Chapter 1

Introduction

Architecture of a software system defines its high-level design structure [2]. With increased complexity and size of software products [11], architectural design modelling has become crucial. No wonder then that this area is, of late receiving increased attention from researchers as well as practitioners. Besides serving as a blueprint for implementation, a few of the important uses of an architectural design model include evaluation, understanding, and testing of architectures. An architectural design model allows an architect to reason about various system properties (e.g., concurrency, complexity, reliability, performance, etc.) at a considerably high level of abstraction.

The increased significance being attributed to models has given rise to the concept of model-driven engineering (MDE). MDE is essentially a software engineering approach with the primary focus on models, rather than on source code. The ultimate goal of MDE is to raise the level of abstraction, and to develop as well as evolve complex software systems by analyzing their models. MDE is being looked upon as the solution to the "complexity blow up" that occurs with increased code size [74]. MDE is still at a nascent stage and there is an urgent need for developing appropriate techniques and tools to support the different model-driven engineering activities.

In recent times, Unified Modelling Language (UML) is being extensively used for representing the architectural models of software systems [19, 20], and has become a standard in MDE. Originally, UML was conceived primarily for modelling object-oriented software systems. However, the elegance of UML has made the applicability of UML to grow beyond software systems. UML is now being used for modelling many other systems such as businesses, hardware, and so on. This can possibly be attributed to the support for a wide range of visual artifacts and views to model different aspects of a system.

Since the sizes of UML architectural models of systems themselves are increasing, efficient ways to analyze these models is needed. However, the task of analyzing UML architectural models poses a formidable challenge since the system information is distributed across several model views. To further complicate the task, models typically undergo changes throughout the design as well as during system evolution. A change to a model element may impact other model elements, or even other model views. In such a situation, it turns out to be tedious on one hand and equally valuable on the other to determine the impact of such model changes. In this context, several issues require investigation. It is desirable that the direct and indirect effects of a change to any design model should be automatically identified. This can help to assess the potential impact of an architectural change on the system. Such a technique can, in turn, be used to help predict the cost and complexity of a change.

Precise analysis of the impact of a change to one model element on other elements, however, is a non-trivial problem. A part of the complexity arises due to the fact that, for large architectures,

determining the impact of a change would require taking into account various types of dependencies that might exist among different model elements. For example, if an operation returning a value changes, then several classes may get affected either due to the variables being used in guard conditions associated with various interactions, or because of operation invocations. This analysis can become even more complex when one tries to consider the relations existing among objects that dynamically form and dissolve during a run of the system. On the other hand, availability of UML impact analysis techniques can be valuable for many software engineering tasks such as understanding, testing, re-engineering, maintenance of large software products.

Slicing is a promising impact analysis technique. The slicing technique was originally used for code analysis. However, due to a paradigm shift from code to the design models, slicing of models has been considered promising. Slicing at the design and analysis stage attempts to compute slices from model architectures, and is termed as model slicing. A common notion on system specification suggests two types of models namely, ADL (Architecture Description Language) and UML. This is in contrast to the use of UML as an ADL [49, 50].

ADLs have been a general topic of research in recent years when it comes to architectural modelling. The necessity for modelling through an architecture description has been recognized by Shaw and Garlan [18]. It is commonly understood that an ADL comprises three essential constituents: components, connectors and configurations [48]. Using these constituents, models of software systems need to be given in a textual representation in case of ADL architectural modelling. Such form of system modelling differs from UML architectural modelling where the software architecture is modelled through a graphical representation comprising various diagrams. In addition to a difference in representation between commonly used ADLs and UML, it is known that ADLs have difficulty in modeling dynamic aspects of software systems [47, 49]. However, this is not the case with UML architectural modelling. This also acts as an incentive for the use of UML models in architectural modelling.

There has been increasing importance on UML design models to support the evolution of large software systems. Design models such as UML architectural models are being maintained and updated from initial development as well as being reverse engineered to more accurately reflect the state of evolving systems. Viewing the entire UML model at one time is impractical and typically of little use for a particular maintenance task. To overcome these shortcomings, model slicing is being advocated by researchers to decompose large architectures into manageable portions. A straightforward benefit of such model slicing would be to automatically extract relevant parts from a very large model.

