Automated completeness check in KAOS

NWOKEJI, Joshua C., CLARK, Tony <http://orcid.org/0000-0003-3167-0739>, BARN, Balbir and KULKARNI, Vinay

Published version

NWOKEJI, Joshua C., CLARK, Tony, BARN, Balbir and KULKARNI, Vinay (2014). Automated completeness check in KAOS. In: INDULSKA, Marta and PURAO, Sandeep, (eds.) Advances in conceptual modeling : ER 2014 Workshops, ENMO, MoBiD, MReBA, QMMQ, SeCoGIS, WISM, and ER Demos, Atlanta, GA, USA, October 27-29, 2014. Proceedings. Lecture Notes in Computer Science (8823). Springer International Publishing, 133-138.

Automated Completeness Check in KAOS

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

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

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

***Abstract*—KAOS is a popular and useful goal oriented requirements engineering (GORE) language, which can be used in business requirements modelling, specification, and analysis. Currently, KAOS is being used in areas such as business process modelling, and enterprise architecture (EA). But, an incomplete or malformed KAOS model can result to incomplete and erroneous requirements analysis, which in turn can lead to overall systems failure . Therefore, it is necessary to check that a requirements specification in KAOS language are complete and well formed. The contribution at hand is to provide an automated technique for checking the completeness and well-formed-ness of a requirements specification in KAOS language. This is accomplished by adding a plug-in on the KTool developed in our previous research.**

I. Introduction

Goal oriented requirement engineering (GORE) is a requirements engineering method, which uses goals and other intentional concepts to analyse and specify the requirements of a system [29]. The essence of GORE is to provide an alternative to the traditional systems development paradigm, where systems requirements are analysed based on the intentions/why/motivations rather than the behaviour of the system. Among varieties of competing GORE languages such as i* [28], GBRAM [2], and BMM [21], KAOS [23] [20] [6] [10] [17] [26] is one of the most popular and widely used languages [10]. It can be used in both early, and late requirement phases, but it mainly focuses on analysing system goals [18]. Since KAOS has been widely used and is representative of other GORE languages, we use it as the basis of our study. Currently KAOS is receiving attention both in research, and practice [19] [20]. For instance, it has been applied in requirements acquisition, specification and analysis [19] [1]; enterprise architecture alignment [22] [8] [5]; solving service mediation problems [3]; model driven development [20]; and business process modeling [16].

The term *requirements* can be defined as a high-level statement that describes what a system does, the services it provides, and the the conditions on the systems, while requirements analysis involves the acquisition, understanding, specification, and elaboration of systems requirements [11], [24]. Requirements analysis/modelling is a core imperatives in systems development, this is because it has the ability to cripple the entire system if incomplete or erroneous [7]. In fact, over 80% of systems failure are attributed to incomplete or erroneous requirements specification and analysis [4], [27]

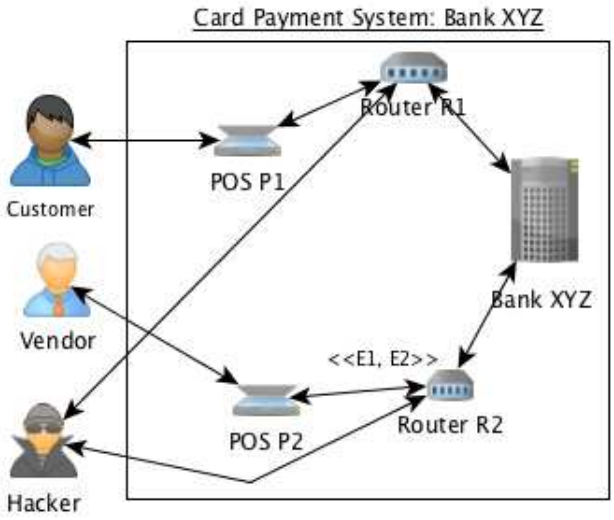

Figure 1: Card Payment System for Bank XYZ Plc.

The 'state of the art' in KAOS is its use as a basis for integrating GORE with *Model Driven Engineering (MDE)* via *Mechanical Language Processing (MLP)*. KAOS can provide a basis for mechanical language processing in MDE such as *model transformation*. For instance Monteiro et al proposed and implemented an automatic transformation between KAOS and i* models, and suggests that such transformation can support a migration from i* model to KAOS model and vice versa, and offers flexibility for choice of GORE language to be used in a software development project [19] [20]. MDE Mechanical language processing such as model transformation depends on inter mapping and manipulations of model elements in the different models concerned [20], thus an incomplete model can lead to incomplete mapping and erroneous MLP in MDE, thereby compromising the benefits offered by integrating GORE with MDE. Few manual techniques for completeness checks in GORE model are available in literature, for instance the use of *Temporal Logic* to check completeness is proposed in [25], while *Goal-Question-Metric (GQM)* techniques has been proposed in [9]. The problem with these manual techniques is that they are rigorous, error prone,

time consuming, difficult to use, and costly. Our contribution towards addressing these problems is to design and implement a tool supported technique for automatically checking the completeness of KAOS goal model to ensure its suitability for MDE mechanical language processing. Our contribution can be divided into two phases, first we construct a complete KAOS Meta-Model by consolidation on six existing Meta-Models found in literature, this is necessary since we couldn't find an existing Meta-Model that contains all KAOS model elements needed to define completeness check, see Table I. In the Second phase of our contribution we use this Meta-Model to implement a *Graphical Editor*-`KTool` for constructing KAOS models. Finally we develop an integration plugin that automatically check the completeness of KAOS model constructed with `KTool`. We apply `KTool` to design a KAOS model for the case study we describe in Section II, and use our automated *Completeness Check* technique to check its completeness automatically. The result suggests that our tool can make it easier, cost and time effective to check complex KAOS models, thus realising the benefits of integrating GORE with MDE.

The rest of this paper is structured as follows: In Section II we briefly describe MDE, and hen use a case study to give an overview of the basic elements of the *KAOS* language. We discuss existing proposal on KAOS completeness, and consolidate them into three *Completeness Checks* in Section III. Section IV provides a description of our tooling processing, starting with the construction of a comprehensive Meta-Model. The demonstration of our tool is presented in Sections V, this is followed by conclusion.

II. MDE AND KAOS

A. Model Driven Engineering

Model Driven Engineering (MDE) is traditionally used to generate executable systems from models and relies on tooling that builds, maintains and manipulates the models, [12]. The most popular language for MBSE is UML that offers sub-languages for specifying *what* a system must do and, at lower levels of abstraction, *how* it must achieve the behaviour. MDE technologies support a range of activities including model construction, model extension, model integration, model maintenance, model analysis and model transformation. The definition of these activities relies on a precise definition of the modelling language being used. The definition of a modelling language consists of a collection of elements. The *abstract syntax* of a language defines its underlying concepts and their relationships, the *concrete syntax* defines how it is represented on the screen or the page, and the *semantics* places constraints on its static structure, dynamic behaviour or both.

