WELL-FORMEDNESS, CONSISTENCY AND COMPLETENESS OF GRAPHIC MODELS

HONG ZHU

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

LIJUN SHAN

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

ABSTRACT

This paper clarifies the notions of well-formedness, consistency and completeness of graphic models. A model or a diagram is well-formed if its structure satisfies the syntactic rules, especially its elements are type compatible if a type system is defined. When models are considered as specifications of software systems, consistency means that there exists a system that satisfies the specification. Completeness means that all the systems that satisfy the specification are what users want. The paper proposes a framework for the formal definition of the abstract syntax and type systems of modelling languages. A formal notation is advanced by extending and adapting BNF. A first order language for the definition of consistency and completeness constraints are also presented and illustrated by examples.

KEYWORDS: Modelling languages, Well-formedness, Consistency constraints, Completeness constraints, Type systems, Formal notations, First order language.

INTRODUCTION

Modelling languages are playing an increasing important role in software development. Typical modelling languages include UML for object oriented software development (OMG 2004), Yourdon (1989) notation and SSADM (Hares 1990) for structured analysis and design, CAMLE modelling language (Zhu and Shan 2005) for the emerging agent-oriented software development. Well-defined visual notations for modelling software systems' structures and behaviours balance well between readability and preciseness due to their semi-formal nature. They have been used for a wide range of development and maintenance activities, such as requirements specification and analysis (Yourdon 1989), formal specifications (Jin and Zhu 1997), system design (Fowler 2004), software validation and verification (Zhu, Jin and Diaper 1999, 2002), program code generation (Quatrani 2003), test case generation (Dalal *et al*. 1999, Paradkar 2005), etc. Model driven development is emerging as a promising methodology to improve both the quality and productivity of software development.

As a means of separation of concerns, the multiple-view approach has been advanced and widely adopted in the design of 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 in the analysis, specification and design of information systems. However, as Finklestein *et al*. (1994) and Hunter and Nuseibeh (1998) pointed out, maintaining consistency between views and completeness of the model is a crucial but difficult issue. It is highly desirable to automatically check model's consistency and completeness (Xu, Jin and Zhu 1996, Kuzniarz 2002); yet, graphic models must be well-formed to be processed and transformed. Unfortunately, these tasks are by no means trivial (Nentwich 2001). Most existing modelling languages, including UML, have no explicitly defined consistency and completeness constraints.

This paper proposes a framework for the formal definition of the abstract syntax and type systems of modelling languages so that well-formedness, consistency and completeness constraints on graphical models can be formally defined for automatic checking.

RELATED WORKS

The past few years has seen a rapid increase in the research on defining consistency conditions and implementing consistency check tools for modelling languages, especially for UML (Pap *et al*. 2001, Andre *et al*. 2000, Paige *et al*. 2002, Astesiano *et al*. 2003). Among the related works on consistency check, Nentwich *et al*.'s (2003) Xlinkit is a flexible tool for checking the consistency of distributed heterogeneous documents. It comprises a language for expressing constraints between such documents, a document management mechanism and an engine that checks the documents against the constraints. In comparison with Xlinkit, our approach is at a higher level of abstraction and implementation independent. Formal methods, such as model checking, have also been used for checking the consistency between multiple views of software specifications, e.g. in (Inverardi *et al*. 2001, Schafer *et al*. 2001). It requires translating models into a formal notation as the input to a model checker, while assumes

that syntactic errors have been removed before the translation. Therefore, checking well-formedness, consistency and completeness before the translation is still necessary.

How to define the syntax and semantics of graphic modelling languages is still an open problem. The specification of UML uses meta-models, which in turn is defined by meta-meta-model, etc. Its documentation spills over 700 pages, yet there is no systematic treatment of model's consistency and completeness.

In our previous work, a set of consistency and completeness constraints was defined for modelling language RDL, which contains data flow diagrams, entity relationship diagrams and state transition diagrams. Automated consistency and completeness checking was implemented as a part of the automated requirements analysis support systems RASS (Xu *et al*. 1995, 1996). In the design and implementation of caste-centric agent-oriented modelling language and environment CAMLE, a set of well-formedness, consistency and completeness constraints was formally defined and automated checkers were implemented (Shan and Zhu 2004). Our experiments demonstrated the approach is effective to improve the quality of software graphic models. Based on our previous work, this paper proposes a general framework for defining well-formedness, consistency and completeness constraints for modelling languages.

