Component Composition in Business and System Modeling

Stuart Kent[*], Kevin Lano[**], Juan Bicarregui[**], Ali Hamie[*], John Howse[*1]

[*]Division of Computing,
University of Brighton, Lewes Rd., Brighton, UK.

[**]Department of Computing,
Imperial College of Science Technology and Medicine,
180 Queens Gate, London, SW7, UK.

http://www.biro.brighton.ac.uk/index.html, biro@brighton.ac.uk

Abstract. Bespoke development of large business systems can be couched in terms of the composition of components, which are, put simply, chunks of development work. Design, mapping a specification to an implementation, can also be expressed in terms of components: a refinement comprising an abstract component, a concrete component and a mapping between them. Similarly, system extension is the composition of an existing component, the legacy system, with a new component, the extension. This paper overviews work being done on a UK EPSRC funded research project formulating and formalizing techniques for describing, composing and performing integrity checks on components. Although the paper focuses on the specification and development of information systems, the techniques are equally applicable to the modeling and re-engineering of businesses, where no computer system may be involved.

1 Introduction

Two techniques are essential to building *large* models, whether they are of businesses or information systems and whether they are specifications, designs or implementations: the ability to abstract away from detail, to see the "big picture"; and the ability to split up models into chunks which can be easily comprehended in isolation. "Splitting" is done by allowing ***multiple appearances*** of a model, which means for example using many small type diagrams instead of one large one. These may be grouped according to certain criteria; for example grouping by ***subject area*** aims to keep related functionality together and reduce coupling between the chunks involved. Such techniques require multiple appearances to be composed in order to construct the complete model, combined with substantial cross-checking between the appearances to ensure integrity and safety. In practice integrity checking is hard to do without substantial tool support (which is virtually non-existent commercially), so attempts are often made to ensure that the grouping is disjoint, often leading to an artificial architecture. The techniques we envisage should not require disjoint grouping.

Abstraction necessitates techniques for mapping an abstract model into a more concrete one, and vice-versa. In practice this is generally done informally and often with little documentation, with the net result that the abstractions are unlikely to be maintained, hence never used to assist with making changes and extending models – exactly the time when they are most useful. The notion of refinement, a concept well known in Formal Methods, is an approach to formally documenting and checking these mappings. A refinement comprises an abstract model, a concrete model and a formal mapping between them. A refinement is accompanied by a series of integrity checks to ensure that the mapping is "complete" in some sense, and that the concrete model retains the behavior of the more abstract model, or, to put it another way, that the abstraction is an appropriate abstraction of the concrete model. A refinement may be viewed as a composition of the abstract and concrete models and the mappings between them.

The nature of and strength of coupling between chunks determines how maintainable and extendible the model is. In component-based development (CBD) the aim is to discover chunks that are generic and flexible enough to be used in many situations. Such components maximize extensibility and changeability of a system and, of course, support reuse.

To summarize, "chunk" equals component, with some components being more generic and flexible than others. In CBD we are interested in building, combining and reusing components, so even bespoke model construction can be regarded as a form of CBD; its just the components are usually not that flexible or generic. Refinement can also be thought of as a composition of components. Techniques for composition and checking integrity are essential to all these activities, and these are not generally supported by CASE tools. The techniques must support overlapping (i.e. non-disjoint) components, and must be usable in practice. In our view, the latter means that they must support the construction of a CASE tool that does not require the developer to be versed in sophisticated mathematical notation or reasoning techniques.

1. This research was partially funded by the UK EPSRC under grant numbers GR/K67304 and GR/K67311.

This paper overviews work being done as part of a UK government funded research project investigating the formal underpinnings of Object Technology (BIRO), as it applies to the problem of component composition and integrity checking. We describe:

