Electronic Notes in Theoretical Computer Science 42 (2001)
URL: http://www.elsevier.nl/locate/entcs/volume42.html 11 pages

Meta-Modelling for Formal Software Development

A. Sreenivas, R. Venkatesh and M. Joseph

Tata Research Design and Development Centre,
Tata Consultancy Services,
Pune, India[1]

Abstract

Formal refinement shows how the specification of a program can be transformed by steps into an executable implementation. The method is sound and rigorous but is not suited to dealing with large and complex programs. Thus, the construction of large programs is often assumed to remain outside the scope of formal representation altogether and informal techniques, sometimes using diagrammatic notations, are used to go from requirements to programs.

In this paper, we show how large programs can be defined in terms of different *views* that are instances of a single meta model. Each view represents one set of properties of the program and their composition defines the specification of the program. Transformations are used to convert views into program fragments that are composed together to build the whole program. The technique has been applied for the construction of many large systems that are in practical use.

1 Introduction

In the formal development of a program, an abstract specification is transformed by a number of steps of refinement into an executable program. The method is supported by a sound logical framework and is suited for the development of relatively small sequential programs. Faced with the problem of developing large and complex programs, however, industrial practice uses a combination of non-formal notations and methods. Different methods are used to express the properties of different aspects of the program and these are combined through the steps of a development process. The functional adequacy of the final program depends very largely on the thoroughness with which the steps are carried out and questions of formal correctness are not addressed.

[1] URLs: www.pune.tcs.co.in, www.tcs.com

In this paper, we describe a method for constructing an industrial-scale software system using multiple models. Rather than using unrelated notations and methods for different aspects of the program, the models in our method are all instances of a single meta- model. This provides a unified way of describing both the models and the system that is constructed using these models. It also leads to a simple and elegant implementation method. The method has been used extensively to construct medium and large-scale programs that are now in operational use in different countries.

We first describe how multiple models can be used for software development effectively in Section 2. We illustrate this using two examples: one using different models in the construction of program analyzers (Section 3) and another using multiple models to decompose a large problem into smaller components (Section 4). Finally, we discuss some problems that remain to be solved when using this approach in Section 6.

2 The use of modeling for software development

The formal refinement [7] of a program starts with an abstract specification $\mathcal{A}$ which is to be transformed into a concrete implementation $\mathcal{C}$. This abstract specification is refined by adding operational detail at each step to arrive at a final implementation.

$$\mathcal{A} \rightarrow \cdots \mathcal{S}_i \rightarrow \mathcal{S}_{i+1} \cdots \rightarrow \mathcal{C}$$

In these steps, $\mathcal{S}_{i+1}$ is a refinement of $\mathcal{S}_i$ and $\mathcal{S}_i$ is an abstraction of $\mathcal{S}_{i+1}$. Since refinement and abstraction are transitive relations, an intermediate specification $\mathcal{S}_i$ is both a refinement of $\mathcal{A}$ and an abstraction of $\mathcal{C}$.

Each intermediate specification $\mathcal{S}_{i+1}$ can be derived from $\mathcal{S}_i$ using rules of refinement. It is also possible to formally prove that $\mathcal{C}$ is a correct implementation of $\mathcal{A}$ in relation to an explicit or implicit underlying semantic model $\mathcal{M}$. Thus formal refinement take place entirely within one model, i.e. one instance $\mathcal{A}$ of a single model $\mathcal{M}$.

In practice, formal refinement is hard to use for programs of any appreciable size (say greater than $10,000-20,000$ lines of code) and the method does not scale up to software development in the large.

The modeling approach constructs $\mathcal{A}$ using different abstract *views* $\mathcal{A}_1 \cdots \mathcal{A}_n$, each defining a set of properties: e.g. structural simplicity, efficiency of implementation and performance. A view, $\mathcal{A}_i$, is an instance of a more general structure which can be represented as a model $\mathcal{M}_i$, e.g. a state transition model for sequences of operations or an entity-relationship model for representing data. Note that $\mathcal{A}$ may not be available separately: $\mathcal{A}$ is used here to represent what we can informally call the 'composition' of the views $\mathcal{A}_1 \cdots \mathcal{A}_n$.

