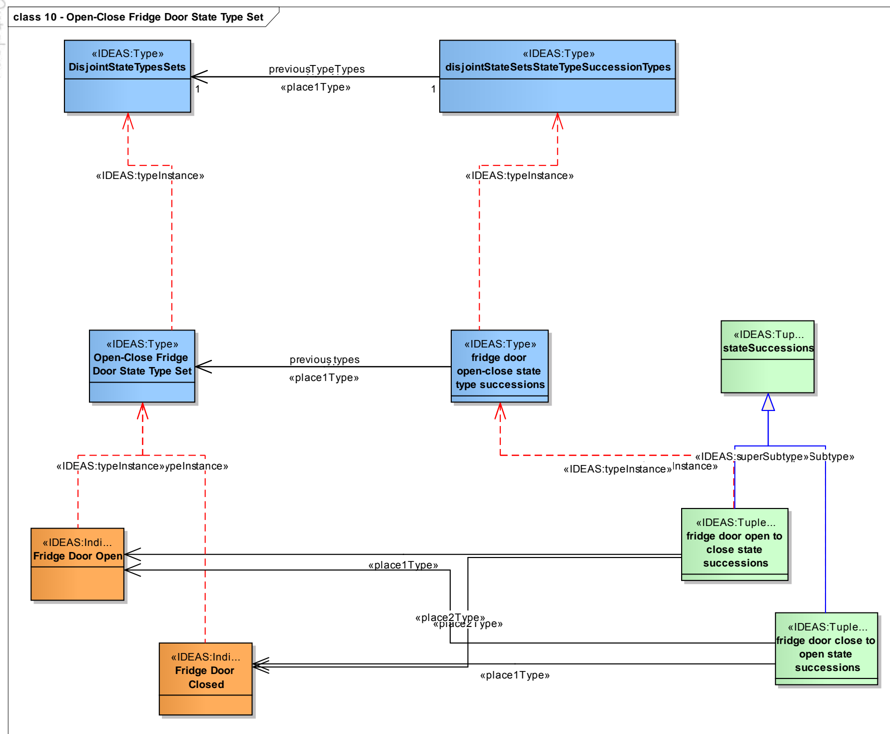


Figure 51 - Open-Close Fridge Door State Type Set

Figure 52 shows how the pattern is inherited though sub-types of superSubType.

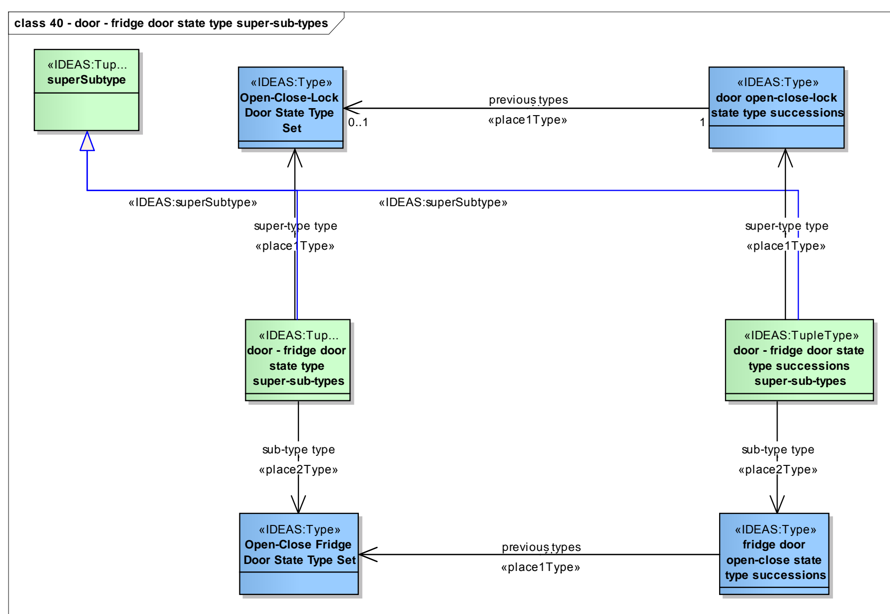


Figure 52 - example of specialisation

Providing a real world semantics for UML State Machine and its UML Components

It is possible to re-construct one from the bottom up real world analysis of example state successions, a real world interpretation of UML StateMachine, and its components. UML

© 2011 BORO Solutions Limited. All Rights Reserved.

State Machines can be views as built from UML Regions, which in turn are built from UML States – so States are a good foundation to start with.

UML State

From a term perspective, IDEAS and UML use of the term 'State' reflects different type levels. In IDEAS, states are individual. In UML, State is an Individual type – a set or collection of IDEAS states. In large part, this is motivated by IDEAS's analysis methodology that grounds the analysis in individuals, whereas the UML approach works at the specification level. It provides examples, but these are at the type level and not grounded in individuals.

Instances of UML States are things such as Door Open – collections of state instances that are disjoint and belong to the same owner. This implies that UML States are the set of all these collections. In MODEM, we have named this 'OwnedStateSets'. All the instances of 'DisjointStateTypesSets' are sub-types of this – as shown for 'OpenCloseLockDoorStateTypeSet' in Figure 53.

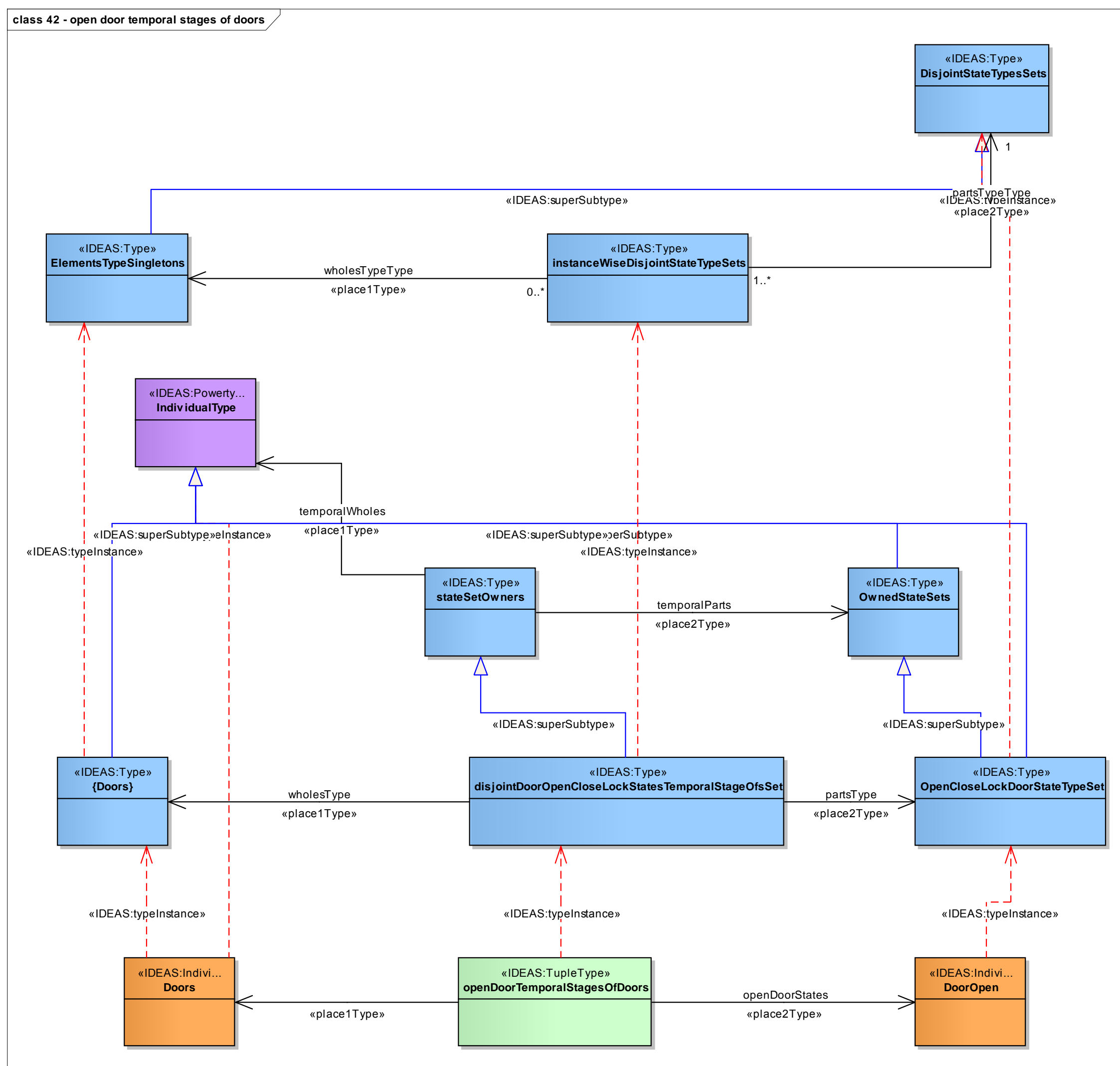


Figure 53 - instances of 'DisjointStateTypesSets' are sub-types of 'OwnedStateSets' example

© 2011 BORO Solutions Limited. All Rights Reserved.

Figure 53 shows the OwnedStateSets' (State) implicit ownership relation – in UML this is inherited by the instances of State from their owning region, which inherits it from the owning region or state machine.

At the general level, this cashes out as 'DisjointStateTypesSets' being a sub-type of the powertype of 'OwnedStateSets' – as shown in Figure 54. The instances of 'DisjointStateTypesSets' are sub-sets of 'OwnedStateSets' that belong to a particular Region. Architecturally, one can look at this as 'DisjointStateTypesSets' explicitly doing the work that the Region partition does implicitly.

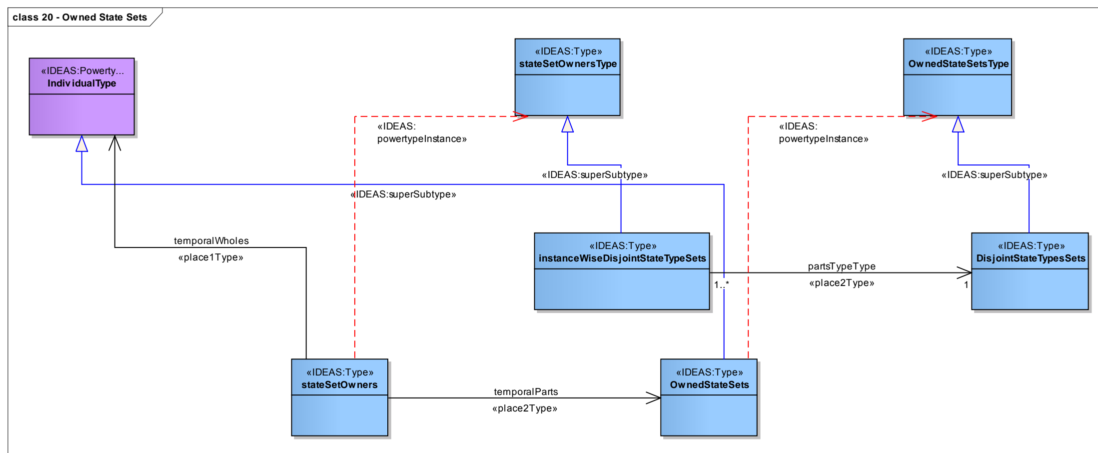


Figure 54 - 'DisjointStateTypesSets' is a sub-type of the powertype of 'OwnedStateSets'

UML Region

The reconstruction suggests that a Region is a set of the state types – a 'DisjointStateTypesSet'. Figure 55 illustrates this using the Door Open-Close-Locked Region example.

