Tool-Supported Verification of Product Line Requirements

Prasanna Padmanabhan
Citrix Systems, Inc.
Fort Lauderdale, Florida, 33309
prasanna.padmanabhan@citrix.com

Robyn R. Lutz[1]
Department of Computer Science
Iowa State University, and
Jet Propulsion Laboratory
rlutz@cs.iastate.edu

Abstract. A recurring difficulty for organizations that employ a product-line approach to development is that when a new product is added to an existing product line, there is currently no automated way to verify the completeness and consistency of the new product's requirements in terms of the product line. In this paper we address the issue of requirements verification for product lines. We have implemented our approach in a requirements engineering tool called DECIMAL (DECIsion Modeling AppLication). DECIMAL is a requirements verification tool with a rich graphical user interface that automatically checks for completeness and consistency between a new product and the product line to which it belongs. The verification uses a SQL database server as the underlying analysis engine. The paper describes the tool and evaluates it in two applications: a virtual-reality, positional device-driver product line and the feature-interaction resolution problem.

1. Introduction

Many software organizations use product lines to encourage reuse, enhance quality, and reduce development time. A recurring difficulty for these organizations is that when a new product is added to an existing product line, there is currently no automated way to verify whether the new product's specific requirements are within the reuse constraints of its product line (Jaring and Bosch, 2002). In this paper we address the issue of requirements verification for product lines. We have implemented our approach in a requirements engineering tool called DECIMAL (DECIsion Modeling AppLication). DECIMAL is an interactive, GUI-driven requirements verification tool that automatically checks for completeness and consistency between a new product and the product line to which it belongs. The verification uses a database as the underlying analysis engine.

[1] The second author's research is supported in part by National Science Foundation Grants CCR-0204139 and CCR-0205588. An early version of this paper was presented at REPL'02 (Padmanabhan and Lutz, 2002).

Requirements verification for a new product in the product line addresses both consistency and completeness. Consistency of a product-line member refers to whether its specification is self-contradictory (Ghezzi et al., 2002), i.e., ensures that "no subset of requirements for this instance conflict" (Doerr, 2002). Completeness of a product-line member refers to whether the specification documents all the needed requirements for that system (Ghezzi et al., 2002), i.e., ensures that "everything the instance is supposed to do is included in the requirements" (Doerr, 2002).

Verifying the consistency and completeness of a product-line member also entails checking that the intended inter-relationships among the selected features are maintained in the new system. In product lines, these relationships are often dependency constraints. A dependency constraint occurs when one decision narrows the choices that the developer has for another decision. In other words, a dependency constraint exists when there is a dependency among product-line variabilities (such as choices of features). For example, in the virtual-reality application described later, the number of buffers must equal the number of sensors for each system in the product line. Thus, building a new product with three sensors requires a choice of three buffers.

The main contribution of DECIMAL is that it automatically checks that dependency relationships among the values of the variabilities (where the value chosen for one variability dictates or constrains the value that another variability can take) are maintained in the new system being specified. In addition, DECIMAL automates completeness, consistency, range, and type checks. Since dependencies are often poorly documented, difficult to verify manually, and prone to be lost in long-lived systems with turnover of experts, automation can support improved requirements verification.

The DECIMAL tool is designed to allow flexibility in decision models. A decision model is a description of the decisions that must be made to build a new product and of the order in which those decisions are to be made. Many decision models enforce a particular ordering of decisions (see discussion below), which is sometimes contrary to the developers' preferences. DECIMAL allows decisions to be made in any order, making it compatible with a broader range of industrial development processes.

The remainder of the paper is organized as follows. Section 2 describes related work in product line requirements verification and tools. Section 3 presents the approach used and its implementation in the tool, with examples drawn from a standard product family (the Floating Weather Station). Section 4 demonstrates the use of DECIMAL on an actual product line of virtual-reality device drivers. Section 5 provides an additional evaluation of DECIMAL's constraint checking capabilities by describing its uses and limits in modeling telephony feature interaction resolutions. Section 6 discusses the envisioned usage of the tool in terms of process, scalability, effectiveness, and future work. Section 7 provides some concluding remarks.