A view is the means by which one set of abstract properties is implemented through a corresponding set of mechanisms, i.e. transforming each $\mathcal{A}_i$

into a corresponding implementation $\mathcal{C}_i$. The program level composition of $\mathcal{C}_1 \cdots \mathcal{C}_n$ gives $\mathcal{C}$, which is intended to be an implementation of $\mathcal{A}$. Each $\mathcal{A}_i$ can be transformed into $\mathcal{C}_i$ manually or by using formal or informal steps of refinement, but this is then required to be done for each such program.

Instead, we make use of the model $\mathcal{M}_i$, of which $\mathcal{A}_i$ is an instance, and implement general transformations for the model. These transformations can be applied to all instances of $\mathcal{M}_i$. Defining transformations at the level of $\mathcal{M}_i$, rather than for each instance $\mathcal{A}_i$ of the model, makes it possible to scale-up the method and handle large programs.

Example: *Let $\mathcal{A}$ be an abstract specification of a compiler. $\mathcal{A}$ can be decomposed to represent the separate steps $\mathcal{A}_i$ in the compilation process. These steps are typically lexical analysis, syntax analysis, and code generation. Lexical analysis can be modelled by regular expressions (or a finite state transition model) and syntax analysis by context-free grammars (with its underlying model). The model for code generation is usually defined informally.*

Compilers have successfully been built for large languages using such models, showing that a multiple-model approach is suitable for scaling-up.

However, there are two obvious shortcomings of using different models. First, each step of refinement is performed informally, as the underlying model is not mathematically defined. Second, and more important, the different $\mathcal{A}_i$'s are not necessarily orthogonal to each other. In such cases, there must exist a consistency relationship $\mathcal{R}$ for all representations of any property that appears in more than one $\mathcal{A}_i$. We show how $\mathcal{R}$ is defined in Section 4.

3 Models for Building Program Analyzers

Program analysis is an area where multiple models are used to represent different sets of properties, e.g. syntactic properties and semantic properties. Each model encompasses a set of properties that are relatively complete and consistent. The models are related through the class of program objects that they describe.

Consider a software system that is implemented as a program in a language $\mathcal{L}$ with well-defined semantics. Using the semantic model of the programming language $\mathcal{L}$ as the underlying theory, a proof-system for $\mathcal{L}$ can be built and used to prove the properties of programs in $\mathcal{L}$ (Figure 1).

In practice, however, one may not need a full-fledged proof-system for programming language $\mathcal{L}$. In many situations, it may be sufficient to have only a weak proof-system which can check for some statically determinable properties of a program $\mathcal{P}$ in $\mathcal{L}$. Many theories [6,5,2,3] have been proposed for program analysis, or the process of statically determining a program's properties.

Every program $\mathcal{P}$ in $\mathcal{L}$ has two *views*, a structural or syntactic view $\mathcal{P}_S$

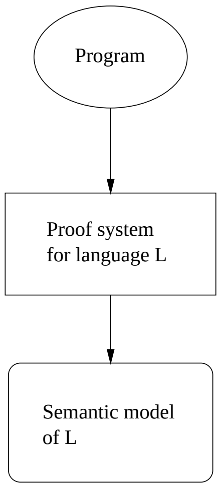

Fig. 1. A program and its proof system

and a semantic or analysis view $\mathcal{P}_A$. The language $\mathcal{L}$ itself has

- an abstract syntax model $\mathcal{L}_S$ and
- a static semantic model $\mathcal{L}_A$.