1.1 Program and Architectural Model Slicing

In this section, we present an overview of the concepts of program and architectural model slicing.

Program Slicing

Program slicing is a program analysis technique. It can be used to extract statements of a program that are relevant to a certain part of a computation. The concept of program slicing was introduced by Weiser [71–73]. A program can be sliced with respect to some slicing criterion. A slicing criterion is a pair $< p, V >$, where *p* is a program point of interest and *V* is a subset of the program's variables. If we attach integer labels to all the statements of a program, then a program point of interest could be an integer *i* representing the label associated with a statement of the program. *A slice of a program P with respect to a slicing criterion* $< p, V >$ *is the set of all the statements of the program P that*

might affect the slicing criterion for every possible input to the program. The program slicing techniques introduced by Weiser [71–73] is now called static backward slicing. It is static in the sense that the slice is independent of the specific input values to the program. It is called backward because the control flow of the program is considered in reverse while constructing the slice.

Static and Dynamic Slicing

Static slicing techniques perform analysis of a static intermediate representation of the source code to compute slices. The source code of the program is analyzed and slices are computed which hold good for all possible input values [73]. A static slice contains all the statements that may affect the value of a variable at a program point for every possible input. Therefore, we need to make conservative assumptions which often lead to relatively larger slices. That is, a static slice may contain some statements which might not be executed during an actual run of the program.

Korel and Laski [31] introduced the concept of dynamic program slicing. *Dynamic slicing* makes use of the information about a particular execution of a program. *A dynamic slice with respect to a slicing criterion $< p, V >$, for a particular execution, contains those statements that actually affect the slicing criterion in the particular execution.* Therefore, dynamic slices are usually smaller than static slices and are more useful in interactive applications such as program debugging and testing. A comprehensive survey on the existing dynamic program slicing algorithms has been reported in Korel and Rilling [32].

Consider the example program given in Figure 1.1. The static slice with respect to the slicing criterion $< 8, sum >$ is $\{1, 3, 4, 5, 8\}$. Now, consider a particular execution of the program with the input value $i = 15$. The dynamic slice with respect to the slicing criterion $< 8, sum >$ for the particular execution of the program is $\{3\}$.

```
   int i, sum, prd;
 1: read(i);
 2: prd = 1;
 3: sum = 0;
 4: while i < 10 do
 5:   sum = sum + i;
 6: end while
 7: prd = prd * i;
 8: i = i + 1;
 9: write(sum);
10: write(prd);
```

Figure 1.1: An example program

Backward and Forward Slicing

As already discussed, a backward slice contains all parts of the program that might directly or indirectly affect the slicing criterion. Thus, a static backward slice can be considered to provide the answer to the question: "*Which statements affect the slicing criterion?*". On the other hand, a *forward slice* with respect to a slicing criterion $< p, V >$ contains all parts of the program that might be affected by the variables in *V* which have been used or defined at the program point *p* [7]. A forward slice provides the answer to the question: "*Which statements will be affected by the slicing criterion?*".

Architectural Model Slicing

Architectural model slicing is a decomposition technique that identifies relevant model parts from across diverse model views that affect a specific model element. In this context, a slicing technique for software architectures should take into account various classifiers and their relations, and objects and their interactions. The UML metamodel extends support to describe structural and behavioral aspects of an architecture. For instance, a structural model such as class diagrams can be used to describe various relations among classes such as aggregation, association, composition, and generalization / specialization. On the other hand, behavioral models such as communication and sequence diagrams can depict a sequence of actions occurring in an interaction through which a use case is realized. Traditional slicing (or program slicing) is usually performed solely based on data and control dependency relationships existing among program statements. However, in architectural slicing of UML design models, several other types of model relations existing between various classifiers such as class-class, class-operation, operation-operation, class-object, object-object, etc. have to be taken into account. Therefore, to perform architecture slicing, it is necessary to first formulate an appropriate intermediate representation that can capture various types of dependencies that may exist among classes, sub-classes, operations, and attributes, and call sequences. These dependencies are termed as view dependencies.