2. Related Work

The need for better and more-automated requirements verification of new product-line members has been widely described. For example, a recent paper by Jaring and Bosch identifies the lack of tool support to identify variability information and the lack of explicit representation of dependencies among variabilities as problematic (Jaring and Bosch, 2002). Similarly, a lack of tool support was cited as an ongoing concern by the Fourth Product Line Engineering Workshop (Bass, Clements and Kazman, 1999) and by Zave (Zave, 2001). More recently, Doerr (2002) has cataloged the many advantages of explicitly modeling relationships among the product-line features and described the importance of thorough completeness and consistency checks.

With regard to definitions, we follow Clements and Northrop in describing a software product line as "a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way" (Clements and Northrop, 2002). [2]

DECIMAL's approach to requirements engineering for product lines is consistent with the FAST (Family Abstraction and Translation Technique) model described by Ardis et al. (Ardis, et al., 2000), and Weiss and Lai (Weiss and Lai, 1999). FAST distinguishes between the *Domain Engineering* phase where the product family requirements are defined (the investment phase) and the *Application Engineering* phase where the family members are produced (the payback phase). The application-engineering environment is used to help build an individual member from product family requirements specified in the domain engineering process. DECIMAL is such a tool. It is useful for both Domain Engineering and Application Engineering as seen in Fig. 1.

[2] A closely related concept to the product line is the product family. A product family is a set of products built from a common set of core assets. Most product lines (including the ones described in this paper) are built as product families.

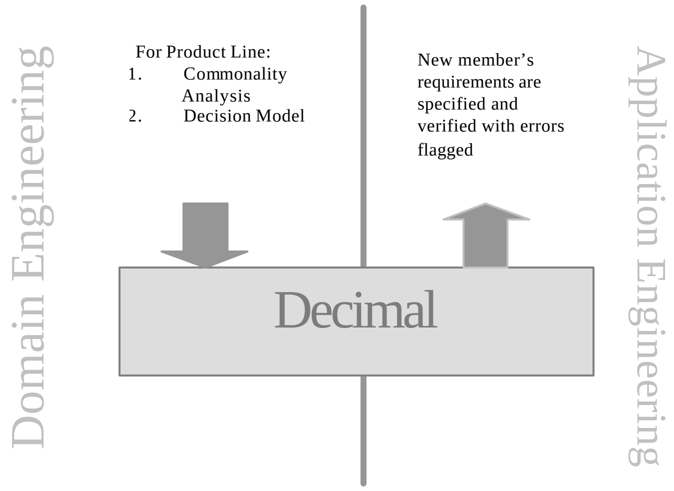


Figure 1. DECIMAL is useful in both Domain Engineering and Application Engineering.

A key asset produced in the Domain Engineering phase is the decision model, which identifies how the individual systems in the product line can vary and in what order the decisions regarding these differences should be instantiated. There are several alternative approaches to the representation of the decision model for a product line. DECIMAL uses a tabular decision model (as do FAST, PuLSE, and KobrA) to describe variations (Doerr, 2002; Atkinson et al., 2002). The tabular decision model has the advantages of readability, lack of ambiguity, and a flexible level of detail.

Lam uses an alternative approach, organizing the decision model as a tree structure (Lam, 1998). The top-level nodes of the tree are functional areas that are common to all the members of the product line. The lower-level nodes are organized as a variability tree with each branch representing choices. Producing a new member of the product line then involves traversing the variability tree via selection (selecting one of the functional areas relevant to the system) and instantiation (instantiating the template core-requirement associated with the functional area selected) decisions at every node of the tree, starting from the root and working towards the leaves. A disadvantage of this approach is that an entire subtree may need to be duplicated when a new option for a variability is added, limiting the scalability of the approach. Lutz has also pointed out that several choices could exist among variability trees, often with no compelling reason to select one over another (Lutz, 2000).

Gomaa describes a domain model developed using either a kernel-first approach (capturing features that are common to all members of the domain) or a view- integration approach where multiple viewpoints are integrated to create the domain model (Gomaa, 1995). The individual members of the product-family are then generated by tailoring the domain model given the desired features in the target system. He recommends the approach for relatively stable, well-understood applications.

Lee et al. take a feature-based approach to the decision model, organizing it according to four feature types: application capabilities, operating environments, domain technologies, and implementation techniques (Lee, et al., 2000). Variabilities are features that can only be present or absent for any particular system in the product line. In addition, relationships among the features are restricted to mutual exclusion, mutual dependency, or hierarchical relationships (e.g., composition). These relationship types are not rich enough to represent the dependencies in many actual product lines, especially if they are not feature-based. The virtual-reality device driver systems described here form such a product line.

