Modular Development and Verification of Domain Requirements via Model Checking

Tanmay Bhowmik
Computer Science and Engineering
Mississippi State University
Mississippi State, MS 39762
tb394@msstate.edu

Nan Niu
Computer Science and Engineering
Mississippi State University
Mississippi State, MS 39762
niu@cse.msstate.edu

Edward B. Allen
Computer Science and Engineering
Mississippi State University
Mississippi State, MS 39762
edward.allen@computer.org

ABSTRACT
A holistic domain model of a software product line is costly to build and difficult to verify and evolve. We propose a framework to incrementally develop domain requirements and to iteratively verify behavioral properties through model checking. We leverage state vectors to derive both local and global properties, and co-develop statechart models with temporal specifications in a modular way. We illustrate our framework using a worked example. The study shows that our framework can effectively detect inconsistencies and tighten the development feedback loop by automatically verifying domain properties.

Categories and Subject Descriptors
D.2.1 [Software Engineering]: Requirements/Specifications;
D.2.4 [Software Engineering]: Verification – Model Checking

General Terms
Design, Verification

Keywords
Domain requirements, product lines, verification, model checking

1. INTRODUCTION
Product line engineering is one of the success stories of software reuse whose purpose is to improve quality and productivity. A product line is a set of software systems that satisfy the needs of a particular domain and is developed from a set of assets in a prescribed way [4]. Treating requirements as assets enhances the effectiveness of reuse because engineers can work on the abstractions closer to the domain’s initial concepts [10].

The domain requirements of a product line comprise mandatory and variable requirements [7]. Mandatory requirements apply to all the members, whereas variable requirements apply to some specific product members. Effectively developing domain requirements is essential for a software product line to achieve mass customization and economies of production [4].

Many contemporary methods define domain requirements holistically [6], much like the waterfall model attempting to define a complete and unchanged set of requirements. However, building a holistic domain model has numerous shortcomings [6, 9]. It requires substantial upfront effort, demands compromises from software practitioners, causes abrupt organizational transitions, and provides little support for product line evolution.

In this paper, we propose a framework to incrementally develop a product line’s domain requirements and to iteratively verify behavioral properties through model checking [3]. Our goal is to make the requirements assets more resilient to change in an inexpensive and flexible way. The contributions of our work lie in the support for requirements assets’ modular development and verification. The software community has long realized that modular systems are easier to produce, maintain, and evolve [13]. In addition, our proposed framework tightens the development feedback loop by automatically verifying modules’ local and global properties.

The remainder of this paper is organized as follows. Section 2 lays the background of our research and discusses the related work. Section 3 proposes the modular development and verification framework. Section 4 presents an application of the framework and reports preliminary results. Section 5 draws some concluding remarks and outlines future work.

2. RELATED WORK
Practitioners have traditionally followed the proactive model to develop a holistic set of assets. Although this model can be effective for developing product lines in a mature and stable domain where product features can be predicted, in practice, the large upfront investment presents a prohibitive adoption barrier for many organizations that could otherwise benefit [6]. One principle to economic development is modularity, i.e., decomposing software into high-cohesion and low-coupling modules [13]. Maintaining a clear separation of concerns has been a driving force for many software engineering methodologies, such as object-orientation and distributed software development.

The desired properties of each module can be checked locally. The feedback allows for incremental and iterative development. Consistency checking aims to uncover inconsistencies of a set of modules given known interconnections [14]. Inconsistency refers to any situation that does not obey some rule or relationship that is prescribed to be held [11]. For example, an inconsistency occurs if a (composed) class diagram contains cyclic inheritance.
In a prior project [14], the authors checked conceptual models against some well-formedness rules through model checking.

![Feedback](image)

**Figure 1. Model checking a software specification.**

Software model checking [3] deals with automated verification of specifications or programs. Figure 1 generalizes the process of model checking software specifications. Chan et al. presented one of the first feasibility studies in this area [2]. A software artifact is modeled using state transition diagrams, e.g., UML statecharts [12]. Specifications are written in some temporal logic formulas, e.g., linear temporal logic formulas [3]. A model checker is employed to verify whether the model satisfies the formulas, and if not, to generate a counterexample. The counterexample may reveal a transition inconsistency, a function inconsistency, a reference to an uninitialized value [2], an erroneous specification, or a disagreement among the modelers [5]. In any case, the result can be analyzed by the software engineer to revise and fix the problems, as indicated by the feedback loop in Figure 1.

