A Proposal for Consolidated Intentional Modeling Language

Joshua C. Nwokeji
School of Science and Technology
Middlesex University, London
England, United Kingdom
J.Nwokeji@mdx.ac.uk

Tony Clark
School of Science and Technology
Middlesex University, London
England, United Kingdom
T.N.Clark@mdx.ac.uk

Balbir S. Barn
School of Science and Technology
Middlesex University, London
England, United Kingdom
B.Barn@mdx.ac.uk

ABSTRACT

Intentional modeling (IM) focuses on intentions and motivations of software systems rather than behaviours. KAOS ("Knowledge Acquisition in autOmated Systems"), and i* ("Distributed Intentionality") are the two popular IM languages used in requirement engineering. Each of these languages are defined as a collection of intentional elements, and intentional properties. However, these intentional elements are fragmented across IM languages, and thus limited in supporting detailed requirement analysis. Our proposed solution is to combine these two languages into a consolidated modeling language using a Model Based Software Engineering (MBSE) language integration technique, in EMF-Ecore, and develop a graphical tool for the new modeling language. The graphical tool is applied on a case study to show that it supports detailed requirement analysis. The rationale behind this paper is to provide the Software Engineering Community with a richer but less cumbersome intentional modeling language that can support detailed requirement analysis, this can reduce the cost associated with incomplete requirement analysis during software development.

General Terms
Intentional Modeling, KAOS, i*, Requirements Engineering

Keywords
Modeling Languages, MBSE, EMF, Requirements Analysis

1. INTRODUCTION

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.

GMLD '13, Montpellier, France
Copyright 2013 ACM 978-1-4503-2044-3
http://dx.doi.org/10.1145/2489820.2489826...$15.00.

Figure 1: Card Payment System for Bank XYZ Plc.

Intentional Modeling (IM) is a modeling paradigm in software development that defines the rationale/why for the existence of a system [20]. According to Yu et al [20], IM makes it easier to understand stakeholders’ goal, thus clarifying the drivers behind business decisions, and providing traceability for system changes. Other benefits of intentional modeling have been noted in [9] [16] [18] to include support for requirement specification, checking the completeness of a requirement specification, and providing alternatives to choose from during system design. Varieties of Intentional Modeling Languages (IMLs) have been used in requirement engineering, popular among these are i* [7] and KAOS [14]. Each of these languages are defined as a collection of intentional elements, and intentional properties. Although they offer a lot of benefits in Requirements Engineering (RE). The comparison of IM key concepts (elements) with popular IM approaches (KAOS, and i*) shown in Table 1 reveals that IM elements are fragmented across IM languages, and no single IM language has sufficient modeling elements to support detailed requirement analysis. For example, i* has no model element for Obstacle, thus can be difficult to support obstacle analysis. While KAOS does not explicitly define the concept of an intentional Actor. We describe a case study below and used it to explain the core model elements of intentional modeling language.

Bank XYZ provides Point of Sale-POS services via en-
Table 1: Comparison of IM Key Concepts with i* and KAOS

<table>
<thead>
<tr>
<th>IM Key Concepts</th>
<th>IM Approaches</th>
<th>i* [7]</th>
<th>KAOS [14]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Actor</td>
<td>✓</td>
<td>✓</td>
<td>?</td>
</tr>
<tr>
<td>Goal</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>HardGoal</td>
<td>✓</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>SoftGoal</td>
<td>✓</td>
<td>✓</td>
<td>?</td>
</tr>
<tr>
<td>Task</td>
<td>✓</td>
<td>✓</td>
<td>?</td>
</tr>
<tr>
<td>Constraint</td>
<td>✓</td>
<td>✓</td>
<td>?</td>
</tr>
<tr>
<td>Agent</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Strategic Dependency</td>
<td>✓</td>
<td>✓</td>
<td>?</td>
</tr>
<tr>
<td>Conflict</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Obstacle</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Legend:
✓-included in the Language
? - Not explicitly defined in the Language
ab-Absent in the Language