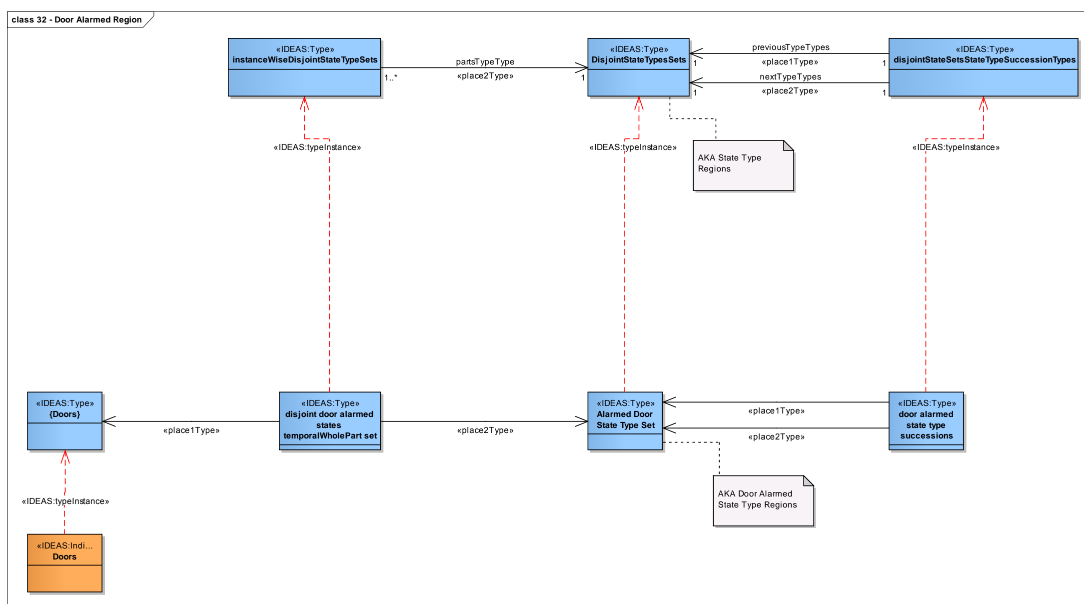


Figure 55 - Door Open-Close-Lock Region example

© 2011 BORO Solutions Limited. All Rights Reserved.

UML State Machine

StateMachines are best regarded as user selected (in other words, a view of a) set of regions ('DisjointStateTypesSets') for a particular owning individual type. This structure closely reflects the structure of the UML State Machine diagram (shown for example in Figure 7 and Figure 8), where the state types are within the boundary of the state machine icon are instances of the state machine. Once the state types are known, the successions can be worked out. Figure 56 and Figure 57 illustrate this using the Door Open-Close-Locked Region and Door Alarmed example. This is also an example of orthogonal and nested regions – as the view includes the orthogonal Open-Close-Lock Door and Alarmed Door regions and nested Alarmed Door and Alarmed Door Level One-Two regions.

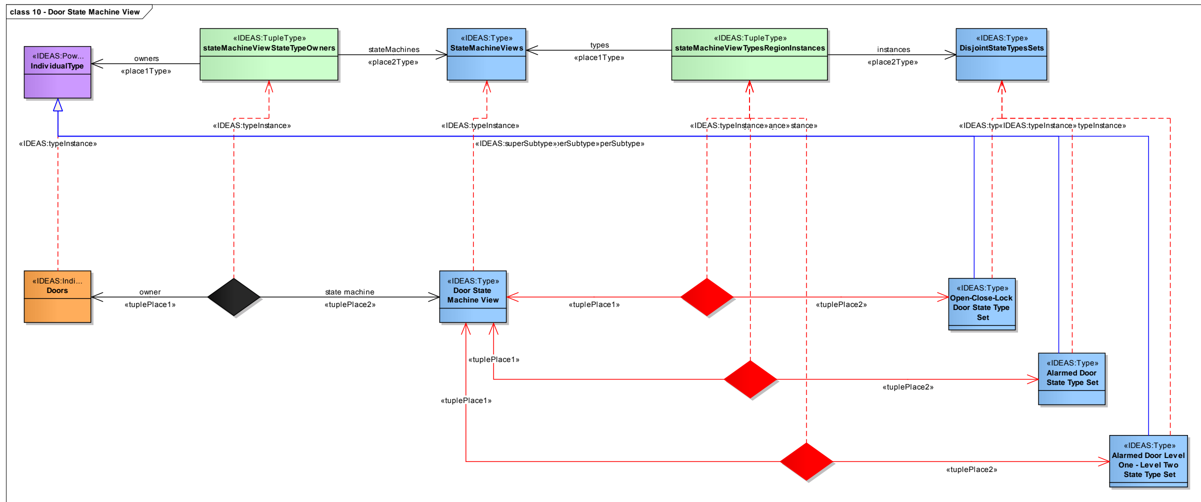


Figure 56 - Door State Machine View

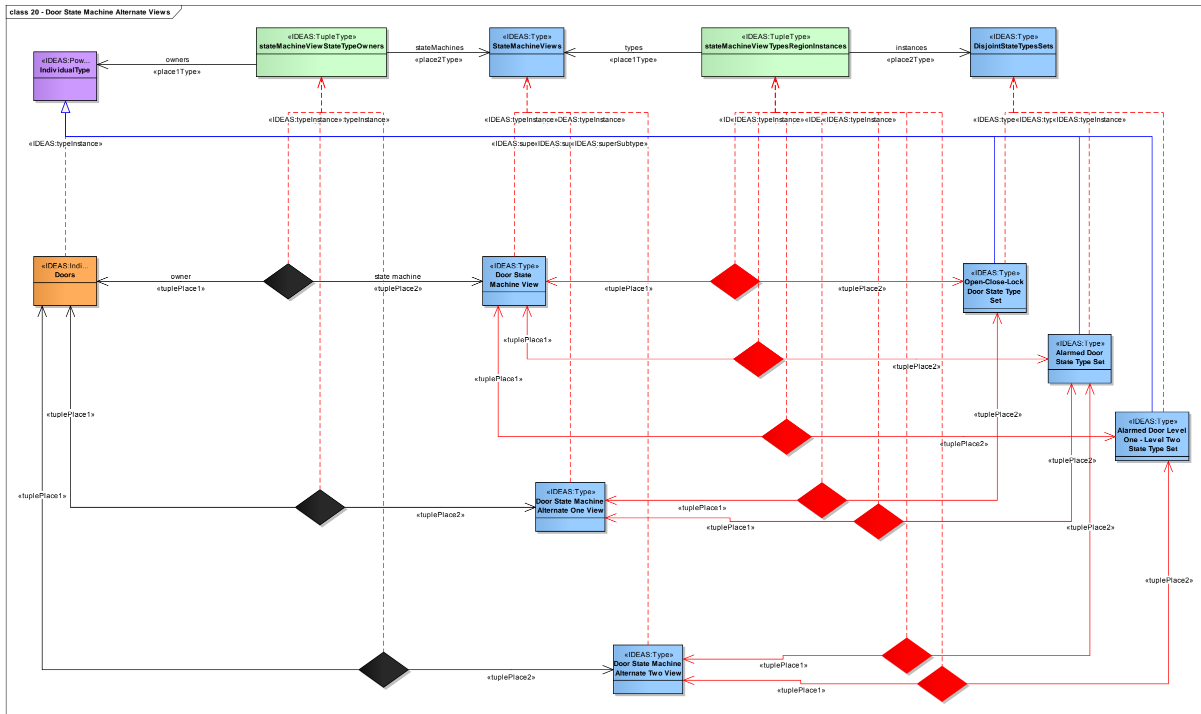


Figure 57 - Alternate State Machine Views

© 2011 BORO Solutions Limited. All Rights Reserved.

UML Repository –MODEM View - State Machine, Region and State

However, the UML state, region and state structure has some quirks, from a real world perspective. There are several equivalent ways to model the same behaviour but in UML, in each model, only one way can be used.

For example, in the real world there are domains where one wishes the same state type to appear in different regions. In UML, one cannot do this in the same model except with a workaround. However, one can do it in UML if these different regions are in different models. The problem in UML is 're-using' state types in more than one region – in one model. Within a model, each modelling choice excludes the other choices.

Similarly, in UML one can model a domain as a single state machine, with several regions, or also as several state machines each with one region or any combination in between this and the original model. Each of these modelling choices can be modelled in UML – but only one in any one UML model. Within a model, each choice excludes all the other choices. This is the issue shown graphically in Figure 6 above.

Furthermore, there seems to be no feature of the real world that suggests which of these state machines or regions is preferred – so the modelling choice is not motivated by real world concerns.

If the state machines and regions in the various models all exist in the real world – then a 'true' model of the real world would allow one to include all of these in the model. One can do this in the MODEM model.

However, UML does not allow this. These types of issues permeate the UML StateMachine structure. The underlying issue here seems to be that UML regards the state machine and region as kinds of repositories where their contents have to be in a single repository. This leads to the question of which repository is the correct one – which as we noted above has no correct answer. MODEM regards the State Machine and Region as views over the repository where the same region can be in multiple state machine views and the same state type in multiple Region views.

UML Transitions

UML Transitions respond to a similar treatment. In the UML specification, a transition is described (p. 587) as follows:

> "Description: A transition is a directed relationship between a source vertex and a target vertex. It may be part of a compound transition, which takes the state machine from one state configuration to another, representing the complete response of the state machine to an occurrence of an event of a particular type."

However, in UML a State is sub-type of Vertex, so some Transitions will link elements other than Vertices. For our analysis, we are interested in those transitions that link States (Owned State Sets). This introduces a delicate issue, one that also illustrates how UML as constructed does not reflect real world semantics. In a particular StateMachine, there is no requirement that there should be a direct transition between each State representing each real world state type succession - this could be represented by a series of transitions. However, in the cases where there is a series of transitions, it is technically possible to represent the same domain with a different StateMachine that does have a direct transition

© 2011 BORO Solutions Limited. All Rights Reserved.

representing the real world state set succession. However, it is not possible in UML to treat these State Machines as views over the same set of real world objects. Simplifying a little, one can assume that UML Transitions include all the real world state set transitions– in other words; UML Transitions is a super-type of stateSetTransitions. We show this in Figure 58.

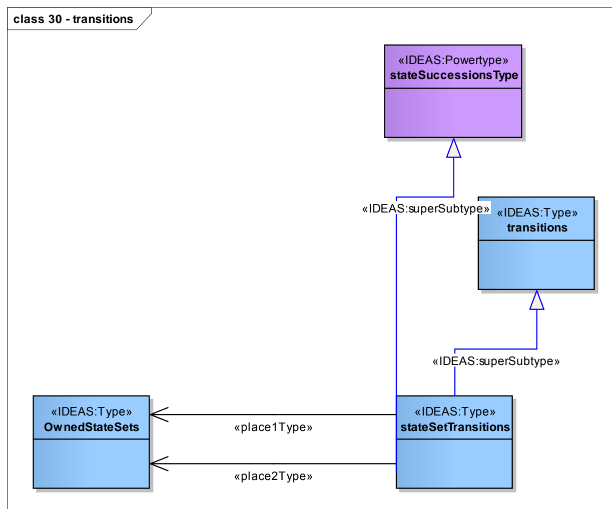


Figure 58 - transitions and state set transitions

The pattern here is much like 'State'. The instances of 'stateSetTransitions' range over the transitions of all States (OwnedStateSets). The MODEM Successions map indirectly onto these stateSetTransitions, as they are at different type levels. The instances of 'disjointStateSetsStateTypeSuccessionTypes' are sub-types of 'stateSetTransitions' – as shown in the example in Figure 59.