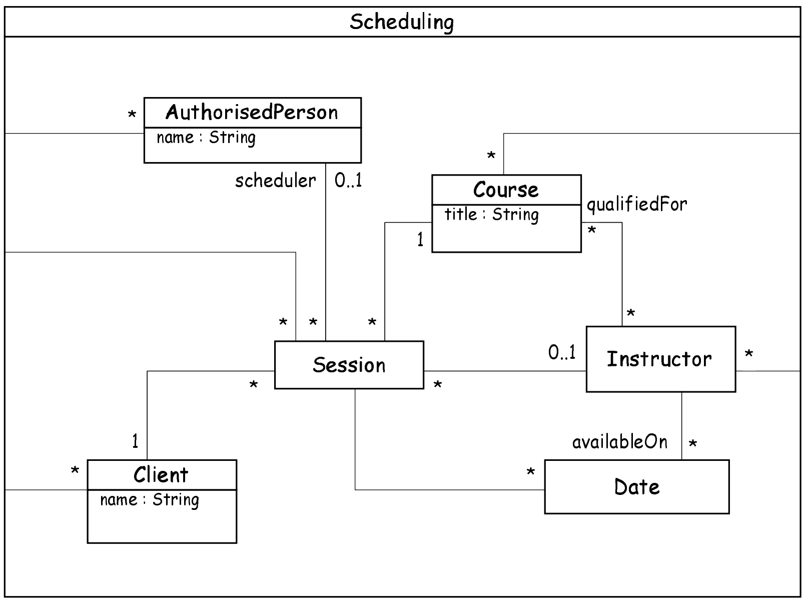
- how components may be precisely described, essential if any meaningful integrity checks are to be carried out;
- a selection of component compositions that must be supported, and the integrity checks that go with them;
- the approaches to semantics being adopted to explore the completeness and integrity of the techniques, themselves, and their possible automation.

The examples used to illustrate are extracted from the arena of information system specification and subsequent development. However it should be stressed that the same modeling techniques can be used to model and develop a business, whether or not a computer system is involved.

2 Describing Components

In order to perform integrity checks, and especially if these checks are to be automated, it is necessary to describe components precisely and with sufficient detail. Recent efforts in object-oriented modeling have made this possible. Specifically Catalysis (d'Souza and Wills 1995, 1997) extends UML (UML 1997) with techniques for describing components precisely and in more detail than is possible with e.g. just the diagrammatic notations supported by UML. In particular, they define a mathematical language for writing down invariants and action pre/post conditions. More recently a diagrammatic notation has been defined which is (nearly) as expressive as the mathematical notation but arguably far more intuitive to use. The details can be found in Kent (1997), which also develops an example demonstrating the inexpressiveness of UML-like diagrammatic notations.

An example of the Catalysis notation in use is given in Figure 1, which shows a type diagram, an invariant and an action specification.



schedule(dates:Set(Date), cl:Client, c:Course,
    p:AuthorisedPerson)
schedule a session of a course

pre
at least there must be a course, client and dates
$c \neq nil \wedge cl \neq nil \wedge dates \neq nil$

post
there is a new session
$\exists s$:Session,
    $s \in new$
    for course c
$\wedge$ $s.course = c$
    scheduled by p *(if* p *is* nil*, then it is assumed that the session is being scheduled directly by the client)*
$\wedge$ $s.scheduler = p$
    for client cl
$\wedge$ $s.client = cl$
    on dates dates
$\wedge$ $s.dates = dates$

invariant
the instructor assigned to teach a course must be qualified for that course, available on the required dates and not assigned to teach another course on those dates

$\forall s$:Session, let i be s.instructor in
if s *has an instructor then*
$i \neq nil \Rightarrow$
    the instructor i *is available*
    $s.dates \subseteq i.availableOn$
    and not assigned to teach another session on the required dates
    $\wedge\ s.dates \cap i.sessions[\neg cancelled].dates = nil$
    and the instructor i *is qualified to teach the course*
    $\wedge\ s.course \in i.qualifiedFor$

Figure 1: Course Administration System - Scheduling

3 Composition

3.1 Multiple Appearances

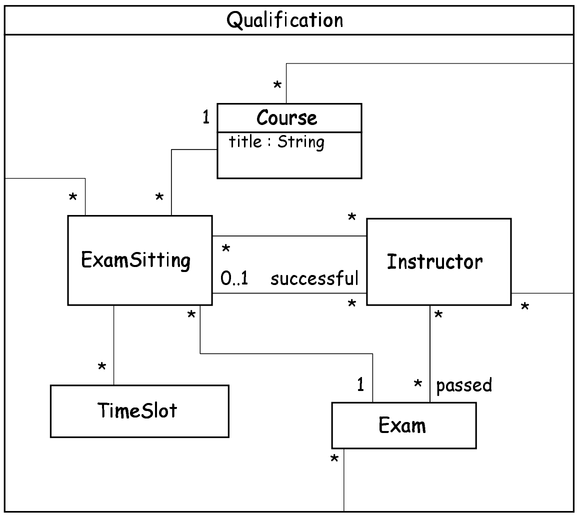


Figure 2: CAS - Instructor Qualification

In order to comprehend the model for a large system it is advisable to provide multiple appearances of the model, that is spread its description over many of diagrams. Often these diagrams are grouped into subject areas. For example, for a course administration system there will not only be diagrams relating to course scheduling, but also diagrams relating to instructor qualification. A type diagram for the latter is given in Figure 2.