Lauenroth and Pohl performed consistency checking of product line requirements [7]. Their main contributions included explicit variability modeling and reasoning in a multi-valued situation (true, false, unknown). However, their approach works with domain requirements that are already developed. Lee et al. proposed Constraints-based Modular Petri Nets (CMPNs) as a formal substitute of informal use cases and a set of guidelines to detect inconsistency and incompleteness in CMPNs which deal with requirements elicitation of a single product with an option of add/remove feature [8]. One of our objectives is to develop domain requirements in a modular way, while handling inconsistency interactively. Checking local properties of a module and its evolution was outlined by Niu and Easterbrook [9]. In order to check global properties, we adopt the concept of state vector (a vector of values of the state variables) [15]. Some state variables cut across multiple modules, which allow us to reason about global properties of the application domain.

### 3. Proposed Framework

The basic tenet of our framework is that when developing a domain’s behavioral requirements, we can start with a set of base requirements and represent them as a collection of high-level states. These requirements can then be modularly refined and verified. Let us denote the set of base requirements as \( R_0 = \{ A, B, C, \ldots \} \), where each element of \( R_0 \) is a high-level state. In our framework, we consider two types of properties that the states should satisfy:

- **Global properties**: These are the properties all the states should follow or the system as a whole should follow.
- **Local properties**: These are the properties declared for a particular high-level state and the corresponding refining states should follow these properties.

Both global and local properties are subject to consistency checking. The properties we declare at this point are global properties since the states in \( R_0 \) are not yet refined. Let us denote the set of global properties as \( P_0 = \{ P_1, P_2, P_3, \ldots \} \). Model checking \( R_0 \) against \( P_0 \) leads to prompt verification or error fixing.

![Figure 2. Statechart for \( R_0 \).](image)

We now refine the base requirements \( R_0 \) modularly, i.e., expand every high-level state by introducing low-level states and transitions. The details introduced are within one module, so that the higher-level view can achieve information hiding [13]. In practice, such modular decomposition will support parallel and distributed development activities within an organization, as well as recent movement toward global software development. Note that some newly introduced details will have broader impacts across the module boundary. In such cases, we need to revise the global property set \( P_0 \). The development activities also need to be coordinated to account for the scattered concerns.

During each refinement, we introduce a set of local properties for the state being expanded. We might introduce some global properties, if necessary. We consider the collection of the low-level states as the state vector of the corresponding high-level state, where the low-level states are the state variables [15]. For example, let us assume that, after expanding the high-level state \( A \) in Figure 2, we have 3 low-level states \( a_1, a_2, \) and \( a_3 \). Thus, \( \langle a_1, a_2, a_3 \rangle \) is the state vector of \( A \). Let the local properties of \( A \) be denoted as \( P_A \).

![Figure 3. \( R_{\text{Domain}} \) derived from \( R_0 \).](image)

Similarly, other high-level modules can be expanded and refined. Figure 3 illustrates the derived domain requirements \( R_{\text{Domain}} \) in which only one-level refinement is shown. However, our framework supports iterative multi-level refinements, e.g., \( a_1 \) can be further expanded to \( a_{11}, a_{12}, \) etc. In this way, the higher-level state is a super-state, and lower-level states are sub-states. Such encapsulation (information hiding) further promotes modularity.

The domain requirements model is derived from the base model by composing modular refinements. For instance, an intermediate domain model is \( (R_0 + \Delta_1) \), where \( \Delta_1 \) is the expansion made to the state \( A \). Accordingly, the property set is updated to \( (P_0 + P_1) \). Model checking is performed after each modular development. Thus, it is important to verify \( (R_0 + \Delta_1) \) against \( (P_0 + P_1) \), since expanded model elements may cause the whole property set to become inconsistent [11]. Detected inconsistencies can be incrementally handled by the analyst. The model and property \( \Delta \)’s defined in our framework are particularly useful to model a
4. A PROOF-OF-CONCEPT EXAMPLE

4.1 Cadence SMV Model Checker

We used the Cadence SMV [1] in our study. It is a symbolic model checking tool that allows us to formally verify temporal logic properties of finite state systems [3]. We chose Cadence SMV for the following reasons:

- It has built-in modular design. We can write SMV code as modules for different expansions and compose them with the code which resulted from previous expansions. We can run them as a unit to verify whether there are any conflicts with the previous expansion(s).
- Its input language is similar to a traditional programming language like C, so the learning curve is not steep.
- It can verify properties written in Linear Temporal Logic (LTL) [3] which is sufficient for our example.
- It is freely available, and is considered fast.

Table 1 shows some LTL modal operators [3]. A detailed discussion about the syntax of Cadence SMV is beyond the scope of this paper. Interested readers are referred to [1].

