The PRAKTOR Metamodel: Mobile Agents as Executable Parametric Object Petri Nets

DIMITRIS MOSTROUS, HABIB GORAINE
School of Computing
University of Central England
Perry Barr, Birmingham, B42 2SU
UNITED KINGDOM
d.mostrous@blueyonder.co.uk habib.goraine@uce.ac.uk

Abstract: Mobile agents have emerged as an attractive paradigm for the conceptualization and realisation of massively distributed and autonomous systems. We describe an original meta-model that can be used to design scalable, highly concurrent multiagent system architectures potentially consisting of very large numbers of fully mobile, persistent agents of arbitrary granularity and complexity. The model consists of an agent modelling language based on High Level Petri Nets and a functional reference implementation that enables the direct reification of design artefacts into executable mobile agent automata.

Key-Words: Mobile Agents, Multiagent Systems, High Level Petri Nets, Persistence, Migration, Distributed Systems, Agent Automata, Model-driven development

1 Introduction

Systems populated by autonomous mobile agents are a prominent example of massively concurrent architectures, where computing entities are under continuous execution which is frequently in conflict and cooperation, and communication is inherently asynchronous [12]. Also, individual agent behaviour can frequently be thought of as comprising of structures that resemble *task control workflows*, given that by definition agents exist with some purposeful *role*, *plan* or *task* encoded into their structure [4]. Thus, with regard to *agent systems modelling*, it would be highly beneficial if we could use a formalism that has the modelling capacity to describe all of the above concepts in an intuitive and at the same time unambiguous way. *High Level Petri Nets* (HLPNs) lend themselves as such a formalism that also seems to have future in terms of integration of formal methods for system analysis and verification [8].

We describe PRAKTOR, an original mobile agent *meta*-architecture introduced in [14], and we outline the concrete elements that enable the reification of agent design artefacts into equivalent, executable Petri net automata. Moreover, we describe the proposed inter-agent communication mechanisms and explain how PRAKTOR attacks the fundamental problem of *agent migration* employing what we will term *hierarchical strong mobility*. The name PRAKTOR originates from the homonymous Hellenic noun which directly translates to the English noun *agent*.

2 Background and Motivation

An agent can be though of as a *situated* entity that *perceives* and may *act upon* its environment in an *autonomous*, *goal-driven* fashion [13]. In addition, a mobile agent is capable of *migrating* to remote agent environments during its execution. More elaborate definitions can be found in [4].

2.1 Characteristics of Intelligent Agents

Although there is no global consensus on a single definition, a software entity that employs certain widely accepted properties would normally be pronounced an intelligent agent. In particular, we expect that an intelligent agent would exhibit the following characteristics [4, 12, 13]:
(a) *Autonomy*: an agent is (semi-)autonomous, and acts in accordance with its goals, normally without user intervention. It should hold a considerable degree of control over its execution and its resource management.
(b) *Flexibility*: An agent should be able to dynamically select simple or composite actions to perform in reply to the perceived state of the external environment.
(c) *Execution independence*: Agents should operate asynchronously and independently to other agents and processes.
(d) *Temporal continuity*: an agent is a continuously running entity with a medium to wide lifespan - it should not be degraded to a mere one-off task.

(e) *Location awareness*: Agents are aware of their location. However, this does not adversely affect the perceived levels of location transparency but rather provides with the additional benefits of locality.
(f) *Social ability*: Agents may *communicate*, *collaborate*, and possibly *negotiate* with other agents and humans in pursuit of their goals.

2.2 Mobile Agents as High Level Petri Nets

Petri Nets [10] are mathematical abstractions that have emerged as an important tool for the modelling and analysis of interacting, concurrent components [9]. Therefore, considering their *modelling power*, we have concluded that in order to define an agent modelling environment atop, it should be enough to utilise extensions that can provide more *modelling convenience*.

The Petri nets that we will be using as the base in PRAKTOR are usually referred to as *High Level Petri Nets* (HLPN), also commonly known as *Coloured Petri Nets* (CP-Nets) [7, 8]. It is assumed that the reader is familiar with the basics, thus we shall limit ourselves to reminding that in CP-Nets tokens are instances of data types, rather than indistinguishable markers [8]. CP-Nets are a concise modelling language compared to ordinary Petri nets with which they still remain equivalent (i.e., *convertible*) in the same way high-level programming languages are equivalent to machine language but are more suitable for practical systems development [8].

