Specifying Consistency Constraints for Modelling Languages

Lijun Shan
Department of Computer Science
National University of Defence Technology
Changsha, 410073, China
Email: lijunshancn@yahoo.com

Hong Zhu
Department of Computing
Oxford Brookes University
Oxford OX33 1HX, UK
Email: hzhu@brookes.ac.uk

Abstract. As graphic modelling languages play an increasingly important role in software development, completeness and consistency have become essential quality attributes of software models. This paper presents a framework for the formal definition of the abstract syntax and type systems of modelling languages that facilitates formal specification and automatic checking of consistency and completeness constraints on graphic models. The approach is illustrated by the specification of CAMLE modelling language and its consistency and completeness constraints. An empirical study of the effectiveness of the framework is reported, which shows that about 85% of errors generated by mutation operators can be detected by automatic consistency checking.

1. Introduction

Modelling languages are playing an increasingly important role in software development as model-driven software development methodology is gaining wide acceptance. Typical modelling languages include UML for object oriented software development [1], Yourdon notation and SSADM for structured analysis and design, CAMLE modelling language [2] for the emerging agent-oriented software development, etc. Well-defined visual notations for modelling software systems balance well between readability and preciseness due to their semi-formal nature. As a means of separation of concerns, the multiple-views principle has been widely adopted in modern modelling languages. By representing different aspects of a system in different views and/or at different levels of abstractions, it provides a powerful vehicle for dealing with the complexity of information systems. However, as pointed out in [3, 4], maintaining consistency between views and completeness of the models is crucial, but difficult. It is highly desirable to automatically check models' consistency and completeness; yet, graphic models must be well-formed to be processed and transformed. Unfortunately, these tasks are by no means trivial. For example, UML [1] does not systematically and explicitly define consistency and completeness constraints, though OCL [5] provides a language facility for specifying constraints on the instances of models. Many research efforts addressing the consistency problems of UML have been reported, c.f. [6, 7, 8, 9, 10]. Although graphic modelling languages such as UML are widely accepted and new modelling languages are being developed, how to define the syntax and semantics of graphic modelling languages is still an open problem.

In [11], we proposed a framework for the definition of the abstract syntax and type systems of graphic modelling languages so that well-formedness, consistency and completeness of graphic models can be formally specified and automatically checked. To demonstrate the effectiveness of the approach, this paper exemplifies the framework with the CAMLE language by defining its abstract syntax and type system and specifying its consistency and completeness constraints. An experiment with CAMLE consistency checking tool is also reported in this paper. It shows that consistency constraints specified and implemented in our approach can achieve an error-detecting rate around 85%.

The remainder of the paper is organised as follows. Section 2 reviews the framework proposed in [11]. Section 3 presents the definition of CAMLE language. Section 4 reports the experiment with the effectiveness of consistency and completeness checking. Section 5 concludes the paper with a discussion of related work and further works.

2. Overview of the framework

2.1 Type systems and well-formedness

Let's first formally define a few basic concepts of modelling languages; see [11] for more details.

A modelling language $\mathcal{ML}$ is called a *multiple-views modelling language*, if it defines a finite set $\boldsymbol{T}\neq\emptyset$ of *types of diagrams*. Each type $T\in\boldsymbol{T}$ of diagrams provides a set of graphical notations to represent a view of the system. A model M in $\mathcal{ML}$ consists of a set $\boldsymbol{D}\neq\emptyset$ of diagrams. Each diagram $D\in\boldsymbol{D}$ has one and only one type T_D in $\boldsymbol{T}$. We write $Type(D)$ to denote the type T_D of diagram D. The subset of diagrams of a type T in a model M is called the *T-view of the model M* (or simply the *T-sub-model* or *T-model*), written $M.T$. Formally, $M.T=\{D|D\in M, Type(D)=T\}$.

$\mathcal{ML}$ is said to be *graphically typed*, iff for each type $T\in\boldsymbol{T}$, $\mathcal{ML}$ defines a finite set N_T of node types and a finite set E_T of relation types. For each type $te \in E_T$ of relations, a relation e of type te in a diagram D of type T can only be specified on certain type(s) of nodes or relations in D.

$\mathcal{ML}$ is *annotationally typed*, iff for each type T of diagrams, the following conditions hold.