How should these be composed? In this case, the subtyping mechanism can be used: the Course Administration System (CAS) is just a subtype of Scheduling and Qualification, with some additional constraints, for example an invariant to say how qualifiedFor in Scheduling is defined in terms of exam passes in Qualification. Subtyping ensures that where a type, association etc. is mentioned twice, e.g. Instructor appears in both Scheduling and Qualification, the composition must only include the type once. This is an example of overlapping components.

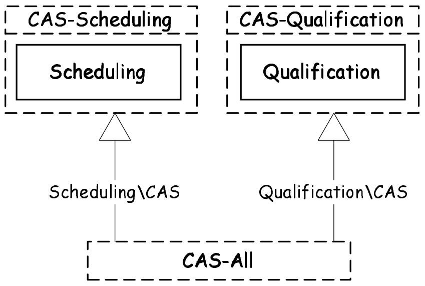


Figure 3: Framework composition

An alternative to using subtyping is to use the more general notion of framework composition supported by Catalysis, where a framework is, at its most general, just a block of development work – a collection of diagrams and textual annotations. In this case, we can imagine two frameworks, dubbed CAS-Scheduling and CAS-Qualification, each containing Figure 1 and Figure 2, respectively. Then a framework CAS-All could be constructed by composing CAS-Scheduling and CAS-Qualification as in Figure 3.

Here the subtyping arrow is being overloaded to mean framework inclusion. The result of this composition is that CAS-All contains the type CAS instead of Qualification and Scheduling. This is achieved by the renamings Scheduling\CAS and Qualification\CAS, where A\B means rename A with B. Using frameworks, composition by subtyping is actually given by Figure 4.

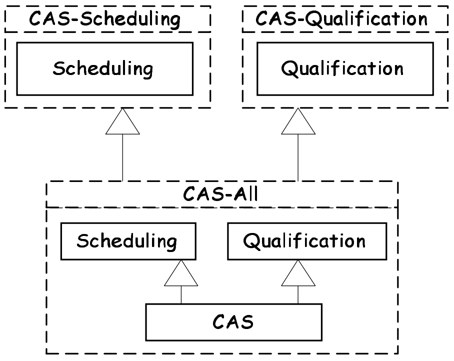


Figure 4: Using subtyping in composition

Thus, in general, multiple appearances are grouped into frameworks and frameworks may be composed to produce other frameworks. We can now make the claim that component = framework, and the composition of components equates to composition of frameworks by inclusion, with the possibility that renamings can occur. Of course, additional types, associations, subtyping relationships, invariants etc. may be added to the resulting composition and these will almost certainly be used to create links between the various parts of frameworks included. Figure 4 is an example of this.

As with subtyping, framework inclusion must be accompanied by a series of integrity checks ensuring that the resulting component remains consistent.

A similar concept to framework, called subsystem, is introduced by Bicarregui et al. (1997)

3.2 Refinement

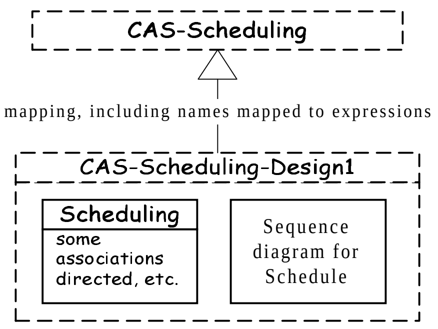

CAS-Scheduling
mapping, including names mapped to expressions
CAS-Scheduling-Design1
Scheduling
some associations directed, etc.
Sequence diagram for Schedule

Figure 5: Constructing a design

A refinement is a triple comprising an abstract model, a concrete model and a mapping between them. Refinements are often used to document the transition from a specification to a design to an implementation. For example, a design of the specification model in Figure 1, would incorporate a sequence diagram, showing how the schedule action is implemented in terms of smaller actions, together with specifications for the new actions introduced and a new type model which may begin to “direct” some of the associations in the specification model, or may even add some *new* types and associations. The framework constituting the design model would be constructed by composing the specification model, which will be subject to an appropriate mapping of language (directing of associations, renamings, etc.), with the sequence diagram and the new actions. Figure 5 shows this construction documented in Catalysis.

