

Abstractions, Composition and Reasoning

Ella E. Roubtsova
Open University of the Netherlands
PO Box 2960, 6401DL
Heerlen, the Netherlands
ella.roubtsova@ou.nl

Ashley T. McNeile
Metamaxim Ltd.
48 Brunswick Gardens
London W8 4AN, UK
ashley.mcneile@metamaxim.com

ABSTRACT

We propose that different process algebraic composition techniques, combined with consideration of the restrictions on the ability of different parts of a system to share data and state, can provide a basis for identifying abstractions at the Platform Independent level of modelling. The paper presents our ideas and is aimed to initiate a discussion about the basis for identification of abstractions and the related areas of composition, reasoning and interface specifications, at the platform independent level.

Categories and Subject Descriptors

D.2.2 [SOFTWARE ENGINEERING]: Design Tools and Techniques; D.2.1 [SOFTWARE ENGINEERING]: Requirements. Specifications; D.3.1 [PROGRAMMING LANGUAGES]: Formal Definitions and Theory

General Terms

Design, Theory

Keywords

CCS composition, CSP composition, Platform Independent Modelling, abstractions, reasoning

1. INTRODUCTION

Dividing a system model into modules representing abstractions is an accepted principle to conquer complexity at any level of modelling. Such abstractions as objects, aspects [12], and services [14] were born as platform specific abstractions. More recently, it has been recognized that identification of aspects at the requirements and architecture level, i.e. at the Platform Independent level (PIM) [9], may give advantages in system evolution [7]. We believe that the same conclusion can be applied to platform independent separation and composition of objects and services as well. Moreover, a better understanding of aspects in models can be achieved by comparing them with other abstractions.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

AOM'09, March 2, 2009, Charlottesville, Virginia, USA.
Copyright 2009 ACM 978-1-60558-451-5/09/03 ...\$5.00.

At the Platform Independent level the models should not be “about functions that the software system must perform. The models should be about abstractions built in the context of business operations” [8]. Although a lot has been done in early recognition of aspects [7, 11, 2], there is no general approach to separation of abstractions at the Platform Independent level resulting in executable models. Abstractions are usually represented in a way that suggests some implementation knowledge, such as presence of methods or functions that the software system performs. In this paper we propose criteria for separating abstractions that share state with each other from those that do not, and therefore communicate by message passing. Our approach results in executable models of systems represented as communicating abstractions of different type.

All types of abstraction engage in event based communication, and we model this in the CCS style using actions: $!a$ for a “send message of type a ” action and $?a$ for a corresponding “receive message of type a ” action. If both source and destination abstractions involved in a message exchange are in the scope of modelling, then the model will contain complementary $!a$ and $?a$ actions. If an abstraction is communicating with the environment (outside of the scope of the model) then the communication actions are unmatched.

An abstraction that has private data and state is called a *service*. Services communicate with each other by message passing. This is modelled using CCS [13] style composition, whereby if one constituent is able to send (engage in $!a$) and another is able to receive (engage in $?a$), there is a *reaction* whereby the corresponding message payload of type a passes from sender to receiver and the states of the sender and receiver advance according to their respective process definitions. This type of composition is inherently non-deterministic: if, for a given state of the constituent services, more than one reaction is possible (as there is more than one pair of matching send and receive enabled) which reaction takes place is not determined. Separation of services means recognition of the source of non-determinism and possible integration problems.

A service can be further decomposed into constituent abstractions that share data and state and co-operate in deciding what send and receive actions the service as a whole can engage in. This is modelled by composing the abstractions that comprise a service using the CSP (Communicating Sequential Processes) parallel composition technique [5]. This style of composition models co-operation between the composed parts on how to behave, whereby if a given action (which could be either a send or a receive) is not possible

for any one of the constituents of the composition, then it is not possible for the composite service. It can be shown [1] that CSP|| composition used in this way yields deterministic behaviour. The abstractions used to model a service are classified as *objects* and *aspects*, as will be described later in the paper.

The behavioural elements of the system are modelled using Protocol Modelling [1]. A protocol model is built from conceptual machines called *protocol machines* which support both CSP and CCS composition. Protocol machines, unlike the pure algebraic process abstractions of CSP and CCS, have a concept of stored data and hence are suitable as a medium for modelling real systems.

We take the extent of data sharing allowed between abstractions to be a requirements issue, and not platform constrained. This means that the decomposition into services, and the consequent choice of composition techniques used to model a system, can be performed at a platform independent level. As the composition techniques have formally defined behavioural semantics, the models can be executable at the PIM stage.