The core intentional elements and properties required to model the requirements of this system are explained below. A detailed definition and descriptions of other intentional elements can be found in [14] [7] [19] [13] [2]. An Actor is an active entity such as human e.g. Vendor or machine e.g. Router capable of performing actions within a system [21]. A Goal is a statement that describes what a system is designed to achieve e.g., successful transactions, or the intentions of a given actor in a system, e.g., steal credit card information [4]. A Goal that has clear criteria for its satisfaction is known as a HardGoal while a SoftGoal is a type of goal without a clear criteria for its satisfaction [7]. A Conflict is a trade off between goals, a situation where the satisfaction of one goal prevents the satisfaction of another [17]. For instance, the goal process customer request on time is in conflict with another goal secure transaction. A System Constraint shows certain regulations on a system, it can be used to implement government regulations on a system [18], e.g., daily transaction ≤ £500. An Obstacle is an undesirable condition to the satisfaction of a given goal in a system [14] [18], for example, the goal card readable can be obstructed by a faulty POS terminal. A Strategic Dependency defines strategic relationship between actors, i.e., how actors depend on each other to achieve their intentions [9] [21], for instance a Vendor will depend on the POS Terminal to satisfy the goal happy customer.

1.1 Challenges

Our aim is to carry out a detailed requirement analysis, check the completeness, and implement the constraints for the Point of Sale (POS) system described above. Thus we need to answer the following research questions:

Research Question 1: Which IML will be used to check the completeness of the requirement specification of this system (described above) with respect to Actor, its Goals, and the Obstacles to the satisfaction of those Goals?

Research Question 2: How can we precisely express constraints such as daily transaction ≤ £500 that occur in the system (described above) and as part of the intentional model in the form of elements such as goals?

To answer these questions we compared the key model elements (concepts) of two IMLs-KAOS, and i* as shown in Table 1. This comparison reveals that IM elements are fragmented across IMLs, thus we cannot find a single IML that explicitly gives us a complete set of these IM element (Actor, Obstacle, Goal, Constraints) required to check completeness and the constraints imposed on our system, e.g., i* has no model element for Obstacle, and Constraints, while KAOS does not explicitly define an Actor, and Constraints. Although the completeness of a system has been defined as assigning all goals to Actors that operationalise them [14,17], we argue that the existence of an Obstacle can hinder the operationalisation of a Goal, thus rendering the system incomplete. From the case study, the Goal- successful transaction can be operationalised by assigning it to an Actor-POS Terminal, but if the Obstacle-card unreadable is not resolved, then the Goal remains unsatisfied and the system incomplete. Therefore, without the existence of these intentional elements (Goal, Actor, and Obstacle), it will be difficult to use any IML to perform a completeness check.

The second research question raises issues to the meaning of IML elements. For example, both KAOS and i* include goals that are essentially constraints on system configurations. However, neither of the IMLs evaluated in this paper provide a language for expressing the configurations over which the constraints can range. Therefore, it is not possible to check whether a goal is correct with respect to the elements that it refers to. Such automated checks are key to providing tool support for IML and are provided in most tools for software engineering.

Our proposition is that a new IML with richer IM elements is needed to address these limitations. The new IML will include all the elements necessary to perform complete IM by constructing a rationalised union of the IMLs under study. In addition, the new IML will include a language for expressing the referents of goals and thereby a route to basic tool support for IM. Therefore, our contribution is to develop a new IML that contains sufficient intentional elements. Secondly, we develop a new graphical modeling tool to support our new language and apply it to the case study above. We also show how our language can be used to support completeness check and constraint analysis using the Object Constraint Language (OCL).

This paper is organised as follows, Section 2 gives a brief description of the two popular IM approaches used for this study, followed by a review of related work in Section 3. In Section 4 we present the design of our new modeling language CIIML and describe the tooling process for the graphical editor in Section 5. Section 6 shows how our graphical editor is applied to model the case study. This is followed by a completeness and constraint check in Section 7, and finally a conclusion in Section 8.