$\mathcal{P}_S$ and $\mathcal{P}_A$ are instances of $\mathcal{L}_S$ and $\mathcal{L}_A$ respectively. This is similar to $\mathcal{A}_i$ and $\mathcal{M}_i$ in Section 2.

We can generalize from language $\mathcal{L}$ to a family, $\mathcal{F}$, of languages with 'similar' semantic models (such as the set of all imperative languages). When this is done, it is possible to describe $\mathcal{L}_A$ and $\mathcal{L}_S$ in terms of more general models:

(i) A structural model of the program, $\mathcal{F}_S$, representing the abstract syntax of the languages in $\mathcal{F}$ but not representing the execution semantics of these languages.

(ii) A program analysis model, or a static semantic model $\mathcal{F}_A$ of languages in $\mathcal{F}$, which represents the theory of program analyses.

The models specific to language $\mathcal{L}$ are instances of the general models:

- $\mathcal{L}_A$ is an instance of $\mathcal{F}_A$ and
- $\mathcal{L}_S$ is an instance of $\mathcal{F}_S$.

These two models or theories are not, and indeed cannot be, fundamentally independent. The single, but complex, semantic model of $\mathcal{L}$, say $\mathcal{L}_\mathcal{M}$ has been projected into two simpler models, $\mathcal{L}_S$ and $\mathcal{L}_A$, each of which is sound and complete. Moreover, these sub-models are related to each other through the underlying semantic model, $\mathcal{L}_\mathcal{M}$ (Figure 2).

The structural or syntactic model, $\mathcal{F}_S$, describes the syntactic views of the class of programming languages represented in $\mathcal{F}$, and the relationships among them.

The theory of program analyses, $\mathcal{F}_A$, forms a model for all analyses possible

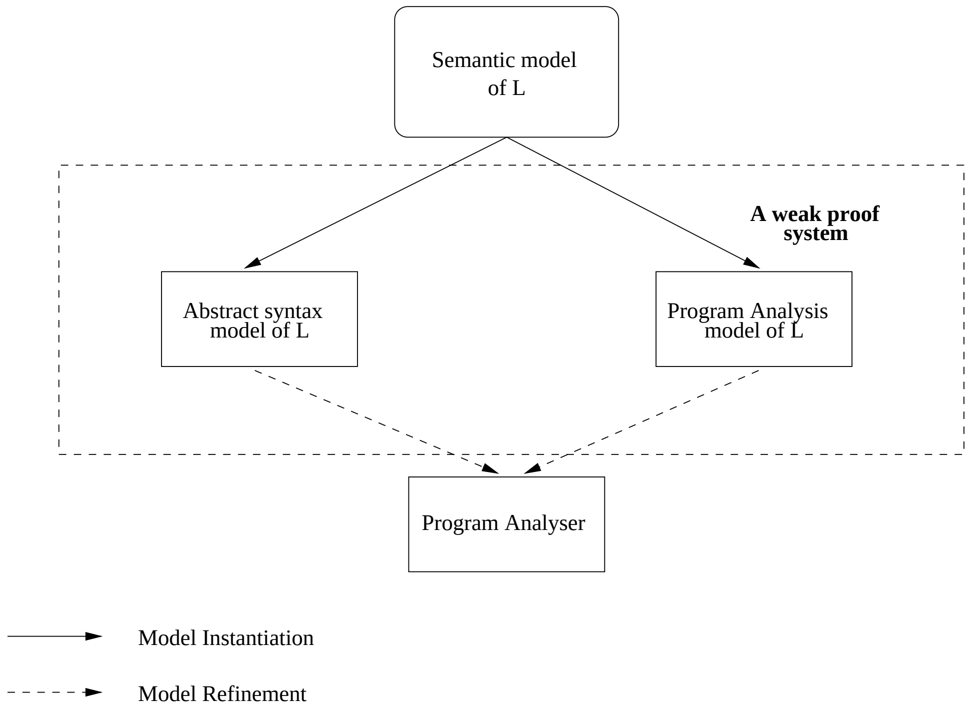

Fig. 2. Program analysis framework as a weak proof system

for the languages in $\mathcal{F}$. For a given analysis, the set of properties that hold at different nodes in $\mathcal{F}_S$ forms a lattice P with a partial ordering $\sqsubseteq$, where

$$p1 \sqsubseteq p2 \Rightarrow (p1 \textit{ holds at node } n \;\Rightarrow\; p2 \textit{ holds at } n).$$