In order to present our ideas in a familiar framework we define a UML profile that extends standard UML semantics with the notions that are needed for the proposed abstractions. In particular, we add the following:

- The notion of *event*, which is either a *send action* or a *receive action*; and *message*, which is the data payload that is passed between abstractions that engage in a send/receive pair. (Note that the words *event* and *action* are already used by UML, and that our use here is somewhat different.);
- the notion of *state derivation*, related to the UML notion of attribute derivation but concerned with determining when events can and cannot take place;
- CSP (Communicating Sequential Processes) parallel composition technique applied to models with data;
- CCS (Calculus of Communicating Systems) composition technique applied to models with data.

The structure of the paper is the following. Section 2 presents a case study we use to illustrate the ideas of the paper. Section 3 defines the criteria of abstraction separation and presents them in a UML profile for Platform Independent Modelling. Section 4 draws conclusions about advantages of early separation of different abstractions.

2. CASE STUDY

Our illustration is a *Public Information System* that keeps up to date the information about current education types and the possible exemptions from courses of one education for those who completed another education. As the information is supplied by different authorized official organizations, each entry goes through a security check. Moreover, the information about new educations and available exemptions must be published on an official publisher-site.

Modelling this system we can identify *Exemption* as a key abstraction. However, an exemption is defined in terms of a pair of educations, so the abstraction *Education* is also required. We also recognize a *Security Check* abstraction. The publishing of the information may be represented using

an abstraction *Publisher*.

Analyzing the relations between abstractions, we determine that the *Education* abstraction recognizes the following events:

Register-Education	Modify-Education	Discontinue-Education
-Date : Date;	-Date : Date;	-Date : Date;
-Name : String;	-Name : String;	Name : String;
-Type: String;	-Type: String;	Type: String;
-Contact: String;	-Contact: String;	Contact: String;
-Courses: Strings;	-Courses: Strings;	Courses: Strings;

The *Exemption* abstraction needs two instances of the *Education* abstraction. The events recognized by the *Exemption* abstraction are also shared with the *Education* abstraction:

Set Up Exemption	Modify Exemption
-Name:String;	-Name:String;
-ThisEducation: Education;	-Date : Date;
-AcceptedEducation :Education;	-ThisEducation: Education;
-Exemption Courses: String;	-Exemption Courses: String;

Remove Exemption
-Name: String;

The *Security Check* abstraction models behaviour that is required to prevent unauthorized modification of the data or state of any *Education* and/or *Exemption*. The events recognized by the *Security Check* are:

Secure Event = All events of classes Education or Exemption;

Set Password	Enter Password	Reset
-Saved Password: String;	-Pass: String;	-Pass: String;
-Pass: String;		

3. PLATFORM INDEPENDENT ABSTRACTIONS

In order to show our criteria for separation abstractions in a familiar framework, we define a UML profile for Platform Independent Modelling. A specification in our profile consists of a *Static View* and a *Dynamic View*.

Static View. A Static View is similar to a UML class diagram but has a different semantics. Its purpose is to show the abstractions of the system and their relationships. An abstraction in this view is a tuple:

$$Abstraction = (Name, A, S, E),$$

- *A* is a finite set of attributes. The set can be empty.
- *S* is a finite, not-empty set of states. A state may be *stored* or *derived*.
 - A **stored** state is a stored attribute with an enumerated type (allowing any one of a set of possible values).
 - A **derived** state [1] is a function that returns an enumerated type. The calculated domain of the function is the stored state and attributes of this abstraction and of all other abstractions whose data and state it may access.
- *E* is a finite, not-empty set of events, each of which is either a send action !*a* to send a message of type *a* or a receive action ?*a* to receive a message of type *a*.