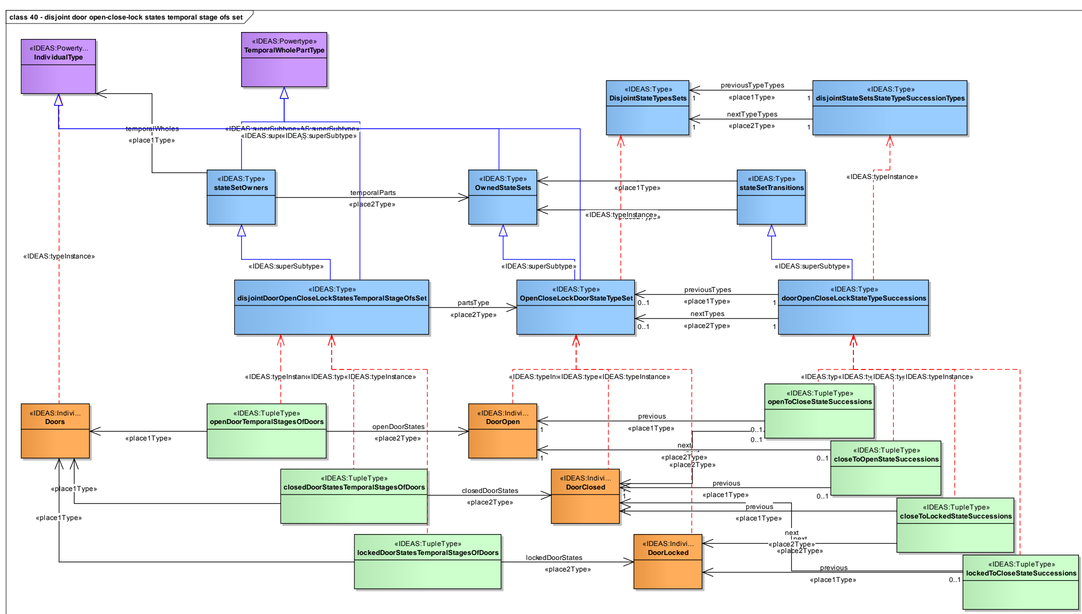


Figure 59 – state set transations example

The inclusion of Compound Transition in a UML Region partitons them into the instances of 'doorOpenCloseLockStateTypeSuccessions'. Architecturally, one can look at this as

© 2011 BORO Solutions Limited. All Rights Reserved.

'doorOpenCloseLockStateTypeSuccessions' explicitly doing the work that the Region partition does implicititly. This pattern generalises to doorOpenCloseLockStateTypeSuccessions' being a sub-type of the powertype of StateSetTransitions – as is shown in Figure 60.

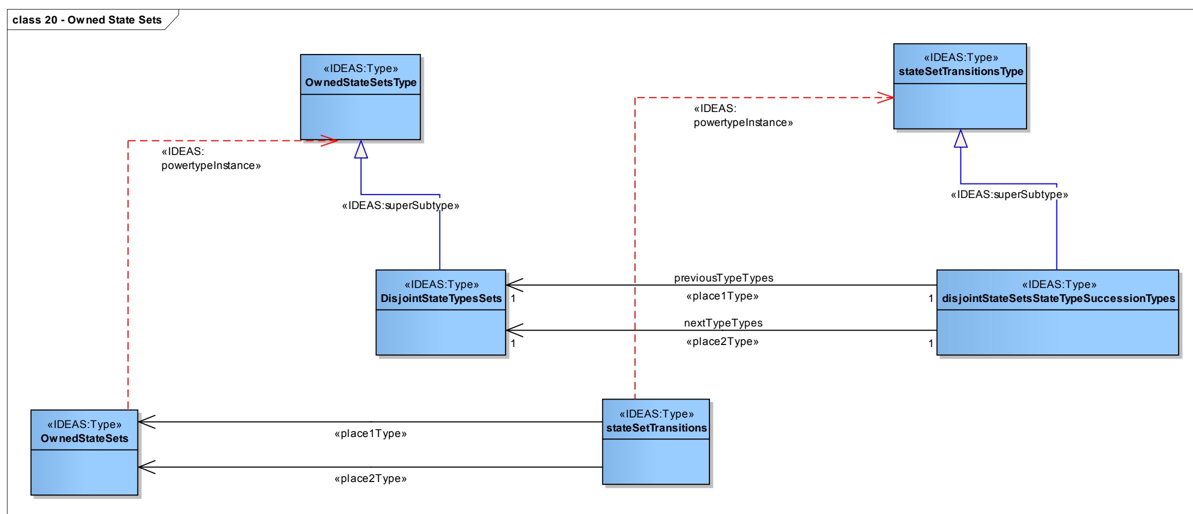


Figure 60 – state set transitions pattern

UML Behaviour and BehaviouredClassifier

In UML, there is a zero-to-one association called 'context' between Behaviour and BehaviouredClassifier. It is described in the specification (p. 446) as follows:

> /context: BehavioredClassifier [0..1]
>
> The classifier that is the context for the execution of the behavior. If the behavior is owned by a BehavioredClassifier, that classifier is the context; otherwise, the context is the first BehavioredClassifier reached by following the chain of owner relationships.

This is the type of which the states are temporal parts. Within UML, the StateMachine's Regions' States inherit this relationship.

Its BehavioredClassifier attribute is re-constructed as an association with the state machine owner – 'stateMachineStateTypeOwners' in the IDEAS model. For example, the Door StateMachine is the set of the two regions: Door Open-Close-Locked Region and Door Alarmed Region. It is owned by the individual sub-type 'Doors' - as shown in Figure 61.

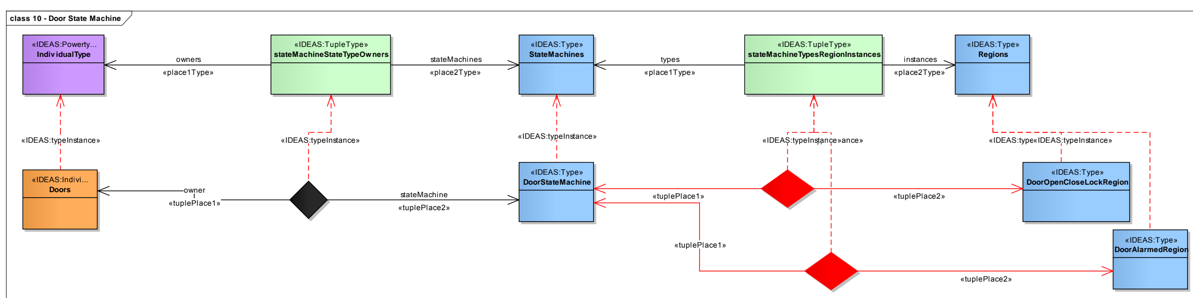


Figure 61 - Door Open Close Lock State Machine

© 2011 BORO Solutions Limited. All Rights Reserved.

UML allows for both multiple StateMachines owned by the same individual sub-type – where the regions 'inherit' this ownership. From a MODEM perspective, this relationship is viewed at the 'DisjointStateTypesSets' (Region) level. If an individual type has one or more 'DisjointStateTypesSets', then each of them has the relation.

Mapping the Real World Semantics back to the UML State Machine

This mapping can serve a number of purposes. For those people familiar with UML is can provide a route map into the MODEM real world semantics. From an audit perspective it helps to ensure completeness, that the functionality of UML State Machines has been captured in the MODEM model.

The relevant elements of Figure 38 - UML State Machines are mapped in earlier sections – the relevant mappings are:

UML StateMachines	MODEM Real World Semantics
StateMachine	State Machine Views
Region	DisjointStateTypesSets
State	OwnedStateSets
Transition (sub-typed)	stateSetTransition
/context: BehavioredClassifier	stateMachineStateTypeOwners

There are also the following UML StateMachine types that need to be mapped:

- UML Initial Pseudostates,
- UML Terminate Pseudostates, and
- UML Final State.

UML Initial and Terminate Pseudostates and Final State

UML uses Initial Pseudostate, Terminate Pseudostate and Final State elements to mark the start and end of behaviours (see the descriptions below). UML reifies these initial and final states, and, once this is done, uses its transition notation to represent the transitions to and from these states.

Entry and Exit ConnectionPointReferences (15.3.1 ConnectionPointReference – p. 544 – in the specification) can be regarded as initial and final states, when the state machine is considered as a sub-machine.

One of the purposes of these states is to mark the initial and final state – shown in the 'initial' row and 'final' column in the succession grid – see Figure 47. As the associated IDEAS diagram (Figure 46) shows, these are best interpreted as sub-types of the state types, rather than different types of states.

The initial pseudostate can be interpreted as the set of states that are the first state in the temporal ordering and the final state as the last state in the temporal ordering. This is shown graphically using arrows in the space-time map in Figure 12.

UML describes the initial pseudostate and a final state as follows.

> "Each region of a composite state may have an initial pseudostate and a final state. A transition to the enclosing state represents a transition to the initial pseudostate in

© 2011 BORO Solutions Limited. All Rights Reserved.

each region. A newly-created object takes its topmost default transitions, originating from the topmost initial pseudostates of each region. A transition to a final state represents the completion of behaviour in the enclosing region. Completion of behaviour in all orthogonal regions represents completion of behaviour by the enclosing state and triggers a completion event on the enclosing state. Completion of the topmost regions of an object corresponds to its termination." p. 566

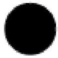


Figure 15.16 - Initial Pseudostate

Figure 62 - UML Initial Pseudostate Icon

UML describes the Final State object as follows.

> "A special kind of state signifying that the enclosing region is completed. If the enclosing region is directly contained in a state machine and all other regions in the state machine also are completed, then it means that the entire state machine is completed." p. 547
>
> "When the final state is entered, its containing region is completed, which means that it satisfies the completion condition. The containing state for this region is considered completed when all contained regions are completed. If the region is contained in a state machine and all other regions in the state machine also are completed, the entire state machine terminates, implying the termination of the context object of the state machine." p. 548

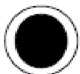


Figure 15.11 - Final State

Figure 63 - UML Final State Icon

UML can be seen as recognising that the Initial and Terminate Pseudostates are not states in the real world by its use of the prefix 'Pseudo-'. The real world semantics for Final State is more problematic if one takes the UML text literally. This implies that the Region enters the final state after the behaviour has completed – and, presumably stays in that state from then onwards. Technically this is similar to the presentist[3] notion of past – in other words; it marks the behaviour as in the past. This is not a state in the sense we have been using here, of a state that an object can be in, one of its temporal parts – as the state only comes into existence after the Region has ceased to exist. This raises concerns about mixing presentist

---

[3] See http://plato.stanford.edu/entries/time/#PreEteGroUniThe for more details on presentism and eternalism. IDEAS adopts an eternalist stance.

© 2011 BORO Solutions Limited. All Rights Reserved.

and eternalist foundations and having multiple inconsistent senses of state in close proximity.

In practical terms, it makes sense to keep things simple and straightforward and interpret Final State as the set of final or end state in the chain of states for each state machine.

Interaction pattern – real world analysis

We start the analysis by giving some context - by reviewing what UML interaction diagrams are.

UML Interaction Diagrams

In UML, an Interaction will have a number of lifelines. These lifelines will contain a number of ExecutionSpecifications. These will be ordered. ExecutionSpecifications will contain MessageOccurenceSpecifications which will be the send or receive end of a Message. The following figures contain the diagrams that describe this.