(a) For each diagram type T, $\mathcal{ML}$ defines a finite number of fields $f_{T,i}$, i=1, …n_T, for the annotations that can be associated to a diagram of type T. For each field $f_{T,i}$ $\mathcal{ML}$ defines a data type $FT_{T,i}$ of the values that can be assigned to field $f_{T,i}$.
(b) For each node or relation type t of diagrams of type T, $\mathcal{ML}$ defines a finite set of fields $f_{t,i}$, i=1, …, n_t, for the annotations that can be associated to the nodes or relations of type t. For each field $f_{t,i}$, $\mathcal{ML}$ defines a data type $d_{t,i}$ of the values that can be assigned to field $f_{t,i}$.

A modelling language $\mathcal{ML}$ is *typed*, iff it is both graphically and annotationally typed.

In a typed $\mathcal{ML}$, a diagram D of type T is *graphically well-formed*, iff each node n is associated to one and only one node type tn, and each relation e of type te on nodes n_1, …, n_k must satisfy the type requirements of te. A diagram D of type T is *annotationally well-formed*, iff the values assigned to the annotation fields of the diagrams, the nodes and the relations of the diagrams are all compatible to the data types. A model M is *well-formed* if all diagrams of M are both graphically and annotationally well-formed.

To define type systems and abstract syntax of modelling languages, a graphical extension of BNF (called GEBNF) was proposed [11], which is summarized in Table 1.

Table 1. GEBNF Notation

Notation	Meaning	Example and explanation
<X>	X is the name of a type of entities	*<Class Diagram>*: the type of entities called class diagrams.
X ::= Y	X is defined as Y	*<Model> ::= <Diagram>** : a model is defined as consisting of a number of diagrams.
X*	Repetition of X (include null)	*<Diagram>**: the entity consists of a number N of diagrams, where $N \geq 0$.
X+	Repetition of X (exclude null)	*<Diagram>+*: the entity consists of a number N of diagrams, where $N \geq 1$.
X | Y	Choice of X and Y	*<Actor node>*|*<Use case node>* means that the entity is either an actor node or a use case node.
X , Y	X and Y	*<Actor node>, <Use case node>*: an entity that consists of an actor node and a use case node.
[X]	X is optional	[*<Actor>*]: Actor is optional.
X Y	Order pairs consists of X and Y	<Actor node> <Use case node>: an element that consists of an order pair of an actor node and a use case node.
/X/	An annotation field named X	*/Use case name/*: the annotation field called use case name.
X: Y	The type of X is Y.	*/Use case name/: Text* : the type of the annotation use case name is text.
(X)	Parenthesis	It is used to change the preferences of the expression.
'abc'	The literal of a string	'*extends*': the literal value of the string 'extends'.
Text[!*F*]	Predefined type *Text* with syntax specified by F in BNF	*'Text'*: a text in any format; '*Text ! <object name> ':' <class name>'* : the text that consists of an object name and a class name separated by a colon.

2.2 Consistency and completeness constraints

Generally speaking, a *consistency constraint C* is a predicate defined on models such that $C(M)$ = *true* means that the model is consistent with respect to the constraint; otherwise, the model is inconsistent and hence, not sound. Informally, a consistency constraint restricts how models should be constructed so that certain types of conflictions in the information specified by the model can be prevented and detected. A *completeness constraint* restricts the construction of the models so that certain types of errors due to the lack of information can be prevented and detected. A violation of a consistency constraint implies that there is an error in the model due to confliction between different parts of the model. Therefore, no system can satisfy the specification of the model. In contrast, a violation of a completeness constraint implies that a certain piece of information is missing. Therefore, there will be a system that the users do not want satisfying the specification.

There are several kinds of constraints that can be defined on modelling languages.

A. Intra-diagram vs *Inter-diagram constraints.* A constraint C is *intra-diagram*, if it is defined on a diagram of a specific type T. It is *inter-diagram*, if it is defined on two or more diagrams.

B. Inter-model vs *Intra-model constraints.* A constraint C is *inter-model*, if it is defined on diagrams of more than one type; otherwise, it is *intra-model.*

For hierarchical modelling languages, constraints can also be classified into vertical and horizontal constraints, and global and local constraints.

