# A Tool-supported Technique for Specification & Management of Model-checking Properties for Software Product Lines

Jing (Janet) Liu[1], Miriam Hauptman[1] and Robyn Lutz[1, 2]
1Department of Computer Science, Iowa State University
2Jet Propulsion Laboratory/Caltech
1-515-294-2735
{janetlj, miriamh, rlutz}@cs.iastate.edu

Birgit Geppert, Frank Rößler
Avaya Labs Research
Software Technology
1-908-696-5116
{bgeppert, roessler}@avaya.com

## ABSTRACT
Property specification in model checking is currently handled without adequately taking software product lines into account. This is largely due to the fact that the available model checkers and property specification tools lack sufficient support for reusing model-checking effort. The challenge is twofold: first, we need to make the properties accurately trace to individual system requirements and models even as they evolve; and second, we need to make the property specification easy to share and reuse among different systems of the same product line. The contribution of this work is a tool-supported technique to guide users in generating, selecting, managing, and reusing product-line properties and patterns of properties. The technique is evaluated in a product-line application. Results show that it improves the reusability and traceability of property specifications for model checking in a product line setting.

## Categories and Subject Descriptors
D.2.1 [**Requirements/Specifications**]: Tools; D.2.4 [**Software/Program Verification**]: Model Checking

## General Terms
Management, Design, Verification.

## Keywords
Model Checking, Property Specification, Software Product Lines, Tool Support.

The original FormulaEditor Tool was developed at Avaya Labs Research, NJ, USA, as part of the SELEX project.

## 1. INTRODUCTION
Model checking is a powerful technique for enhancing the quality of software systems [6], e.g., by identifying flaws that would not have been caught otherwise ([14], [18]). However, there is currently insufficient support for model checking in product lines, most specifically, for property specification and management. A software product line is a set of software systems developed by a single company that share a common set of core requirements yet differ amongst each other according to a set of allowable variations [37].

In this paper we present a detailed technique, supported by a property specification and management tool, FormulaEditor, for helping with the model checking of software product lines. The core of the technique is a product-line-oriented user interface to guide users in generating, selecting, managing, and reusing useful product line properties, and patterns of properties for model checking. The tool also associates the properties with the requirements, models and verification results of each product in the product line so that any changes can be readily traced and the properties updated accordingly. The technique is evaluated in applications to telecommunication-protocol and cardiac-pacemaker product lines.

In a product line, some requirements, called *commonalities*, are shared by all the products. For example, a commonality for the pacemaker product line described below is, "When a heartbeat is detected during the SenseTime, the pacemaker shall not generate a pulse." This property is to keep the pacemaker from giving a pulse when it is unneeded by the patient. The difficulty with reuse across a product line is that the differences among the products, called their *variations,* can complicate the implementation and verification of the properties. For example, some pacemakers can distinguish whether a patient is exercising or at rest, and adjust the SenseTime accordingly. This variation means that the specification of the common property described above, in fact, varies slightly among products.

Verification that each new system built in a product line satisfies the common properties takes many forms including inspection, state-based simulation, and testing [2], [24], [37]. However, these techniques do not provide the coverage or level of assurance needed for products in some domains. For example, in our work with both communication protocols and pacemakers, we found

that more rigorous verification to show key properties held in each new product was needed. Model checking, in particular, can provide insights into rare and boundary cases. We thus wanted to be able to model check these properties in the product lines.

Product-line verification, like product-line engineering in general, tries to reuse whatever is common across the product line to reduce the cost and increase the quality of each new product. However, correctly specifying and keeping track of the properties across a product line is a challenge. This is especially true in large product lines where new features are added regularly in response to market pressures. The many commonalities in a product line urge reuse, but the variations among the products demand very careful management of that reuse.

The challenge for model checking a product line is twofold. First, we need to make properties precisely and accurately trace to individual system requirements and models even as they evolve. Second, we need to make the property specifications easy to share and reuse among different systems of the same product line. For example, how do we know if a property should still be satisfied in the presence of some specific variations? For a new system in the product line, how can we easily reuse properties from other product-line members? In this paper we present a technique that is able to associate properties with variations without compromising the completeness or accurateness of properties for individual products. The property specifications captured and maintained in the tool thus become reusable assets of the product line. Thus, verification of common patterns can be enforced in all products in the product line. By making it easier to specify and manage the properties, we hope to extend the use of model checking in product lines. Moreover, this approach and tool can benefit not only safety-critical product lines (e.g., pacemakers, mobile communication devices for emergency workers, constellations of satellites, and medical-imaging systems), but also single systems that experience frequent maintenance or evolution.

The rest of this paper is organized as follows. Section 2 presents needed background information and related work. Section 3 describes the method by presenting a motivating example followed by design rationale and tool support. Section 4 evaluates our technique in a pacemaker product line case study and discusses the results. Finally, Section 5 offers some concluding remarks and possible future work.

## 2. BACKGROUND & RELATED WORK

Formal verification, the application of rigorous mathematical reasoning to prove that a system satisfies certain properties (or formal specifications) [33], has gained increasing importance in the software industry. This is particularly true for safety-critical and mission-critical software systems.

Model checking gained favor among various formal verification techniques because of its automated verification procedure and the close resemblance between modeling language and high-level programming languages [6]. A model checker accepts formal models of system requirements and design [14], [27] or sometimes implementation [7], as well as some desired or undesired properties of the system [6], and then employs systematic exploration of the execution paths (exhaustive exploration if the model size is manageable) of the model to see if those properties are satisfied or not [16]. Researchers have used model checking to find flaws in software designs that are otherwise hard to detect [14], e.g., by testing.

If a single product can be decomposed into a set of units that can be designed and implemented separately, we call them "individually-behaving units" (e.g., components in component-based system [2] or collaborations in collaboration-based protocol design [12]). A product line can also be viewed as different compositions of such units [30]. No matter which form the product line is taking, one obstacle to model checking product lines is reuse management that accurately reflect the variations. The commonalities (or common units) in product lines make it possible to reuse some model checking effort (e.g., models created, properties specified). However, the variations (or different units) and the new dependencies (constraints between commonalities and variations, or among variations) or different composition of units they introduce can make it difficult to identify the properties to verify for each variation introduced. This currently limits the potential for reuse of model-checking assets.