TYPE SYSTEMS AND WELL-FORMEDNESS

This section presents a framework for the formal definition of the abstract syntax and type systems of graphic modelling languages so that well-formedness of graphic models can be automatically checked.

The Structure of Typed Modelling Languages

In a multiple-views modelling language, a model often consists of several types of diagrams. Each type of diagrams represents a particular view to the system, which is often also called a model or sub-model of the system in the literature.

Definition 1 (Multiple-view modelling languages)

A diagrammatic modelling language $\mathcal{ML}$ defines a finite set $\boldsymbol{T}\neq\emptyset$ of *types of diagrams*. For each type $T\in\boldsymbol{T}$ of diagrams, $\mathcal{ML}$ defines 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 $T\in\boldsymbol{T}$, denoted by $Type(D)$. The subset of diagrams of type T in a model M is called the *T-view of the model M* (or simply the *T-sub-model* or even the *T-model*), and denoted by $M.T$. Formally,

$$M.T=\{D\in M \mid Type(D)=T\}.$$

A modelling language $\mathcal{ML}$ is called a *multiple-view modelling language*, if $||\boldsymbol{T}||>1$; otherwise, it is called a *single-view language*. □

This paper is concerned with multiple-view modelling languages. However, the theory developed in the paper equally applies to single-view languages.

For example, UML provides notations for a number of different types of diagrams, including use case diagrams, activity diagrams, sequence diagrams, communication diagrams, component and package diagrams, state machine diagrams, class diagrams, etc. The set of activity diagrams of a system is called the activity model of the system. Similarly, in structured analysis and design methods (Yourdon 1989), there are entity relationship diagrams, dataflow diagrams, control flow diagrams, and state transition diagrams.

Modern modelling languages are also often typed in the sense that nodes and relations between the nodes are classified into various types and visually represented using different graphic notations. This can significantly improve the readability of the diagrams. Moreover, this enables modelling tools to check if a model is well-formed so that a certain set of obvious errors in modelling can be detected and prevented. A diagram is well-formed only if the diagram satisfies the type compatibility constraints. For example, in UML use case diagrams, two types of nodes can be drawn in different visual notations: the use case nodes in the form of oval circles and the actor nodes in the form of stick figures. Arrowed dash lines can only be drawn between use case nodes. A solid line without arrows can be drawn between an actor and a use case. Otherwise, the use case diagram is not well-formed. These lines in different styles represent different types of relations between the nodes.

Definition 2 (Graphically typed modelling language)

A modelling language $\mathcal{ML}$ is *graphically typed*, if

(a) For each type $T\in\boldsymbol{T}$ of diagrams in $\mathcal{ML}$, the language $\mathcal{ML}$ defines a finite set $N_T\neq\emptyset$ of types of nodes, and a finite set $E_T\neq\emptyset$ of types of relations among the nodes.

(b) 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.

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 the nodes or relations that each relation e connects satisfy the type requirements of e's type te. □

Most relations in modelling languages are binary, hence represented as lines in various graphic styles, such as dashed lines, solid lines, double lines, thin lines, or thick lines, etc. They can also be *directed*, *bi-directed* or *undirected* with various styles of arrows. Hence, such relations in a model are also often called edges, or arcs or arrows. Relations are usually associated with nodes, but sometimes associated with other relations in the diagram. For example, in UML class diagrams, a relation can be defined between a node and an association, which is also a relation. Relations can also be specified on more than two nodes. For example, a swim-lane in an activity

diagram of UML specifies that a set of activity nodes are the actions taken by one actor. Thus, it is in fact a relation represented in the form of a set though not stated as such in UML manual. This is also an example that a relation may be visually represented other than as an edge or line. Relations between model elements in a diagram may be implicitly specified through the positions of the elements drawn in the diagram. Another example is that, in UML sequence diagrams, a message arrow is drawn above another to indicate that former is sent before the latter.

Elements in a diagram are often annotated with text and numeric values of various syntax formats. For examples, nodes in various types of diagrams are almost always associated with text string as its name. Relations may also be named. For example, an arrow in data flow diagram can be associated with a name of a data entity to represent the data flow from one process to another. Type systems in modelling languages can also be defined on the data types annotated to various types of nodes and relations. A diagram is not only needed to be graphically well-formed, but also well-formed with respect to the annotations in the sense that the values associated to each node and each relation are type compatible with their required date types and formats.