C. Vertical vs. Horizontal constraints. A constraint C is *horizontal* if it is defined between diagrams of the same abstraction level. A *vertical* constraint C is defined between diagrams that have refinement relationships between them.

E. Local vs Global constraints. A constraint C is *global* on a particular type of diagrams, if it is defined on the whole set of diagrams of the type. Otherwise, it is *local constraint.*

Given a definition in GEBNF, a first order language can be derived as follows for the formal definition of consistency and completeness constraints. Let φ and ρ be n-ary operator and relation, respectively.

- *Expressions* are formed by finite applications of the following constructions.
  - *Variables* and *constants* are expressions;
  - $\varphi(e_1, e_2, \ldots, e_n)$ is an expression, if $e_1, e_2, \ldots, e_n$ are;
  - $e.f$ is an expression, if e is and f is a field;
  - $e.t$ is an expression, whose value is the set of the elements of type t in e, if e is an expression and t is a type;
  - *Type(e)* is an expression, if e is. *Type(e)* is e's type.
- *Statements* are formed by finite application of the following constructions.
  - $\rho(e_1, e_2, \ldots, e_n)$ is a statement, if $e_1, e_2, \ldots, e_n$ are expressions; in particular, $e_1 = e_2$, $e_1 \in e_2$ are statements.
  - $\neg\rho$, $\rho_1 \Rightarrow \rho_2$, $\rho_1 \Leftrightarrow \rho_2$, $\rho_1 \wedge \rho_2$, and $\rho_1 \vee \rho_2$ are statements, if ρ, ρ_1 and ρ_2 are statement;
  - $\forall X \in E.S$ and $\exists X \in E.S$ are statements, if X is a free variable in statement S.

3. Definition of CAMLE Language

As shown in the following GEBNF formula, a CAMLE model consists of three sub-models.

<CAMLE model>::=

<Caste model>, <Collaboration model>, <Behaviour model>

The following subsections present the abstract syntax of the sub-models and some examples of constraints. The definitions of the syntax of various text types are omitted for the sake of space.

3.1 Caste model

From agent-oriented view of information systems, an organization consists of a collection of agents. The agents stand in certain relationships by being a member of certain groups and playing certain roles, i.e. in certain castes. The caste model describes the castes in the system and the structural relationships between them. This organizational structure is captured in a caste diagram. Fig. 1 is an example.

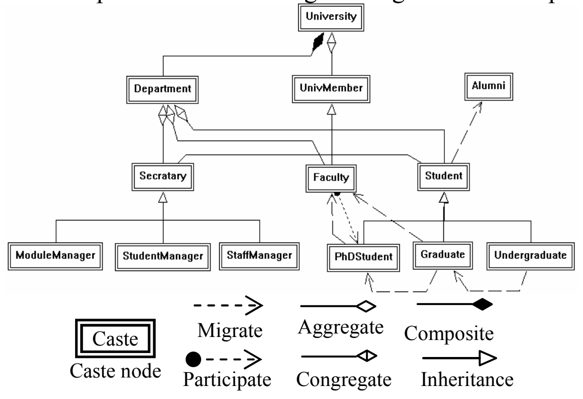

Fig 1. Caste diagram: Example and notation

In GEBNF, caste model is defined as follows.

```
<Caste model>::= <Caste diagram>
<Caste diagram>::= /Title/:Text ! 'main', <Caste node>+,
    <Inheritance relation>*, <Migration relation>*,
    <Participation relation>*, <Aggregation relation>*,
    <Congregation relation>*, <Composition relation>*
<Caste node>::= /Name/: <Caste Name>
<Inheritance relation>::= <Caste node> <Caste node>
<Migration relation>::= <Caste node> <Caste node>
<Participation relation>::= <Caste node> <Caste node>
<Aggregation relation>::= <Caste node> <Caste node>
<Congregation relation>::= <Caste node> <Caste node>
<Composition relation>::= <Caste node> <Caste node>
```

The following is an example of intra-diagram consistency constraints on caste diagram.