2. IM APPROACHES

KAOS is a requirement engineering framework whose primary focus is to explicitly represent all system goals, the
conflicts, and obstacles between these goals, the objects that are responsible for satisfying these goals, and the operations that are triggered as a result of the interaction between the goals, and objects [5] [9]. The KAOS meta-model shown in Figure 2 consists of Goal Model, Object Model, Operation Model and a Responsibility Model [5]. The goal model is a graph that represents system goals, subgoals, obstacles, and the agents responsible for satisfying the goals [19]. The object model defines the passive and active objects in a system such as agents, entities and associations [14]. The operation model describes the actions/activities that agents perform to satisfy the goals assigned to them, while the responsibility is a derived model that assigns responsibility to agents.

As described in [2, 9, 21], and shown in Figure 3, the i* language offers two types of models namely strategic dependency (SD), and strategic rationale (SR) models. An SD model describes the relationship where one actor called the dependee depends on another actor called the dependor to satisfy a given intentional property called the dependum, the four types of SD corresponds to the types of dependum, e.g. if the dependum is a goal, the SD is called Goal Dependency, and so on [1]. The strategic rationale (SR) model is another graph of nodes and links that describe the intentional property of each actor, and explain the rationale behind each actor's dependencies [1, 7]. Four types of links can be used to describe actors intentions. The means-ends link shows that an Actor can perform a task (means) to achieve a particular goal (end); a task decomposition link describes the steps that can be taken to perform a given task; a task can be decomposed into subtask, subgoal, softgoal, and resource [1]. The contribution link describes the type of contribution offered by each intentional element to another [1].

3. RELATED WORK

Various Model-Driven Engineering approaches such as model transformation and integration [12] [13] have been used in research and practice to derive integrated IML from existing language definitions, as discussed below. However, a major set back to current approaches is that they integrate parts of IML required for their problem domains, leaving out some important ones, and failed to show how their unified models can be used to support detailed requirement analysis, such as expressing and resolving Obstacles, and checking Constraints.

3.1 Model Integration Approach

Model integration techniques have been applied in [13] to design the so called unified goal oriented language (UGL). This technique integrates the basic concepts of three intentional modeling languages namely KAOS, i*, and GRL by applying a comparative analysis to their language definitions, and identifying similar modeling concepts in the three languages. The similar concepts identified are then combined into a single super class, e.g. the operation, requirement, expectation model elements from KAOS, and the task model element from i* are combined into the superAction class in UGL. A major contribution of this approach is to define a graphical tool in EMF/GMF that allows a model defined in one language (say KAOS) to be viewed in another language (say i*).

Similarly, the meta-model for a unified requirement modeling language (URML) is presented in [6]. This approach integrates various aspects of system modeling such as goal modeling, feature modeling, product line modeling, and requirement modeling into the so called URML. This approach claims that such integration can encourage a "homogeneous visualization" of requirement across all the multi-disciplines involved in the software design process. Basic model elements from KAOS, i*, BMM, and TOGAF have been integrated in [4] to develop a language called ARMOR. This
language is an attempt to incorporate requirement engineering into enterprise architecture.

3.2 Model Transformation Approach

A one directional transformation between i* and KAOS has been proposed in [11], and implemented in [12]. This approach defines a transformation relation between i* and KOAS in Atlas Transformation Language (ATL), and argues that such a transformation makes it easier to understand the differences between both languages, convert one model to another, and assist developers to make informed decision on the language to use in requirement analysis. A method to derive object-oriented conceptual models from i* has been proposed in [1]. This approach defines set of transformation rules that allows an i* model to be transformed into a conceptual model used for software development.

3.3 Empirical Comparison

Empirical comparison of IM languages have been reported in [3, 10, 15, 19]. Matulevicius and Heymans compared the quality of KAOS and i* in [10] by means of a *semitic quality framework* experiment and infer that model quality is a function of user experience. [15] reports an empirical experiment to compare the suitability of KAOS, i*, and NFR (Non Functional Requirement) in modeling collaborative systems. The study reveals that the three IM languages require further enrichment and modifications in order to model collaborative systems. A set of frameworks have been used to compare i* and KAOS in [3, 19]. The aim of these comparisons is to determine the potentials, and setbacks of IM languages in other aspects of modeling such as business process modeling.

A major limitation of these approaches is that none of them show how intentional elements such as obstacles, conflicts, and domain properties can be applied to check completeness or consistency of a model, nor show how the integrated languages can support detailed requirement analysis such as checking constraints.