B. KAOS

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 [10] [15]. As seen in Figure 2, KAOS is a composite Meta-Model consisting of a *Goal, Object, Operation* and *Responsibility* sub Meta-Models [10] [23]. To describe these sub Meta-Models, and the core elements of KAOS we use the case study described as follows: Consider the requirement analysis of a *Card Payment System for Bank XYZ* shown in Figure 1, *Bank XYZ* provides *Point of Sale-POS* services via an encrypted but slow or unencrypted but fast wireless *Routers*(E1 and E2 respectively) to *Vendors*-who sell products to *Customers*, and upload daily transactions to the *Bank*; the *Bank* should also protect all transactions from *Hackers*-who steal *Smart Card* information from *Customers*. In Sections II-C, II-D, II-E, and II-F we use this case study to describe the elements of the *Goal, Object, Operation, and Responsibility* models respectively.

C. The Goal Model

KAOS Goal Model is a graph of nodes and links that describes the goals of a system, the conflict between these goals, the obstacles to the satisfaction of the goals [23]. The nodes in a KAOS goal model represents model elements such as Goals, and Obstacles, while the links are used to define the relationships between the nodes. The core and basic elements of KAOS goal model are described below using examples from the case study, however a more detailed description KAOS goal model and all its elements can be found in [23].

Goal is a statement that describes what a system tends to achieve *e.g.*, *successful transactions*, or the intentions of a given actor in a system, *e.g.*, *steal credit card information* [8]. A *parent* goal may be decomposed, or *refined*, into *child* sub-goals [20]; a *leaf* goal has no children. A KAOS goal model is said to be *complete* when all leaf goals are assigned to agents [23]. A **Conflict** is a trade off between goals, a situation where the satisfaction of one goal prevents the satisfaction of another [23]. For instance the goal *process customer request on time* is in *conflict* with another goal *secure transaction*. **Requirement** is a type of goal that has a clear criteria for its satisfaction and are usually assigned to Software Agents, while **Expectation** has no clear criteria for its satisfaction and are assigned to Environment Agents, for instance the goal *happy customer* is an expectation [23]. An **Obstacle** is an undesirable conditions to the satisfaction of a given goal in a system [23] [25], for example, the goal *card readable* can be obstructed by a *faulty POS terminal*. **Domain Property** is a condition which most hold for a goal to be satisfied, from our case study it is expected that the POS infrastructure should be available before the goal *Successful transaction* can be met, thus *POS infrastructure available* is an example of a domain property. **Domain Invariant** is a type of domain property that must be true in all states of the system for a goal to be achieved [17]. **Domain hypothesis** is a type of domain property that is expected to be true in order to satisfy a goal [23].

KAOS goal model also consist of links for instance, the **RefinementLink** is used to show a refinement relationship between a parent goal, and expectation and requirement. There are two different types of refinement: *Or (optional)*-refinement

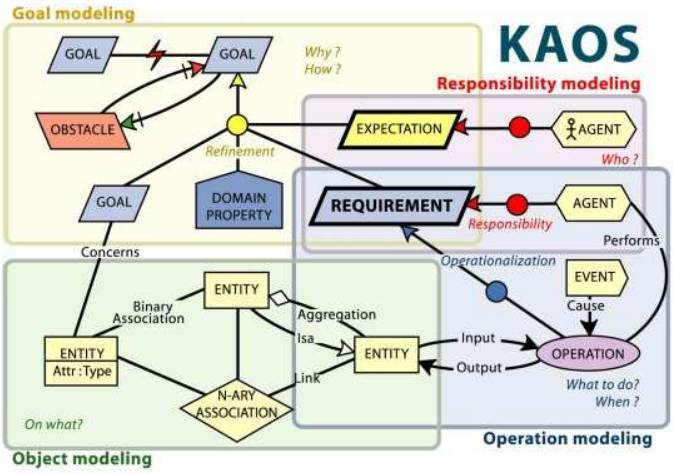

Figure 2: MM3-KAOS Meta-Model in [23]

links a parent to children where the satisfaction of any child leads to the satisfaction of the parent; *And (mandatory)*-refinement links a parent to children where the satisfaction of the parent requires the satisfaction of all the children [10]. **ObstructionLink** shows an obstruction relationship between a goal and an obstacle, while **ConflictLink** shows a conflict relationship between two goals. **ResolutionLink** shows that a requirement has been used to resolve an obstacle. **OperationalisationLink** shows an operationalisation relationship between an object and a requirement; **Responsibility Link** is used to assign a requirement to a software agent, while **Assignment Link** is used to relate expectation with a software agent, other types of links and model elements can be found in KAOS tutorial available in [23].

D. The Object Model

The KAOS object model (shown in the lower left of figure 2) defines the relationship between the objects and the goals that concerns these objects. It consists of passive, and active objects that meets the requirements of a system [23]. Other elements of the object model are listed below. **Entity** is a passive objects that satisfies a given goal of a system [23], for instance *credit card* is an entity that will satisfy the goal *Successful transaction* **Agents** are active physical objects such as human *e.g.*, *Vendor*; or logical object such as machine *e.g.*, *Router* capable of performing operations in a system, [23]. **SoftwareAgent** is a type of agent that has logical manifestation in a system *e.g. Router*; while an **EnvironmentAgent** is a type of agent that performs physical operation in a system *e.g.* [23]. **Association** is a special type of link used to show the relationship between system objects [23].

E. The Operation Model

The KAOS operation model (shown in lower right of figure 2 describes all the operations that agents performs in order to satisfy a given goal [23], for instance the goal *successful transaction* can be satisfied when the agent *Customer* performs the operation *insert card*. The KOAS operation model contains the following model elements: **Operation** is used to describe the behaviours or actions of an agent [23], for instance a *Customer* can *insert card or input pin* **Events** are significant change of state of produced by operations [23]. **Input Link** shows an input relationship between an object (entity) and the operation it performs [23]. **Output Link** shows an output relationship between an object (entity) and its operation [23].

F. The Responsibility Model

KAOS responsibility model (figure 2, upper right) defines the relationship between goals and object, by showing which agent is responsible for what requirement and expectation, using a responsibility link. Its constructs are derived from both the goal model and the object model [23]. It has the following elements: **Responsibility Link** is used to assign a leaf goal to an agent, it is represented by either red circle (showing that the leaf goal is a requirement) or a pink circle (showing that the leaf is an expectation) [23].