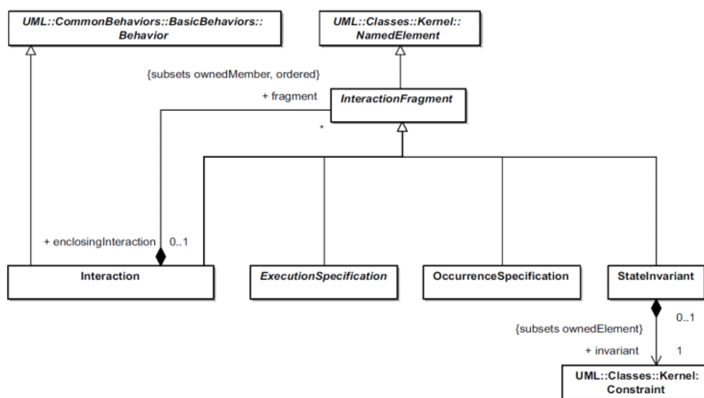


Figure 64 - UML Specification "Figure 14.3"

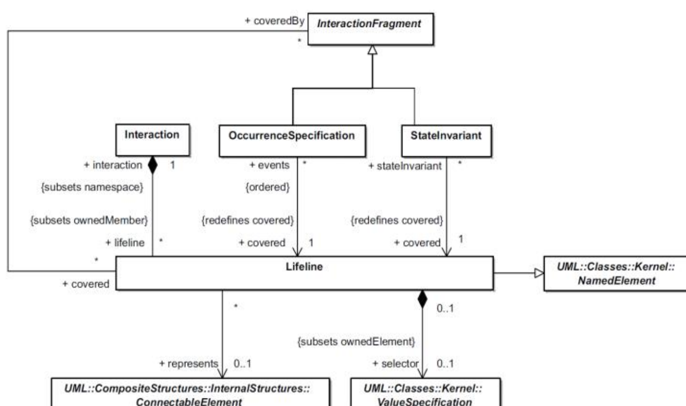


Figure 65 - UML Specification "Figure 14.4"

© 2011 BORO Solutions Limited. All Rights Reserved.

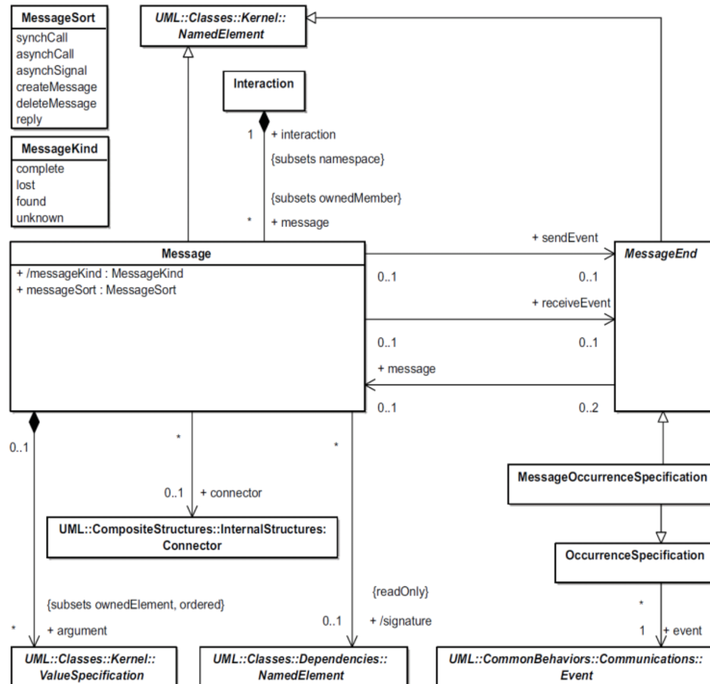


Figure 14.5 - Messages

Figure 66 - UML Specification "Figure 14.5"

Grounding the Interaction View analysis in 'Eat Restaurant Meal'

We grounded the analysis in the 'Eat Restaurant Meal' Interaction example. We identified the patterns initially in a particular instance of this: 'Fred Eating Meal in Restaurant - 21st Jan 2010'. We then generalised the pattern to the 'Eat Restaurant Meal' Interaction level and then again to an Interaction View level. We follow this analysis path here, starting with Participations in 'Eat Restaurant Meal'.

Participations in 'Eat Restaurant Meal'

In the 'Eat Restaurant Meal' Interaction example contains lifelines. The real world semantics for this, as shown in the space-time map in Figure 24 is that these lifelines are participations in the interaction – what MODEM calls interaction roles. Figure 67 shows these in IDEAS format for the Eat Restaurant Meal interaction. It makes explicit how parts of the participant are also parts of the interaction.

© 2011 BORO Solutions Limited. All Rights Reserved.

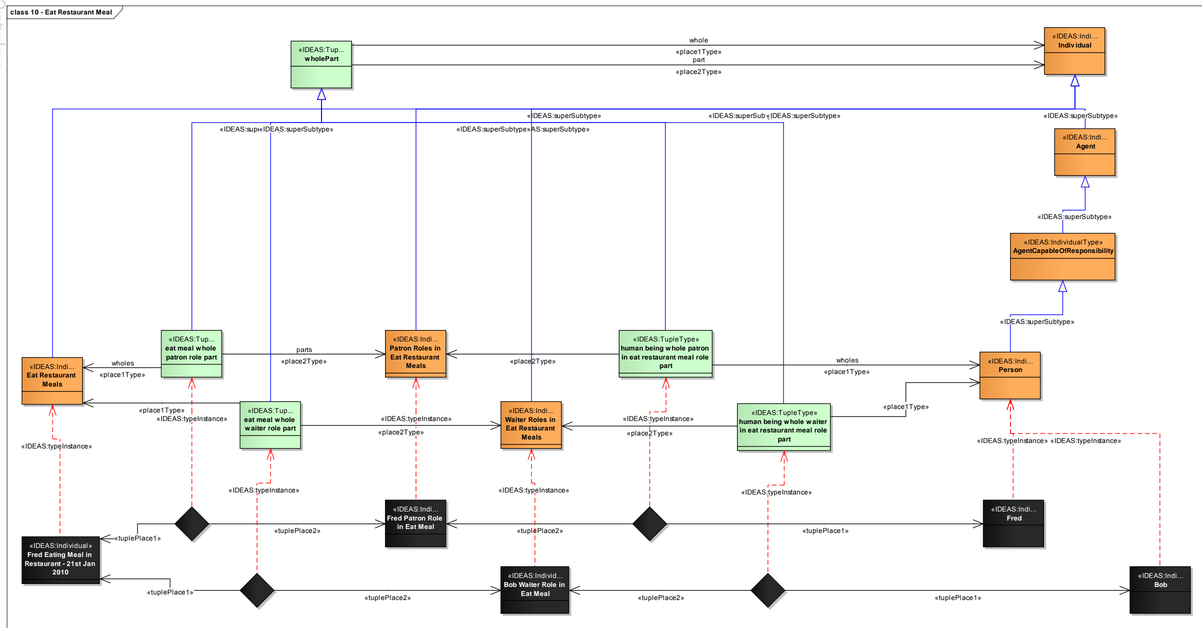


Figure 67 - Participation in Eat Restaurant Meal

Figure 68 shows this generalised and so the underlying pattern made explicit.

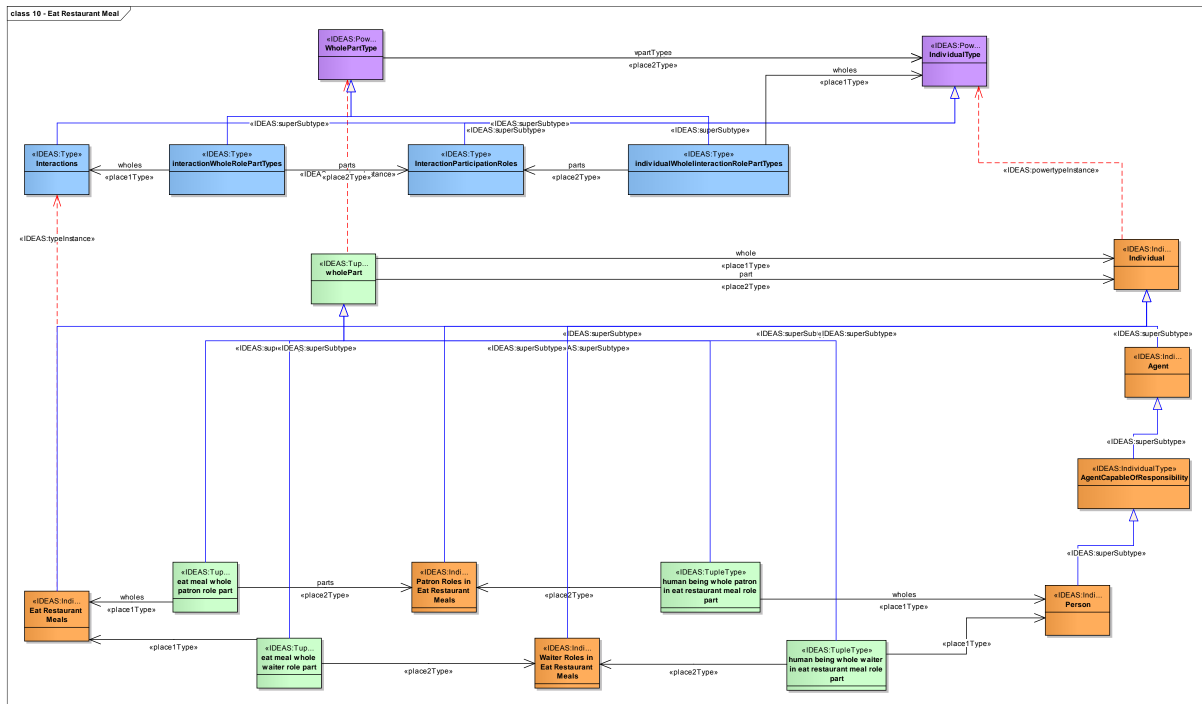


Figure 68 – Interaction Role - Participation pattern

‘Patron Roles in Eat Restaurant Meals State Machine View’

These Interaction Roles are also divided into states, which are part of a state machine. This is shown in IDEAS format in Figure 69 for the Patron role. This also shows how the collection of the states are a ‘DisjointStateTypesSets’ and part of a state machine view - the ‘Patron Roles in Eat Restaurant Meals State Machine View’.

© 2011 BORO Solutions Limited. All Rights Reserved.

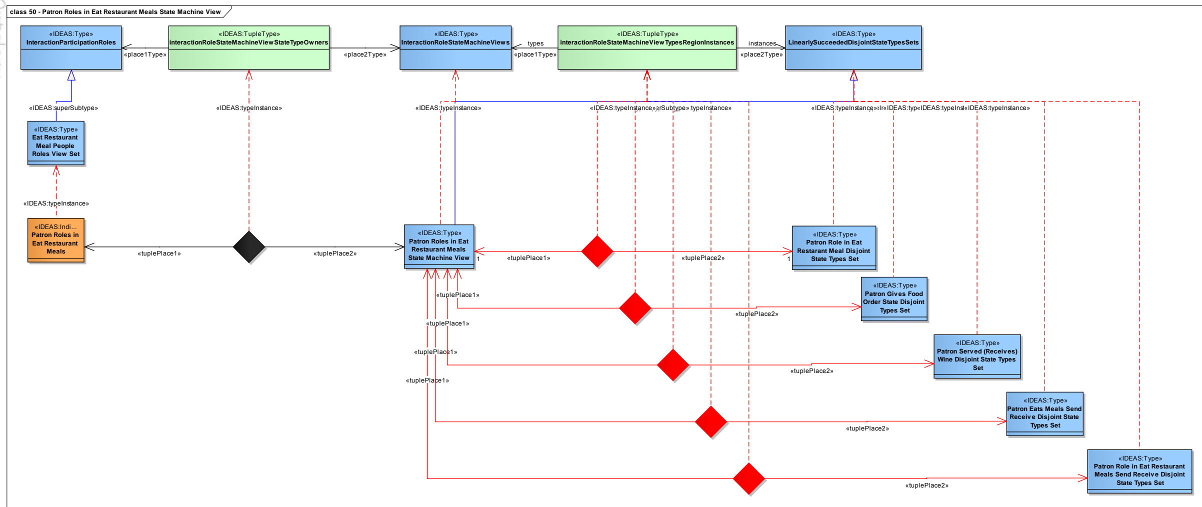


Figure 69 - Interaction Role States

The interaction role states are then further divided into instance-wise disjoint sub-states. In UML these sub-states are message ends – this is not a constraint in MODEM. Figure 70 shows this for 'Patron Orders Food' – this is a limit case where these is only one sub-state.