Definition 3 (Annotationally typed languages)

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

(a) For each diagram type T, the language $\mathcal{ML}$ defines a fixed finite number of fields $f_{T,i}$, i=1,…n_T, for the annotations that can be associated to a diagram of type T, and for each field $f_{T,i}$ a given data type $FT_{T,i}$ of the values that can be assigned to the field $f_{T,i}$.

(b) For each node or relation type t in 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, and for each field $f_{t,i}$, a given data type $d_{t,i}$, i=1, …n_t, of the values that can be assigned to the field $f_{t,i}$.

A diagram D of type T in $\mathcal{ML}$ is *annotationally well-formed*, iff the values assigned to the annotation fields of the diagrams, the nodes and the relations in the diagrams are all compatible to the data types defined by $\mathcal{ML}$. □

Annotationally typed modelling languages further restrict the freedom in the annotations associated to the elements of the models to prevent errors and facilitate automated reinforcement of the quality of models. Unfortunately, not all modelling tools have taken the advantages of such type systems.

Definition 4 (Typed modelling languages)

A modelling language $\mathcal{ML}$ is *typed*, iff it is both graphically and annotationally typed. A model M in the typed $\mathcal{ML}$ is *well-formed* (or *well-typed*), if all diagrams of M are both graphically and annotationally well-formed. □

Another vehicle to deal with the complexity in modelling is the hierarchical decompositions of systems so that diagrams at different levels of abstraction represent a complicated system with different granularities and contain different amount of details. Consequently, a view to a system may have a set of diagrams of the same type to describe the system at different levels of details.

A typical example of language facilities that support hierarchical levels of abstraction is in the dataflow and control flow diagrams in structured analysis and design methods. A data flow model of a system may contain a number of data flow diagrams at different levels of abstraction. At the top level, a system is modelled by a dataflow diagram called context diagram that only contains one process node and a number of data flows that represent the information flow from external entities into the system and the output produced by the system. A process in a dataflow diagram can be refined into a lower level dataflow diagram to specify how the process is statically structured and how it dynamically works.

It is worthy noting that two diagrams of the same type are not necessarily ordered by the refinement relation. They may be at the same level of abstraction, but represent different aspects of the system. For example, two activity diagrams may represent the interactions between a system and its environment in two different scenarios.

Definition 5 (Hierarchical modelling languages)

A modelling language $\mathcal{ML}$ is a *hierarchical modelling language on its type T*, if the following conditions hold.

(a) The T-submodel $M.T$ of a model M in $\mathcal{ML}$ can have more than one diagram.

(b) The user can define a binary relation $\prec_T$ on the subset $M.T$ of diagrams so that $D_1 \prec_T D_2$ means that diagram D_2 is a *refinement* of diagram D_1. In the sequel, we will also say that D_1 is *at the higher level* than D_2, or D_2 is *at the lower level* than D_1.

(c) The user can define a binary relation $\succ\!\prec_T$ on the subset $M.T$ of diagrams so that $D_1 \succ\!\prec_T D_2$ means that diagram D_1 and D_2 are *at the same level of abstraction*.

A model is *well-formed with respect to the refinement relation*, iff the following conditions hold.

(i) The relation $\prec_T$ has asymmetry, transitivity and irreflexity;

(ii) The relation $\succ\!\prec_T$ is an equivalence relation, i.e., it has reflexity, transitivity, and symmetry;

(iii) For all D_1 and D_2 in M_T, $D_1 \succ\!\prec_T D_2$ implies that both $D_1 \prec_T D_2$ and $D_2 \prec_T D_1$ are not true;

where, a binary relation < on a domain D is *transitive*, if

$$\forall x,y,z \in D.(x<y \,\&\, y<z \Rightarrow x<z).$$

Relation < is *asymmetric*, if

$$\forall x, y \in D.(x<y \,\&\, y<x \Rightarrow x=y).$$

Relation < is *irreflexive*, if $\forall x \in D.(\neg(x<x))$.

Relation < is *reflexive*, if $\forall x \in D.(x<x)$.

Relation < is *symmetric*, if $\forall x, y \in D.(x<y \Rightarrow y<x)$.