III. Related Work

In their research Lamsweerde and Letier propose a stepwise approach for checking the completeness of KAOS model which is based on *Temporal Logic*, and using *pre, post, and trigger* conditions to specify a set of KAOS *Elaboration Criteria* [25]. Their approach involves four *Elaboration Criteria* for checking well formedness of KAOS model, these include: *Goal Elaboration, Object/Operation Capture, Goal Operationalisation, and Responsibility Assignment*. In *Goal Elaboration* criteria, all *Goals* are refined using the *AND/OR Refinement* links until they become *Leaf Goals (Expectation, Requirement or Domain Property)*; *Object/Operation Capture* criteria ensures that *Goals* are conceptually linked to *Objects(Agents)*, and *Operations* are identified; while *Goal Operationalisation* criteria checks that each *Object* performs *Operations* to satisfy a given *Goal*. In *Responsibility Assignment* all Operations must be assigned to *Agents* [25]. These *Elaboration Criteria* corresponds to the *Completeness Criteria* defined in KAOS tutorial [23], which are expressed textually without any formalisation or automation. A *Goal-Question-Metric-GQM* has been proposed in [9] as a technique for measuring the completeness and complexity of KAOS model. The *GQM* approach uses three layers of abstraction which includes the *Conceptual, Operational, and Quantitative Layers*. The parent *Goal* are identified at the *Conceptual Layer*, in *Operational Layer* the *Goals* are refined into questions/criteria similar to KAOS elaboration and completeness criteria described above, while the *Quantitative Layer* defines a set of metrics used to check that the criteria in the operational layer are met [9], once these questions have been answered the KAOS model is presumed complete. A common setback with these approaches is that there is no form of automation in them, and they can be difficult, rigorous, error prone, and costly.

Our aim is to implement a tool that will automate *Completeness Checks* in KAOS, thus reducing the rigours, cost, and errors involved in manual techniques such as the use of

Temporal Logic, and GQM, and improving the use of mechanical language processing to integrate GORE with MDE. To achieve this, we consolidate the KAOS *Elaboration Criteria proposed in [25], the Completeness Criteria proposed in [23], and the GQM in [9] into three Completeness Checks itemized below, and implement a tool that automate them:*

- ***Completeness Check 1****-All Goals must be refined until they become either Leaf Goals (Expectation or Requirement) or Domain Property (DomainHypthesis or DomainInvariant.*
- ***Completeness Check 2****-All Leaf Goals must be assigned to Agents (SoftwareAgent or EnvironmentAgent.*
- ***Completeness Check 3****-All SoftwareAgents must be assigned to Operations.*

In the following Sections we explain the processes involved in automating these checks.

IV. The Tooling Process

In this Section we describe the processes involved in implementing the tool that automates the three Completeness Checks defined in Section III. We use Eclipse Modelling Framework-EMF *as the development platform for our project.* EMF *is an open source tool, ubiquitous, and provide basis for Model Driven Development. The processes involved in the development of our tool is summarised in Figure 4, as follows: First we develop a complete KAOS Meta-Model, then using this Meta-Model we implement a Graphical Editor* (KTool) *for constructing KAOS models, and then design an integration plugin that automate the Completeness Checks in* KTool. *Each of these steps is described below.*

A. Our KAOS Meta-Model

Our intention is to implement KTool *in an MDE tooling environment that supports tool generation from a completely defined, and java annotated KAOS Meta-Model. We found and analysed six existing KAOS Meta-Models in literature and labelled them as* MM1 to MM6 *as shown in Table I. Our analysis suggests that none of the existing KAOS Meta-Models is complete relative to KAOS model elements needed for the Completeness Checks. For instance the* Domain Property *element needed for* Completeness Check 1 *is not contained in Meta-Models (MM1, MM2, and MM4, and MM6), therefore our second contribution is to consolidate six existing KAOS Meta-Models found in literature into a complete Meta-Model suitable for MDE tooling, and MLP. The result of our analysis is shown in Table I, KAOS model elements gathered in literature are shown at the left hand side of the table, while the six Language definitions (Meta-Models) found in literature are shown as* MM1 to MM6 *at the right hand side of the table. We mapped the key concepts (model elements) of KAOS against these six approaches found in literature starting from MM1 to MM6. The key concepts contained in each Meta-Model are shown as* $\surd$*, while those lacking are marked with x. Key concepts which are not explicitly defined in the Meta-Model are shown as* ∂*. This mapping suggests that each of these approaches either lack some KAOS model elements/key concepts, or focus exclusively on concrete syntax, and are thus incomplete, and unsuitable for MDE tooling, and MLP.*

In Figure 3, we present the abstract syntax *for our proposed complete KAOS Meta-Model. The* root element/container-KAOS *contains two* abstract super classes- **Node** *and* **Link**. *A Node is used to represent a model elements (key concept) such as* Agent, and Goal, *while a Link represents the relationship between model elements. For instance* ObstructionLink *assigns* Goals *to* Obstacles, *thus showing an obstruction relationship between a Goal and an Obstacle. A* Node *can be any of these three abstract classes:* **RefinableNode, Object, and OperationNode**. ***RefinableNode*** *is an abstract class for those model elements which can be decomposed into sub-nodes. e.g* Goal and Obstacle *are types of RefinableNode because they can be refined into SubGoals, and SubObstacles respectively. A Goal or SubGoal can be refined into* Expectation and Requirement, *while Obstacle can be refined into SubObstacle. We use the same class to depict both Obstacle and SubObstacle since there are no major different between them.* Domain Properties *are conditions attached to Goal thus we classify them as types of Goal. DomainInvariant, and DomainHypothesis are types of Domain Property.* Object *is an abstract class for both active and passive model elements such as Agents and Entity respectively. Agents are active entities and can be either SoftwareAgent or EnvironmentAgent. SoftwareAgent are responsible for Requirements, while EnvironmentAgent are assigned to Expectations. Entities are passive objects/material used by agents to satisfy a Goal e.g. Credit Card, while Associations are used to show relationships between passive objects. The* OperationNode *is used to abstract operations elements such as Operations and Events. When Agents performs Operation, Events are produced [23].*

There are so many types of Link *in KAOS as defined in [23], each link is used to show the type of relationship that can exist between two or more model elements. For instance a ConflictLink shows the relationship between two Goals which are in Conflict, a ResolutionLink assigns an Obstacle to a Requirement that resolves it, while ResponsibilityLink assigns Agents to Requirements. Other types of Link are all shown in Figure 3, and their descriptions can be found in Section II-B, and in KAOS tutorial [23] .*