Thus, there is a need for property specification techniques that can incorporate the concept of product-line variation and reuse. When it comes to the product-line context, it is not sufficient to simply tie those specifications into model checking in an ad-hoc manner. The enumeration of all possible properties for a single product can be overwhelming if no systematic reuse is applied. Also, unless automatic traceability from requirements to properties and from properties to models exists, the properties are not likely to be maintained as the system evolves.

Existing work has indicated the possibility of successfully conducting model checking for software product lines, e.g., Kishi and Noda [19] proposed an approach that models product-line variations in UML models and then translated them into SPIN models; Li, Krishnamurthi, and Fisler [22] have exploited compositional verification in the product-line context by automatically checking interfaces of separate features using the labeling algorithm in CTL model checking; and Robby, Dywer, and Hatcliff [32] have constructed Bogor, an extensible model-checking framework that can be customized to tailor to different application domains, e.g., to be used as a back-end model checker for Cadena [5]– an integrated environment for building and modeling CORBA Component Model systems – that can be used to develop model-driven component-based product lines.

A major issue remaining unaddressed in work to date is the management of property specifications at the product-line scope. Traditionally, the properties being verified are derived from requirements [20] or subsystem/component/interface specifications [21]. Several techniques have been developed to ease the difficulty of translating informal (natural language) specifications into formal ones (e.g., temporal logic formulas [16]), such as the Property Specification Patterns [9] and various work that helps select and adapt those patterns ([7], [17], [26], [29], and [34]), a set of tools to help edit the LTL temporal logic properties in a communication diagrams ([3], [35]), techniques that translate a subset of natural language ([15], [20]) or specification language (syntactic sugar) [4] into temporal logics, and syntax-directed editing environment [31]. However, these techniques, to the best of our knowledge, do not treat property specification in a reusable setting.

## 3. METHOD

This section describes our property specification technique for software product lines, including a motivating example, the design rationale behind the work, and a description of the tool support.

### 3.1 Motivating Example

The work was motivated by the need to model check a family of communication protocols that resulted from an Avaya refactoring project [11], [12]. In the following, we call the members of the protocol family "protocol variants". Each protocol variant is composed of a set of smaller building blocks (called *collaborations*) that encapsulate behavior across agent boundaries. Agents are the distributed participants in the communication the protocol regulates Common collaboration examples are connection establishment, connection tear down, and authentication.

There are two mechanisms for composing collaborations. The first mechanism is message multiplexing. If we treat each collaboration as a micro-protocol (in contrast to the composite protocol after composition), the outbound messages sent by the micro-protocols (we call them "micro-messages") must be multiplexed together to form a composite message and be considered as one unit for transportation over the network. Consequently, any incoming composite messages need to be demultiplexed into micro-messages to be handled by micro-protocols upon arrival.

The second mechanism is sequencing, which manages the causal dependencies among the collaborations. We can divide a protocol agent into several roles, one for each collaboration in which the agent participates. The roles represent independent state machines. Protocol sequencing takes care that the roles are executed in the right order.

In contrast to traditional protocol design where a protocol is viewed as the composition of protocol agents (each of which can be represented by a state machine), the collaboration-based design enables users to analyze, test, and change cross-cutting behavior independently, thus allowing easier evolution and maintenance ([11], [12]).

Fig. 1 shows part of a protocol for registering IP phones at IP telephony servers. There are four agents, namely the *endpoint*, the *station server*, the *gatekeeper*, and the *environment.* (The *environment* is not shown in Fig. 1 but is needed for modeling user input and making the protocol state machine finite). The *environment* and *gatekeeper* together with the *endpoint* implement the *authentication* collaboration. It is a four-way handshake to authenticate an *endpoint*. The *environment*, *endpoint, gatekeeper*, and the *station server* collaborate to implement the *associate-station* collaboration. It consists of one two-way handshake between each of the two adjacent agents in order to allocate necessary resources for an *endpoint.*

As a motivating example, a simplified protocol product line consists of three protocol variants: 1) the authentication protocol includes the *authentication* collaboration only, 2) the associate-station protocol includes the *associate-station* collaboration only, and 3) the authentication-associate-station protocol includes the composition of the two collaborations. The complete protocol product line is much more complex due to a larger number of collaborations and compositions.

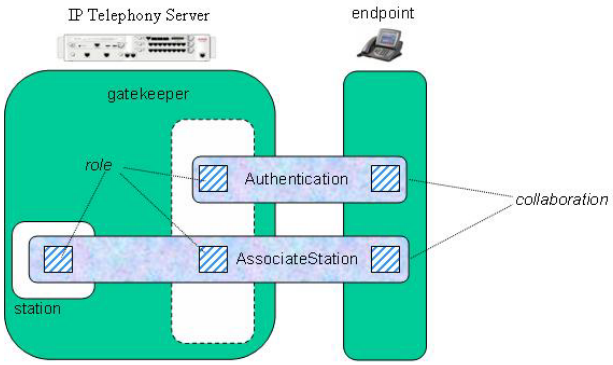


**Figure 1. Registration Protocol (Partial Overview).**

An example property (specified as a natural language description) for the authentication protocol is: "It is always the case that whenever the *environment* receives a REGISTER_CONFIRM message, the *authentication* role in the *gatekeeper* is in *authenticated* state". An example property for the associate-station protocol is "It is always the case that whenever the *environment* receives a REGISTER_CONFIRM message, the *associate-station* role in the *gatekeeper* is in *bound* state and the *associate-station* role in the *station server* is in *stim* state".

The motivation for this tool is the recognition that the properties for these two protocols have a lot in common. For example, the example properties in both protocols share the following parameterized property: It is always the case that whenever the *environment* receives a REGISTER_CONFIRM message, the *A* role in the *B agent* is in *C* state. A, B, C are parameters to be instantiated by concrete role, agent, or state in that specific protocol. A similar pattern can apply to the other properties between the two protocols. This led us to provide parameterized properties that could be reused for different collaborations in different protocol variants.