□

The refinement relation is often specified by modellers through annotations on diagrams.

Example 1. (Refinement relation between SSADM dataflow diagrams)

SSADM's numbering rules for specifying the refinement relations between dataflow diagrams follow.

In the level 1 data flow diagram, the processes are numbered as 1, 2, …, *K*. If a process in the level *n* data flow diagram is numbered as *x*, the data flow diagram that refines process *x* must also be numbered as *x*, and the processes in the diagram *x*, which is at level *n*+1, must be numbered as *x*.1, *x*.2, …. We can prove that a model that follows this scheme of numbering is well-formed with respect to the refinement relation. □

Notation for Defining Syntax and Type Systems

To define the type system and abstract syntax of a modelling language, we propose the following notation given in Table 1, which is called GEBNF (Graphically Extended BNF).

Example 2. (Use case diagrams)

For example, the following GEBFN formulas define the structure and types for use case diagrams of UML.

<*Use Case View*>::= <*Use Case Diagram*>+
<*Use Case Diagram*>::=
    <*Actor*>*, <*Use Case*>+, <*Association*>*,
    <*Generalisation*>*, <*Extend*>*, <*Include*>*,
    <*Scope*>

The above GEBNF formula defines that a use case diagram consists of at least one use case node and some actor nodes, actor-use case association and some include extend and generalisation relations. The following GEBNF formulas define the annotations and types of the nodes and relations in use case diagrams.

<*Actor*> ::= /*Actor Name*/: <*Name*>, <*Attribute*>*
<*Use Case*> ::= /*Use Case Name*/: <*Name*>
<*Association*> ::= <*Actor*> <*Use Case*>
<*Generalization*> ::= <*Use Case*> <*Use Case*>
<*Extend*> ::= <*Use Case*> <*Use Case*>
<*Include*> ::= <*Use Case*> <*Use Case*>
<*Scope*> ::= <*Use case node*>+.

According to the above definition, a use case diagram that contains a line between two actor nodes is not well-formed, because there is no such relation type. □

Table 1: GEBNF Notation

Notation	Meaning	Example and explanation
<X>	X is a concept or a type of entities in the model	<*Model*> and <*Diagram*> represent the concepts of models and diagrams, respectively.
X ::= Y	X is defined as Y	<*Model*> *::=* <*Diagram*>* : a model is defined as 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, the union of 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*>]: element of 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 as 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'	Terminal element, the literal value of a string	'*extends*': the literal value of the string 'extends'.
Text [!*F*]	Predefined type Text with syntax specified by F, where F is a 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.

CONSISTENCY AND COMPLETENESS

The type systems of modelling languages discussed above and the well-formedness conditions based on the type systems can prevent and detect a large number of errors in modelling. However, they alone are insufficient to detect more complicated errors such as those across the boundary of a diagram, even the boundary of a type of sub-model. Therefore, consistency and completeness constraints are defined on models to facilitate the detection and prevention of such errors in modelling.

Consistency 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 conflicts in the information specified by the model can be prevented and detected.

There are several taxonomies of consistency constraints that can be defined on modelling languages, which are discussed as follows.

Intra-diagram vs. *Inter-diagram constraints*
A consistency constraint C is called *intra-diagram*, if it is defined on a specific type T of diagram of the model in the form of

$$C(M) \Leftrightarrow \forall D \in M.T.\ C'(D),$$

where C' is a predicate defined on D.

A consistency constraint C is *inter-diagram*, if it is defined on two or more diagrams. For example, a consistency constraint C that is defined on two diagrams of type T is an inter-diagram consistency constraint, where

$$C(M) \Leftrightarrow \forall D, D' \in M.T.C'(D, D'),$$

and C' is a predicate defined on D and D'.

Inter-model vs. *Intra-model constraints*
A consistency constraint C is called *inter-model*, if it is defined on diagrams of more than one type, say between diagrams of types T_1 and T_2, so that

$$C(M) \Leftrightarrow \forall D \in M.T_1, D' \in M.T_2.C'(D, D'),$$

where C' is a predicate defined on D and D'.