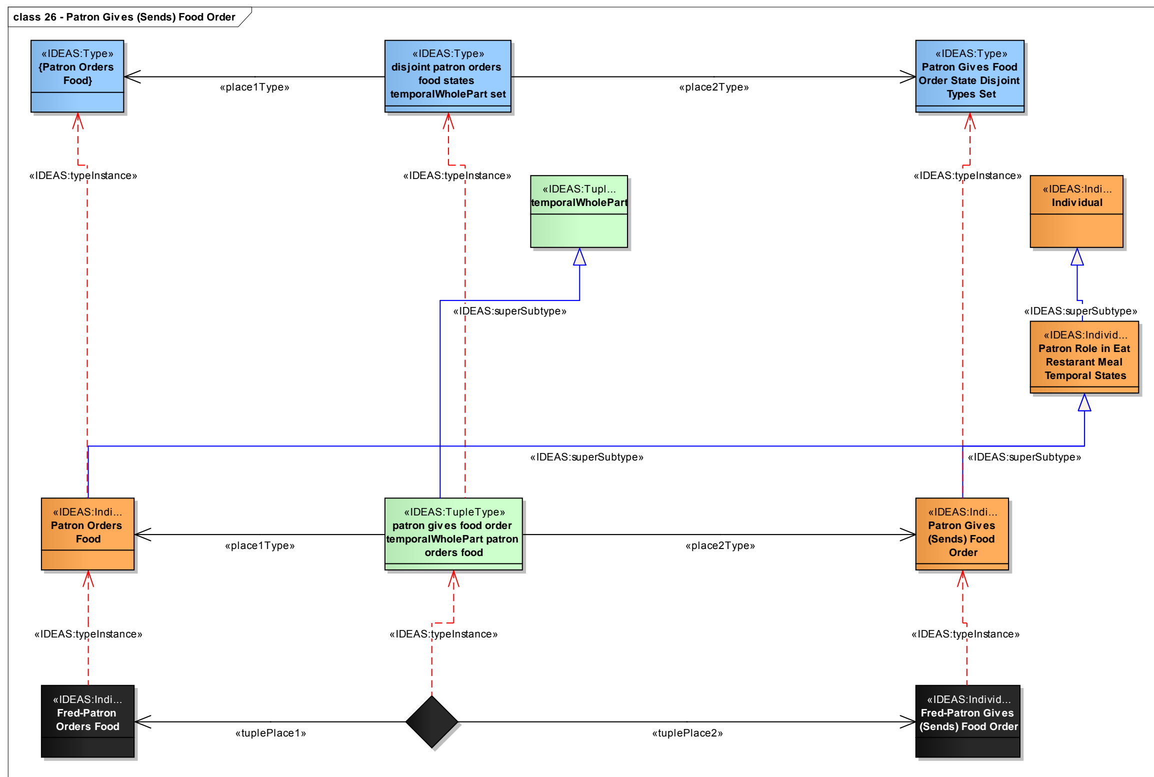


Figure 70 - 'Patron Orders Food' sub-states

'Patron Gives Food Order to Waiter' - MODEM Interaction Roles State Interactions

The 'Patron Gives (Sends) Food Order' in Figure 70 is the send end of a state interaction (message in UML –speak) that is received by 'Waiter Takes (Receives) Food Order' – the state interaction is 'Patron Gives Food Order to Waiter'. In the real world, the patron will say

© 2011 BORO Solutions Limited. All Rights Reserved.

something to the waiter or the waiter will pour wine or serve food. The IDEAS representation of this state interaction is shown in Figure 71.

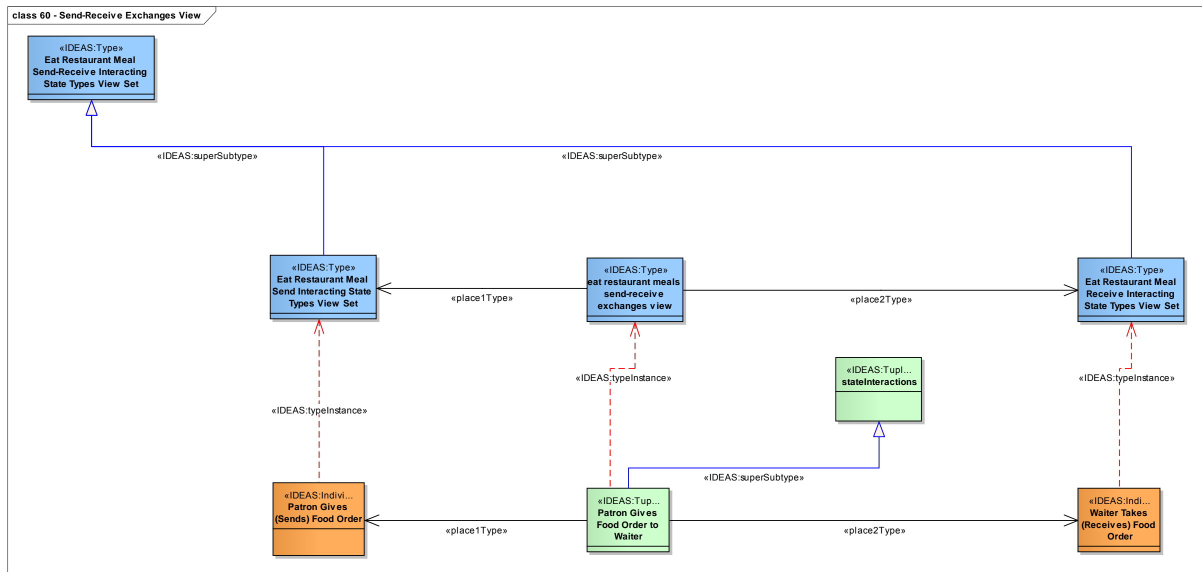


Figure 71 - State Interaction

'Eat Restaurant Meals People Role Interaction View' - MODEM Interaction Views

The 'Eat Restaurant Meals People Role Interaction View' contains the 'Eat Restaurant Meal People Roles View Set', which contains the interaction roles chosen for the view. It also contains the 'Eat Restaurant Meal Send-Receive Interacting State Types View Set', which contains the interaction state types chosen for the view. This is shown in Figure 72.

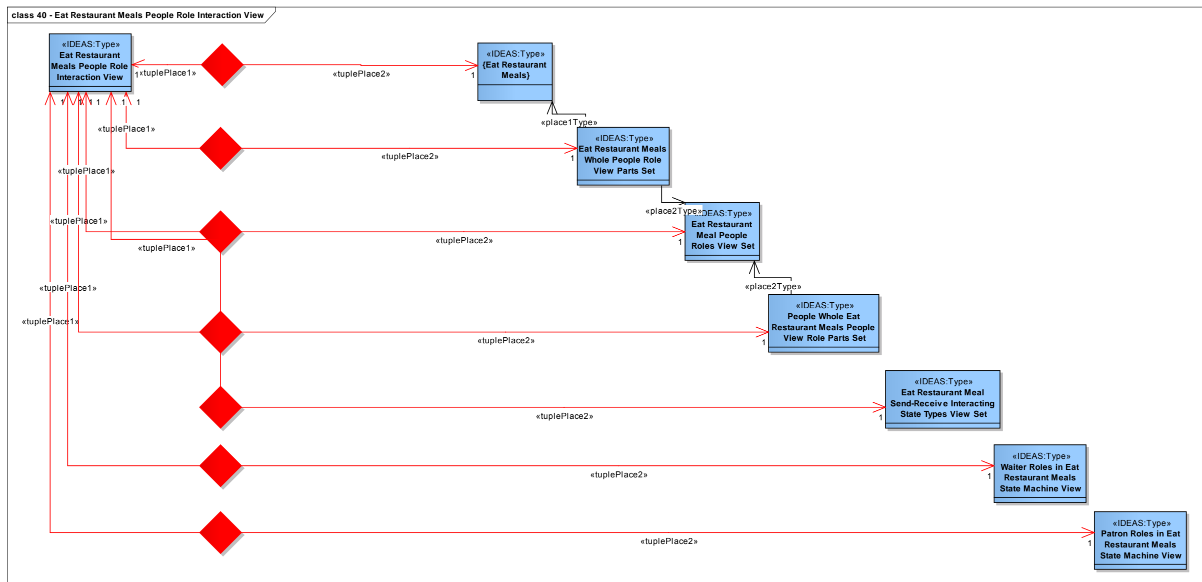


Figure 72 - 'Eat Restaurant Meals People Role Interaction View'

MODEM Interaction type level

To capture the semantics of the Interaction diagram, this example needs to be generalised up a type level. '{EatRestaurantMeals}' along with its components are taken up to

© 2011 BORO Solutions Limited. All Rights Reserved.

InteractionSingleton in Figure 73. ‘Eat Restaurant Meal Send-Receive Interacting State Types View Set’ is taken up to ‘SendReceiveInteractingStateTypesViewSet’ in Figure 74. ‘Eat Restaurant Meals People Role Interaction View’ is taken up to ‘InteractionView’ in Figure 75.