Since the authentication-associate-station protocol is the composition of the above two collaborations, not only properties from the two collaborations need to be verified again in this protocol, but also the properties regarding the compositional logic need to be verified. This includes making sure that the compositional mechanism is correctly enforced as well as that the composed protocol is behaving as expected. An example of the former is that any incoming composite message is always completely consumed, meaning that all of its inbound micro-messages are always eventually consumed. An example of the latter is that in the authentication-associate-station protocol, the *authentication* collaboration has to be successful before the *associate-station* collaboration can be invoked.

An example property for the authentication-associate-station protocol is: "It is always the case that whenever the *environment* receives a REGISTER_CONFIRM message, the *Associate-Station* role in the *gatekeeper* is in *bound* state, the *Authentication*

role in the *gatekeeper* is in *authenticated* state, and *Associate-Station* role in the *station server* is in *stim* state".

Due to space limitations we cannot list all the properties of the authentication-associate-station protocol here. In fact, if more than one collaboration is involved in a protocol, the number of relationships among incoming/outgoing messages, inbound/outbound micro-messages, and the various states of different roles in the protocol agents, can be exponential.

The key to addressing this problem is to provide tool support to allow recurring patterns in the product line to be identified and instantiated so that a batch of properties can be created and reused. The use of recurring patterns not only contributes to the completeness of the properties, but also speeds up specification.

The potential patterns include but are not limited to the following two categories: patterns of collaborations and patterns of compositional logic. An example of the former is that "It is always the case that whenever the *environment* receives a REGISTER_CONFIRM message, the *A* role in the *B agent* is in *C* state". An example of the latter is that "any incoming composite message is always completely consumed". Those patterns, once identified, became an asset of the product line so that verification of common patterns can be enforced in all products.

## 3.2 Design Rationale

A straightforward way to check the properties for each of the three protocol variants of the protocol product line described above is to specify each of them one by one and invoke a model checker to verify them. However, for even a small-scale product line, e.g., of twenty collaborations and ten possible compositions, the work involved in tracking all the properties to be verified for each protocol variant became quite hard to manage. Gearing the product-line property specification process to reuse is therefore of great importance to model-checking industrial product lines.

We identified the following needs for property specification and management in a product-line scope. These needs provided the design rationale behind the tool we developed:

1. The tool needs to keep track of which requirement(s) each property specification is derived from and to make use of common property patterns in the product line. This includes: 1) associating properties with individual requirements. Note that the satisfaction of one requirement may entail verification of several properties, each targeting a different aspect of the requirement, e.g., requirement on composition mechanism can turn out to be properties regarding all the micro-messages being consumed correctly; and 2) introducing product-line specific property patterns that can be instantiated for individual members of the product line. In contrast to the Property Specification Patterns [9], the product-line specific patterns are intended as reusable assets for the specific product line only. It is thus possible that two different product-line patterns share the same Property Specification Pattern while differing in the instantiation rules for their parameters (e.g., some parameters may only be able to be instantiated by messages from a certain agent) or their scope (e.g., some pattern can only be used in certain protocol variants).

2. The tool needs to keep track of which product-line member it is targeting. This includes associating properties with their models, as well as with the verification results. This is because the models determine the validity of the properties being specified. If a property is shown to be false, the requirement may need to be modified (e.g., such a property may be an explanation for an ambiguous requirement).

The next section will describe how the above design rationale is supported by our tool.

## 3.3 Tool Support

The work presented in this section demonstrates a model checking management tool, FormulaEditor, which we developed to support the property specification and management for product lines.

The architecture overview of the tool is presented in Fig. 2. The figure shows the tool being applied to a product line of three members A, B, and C. The part enclosed in dotted lines is not part of the tool but serves as input to it.

The tool has three main functions: project management, property editing and model checking.

**Project management**. As mentioned before, property specification is not an isolated process. Therefore, the tool helps users manage the resources needed for linking the property specification with the rest of the model checking and product line development effort.

Each product line is managed as a project. The tool provides a project configuration interface to let users create and specify settings for resources that may be shared by the entire product line (e.g., model checker location, common property file etc.). Those common settings are loaded every time the project is opened, e.g., the common-property pattern file will show up in property-editing for all models in the project.

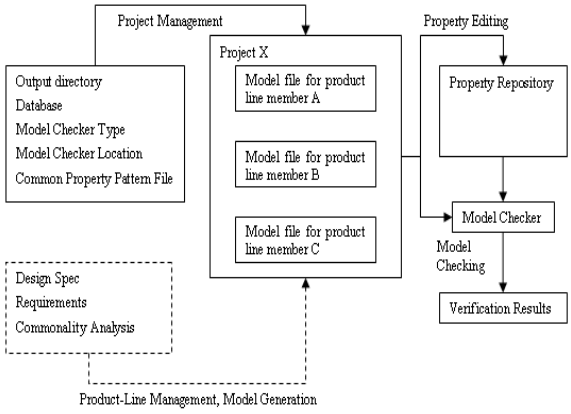


**Figure 2. FormulaEditor Architecture Overview.**

A property table contains all the properties specified for that product (e.g., see Fig. 5). The importing-property-table and import-project facility in the project management interface allow a set of properties specified for one system or one product line to be copied to another system or product line, leaving out the ones that are no longer valid for the new system. This is accomplished by automatically detecting atoms in the property that do not belong to the system. Version control is enforced at single-system

granularity by detecting and clearing out-of-sync verification results (i.e., a verification done before its model file changed is out of sync).

**Property editing**. Property editing serves to generate product-line specific patterns and product-specific properties. Fig. 3 shows the interface for property editing. It is divided into three areas: the upper area for selecting the building blocks of a property, the middle area for composing a property in natural language, and the lower area for composing or viewing a property as a temporal logic formula. The preset patterns are shown in the pattern selection part of the upper area. The default patterns are a complete set of basic LTL and CTL patterns that can be used to form any other LTL and CTL formulas [16].