We propose a class of such High Level Petri nets as the means by which we can model agents that can exhibit the characteristics mentioned in §2.1. Petri nets are also executable; it is possible, given an initially *marked* net, to simulate the behaviour of that net by applying a set of execution rules – the *next state function*. These rules define the number and actual distribution of tokens in a Petri net, i.e. the *state* of the net [9]. Execution occurs in *atomic* steps, which are known as the *firing* of transitions. The rationale behind using the formalism of HLPNs is further enforced by three factors: first, the inherent ability of Petri nets to explicitly represent control flow intermixing parallelism, mutual exclusion, iteration, conditionality, and sequential behaviour; second, the potential analysability (*decidability*) of Petri nets against various dynamic properties such as *liveness*; thirdly, the intuitive graphical nature of Petri nets.

We define a *PRAKTOR Agent* as "an autonomous software component that has fully encapsulated data, methods, and architecture, and that has full control of its logical thread(s) of control that it uses for the sole purpose of fulfilling its embedded logic, which is internally encoded in the form of an executable *High Level Petri Net Automaton*. It may communicate only through *asynchronous message passing*."

2.3 The Object Oriented Perspective

PRAKTOR CP-nets, their constituent elements, and the data tokens they hold, are by definition instances of appropriate object-oriented *abstract data types*. This allows the possibility of having CP-nets as the *data tokens* of a CP-net's places. The latter capability can be found in the literature wherein such nets are usually termed *Reference Nets* [1]. An immediate consequence of the aforesaid property is that *tokens* not only represent values but may also have methods that can be invoked upon them during execution.

We further impose that PRAKTOR nets are *strongly typed* and thus, a place can only be eligible to hold multisets of instances over its assigned object type and its subtypes, rather than data tuples as in CP-Nets [8]. An implementation *must* enforce *type safety*, either by virtue of native *parametric polymorphism* support, or by runtime type consistency checks. Ideally, a model can be statically verified before being transliterated to executable format by the target language compiler. PRAKTOR nets can thus be labelled *Executable Parametric Object Petri nets*.

3 PRAKTOR Modelling Language

We have adopted a series of well-studied extensions to Petri nets [see 9, 8] in order to achieve the necessary level of abstraction required for practical agent design. These include *inhibitory* and *test* arcs, together with *transition priorities* [9]; *reserve* arcs that do not consume their tokens; and *total capacity* restrictions on places [1]. These will be introduced through our description of the PRAKTOR Modelling Language (PML).

3.1 Basic Agent Net Building Blocks in PML

We present the main structural elements of the PML, along with explanations on notation. The reader should note that PML does not use standard CP-Net notation, as can be seen in the Bootstrap Scheduler of agent nets (Fig. 1).

3.1.1 Transitions and Transition Guards

Transitions represent a computation step of arbitrary granularity graphically denoted as a solid bar, and labelled with a short verb phrase that describes the task. Additionally, an extra identifier of the general format "(T*n*)", where $n \in \aleph^+$, can prefix the description, like e.g. "*(T2) Execute scheduler command*" (see Fig. 1).

Optionally, a Boolean expression can be attached to a transition, and specify that we only accept bindings (inputs) for which it evaluates to true [8]. The input to that expression is the set of multisets produced by evaluating the arc expressions (or weights) of all input arcs of the current transition (see §3.1.3). Such expressions are commonly termed *transition guards* [8] and in PML, if the result is positive, they provide the new bindings (i.e., *sub-bags* of the evaluated token multisets) that may be used as the *enabling tokens* (i.e. actual input tokens) of the transition, if it fires. An integer $p \in [0,255]$ can be shown inside a transition to denote its priority.