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
<th>Diagram</th>
</tr>
</thead>
<tbody>
<tr>
<td>X ⊗</td>
<td>◦ should hold at the next state.</td>
<td><img src="https://via.placeholder.com/150" alt="Diagram" /></td>
</tr>
<tr>
<td>G ⊗</td>
<td>◦ should hold on the entire subsequent path.</td>
<td><img src="https://via.placeholder.com/150" alt="Diagram" /></td>
</tr>
<tr>
<td>F ⊗</td>
<td>◦ should hold eventually (somewhere on the subsequent path).</td>
<td><img src="https://via.placeholder.com/150" alt="Diagram" /></td>
</tr>
</tbody>
</table>

4.2 Modular Development and Verification

Let us consider a hypothetical Online Railway Reservation System (ORRS) with two major states: query and payment. The query state checks the availability of seats according to the necessity of the customer, and the payment state completes the payment for the reservation. One variant of the ORRS might not include a concession for minors, so a child’s ticket is a variable requirement. The railway also provides passengers with food, and has two alternative payment methods: credit card or check.

![Figure 4. Base ORRS domain requirements.](https://via.placeholder.com/150)

Figure 4 shows the base requirements $R_0$. Two global properties are considered at this point and are expressed in natural language.

- $P_0.1$: The system shall work only for the current transaction, i.e. information from the previous reservation should not affect the current reservation.
- $P_0.2$: After every reservation, the system shall rollback so that another reservation can take place.

The high-level state $Query$ is expanded, where its state vector is $<Source, Destination, Date, No. of Seats, No. of Adults, No. of Minors, Date of Birth>$, as shown in Figure 5. $Source, Destination, Date,$ and $No. of Seats$ can be considered as mandatory requirements; $No. of Minors$ and $Date of Birth$ are variable ones. We derive the following local properties of $Query$:

- $Pq1$: A reservation may include only adults, or both adults and minors.
- $Pq2$: If there is a minor, then the system shall require a date of birth.

![Figure 5. Modular development of Query.](https://via.placeholder.com/150)

Figure 5 shows two global properties $Pq1$ and $Pq2$.

![Figure 6. Cadence SMV code for verifying Query expansion.](https://via.placeholder.com/150)

Figure 6 shows a code snippet written for Cadence SMV to verify the expansion and the properties. $Qudiult$ and $Qgen$ (short form of general query) are the query results for adult only reservation and with minors reservation respectively. We consider two different results to indicate that two product variants might have two different features as mentioned earlier. In the query module, the values of the state variables are initialized to zero, except for $Date of Birth$, since it depends on $No. of Minors$ to hold the global property $Pq1$. Temporal properties $mutex$ and $sequence$ are written to verify the local properties $Pq1$ and $Pq2$ respectively. However, the model checker reports $mutex$ is not satisfied. The counterexample reveals the omission of the conjunct, $¬no_of_minors$, in $Qudiult$. After revision, the model checker successfully verifies the properties. Therefore, we accept our expansion to derive a new set of requirements with properties.

The $Payment$ state is expanded with alternative initial sub-state that models variability in the product line, i.e., the payment could be made by either credit card or check. The local properties include:

- $Pp1$: A transaction can include either card or check, not both.
- $Pp2$: Card should require $Card Information$ and Check should require $Check Information$.
- $Pp3$: Information from the state $Query$ should be available to $Payment$ state since it is required to do the calculation.

Figure 7 shows the final model $R_{Domains}$ for ORRS, in which both high-level states’ refinements are given. Figure 8 shows a code snippet for verifying $R_{Domains}$ by stressing the $Payment$ module.
The inconsistent or invalid, domain model. and ameliorated the risk of building a holistic, but often refinement-verification-improvement tightened the feedback loop, modeling errors, specification errors, and initialization errors. The effectively detected and resolved inconsistencies caused by coordinated but orthogonal plans. In the ORRS study, we so that it was carried out by a single person, our framework has conventional waterfall-like assets development often prevent importance for a software product line. However, the significant framework for parallel development coordination is in order. The proposed framework is a work in progress, and can be extended in many directions. First, a template to capture high-level requirements and global properties might be developed. Second, more sophisticated features, such as variation points and constraints, require further investigation. Third, applying our framework for parallel development coordination is in order. Fourth, we want to explore mechanisms to tolerate inconsistency [11], as opposed to instantly resolve detected inconsistency. This can bring the benefit of late variability binding in the product line.

5. CONCLUSIONS

Developing a consistent set of domain requirements is of great importance for a software product line. However, the significant upfront cost and inflexible consistency checking associated with conventional waterfall-like assets development often prevent practitioners from adopting product line practices.

In this paper, we have proposed a framework for modular development and consistency checking of domain requirements in a product line. We have demonstrated the framework by working out an example containing mandatory, variable, and alternative requirements. Although the example is kept within a small scope so that it was carried out by a single person, our framework has the potential to excel when different teams (possibly geographically distributed) perform parallel development with coordinated but orthogonal plans. In the ORRS study, we effectively detected and resolved inconsistencies caused by modeling errors, specification errors, and initialization errors. The refinement-verification-improvement tightened the feedback loop, and ameliorated the risk of building a holistic, but often inconsistent or invalid, domain model.

The proposed framework is a work in progress, and can be extended in many directions. First, a template to capture high-

6. REFERENCES