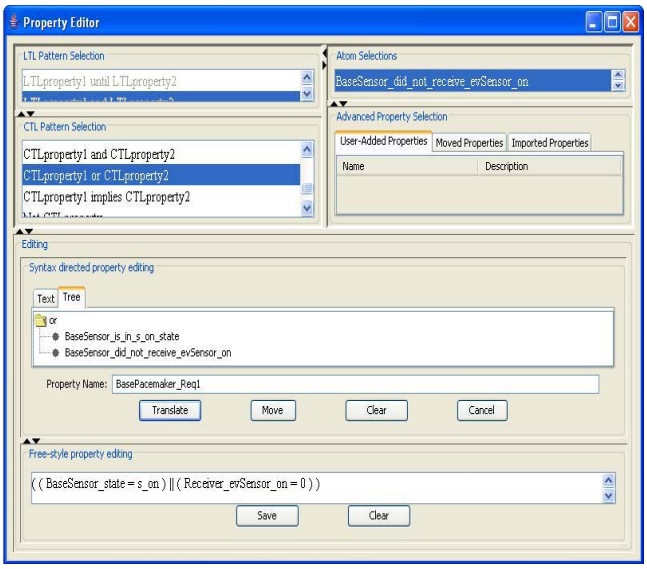


**Figure 3. Property Editing Overview.**

To provide traceability between the properties and the requirements, the tool also supports domain-customized natural language descriptions for properties and allows users to tag properties with the requirements from which they are derived.

In the product-line domain-engineering phase (i.e., generating product-line reusable assets [37]), users can create parameterized properties from the set of preset patterns (e.g., basic LTL and CTL patterns or Property Specification Patterns [9] that recur in the product line). Those new parameterized properties then can be added to the preset patterns (stored in a common-pattern-file). The patterns can be reused for property specification both within the same system and across different systems in the product line.

In the product-line application-engineering phase (i.e., generating product-specific assets [37]), users can instantiate the above-mentioned patterns (both preset ones and the ones created by the user on-the-fly) by replacing the parameters in the pattern with concrete properties for the model file. They can also replace the parameters with other patterns to customize a generic pattern.

The tool provides two mechanisms to ensure that the instantiation process is done correctly. The first is to present users with a selection of properties to instantiate the patterns. These include the atomic properties (which we call "atoms", e.g., a variable defined in the model holding a value) extracted from the model as well as past specified properties and patterns for this model. The second mechanism is to provide a syntax-directed editing environment to prevent ill-formed properties from being created (e.g., mixing LTL with CTL properties) or saved (e.g., not-fully-instantiated properties, bad syntax) into the property pool for a specific product member.

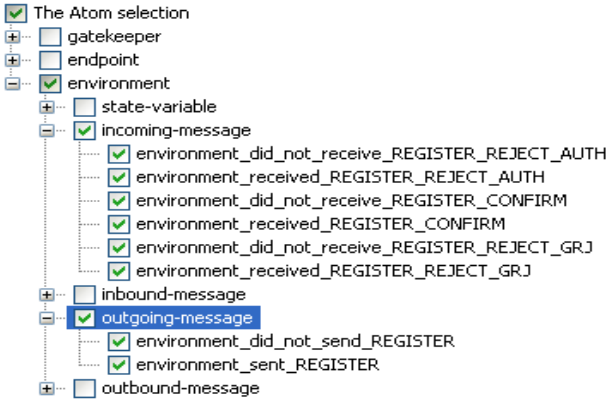


**Figure 4. Atom Selection Overview.**

For example, "( ForAllPaths always ( ( environment_sent_REGISTER ) implies ( ForAllPaths eventually CTLproperty ) ) )" is the natural language description of a parameterized property derived from the Response Property Pattern [9], yet partially instantiated with the "environment_sent_REGISTER" message. The "LTLproperty" and the "CTLproperty" are parameters to be filled with specific properties in the application engineering phase. The "environment_sent_REGISTER" message is selected from the "atom selection overview" window, popped out by clicking the "Atom Selections" in the upper area of Fig. 3. A screenshot of the atom selection overview is shown in Fig. 4.

FormulaEditor recognizes variable declarations in SMV models as atoms in a model. The tool then translates the atoms into its corresponding basic event description, following a set of translation rules approved by the domain engineers. In this way the atoms presented to a user are domain-customized descriptions, rather than their original forms in the model. For example, in the SELEX project, the variable Authenticate_In denotes an inbound micro-message, and we are only interested in whether the micro-message is consumed or not (meaning that Authenticate_In is zero or not). Therefore even though in the model Authenticate_In can range from 0 to 9, only two atoms are extracted, i.e., "Authenticate_In _consumed" and "Authenticate_In _waiting".

**Model checking**. For the model checking part of the tool, both the property and the model are fed into a model checker. (In the work described here, we used Cadence-SMV [27] as the backend model checker, but we also have used CMU-SMV [28].) The tool provides the following facilities to help manage the model checking process in a product-line setting (as seen in Fig. 5)

1. Automatically formats a generic LTL or CTL property to the form recognized by the back-end model checker; allows verification of multiple properties at a time.

2. Provides easy access to all information associated with a property (e.g., verification results, temporal logic type, counterexample, requirement it comes from etc.) and can group the properties according to different criteria (e.g., group by verification results, or group by requirements).

3. Supports conditional verification when it is allowed by the underlying model checker (Cadence-SMV in this case), i.e., users can select some existing properties as assumptions for another property. Those assumptions can be turned on or off before checking, and the enabled assumptions are an integral part of the verification result.

# 4. CASE STUDY

In the previous sections we introduced our tool-supported technique based on the following two assumptions:

1) Property specification for model checking in an ad-hoc manner (without automatic linkage from the requirements to the models) is not likely to be reused as the system evolves.

2) Property specification for product lines without support for reuse can be tedious and hard to manage.

The claim is that since our technique overcomes the above two obstacles, it allows product-line property specification being conducted in a more efficient and sustainable manner.

In this section we step through one case study that we used to test the above claim. Since the focus of the technique is on property specification and management, we do not address model generation here.