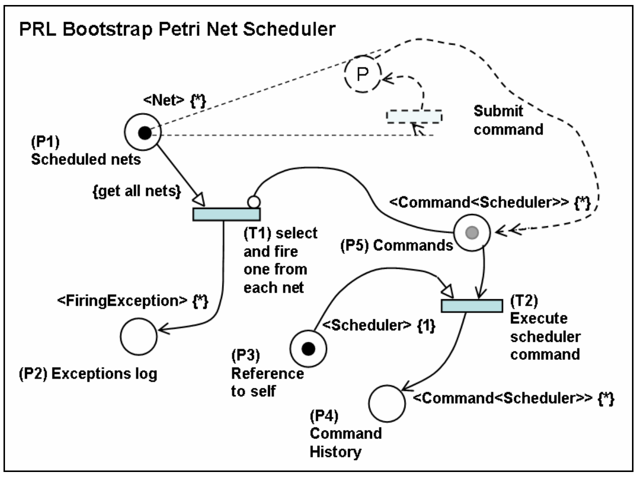

Fig. 1 PRAKTOR Runtime Library (PRL) scheduler net, which may be used to execute any number of agent nets.

3.1.2 Places

Places act as input and/or output buffers to one or more transitions, holding the data *tokens* of the agent being modelled. By convention, places are drawn as circles or ellipses, and in PML we annotate them with three pieces of information.

Firstly, places are *required* to have an annotation of the format <*T*> where the *less than* and *greater than* symbols always surround a *type name*, abstractly denoted *T* in this example. Notice that we do not define the internal structure of type name *T*, therefore nested type parameterisation is possible if the target implementation language supports parametric polymorphism.

The optional parts of information are the *total capacity* [1] and the *place name*. *Total capacity* is shown in the format *{k}* where *k* denotes the upper limit, and $k \in \aleph^+$. A special case exists whereby the wildcard '*' can be used instead of a positive number for *k*, in which case the interpretation is *{+∞}*. The requirements on names are relaxed and in general the guideline is that place names should normally consist of a descriptive *plural noun phrase*, and when required, they could be annotated with an extra prefix "(P*n*)", where $n \in \aleph^+$, (e.g. "*(P1) Scheduled nets*" – also see Fig. 1). Normally, place types and integer capacities are placed next to each other adjacent to the first hemisphere of their associated place, and place names adjacent to the third, for consistency.

3.1.3 Arcs and Arc Expressions

Transitions, which represent computation, and places, which represent data, are connected using *directed edges* (arcs). Arcs enable the modelling of input-output functions of each discrete computation (i.e., each transition) in the agent being modelled. Arcs are normally annotated with an expression that evaluates (e.g. using variable *unification*) to a multiset over the type of the arc's associated place. The cardinality of the resulting multiset may be a function of the input tokens, thus PML supports *flexible* arcs, as defined in [11].

In PML, arcs can be annotated with (a) an expression in *bag* notation [8], or (b) an expression in *Object Constraint Language* (*OCL*) [15], or (c) an integer $w \in \aleph^+$, that represents arc weight [9]. A short *verb phrase* may be also added as an extra description; only one of (a), (b), or (c) can be used on a single arc. When (a) or (b) is used, the expression is expected to appear within curly brackets. Expressions attached to arcs must *always* evaluate to multisets (i.e. bags), irrespective of the actual notation used. *OCL* also supports the building of expressions that evaluate to *bags* (see [15]), but using a different syntax than the one found in CP-nets literature (see [8]). An alternative to the abstract expression formats is the direct use of *language-specific code* in place of arc annotations. When code is to be used directly as an arc expression, a short verb phrase may optionally be shown in the PN graph for conciseness, instead of the actual code.

Arc weights are not always required, and when omitted, the interpretation should be that weight = 1. Note that an arc can *only* be annotated with either an integer weight *or* an expression, but *never* both, as arc expressions are always assumed to *suppress* weights in PML.

Inhibitor arcs [9, 1] are *always* directed from a place to a transition, and are drawn using a small hollow circle instead of an arrowhead at the transition end. *Test* arcs [1] are drawn with a solid circle at the transition end. Non-consuming (*Reserve* [1]) input arcs are depicted as edges with a hollow arrow tip at the transition end. Regular input-output arcs are drawn with a plain arrowhead at the transition end [9]. The PML notation for arcs is illustrated in Fig. 2 below.

PML Arc Notation

Name	Symbol	Input	Output
Reserve Arc	──▷	✓	✗
Regular Arc	──>	✓	✓
Inhibitory Arc	──○	✓	✗
Test Arc	──●	✓	✗

Fig. 2 PML Arc Notation, indicating whether each type of arc can be an input and/or an output of a transition.