For example, properties in P may be a set of tuples of identifiers and constants, where $\{\langle x, c_1\rangle, \langle y, c_2\rangle\}$ holds at $n \;\Rightarrow\; x$ has a value c_1 and y has a value c_2 at n.[2]

An analysis $\mathcal{L}_A$ is a function from the abstract syntax $\mathcal{L}_S$ to P, i.e.

$$\mathcal{L}_A : \mathcal{L}_S \rightarrow P$$

The semantics of $\mathcal{L}$ is typically defined as a *semantic function* over $\mathcal{L}_S$. Thus, the denotational semantic description of $\mathcal{L}$ would be a function:

$$[\![\,.\,]\!] : \mathcal{L}_S \rightarrow D$$

where D is a set of *denotations* representing the behaviour of the program.

Since an analysis is a 'weaker' version of the dynamic semantic model: it can be correct only if it is *consistent* with the semantic model. We now define consistency formally.

For each analysis, an abstraction function α can be defined from denotations D to property values P, which define how a property relates to dynamic semantic values. That is

$$\alpha : D \rightarrow P$$

[2] In such a lattice, $\{\langle x, c_1\rangle, \langle y, c_2\rangle\} \sqsubseteq \{\langle x, c_1\rangle, \langle y, \perp\rangle\}$ where $\perp$ represents *not a constant.* Similarly, $\{\langle x, c_1\rangle, \langle y, c_2\rangle\} \not\sqsubseteq \{\langle x, c_1\rangle, \langle y, c_3\rangle\}$ where $c_2 \neq c_3$.

An analysis $\mathcal{L}_A$ is said to be consistent (equivalently, *safe* [5]), iff

$$\forall x \in S.\ \mathcal{L}_A(x) \sqsubseteq \alpha(\llbracket\ x\ \rrbracket)$$

If the analysis predicts that some property p holds for some x, then it is *guaranteed* to hold at execution time. This constraint defines the relationship $\mathcal{R}$ (Section 2) which must hold for different models to be consistent with each other.

Projecting the complex semantic model $\llbracket\ .\ \rrbracket$ into the two simpler models, $\mathcal{L}_S$ and $\mathcal{L}_A$ and ensuring that $\mathcal{L}_A$ is consistent with $\llbracket\ .\ \rrbracket$ enables us to apply the analysis to a *set* of similar languages (such as all imperative languages) and the set of program properties covered by a single theory of analysis uniformly.

3.1 Darpan

Darpan [12] is a program analysis framework based on projecting a complex model into multiple models, as shown in Figure 2. In this figure, Darpan can be viewed as the dotted box, which supports analysis of programs written in a family of related languages.

Here, the syntax, $\mathcal{F}_S$ of the family of related languages corresponds to the notion of the model $\mathcal{M}_i$ of Section 2; similarly for $\mathcal{F}_A$, the analysis model. Note, that the syntax and analysis models of a *single* language $\mathcal{L}$ would correspond to the $\mathcal{A}_i$ of Section 2.

Darpan accepts a specification of the desired analysis (i.e. an *instance* of $\mathcal{F}_A$) and *generates* a program analyser corresponding to the specification. This analyser works on any instance of $\mathcal{F}_S$ to produce the analysis results. As the analyser works on an instance of $\mathcal{F}_S$, it is independent of the source language in which the program was written. This also enables the analysis of programs which comprise different components in different languages. We have used Darpan-generated analysers to analyse programs which comprised components in C, C++ and Pascal. Figure 3 represents how Darpan generates analysers from specifications.