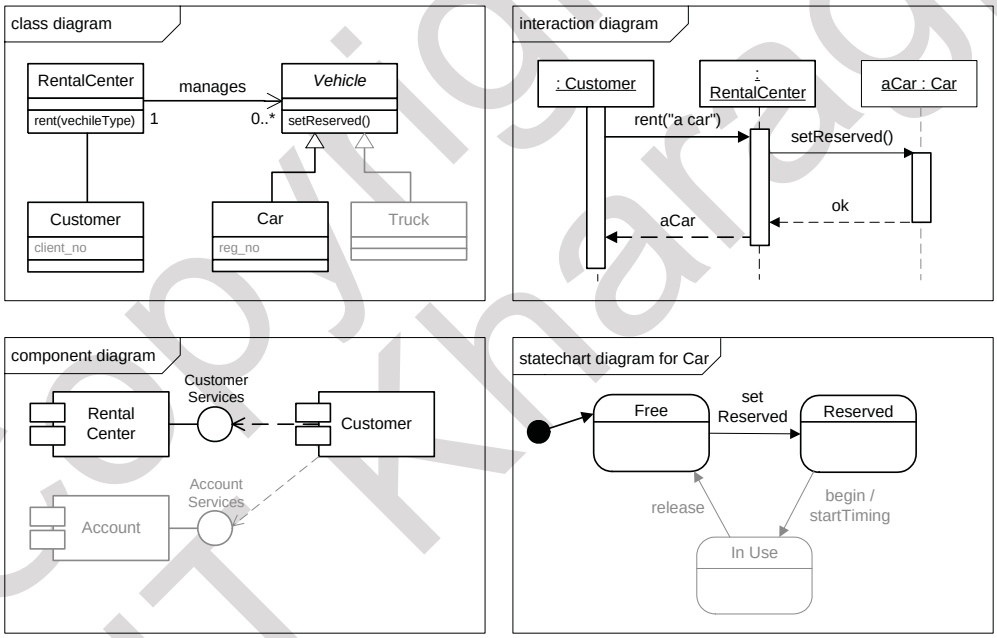

Figure 1.2: Dependencies among different model views for an example UML model

Figure 1.2 illustrates an example of a set of dependencies among different views in a UML model describing a simplified car rental system [61]. The figure shows three types of diagrams. The class diagram shows a `Customer` and a `RentalCenter` managing a set of `Vehicle` entities (i.e. a `Car` and `Truck`). The interaction diagram shows an anonymous instance of a `Customer` renting a car from `RentalCenter`. The `RentalCenter` sets a `Car` instance reserved and returns it to the `Customer`. The statechart diagram shows three states for `Car`. The dependent parts among the diagrams are shown in black while the independent parts are in gray.

Consider that a model slice for the car rental scenario with respect to an anonymous instance of

`Customer` is to be computed. This specification of a condition based on which the slice is computed is termed as a slicing criterion. Therefore, all the parts marked black do not contribute to the required model slice. This also illustrates the role of a slicing criterion in the formation of the model slice. Based on this, the model slice can be given by the set of model elements {`objects` : `Customer` and : `RentalCenter`, operations `rent`() and `setReserved`(), and state `Reserved`}.

1.2 Motivation for Our Work

Any efficient and precise technique for analysis of UML models has a scope of wide applications. For example, it can be used to construct partial model views to understand large and complex systems. Further, it can not only be used to understand software system scenarios but also to understand complete systems displaying dynamic behavior (e.g., Business Process Modeling, or Manufacturing Systems). In this context, the following critical observations motivate our present research.