B. KAOS Tooling

We implement our KAOS *tool called* KTool *in Eclipse Graphical Modeling Framework (GMF) using* EuGENia-*a platform for Model Driven Software Development [14]. The diagram in Figure 4 describes the steps involved in* KAOS *tooling. First we design the* abstract syntax or Meta-Model *for* KAOS *using Eclipse Modeling Framework (EMF/Ecore) and annotate it as* KAOS.ecore. *The next step involves converting the* KAOS.ecore *model to an EmFaTic file and annotating it with java so as to describe the attributes of the objects/nodes and their connectors/links. EmFaTic is a dependent development environment that allows *.ecore files to be annotated with Java. The* EmFaTic code *that generates our KAOS tool is shown in the later part of Section IV-B, Line 1 declares the*

Table I: Mapping of key KAOS concepts against the existing approaches

KAOS Key Concepts \ Existing Meta-Models	MM1 [20]	MM2 [6]	MM3 [23]	MM4 [10]	MM5 [17]	MM6 [26]
Goal	√	√	√	√	√	√
Domain Property	x	x	√	x	√	∂
Domain Invariant	x	x	∂	x	√	x
Domain Hypothesis	x	x	∂	x	√	x
Requirement	√	∂	√	√	√	√
Expectation/Assumption	√	∂	√	√	√	√
Obstacle	√	x	√	∂	x	∂
Conflict	∂	√	∂	√	x	√
Refinement Link	x	∂	√	x	√	x
And refinement	√	∂	∂	∂	∂	∂
Or Refinement	√	∂	∂	∂	∂	∂
Obstruction Link	√	x	∂	x	x	x
Resolution/Solution link	√	x	∂	x	x	∂
Conflict Link	x	x	∂	x	x	x
Operationalization link	x	x	∂	x	x	x
Object	√	√	∂	√	√	√
Entity	√	√	√	√	√	√
Agent	√	√	√	√	√	√
Software Agent	∂	x	∂	∂	√	∂
Environment Agent	∂	x	∂	∂	√	∂
Association	x	x	√	x	√	x
Operation/Action	x	√	√	√	√	√
Event	x	√	√	√	√	√
Input Link	∂	x	∂	x	x	x
Output link	∂	x	∂	x	x	x
Assignment link	∂	x	∂	x	x	x

Legend:
MM-Meta-Model;
√-included in the Meta-Model
x-Not Included in the Meta-Model
∂-Implied in the paper but not included in the meta-Model

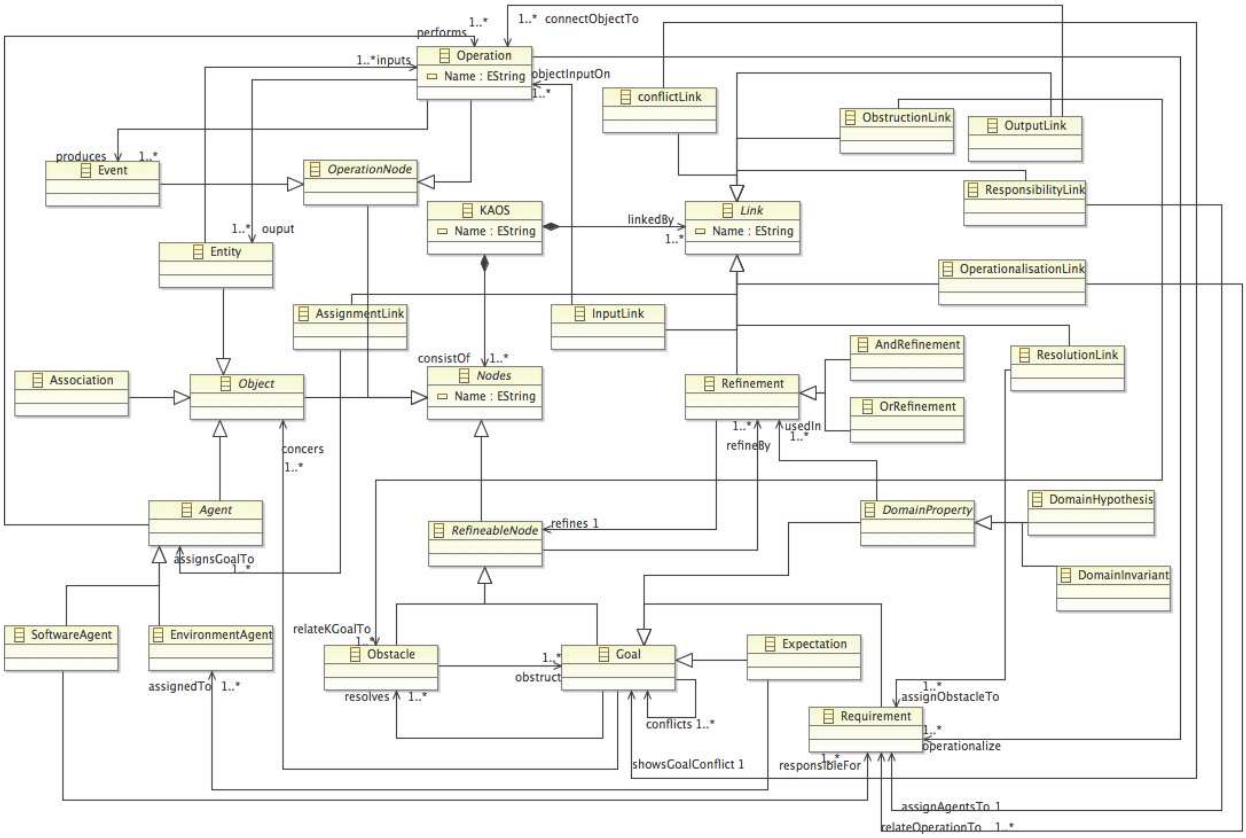

Figure 3: Our Proposed Meta-model

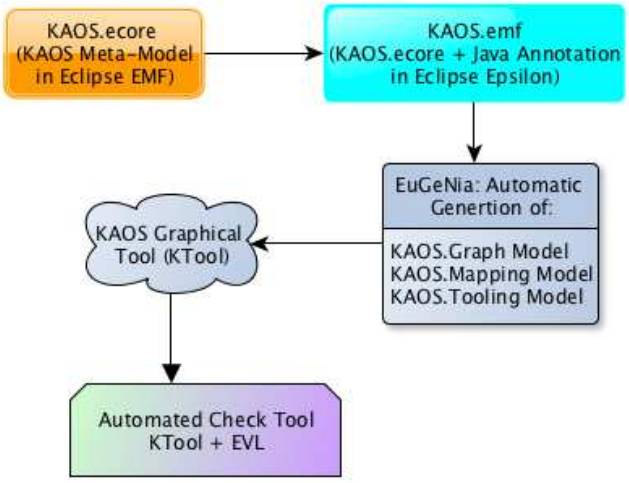

Figure 4: The KAOS Tooling Processes

namespace *which specifies the location of* http://Kaos.ecore *as "http://kaos/1.0". Line 5 tells* EuGENia *that the root element or container is* kaos*, and therefore should not be included in the diagram. The* gmf.node *annotations from Line 17 to 70, and Line 112, 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 palette, see Figure 5.* Connectors or Links *are specified in EmFaTic file using the* gmf.link *annotations as seen in Line 10, and Lines 74 to 107, and defines attributes such as target-decorations, source, and target of the connector. After annotating all the* KAOS.ecore *model with java, then we used EuGeNia to generate the models that describes the attributes of KAOS graphical editor. 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 [14]. Once these models are generated, then the diagram plugin for KAOS is created, and finally after running a new eclipse runtime the KAOS graphical editor is generated.*