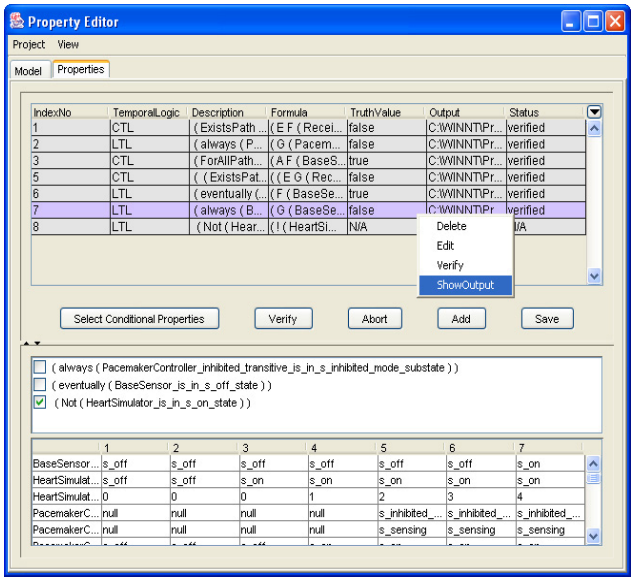


**Figure 5. Model Checking Management Overview.**

## 4.1 Pacemaker Product Line

The second application of FormulaEditor was to a cardiac pacemaker product line.

In previous work ([23], [24], [25]), we developed state machine models for each of the products in this product line using UML statecharts. We chose Cadence-SMV as our model checker because of the existing knowledge on translating UML statecharts to Cadence-SMV (e.g., [36]). The model translation is currently done manually. We take the safety properties to check from the systems' safety requirements.

The reason for choosing this product line as our case study is that it is an example of a safety-critical product line that would benefit from the added assurance of formal verification. In addition, this product line keeps growing, so being able to maintain traceability among properties as the models evolve and manage the re-verification process are essential if the project is to make a smooth transition into model checking from model-based development.

A pacemaker [10] is an embedded medical device designed to monitor and regulate the beating of the heart when it is not beating at a normal rate. It consists of a monitoring device embedded in the chest as well as a set of pacing leads (wires) from the monitoring device into the chambers of the heart. Here, we only consider a single-chambered product line of pacemakers that does pacing and sensing in the heart's ventricles.

The product line consists of the following four products:

**BasePacemaker** – This product has the basic functionality shared by all pacemakers: generating a pulse whenever no heart beat is sensed during the sensing interval.

**ModeTransitivePacemaker** – This product can switch between InhibitedMode and TriggeredMode during runtime. In the InhibitedMode, the pacemaker acts exactly like a BasePacemaker. In the TriggeredMode, a pulse follows every heartbeat to provide a different type of therapy.

**RateResponsivePacemaker** – This product acts similarly to the BasePacemaker but contains an extra sensor allowing it to adjust its sensing interval according to the patient's current activity level: LRLrate, for a patient's normal activities and URL rate, for when a patient is exercising.

**ModeTransitive-RateResponsivePacemaker** – This product combines the features of the ModeTransitivePackemaker and the RateResponsivePacemaker.

We model the pacemaker as a component-based system [2] , meaning that components serve as the functional units that are composed for each product. The following components were modeled in Cadence-SMV: a sensing component (BaseSensor) that senses heart beat, a stimulation component (PulseGenerator) that generates pulses to the heart, a controlling component (PacemakerController) that configures different pacing and sensing algorithms and issues commands, MotionSimulator and HeartSimulator components that simulate the patient's motion and heart beat activities respectively, and a ExtraSensor component that senses a patient's activity level.

## 4.2 Product Line Property Specification

In this section, we describe the entire model-checking process, but focus on Steps 3-5 which use the FormulaEditor tool.

*1. Obtain the set of safety critical requirements that need to be model checked.*

**Sample Requirements for BasePacemaker:**

**R1**. When PacemakerController is in Sensing state and it receives a sensed event from BaseSensor, then PacemakerController shall command the BaseSensor to turn off at the next time unit.

**R2.** Whenever PacemakerController is in Refactoring state, PulseGenerator shall remain in idle state.

**R3.** When PacemakerController is in Sensing state and the SenseTime is up, the PacemakerController shall command the BaseSensor to turn off at the next time unit.

**Sample Requirements for ModeTransitivePacemaker:**

**R1, R2: same as BasePacemaker**

**R3.1** In Inhibited mode, when PacemakerController is in Sensing state and the SenseTime is up, the PacemakerController shall command the BaseSensor to turn off at the next time unit.

**R3.2** In Triggered mode, when PacemakerController is in Sensing state and there is no heartbeat sensed by BaseSensor, the PacemakerController remains in Sensing state the next time unit.

*2. Conduct a commonality analysis [37] of the requirements to find recurring requirements.*

We show such requirements here with the same number in different products (e.g., R1 in BasePacemaker and R1 in ModeTransitivePacemaker) as common requirements and requirements with associated numbering (e.g., R3.1 and R3.2 in ModeTransitivePacemaker) as a variation.

*3. Starting from the product that has the fewest variabilities, build a core set of properties and patterns to be reused from its requirements*

We started from BasePacemaker, as all its requirements appeared in whole (i.e., R1, R2, R5) or in part (R3, R4) in the other three products. By adopting an incremental process, specifying the product with fewer variabilitiees first can maximize the reusability of any specified property set. Each time we add a new property, we first scan through the list of *properties* already specified or imported to see if we can reuse any existing ones by minimal modification effort. If not, we scan through the *patterns* to find one that we can instantiate with minimal instantiation effort. If we compose a new pattern out of existing patterns for a new property, we save that pattern to the common pattern file for product line reuse.

For example, the following pattern set was built for BasePacemaker (we use the natural language description here rather than the temporal logic formula because that is the format we view when we search for possibility of reuse):

Pattern 1:( always ( ( LTLproperty1 and LTLproperty2 ) implies ( next LTLproperty ) ) )

Pattern 2: ( always ( LTLproperty1 implies LTLproperty2 ) )

Pattern 1 was built while specifying the property for R1. Pattern 2 was built while specifying the property for R2. These patterns were subsequently reused in the property specification for other requirements. When we specified properties for R3 and R5, we used Pattern 1 and Pattern 2 separately. When we specified property for R4, we reused property for R3 with one replacement of an atom in it.

*4. For each product, build its own property set by importing likely property sets previously specified. These are then reused and the product's own variations added (i.e., modifying properties and instantiating patterns).*

For example, both ModeTransitivePacemaker and RateResponsivePacemaker share requirements R1, R2, and R5 with BasePacemaker. They differ only in R3 and R4 by introducing their own variations: Inhibited and Triggered mode for ModeTransitivePacemaker, LRLrate and URLrate for RateResponsivePacemaker.