Darpan has been successfully used to analyse real-life programs for different applications such as *reverse-engineering* [9] (i.e. extracting some desired 'meaning' from a program) and *standards-checking* [13] (i.e. checking whether a program in a given language conforms to a set of defined programming rules).

4 Relating Multiple Models

A complex system can be viewed in terms of different sets of properties. Each set of properties may require a different notation, with its own underlying model.

Consider a program $\mathcal{P}$ which is described using three specifications, $\mathcal{A}_1$, $\mathcal{A}_2$ and $\mathcal{A}_3$, with underlying models $\mathcal{M}_1, \mathcal{M}_2$ and $\mathcal{M}_3$ respectively, as discussed in Section 2. It is important to establish that $\mathcal{A}_1, \mathcal{A}_2$ and $\mathcal{A}_3$, are relatively consistent so that there is certainty that the representations do not conflict.

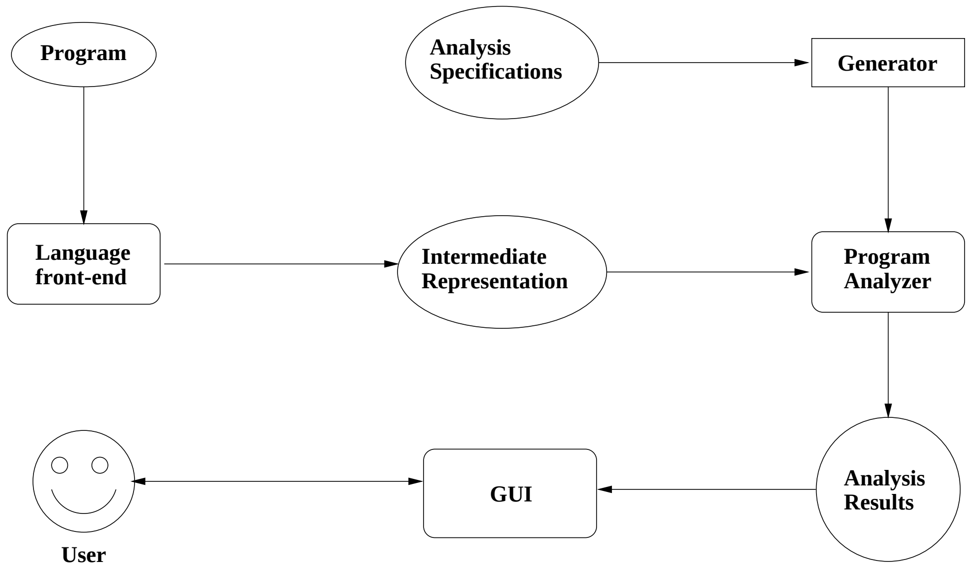

Fig. 3. Darpan – an architecture diagram

Relative consistency between different representations $\mathcal{A}_1, \mathcal{A}_2$ and $\mathcal{A}_3$, may be defined as follows:

Definition 4.1 If p is a property of $\mathcal{M}$ and p is represented in $\mathcal{M}_1$, $\mathcal{M}_2$ as p_1 and p_2 then there must exist relations R_1, R_2, R_{12}, R_{21} such that

$$p_1 = R_1(p) = R_{21}(R_2(p))$$
$$p_2 = R_2(p) = R_{12}(R_1(p))$$

Message Sequence Charts (MSC) [8] and state diagrams are two different notations used to specify the behaviour of a system. A message sequence chart M is a quintuple $\langle E, <, L, T, P\rangle$ where E is a set of events, $< \subseteq E \times E$ is an acyclic relation, P is a set of processes, $L : E \rightarrow P$ is a mapping that associates each event with a process, and $T : E \rightarrow \{s, r\}$ is a mapping that describes the type of each event (send or receive).