4. CONSOLIDATED IML (CIML)

In Figure 4, we present the abstract syntax of our new language: CIML. The root element/container consists of two abstract classes: Node, and Link. A Node is used to represent a model element such as Actor, and Goal, while a Link represents the relationship between model elements. For instance, ObstructionLink assigns Goals to Obstacles, thus showing an obstruction relationship. A Node can be any of these four abstract classes: DependableNode - a type of node that has a dependency relationship implies that it can allow other nodes to depend on it in which case it is called...
Figure 4: The CIML Abstract Syntax (Meta-Model)
(Depender), and it can depend on other nodes (Depended) to satisfy its intentions [7]; Domain Element—any model element that is part of the system’s environment e.g., domain property; Intentional Property describes the objectives of an Actor within a system e.g., Goal [7]; RefinableNode is an abstract class that models some intentional properties that can be decomposed into sub properties e.g. a goal has subgoals. Dependency shows the type of dependency relationship that can exist between two actors, the various types have been listed as Enumeration literals in Figure 4.

A DependableNode can be an Actor which can be mapped into Actor, Position, and Roles in i* [7], and Environment-Agent in KAOS [14]; or Agent—the same as Agent and Software-Agent in i* and KAOS, respectively. Goal [7, 14], Obstacle [14], Task [7], and Constraint are types of Intentional Property. Unlike KAOS and i*, constraint is an intentional property in CIML, because it represents a restriction imposed on a system by either an internal stakeholder such as Actor/Agent or an external stakeholder such as Government. In any case, it represents intention of a stakeholder, thus an intentional element. Constraints are usually satisfied before or after using the system are respectively known as pre and post conditions [14]. A Goal can be a Hard-Goal, which implies a clear criteria for its satisfaction, or a non-functional-NFGoal, which implies lack of clear criteria for its satisfaction e.g. ensure happy customer. DomainElement in CIML can be a Domain Property [14], Activity, or a Resource [7], in i* resource is a type of an intentional element and defined as an entity such as data used by an actor [7]. From this definition we categorise resource as a type of domain element rather than intentional element because we argue that a resource such as debit card does not represent actor’s intention. Activity is used to represent the actions performed by a actor or agent to satisfy its intention with the domain of the system. A (strategic) dependency relationship can exist between Actors. The dependency relationship between Actors is established when an Actor (called dependee), depends on another Actor(s) (called dependee) to achieve an intention (called dependum). For instance, the Vendor depends on the POS terminal to achieve the constraint daily transaction not greater than £500. A dependum can be a HardGoal, a NFGoal, a task, and/or a Constraint. The four types of strategic dependency in CIML are listed as enumeration at down right side of Figure 4. Each strategic dependency corresponds to the type of the dependum. For instance, if the dependum is HardGoal, the strategic dependency is called HradGoal dependency [7].

Refinement Link is used to decompose RefinableNodes into sub-nodes. They can be And refinement which shows a compulsory alternative, or Or refinement which shows a non-mandatory alternative [14]. DependencyLink shows the type of dependency between actors [7]. ConflictLink shows a conflict relationship between goals, resolutionLink shows that an obstacle has been resolved, while assignmentLink assigns an actor/agent to a goal [14]. Each element in CIML is equivalent to one or more elements in KAOS, and/or i*. We demonstrate this by mapping CIML elements with elements of KAOS and i* as described in Table 2 For instance, AssignmentLink is mapped into OperationalisationLink, assignmentLink, and responsibilityLink in KAOS [14], and contribution/means-end link in i*. Similarly, an Actor in CIML is equivalent to Actor, Position, and Roles in i*, or EnvironmentAgent in KAOS. Other examples are shown in Table 2.