Previous approaches to decision modeling are hierarchical in nature and therefore introduce orderings of decisions that can complicate constraint checking. The partial ordering of decisions in FAST, for example, could potentially violate dependencies among variabilities. Suppose that decisions regarding the values of three variabilities, V1, V2 and V3, are ordered such that values for V1 and V2 must be selected before selecting values for V3. In that case, a dependency of the form, "if you select value 'a' for V3, then you must select a value 'b' for V1" can potentially violate the constraint. Selecting the value 'c' for V1 and then 'a' for V3 can force a change in the decision made earlier regarding the value of V1 in order to satisfy the dependency constraint. Similarly, decision models in which all decisions at level *l* are made before decisions at level *l+1* can cause problems.

Such ordering of decisions tends to be unduly restrictive on program developers who create new programs by modifying old programs (Parnas, 1976) and makes requirements evolution more burdensome to verify. DECIMAL overcomes some of these problems. It does not enforce an ordering of decisions, but allows users the freedom to make decisions in any order (including partial or total ordering) consistent with the organization's or project's development process.

DECIMAL also allows requirements evolution, a constant factor in large projects, without a resulting exponential growth in memory. Addition of a new variability or new commonality causes only a linear increase in storage space. DECIMAL, like FAST and unlike several of the other decision models, also supports multiple types of variabilities (e.g., Integer and

Floating Point) rather than just Boolean and Enumerated. DECIMAL's flexibility was designed to help bridge the gap between the variety of ways in which companies actually develop product lines and the prescriptive nature of many decision modeling approaches.

3. Approach

In this section, we describe the approach used to perform the analysis and the implementation of DECIMAL. We describe the application of DECIMAL in both the domain-engineering (defining the product line) and the application-engineering (developing each new product in the line) phases. Product-line-wide requirements are specified in the domain-engineering phase. The artifacts produced by the domain-engineering phase – i.e., the commonality analysis and the decision model – serve as the input for the application-engineering phase, which verifies the consistency, completeness, and range and type correctness of the requirements for a new member. DECIMAL has a rich graphical user interface with wizards to drive the user through both the specification and the analysis process. Although DECIMAL uses SQL database queries to perform the actual analysis, its ease-of-use and intuitive user-interface hide the complexity of the database engine underneath. The end-user is not expected to be familiar with database technologies and can instead concentrate on the actual requirements analysis process.

A version of the Floating Weather Station (FWS) product family (Weiss and Lai, 1999) provides examples. We have extended the FWS example slightly with some additional features and feature dependencies to better illustrate DECIMAL's capabilities. Floating Weather Stations are buoys that float at sea. They are equipped with sensors that monitor wind speed. Each FWS has an onboard computer to store a recent history of wind-speed data and a radio transmitter to transmit the wind speed at periodic intervals. They can be configured in various ways, including different types and numbers of wind-speed sensors, and different lengths of time covered by the history of wind-speed readings that they maintain.

3.1. Domain Engineering Activities

The key domain engineering activities (Weis and Lai, 1999) are:

1. the Commonality Analysis, which identifies:

- the assumptions that characterize both what is common to all members of the family (called *Commonalities*), and how the family members may vary from one another (called *Variabilities),* and

- a set of constraints (often dependency relationships among variabilities) specified in predicate logic. The constraints must be satisfied by every system in the family.

2. the Decision Model, which describes:
   - the decisions that must be made, and
   - the order in which they are made, to produce a new system in the family. The variabilities are parameterized (called *parameters of variability*) to represent the decision(s) represented by the variability, the range of allowable values, the time at which the value is fixed (e.g., bound at compile time), and the default value.

Table 1 shows some parameters of variation for the FWS product line. (See (Padmanabhan, 2002) for a fuller treatment of the commonality analysis).