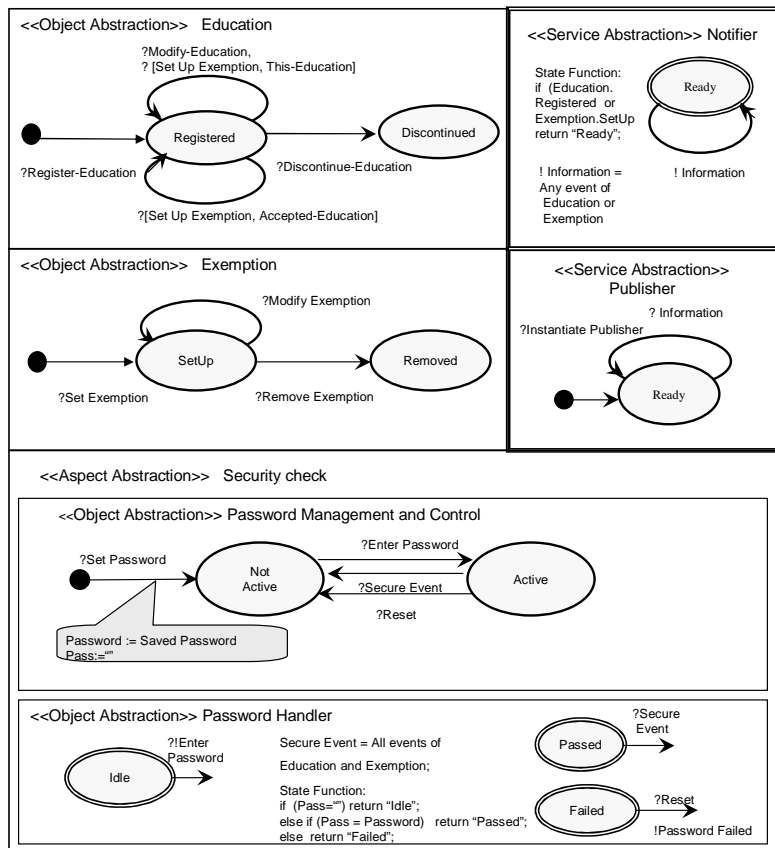
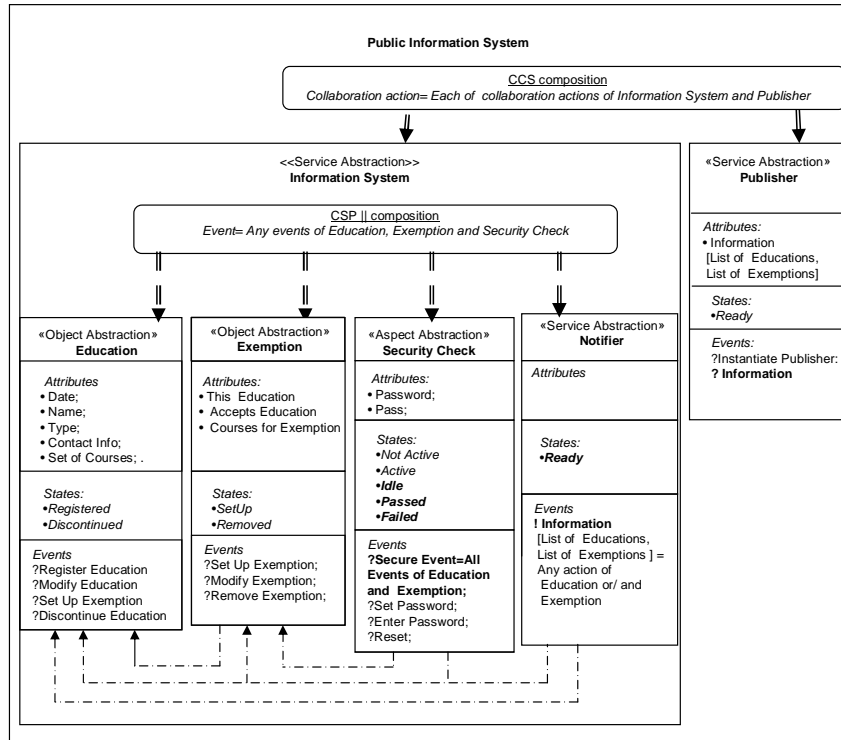


Figure 1: Static and Dynamic views presenting different abstractions

An abstraction is depicted as a box (Figure 1) and uses one of three stereotypes: $\langle\langle\text{Object Abstraction}\rangle\rangle$, $\langle\langle\text{Aspect Abstraction}\rangle\rangle$ and $\langle\langle\text{Service abstraction}\rangle\rangle$.

An **Object Abstraction** models an object in the domain. An object is modelled as one or more CSP composed protocol machines.

An **Aspect Abstraction** is an *Object Abstraction* that abstracts over the events and states of other objects, using *generalized events* and *generalized states*.

- A **generalized event** is an alias that represents a set of events, each of which cause *the same behaviour* (i.e., the same state change and same changes to local storage values) of the aspect.

- A **generalized state** is an alias that represents a set of states of this and other abstractions, each of which plays the same role (starting state or ending state) in the transitions of the aspect behaviour.

The *Security Check* is an example of an aspect abstraction with an abstract event *Secure Event*. Each of events of *Education* and *Exemption* is a *Secure Event* specified in the *Security Check* aspect.