$<$ is a partial ordering relation among *all* events that are described by an MSC. $<_P$ is an ordering among the events of a process P. Let $E_P = \{e | e \in E \wedge L(e) = P\}$, the set of events on P. $<_P$ can then be defined as: $<_P = < \cap (E_P \times E_P)$. Note that $<_P$ defines a sequence of events over P.

Although MSCs and state-diagrams are two different views of the system they are not independent of each other. Both specify legal sequences of method invocations. Hence they have to be consistent.

Example: *Consider a simple central locking system (CLS) for a car. The system has a central control for two motors (RM, LM), one for the right door and another for the left door. The control may either lock both doors or unlock them.*

The MSC corresponding to these is shown in Figure 4. The control sends asynchronous messages to both the motors to lock. Each motor sends a confirmation after the locking operation is completed. Similarly, asynchronous

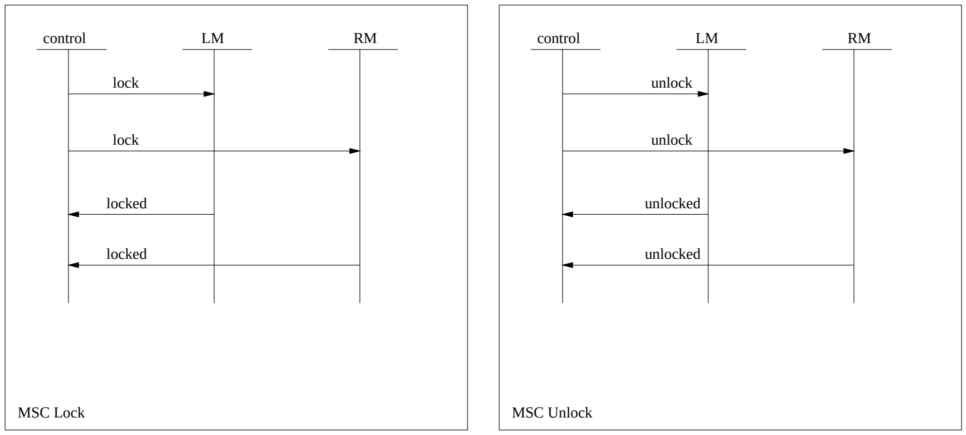

Fig. 4. A message sequence chart for central locking

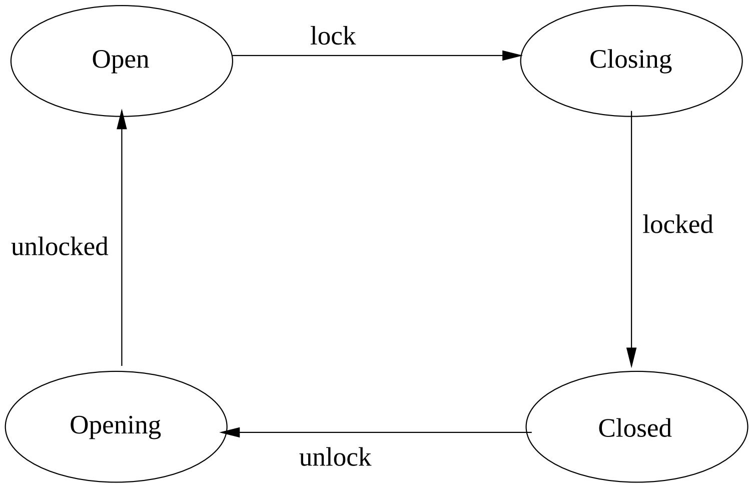

Fig. 5. State diagram for a motor for central locking

messages are sent for unlocking the doors.