Thus, for each product, we imported the entire property table of BasePacemaker and then modified the properties for R3 and R4 respectively. By doing this, we shared the property set built for BasePacemaker. We also shared the pattern set because the patterns identified while specifying properties for BasePacemaker were saved to the common pattern file and became part of the property editing interface.

For ModeTransitive-RateResponsivePacemaker, we first imported the property tables for ModeTransitivePacemaker and then appended the property table for RateResponsivePacemaker, deleting any repeated properties and renaming different properties that happened to have the same name (the deleting and renaming process was manually done). In contrast with importing the property table only from the BasePacemaker, or only from ModeTransitivePacemaker or RateResponsivePacemaker, being able to import multiple tables had the benefit of maximizing the property pool for ModeTransitive-RateResponsivePacemaker. A discussion of this is given in Section 4.3.

*5. Invoke the underlying model checker to check all properties in the property set for each product. This was done in both incremental mode as the property set was being built and in batch mode to verify or re-verify as commonalities were imported.*

Step 5 interleaves with step 4 in that for each property set built, we model check it with Cadence-SMV right away. This is very important since we build the property set as the variabilities accumulate, so that a flaw detected in the model or in the property for one product will not propagate to other products that share its requirements or designs.

## 4.3 Discussion

1. *Effort saved by using this technique*

At the beginning of the process (step 1 in the above section), we identified 28 requirements to be model-checked. In BasePacemaker, by the end of the process, 2 property patterns were identified, 2 properties were specified by instantiating those two patterns, and 1 property was specified by reusing another property specified for BasePacemaker with some modifications. In the other three products, 3 properties were specified by reusing existing properties in BasePacemaker without any modification.

In ModeTransitivePacemaker and RateResponsivePacemaker, an additional 3 properties were specified by reusing BasePacemaker properties with some modifications. In ModeTransitive-RateResponsivePacemaker, 6 properties were specified by reusing ModeTransitivePacemaker properties with some modifications.

Modifying previously specified properties is similar to changing the instantiation of patterns as both the old and the new properties share similar structures. Therefore, among the property specification for the 28 requirements, 17 of them (about 61%) actually benefited from the product-line specific pattern approach.

2. *Pattern reuse and property reuse*

There are two types of reuse that were useful in this case study: pattern reuse and property reuse. Both can be used for specifying properties for a single product or across a product line. However, property reuse has the limitation that a property to be reused cannot contain any variables not defined in the underlying model. Thus, property reuse is more suitable for commonalities within a product-line scope, and carries the assumption that models for different products in the same product line adhere to a consistent naming conventions. Pattern reuse, on the other hand, does not have the above limitation and can be reused in any system. However, pattern reuse usually requires more instantiation effort than property reuse because it is more general.

A possible solution is to have partially instantiated patterns for reuse, i.e., we can replace the variation part in a property by generic parameters, or instantiate the common part in a pattern by atoms shared by all products.

3. *Utilization of verification results*

Two types of verification results were found helpful in identifying flaws:

1) Properties derived from requirements that were supposed to be true but produced counterexamples. In this situation, the counterexamples provided by the model checker were helpful in locating the faults. For example, the property for R2 in the ModeTransitive-RateResponsivePacemaker was initially shown to be false. The counterexample showed that the PulseGenerator stayed in Pulsing state after generating a pulse, indicating an error in the state-transition logic for the PulseGenerator component. An inspection of the model revealed an inconsistency when translating from UML-statechart to Cadence-SMV code.

The counterexample itself sometimes revealed the problem without going back to the model. For example, when we changed "PacemakerController shall command the BaseSensor to turn off at the next time unit" in R1 for BasePacemaker to "BaseSensor shall be in the off state at the next time unit", the property derived from this modified requirement was shown to be false. The counterexample showed that the state-change for BaseSensor lagged two time units behind the command-issuing in PacemakerController. Thus, either the requirement had to be modified to be "When PacemakerController is in Sensing state and it receives a sensed event from BaseSensor, then PacemakerController shall command the BaseSensor to turn off at the next three time unit" in order to be valid for the system, or the system design (the model here) had to be changed.

The counterexamples showed the values that each involved variable took in a trace leading to the false result. Currently, variable names shown in counterexamples are the same as in the model. Thus, good naming conventions helped locate faults in the system design. We realize that there currently exists a gap between variable names in the counterexample and variable declaration in the atom-selection, as the latter is sugar-coated by giving domain-customized descriptions. We plan to address this issue by interpreting counterexample in terms of atoms.

2) Properties derived from variations of requirements that were supposed to be false but verified true

Verifying variations of requirements helped identify several vacuously true properties [1], i.e., properties that hold due to undesired reasons. For example, if we were to change requirement R4.1 for RateResponsivePacemaker from "when PacemakerController is in LRL (normal rate) state and in Sensing state, and the SenseTime is up, the PacemakerController shall command the PulseGenerator to generate a pulse at the next time unit" to "when PacemakerController is in LRL (normal rate) state and in Sensing state, and the SenseTimeShort is up, the PacemakerController shall command the PulseGenerator to generate a pulse at the next time unit" and it still held for the system, then we would need to check if the PacemakerController always commanded the PulseGenerator to generate a pulse disregard of SensTime up or not.

In this case study, we changed different atom instantiations when reusing a pattern or property, to detect potentially vacuously true properties. As we mentioned before, the benefit of maximizing the property pool was for exactly this reason, since two similar properties are not necessarily both true in the same model.

An added value of the tool-supported technique was in detecting hidden requirements by testing different variations of one requirement. The tool makes it convenient to instantiate different property variations. Moreover, those variations specified for one requirement in a product could be shared via property-table-import by other products if they had a similar requirement, and be quickly verified using batch-verification.