The graphical convention for our KAOS tool uses a 'qualified name of a Java class that implements Figure'. Goal, and SubGoal *are represented with a 'rounded rectangle' without an icon, while* Obstacles *are represented with a 'rounded rectangle' that has an icon.* Expectation *is represented with a 'rounded rectangle' with a 'dashed' thick border, while* Requirement *is modelled as a 'rounded rectangle' with a 'dotted' thick border.* Domain Property *is represented as an 'ellipse' with Icon, a 'dotted' ellipse implies that the domain property is an invariant, while a 'dashed' ellipse shows that it is a hypothesis. A* Software Agent *is modelled as 'rectangle' with an icon, while* Environmental Agent *is modelled as a 'rectangle' without an icon. An* Entity *is modelled as an 'ellipse' without an icon, while an* Operation *is an 'ellipse' with an icon. We use an 'ellipse' that has an icon and a 'dotted' border to model* Events*, while a solid 'ellipse' is used for the* And Refinement*, and a hollow 'ellipse' is used for* Or Refinement*. Similarly the links are annotated with* 'qualified name of a Java class that implements the org.eclipse.draw2d.RotatableDecoration interface'. *All the Links are connectors with different heads/pointer: the* Association Link *is modelled with a 'filledsquare' head, the* Operationalisation Link *has a 'arrow' head,* obstruction Link *has a 'filledrhomb' head,* Conflict Link *has a 'square' head. The* output Link *is 'dotted' connector with a 'filledsquare' head, and the* Input Link *is a 'dashed' connector with a 'filledsquare' head. The* Resolution Link *is 'dotted' connector with a 'square' head, while* Refinement Link *has a 'filled-closedarrow' head, and the* Assignment Link *is a 'dashed' connector with a 'filledsquare' head. These connectors and objects are shown in the palette of Figure 5, as used in the modelling of our case study.*

```
1  @namespace(uri="http://kaos/1.0",
2  prefix="kaos")
3  package kaos;
4
5  @gmf.diagram(foo="bar")
6  class Kaos {
7    val Link[+] has;
8    val Node[+] contains;}
9
10 @gmf.link(target.decoration="filledclosedarrow",
11 source="froma", target="toa", color="0,0,0")
12 abstract class Link {
13   attr String name;
14   ref Node[1] froma;
15   ref Node[1] toa;}
16
17 @gmf.node(label="name")
18 abstract class Node {
19   attr String name;}
20
21 @gmf.node(label.icon="false",
22 boder.color="0,0,0")
23 class Goal extends Node {}
24
25 @gmf.node(label="name")
26 class Obstacle extends Node {}
27
28 @gmf.node(label="name", border.style="dash",
29 border.width="3")
30 class Expectation extends Node {}
31
32 @gmf.node(label="name", border.style="dot",
33 border.width="3")
34 class Requirement extends Node {}
35
36 @gmf.node(label="name", figure="ellipse",
37 border.style="dot", boder.color="0,0,0")
38 class DomainInvariant extends Node {}
39
40 @gmf.node(label="name", figure="ellipse",
41 border.style="dash", boder.color="0,0,0")
42 class DomainHypothesis extends Node {}
43
44 @gmf.node(label="name", figure="rectangle")
45 class SoftwareAgent extends Node {}
46
```

```
@gmf.node(label="name", figure="rectangle",
label.icon="false")
class EnvironmentAgent extends Node {}

@gmf.link(target.decoration="filledsquare",
source="froma", target="toa")
class Association extends Link {}

@gmf.node(label="name", figure="ellipse",
label.icon ="false")
class Entity extends Node {}

@gmf.node(label="name", figure="ellipse")
class Operation extends Node {}

@gmf.node(label="name", figure="ellipse",
border.style="dot")
class Event extends Node {}

@gmf.node(label.icon="false",figure="ellipse",
color="0,0,0", border.color="0,0,0", size="1,1")
class And extends Node {}

@gmf.node(label.icon="false",figure="ellipse",
border.color="0,0,0", size="1,1")
class Or extends Node {}

@gmf.link(target.decoration="square",
source="froma", target="toa")
class OperationalizationLink extends Link {}

@gmf.link(target.decoration="arrow",
source="froma", target="toa")
class ResponsibilityLink extends Link {}

@gmf.link(target.decoration="filledrhomb",
source="froma", target="toa")
class ObstructionLink extends Link {}

@gmf.link(target.decoration="square",
source="froma", target="toa")
class ConflictLink extends Link {}

@gmf.link(target.decoration="filledsquare",
source="froma", target="toa", style="dot")
class OutputLink extends Link {}

@gmf.link(target.decoration="filledsquare",
source="froma", target="toa", style="dash")
class InputLink extends Link {}

@gmf.link(target.decoration="square",
source="froma", target="toa", style="dot",
width="2")
class ResolutionLink extends Link {}

@gmf.link(target.decoration="filledclosedarrow",
source="froma", target="toa")
class Refinement extends Link {}

@gmf.link(target.decoration="filledsquare",
source="froma", target="toa", style="dash",
width="2")
class AssignmentLink extends Link {}

@gmf.node(label.icon="false", boder.style="dash")
class SubGoal extends Node {}
```

As a demonstration, we apply our KTool *to construct a simple model of the case study described in Section II, for the sake of space and page limitation we did not show all the sub-models of KAOS in this example, we only constructed a simplistic model of the case study using few model elements.*