Parameter	Meaning	Value Space	Binding Time	Default
P5: MaxSensorPeriod	Maximum sensor period	[1..600]	Translator Construction	600
P6: MaxSensors	Maximum number of sensors on board an FWS	[2..20]	Translator Construction	20
P7: MaxTransmitPeriod	Maximum transmission period	[1..600]	Translator Construction	600
P8: MinLow	Minimum number of low-resolution sensors	[2..MaxSensors-2]	Translator Construction	2
P11: SensorPeriod	Sensor Period	[1..MaxSensorPeriod] sec	Specification	5
P12: SensorRes	The resolution of each sensor	For each sensor, one value in {LOWRES, HIGHRES}	Specification	LOWRES
P13: TransmitPeriod	Transmit Period	[1..MaxTransmitPeriod]	Specification	10

Table 1. Parameters of variation for the Floating Weather Station product line (excerpts)

Dependency constraints among the variabilities are also specified in DECIMAL. Some examples of constraints for the FWS are:

- The minimum number of low resolution sensors must be between 2 and 2 less than the maximum number of sensors: (MinLow >= 2) and (MinLow <= MaxSensors – 2)
- The sensor period must be less than the maximum sensor period of the sensors: SensorPeriod <= MaxSensorPeriod
- High Resolution Sensors must have a sensor period that is at least half the maximum sensor period: (SensorRes = ‘HIGHRES’) => (SensorPeriod > MaxSensorPeriod/2)

These constraints will be automatically checked in DECIMAL for every new FWS that is built in the product line. Our implementation of constraint checking is similar to the approach followed by Feather in that a database server functions as the backend analysis engine to perform the checks (Feather, 1998).

3.2. Application Engineering Activities

DECIMAL stores the domain model data that the user inputs in a backend SQL Server database. Four tables – a Commonalities Table, a Variabilities Table, a Constraints Table, and a Specific Products Table (requirements specific to individual product line members) are stored. Table 2 shows the Commonalities Table.

Fno	*Cname*	Cdesc
1	cTransmit	At fixed intervals, the FWS transmits messages containing and approximation of the current *wind speed* at its location.
1	cWeightedAvg	The wind speed value transmitted is calculated as a *weighted average* of the sensor readings, calculated over several readings for each sensor.
1	cSoftwareDriver	Each sensor comes equipped with a software driver for it and a unique identifier.
1	cRelySens	Each sensor on board an FWS has a way to indicate its reliability.

Table 2. Commonalities table

Table 3 shows a Specific Products Table for a FWS product family consisting of three members, FWS1001-1003.

F. No.	*Fmemname*	MaxSensors	MinLow	MaxSensorPeriod	SensorRes	SensorPeriod	cTransmit	cWeightedAvg
1	FWS1001	8	4	6	HIGHRES	3	Yes	Yes
1	FWS1002	4	3	3	HIGHRES	2	Yes	Yes
1	FWS1003	2	2	2	LOWRES	2	Yes	No

Table 3. Specific Products table

Constraints in the Constraints Table (Table 4) are stored as database queries. In order for the database management system to perform the analysis of the constraints specified in predicate logic, these constraints must be converted to a language that the database server understands. This language is called Transact Structured Query Language (T-SQL, for short). The syntax of a simple T-SQL SELECT query used to extract data from a database table is

SELECT <*column_list*> FROM <*tablename*> WHERE <*condition*>

This statement retrieves those records from the table where <*condition*> holds true. Only those columns specified by <*column_list*> are retrieved. To determine whether all existing members, including a new member being added, satisfy an assertion *P* or not, we check for any member in the product line such that *not P* is true. The corresponding T-SQL statement to check this would be: SELECT * FROM FWS WHERE *not P*. (Note that "*" is a wildcard meaning "All Columns" in this example, and "FWS" is the name of Table 3.)

Fno	Frule	Frulesql
1	MinLow >= 2	SELECT * FROM FWS WHERE not (MinLow >= 2)
1	TransmitPeriod <= MaxTransmitPeriod	SELECT * FROM FWS WHERE not (TransmitPeriod <= MaxTransmitPeriod)
1	SensorRes = 'HIGHRES' => SensorPeriod > MaxSensorPeriod/2	SELECT * FROM FWS WHERE not (not(SensorRes = 'HIGHRES') OR (SensorPeriod > MaxSensorPeriod/2))

Table 4. Constraints table

3.2.1 Consistency checking

To detect inconsistencies, the SQL query representations of the constraints in the Constraints Table (Table 4) are executed one by one. The rows returned by executing a query indicate the family members that are inconsistent. For example, the constraint "SensorRes = 'HIGHRES' => SensorPeriod > MaxSensorPeriod/2" is cast into the following T-SQL query.

```
SELECT * FROM FWS
WHERE not (not(SensorRes = 'HIGHRES')  OR (SensorPeriod > MaxSensorPeriod/2))
```

1	FWS1001	8	4	6	HIGHRES	3	Yes	Yes

Executing this query returns one row, indicating that there is an inconsistency in FWS1001, namely that the sensor period for the high resolution sensor is less than half the maximum sensor period in that system.