<table>
<thead>
<tr>
<th>CIML Element/Symbol</th>
<th>Mapped Elements in i*</th>
<th>Mapped Elements in KAOS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Actor</td>
<td>Actor, Role, and Position</td>
<td>Environmental Agent</td>
</tr>
<tr>
<td>Agent</td>
<td>Agent</td>
<td>Software agent</td>
</tr>
<tr>
<td>Goal</td>
<td>Goal</td>
<td>Goal</td>
</tr>
<tr>
<td>Resource</td>
<td>Resource</td>
<td>Entity</td>
</tr>
<tr>
<td>HardGoal</td>
<td>HardGoal</td>
<td>Requirement</td>
</tr>
<tr>
<td>NFGoal</td>
<td>Softgoal</td>
<td>Expectation</td>
</tr>
<tr>
<td>Task</td>
<td>Task</td>
<td>Events</td>
</tr>
<tr>
<td>Obstacle</td>
<td>none</td>
<td>Obstacle</td>
</tr>
<tr>
<td>Conflict</td>
<td>none</td>
<td>Conflict</td>
</tr>
<tr>
<td>Constraint</td>
<td>none</td>
<td>Pre, and post conditions</td>
</tr>
<tr>
<td>Activity</td>
<td>none</td>
<td>Operation</td>
</tr>
<tr>
<td>RefinementLink</td>
<td>DecompositionLink, isa, is-part-of</td>
<td>RefinementLink</td>
</tr>
<tr>
<td>ObstructionLink</td>
<td>square</td>
<td>ObstructionLink</td>
</tr>
<tr>
<td>DependencyLink</td>
<td>DependencyLink</td>
<td>none</td>
</tr>
<tr>
<td>AssignmentLink</td>
<td>AssignmentLink</td>
<td>OperationalizationLink, Responsibility Link, Assignment Link</td>
</tr>
</tbody>
</table>

Table 2: Mapping of key CIML concepts with i* and KAOS.
5. THE CIML TOOLING

We developed the CIML tool in the Eclipse Graphical Modeling Framework (GMF) using EuGENia [8]. A tool for developing graphical editors. The diagram in Figure 5 describes the steps involved in CIML tooling. First, we design the abstract syntax or Meta-Model for CIML using Eclipse Modeling Framework (EMF) and annotate it as ciml.ecore as shown in Figure 5. The next step involves converting the ciml.ecore model to an EmfaTic file and annotating it with Java to describe the attributes of the objects/nodes and their connectors/Links. Emfatic is a development environment that allows *.ecore files to be annotated with Java. To further explain the CIML tooling process, a sample of the Java annotated ciml.ecore is shown in the latter part of Section 5. Line 1 declares the namespace which specifies the location of ciml.ecore, i.e., "http://ciml/1.0". Line 4 tells EuGENia that the root element or container is ciml, and therefore should not be included in the diagram. The gmf.node annotations in Line 10 are used to specify that a particular Eclass i.e Ecore Class is a node, and further defines the attributes such as figure, label, and icon for a particular node. Nodes are called Objects in the palette, see Figure 6. Connectors or Links are specified in the EmfaTic file using the gmf.link annotations as seen in Line 15, and defines attributes such as target-decorations, source, and target of the connector.

After annotating all the Eclasses with java, then we used EuGENia to generate the models that describes the attributes of the CIML graphical editor, these include ciml. There are three basic types of models generated by EuGENia, these include the Graph Model, Tooling Model, and Mapping Model. The Graph Model describes the elements of the graphical editor such as connectors, labels, decorations; Tooling Model specifies the elements tools that will be available in the palette of the graphical editor; and the Mapping Model maps the elements in the Graph Model and creation tool with the Meta-Model defined in Ecore [8]. Once these models are generated, then the diagram plugin for CIML is created. Finally, after running a new eclipse runtime the CIML graphical editor is generated. The description of all the symbols used for each CIML model element is provided in Table 2, while the graphical convention is shown in Figure 8.

G mócikem 10

6. APPLYING THE CIML TOOL

We used our CIML tool to model the case study described in Section 1 as shown in Figure 6. The Goal (G1) of Bank XYZ is to provide excellent POS Services, this implies that all transactions must be successful, secure, and customers must be happy. Thus its refinement into three compulsory alternative SubGoals SG1-successful transaction; and SG2-secure transaction; and SG3-Happy Customer using the And Refinement Link. SG1-successful transaction means that all smart cards must be readable, and daily transaction must be uploaded successfully. We thus refined SG1 into two HardGoals: HG1-Card readable, and HG2-daily transaction uploaded. Similarly SG2-secure transaction implies that the smart card information is encrypted, or that the vendor upload daily transactions using the secure encrypted option E1. This lead to refinement into non-compulsory alternatives HardGoals: HG3-smart card information encrypted, or HG4-daily transaction uploaded with secure option. The subgoal SG3-happy customer means that all customer requests are processed on time. The Vendor will depend on the POS terminal to achieve this goal. This shows an instance of strategic dependency. However, the hardgoal HG5-process