The refinement itself may also be thought of as a composition of the specification and design models (with appropriate renamings to distinguish the two models) and the mapping between them. In this case, the mapping would be derived automatically from the mapping used to construct the design model. A series of integrity checks would be required to ensure that the refinement was consistent, which would in turn ensure that the construction of the design model was valid. In this example, the mapping must be checked for completeness, specifically that all navigation routes in the specification model are retrievable in the design, i.e. information has not been lost (this is similar to the checks required on retrieval mappings used with refinement in VDM – Jones, 1990); and the implementation suggested by the sequence diagram would be required to guarantee the post-condition of schedule as specified in Figure 1. This requires propagation of the post-conditions of all the component actions as the sequence diagram is traversed. It can be expressed formally as a mathematical proof in an appropriate programming logic.

Refinement may be used to document relationships between abstract and concrete components. Thus it has applications other than design, for example showing that a more abstract specification (of a business or a system) is a true abstraction of a more detailed one.

3.3 Generic components

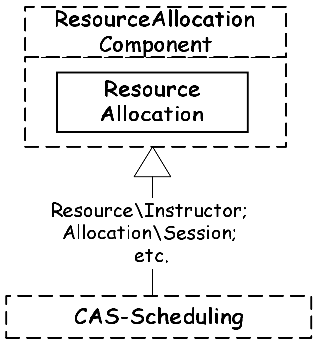


Figure 6: Specialization

Probably the best way to identify generic components is to mine existing developments. The recommended approach is to look for similarities and patterns in a range of similar developments, generalize those into a single specializable component which can then be inserted back into the original developments, hence reducing the maintenance burden, and/or reused in new developments.

For example, the scheduling part of the course administration system is a specialization of a standard component that appears in e.g. car rental, airline reservations etc., namely resource allocation. A type diagram for this general component is given in Figure 7. This would be packaged up with invariants, action specifications etc. into a framework. The framework that is Figure 1 is then a simple specialization of this component, as illustrated by Figure 6.

In general, there could be many generic components in a repository, and a model for a system – specification, design, implementation or combination of all three, could be constructed by composing specializations of these components. Again framework inclusion could be used to document such compositions.

4 Semantics

Work in the BIRO project (BIRO) is focussed on providing semantic underpinnings for rigorous OO modeling notations such as Catalysis and Syntropy, with the aim of checking the integrity of the notations themselves and of providing support to the automation of the techniques. Two approaches have been adopted, though the differences between them are more of style than substance.

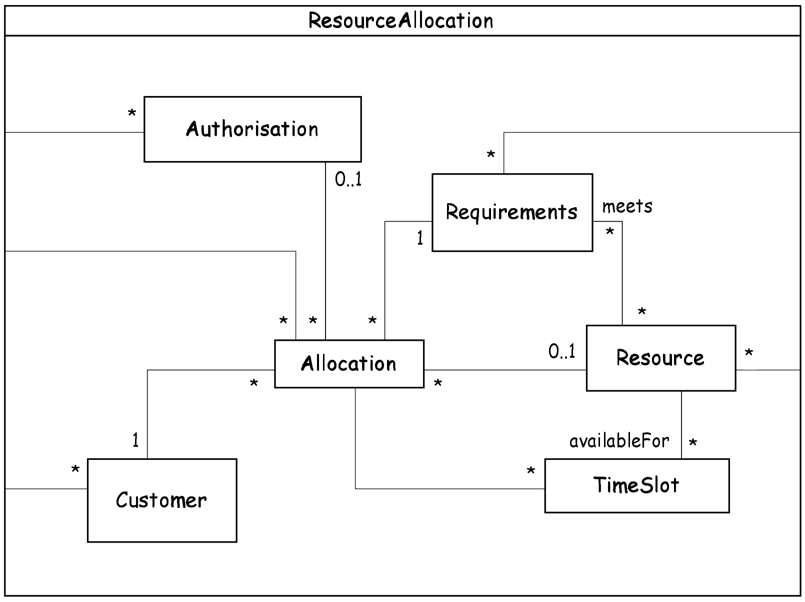


Figure 7: Resource Allocation Component

4.1 Object Calculus Approach

The first approach, adopted by the team at Imperial College, is based on the Object Calculus (Fiadeiro and Maibaum, 1992). The work has focussed on Syntropy (Cook and Daniels, 1994), which, like Catalysis extends well known object modeling notations (in this case OMT and statecharts) with mathematical annotations to precisely characterize detailed behavior. Essentially elements of Syntropy models (e.g. classes, associations, states, events etc.) are mapped into theories of a simple temporal logic. These are combined via the co-limit construction in category theory, which is essentially a form of disjoint union with some identification of symbols in different theories, not dissimilar to the renamings used with the framework inclusion mechanism of Catalysis. The level of construction is quite fine-grained so that only those parts of a Syntropy model required to e.g. specify an event are included in the theory corresponding to that specification. This allows locality (frame) conditions to be introduced at the most appropriate level. The details of this approach can be found in Bicarregui et al. (1997).