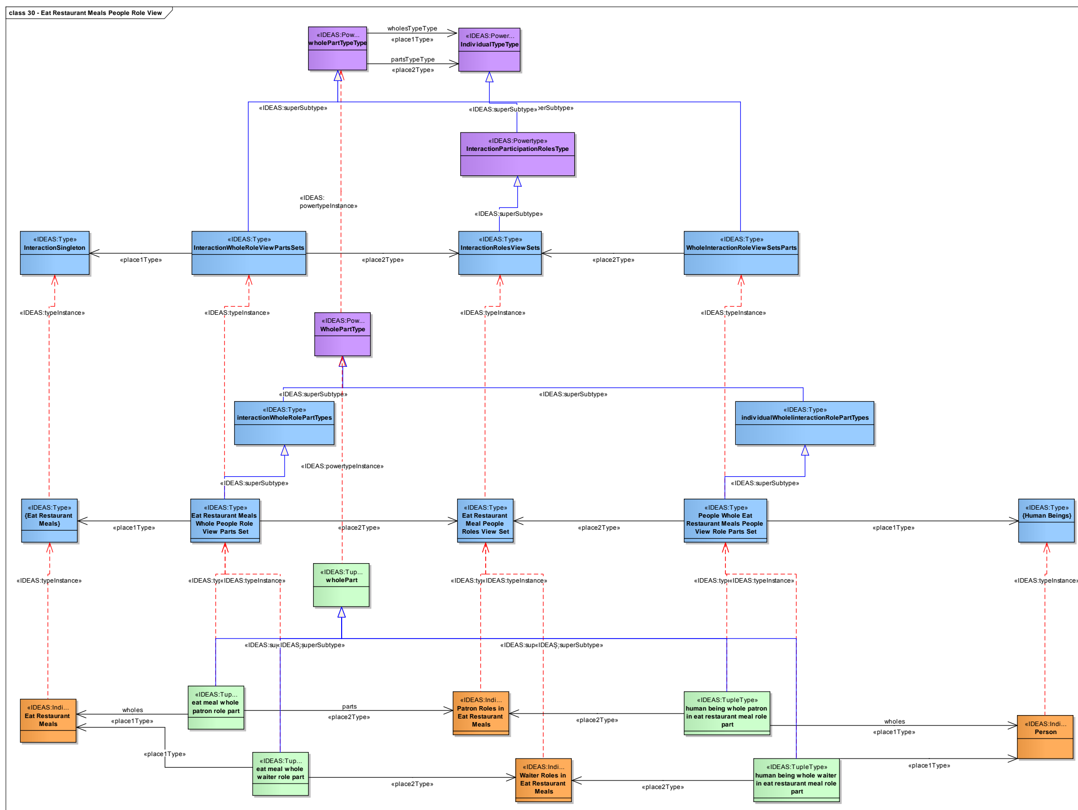


Figure 73 - Interaction Singletons

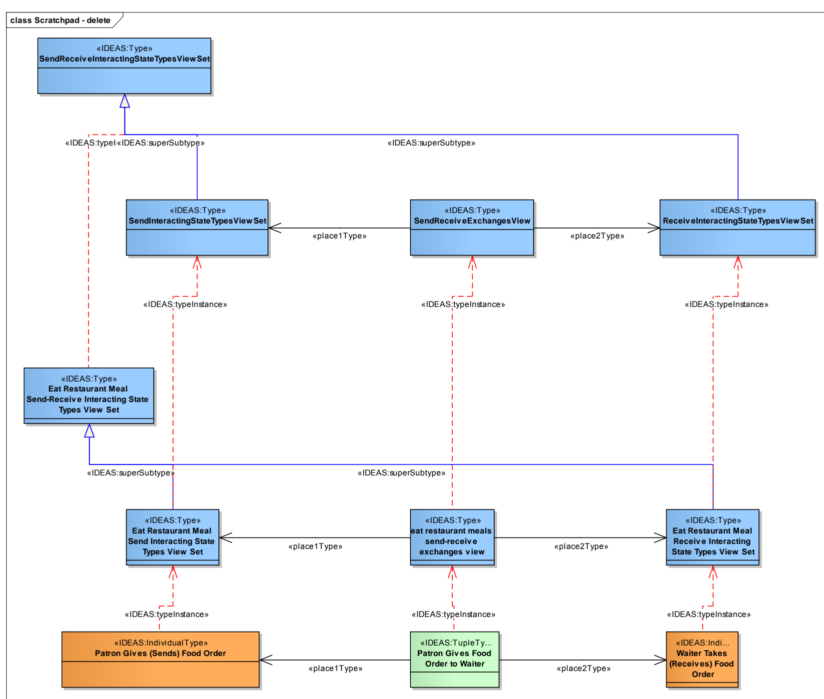


Figure 74 - SendReceiveInteractingStateTypesViewSet

© 2011 BORO Solutions Limited. All Rights Reserved.

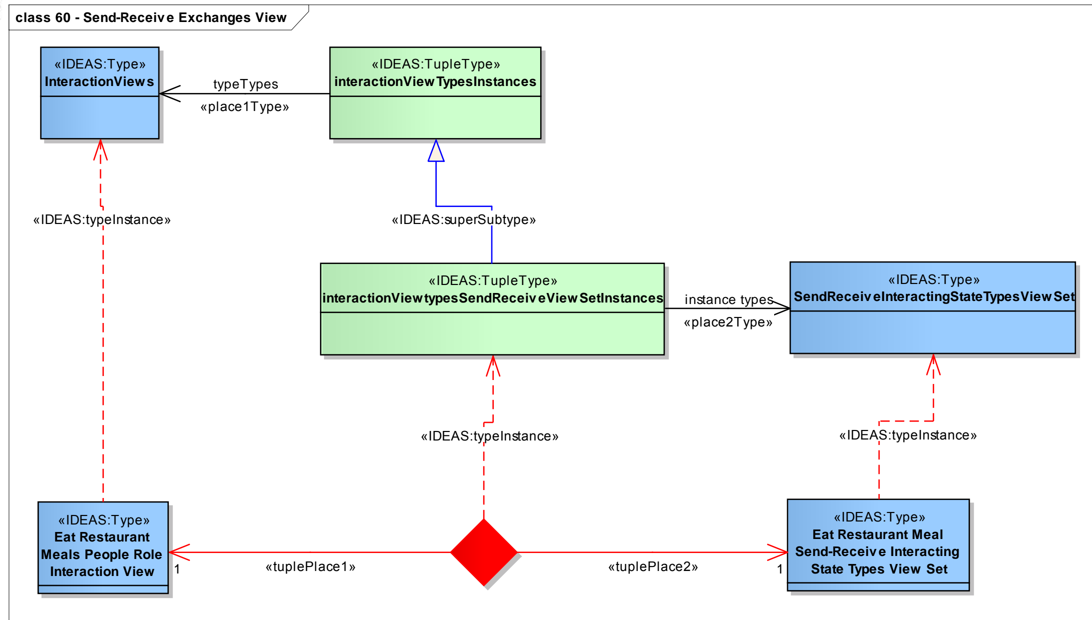


Figure 75 – InteractionViews and Eat Restaurant Meals People Role Interaction View

MODEM Interaction View and components

This type generalisation reveals a pattern with several components. Firstly, there is the 'InteractionView' that reflects the choices of what the view should contain, which interaction roles (lifelines) and then which state interactions (messages) between those roles.

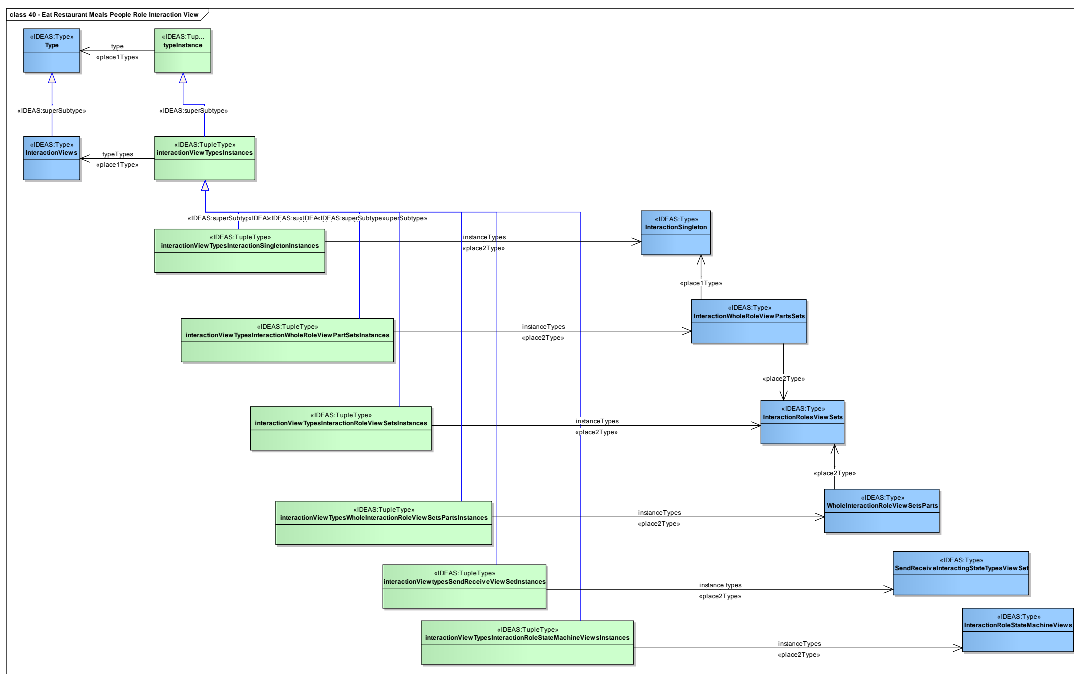


Figure 76 - InteractionViews

© 2011 BORO Solutions Limited. All Rights Reserved.

The ‘InteractionView’ contains the interaction participation components shown in Figure 77. It also contains the ‘InteractionRoleStateMachineViews’ for each Interaction Role, which contain the role state types selected for the view, shown in Figure 78. And the ‘SendReceiveInteractingStateTypesViewSet’, which contains the state interaction types selected for the view, shown in Figure 79.

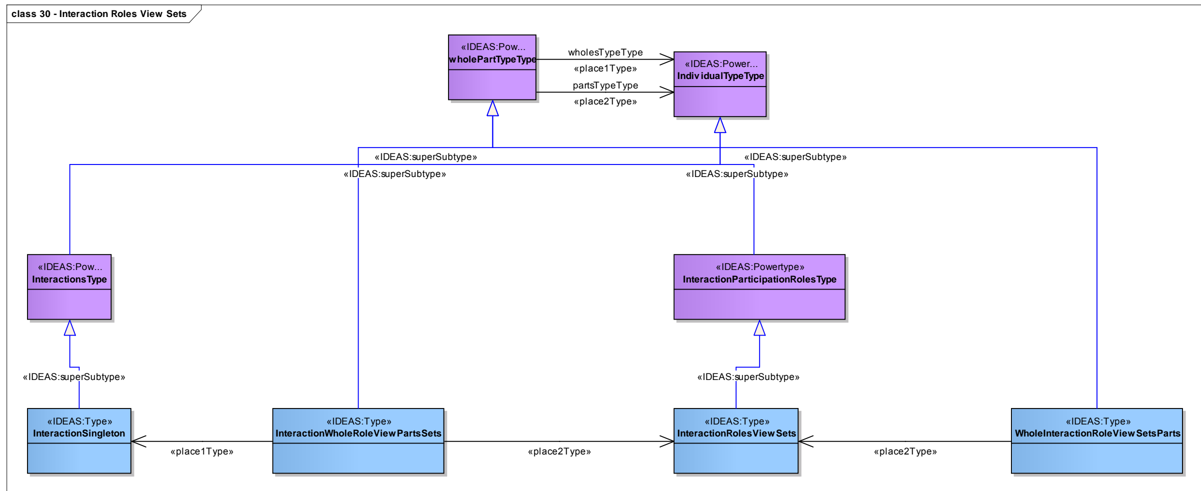


Figure 77 - Interaction View components

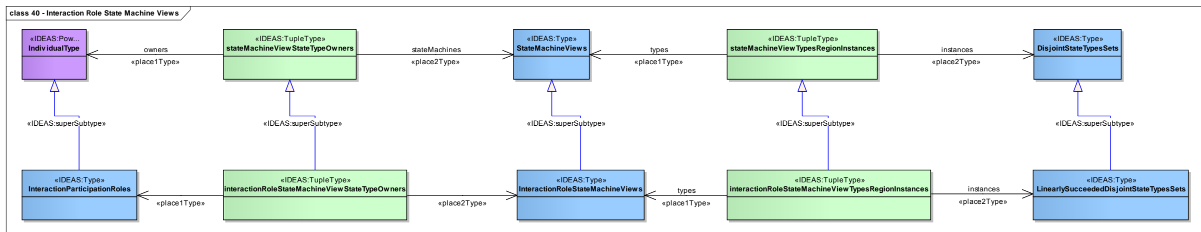


Figure 78 – InteractionRoleStateMachineViews

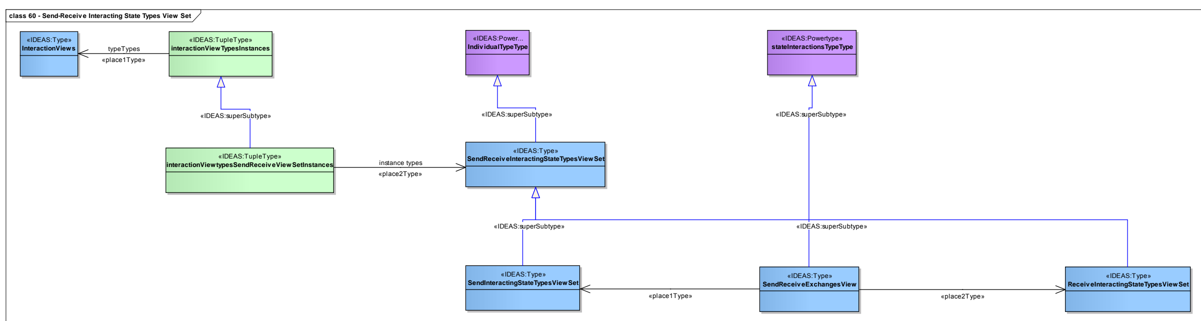


Figure 79 – SendReceiveInteractingStateTypesViewSet

Mapping the Real World Semantics back to the UML Interaction

As noted in the State Machine mapping, this can serve as a route map from UML into the MODEM real world semantics or as an audit tool to check completeness.

© 2011 BORO Solutions Limited. All Rights Reserved.