- Typical software systems of today are inherently large and complex. The importance of modeling the architecture of such systems is well recognized by practitioners. UML has rapidly established itself as a standard modeling language for such applications.
- While architectural models are playing significant parts in software engineering, model analysis tools have not kept pace, and lack maturity as compared to even the tools for program analysis.
- The trend of growing sizes of software architectures is making them less amenable to manual analysis. It, therefore, has become difficult to distinguish parts of contemporary architectures for reuse, debug, impact analysis, maintenance, and critical system component identification. Therefore techniques to automatically analyze large architectures are urgently needed.
- Slicing can help to decompose and identify dependencies across architectural model views. This can help to perform cause-effect analysis when certain changes are required to an architecture.
- Most of the work reported in the literature address development of techniques based on slicing the architectural description of a system given in different architecture description languages (ADLs) such as Aesop, C2, Darwin, Meta-H, Rapide, UniCon, and Wright [3]. The research work on slicing UML architectural models has scarcely been reported in the literature, though UML is popular as an ADL [49, 50].

From the above observations, it is clear that, there is a pressing necessity to devise slicing algorithms for UML architectural models. With this motivation, we identify the major objectives of our work.

1.3 Objectives of Our Work

In this section, we present the main objectives of the work presented in this thesis. The primary goal of our work is to develop efficient and precise slicing algorithms for UML architectural models. Towards this broad goal, we identify the following objectives.

1. We plan to develop an intermediate representation for a software architecture by extracting relevant information from UML structural and behavioral models into a single system model.

2. We plan to investigate the development of a suitable framework for computing static slices for UML architectural models using the proposed intermediate representation.

3. Next, we plan to extend our framework to compute dynamic slices for UML architectural models.

4. Further, we plan to improve upon the precision of the computed dynamic slices by considering the state information of objects.

5. In addition to investigating our static and dynamic model slicing algorithms theoretically, we wish to implement them to experimentally verify their performance, correctness and preciseness.

6. Finally, we plan to carry out an analysis of our experimental studies to draw broad conclusions about the realized static and dynamic model slices, time and space requirements, as well as about scalability, and other relevant issues.

Figure 1.3 presents a conceptual schema of our work targeted towards meeting the aforementioned objectives. It shows that a key component of our work focuses on coming up with a representation of UML models that is suitable for model slicing. The conceptual schema depicted in Figure 1.3 comprises

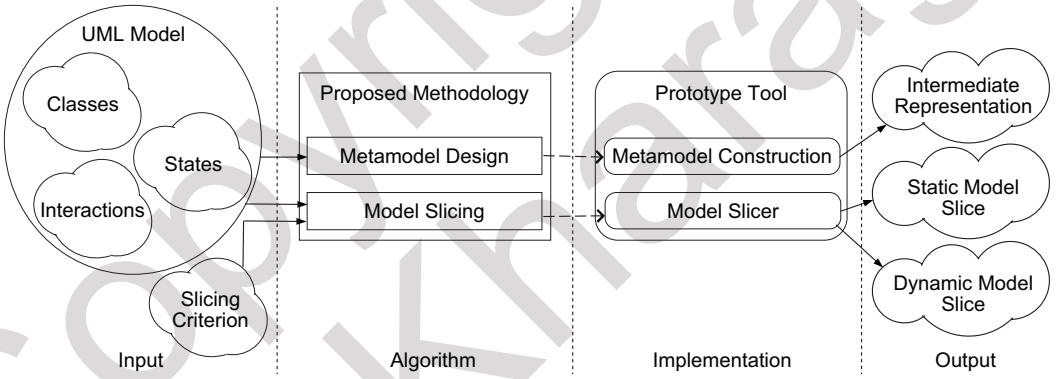

Figure 1.3: Conceptual schema of the planned work

four blocks viz., `Input, Algorithm, Implementation` and `Output` respectively. The `Input` block depicts the different inputs to be given, namely, a UML model and a slicing criterion, that can be given to the `Algorithm` block. Moreover, a UML system design is supposed to be given using the class, interaction, and state models. Next, the box titled "Proposed Methodology" represents the `Algorithm` block depicting that this thesis deals with proposing `Metamodel Design` and `Model Slicing` algorithms. Further, the `Implementation` block indicates that the experimental studies presented in this thesis are carried out through the development of different prototype tools. This is represented using a box titled "Prototype Tool" for the corresponding methodologies proposed in the `Algorithm` block. Subsequently, the `Output` block shows the expected outcome from the thesis work viz., intermediate representation, static and dynamic model slice. In the context of the above identified objectives for our work, the block marked `Algorithm` in the conceptual schema corresponds to the objectives $1-4$ while the `Implementation` block corresponds to the objectives $5-6$.