In a caste diagram, each node has a unique name, i.e.
$\forall D\in M$.<Caste diagram>.$\forall X, Y\in D$.<Caste node>.
(X./Name/= Y./Name/ $\Rightarrow X=Y$)

3.2 Collaboration models

Collaboration models describe the dynamic structure of a system from the perspective of communication. As shown in Fig 2, there are two types of nodes in a collaboration diagram. An agent node represents a specific agent, while a caste node represents any agent in a caste. An interaction edge from node *A* to *B* indicates that *A*'s visible actions are observed by agent *B*. The actions are annotated on the links.

The definition of collaboration model follows.

```
<Collaboration model>::= <Collaboration diagram>+
<Collaboration diagram>::=
    /Title/: Text ! [(<Agent Name> | <Caste Name>) '*']
    ('main'| <Scenario description>),
    <Agent node>*, <Caste node>)*, <Interaction>*,
    [<Environment boundary>]
<Agent node>::= /Name/: Text !<Agent Name>
<Environment boundary>::= <Caste node>*, <Agent node>*
<Interaction>:: = /Action List/: Text ! <Actions>,
  (<Agent node>|<Caste node>)(<Agent node>|<Caste node>)
```

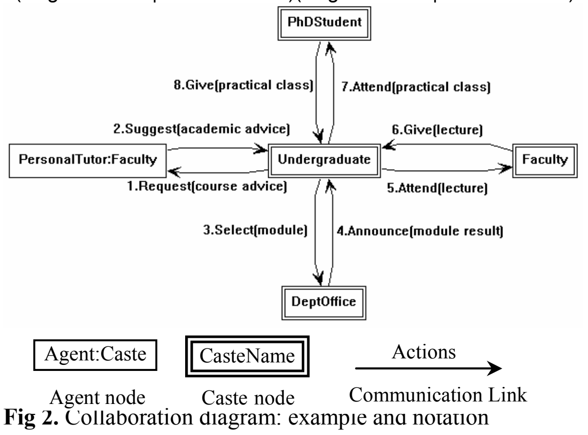

Fig 2. Collaboration diagram: example and notation

An example of intra-diagram consistency constraint on collaboration diagrams is given below.

Each caste or agent node must have a unique name.
$\forall D\in M$.<Collaboration diagram>.($\forall X, Y\in D$.(<Caste node> $\cup$ <Agent node>). (X./Name/= Y./Name/ $\Rightarrow X=Y$))

A caste is a compound caste if it is composed of a number of other castes; otherwise, it is atomic. Each compound caste has a collaboration model and a behaviour model, while each atomic caste only has a behaviour model. Thus, a collaboration model may contain a hierarchy of sub-models on various abstraction levels. When an agent in a system is decomposed into a set of components, a collaboration model is constructed for the compound agent to specify the interactions between its components.

Collaboration model on each abstraction level may contain a general diagram and a set of specific diagrams. A general diagram serves as a declaration of what castes and their instance agents are involved in collaborations, while the specific diagrams define the details of the collaboration protocols in various scenarios. Constraints on the collaboration diagrams at the same abstraction level are horizontal constraints. The following is such an example.

Every agent node in general diagram G must appear in at least one specific diagram in the same collaboration model.
$\forall n\in G$.<Agent node>.($\exists D\in \boldsymbol{S}$.($n\in D$. <Agent node>)
*where **S** is the set of specific diagrams of the same level, CName(n) denotes CasteName part of node n, $\exists n\in \boldsymbol{S}$.<X> is an abbreviation of $\exists D\in \boldsymbol{S}.\exists n\in D$.<X>.*

The following is an example of vertical constraints. It is imposed on the models at different levels.

The set of agents and castes in C's environment described in M must be equal to the set of agents and castes in M_C's environment description.
$n\in M_C$.<Environment boundary>
$\Leftrightarrow\exists\alpha\in G$.<Interaction>.($n$=$Begin(\alpha)\wedge C$ =$End(\alpha)$)
where G is the general diagram in M.

3.3 Behaviour model

Behaviour model of a system consists of two types of diagrams: behaviour diagrams and scenario diagrams. A behaviour diagram contains a set of behaviour rules to specify the caste's behaviour in certain scenarios. There are six different kinds of arrows that connect different kinds of nodes in behaviour diagrams. A scenario diagram describes a typical situation in the operation of the system. Scenario diagrams are referred to in behaviour diagrams. Fig 3 shows an example of behaviour diagram.