Figure 5: CIML Tooling Process

A CIML model is linked to a Class_Model. This provides a means for modelling the system configurations used as the referents of goals and other IM elements. We assume that the reader is familiar with standard class models involving classes, attributes, associations and inheritance and therefore we do not include the details in figure 4. We assume that the modeller is free to construct an information model in the form of classes that expresses just those aspects of the system required by the intentional elements. Given that each intentional model has a class model whose instances are the referent of goals etc., it is possible to use a precise language to express the goal conditions. Of course, part of the reason for constructing an IM during the early stages of system development is to be informal. Of course, part of the reason for constructing an IM during the early stages of system development is to be informal. However, a possible route to precision, by transforming informal constraints to formal constraints, is attractive, and is a prerequisite for tool support. A suitable language for defining such conditions is OCL since this is defined to have formal constraints to formal constraints, is attractive, and is a prerequisite for tool support. A suitable language for defining such conditions is OCL since this is defined to have formal constraints to formal constraints, is attractive, and is a prerequisite for tool support. A suitable language for defining such conditions is OCL since this is defined to have formal constraints to formal constraints, is attractive, and is a prerequisite for tool support. A suitable language for defining such conditions is OCL since this is defined to have formal constraints to formal constraints, is attractive, and is a prerequisite for tool support. A suitable language for defining such conditions is OCL since this is defined to have formal constraints to formal constraints, is attractive, and is a prerequisite for tool support. A suitable language for defining such conditions is OCL since this is defined to have formal constraints to formal constraints, is attractive, and is a prerequisite for tool support. A suitable language for defining such conditions is OCL since this is defined to have formal constraints to formal constraints, is attractive, and is a prerequisite for tool support. A suitable language for defining such conditions is OCL since this is defined to have formal constraints to formal constraints, is attractive, and is a prerequisite for tool support. A suitable language for defining such conditions is OCL since this is defined to have formal constraints to formal constraints, is attractive, and is a prerequisite for tool support. A suitable language for defining such conditions is OCL since this is defined to have formal constraints to formal constraints, is attractive, and is a prerequisite for tool support. A suitable language for defining such conditions is OCL since this is defined to have formal constraints to formal constraints, is attractive, and is a prerequisite for tool support. A suitable language for defining such conditions is OCL since this is defined to have formal constraints to formal constraints, is attractive, and is a prerequisite for tool support. A suitable language for defining such conditions is OCL since this is defined to have formal constraints to formal constraints, is attractive, and is a prerequisite for tool support. A suitable language for defining such conditions is OCL since this is defined to have formal constraints to formal constraints, is attractive, and is a prerequisite for tool support. A suitable language for defining such conditions is OCL since this is defined to have formal constraints to formal constraints, is attractive, and is a prerequisite for tool support. A suitable language for defining such conditions is OCL since this is defined to have formal constraints to formal constraints, is attractive, and is a prerequisite for tool support. A suitable language for defining such conditions is OCL since this is defined to have formal constraints to formal constraints, is attractive, and is a prerequisite for tool support. A suitable language for defining such conditions is OCL since this is defined to have formal constraints to formal constraints, is attractive, and is a prerequisite for tool support. A suitable language for defining such conditions is OCL since this is defined to have formal constraints to formal constraints, is attractive, and is a prerequisite for tool support. A suitable language for defining such conditions is OCL since this is defined to have formal constraints to formal constraints, is attractive, and is a prerequisite for tool support. A suitable language for defining such conditions is OCL since this is defined to have formal constraints to formal constraints, is attractive, and is a prerequisite for tool support. A suitable language for defining such conditions is OCL since this is defined to have formal constraints to formal constraints, is attractive, and is a prerequisite for tool support. A suitable language for defining such conditions is OCL since this is defined to have formal constraints to formal constraints, is attractive, and is a prerequisite for tool support. A suitable language for defining such conditions is OCL since this is defined to have formal constraints to formal constraints, is attractive, and is a prerequisite for tool support. A suitable language for defining such conditions is OCL since this is defined to have formal constraints to formal constraints, is attractive, and is a prerequisite for tool support. A suitable language for defining such conditions is OCL since this is defined to have formal constraints to formal constraints, is attractive, and is a prerequisite for tool support. A suitable language for defining such conditions is OCL since this is defined to have formal constraints to formal constraints, is attractive, and is a prerequisite for tool support. A suitable language for defining such conditions is OCL since this is defined to have formal constraints to formal constraints, is attractive, and is a prerequisite for tool support. A suitable language for defining such conditions is OCL since this is defined to have formal constraints to formal constraints, is attractive, and is a prerequisite for tool support. A suitable language for defining such conditions is OCL since this is defined to have formal constraints to formal constraints, is attractive, and is a prerequisite for tool support. A suitable language for defining such conditions is OCL since this is defined to have formal constraints to formal constraints, is attractive, and is a prerequisite for tool support.
customer request on time means that a faster but less secure network will be used, this is in Conflict with SG2-secure transaction.