The use of Object Calculus is motivated in part by a desire to develop logics and reasoning systems most appropriate for object-oriented models. In addition the category-theoretic basis provides a powerful and general approach to composition.

4.2 Larch Approach

The second approach, adopted by the team at Brighton, favors Larch (Guttag and Horning, 1993). Larch is a mature language which comes with a toolset including a sophisticated proof assistant. Its choice was motivated by the desire not to be engaged in the design of logics and reasoning systems, but instead to focus on elaborating the meaning of the modeling notations themselves. The main differences with the Object Calculus are:

- the logical language used is a dialect of first-order predicate calculus, instead of the temporal logic employed in the Object Calculus;
- the method of composition is not expressed explicitly in terms of category theory, although it could be given a category theoretic foundation; renamings are supported on composition, though the mechanism is not as general as category theory, in that a name can not be substituted for an expression.

Larch was originally applied to developing the semantics of Syntropy (Hamie and Howse, 1997). The focus has now shifted to UML with Catalysis extensions, largely to keep in line with standardization efforts. The modeling of types, associations, actions, states, attributes, etc. is similar to that of Object Calculus, except that it is encoded in the FOPL dialect of Larch, rather than in temporal logic.

Composition in Larch is achieved by a theory inclusion mechanism which supports renaming. This corresponds very closely to the Catalysis framework inclusion: frameworks are mapped to Larch theories, and framework inclusion maps to theory inclusion. Larch does not support mappings of names to expressions, so this aspect of framework inclusion is not supported though “get arounds” are possible. Current work is using composition to take a fine-grained approach to the building up a model, by treating all notational elements (types, associations, invariants, actions specifications, etc.) as generic components which are specialized and then used to build other notational elements (e.g. types and associations are composed to build type diagrams). The details can be found in (Hamie et al., 1997). An overview is given in (Kent et al., 1997).

The Larch semantics is currently being used as the basis of a mapping into Prolog with the aim of providing automated support to the checking and animation of specifications. Although the proof assistant tool could be used for some of this, it requires too much mathematical knowledge to be used in general. However, we do plan to use it to help check the integrity of the semantics being built.

4.3 Pictorial Approach

A third approach to semantics being considered is to make use of the diagrammatic notation developed in Kent (1997). This is far more expressive than e.g. UML diagrams, and can, in fact, be used to characterize the meaning of type diagrams and state diagrams. (Kent et al. 1997) suggests a bootstrapping approach whereby this notation is given a precise semantics e.g. in Larch, and then is itself used to give a semantics to other diagrams. A problem observed with this is that the notation as it stands can not express information carried by sequence diagrams. Work under way (Kent and Gil, 1997) is extending the notation to a 3 dimensional modeling notation, which, it is argued, can carry all the information that can only currently be conveyed by a series of 2D diagrams together with textual annotations. The 3D model can therefore be viewed as a diagrammatic view of the single, conceptual model that unifies the various modeling notations; indeed, it appears that the 2D UML diagrams are simply projections of the 3D model.

5 Conclusions

The paper has argued that component composition is essential to the construction and development of large business and system models, with the pre-requisite that components must be described precisely to support the integrity checks that accompany composition. Some examples of describing components precisely and of different forms of composition have been given. Three approaches to semantics have been described, demonstrating the semantic support available for component composition.

Our focus for further work is in three areas: development of notation, semantics and providing support for CASE tools.

Development of notation. This work is continuing apace. The main motivation behind it is to provide intuitive, visual yet expressive notations for describing models precisely and in sufficient detail. The most recent result is the development of notation for expression logical constraints visually and a 3D notation that is still under development.

Semantics. The main aim of the semantics work is to ensure the integrity of the notations employed and to obtain a precise definition of the (possible alternative) meaning(s) of that notation. A secondary aim is to assist with automation in CASE tools. There are two aspects to the semantics work: semantics of notations, and elaboration of the integrity checks that need to be performed to ensure consistent models and consistent compositions.