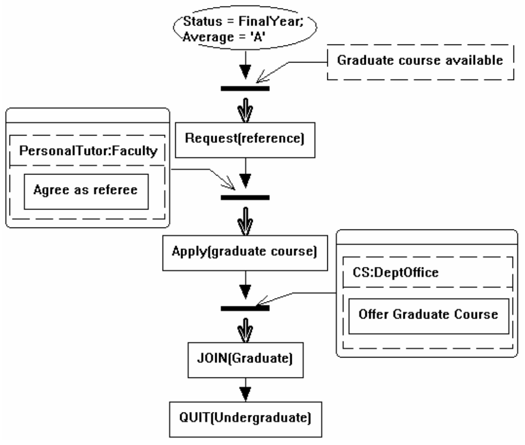

Fig 3. Example of behaviour diagram

The following is the abstract syntax of behaviour models in GEBNF, where details of various types of the links are omitted for the sake of space.

<Behaviour model>::= <Behaviour diagram>+,<Scenario diagram>*
<Behaviour diagram>::=
    /Title/: **Text** ! (<Agent Name> | <Caste Name>),
    <Logic connector>*, <Scenario node>*, <Activity node>+,
    <Action link>+, <Condition node>*,<Transition node>+,
    <Logic link>*, <Temporal relation>*
<Activity node> ::= [</Time stamp/: **Text** ! <Time stamp>],
    [/Repetition/: **Text** ! <Repetition expression>],
    (<Action node> | <State node>)
<Action node> ::= /Action/: **Text** ! <Act>
<State node> ::= /State/: **Text** ! <Predicate>
<Scenario node>::= (/Scenario name/: **Text** ! <Scenario name>)
    | <Scenario diagram>
<Scenario diagram>::= [/Scenario name/: **Text** ! <Scenario name>],
    <Logic connector>*, <Scenario node>*, < Activity node>+,
    <Logic link>*, <Temporal relation>*, <Swim lane>*
<Swim lane> ::=
    /Actor/: **Text** ! <Actor specification>, < Activity node>*

The following is a constraint on behaviour models.

*Scenarios referred to in behaviour diagrams by scenario reference nodes must be defined in scenario diagrams. Let **N** be the set of scenario nodes in the behaviour model, **S** be the set of scenario diagrams.*
$\forall n\in \mathbf{N}.\ (n./Scenario\ name/ = s_c \Rightarrow$
$\exists n'\in \mathbf{S}.<Scenario\ diagram>\ (n'./Scenario\ name/ = s_c))$

3.4 Inter-model Consistency

There are totally 9 inter-model constraints defined for CAMLE and implemented in the automatic checking tool. This subsection gives some examples of constrains between different types of models.

(A) Between collaboration and caste models

Let ***CM*** and ***AM*** be the collaboration and caste model of a system, respectively.

Castes specified in the collaboration model must be defined in the caste model. Formally,
$\forall n\in \boldsymbol{CM}.(<Agent\ node> \cup <Caste\ node>).$
$\exists n'\in \boldsymbol{AM}.<Caste\ node>.(CName(n) = n'./Name/)$

Let x be a caste in the system, M_x be the collaboration model for x. For models M_A and M_B in ***CM***, we say that M_B is an *immediate refinement* of model M_A and write $M_B \lhd M_A$, if B is the component caste of A. The following is an example of global constraints.

*The hierarchical structure of the **CM** must be consistent with the whole-part relations between castes defined in caste diagram. Formally,*
$\forall M_A, M_B\in \boldsymbol{CM}.$
$(M_B \lhd M_A \Leftrightarrow <B, A> \in \boldsymbol{AM}.<Aggregation\ relation>)$

(B) Between behaviour and caste model

A behaviour diagram defines the behaviour of a caste and the caste must be in the caste model.
$\forall D\in M.<Behaviour\ diagram>.\exists n\in M.<Caste\ node>.$
$(D./Title/ = n./Name/).$

In a behaviour diagram D_B for caste B, the description of scenarios may refer to the agents in the environment of B. Let $ReferredAgents(D_B)$ be the set of agents referred to in scenarios in D_B.