A **Service Abstraction** is a CSP \parallel composition of objects and aspects. The objects and aspects within a service have access to each other's state and data – we depict this relation using a dashed arrow (Figure 1). The objects and aspects of a service cannot read data or state of other services.

For example, *Notifier*, *Information System* and *Publisher* are services. *Information System* is the result of the CSP \parallel composition of the objects *Education* and *Exemption*, the aspect *Security Check* and the service *Notifier*. The communication of the services is specified as a pair of actions *!Information* and *?Information*.

Composition Techniques. All abstractions are composed using two composition techniques: the *CSP \parallel composition rules* and the *CCS composition rules*. - *CSP \parallel composition rules* are applied across the abstractions *within* a service, and is over the alphabet of all events (send actions and receive actions) that the abstractions within the service support. - *CCS composition rules* are applied across services, and represents the way services communicate by message passing. If two services can engage in complementary communication actions *!a*, *?a*, a message of type *a* is passed between them.

Note that these composition techniques do not correspond to the UML composition notion [10]. Each of them may be considered as an abstraction itself, as each of them defines an algorithm for composition. However, these abstractions are the same for any system and should conform to a standard specification. That is why we show them at the static view as labels in a box with round corners: *CSP composition* and *CCS composition*. The labels are applied to boxes that model the abstractions being the results of composition: *Information system* and *Public Information system*. The composed abstractions are attached to this label by two parallel lines with arrows for CCS composition or by two dashed parallel lines for CSP composition (Figure 1).

Dynamic View.

In the Dynamic View an abstraction (an object, aspect or service) is a non-empty finite set of protocol machines PM_i

$$A = \{PM_1, PM_2, \dots, PM_i\}, i \in N$$

For example, abstraction *Education* contains one protocol machine. Abstraction *Security Check* has two protocol machines corresponding to it: *Password Management and Control* and *Password Handler*.

A Protocol Machine is a tuple $PM = (S, D, E, T)$,

- $S = \{s_1, s_2, \dots, s_n\}$, $n \in N$, is a non-empty finite set of stored states. A stored state has a corresponding set of attribute values, including the state name.
- $D = \{d_1, d_2, \dots, d_k\}$, $k \in N$, is a finite set of derived states calculated using the states of the machine itself and other protocol machines. D can be empty.
- $E = \{e_1, e_2, \dots, e_m\}$, $m \in N$, is an alphabet (a non-empty finite set) of events, each being either a send action or a receive action.
- $T = \{t_1, \dots, t_p\}$, $p \in N$, is a set of transitions. A transition t can be of types $t = (s_1, e, s_2)$, $t = (d_1, e, \text{any state})$ or $t = (\text{any state}, e, d_2)$.

A protocol machine is represented as a set of state diagrams, where

- a stored state is depicted as a one-line ellipse. For example, state *Active* is a stored state of machine *Password Management and Control*;
- the values of attributes in a stored state are presented near the state in a bubble, as it is shown for *Password Management and Control*;
- a derived state is presented as a double line ellipse. For example, states *Idle*, *Passed* and *Failed* are the derived states of machine *Password Handler*;
- the rules of derivation of generalized events and states are presented as expressions of the protocol machine. Examples of the expressions can be seen in the specification of *Password Handler*;
- a transition is depicted
 - as an arc, i.e. a pair of states labeled by the events that cause this transition or
 - as an arrow coming from or to a derived state and labeled by the events that cause this transition.

A Protocol Machine behaves as follows.

- **Event handling.**
 - A Protocol Machine can only engage in an action when in a quiescent state.
 - A Protocol Machines ignores any event that is not in its alphabet.
 - Depending on its state, a Protocol Machine either accepts (can engage in) it or refuses (cannot engage in) an event (a send action or a receive action) that is its alphabet. Whether a machine accepts or refuses an event is determined by the state of the machine both before and after the event (this is described in more detail in [1]).
- **CSP composition rules.**
 - Acceptance or refusal of an event, e , by a composite

machine, A , is determined as follows:

- If all machines comprising A that have e in their alphabet accept e , then A accepts (can engage in) e .
- If at least one of those machines refuses e , then A refuses (cannot engage in) e .

For example, in our case study, event *SetExemption* is accepted if:

- (1) two instances of protocol machine *Education This-Education* and *Accepts-Education* are in state *Registered*;
- (2) the instance of protocol machine *Exemption[This-Education,Accepts-Education]* is in state *SetUp*; and
- (3) the *Security Check* is in state *Active* and *Passed*.

• CCS composition rules.