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

Vertical vs. Horizontal constraints
A consistency constraint C is a *horizontal constraint* if it is defined between diagrams of a type T at the same abstraction level. Formally,

$$C(M) \Leftrightarrow \forall D_x, D_y \in M.T[D_x \succ\!\prec_T D_y \Rightarrow C'(D_x, D_y)],$$

where C' is a predicate on two diagrams of type T.

A *vertical* consistency constraint C is defined between diagrams that have refinement relationships between them, that is, in the form of

$$C(M) \Leftrightarrow \forall D_x, D_y \in M.T[D_x \prec_T D_y \Rightarrow C'(D_x, D_y)],$$

where C' is a predicate on two diagrams of type T.

Local vs Global constraints
A consistency constraint C is called *global* on a particular type of diagrams, if it is defined on the whole set of diagrams of the type. Otherwise, it is called *local constraint*. For example, a global consistency constraint C can be defined in the following form.

$$C(M) \Leftrightarrow \forall D_x \in M.\ T_1[C'(D_x, M.T_2)],$$

where C' is a predicate defined on a diagram of type T_1 and a set of diagrams of type T_2.

Completeness Constraints

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.

Both consistency constraints and completeness constraints can be specified in the form of predicates. It is hard to distinguish one from the other in their syntactic structures. However, the consequence of the violation of a consistency constraint differs from that of a completeness constraint.

A violation of a consistency constraint implies that there is an error in the model due to confliction between different parts of the model. The error must be modified in order to obtain a sound model. Otherwise, the model will not make sense. If the model serves as a specification of a system to be implemented, there will be no system that satisfies it. A consistency constraint, therefore, is a correctness criterion. Therefore, a violation of a consistency constraint means that the model is incorrect. In an automated modelling tool that checks the consistency of the models, an error must be reported once a violation of a consistency constraint is detected.

In contrast, a violation of a completeness constraint implies that a certain piece of information is missing. Thus, more information should be added into the system. Otherwise, the model leaves a space for ambiguity and different interpretations. If a model serves as a specification of a system to be implemented, incompleteness does not mean that there is no system that satisfies the specification. Instead, there may be a too wide range of choices so that a system that satisfies the specification may still have unexpected properties and behaviours on certain aspects seriously. In such cases, incompleteness may result in an implementation of a wrong system. Therefore, there is no guarantee that a system developed according to the model will always lead to a right system. It is practically impossible to work out which is the implementation that the users actually want due to the lack of crucial pieces of information.

However, a model's incompleteness can often be intentional, for example, when the model is constructed incrementally so that information is gradually added into the system through a series of stages. Thus, incompleteness should not be treated as incorrectness, while the identification of what is missing in the model may be very useful as a guide to the modeller in searching for required information. In an automated modelling tool that checks the completeness of the models, a violation of a completeness constraint should, therefore, be reported as warnings, rather than errors.

Formal Notation for Defining Constraints

The notation for defining type systems for graphical modelling languages proposed above is not sufficient to define the consistency and completeness constraints. The following proposes a first order language for formal definition of such constraints based on the type system.

Let φ be an n-ary operator defined on the type $t_1, t_2, \ldots, t_n$, that results in a value of type t. Let ρ be an n-ary relation defined on the type $t_1, t_2, \ldots, t_n$.

- *Expressions* are formed by finite applications of the following constructions.
  - *Variables* of various types are expressions of their own types.
  - *Constants* are expressions of their own types.
  - $\varphi(e_1, e_2, \ldots, e_n)$ is an expression of type t, if $e_1, e_2, \ldots, e_n$ are expressions of types $t_1, t_2, \ldots, t_n$, respectively.
  - $e.f$ is an expression, if e is an expression of type t and f is a field defined by the language $\mathcal{ML}$ for the type t. The type of $e.f$ is ft, if the type for field f is defined to be of type ft by $\mathcal{ML}$.
  - $e.t$ is an expression, whose value is the set of the elements of type t in e, where type t is defined in $\mathcal{ML}$.
  - *Type(e)* is an expression if e is an expression. The value of *Type(e)* is the type of e.
- *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 of types $t_1, t_2, \ldots, t_n$, respectively; in particular, $e_1 = e_2$ and $e_1 \in e_2$ are statements, if e_1 and e_2 are expressions.
  - *Type*$(e) = t$ is a statement, if e is an expression and t is a type name.
  - $\neg\rho$, $\rho_1 \Rightarrow \rho_2$, $\rho_1 \Leftrightarrow \rho_2$, $\rho_1 \wedge \rho_2$, and $\rho_1 \vee \rho_2$ are statements, if ρ, ρ_1 and ρ_2 are statements.
  - $\forall X \in E.S$ and $\exists X \in E.S$ are statements, if X is a free variable in statement S.