4. *Support for evolution*

1) Product-line change adaptation. Product lines evolve quickly, so to be effective, a tool must readily accommodate frequent changes. We categorize possible changes to a product line and discuss how FormulaEditor handles each of them: (a) New commonality: augment the core property/pattern set to include the new commonality, and in each product in the product line, import the property/pattern derived from this commonality and make modifications when necessary. (b) New variability: augment the property set for all affected products to include this new variability, and augment the common pattern file if new patterns are introduced during property-derivation for such a variability. (c) Delete/modify existing commonality: update the core property set, and update the property set for every product in the product line. The common pattern file may be updated as well if new patterns are introduced during modification. (d) Delete/modify existing variability: update the property set for all affected products. The common pattern file may be updated as well if new patterns are introduced.

2) Model change adaptation using conditional verification. Note that sometimes the property update is done ahead of the model update, i.e., the model has not yet incorporated the new

requirement, since the update of the model usually takes more time than updating the properties. The updated added properties can then be set as *conditional properties* that are assumed to be true for the incomplete model. Existing properties can continue be verified given those conditional properties. Once the model is completed, we restore those properties as normal properties and verify them as before.

3) Property change adaptation. One restriction of the approach mentioned in the paper is that while it is useful in incrementally building a property set for each product in a product line, changes in properties require manually updating each property set affected. Importing property tables can speed this process, but can still be labor-intensive given a large product line context.

A better solution could be to specify just the composition of a property specification from parameterized building blocks and automatically instantiate the full property specification once we instantiate the decision model for the product line [37]. Geppert, Li, Rößler, and Weiss [13] have successfully applied the idea to test-case specification generation for product lines.

5. *Mapping from requirements to properties*

One benefit of property-editing tools (see the Related Work section) is to help users develop formal, unambiguous requirements suitable for model checking. This, when reflected in our technique, is detecting one-to-many mappings between requirements and atoms. For example, the initial requirement R1 for BasePacemaker was "when PacemakerController is in Sensing state and it receives a sensed event from BaseSensor, then BaseSensor should be turned off immediately". The requirement "BaseSensor should be turned off" is associated with a chain of events in the model, i.e., "PacemakerController sent evSensor off", "BaseSensor received evSensor Off" and "BaseSensor is in off state". Those events, although all mapping to the same requirement, do not happen within the same step. Thus, instantiating the same pattern with different atoms generated different verification results. Tying the property specification to the model helped us identify potential ambiguities (since the atoms for instantiating the patterns are all extracted from the model). With the backend model checker integrated into the tool, we were able to quickly verify a group of properties to get detailed information about how to dismiss the ambiguity, e.g., by changing "BaseSensor should be turned off immedately" to "PacemakerController shall command the BaseSensor to turn off at the next time unit".

The insight gained by inspecting one-to-many mappings can also help generate accurate properties for more complex component-based systems or telecommunication protocols, where the delay in response-chain inside components or in network communication can significantly affect the validity of certain requirements.

Another benefit is that by reusing properties and patterns in the product-line scope, users can significantly reduce the effort of deriving the right properties from requirements. In addition, if a pattern is found to be incorrect, then the properties instantiated from it need to be corrected accordingly.

6. *Possible applications*

Although this technique starts from a product-line point of view, it is not confined to use in a product line setting. We have, e.g., applied our technique to NASA's Simplified Aid for EVA Rescue (SAFER) system [8]. Initial results suggest that this technique can help re-engineer existing property specifications for legacy systems to support reuse in future system developments.

# 5. CONCLUSION

The work described here provides a tool-supported technique that guides users in structured reuse of property specifications for model-checking the members of a product line. Properties specified via the tool are traceable to the underlying product-line requirements, the SMV models, and the verification results. The tool enables reuse of shared product-line properties, as well as of product-line-specific patterns of properties, while carefully preserving any distinctions among the product-line members. Results from application to the two product lines show that the tool also can manage the changes and re-verification needed as the product line evolves (e.g., as new members or features are added). By making it easier to specify, manage, and reuse properties, we hope to make model-checking of product-line systems more practical.

In the future we plan to make the technique more flexible by: 1) making the atom-extraction rules easier to modify so that users can change them at the time of specification; 2) investigating the automatic instantiation of property patterns; and 3) allowing properties specified elsewhere to be managed more easily, by extending the property reuse management capability to allow clean interfaces with other tools.

# 6. ACKNOWLEDGMENTS

This research was supported by the National Science Foundation under grants 0204139, 0205588 and 0541163. We thank Dr. Ben Di Vito for providing the CMU-SMV code and properties for the SAFER case study; Dr. Jeffrey Thompson for feedback on the models and properties in the pacemaker case study; and Josh Dehlinger, Hongyu Sun and Wei Zhang for feedback on an earlier version of the tool. Our gratitude is furthermore extended to Dr. David M. Weiss for his support in conducting this research.

# 7. REFERENCES

[1] Armoni, R., Fix, L. et. al. Enhanced Vacuity Detection in Linear Temporal Logic. In *Proc. of CAV'03* (Boulder, USA, July 8-12, 2003). Springer 2003, 368-380.

[2] Atkinson, C. et. al. *Component-Based Product Line Engineering with UML*. Addison-Wesley, 2002.

[3] Autili, M., Inverardi, P., and Pelliccione, P. A scenario based notation for specifying temporal properties. In *Proc. of the SCESM'06* (Shanghai, China, May 27, 2006). ACM Press, 21-28.

[4] Beer, I. et. al. The Temporal Logic Sugar. In *Proc. of CAV' 01* (Pasadena, CA, June 20-23, 2001). Springer, 363-367.

[5] Childs, A. et. al. CALM and Cadena: Metamodeling for Component-Based Product-Line Development. *IEEE Computer, 39,* 2 (Feb. 2006), 42-50.

[6] Clarke, E. M., Grumberg, O., and Peled D. A. *Model Checking*. The MIT Press, 2000.

[7] Corbett, J. et. al. Bandera: Extracting Finite-state Models from Java Source code. In *Proc. of ICSE'00* (Limerick, Ireland, June 4-11, 2000). ACM Press, 439-448.

[8] Di Vito, B. High-automation proofs for properties of requirements models. *International Journal on software Tools for Technology Transfer, 3,* 1 (Sept. 2000), 20-31.

[9] Dwyer, M. B., Avrunin, G.S., and Corbett, J.C. Patterns in property specifications for finite-state verification. In *Proc. of ICSE'99* (LA, USA, May 16-22, 1999). ACM Press, 1999, 411-420.