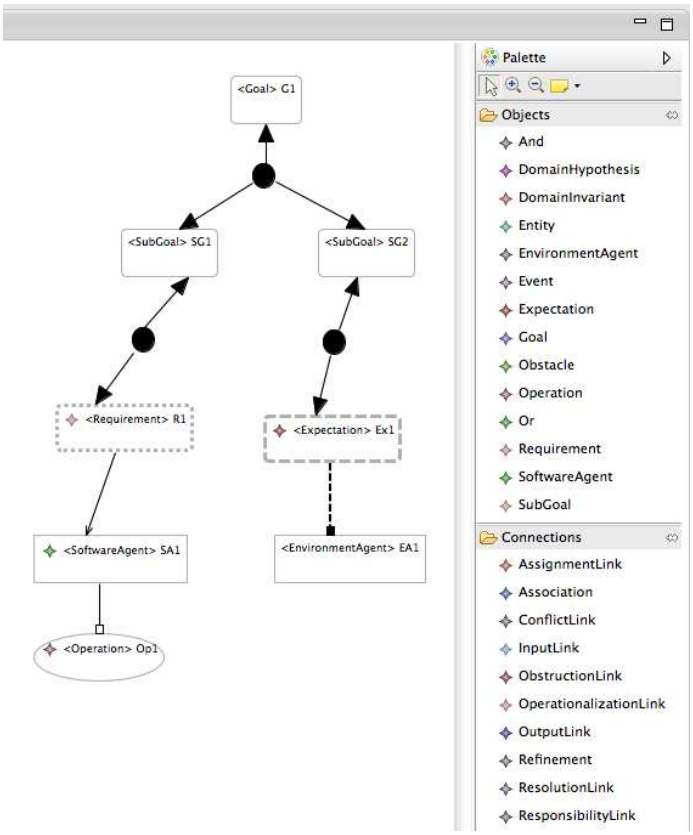

Figure 5: Example of KAOS Model in our KTool

As shown in Figure 5 the root Goal of Bank XYZ G1-Provide excellent Point of Sale (POS) Services means that all transactions must be successful, and secure. G1 is then refined using the And Refinement *into two SubGoals: SG1-Secure Transaction from Hackers, and SG2-Successful Transaction. In order to ensure Secure Transactions from Hackers, the Smart Card Information should be encrypted, thus we refine the SubGoal: SG1-Secure Transaction from Hackers into the Requirement: R1-Smart Card Information encrypted and assign it to the SoftwareAgent: SA1-Encrypted Router using the ResponsibilityLink. Similarly to ensure Successful Transaction all Customers should be provided with a Smart Card, thus we refine the SubGoal: SG2-Successful Transaction into the Expectation:Ex1-Provide Smart Card to Customers and assign it the EnvironmentAgent: EA1-Bank XYZ using the AssignmentLink. The SoftwareAgent: SA1 is then assigned to an Operation: OP1 using the OperationalisationLink.*

V. Integration Plugin: EVL + KTool

The Final steps in our KAOS Tooling Process *is to create an integration plugin that allows our* KTool *to automatically check the completeness of KAOS Model. We started the integration plugin development by first creating a new* eclipse plugin project *where the constraint can be defined. This is followed by adding the location of the user interface, and the emf validation to the list of dependencies. Then we define the* completeness checks *as a set of constraints written in*

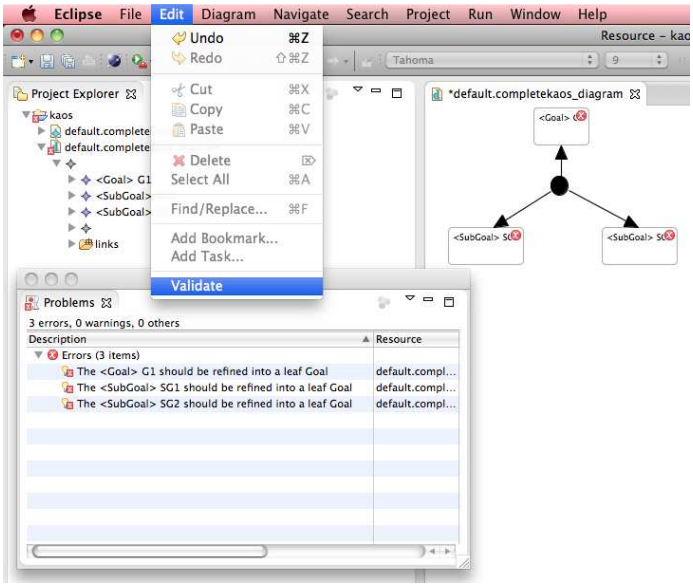

Figure 6: Completeness Check 1a

Epsilon Validation Language-EVL. EVL *is a Domain Specific Language (DSL) used to define and validate a set of constraints that can check the completeness or well formedness of a model [13]. The EVL codes for* completeness checks 1, 2, and 3 *are shown and further explained in Sections V-A, V-B, and V-C. The next step is to define the extensions that will allow us to bind the EVL constraints to our* `KTool`*, and final we run the new configuration to ensure that the plugin development has been implemented correctly. A detailed step by step process for creating an integration plugin in* Ecplipse Modelling Framework-EMF *can be found in the Epsilon Project.*[1] *In the following Sections, we use our case study to describe how Completeness Checks 1, 2, and 3 have been implemented.*

A. Completeness Check 1

Completeness Check 1 ensures that all Goals and SubGoals are refined to Leaf Goals (Expectation, or requirement) or into Domain Property using either And Refinement or the Or Refinement. Therefore Completeness Check 1 is satisfied when every Leaf Goal is Requirement, or Expectation, or DomainHypothesis, or DomainInvariant. The EVl *constraint for Completeness Check 1 is shown in the listing below:*

```
1  context Goal{
2    constraint ToLeaf{
3      check: Refinement.all.exists
4      (g|g.froma.isKindOf(And) or
5      g.froma.isKindOf(Or) and
6     g.toa.isKindOf(Requirement) or
7     g.toa.isKindOf(Expectation) or
8     g.toa.isKindOf(DomainInvariant) or
9     g.toa.isKindOf(DomainHypothesis))
10       message:"The " + self.name +
11       " should be refined into a leaf Goal"
12  }
```

[1]http://www.eclipse.org/epsilon/doc/articles/evl-gmf-integration/

```
13  }
14
15  context SubGoal{
16    constraint ToaLeaf{
17      check: Refinement.all.exists
18      (g|g.froma.isKindOf(And) or
19      g.froma.isKindOf(Or)
20      and
21       g.toa.isKindOf(Requirement) or
22       g.toa.isKindOf(Expectation) or
23       g.toa.isKindOf(DomainInvariant) or
24     g.toa.isKindOf(DomainHypothesis))
25        message:"The " + self.name +
26         " should be refined into a leaf Goal"
27  }
28  }
```

The term context in Line 1 is an EVL *keyword showing the model element where the constraint is defined, in this case Goal. The constraint keyword in Line 2 specifies that the name of the constraint is ToLeaf, while the check keyword in Line 3 defines the algorithm for the constraint. The message keyword in Line 10 shows the output message if the check is not satisfied. To demonstrate Completeness Check 1, we use our KTool to construct a simple but incomplete KAOS model of the case study we described in Section II. As shown in Figure 6 the root Goal of Bank XYZ G1-Provide excellent Point of Sale (POS) Services is refined into SG1-Secure Transaction, and SG2-Successful Transaction, then we validate this KAOS model by clicking on the validate button in* Eclipse Modelling Framework-EMF *as shown in Figure6. When the validate command is executed, our tool checks if this KAOS model satisfies the constraints for completeness check 1. If this is true, there will be no error sign in context model elements concerned, or error message in the problem dialogue box of EMF. But if false (as in this case) an error signs will appear in the context model elements in this case G1, SG1, and SG2; this will be accompanied by error messages, displayed in the problem dialogue box. The error messages* "The ¡Goal¿ G1 , and The ¡SubGoals¿ SG1, and SG2-should be refined into a leaf Goal" *indicates that the model is incomplete and malformed but gives instruction on what the user should do to make the model complete.*