Although the hardgoal HG1: card readable can be satisfied by assigning it to the Agent; Ag1-POS terminal, the Obstacle O1: card unreadable can hinder its satisfaction. To resolve this obstacle we have to find its possible causes such as a faulty POS terminal or faulty smart card. Then, we refine the obstacle into O2: faulty POS terminal or O3: card unreadable, and resolve them by assigning an agent/actor to each of them. Obstacle O3 is refined into a hardgoal HG6-replace card, and assigned to the Actor; At1-Vendor, while Obstacle O2 is assigned to an Actor; At2-Repair Engineer. This provides a means of analysing and resolving obstacles in CIML using three intentional elements (Actor, Agent, Goal, and Obstacle), which is one of our contributions in this paper. Other hardgoals are assigned to Actors, and Agents as shown in the Figure. The Constraint, C1-daily card transaction ≤ £500 imposed on the Actor; At3-Customer is enforced by assigning it to the Agent; Ag1-POS terminal. The Resource; R1-smart card can be used by the Vendor or the Customer.

Each CIML model is associated with a class model that describes the referents for the intentional modelling elements. We do not require any bespoke tooling for class modelling since it is ubiquitous. Figure 7 shows a class model for the case study. In overview, each shop contains a collection of POS terminals that maintain a history of their transactions. Each transaction involves a card that is owned by a customer and that may have become unreadable (perhaps the risk of a card being unreadable increase with the number of uses). Each transaction contains the time it takes for confirmation to be returned by the bank. Both the shop and the bank are external connections on a network. Paths through the network involve routers that may be encrypted or not and which have processing speeds (perhaps the level of encryption affects the speed of processing). A hacker performs a number of sniffs on routers that may or may not be successful (depending on the level of encryption).

7. VERIFICATION

The CIML provides a rationalised union of the intentional elements required to be complete with respect to a system. Our claim is that this new language allows us to check for completeness and this section uses OCL to specify this property of a CIML model. In addition, CIML includes system configurations expressed as class models. Therefore, OCL can also be used to express constraints over the system configurations including, where appropriate, intentional model elements.

7.1 Completeness

A requirement specification is complete with respect to Actor, Goal, and Obstacle if all goals have been assigned to an actor or agent, and all obstacles to the goal have been resolved. All of the assignments in a model are returned by the following query:

class ciml::assignments() =
Resolution links are returned as follows:

```ocl
context CIML::resolutions() =
  links->select(l | l.oclIsKindOf(Resolution_Link))
```

Goals are returned by the following query:

```ocl
context CIML::goals() =
  nodes->select(n | l.oclIsKindOf(Goal))
```

Obstacles are returned by the following query:

```ocl
context CIML::obstacles() =
  nodes->select(n | l.oclIsKindOf(Obstacle))
```

An obstacle is resolved when there is a resolution link for it:

```ocl
context Obstacle::resolved(model) =
  model.resolutions->exists(r | r.resolves->includes(self))
```