DECIMAL does not yet check to see if the constraints themselves are consistent. For example, the following rules are inconsistent: V1 = 5 => V2 = False; V2 = False => V1 < 3. In the future, we want to extend DECIMAL's capability to detect such problems before checking any new members.

3.2.2. Completeness checking

Detecting incompleteness (whether there exists a product-line member that did not satisfy a particular commonality), is straightforward in DECIMAL. For example, the statement SELECT * FROM FWS WHERE cWeightedAvg = 'No' returns one record.

1	FWS1003	2	2	2	LOWRES	2	Yes	No

This means that the system member FWS1003 is not complete with respect to the commonality cWeightedAvg, i.e., that its specification is currently missing a requirement specified as common to all systems in the product line.

Sometimes commonalities are unintentionally omitted from a new member. The completeness checking serves as a "checklist" for the application engineer to ensure that no commonality is inadvertently left out of the application engineering artifacts. DECIMAL does not enforce commonalities but instead supports verification that all the requirements labeled as commonalities are planned capabilities of the new system.

3.2.3. Range and type correctness checking

DECIMAL also performs range and type correctness checking to verify that the values of variabilities selected for the new member fall in the range and are of the same data type as specified for the product line. This functionality is programmatically built into DECIMAL's front-end. For example, MaxTransmitPeriod must be an integer in the range 1-600. If a new member of the product line is being constructed with the value of the variability equal to 620.5, then DECIMAL will flag both an out-of-range and an incorrect data type in error messages. The developer can then correct the error(s) and re-run DECIMAL to check the value.

The tool is used at specification time, when most bindings occur. The values of variabilities instantiated later (e.g., at run-time), for which future values are not currently known, assume the default values for their parameters of variability. The tool does not provide run-time checks.

3.2.4. Handling of near commonalities in DECIMAL

As described above, a near commonality is a commonality that is true for almost all family members. This occurs when an existing product-line commonality is intentionally not built into a new member of the system. Near commonalities often occur when a new system in the product line is descoped (e.g., due to budget or schedule constraints), when a next-generation prototype is built, or when a testbed is constructed without certain features common to the rest of the systems in the product line.

Near commonalities can be handled either as variabilities or as constrained commonalities which are invariant over the domain. In order to represent a near-commonality as a variability, a new Boolean parameter of variation is added. This parameter of variation is "true" for the majority of members that satisfy all commonalities and "false" for the few that don't. An alternative is to represent a near-commonality (NC) as a commonality of the form "If not member i, then NC." Such statements are constrained commonalities which are invariant over the domain.

For example, if all the members of the FWS product line except the baseline product calculate a weighted averaging of measurements, whereas the baseline product calculates an unweighted averaging, this near commonality can be represented as a variability ("The average calculated by a FWS may be weighted") with an associated Boolean parameter of variation (IsWeighted) that is true for all members except the baseline FWS. Alternatively, the near commonality can be represented as a commonality ("All members except the baseline use weighted averaging"). The latter representation provides more information in that it is immediately clear that all but one instantiation of the product line require weighted averaging. However, with this representation, if a second system is then built without weighted averaging, the product-line commonality must be updated. Representing a near commonality as a variability is thus preferable if it is possible that additional systems will be built without that feature, since no update to the product-line specification needs to occur in that case.

DECIMAL supports the representation and checking of near commonalities by allowing a commonality to be designated as a near-commonality either during domain engineering (when the specification is created) or later during application engineering when it turns out that a commonality will intentionally not be satisfied by the new member. For example, the weighted average could be explicitly specified as a near-commonality When a requirement is specified as a near commonality ("NC") in the Commonality Table, the automated completeness checking in DECIMAL will not require that the near commonality be present in the member being checked (e.g., the baseline FWS). This expressiveness comes at a cost, which is a weakness of the current tool. If a new member is subsequently built that requires weighted averaging, the specification must be changed back from a near commonality to a commonality in order for DECIMAL to include it in future automated checks. Note, too, that if the product-line architecture relies on the commonality, the architectural consequences must be carefully weighed against the advantages of making it a near-commonality.

4. Case Study

DECIMAL was used to specify and analyze a product line of virtual-reality positional device drivers, developed as part of the VR Juggler project. VR Juggler is an active research project headed by Cruz-Neira at Iowa State University's Virtual Reality Applications Center (VRAC). VR Juggler is an open-source, virtual-reality, application-development framework used by companies such as John Deere and Boeing to validate requirements and designs.