The relevant mappings are:

UML Interactions	MODEM Real World Semantics
Interaction	(InteractionView / Interaction)
Lifeline	Interaction Role
ExecutionSpecification	(Temporal State)
MessageOccurenceSpecification	(Temporal State)
Message	StateInteractionType

Brackets indicate the mappings are not direct.

UML Interaction

UML's description for an Interaction is: "An interaction is a unit of behavior that focuses on the observable exchange of information between ConnectableElements." Its semantics is: "Interactions are units of behavior of an enclosing Classifier. Interactions focus on the passing of information with Messages between the ConnectableElements of the Classifier."

The analysis shows that the UML Interaction encompasses both the Interaction itself (a unit of behaviour) and the Interaction View (the "observable exchange of information" from that view). There is no mechanism within UML to distinguish the two. Hence, if we were to consider the 'Eat Restaurant Meal' Interaction from different perspectives – for example, the food and wine perspective – then in UML this would be a different interaction. Whereas in the real world (and in MODEM), it is a different view over the same underlying interaction.

UML Lifeline

UML's description for a Lifeline is: "A lifeline represents an individual participant in the Interaction." Its semantics is: "The order of OccurrenceSpecifications along a Lifeline is significant denoting the order in which these OccurrenceSpecifications will occur. ... The semantics of the Lifeline (within an Interaction) is the semantics of the Interaction selecting only OccurrenceSpecifications of this Lifeline."

The UML lifeline directly corresponds to the MODEM Interaction Role. The analysis clarifies the mereology of the participation of something in the Interaction – it is the participating part of the partipant. The Interaction View chooses which Interaction Roles appear in the view.

UML ExecutionSpecification

UML's description for a Lifeline is: "An ExecutionSpecification is a specification of the execution of a unit of behavior or action within the Lifeline. The duration of an ExecutionSpecification is represented by two ExecutionOccurrenceSpecifications, the start ExecutionOccurrenceSpecification and the finish ExecutionOccurrenceSpecification." Its semantics is: "The trace semantics of Interactions merely see an Execution as the trace <start, finish>. There may be occurrences between these. Typically the start occurrence and the finish occurrence will represent OccurrenceSpecifications such as a receive OccurrenceSpecification (of a Message) and the send OccurrenceSpecification (of a reply Message)."

© 2011 BORO Solutions Limited. All Rights Reserved.

From a real world perspective, this is a state type of the lifeline (interaction role). Where the state types in a lifeline form a state machine view. The interaction view not only selects the lifelines, it selects which state types in the lifeline appear in the view. A different view may select the same lifeline but a different set of state types. In UML this would need to be a different interaction – in MODEM a different view over the same interaction.

UML MessageOccurenceSpecification
UML's description for a MessageOccurenceSpecification is: "Specifies the occurrence of events, such as sending and receiving of signals or invoking or receiving of operation calls. A message occurrence specification is a kind of message end. ..."

From a real world perspective, this is a state type of the ExecutionSpecification and hence an indirect state type of the lifeline (interaction role). Where the state types in the ExecutionSpecification are nested regions in the lifeline state machine view. The interaction view will select the state machine view, and so select the nested state types. A different view may select the same lifeline and ExecutionSpecification but a different set of nested state types. In UML this would need to be a different interaction – in MODEM a different view over the same interaction.

UML Message
UML's description for a Lifeline is: "A Message defines a particular communication between Lifelines of an Interaction.... A Message associates normally two OccurrenceSpecifications - one sending OccurrenceSpecification and one receiving OccurrenceSpecification. " Its semantics is: "The semantics of a complete Message is simply the trace <sendEvent, receiveEvent>...."

Communication is interpreted generously here. It reflects any kind of causal dependency between the interaction roles – for example, the pouring of wine or the serving of food. It is not restricted to the passing of information.

In UML the message associates a sending OccurrenceSpecification with a receiving OccurrenceSpecification . The real world interpretation of this, is of a sending state of one interaction role causing a receiving state in another (possibly the same) interaction role; in other words, of a causal relationship between these two states.

Summary

In this section, the real world analysis in the second section has been formalised into IDEAS MODEM models. In addition a mapping from UML to MODEM has been given. This has highlighted the real world semantic variation points.

Real world semantic variation points
One can understand these variation points better if they are put into context. Both UML's State Machine and Interaction diagram structures are creatures of their history. The two were included in UML, but not integrated which led to some of the variation we have found.

Furthermore, the StateMachine/Region structure is a result of its development history. It started out as an extension to the Harel Statechart with various features added over time, including orthogonal regions. It was also strongly influenced by the requirement for an

© 2011 BORO Solutions Limited. All Rights Reserved.

executable behaviour specification. The following comments need to be read with this in mind.

Diagram versus repository-centric architecture

Broadly speaking, UML State Machines and Interactions partition behaviour into stove piped silos. States in one partition cannot appear in the other partition. Furthermore, these partitions are separated from other UML structures, that mean State Machines, Interactions and States cannot, for example, be generalised or specialised.

Within State Machines, StateMachines and Regions further partition behaviour into smaller silos. Each StateMachine specifies the behaviour in its silo. Within the StateMachine silos, behaviour is further partitioned into regions. One consequence of this is (as noted earlier) that each state can only appear in each Regionsilo, and so each StateMachine silo once.

In the MODEM architecture, there are not these partitions. For example, the underlying state succession behaviour is a pattern within a much more closely inter-linked network of behaviour.

Another way of explaining this is that UML has a diagram-centric approach, where each diagram is a direct representation of a segmented part of a repository. MODEM has a diagram-as-view approach, where the view (state machine interaction) looks at a type of pattern in the underlying repository – and the same objects can appear in a number of views.

Arbitrary structures - from the perspective of real world semantics

As earlier comments in the text have indicated, UML's StateMachine contains arbitrary structures (from the perspective of real world semantics). We looked at an example (Figure 4) where one can model part of an enterprise as a single state machine, with several regions or several state machines each with one region or any combination in between – and each modelling choice excludes the other choices. It is clearly pointless to ask which of these better reflects the real world, as this is not the point. The differences are just modelling structures with no real world equivalents, they are (from a real world perspective) arbitrary. On the other hand, MODEM has a clear real world semantics, which reflects real patterns in the world.

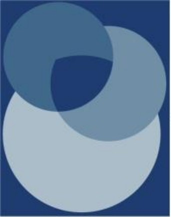
Artificial constraints

UML's partitioning approach introduces some artificial constraints. One example is the specialisation of state machines with a restricted set of states in one example – illustrated in Figure 5 and Figure 52. Another is the individual sub-type's hierarchy of 'DisjointStateTypesSets' illustrated in Figure 48. While there are more or less convoluted workarounds to these constraints, the result is a distorted picture of the real world.

Modelling framework trade-offs

The choice of a modelling framework involves trade-offs – and these will depend upon the requirements. While UML has an overriding requirement for an executable behaviour specification, EA has a strong requirement for a clear real world semantics. The different requirements will lead to different assessments of the benefits and costs in the trade-off – and so different rational decisions. EA's requirement for a clear real world semantics is

© 2011 BORO Solutions Limited. All Rights Reserved.



directly reflected in the MODEM structures. Whether UML's requirements explain and justify their choices on these issues is outside the scope of this work.

In addition, UML was designed without the help of a top ontology. Introducing a top ontology changes the shape of the trade-offs. Further work may well show that there is no need to choose between a clear real world semantics and an executable behaviour specification. Indeed, it may turn out that a clear real world semantics is a better basis for an executable behaviour specification.

© 2011 BORO Solutions Limited. All Rights Reserved.

Appendices

Appendix A – MODAF UML Behaviour Scope

Appendix B – State diagram as a mathematical structure – example definition

Appendix C – State machines as a formal structure in the UML Superstructure Specification

© 2011 BORO Solutions Limited. All Rights Reserved.

Appendix A – MODAF UML Behaviour Scope

This appendix provides an overview of the current uses of UML Behaviour in MODAF.

This is a summary of the four views that use UML Behaviour

View	Name	Used for	Data objects
OV-6b	Operational State Transition Description	Analysis of business events. Behavioural analysis. Identification of constraints (input to SRD).	States (each associated with a mission, node or operational activity.) State transitions (each associated with an event).
OV-6c	Operational Event-Trace Description	Analysis of operational events. Behavioural analysis. Identification of non-functional user requirements (input to URD). Operational test scenarios.	Lifelines (each associated with a Node).
SV-10b	Resource State Transition Description	Definition of states, events and state transitions (behavioural modelling). Identification of constraints (input to System Requirements Document).	Resources. States (associated with a resource or function). State transitions (each associated with an event)
SV-10c	Resource Event Trace Description	Analysis of resource events impacting operation. Behavioural analysis. Identification of non-functional system requirements (input to System Requirement	Lifelines (each associated with a functional resource or a system port).

© 2011 BORO Solutions Limited. All Rights Reserved.

Document).

From http://www.mod.uk/NR/rdonlyres/E631B417-8FB1-4D0D-AE96-6065AED1345F/0/20090216ViewsummaryU.pdf - © Crown Copyright 2004-2010

The following are examples of the views provided by the MOD (© Crown Copyright 2004-2010).

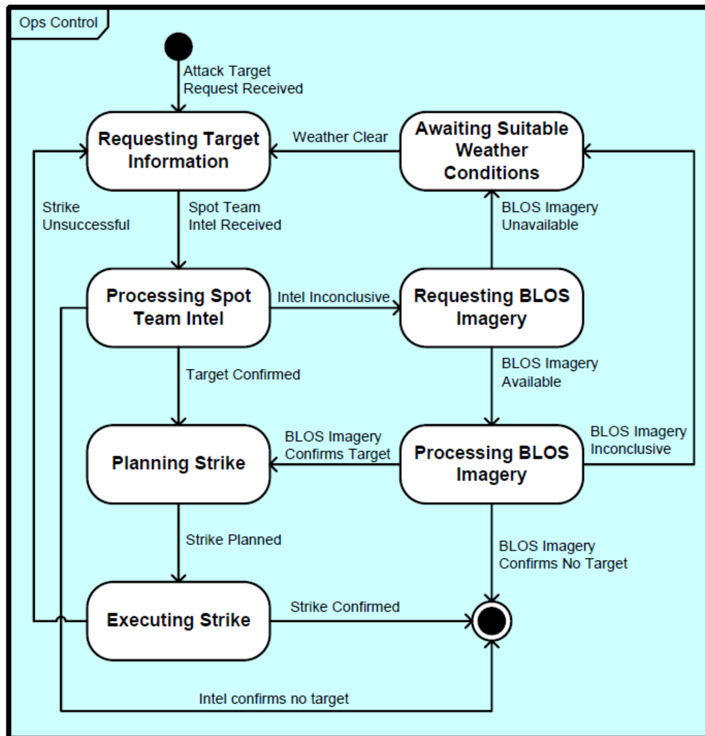


Figure 80 - OV-6b example

© 2011 BORO Solutions Limited. All Rights Reserved.