A goal is assigned in a model when the following predicate is satisfied:

```ocl
context Goal::isAssigned(model) =
  model.assignments()->exists(a |
    a.goal = self and
    let target =
      a.assigns_intentional_property_to
    in
target.oclIsKindOf(Agent) or
target.oclIsKindOf(Actor)
  )
```

The following OCL expression can check completeness:

```ocl
context CIML inv Completeness:
  goals()->forall(g | g.isAssigned(self)) and
  obstacles()->forall(o | o.resolved(self))
```

The check for completeness is therefore, precisely defined and acts as a specification for any CIML tooling. This is possible because CIML is defined as a meta-model in a standard language, as opposed to the definitions of other IMLs, such as i* and KAOS that are often defined using concrete syntax based notations.

### 7.2 Constraints

Expressing the system as a class model allows us to precisely encode constraints using OCL. The constraints can be invariants that must be true of an instance of the class model in all stable configurations, or can encode intentional model elements, such as goals, that represent desirable system configurations.

The following OCL constraint requires that there is a limit of £500 on daily transactions at any given POS terminal:

```ocl
context Shop inv:
  terminals->forall(t |
    transactions.groupByDay()->forall(T |
      T.amount->sum <= 500))
```

where the query operation `groupByDay()` maps a set of transactions \( T = \{ t_1, t_2, t_3, \ldots, t_n \} \) to a set of disjoint sub-sets \( \{ T_1, T_2, \ldots, T_k \} \) such that the transactions in each \( T_i \) occur on the same day.

We can express the condition that routers process information at different speeds depending on whether they use encoding or not:

```ocl
context Network inv:
  routers->forall(r1 r2 |
    if r1.encrypt then
      r1.encrypt and not(r2.encrypt)
    else
      true
    end)
```

The diagram represents a class model for the case study, showing relationships between different entities such as Shop, POSTerminal, Transaction, Card, Customer, Vendor, Bank, Network, Router, and Item. Each entity and relationship is defined with attributes and class hierarchy, allowing for a comprehensive understanding of the system's architecture and constraints.
implies r1.speed > r2.speed
)

A goal, from a customer’s point-of-view, is that a transaction should not take too long. Of course an acceptable duration will depend on the situation, but we assume that it is a known constant transDur. OCL does not allow us to express a desirable constraint, i.e., one that might be true but which might be (reasonably) refuted by the goal of another system agent. Furthermore, goals are unlike OCL constraints in that they may be blocked through obstacles that need resolution.

Therefore, we suggest that in order to encode goals and other intentional modelling elements as expressions in OCL, we will require additional OCL features. The design of such features is left as further work. However, we propose a simple extension here that allows goals to be defined from the perspective of a stakeholder in the system. Assuming that there is a query operation called getPathToBank() defined for a transaction that returns a network path that can be used to send transaction data to the bank, then:

```java
1. context Transaction goal Wait for Customer:
2. getPathToBank().routers
3. ->iterate(r time_taken = 0 |
4. time_taken + r.speed) < transDur
```

Clearly there is more work to be done here in order to express intentional elements as precise constraints in OCL. However, we claim that the CIML is a contribution and that allowing system states to be attached to intentional models via class models is a significant step in achieving this. Further work will include using stereotypes on class model elements to attach them to specific IM elements, and extending OCL in order to support different types of IM element.

8. CONCLUSION AND FUTURE WORK

In this research we analysed two major IML used in industry and research, and discovered that intentional elements are fragmented across these languages, thus their limitation in singularly supporting certain analysis required in requirement engineering. We address this limitation by proposing a new language with richer but less cumbersome intentional elements, and design a graphical editor for our new language. We applied our language to model a case study using the graphical editor and further show how the completeness and some constraint can be checked with our model using OCL.

Our next step will be to automate these checks by encoding the completeness and constraint check into our graphical
editor using Epsilon Validation Language (EVL). Also, we intend to develop a concrete syntax for our language, and publish it over the internet as an Open Source Software. For further validation, we will apply our language in other areas of Software Engineering, such as Enterprise Architecture Alignment.

9. REFERENCES