Semantic coverage is nearly in place for the core UML/Catalysis notation: type diagrams, state diagrams, invariants, pre/post conditions and instance (object) diagrams. Sequence and collaboration diagrams are to be considered next. Work on the completeness, expressiveness and integrity of the notation described in Kent (1997) is also required. The essential semantics for composition, as described in this paper, is also in place.

The elaboration of integrity checks goes hand in hand with the investigation of different uses of composition. In essence the single result that has to be demonstrated is that the model resulting from the composition and extension/specialization of other models is consistent, and this amounts, in our semantics, to showing that the corresponding logical theory is consistent. However, this is next to useless without identifying specific, simple steps that need to be undertaken to guarantee this result. These steps are largely dependent on what composition is being used for and the kind of mappings (renamings etc.) required to make the composition work. A case in point is the work which has begun on defining the mappings used in refinement and the integrity checks that must accompany these (Lano and Bicarregui, 1997).

CASE tool support. Although the use of logics to characterize the semantics suggests that automated reasoning techniques could be used as a basis of automation, this has not been tried in practice. Recently, we have begun to map some of the Larch semantics into Prolog to explore the possibilities of checking and animating models. Another approach under consideration is to directly encode the diagrams of (Kent, 1997) and (Gil and Kent, 1997) into a tool, and use algorithms for manipulating graphs to automate some aspects of composition and integrity checking. A third idea is to maintain a repository of example situations (expressed as UML instance diagrams or scenarios described using UML sequence diagrams and Catalysis filmstrips), and use this repository to check consistency of models. For example, if two appearances of a model are inconsistent then mathematically this would mean that there would be no example situation satisfying both models, and this would be discovered when the models were checked against the repository. In fact the repository idea would provide a stronger check than mathematical consistency, as it would require the models to be consistent with at least the real world situations identified as desirable by virtue of them being placed in the repository.

6 References

BIRO project Web page. http://www.biro.brighton.ac.uk/biro/index.html.

Bicarregui J., Lano K. and Maibaum T. (1997) Objects, Associations and Subsystems: a hierarchical approach to encapsulation, in *Proceedings of ECOOP97*, LNCS 489, Springer-Verlag.

Cook S. and Daniels J. (1994) *Designing Object Systems*, Prentice Hall Object-Oriented Series.

D'Souza D. and Wills A. (1995) *Catalysis: Practical Rigour and Refinement*, technical report available at http://www.iconcomp.com.

D'Souza D. and Wills A. (1997) *Component-Based Development Using Catalysis*, book submitted for publication, manuscript available at http://www.iconcomp.com.

Fiadeiro J. and Maibaum T. (1991) Describing, Structuring and Implementing Objects, in de Bakker et al., Foundations of Object-Oriented Languages, LNCS 489, Springer-Verlag.

Gil Y. and Kent S. (1997) *Three Dimensional Notations for Software Modeling*, in preparation.

Guttag J. and Horning J. (1993) *Larch: Languages and Tools for Formal Specifications*, Springer-Verlag.

Hamie A. and Howse J. (1997) *Interpreting Syntropy in Larch*, Technical Report ITCM97/C1, University of Brighton, available at http://www.biro.brighton.ac.uk/index.html.

Hamie A., Howse J. and Kent S. (1997) *Compositional Semantics for Object-Oriented Modeling Notations*, in preparation.

Jones C. (1990) *Systematic Software Development using VDM (2nd edition)*, Prentice Hall.

Kent S. (1997) *Constraint Diagrams: Visualizing Invariants in Object-Oriented Models*, in Proceedings of OOPSLA97, ACM Press, to appear.

Kent S., Hamie A., Howse J., Civello F. and Mitchell R. (1997) Semantics Through Pictures: towards a diagrammatic semantics for object-oriented modeling notations, in Procs. of ECOOP'97 workshop on Precise Semantics for Object-Oriented Modeling Techniques, LNCS series, Springer Verlag, to appear.

Lano K. and Bicarregui J. (1997) Refinement Through Pictures: Formalizing Syntropy Refinement Concepts, in *Proceedings of BCS FACS/EROS ROOM Workshop*, BIRO technical report GR/K67311-2.00, Department of Computing, Imperial College.

UML (1997) *Unified Modeling Language v1.0*, Rational Software Corporation, available at http://www.rational.com.