Every referred agent in a behaviour diagram must have its caste defined in the caste model. Formally,
$\forall D\in M.<Behaviour\ diagram>.\forall a\in ReferredAgents(D).$
$\exists n\in \boldsymbol{AM}.<Caste\ node>.(CName(a)= n./Name/).$

(C) Between collaboration and behaviour model

An action of a caste C described in a scenario Sc is called a referred action of C in Sc. We write $ReferredActions(C, Sc)$ to denote the set of referred actions of caste C in scenario Sc.

Every referred action in a scenario used in a behaviour diagram must be a visible action of the caste.
$(Type(Sc) = <Scenario\ node>) \Rightarrow$
$\forall\ n \in ReferredActions(C, Sc).(n \in VisibleActions(C)).$

Table 2 summarises the total numbers of consistency and completeness constraints on CAMLE.

Table 2. Summary of CAMLE's Constraints

		Horizontal Consistency	Vertical Consistency	
			Local	Global
Intra-model	Intra-diagram	17	–	–
	Inter-diagram	9	4	–
Inter-model		5	4	1

4. Effectiveness of Consistency Check

The well-formedness, consistency and completeness constraints have been implemented as automated checking tools as an integral part of the CAMLE modelling environment [2]. Once invoked, the tool checks the model and reports the diagnostic information about the inconsistency or incompleteness, if any. The violation of a constraint is reported as an error and a warning. There are totally 21 types of errors and 15 types of warning messages.

A number of case studies have been conducted to model systems including Amalthaea [12, 13], online auction [14], United Nations' Security Council, etc. In our experiences, the automatically checking consistency and completeness was helpful in detecting errors during model construction.

We have also conducted a systematic evaluation of the effectiveness of CAMLE's consistency checker using data mutation analysis to measure the checker's error detecting ability. Data mutation analysis as a software testing method was introduced in [15]. It was designed for testing software systems that have input data of highly complicated structures, such as diagrammatic models. The process of applying data mutation analysis method to our modelling tool consists of the following steps. The first step develops a number of models that passes the consistency checking. These models are called the *seeds*. The second step derives a set of *mutants* from each seed by systematically applying a set of data mutation operators. Each mutant is obtained by one application of one mutation operator on the seed so that it is slightly different from the original model. The mutation operators are design in such a way that it will in most cases make a consistent model inconsistent. Therefore, in most cases, a mutant contains one artificially inserted defect. In our experiment, totally 24 types of mutation operators are designed for CAMLE models. The Amalthaea, the online auction and the United Nations' Security Council models are taken as the seeds, from which totally 7152 mutants are generated. In the third step, the consistency checker is executed to check the consistency of the mutants. According to the output of the checker, the mutants are classified into *dead* or *alive*. A mutant is *dead* if the checker's output on it is different from that on the seed. Otherwise, it is *alive*. In other words, a mutant is dead if and only if the checker detected that it is inconsistent. The effectiveness of the consistency checker can therefore be measured by the percentage of mutants that are killed. A mutation analysis tool was implemented to automatically generate mutants as test cases, check the consistency of all mutants, and calculate statistics. Table 3 shows the results of our experiment. The dead mutant scores in the three suites are around 85%.

Table 3. Results of the Effectiveness Study

Seed	#Mutant	#Dead	#Alive	%Dead
Amalthaea	3065	2692	373	87.83%
Auction	3095	2579	516	83.33%
UNSC	992	821	171	82.76%
Total	7152	6092	1060	85.18%

In general, a mutant may remain alive for two possible reasons. First, it is still consistent and complete even though it is different from the seed. Second, the checker is incapable to detect the inconsistency or incompleteness. In the experiments, the outputs of the checker on each mutant are manually analysed to see if the results are correct. Therefore the effectiveness measurement not only tests the implementation of the consistency checker, but also evaluates the design of the consistency rules.

5. Conclusion

In this paper, we present the definition of the abstract syntax and type system of CAMLE modelling language in the graphically extended BNF. The consistency and completeness constraints for CAMLE are formally defined in the first order logic language derived from the abstract syntax and type system. These constraints are implemented in an automatic checking tool. An evaluation of the consistency constraints in detecting errors demonstrated that the approach is valid and of high error detecting ability.