Each of the motors can be modeled by a state diagram, as shown in Figure 5. This state diagram, say $\mathcal{D}$, represents all *the sequences of messages that a motor can receive. Let $\mathcal{L}_D$ be the language accepted by the state diagram $\mathcal{D}$.*

The sequence of messages sent to each motor in the MSC must conform to the sequences of events that are allowed by the state diagram. This can be stated formally as $<_P \in \mathcal{L}_P$. This ensures consistency between the MSC definition and the state diagram definition.

5 A Meta Modelling Framework

As seen from the two examples, there is a repeated need to view systems and programs at different levels of abstraction, and to relate these different views as aspects of a consistent whole. This suggests the need for a meta-model within which the different views can be related and integrated. For example,

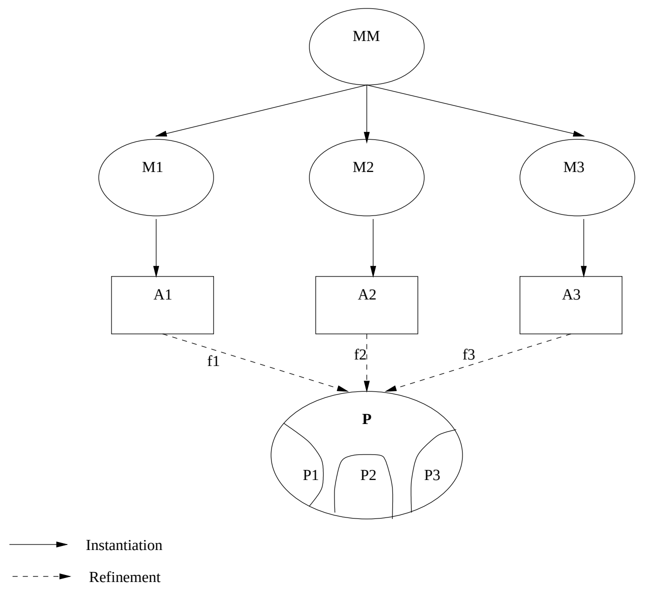

Fig. 6. Different views of a system with their models and meta-model

in Figure 6, the models $\mathcal{M}_1, \mathcal{M}_2$ and $\mathcal{M}_3$ can be related by defining them in terms of a meta-model MM.

$\mathcal{A}_1$, $\mathcal{A}_2$ and $\mathcal{A}_3$ are instances of models $\mathcal{M}_1$, $\mathcal{M}_2$ and $\mathcal{M}_3$ respectively. P is a composition of $\mathcal{P}_1$, $\mathcal{P}_2$ and $\mathcal{P}_3$, each of which is a refinement of $\mathcal{A}_1$, $\mathcal{A}_2$ and $\mathcal{A}_3$ respectively. This refinement is implemented by transformation functions f_1, f_2 and f_3.

A meta-model that enables construction of models such as $\mathcal{L}_S$ and $\mathcal{L}_A$ in the example of Section 3, and MSCs and state diagrams in the example of Section 4, consists of entities, relations and rules.

$MM = \langle E, R, r \rangle$

where

- E is a set of entity names, each of which represents a tuple of properties,
- R is a set of relations $E \times E$, between entities,
- r is a set of well-formedness rules which must be satisfied for any instance of this model to be consistent.

5.1 Adex

Adex [11] is a modelling framework that allows models and their instances to be specified. Adex supports three levels of modelling:

Meta-modelling This is similar to MM in Section 4. MM consists of meta-objects and meta-associations between the meta-objects.

Modelling This is to enable modelling of the $\mathcal{M}_i$'s (Section 2) as instances of MM. These models contain objects and associations which are instances of the corresponding meta-objects and meta-associations. Adex also has a language in which well formedness rules for these models $\mathcal{M}_i$ can be stated.

Program modelling These are instances of models $\mathcal{M}_i$ and correspond to $\mathcal{A}_i$ of Section 2. Objects and associations in this model are instances of the corresponding objects and associations of the corresponding model. Adex checks that all program models are well-formed according to the rules specified in the corresponding model.