VR Juggler provides an application framework and set of C++ classes for writing virtual reality applications (Just, et al., 1998). It supports a wide variety of input devices to read external data-positional devices (such as motion trackers in 3D space), analog devices (such as pedals, steering wheels and joysticks), digital devices (such as wands and mouse buttons), and gloves (such as the CyberGlove™). Fig. 2 shows a class hierarchy for the device drivers. The four positional device drivers for digital, positional, analog, and glove devices form the software product line that we specified and analyzed with DECIMAL.

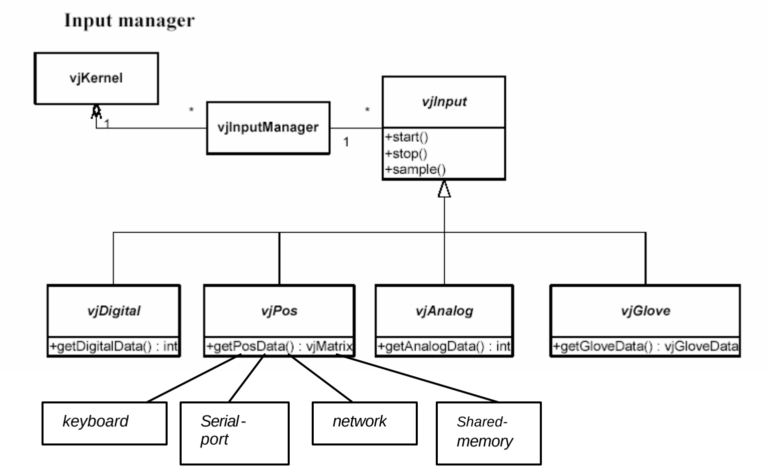


Figure 2. VR Juggler device driver class hierarchy. Reproduced with permission from (Bierbaum, 2000).

The commonality analysis of the positional device driver product line extended the FAST method with the representation of dependencies among variabilities, a topic of particular concern in the VRJuggler domain due to the growing number of novice VRJuggler users. The product line was specified with help from VRAC domain experts. They reviewed our preliminary requirements specification and provided corrections and additional domain information. Excerpts from the commonality analysis are given below with particular emphasis on the variabilities and the interdependencies among their values.

Dictionary of Terms

- *Device Sampling Thread*- The microkernel is implemented as a separate thread that executes periodically in a loop. This thread spawns two other threads- a display thread and a *device sampling thread*, which is owned by an instance of the input device.
- *4X4 Matrix* - The matrices used to transform points in 3D space are of size 4X4 with each of the rows representing a homogenous vector (x, y, z, w).

$$\begin{pmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{pmatrix}$$

d, *h* and *l* are used for transforming world coordinates into screen coordinates for output on a 2D monitor while *m*, *n*, and *o* are used for meant for shearing and perspective viewing.

Commonalities

The following statements are some of the basic assumptions about the Positional Device Driver domain. They are true of all Positional Device Drivers.

- C1 (cMatrix). All positional device drivers get positional data as a 4X4 matrix.
- C2 (cConfig). All the configuration functions in all drivers request the following data: port data, baud rate and instance name.
- C3 (cInherit). All drivers inherit the basic base class (vjInput) methods to start sampling, stop sampling and update devices.

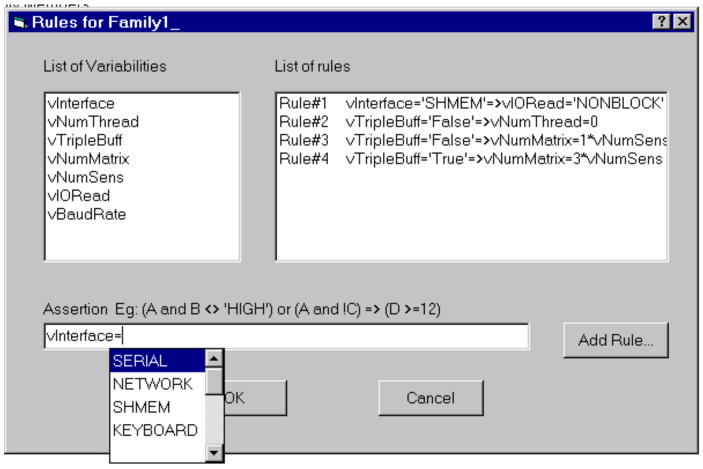
Variabilities

The following statements describe how the Positional Device Drivers can vary:

- V1 (vInterface). The *interface* for reading input data can vary. It can be one of: serial port, network, keyboard, or shared memory. It is an *enumerated* type of variability.
- V2 (vNumThread). The *number of device sampling threads* can vary. It can be one of *{0, 1, 2}*. It is an *integer* type of variability.
- V3 (VNumMatrix). The *number of 4X4 matrices* into which the positional data from the sensors is stored can vary from 1-12. It is an integer type of variability.
- V4 (vBaudRate). The *Baud Rate* of sensors can vary from 1 to 115200 as follows: *{150, 200, 300, 600, 1200, 1800, 2400, 4800, 9600, 19200, 38400, 57600, 76800, 115200}*. It is an enumerated type of variability.
- V5 (vTripleBuff). The driver can use or not use *triple buffering algorithm: {True, False}*. It is a *Boolean* type of variability.
- V6 (vNumSens). *NumSensors* (the number of sensors from which data can be read) can be one of *{1 to 108}*. It is an *integer* type of variability.
- V7 (vIORead). The type of *I/O read* can be either Blocking or Non-Blocking. It is an enumerated type of variability. In a blocking call, the thread blocks (waits) at the read system call until there is data available to read. In a non-blocking system call, the thread performs other activities until there is data available.

Examples of Dependencies among Variabilities

A dependency exists when the choice of a value for one variability constrains the choice of value for another variability. The following are examples of such dependencies in the VRJuggler positional device drivers product line. Fig. 3 shows the dialog box provided by DECIMAL to add rules to the set of assertions to be automatically checked for the product line.



```
vInterface
vNumThread
vTripleBuff
vNumMatrix
vNumSens
vIORead
vBaudRate
```

```
Rule#1   vInterface='SHMEM'=>vIORead='NONBLOCK'
Rule#2   vTripleBuff='False'=>vNumThread=0
Rule#3   vTripleBuff='False'=>vNumMatrix=1*vNumSens
Rule#4   vTripleBuff='True'=>vNumMatrix=3*vNumSens
```

```
vInterface=
```

Figure 3. Entering dependency constraints in DECIMAL

- D1: Drivers that read from shared memory do not use a blocking call (Instead, they poll periodically to see if data is available in the shared memory pool): vInterface = “shared memory’ => vIORead = ‘Non-Blocking’
- D2: Devices that do not use triple-buffering do not require device sampling threads to read input data: vTripleBuff = ‘False’ => vNumThread = 0
- D3: If a triple buffering algorithm is not used, then the number of 4X4 matrices into which data is read equals the number of sensors. (Each sensor reads in data into just one matrix): vTripleBuff = ‘False’ => (vNumMatrix = 1 * vNumSens)
- D4: If a triple buffering algorithm is used, then the number of 4X4 matrices into which data is read is three times the number of sensors (Each sensor reads in data into three matrices): vTripleBuff = ‘True’ => (vNumMatrix = 3 * vNumSens)

Parameters of Variation

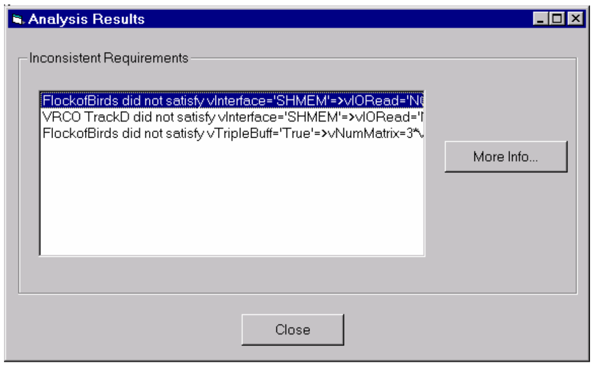
Table 5 shows some of the parameters of variation and their relationships to the variabilities for the VRJuggler positional device drivers product line.