In recent years, the consistency of multiple-viewed models has been an active research topic. The existing works fall into two approaches: the transformation approach, and the meta-logic/meta-programming approach. The first approach translates diagrammatic models into a formal notation such as B [16], Promela and LTL [17], CSP [18], first-order logic [19], etc. and then applies model checking or automated proof tools; see [20] for a survey. Such methods take advantages of existing formal techniques and tools, and are capable of reasoning deeply about the semantics of the models. The second approach applies formalisms at meta-model level by explicitly defining consistency constraints on the modelling languages. The consistency constraints are expressed with some formalism such as conceptual graphs [21], attributed EBNF [22], OCL [23], description logic [24], graph-grammars [25], etc. Tools have been developed to enable checking models' consistency against the constraints as CASE tools such as OCL Environment [23] and Xlinkit [26, 27], or as plug-ins of existing modelling environments, such as MCC as a plug-in of Poseidon for UML [24] and as a plug-in of Fujaba tool suit [25]. In comparison with the transformation approach, the meta-level approach is more practically applicable. The detected inconsistencies can be directly located in the original models, thus provide more instructive information on how to revise the model. The success of this approach replies on a set of well defined and explicitly specified consistency constraints, which is still an open problem.

Our work belongs to the second approach. It is based on our previous work on modelling tools for structured methods [28]. The main contributions of the work reported in this paper are three-fold. First, a framework of first order language that is capable of specifying constraints on multiple view modelling languages is proposed based on a theory of the structure and type systems of modern modelling languages and the taxonomy of constraints. Second, it demonstrates the practical applicability of the framework by specifying an adequate set of the constraints on a non-trivial modelling language. Third, it provides empirical evidence that the approach is effective and efficient to detect errors in models.

In comparison with existing works, especially those based on OCL, our language is more expressive. It is capable of specifying constraints across the boundaries between diagrams. Existing OCL-based works e.g. [23] are mainly checking on the well-formedness constraints specified in UML 2.0 documentation, which represent restrictions on the uses of individual elements [1], thus belong to intra-model constraints. They do not address inter-diagram consistency of models. A complete set of consistency and completeness constraints in OCL for UML has not been reported in the literature as far as we know. OCL was originally designed for writing constraints on instances of a system as a part of a model. Meta-level constraints written in OCL tend to be complex and lengthy as shown in [23]. It is also questionable if OCL is capable of specifying inter-diagram and inter-model constraints. The existing tools such as Dresden OCL Toolkit, Kent OCL library, OSLO [29] facilitate the use of OCL rather than the consistency and completeness problems. It is yet to be proved that they are capable of handling complicated constraints.

In [30], an empirical study was reported on uses of real industrial examples to investigate the occurrence frequency of various types of inconsistency in modelling. In this paper, we reported the case study on the effectiveness of consistency checking through artificially produced inconsistent models. These works have different purposes, but the methods are complementary in the research on the consistency problem.

We are currently further investigating how to formally specify UML and define its consistency and completeness constraints.

Acknowledgement

The work reported in this paper is partly funded by The Ministry of Science and Technology of China in the High-Technology R&D Programme (863 Programme) under the grants 2002AA116070 and 2005AA113130. The authors would like to thank Dr. Ian Bayley of Oxford Brookes University for his comments on the earlier versions of the paper and valuable discussions on the related topics.

References

[1] OMG, *Unified Modelling Language: Superstructure*, Version 2.0, formal/05-07-04.

[2] Zhu H and Shan L. Caste-Centric Modelling of Multi-Agent Systems: The CAMLE Modelling Language and Automated Tools. In *Model-driven Software Development*, Beydeda S and Gruhn V (eds), Springer, 2005, pp57-89.

[3] Finklestein A, Gabbay D, Hunter A, Kramer J, Nuseibeh B. Inconsistency handling in multi-perspective specifications, In *IEEE TSE*, Vol. 20, No. 8, 1994, pp569-578.

[4] Hunter A, Nuseibeh B. Managing inconsistent specifications: reasoning, analysis and action, In *ACM TOSEM*, Vol. 7, No. 4, October 1998, pp335-367.

[5] OMG, *OCL 2.0 Specification*, Version 2.0 ptc/2005-06-06

[6] Kuzniarz L, Reggio G, Sourrouille J L & Huzar Z (eds). *Workshop on Consistency Problems in UML-based Software Development* at UML'02.