For the model to be well formed/complete, the leaf goal should either be a Requirement, Expectation, DomainInvariant or DomainHypothesis. *Thus we further refine the SubGoals: SG1 and SG2 into Leaf Goals- Requirement: R1, and Expectation: EX1 respectively , and revalidate it. The result of the revalidated model is shown in Figure 7. The error signs have moved from conetx G1, SG1, and SG2 to R1 and Ex1, this means that although the Goals and SubGoals have been completely refined into Leaf Goals, and Completeness Check 1 has been satisfied, the model is not yet complete because Completeness Check 2 has not been satisfied.*

B. Completeness Check 2

In Completeness Check 2 each Leaf Goals must be assigned to at least one Agent, this implies that Expectation *must be assigned to an* Environment Agent *using* AssignmentLink, *and* Requirement *must be linked to a* SoftwareAgent *using*

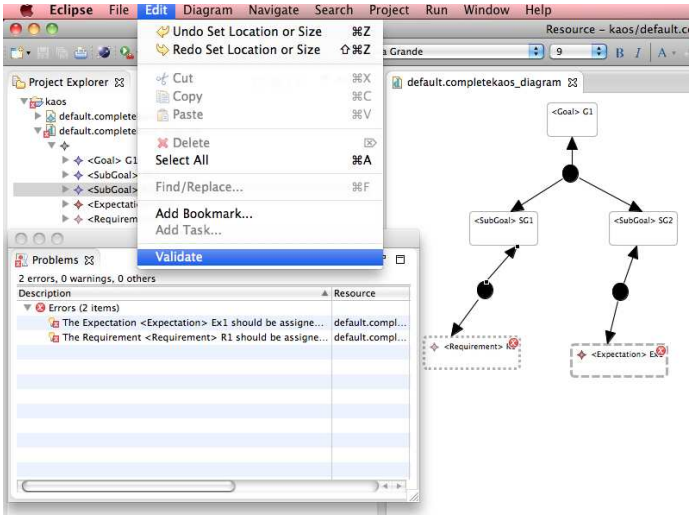

Figure 7: Completeness Check 1b

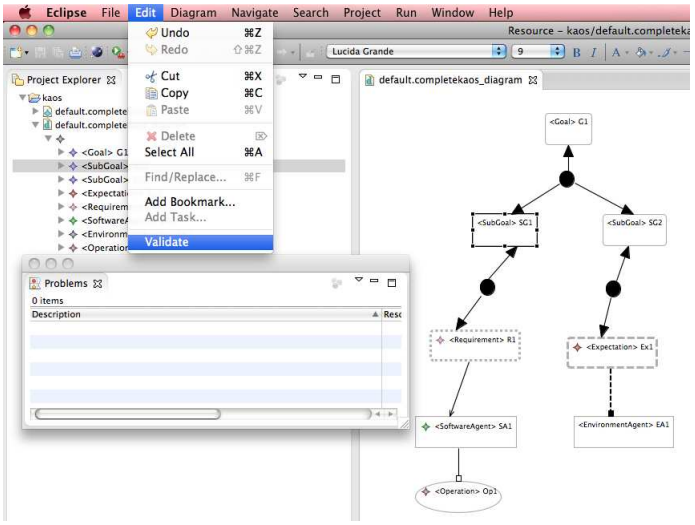

Figure 9: Fixing Completeness Check 3

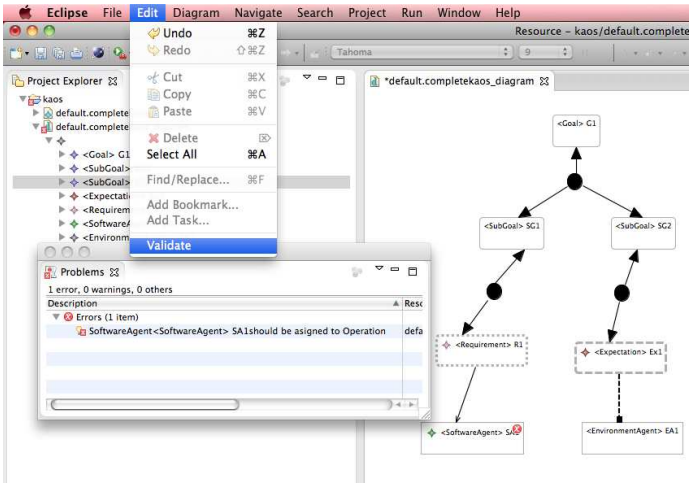

Figure 8: Completeness Check 2

Responsibility Link. *The* EVL constraint *for Completeness Check 2 is shown in the listing below. The error messages and signs in Figure 7 shows that the Requirement: R1, and the Expectation: Ex1 have not been assigned to Software and Environment Agents respectively, thus Completeness Check 2 has not been satisfied.*

```
context Requirement{
  constraint AssignedToSoftwareAgent{
    check: ResponsibilityLink.all.exists
     (I|I.froma=self and
    I.toa.isKindOf(SoftwareAgent))
      message:"The Requirement " + self.name +
       " should be assigned to a Software Agent"
  }

}
context Expectation{
  constraint AssignedToEnvironmentAgent{
    check: AssignmentLink.all.exists
    (I|I.froma=self and
    I.toa.isKindOf(EnvironmentAgent))
      message:"The Expectation " + self.name +
       " should be assigned to an Environment Agent"
  }

}
```

To satisfy Completeness Check 2, we simply assign a SoftwareAgent:SA1-Encrypted Router to Requirement-R1, and an Environment Agent-EA1-Bank XYZ to the Expectation-Ex1, and then re-validate the model. The new model in Figure 8 shows that each Leaf Goal has been assigned to at least one Agent, and Completeness Check 2 has been satisfied. However the model is not yet complete because the SoftwareAgent: SA1 has not been operationalised. This will be done in Completeness Check 3.