1.4 Overview of the Work Done

Our work mainly focuses on proposing appropriate slicing techniques for UML architectural models. However, the distribution of related information in diverse model elements makes slicing of UML models a complex problem. In this section, we present an overview of the work carried out to meet the thesis objectives set in the previous section.

We have proposed an algorithm *ModelGraph* to construct a metamodel by first extracting all relevant information from a UML model specifying a software architecture into an intermediate representation which we call a Model Dependency Graph (MDG). This is followed by our model slicing algorithms. The first of these proposed algorithms computes a static model slice and has been named as *Static Slicing of UML Architectural Models* (SSUAM). However, such a static model slice is large and comprises more number of model elements. To compute a model slice with fewer model elements we need to consider specific values for class attributes, fragment parameters, guard conditions etc. during slice computation. Such a dynamic model slice is computed using our second model slicing algorithm. Our proposed dynamic model slicing algorithm has been named as *Dynamic Slicing of UML Architectural Models* (DSUAM). For a given slicing criterion, our slicing algorithms traverse the constructed MDG to identify the relevant model elements that are directly, or indirectly affected during the execution of a specified scenario. One novelty of our approaches is computing a model slice based on a single model synthesized from various structural and behavioral UML models, as against independently processing separate UML models, and determining the implicit inter-dependencies among the different model elements distributed across model views. Further, to improve the precision of the computed dynamic model slices, we have proposed the use of state model information. This necessitates to consider state models during construction of our metamodel MDG. To realize this objective, we have proposed our modified *ModelGraph* algorithm which incorporates object's state information in our metamodel and makes MDG capable in using it to compute more precise slices. Finally, we have proposed our model slicing algorithm which improves the precision of DSUAM slices and which has been named *State-based Dynamic Slicing of UML Models* (SDSUM). We have also developed four prototype tools to implement our proposed algorithms and perform experimental studies.