3.2 Place Proxy and Secure Communication

An essential element of a multiagent architecture is the communication infrastructure that it provides to agents. In PML, there are mainly two occurrences where communication would be beneficial: first, we have *intra-agent communication*; second, we have *inter-agent communication*. In order to address the communication problem we have introduced a new modelling primitive, the *Place Proxy*, which enables controlled and secure message passing between Petri net components in a simple and elegant way. It is called a *proxy* because it adds no functionality to its *subject* [5], but rather it adds *permissions* and a level of indirection when there should not be direct binding between two nets. It acts as a *protection proxy*, a *virtual proxy*, a *remote proxy*, a *synchronization* proxy, and a *smart reference* [5]. All of that is possible because a *Place Proxy* is only a thin wrapper for a real place (or another *proxy*).

In a PML model, a *Place Proxy* is shown as a dotted, hollow place, with the letter '*P*' printed in its centre. It can be used wherever a place could have been used, but it *must* eventually be connected to a place *or* another proxy by a dotted line segment in order to be valid; we shall say that it must have a *subject* - a term introduced in [5].

3.3 Modular Net Composition

In PML, it is possible to take a modular approach to agent design, and this is achieved by allowing our Petri nets to be *composite* entities, connected using *Place Proxies* (see Fig. 3). Subnets can be added to a net to promote reusability of functional units. In particular, composition is *recursive* and thus a single model is allowed to have multiple nested levels of *subnets*. Petri net models that support this feature are commonly known as *Hierarchical Petri Nets* [6]. The semantics of composition are flexible, and a *subpage* (term borrowed from [8]) can be attached to a container net in two ways.

The first we shall term *conflicting composition*, in which we demand that the transitions of the subnet are *added* to the top level net structure, thus the *subpage* is always executed *as if* it was part of the parent net (it's *supernet*). When some of its transitions are enabled they may execute instead of one of the enabled supernet's transitions.

The second way in which we can attach a subpage is by *concurrent composition*. When this takes place, the transitions of the subpage are *not* added to the supernet structure; execution must take place in another thread of control. This feature is very useful in practical terms because it allows us to better allocate physical threads of control to various Petri nets that may constitute an agent's net.

Connecting a Reusable *Subpage* using *Place Proxies*

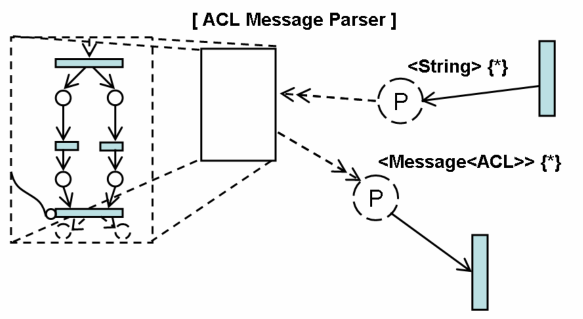

Fig. 3 Composition example, showing a message parser.

3.4 Inter-Agent Communication

By definition, *Place Proxies* can be used as tokens in the places of our nets. Communication between two or more agents using place proxies is always *asynchronous* and can be visualised, with *places* for downstream (incoming) messages and *tokens* for upstream (outgoing) messages, as shown in Fig. 4. Thus, *multicast communication* is also possible, by using multiple tokens for multiple recipients.

Communication between remote agent-nets occurs through *Remote Place Proxies*, as if they were local. The exchange of *Place Proxies* in the form of *tokens* provides a secure, uniform interface toward all aspects of agent communication by providing controlled access to specified places of their nets. This concludes our PML overview.

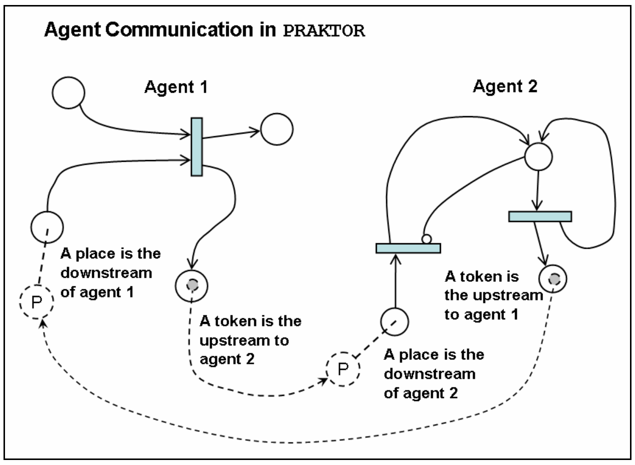
Agent Communication in PRAKTOR
Agent 1
Agent 2
A place is the downstream of agent 1
A token is the upstream to agent 2
A token is the upstream to agent 1
A place is the downstream of agent 2
P
P

Fig. 4 PML and inter-agent communication - simplified.