Consistency and completeness constraints can be formally specified as statements of the first order language defined above.

Example 3. (Example of consistency constraint)

A consistency constraint for use case diagram is that if a use case node A extends use case node B, use case node A drawn within the scope box implies that node B is also within the scope box. This can be specified as follows.

$$\forall D \in M.\langle \textit{Use case diagram} \rangle [\forall X \in D.\langle \textit{Use case node} \rangle (X \in D.\langle \textit{Scope} \rangle \Rightarrow \forall Y \in D.\langle \textit{Use case node} \rangle ([X,Y] \in D.\langle \textit{Extend relation} \rangle \Rightarrow Y \in D.\langle \textit{Scope} \rangle))].$$

□

Example 4. (Example of completeness constraint)

In use case driven requirements engineering, a typical usage of UML is to define the functions of an information system by a use case diagram. For each use case, an activity diagram defines the interactions between the users and the system. The following formally specifies this completeness constraint.

$$\forall D \in M.\langle \textit{Use case diagram} \rangle [\forall X \in D.\langle \textit{Use case node} \rangle [\exists AD \in M.\langle \textit{Activity diagram} \rangle (X./\textit{Use case name}/= AD./\textit{Title}/)]].$$

□

CONCLUSION

In this paper, we clarified the notions of well-formedness, consistency and completeness in the context of graphic modelling languages. The BNF notation for the definition of syntax of textual programming languages was adapted and extended for the definition of abstract syntax and type systems of graphic modelling languages. In comparison with other notations, such as meta-model, it is simple, precise and widely applicable. We believe that definitions in the GEBNF notation can be easily translated into data structure for implementation of modelling tools for automatic checking well-formedness, and to translate into machine understandable notations such as XML. Based on the abstract syntax and type definitions in GEBNF, a first order language for the specification of consistency and completeness constraints was presented and illustrated by examples. Such specifications can be easily translated into automatic consistency and completeness checkers according to our previous experiences in the design and implementation of modelling tools.

We are applying the proposed framework and notations to the definition of nontrivial modelling languages and modelling environment. It is worth further investigating how to formally specify the whole UML and define its consistency and completeness constraints.

REFERENCES

Andre, P., Romanczuk, A., Royer, J-C. 2000. "Check the Consistency of UML Class Diagrams Using Larch Prover". *Proc. of 3rd Rigorous Object-Oriented Methods Workshop*, Clark T., (ed.), BCS.

Astesiano, E. & Reggio, G. 2003. "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.

Dalal, S. R., et al. 1999. "Model-based testing in practice", *Proc. of ICSE* '99, 285-294.

Finklestein A, et al. 1994. "Inconsistency handling in multi-perspective specifications", *IEEE TSE* 20(8), 569–578.

Fowler, M. 2004. "*UML Distilled: A Brief Guide to the Standard Object Modeling Language"*, Addison Wesley.

Hares, J.S. 1990. "*SSADM for the Advanced Practitioner*". John Wiley and Sons.