In addition, functions of the kind described as f_1, f_2 and f_3 in Section 4 are defined to transform instances of models to corresponding fragments of a program. These functions can be implemented in a model traversal and transformation language supported by Adex. Thus, if f_1 f_2 and f_3 are correct, then $\mathcal{P}_1$, $\mathcal{P}_2$ and $\mathcal{P}_3$ will always be a correct refinement of $\mathcal{A}_1$, $\mathcal{A}_2$ and $\mathcal{A}_3$.

Adex is the modelling repository for the program development environment, MasterCraft [14], where UML is used to represent models. The meta modelling capability of Adex has enabled different modelling notations to be supported, such as entity-relation (E-R) models and even combinations betweem E-R and UML models.

6 Discussion

The modelling technique described in this paper has been successfully applied to data-modelling and transforming data-models to their corresponding program fragments. However, any large program model consists of a behavioural model and a static or data-model.

The technique therefore needs is being extended to cover behavioural specifications. A comprehensive solution to this problem includes the following:

- Defining all the different kinds of behavioural specifications in a common notation, such as UML [10],
- Rules for defining the well-formedness of these models, and
- Transformation functions generate at implementations of these models.

Work along these lines has been described also in Harel [4] and Clark et al [1]. Further work still remains to apply such techniques to industrial scale problems

7 Conclusions

In this paper, we have presented an approach to developing complex systems using multiple, related models. The models allow different sets of program properties to be represented as different views, or instances of models, that are derived from a common meta model. The technique has been used for developing large applications.

The use of multiple models brings many industrial problems within the scope of formal modelling. The method can be used by practising software engineers and integrated into a software development lifecycle. Work is continuing on developing the technique further to allow functional operations of the application to be modelled within the same framework.

References

[1] T. Clark, A. Evans, S Kent, S Brodsky, and S Cook. A feasibility study in rearchitecting uml as a family of languages using a precise oo meta-modeling approach. Technical report. Available from http://www.puml.org.

[2] P. Cousot and R. Cousot. Abstract interpretation : A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Symposium on Principles of Programming Languages*, pages 238 – 252. ACM, 1977.

[3] P. Deransart, M. Jourdan, and B. Lorho, editors. *Attribute Grammars—Definitions, Systems and Bibliography*, volume 323 of *Lecture Notes in Computer Science*. Springer Verlag, 1988.

[4] David Harel. On visual formalisms. *Communications of the ACM*, 31(5):514–530, May 1988.

[5] M.S. Hecht. *Flow Analysis of Computer Programs*. Elsevier, North-Holland, 1977.

[6] T. J. Marlowe and B. G. Ryder. Properties of dataflow frameworks: A unified model. *Acta Informatica*, 28:121 – 163, 1990.

[7] C. Morgan, editor. *On the refinement calculus*. Springer Verlag, 1992.

[8] *Message Sequence Chart (MSC)*. 1996. ITU-TS. Recommendation Z.120.

[9] Ravindra Naik. Revine: A framework for reverse-engineering. Technical report, TRDDC, Pune, 2000.

[10] *Unified Modeling Language specification v 1.3*. Object Management Group. Available from http://www.rational.org/uml.

[11] S. Reddy, J. Mulani, and A. Bahulkar. Adex – a meta modeling framework for repository-centric systems building. In *Tenth International Conference on Management of Data*, 2000.

[12] S. Shah, A. Sreenivas, and H. Pande. Techniques for generic program analysis. Technical report, TRDDC, Pune, November 2000.

[13] Assent: Automatic specification-driven standards enforcement tool. Tata Consultancy Services, 1999.
http://www.tcs.com/products/assent/htdocs/assent_index.htm.

[14] Mastercraft: Integrated development framework for distributed applications. Tata Consultancy Services, 1999.
http://www.tcs.com/products/mastercraft/htdocs/mastercraft_index.htm.