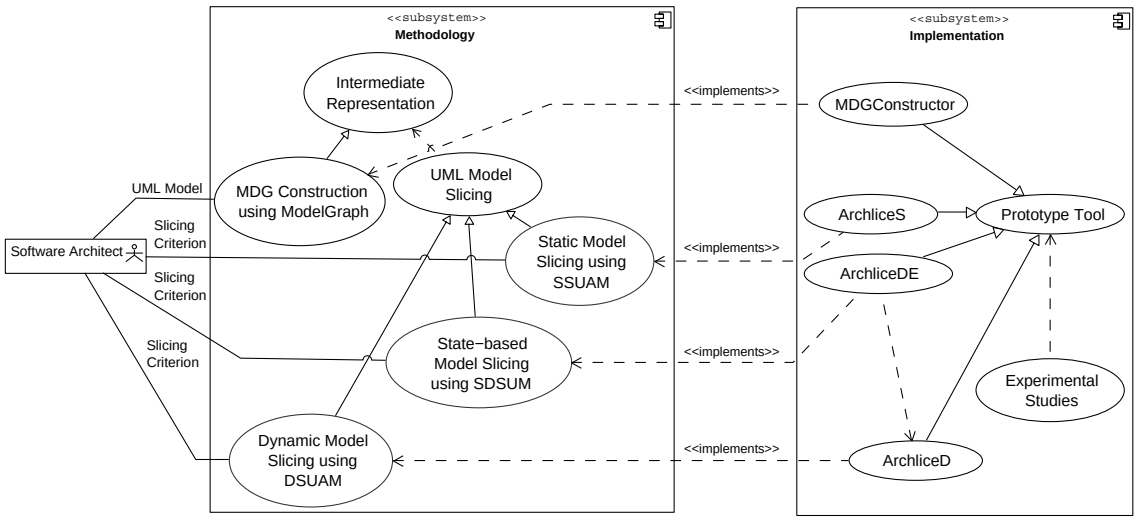

Figure 1.4: Use case model of the work done

A use case model for our accomplished work is shown in Figure 1.4. The use case model comprises two subsystems viz., *Methodology* and *Implementation*. The *Methodology* subsystem consists of proposing the metamodel for UML models, and its subsequent use in our different static and dynamic model slicing algorithms. In the use case model of Figure 1.4, the use cases have been labelled similar to the names given to each of our proposed algorithms. On the other hand, the *Implementation* subsystem deals with the prototype implementation of the algorithms proposed in the former subsystem, and its use in carrying out our experimental studies on model slicing. The implementation of our proposed algorithms involves development of four prototype tools namely, *MDGConstructor* for UML metamodel construction, and *ArchliceS, ArchliceD* and *ArchliceDE* for static, dynamic, and state-based model slicing respectively. These prototype implementations have been depicted in the model of Figure 1.4 using the use cases labeled similar to the names given for each of the developed tools in the *Implementation* subsystem. Furthermore, each of those use cases have been associated with one of our proposed algorithms in the corresponding *Methodology* subsystem using a stereotype <<implements>> .

The use case model depicts an actor which is shown to be a *Software Architect*. It also shows that an architect can take a UML model and construct its metamodel representation using our *ModelGraph* algorithm. Further, after a UML metamodel is obtained, using appropriate slicing criterion an architect can compute model slices using our static and dynamic model slicing algorithms. In addition, an architect can use the computed model slices for different applications listed in Section 1.5. However, the use of model slices in different applications has not been depicted in the use case model of Figure 1.4.

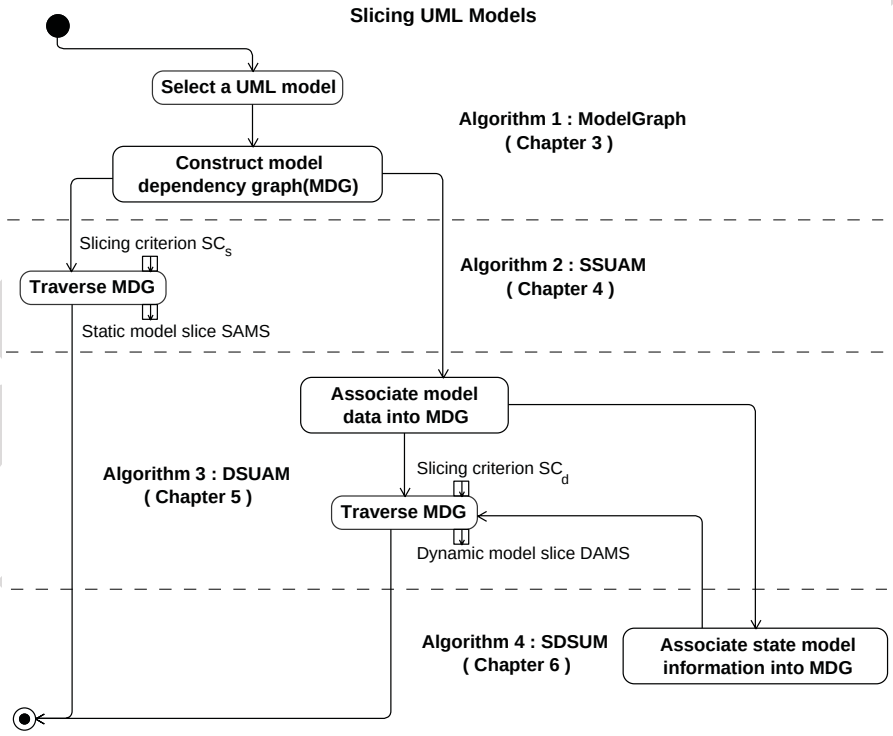

Figure 1.5: Activity diagram representation of work done

Our work concerning computation of slices from UML models has schematically been shown in Figure 1.5 using an activity diagram. The activity diagram in Figure 1.5 shows how the different parts of our work are presented in the thesis. The different parts depicted as thesis chapters have been shown

separated through a dashed line. Our proposed algorithms namely, *ModelGraph*, SSUAM, DSUAM, and SDSUM shown in different parts of the activity diagram also indicate the corresponding Chapters 3-6 in which they have been presented. This organization is elaborated further in Section 1.6. It should be noted here that, in comparison to an informal diagram, we have preferred to use UML diagrams in Figures 1.4 and 1.5 respectively to convey a simplified picture of our work carried out in this thesis.