4 PRAKTOR Execution Layer

PML models are meant to be directly executable, after *arc expressions* have been specified in code. Currently, the IDE for graphical PML development is at an early stage, but a *reference implementation* of a PRAKTOR Runtime Library (PRL) has been developed in [14], and it includes representations of *all* PML elements in a total of approx. 150 classes. The PRL has been built using the Java 1.5 SDK (beta 1) which currently supports parametric polymorphism, and it allows manual, programmatic creation of executable PML-nets. Also, the PRL introduces the abstractions of *Portals* and *Dispatchers*, where the first act as gateway servers for incoming agents and the second encapsulate the network transport mechanics for the purposes of migration. Hence, all features we shall advertise here have been confirmed to work in preliminary experimentations with real traveller worker-agents, running in a small local area network. The results in terms of agent footprint were encouraging, as it was evident that the size of a PML-net is negligible compared to the data it would normally be expected to carry. Hosts have been successfully populated with thousands of concurrently executing agents and the general performance of the PRL has been found to be high and scalable.

4.1 Persistence and Hierarchical Migration

In terms of identified degrees of mobility, we usually differentiate between *strong* and *weak*. *Strong* mobility is the type of migration in which an agent can, at any time during its execution, encode and dispatch itself to another destination on the network, in which destination the agent will *resume* execution without appearing to have lost *any* of its previous state. *Weak* mobility refers to the same situation but the requirement on state preservation is relaxed and only instance variables are preserved. The latter means that the execution state of an agent can be lost during the transfer process; for example, stack frame information and the program counter may not be captured with the agent's state.

PRAKTOR supports *strong mobility*, and also allows the migration of groups of multi-threaded agents through the network with full state reconstitution and resumption upon arrival to the new host. Exact state preservation is a *direct* result of the inherent state representation encoded in the *marking* of the constituent PML-net(s) of an agent – since firing is *atomic*, we define migration in such a way that it occurs in *between* two transition occurrences (firings), thus preserving the state of the net(s).

4.2 Agent Encapsulation

Agents should *not* directly invoke operations *or* have any public operations to be invoked by other agents – instead, *Place Proxies* are exchanged to enable dynamic and asynchronous message passing (see Fig. 5). Commands can be submitted to agents and have them stop, freeze, persist, or migrate to remote hosts transparently. Interactions between agents are governed by *protocols*, and *not* by *interfaces*. Finally, an agent's lifecycle management can be delegated to itself, in contrast to traditional network-aware component models where distributed garbage collection is inevitable.

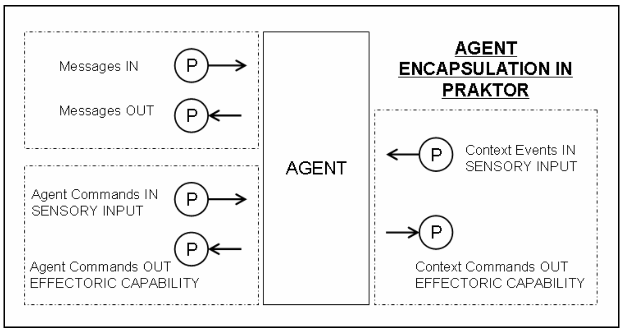
Messages IN
Messages OUT
Agent Commands IN SENSORY INPUT
Agent Commands OUT EFFECTORIC CAPABILITY
AGENT
AGENT ENCAPSULATION IN PRAKTOR
Context Events IN SENSORY INPUT
Context Commands OUT EFFECTORIC CAPABILITY
P

Fig. 5 Agent Encapsulation

4.3 The Scalable Concurrency Layer

The PRL employs a portable concurrency layer that enables the logical mapping of physical resources to

Petri net components, and is able to condense more than one agent (it's PML-net) into a physical *thread of control*. This has certain advantages: first, it allows flexible resource allocation, because agents can be grouped in an asymmetric way to achieve a balance in performance and resource consumption; second, it allows a system to consist of very large numbers of agents, with only a fraction of physical threads. Groups of agents may share physical schedulable resources, and agents are also able to attach new agents to their current thread of control.