[10] Ellenbogen, K.A., and Wood, M.A. *Cardiac Pacing and ICDs*. Blackwell Science, Inc. 2005.

[11] Geppert, B., Mockus, A., and Rößler, F. Refactoring for Changeability: A way to go? In *Proc. of METRICS 2005* (Como, Italy, Sept. 19-22, 2005). IEEE Computer Society 2005, 13.

[12] Geppert, B., and Rößler, F. Effects of Refactoring Legacy Protocol Implementations: A Case Study. *In Proc. of METRICS 2004* (Chicago, USA, Sept. 14-16, 2004). IEEE Computer Society 2004, 14-25.

[13] Geppert, B., Li, J., Rößler, F., and Weiss, D. Towards Generating Acceptance Tests for Product Lines. In *Proc. of ICSR'04* (Madrid, Spain, July 5-9, 2004). Springer 2004, 35-48.

[14] Havelund, K., Lowry, M., and Penix, J. Formal Analysis of a Space-Craft Controller using SPIN. *IEEE Trans. on Software Engineering, 27,* 8 (Aug. 2001), 749-765.

[15] Holt, A. Formal Verification with Natural Language Specifications: Guidelines, Experiments and Lessons So Far. *South African Computer Journal, 24,* (Nov. 1999) 253-257.

[16] Huth, M., and Ryan, M. *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press, 2004.

[17] Jörges, S., Margaria, T., and Steffen, B. FormulaBuilder: a tool for graph-based modeling and generation of formulae. *In Proc. of ICSE'06* (Shanghai, China, May 20-28, 2006). ACM Press, 815-818.

[18] Kaivola, R. Formal Verification of Pentium Pentium® 4 Components with Symbolic Simulation and Inductive Invariants. In *Proc. of CAV 2005* (Edinburgh, UK, July 6-10, 2005). Springer 2005, 170-184.

[19] Kishi, T., and Noda, N. Formal Verification and Software Product Lines. *CACM, 49,* 12 (Dec. 2006). 73-77.

[20] Konrad, S., and Cheng, B. H. C. Facilitating the Construction of Specification Pattern-based Properties. In *Proc. of RE'05* (Paris, France, Aug. 29-Sept.2, 2005). IEEE Computer Society, 329-338.

[21] Kurshan, R. P. Evolution of Model Checking into the EDA Industry. In *Proc. of the Second Int'l Conf. on Automated Technology for Verification and Analysis (ATVA)* (Taipei, ROC, Oct.31-Nov.4, 2004). Springer, 2-6.

[22] Li, H., Krishnamurthi, S., and Fisler, K. Modular Verification of Open Features Using Three-Valued Model Checking. *Automated SW Eng., 12,* 3 (July 2005), 349-382.

[23] Liu, J., Dehlinger, J., and Lutz, R. R. Safety Analysis of Software Product Lines Using State-Based Modeling. In *Proc. of ISSRE 2005* (Chicago, USA, Nov. 8-11, 2005). IEEE Computer Society 2005, 21-30.

[24] Liu, J., Dehlinger, J., and Lutz, R. R. Safety Analysis of Software Product Lines Using State-Based Modeling. *Journal of Systems and Software*. To Appear. http://dx.doi.org/10.1016/j.jss.2007.01.047.

[25] Liu, J., Dehlinger, J., Sun, H., and Lutz, R. State-Based Modeling to Support the Evolution and Maintenance of Safety-Critical Software Product Lines. In *Proc. of the 5th Workshop on Model-Based Development for Computer-Based Systems (MBD'07)* (Tucson, USA, March 29, 2007). IEEE Computer Society, to appear.

[26] Loer, K., and Harrison, M. D. An integrated framework for the analysis of dependable interactive systems (IFADIS): Its tool support and evaluation. *Automated Software Engineering, 13,* 4 (Oct. 2006), 469-496.

[27] McMillan, K. *Symbolic Model Checking*. Kluwer Academic Publishers, Boston, MA, 1993.

[28] Model Checking Group at CMU, 2006. The SMV System. http://www.cs.cmu.edu/~modelcheck/smv.html.

[29] Mondragon, O., Gates, A. Q., and Roach, S. Prospec: Support for Elicitation and Formal Specification of Software Properties. In *Proc. of the 3rd Workshop on Runtime Verification (RV'2003)* (Boulder, Colorado, July 14, 2003). Elsevier, 1-22.

[30] Northrop, L., and Mcgregor, J. Components As Products. *News at SEI, 6,* 1 (First Quarter 2003), www.sei.cmu.edu/news-at-sei/columns/software-product-line/2003/1q03/software-product-lines-1q03.htm.

[31] Reps, T. W., and Teitelbaum, T. *The Synthesizer Generator: a system for constructing language-based editors*. Springer-Verlag, 1989.

[32] Robby, Dywer, M. B., and Hatcliff, J. Bogor: A Flexible Framework for Creating Software Model Checkers. In *Proc. of Testing: Academia & Industry Conf. – Practice And Research Techniques (TAIC PART)* (Windsor, United Kingdom, Aug. 29-31, 2006), 3-22.

[33] Rushby, J. *Formal Methods and their Role in the Certification of Critical Systems.* Technical Report SRI-CSL-95-1, Computer Science Laboratory, SRI International, Menlo Park, CA, 1995.

[34] Smith, R. L. , Avrunin, G. S., Clarke, L. A., and Osterweil, L. J. Propel: an approach supporting property elucidation. In *Proc. of ICSE 2002* (Orlando, USA, May 19-25, 2002). ACM Press, 11-21.

[35] Smith, M. H., Holzmann, G.J., and Etessami, K. Events and Constraints a graphical editor for capturing logic properties of programs. In *Proc. of RE'01* (Toronto, Canada, Aug.21-31, 2001). IEEE Computer Society, 14-22.

[36] Van Langenhove, S., and Hoogewijs, S. Integrating Cadence SMV in the Verification of UML Software. In *Proc. of the 8th Dutch Proof Tools Day* (Nijmegen, The Netherlands, July 2004), 15-29.

[37] Weiss, D. M., and Lai, C. T. R. *Software Product Line Engineering: A Family-Based Software Development Process*. Addison-Wesley, 1999.