Hunter A, and Nuseibeh B. 1998. “Managing inconsistent specifications: reasoning, analysis and action”, *ACM TOSEM* 7(4), 335-367.
Inverardi, P., Muccini, H., Pelliccione, P. 2001. “Automated check of architectural models consistency using SPIN”. *Proc. of ASE’01,* San Diego, California, 346.
Jin, L. and Zhu, H. 1997. “Automatic generation of formal specification from requirements definition”, *Proc. of ICFEM’97*, Hiroshima, Japan, 243-251.
Kuzniarz, L. et al. (eds.) 2002. “*Consistency Problems in UML-based Software Development*” Proc. of UML’02, Research Report. Blekinge Institute of Technology.
Nentwich, C., Emmerich, W. & Finkelstein, A. 2001. “Static Consistency Check for Distributed Specifications”. *Proc. of ASE’01,* Coronado Island, CA, 115-124.
Nentwich, C., Emmerich, W., & Finkelstein, A. 2003. “Flexible Consistency Check”. *ACM TOSEM* 12(1), 28-63.
OMG, 2004. “*Unified Modeling Language: Superstructure”.* Version 2.0, formal/05-07-04.
Paige, R. F., Ostroff, J. S., and Brooke, P. J. 2002. “Check the Consistency of Collaboration and Class Diagrams using PVS”. *Proc. of 4th Workshop on Rigorous Object-Oriented Methods*, London, British Computer Society.
Pap, Z. S. *et al.* 2001. “Completeness and Consistency Analysis of UML Statechart Specifications”. *Proc. of IEEE Design and Diagnostics of Electronic Circuits and Systems Workshop*, 83-90.
Paradkar, A. 2005. “Case studies on fault detection effectiveness of model based test generation techniques”, *Proc. of A-MOST* '05, *ACM SIGSOFT Software Engineering Notes* 30(4).
Quatrani, T. 2003. “*Visual Modelling with Rational Rose 2002 and UML”*, Addison Wesley.
Schafer, T., Knapp, A., & Merz, S. 2001. “Model Check UML State Machines and Collaborations”. *Workshop on Software Model Check,* Paris.
Shan, L. and Zhu, H. 2004. “Consistency Check in Modeling Multi-Agent Systems”. *Proc. of COMPSAC*’04, IEEE CS, Hong Kong, 114-121.
Xu, J. and Zhu, H. 1996. “Requirements analysis and specification as a problem of software automation -- Some researches on requirements analysis”. *Proc. SEKE*'96, Nevada, USA, 457-464.
Xu, J., Jin, L., & Zhu, H. 1996. “Tool support of orderly transition from informal to formal descriptions in requirements engineering”. *Proc. of IFIP'96*, 199-206.
Xu, J., Zhu, H., et al. 1995. “From requirements definition to formal functional specification -- A transformational approach”. *Science in China*, Supp. 38 (Sept.).
Yourdon E. 1989. “*Modern structured analysis”*. Prentice-Hall, Englewood Cliffs, NJ.
Zhu, H. and Shan, L. 2005. “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, 57-89.
Zhu, H., Jin, L., and Diaper, D. 1999. “Application of Task Analysis to the Validation of Software Requirements”, *Proc. SEKE'99*, Kaiserslautern, Germany, 239-245.
Zhu, H., Jin, L., Diaper, D. 2002. “Software requirements validation via task analysis”, *Journal of System and Software* 61(2), 145-169.

AUTHOR BIOGRAPHIES

HONG ZHU is a professor of computer science at Oxford Brookes University, UK. He obtained his BSc, MSc and PhD degrees in Computer Science from Nanjing University, China, in 1982, 1984 and 1987, respectively. He worked for Nanjing University as a lecturer, associate professor and then full professor from August 1987 to November 1998. From October 1990 to December 1994, he was a research fellow at Brunel University and then the Open University, UK, while on leave from Nanjing University. He joined Department of Computing of Oxford Brookes University in November 1998 as senior lecturer in computing and became a professor of computer science in October 2004. He is a member of British Computer Society, ACM, IEEE Computer Society, China Computer Federation, and China Artificial Intelligence Association. His research interests are in the area of software engineering including software development methodology, software testing, agent technology, automated software development tools, etc. He has published widely, which include two books, five peer reviewed book chapters, twenty two refereed journal papers in English, twelve refereed journal papers in Chinese, and more than sixty papers in refereed international conference/workshop proceedings. He has won a number of prizes in China for his research achievements, which include the Premier’s Award of Distinguished Young Scientists in China awarded by the National Natural Science Foundation of China, and Professorship of Cheung Kong Scholars Programme by the Ministry of Education of China. His email address is hzhu@brookes.ac.uk. His webpage can be found at http://cms.brookes.ac.uk/staff/HongZhu.

LIJUN SHAN is a PhD candidate at the Department of Computer Science of the National University of Defence Technology, where she obtained BSc and MSc degrees in Computer Science in 2000 and 2003, respectively. Her research interest is in software development methodology, in particular, the agent-oriented methodology, service-oriented software engineering and model-driven software development. She has developed an agent-oriented modelling language CAMLE and implemented its automated modelling environment. She has published 7 papers in referred international conference proceedings and 2 peer reviewed book chapters, and has one paper accepted by an international journal. Her email address is lijunshancn@yahoo.com.