Abstract

Object-oriented (OO) software technology is by far the most effective approach for software development. Although, it contributes greatly to the advance of software industry, it still suffers from several drawbacks, especially its well-known fragility for capturing *concerns* [37] crosscutting multiple objects. The crosscutting factor is due to the tangled code implementing the concern among those objects. As a result, the ultimate result is high cost.

As an extension of the OO technology, aspect-oriented software development (AOSD) comes to solve this problem. It emerges as a promising future software paradigm with high reusability, openness among many other advanced characteristics. Throughout this dissertation, the terms concern [37], aspect [17], feature [28, 29], collaboration [10, 27, 35], hyperslice [36], component are interchangeably used. Even though these terms address different levels of concern granularity at various stages in the software lifecycle, they can be equivalently treated. More specifically, the dissertation inclines toward the standard definition of hyperslices which are functions or services offered by the system through the collaboration of several objects.

Feature-oriented software is a special case of AOSD in which a feature corresponds to a functional requirement of the system. In essence, a system is constructed by composing several features. Each feature is independently specified, implemented and maintained. Those features communicate with each other via well-defined interfaces. Based on this design style, a system can be flexibly built depending on the decision of whether or not to include a specific feature.

However, to realize such an ideal software paradigm, one of the key issues is to ensure that those separately specified and implemented features do not conflict to each other when composed - the *feature consistency* issue. This dissertation takes a formal approach toward this problem and its related issues. Basically, the dissertation consists of three parts.

First, a formal model of feature-oriented software is introduced in which each feature is separately encapsulated by a state transition model. The formal model of a feature also provides the interface at which other features are plugged with. With respect to some rules, interface states among different features are mapped to each other so that member features can be composed together to form the target composite system.

Second, the core of this dissertation is about the theoretical foundation of consistency verification among features. A feature is *consistent* with another if during its execution, it does not violate the inherent CTL properties of the latter feature. The verification approach - open incremental model checking (OIMC) - is done via assumption model checking [6, 20]. The difference of OIMC when compared with the other modular model checking techniques [13, 18, 20, 31] is in its *incremental openness*. Under the approach, systems are considered as ever-evolving. A typical evolution scenario is to incorporate new modules (*extensions*) to an existing system (*base*) consistently. In such a circumstance, the new method is executed only in the extension. On the contrary, traditional model checking

methods are required to re-run over the whole system, even though possibly in modular fashion. This is the most prominent feature of this challenging area. A sound theoretical foundation of OIMC is proposed in the dissertation. Its goal is to provide necessary conditions under which the extension components do not violate key properties of the base component. In addition, the soundness and scalability of this incremental verification approach are also discussed. Except in some extreme cases of feature composition, the result delivered by OIMC is sound. Moreover, under the consistency condition, OIMC also preserves its complexity for any subsequent extensions, i.e. scalable.

The final part of the dissertation briefs some possible applications of the proposed theory such as the component specification and composition. In addition, it shows the relationship between the formal specification model of feature-oriented software and its implementation. More specifically, this part shows a basic code-transforming mechanism from the formal model into a specific programming language, e.g. Java. Because features are independently specified, their respective codes are independently generated according to some transformation rules. At the end, the corresponding codes of the features are composed [36] or woven [38] together in a well-defined manner.