4.4 Hierarchical Execution Contexts

PRAKTOR currently provides rudimentary support for concrete representations of the computational environment. *Agents* exist within *Contexts*, in which basic services are being offered to enable dynamic agent lookup, isolated execution and communication space, and basic system bookkeeping related to the arrival and departure of agents. *Contexts* can be modelled and implemented using the PML/PRL. Consequently, PRL *supports the migration of a context* together with *all* agents that are inside it, in a grouped manner. Also, as *Contexts* are composite components, a context may migrate (or, for example, hibernate to disk) together with all its *subcontexts* and their *agents*, recursively.

5 Conclusion

An agent architecture that is fundamentally based on Petri Nets has manifold advantages: firstly, it models explicit concurrency inside the agents themselves; secondly, it provides an intuitive and at the same time analysable model of individual agent behaviour. Petri net based models have already been researched in [4, 3] for the purposes of design and simulation, but with limited practical applicability. PRAKTOR is founded upon PML, a typesafe modelling language that is concise and precise, with clearly identified semantics. PML does not enforce specific agent architectures. However, unlike purely modelling languages, it provides directly executable versions of agent models by virtue of its one to one mapping of PML constructs to a specialised software library (PRL) that has been already implemented. PRAKTOR Agents may consist of an arbitrary number of physical threads of control. Moreover, agents that are produced by this architecture are persistent and thus exhibit *strong* mobility in a portable way. Analysis of PML models and their properties has not been attempted yet, but remains as a future research goal.

References:

[1] Christensen, S., Hansen, N. D. *Coloured Petri Nets Extended with Place Capacities, Test Arcs and Inhibitor Arcs* **in** Marsan, A. M., *LNCS, Vol. 691*, pp 186-205, Springer-Verlag, 1993.

[2] Decker, K. *et al* Distributed Intelligent Agents. *IEEE Expert*, Vol.11, No.6, pp 36-46, 1996.

[3] Duvigneau, M., Moldt, D., and Rölke, H. *Concurrent Architecture for a Multi-agent Platform* **in** Giunchiglia, F., Odell, J., and Weiß, G. (eds) *Agent-Oriented Software Engineering III*, 2002.

[4] Ferber, J. *Multi-Agent Systems: An Introduction to Distributed Artificial Intelligence*. Addison-Wesley, 1999.

[5] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.

[6] Huber, P., Jensen, K., and Shapiro, R.M *Hierarchies in Coloured Petri Nets* in Proc. of the 10th Int. Conference on Application and Theory of Petri Nets, Bonn, Germany, pp 192-209, 1989.

[7] ISO/IEC FCD 15909: Information Technology - *High Level Petri Nets – Concepts, Definitions and Graphical Notation*, 21 June 1998, ISO/IEC JTC1/SC7 N1947.

[8] Jensen, K. *An Introduction to the Theoretical Aspects of Coloured Petri Nets* **in** Bakker J.W. de, Roever W.-P. de, Rozenberg, G. (eds.) *A Decade of Concurrency. LNCS Vol. 803*, pp 230-272, Springer-Verlag, 1994.

[9] Peterson, J. L., *Petri Net Theory and The Modeling of Systems*, Prentice-Hall Inc, 1981.

[10] Petri, C. A., *Communication with Automata*, M.I.T. Memorandum MAC-M-212, Massachusetts Institute of Technology, 1962.

[11] Reisig, W. *Petri nets and algebraic specifications*. Theoretical Computer Science, Vol. 80, No. 1-2, pp 1-34, 1991.

[12] Sycara, K. P. Multiagent Systems. *AI MAGAZINE*, pp. 79-92, Summer 1998.

[13] Wooldridge, M. *Intelligent Agents* **in** Weiss, G. (ed) *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence*. The MIT Press, 1999.

[14] Mostrous, D. *Engineering Multiagent Systems with PRAKTOR*, BSc(Hons) Final Year Project, UCE Birmingham, 2004.

[15] Object Management Group. UML 2.0 OCL Final Adopted specification, available online at http://www.uml.org.