[7] Kuzniarz L, Huzar Z, Reggio G, Sourrouille J L, Staron M (eds). *Workshop on Consistency Problems in UML-based Software Development II* at UML'03.

[8] Huzar Z, Kuzniarz L, Reggio G, Sourrouille J L (eds). *Third Workshop on Consistency Problems in UML-based Software Development* at UML'04.

[9] Paige R F, Ostroff J S, and Brooke P J. Check the Consistency of Collaboration and Class Diagrams using PVS. In *Proc. of 4th Workshop on Rigorous Object-Oriented Methods*, London, British Computer Society, 2002.

[10] Astesiano E & Reggio G. An Attempt at Analysing the Consistency Problems in the UML from a Classical Algebraic Viewpoint. *Recent Trends in Algebraic Development Techniques, Selected Papers of the 15th Int. Workshop WADT'02*, LNCS, Springer Verlag, 2003.

[11] Zhu H and Shan L. Well-Formedness, Consistency and Completeness of Graphic Models. In *Proceedings of the 9th International Modelling and Simulation* (UKSim'06). pp47-54.

[12] Zhu H. Formal Specification of Evolutionary Software Agents. *Proc. ICFEM'2002*, Springer LNCS 2495, pp249-261.

[13] Shan L and Zhu H. Modelling and specification of scenarios and agent behaviour. In *Proc. of IEEE/WIC conference on Intelligent Agent Technology* (IAT'03), IEEE CS, pp32-38.

[14] Zhu H and Shan L. Agent-Oriented Modelling and Specification of Web Services. In *Proc. of* WORDS'05, pp152-159.

[15] Shan L and Zhu H. Testing Software Modelling Tools Using Data Mutation. Accepted by AST'06 at ICSE 2006.

[16] Marcano R and Levy N. Using B formal specifications for analysis and verification of UML/OCL models. In [6]

[17] Inverardi P, Muccini H, Pelliccione P. Automated check of architectural models consistency using SPIN. In *Proc. of ASE'01,* pp346-356

[18] Küster JM, Stehr J. Towards Explicit Behavioral Consistency Concepts in the UML. In *Proceedings of the 2nd International Workshop on Scenarios and State Machines: Models, Algorithms, and Tools*, 2003

[19] Kaneiwa K and Satoh K. Consistency Checking Algorithms for Restricted UML Class Diagrams. In *Proceedings of Foundations of Information and Knowledge Systems: 4th International Symposium, FoIKS 2006* (FoIKS2006), pp219–239. LNCS 3861, Springer

[20] Sourroulle J L, Caplat G., A Pragmatic View about Consistency Checking of UML Models. In [7]

[21] Sunetnanta T T and Finkelsteing A. Automated Consistency Checking for Multiperspective Software Specifications. In *Workshop on Advanced Separation of Concerns at ICSE 2001.*

[22] Xia Y and Glinz M. Rigorous EBNF-based Definition for a Graphic Modeling Language. In *Proceedings of APSEC 2003*, IEEE Computer Society Press.

[23] Chiorean D, Pasca M, Cârcu A, Botiza C, Moldovan S. Ensuring UML Models Consistency Using the OCL Environment. In *Electr. Notes Theor. Comput. Sci. 102*: 99-110 (2004)

[24] Simmonds J M. Bastarrica C. A Tool for Automatic UML Model Consistency Checking. In *Proc. of ASE'05.*

[25] Wagner R, Giese H and Nickel U A. A Plug-In for Flexible and Incremental Consistency Management. In [7]

[26] Gryce C, Finkelstein A, and Nentwich C. Lightweight Checking for UML Based Software Development. In [6]

[27] Nentwich C, Emmerich W, & Finkelstein A. Flexible Consistency Check, In *ACM TOSEM* 12(1), pp28-63, 2003.

[28] Xu, J. and Zhu, H., Requirements analysis and specification as a problem of software automation -- Some researches on requirements analysis, in *Proc. SEKE'96*, pp457~464.

[29] Accessible at http://www-st.inf.tu-dresden.de/ocl/

[30] Lange1 C, Chaudron M R V, Muskens J, Somers L J, Dortmans H M. An Empirical Investigation in Quantifying Inconsistency and Incompleteness of UML Designs. In [6].