If protocol machine A is in the state where it can engage in a send action $!a$ and protocol machine B is in the state where it can engage in the complementary receive action $?a$, then communication a can take place. A and B then advance their state and update their local storage as defined by their respective machine definitions. If either A cannot engage in $!a$ and/or B cannot engage in $?a$, no reaction takes place.

For example, services *Information System* and *Publisher* are composed using the CCS composition rules.

The CSP \parallel composition technique applicable for composition of protocol machines results in deterministic behaviour [1]. As it has been proven in [4], the result of the composition possesses the property of observational consistency or local reasoning. *Local reasoning* is the ability of understanding some properties of system behaviour based on examining its abstractions one by one [6]. Local reasoning relates properties of an abstraction to properties of the result of the composition.

For example, only examining the behaviour of the *Security Check* aspect, it is possible to say that the sequence *?Set Password; ?Enter Password; ?Enter Password; ?Modify Exemption* does not belong to the behaviour of the modeled system. It is not possible to enter password twice without a reset.

Local reasoning even allows reasoning when there is no formal statement of properties or requirements. In this case the behaviour of each of abstractions are used as properties that the behaviour of the result of composition of those abstractions should have.

4. ABSTRACTIONS AND REASONING CONCLUDING QUESTIONS

In this paper we have proposed criteria for separating abstractions at the PIM level of modelling based on the sharing of data and state and the consequent applicability of different composition techniques.

Also we have proposed a PIM UML profile with two different composition techniques: CSP parallel composition and CCS composition.

- The interfaces of abstractions in this profile are defined in terms of events.
- The dependencies are defined in terms of states.
- The models in this profile are executable models. For example, the ModelScope tool [3] generates Java code from the

meta-description of the PIM protocol models. The models can be implemented in any other programming language as well, and can be used for simulation.

- The CSP parallel composition technique applicable for composition of abstractions results in deterministic behaviour [1] being a set of sequences of events. As it has been proven in [4], the result of the composition possesses the property of observational consistency or local reasoning.

- The CCS composition is applicable for composition of abstractions that do not share data and communicate via messages. Although this composition necessarily introduces non-determinism associated with "race conditions", where two or more communications are simultaneously possible, it is conceptually simple enough to allow reasoning about the whole (for instance, establishing progress of the overall collaboration) already at the Platform Independent level.

The criteria for separating abstractions at the PIM level and the consequences of their choice need a discussion. With this paper we would like to initiate such a discussion about possible criteria of abstraction separation and their advantages and disadvantages for design and analysis.

5. REFERENCES

- [1] A. McNeile, N. Simons. Protocol Modelling. A modelling approach that supports reusable behavioural abstractions. *Software and System Modeling*, 5(1):91–107, 2006.
- [2] A. Rashid, A. Moreira. Domain Models are NOT Aspect Free. *LNCS 4199, ACM/IEEE 9th International Conference on Model Driven Engineering Languages and Systems*, pages 155–169, 2006.
- [3] A. McNeile, N. Simons. www.metamaxim.com/. 2008.
- [4] A. McNeile, E. Roubtsova. CSP parallel composition of aspect models. In *AOM '08: Proceedings of the 2008 AOSD workshop on Aspect-oriented modeling*, pages 13–18, New York, NY, USA, 2008. ACM.
- [5] C. Hoare. *Communicating Sequential Processes*. Prentice-Hall International, 1985.
- [6] D. Dantas, D. Walker. Harmless advice.
- [7] Early Aspects. <http://www.early-aspects.net/>.
- [8] M. Jackson, R. C. Laney, B. Nuseibeh, L. Rapanotti. Relating software requirements and architectures using problem frames. In *Proceedings of the IEEE Joint International Conference on Requirements Engineering*, pages 137–144, 2002.
- [9] OMG. MDA. <http://www.omg.org/mda/>.
- [10] OMG. *Unified Modeling Language: Superstructure version 2.1.1 (with change bars) formal/2007-02-03*. 2003.
- [11] R. Chitchyan, A. Rashid, P. Rayson, R. W. Waters. Semantics-based Composition for Aspect-Oriented Requirements Engineering. *Conference on Aspect-Oriented Software Development, Vancouver, Canada*. ACM, pages 36–48, 2007.
- [12] R. Filman, T. Elrad, S. Clarke, M. Akşit. *Aspect-Oriented Software Development*. Addison-Wesley, 2004.
- [13] R. Milner. *A Calculus of Communicating Systems*, volume 92. 1980.
- [14] T. Erl. *Service-Oriented Architecture (SOA): Concepts, Technology, and Design*. Prentice Hall, 2004.