C. Completeness check 3

Completeness Check 3 can be satisfied each SoftwareAgent is assigned to an Operation using the OperationalisationLink. The EVL constraint for this check is shown in the listing below.

```
context SoftwareAgent{
 constraint AssignedToOperation{
   check:OperationalizationLink.all.exists
   (l|l.froma=self and l.toa.isKindOf(Operation))
    message:"SoftwareAgent" + self.name +
     "should be asigned to a Operation"

  }
}
```

To satisfy Completeness Check 3, we assign an Operation: OP1 to the SoftwareAagent using OperationalisationLink and revalidate the model. The result as shown in Figure 9 is a complete KAOS model without of error signs and error messages.

VI. Conclusion

In this research we propose a tool that can automatically check the completeness, and well formedness of KAOS model. Apart from other benefits of model checking, completeness check in KAOS can reduce the cost associated with incomplete

Goal Oriented Requirement Models, *especially when using KAOS as a basis for integrating GORE with MDE. Automating the completeness model checking in KAOS can reduce the error, difficulty, and cost associated with existing manually or semi-automatic KAOS completeness checks which can benefit the requirement engineering community in both industry and research. We aim to take this work further, thus in our future work we tend to demonstrate how a complete KAOS model can facilitate MLP by proposing an automatic transformation from a textually defined Goal model to KAOS and vice versa.*

REFERENCES

[1] *Fernanda M. Alencar, Beatriz Marin, Giovanni Giachetti, Oscar Pastor, Jaelson Castro, and Joao Henrique Pimentel. From i* requirements models to conceptual models of a model driven development process. In* PoEM*, pages 99–114, 2009.*

[2] *Annie Anton. Goal-based requirements analysis. In* ICRE*, pages 136–144, 1996.*

[3] *C. Asuncion, D. Quartel, S. Pokraev, M. Iacob, and M. van Sinderen. Combining goal-oriented and model-driven approaches to solve the payment problem scenario. In* 8th Semantic Web Services Challenge Workshop, SWSC 2009, Eindhoven, The Netherlands*. SEALS Project (Semantic Evaluation at Large Scale), 2010.*

[4] *David Baccarini, Geoff Salm, and Peter ED Love. Management of risks in information technology projects.* Industrial Management & Data Systems*, 104(4):286–295, 2004.*

[5] *Salah Baina, Pierreyves Ansias, Michael Petit, and Annick Castiaux. Strategic business/it alignment using goal models.*

[6] *Anne Dardenne, Axel van Lamsweerde, and Stephen Fickas. Goal-directed requirements acquisition.* Sci. Comput. Program.*, 20(1-2):3–50, 1993.*

[7] *Wilco Engelsman, Dick Quartel, Henk Jonkers, and Marten van Sinderen. Extending enterprise architecture modelling with business goals and requirements.* Enterprise Information Systems*, 5(1):9 – 36, 2011.*

[8] *Wilco Engelsman and Roel Wieringa. Goal-oriented requirements engineering and enterprise architecture: Two case studies and some lessons learned. In* REFSQ*, pages 306–320, 2012.*

[9] *P. Espada, M. Goulao, and J. Araujo. Measuring complexity and completeness of kaos goal models. In* Empirical Requirements Engineering (EmpiRE), 2011 First International Workshop on*, pages 29 –32, aug. 2011.*

[10] *W. Heaven and A. Finkelstein. Uml profile to support requirements engineering with kaos.* Software, IEE Proceedings -*, 151(1):10 – 27, feb. 2004.*

[11] *Sommerville Ian.* Software Engineering*. Addison Wesley, 6th edition, 2006.*

[12] *E. Kindler. Model-based software engineering: the challenges of modelling behaviour. In* Proceedings of the Second International Workshop on Behaviour Modelling: Foundation and Applications*, page 4. ACM, 2010.*

[13] *Dimitrios Kolovos, Louis Rose, Richard Paige, and Fiona AC Polack. The epsilon book.* Structure*, 178:1–10, 2010.*

[14] *Dimitrios S Kolovos, Louis M Rose, Saad Bin Abid, Richard F Paige, Fiona AC Polack, and Goetz Botterweck. Taming emf and gmf using model transformation. In* Model Driven Engineering Languages and Systems*, pages 211–225. Springer, 2010.*

[15] *Alexei Lapouchnian. Goal oriented requirement engineering: An overview of the current research, 2005.*

[16] *Ivan Markovic and Marek Kowalkiewicz. Linking business goals to process models in semantic business process modeling. In* EDOC*, pages 332–338, 2008.*

[17] *Raimundas Matulevicius and Patrick Heymans. Analysis of kaos metamodel: Technical report, 2005.*

[18] *Raimundas Matulevicius and Patrick Heymans. Comparing goal modelling languages: An experiment. In* REFSQ*, pages 18–32, 2007.*

[19] *R. Monteiro, J. Arau andjo, V. Amaral, and P. Patri andcio. Mdgore: Towards model-driven and goal-oriented requirements engineering. In* Requirements Engineering Conference (RE), 2010 18th IEEE International*, pages 405 –406, 27 2010-oct. 1 2010.*

[20] *R. Monteiro, J. Araujo, Vasco Amaral, M. Goulao, and P. M. B. Patricio. Model-driven development for requirements engineering: The case of goal-oriented approaches. In Ricardo Machado Joao Pascoal Faria, Alberto Silva, editor,* 8th International Conference on the Quality of Information and Communications Technology (QUATIC 2012)*, number 8 in Quality of Information and Communications Technology, pages 75–84. IEEE Computer Society, 09 2012.*

[21] *OMG. Business motivation model, 2010.*

[22] *D. Quartel, W. Engelsman, H. Jonkers, and M. van Sinderen. A goal-oriented requirements modelling language for enterprise architecture. In* Enterprise Distributed Object Computing Conference, 2009. EDOC '09. IEEE International*, pages 3 –13, sept. 2009.*

[23] *Respect-IT. A kaos tutorial, 2007.*

[24] *D.T. Ross and Jr. Schoman, K.E. Structured analysis for requirements definition.* Software Engineering, IEEE Transactions on*, SE-3(1):6–15, Jan 1977.*

[25] *Axel van Lamsweerde and Emmanuel Letier. Handling obstacles in goal-oriented requirements engineering.* IEEE Transactions on Software Engineering*, 26:978–1005, 2000.*

[26] *Vera Maria Bejamim Werneck, Antonio de Padua Albuquerque Oliveira, and Julio Cesar Sampaio do Prado Leite. Comparing gore frameworks: i-star and kaos. In* Ibero-American Workshop of Engineering of Requirements*, Val Paraiso, Chile, July 2009.*

[27] *Khim Teck Yeo. Critical failure factors in information system projects.* International Journal of Project Management*, 20(3):241–246, 2002.*

[28] *Eric Yu. Modeling strategic requirement for process reengineering, 1995.*

[29] *Eric Yu, Paolo Giorgini, Neil Maiden, and John Mylopoulos. Social modeling for requirement engineering:an introduction, 2009.*