1.5 Applications of Model Slicing

Model slicing is an extension of program slicing concepts to architectural models. From a modest beginning, program slicing techniques have now ramified into a powerful set of tools for use in such diverse applications as program understanding, program verification, debugging, software maintenance and testing, metric computation, dead code elimination, reverse engineering, parallelization of sequential programs, software portability analysis, reusable component generation, program integration, compiler optimization, etc. [8, 66]. Excellent surveys on applications of program slicing are reported in [8, 22, 32, 44, 52, 66]. The wide range of applications of program slicing can induce a plethora of similar as well as novel applications in hitherto uncharted area of model slicing. We can classify these diverse applications of model slicing into the following broad areas.

- Metrics for UML models.
- Understanding large architectures.
- Early reliability prediction
- Impact analysis of design changes.
- Regression test case selection.
- Identifying critical model elements.
- Test Case Generation and Testing Effort Estimation.
- Animation.

1.6 Organization of the Thesis

The rest of the thesis is organized into chapters as follows.

Chapter 2 discusses few basic concepts that have been used in the subsequent chapters of this thesis. We first briefly discuss the relevant UML 2.0 concepts. This is a followed by a brief review of few basic definitions and essential concepts on program and model slicing.

Chapter 3 presents our framework to synthesize UML models into our MDG metamodel. We first discuss a few concepts that are used in describing our metamodel along with its structural design. Next, we present our *ModelGraph* algorithm to construct the MDG along with its pseudocode. This is followed by an analysis of the complexity of our *ModelGraph* algorithm. Subsequently, we describe a prototype tool implementation *MDGConstructor* based on our *ModelGraph* algorithm for constructing MDG for UML models. Finally, we present the results of the experimental studies carried out using our prototype tool.

Chapter 4 presents a technique for static slicing of UML architectural models. We first present a few definitions pertinent to static model slicing. We then present our proposed technique SSUAM

for static slicing of UML models based on the metamodel MDG (discussed in Chapter 4) along with its pseudocode. The complexity analysis of SSUAM is next discussed. Subsequently, based on our SSUAM algorithm we describe a prototype tool implementation named *ArchliceS*. We also present the experimental results obtained in slicing several UML models using *ArchliceS*.

Chapter 5 presents a technique for dynamic slicing of UML models. We first discuss some issues which are necessary for understanding dynamic model slicing. We then present a few definitions related to dynamic slicing. This is followed by a description of our proposed technique DSUAM for dynamic slicing of UML architectural models along with its pseudocode. A complexity analysis for DSUAM is next presented. Finally, we present an implementation of our prototype tool *ArchliceD* based on our DSUAM algorithm for dynamic slicing of UML architectural models and the experimental results obtained using the prototype tool.

Chapter 6 reports our approach to improve the precision of model slices. We first present a few concepts for synthesizing the state information into the MDG and its outcome on precision of the computed dynamic slices. Next, we present our proposed technique SDSUM for state-based dynamic slicing of UML models along with its pseudocode. An analysis of complexity of SDSUM is then reported. This is followed by a discussion on the correctness and precision of dynamic slices computed using SDSUM in comparison to DSUAM. Finally, we describe a prototype tool implementation *ArchliceDE* that implements our SDSUM algorithm for state-based dynamic slicing of UML models. The implementation of *ArchliceDE*, in fact extends *ArchliceD* (discussed in Chapter 5) to consider state information in the model slicing process. We finally present the experimental results obtained using our prototype tool.

Chapter 7 concludes the thesis with a critical evaluation of our work. We also highlight the important contributions made by our work. Finally, we discuss the possible future extensions to our work.