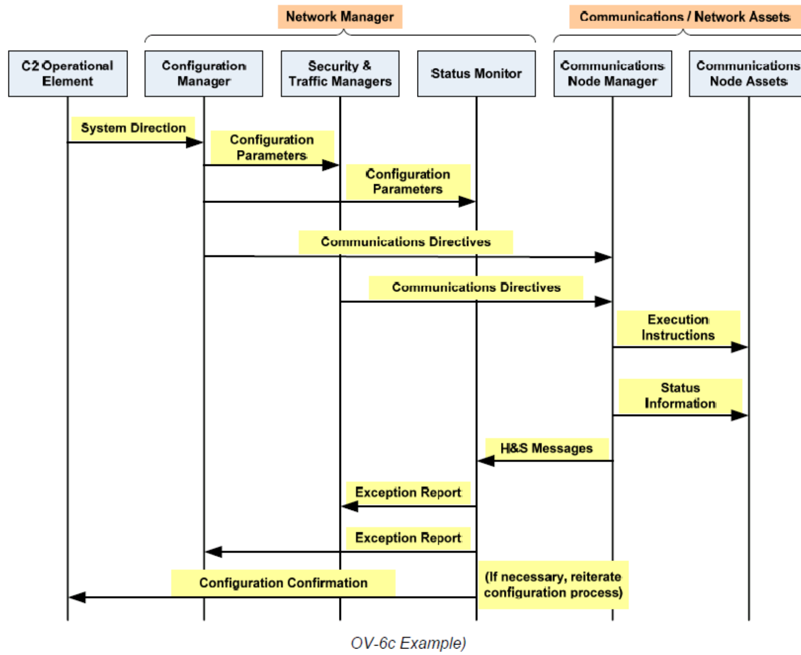


Figure 81 - OV-6c example

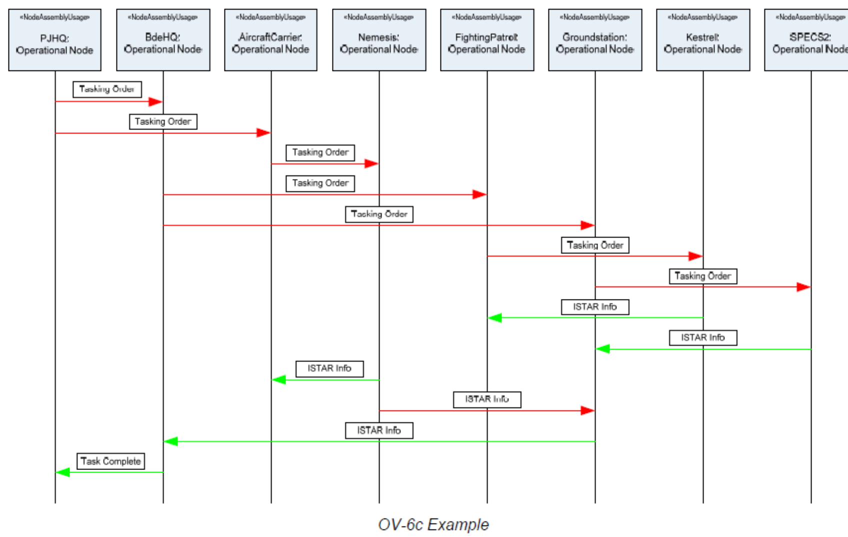


Figure 82 - OV-6c example

© 2011 BORO Solutions Limited. All Rights Reserved.

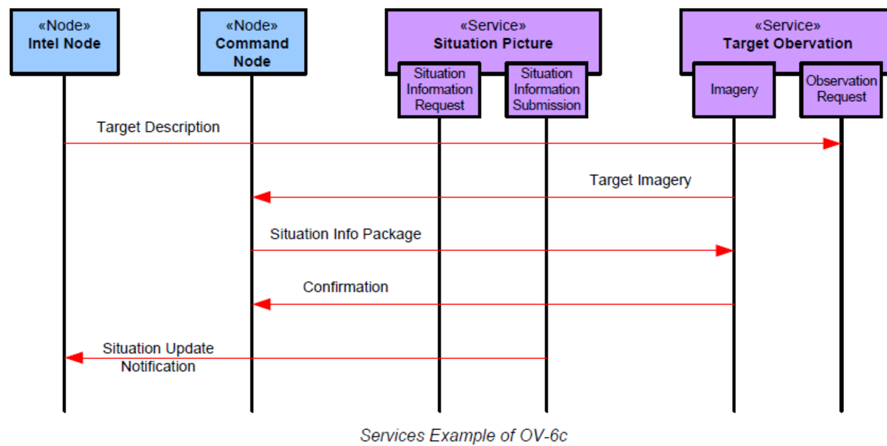


Figure 83 - OV-6C - Services example

From http://www.mod.uk/NR/rdonlyres/97A0CD8D-3673-4DF4-A754-0AFB5CD010BB/0/20100426MODAFOVViewpoint1_2_004U.pdf - © Crown Copyright 2004-2010

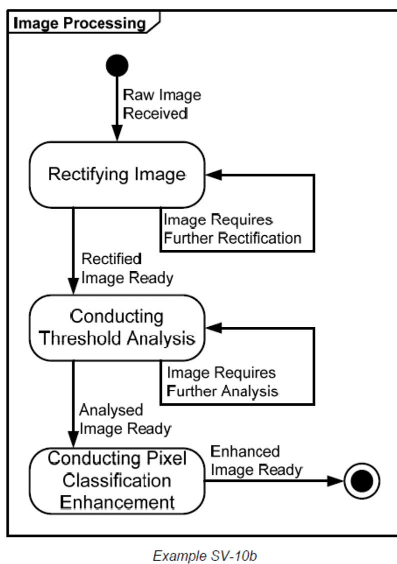


Figure 84 - SV-10b example

© 2011 BORO Solutions Limited. All Rights Reserved.

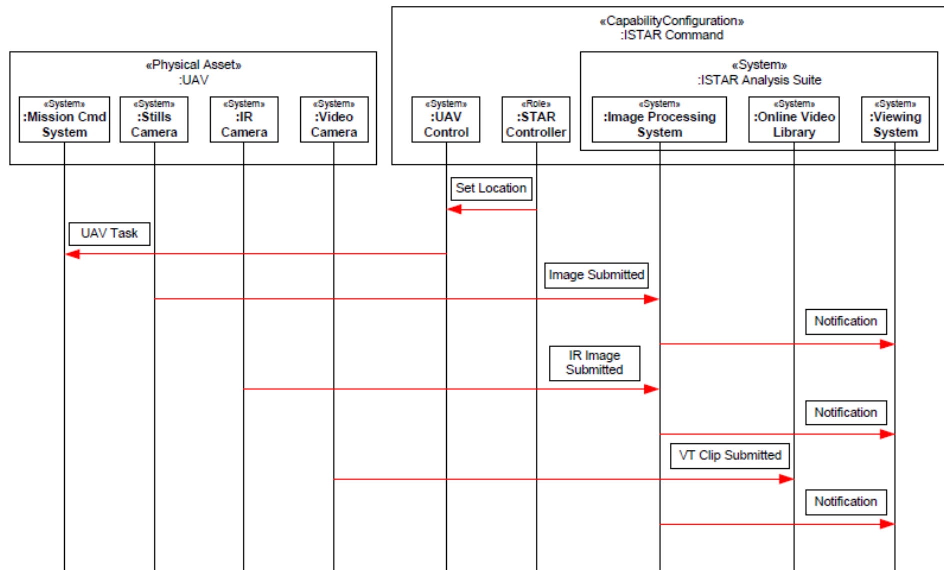


Figure 85 – Systems Event-Trace Description example

From http://www.mod.uk/NR/rdonlyres/8FF14D0F-90DB-4CC8-B8B1-1D03538A478F/0/20100426MODAFSVViewpointV1_2_004U.pdf. - © Crown Copyright 2004-2010

© 2011 BORO Solutions Limited. All Rights Reserved.

Appendix B – State diagram as a mathematical structure – example definition

A classic form of a state diagram for a finite state machine is a directed graph with the following elements (Q, Σ, Z, δ, q_0 ,F):

- States Q: a finite set of vertices normally represented by circles and labelled with unique designator symbols or words written inside them;
- Input symbols Σ: a finite collection of input symbols or designators;
- Output symbols Z: a finite collection of output symbols or designators;

The output function ω represents the mapping of ordered pairs of input symbols and states onto output symbols, denoted mathematically as $\omega : \Sigma \times Q \rightarrow Z$.

- Edges δ: represent the "transitions" between two states as caused by the input (identified by their symbols drawn on the "edges"). An 'edge' is usually drawn as an arrow directed from the present-state toward the next-state. This mapping describes the state transitions that is to occur on input of a particular symbol. This is written mathematically as $\delta : \Sigma \times Q \rightarrow Q$
- Start state q_0: The start state $q_0 \in Q$ is usually represented by an arrow with no origin pointing to the state. In older texts, the start state is not shown and must be inferred from the text.
- Accepting state(s) F: If used, for example for accepting automata, $F \in Q$ is the accepting state. It is usually drawn as a double circle. Sometimes the accept state(s) function as "Final" (halt, trapped) states.

See:

Taylor Booth (1967) Sequential Machines and Automata Theory, John Wiley and Sons, New York.

John Hopcroft and Jeffrey Ullman (1979) Introduction to Automata Theory, Languages, and Computation, Addison-Wesley Publishing Company, Reading Mass, ISBN 0-201-02988-X

© 2011 BORO Solutions Limited. All Rights Reserved.

Appendix C – State machines as a formal structure in the UML Superstructure Specification

The UML Specification for the most part treats state machines as a formal structure. Where a real world semantics is discussed, this is often done informally. The extracts below from the UML Superstructure Specification and a wiki entry describing UML State Machines are intended to give a flavour of this formal description – and the paucity of real world semantics.

Extracts

> Description
>
> A region is an orthogonal part of either a composite state or a state machine. It contains states and transitions.

p. 563 - Section 15.3.10 Region (from BehaviorStateMachines) - UML Superstructure Specification, v2.3

> Notation
>
> A composite state or state machine with regions is shown by tiling the graph region of the state/state machine using dashed lines to divide it into regions. Each region may have an optional name and contains the nested disjoint states and the transitions between these. The text compartments of the entire state are separated from the orthogonal regions by a solid line.

p. 565 - Section 15.3.10 Region (from BehaviorStateMachines)

> Composite state
>
> A composite state either contains one region or is decomposed into two or more orthogonal regions. Each region has a set of mutually exclusive disjoint subvertices and a set of transitions. A given state may only be decomposed in one of these two ways. In Figure 15.35 on page 575, state CourseAttempt is an example of a composite state with a single region, whereas state "Studying" is a composite state that contains three regions

p.566 - Section 15.3.11 State (from BehaviorStateMachines, ProtocolStateMachines) - UML Superstructure Specification, v2.3

> 15.3.12 StateMachine (from BehaviorStateMachines)
>
> State machines can be used to express the behavior of part of a system. Behavior is modeled as a traversal of a graph of state nodes interconnected by one or more joined transition arcs that are triggered by the dispatching of series of (event) occurrences. During this traversal, the state machine executes a series of activities associated with various elements of the state machine.
>
> ...
>
> Description
>
> A state machine owns one or more regions, which in turn own vertices and transitions.

© 2011 BORO Solutions Limited. All Rights Reserved.

The behaviored classifier context owning a state machine defines which signal and call triggers are defined for the state machine, and which attributes and operations are available in activities of the state machine. Signal triggers and call triggers for the state machine are defined according to the receptions and operations of this classifier.

As a kind of behavior, a state machine may have an associated behavioral feature (specification) and be the method of this behavioral feature. In this case the state machine specifies the behavior of this behavioral feature. The parameters of the state machine in this case match the parameters of the behavioral feature and provide the means for accessing (within the state machine) the behavioral feature parameters.

A state machine without a context classifier may use triggers that are independent of receptions or operations of a classifier, i.e., either just signal triggers or call triggers based upon operation template parameters of the (parameterized) statemachine.

p.579 - Section 15.3.12 StateMachine (from BehaviorStateMachines)- UML Superstructure Specification, v2.3

To relate this concept to programming, this means that instead of recording the event history in a multitude of variables, flags, and convoluted logic, you rely mainly on just one state variable that can assume only a limited number of a priori determined values (e.g., two values in case of the keyboard). The value of the state variable crisply defines the current state of the system at any given time. The concept of state reduces the problem of identifying the execution context in the code to testing just the state variable instead of many variables, thus eliminating a lot of conditional logic. Moreover, switching between different states is vastly simplified as well, because you need to reassign just one state variable instead of changing multiple variables in a self-consistent manner.

States - http://en.wikipedia.org/wiki/UML_state_machine

© 2011 BORO Solutions Limited. All Rights Reserved.