Parameter	Meaning	Value Space	Binding Time	Default
vInterface	Type of interface for reading input data.	{SERIAL, NETWORK, SHMEM, KEYBOARD}	Specification	SERIAL
vNumThread	The number of threads to sample device data.	[0..2]	Specification	0
vNumMatrix	The number of 4x4 matrices to store the positional data read from the sensors.	[1..12]	Specification	4
vBaudRate	The baud rate of the external input device. Allowable range may differ between manufacturers.	*{150, 200, 300, 600, 1200, 1800, 2400, 4800, 9600, 19200, 38400, 57600, 76800, 115200}*	Compilation	38400
vTripleBuff	Whether or not a triple buffering algorithm is used to read and store input data.	True, False	Specification	True
vNumSens	The number of sensors	[0..108]	Specification	0
vIOread	Type of system call used to read input data from the device	{BLOCK, NONBLOCK}	Specification	BLOCK

Table 5. Parameters of variation

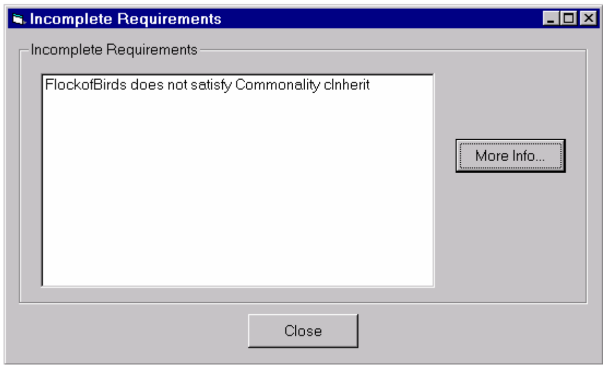
We next explain the analysis process and its results. For illustration purposes, only two potential members of the product line are shown: Flock of Birds (Ascension) and VRCO TrackDaemon (VRCO).

The user invokes the consistency checker over the members of the product line by selecting the product line name and clicking on the appropriate button in the main toolbar. Fig. 4 shows the result of the consistency analysis. FlockofBirds does not satisfy the first or fourth rules (dependency constraints D1 and D4 above). VRCO TrackD does not satisfy the first rule (D1). We found that the automated constraint checking useful for quickly checking alternate sets of requirements (similar to rapid prototyping).



```
FlockofBirds did not satisfy vInterface='SHMEM'=>vIORead='N
VRCO TrackD did not satisfy vInterface='SHMEM'=>vIORead='
FlockofBirds did not satisfy vTripleBuff='True'=>vNumMatrix=3*v
```

Figure 4. Consistency checking



```
FlockofBirds does not satisfy Commonality cInherit
```

Figure 5. Completeness checking

The user invokes the completeness checker by selecting the product-line name and clicking on the appropriate button in the main toolbar. Fig. 5 shows the result of the completeness analysis. Since FlockofBirds does not satisfy one commonality (C3 above), it is flagged by DECIMAL as not being a full family member for the application engineer's attention.

Range and type checking are also invoked by selecting the appropriate button in the main toolbar. DECIMAL flags any variabilities that were assigned values not in the range or type specified for the product line. For example, DECIMAL flagged vNumThread, which initially had a value of "4" for FlockofBirds, as not passing the range checker.

It is worth noting that several of the common requirements for the positional device drivers relate to class inheritance. The positional device drivers were not explicitly considered as a product line by the developers, but they are in fact a set of similar systems with shared assets, i.e., a product line. An important consequence of this analysis was that VRAC has become aware of the device drivers as a product line with the requirements engineering products and analyses as reusable assets. DECIMAL's support for verification of dependency constraints will be especially useful in the future, since new positional device drivers will be regularly added to the existing set in response to the rapid evolution of this virtual-reality technology.

5. Feature Interaction Resolution

In this section, we describe an evaluation of how well DECIMAL can be used to model the feature interaction problem. Feature interaction resolution has been a recurring and difficult problem for requirements engineers of telecommunications switch product lines. This problem was selected to exercise DECIMAL and to more clearly distinguish what it can and cannot do.

Telecommunication switches can be modeled as a product line. All switches in such a product line offer a common set of basic *services* (stand-alone functionalities) which are commonalities, as well as optional *features* that provide additional functionality to an existing feature or service. A *feature interaction* occurs when one feature affects the behavior of another feature. Feature interactions are often undesirable in that they may violate a requirement (Li et al., 2002), and can sometimes lead to non-deterministic behavior. With feature interactions, situations may occur that are inconsistent with intended behavior or that are unexpected from the user's point of view.

A simple example of feature interaction occurs between telephony features "Originator Call Screening" (OCS) and "Call Forwarding" (CF). Suppose that, according to OCS, all calls from a subscriber, *A*, to a particular person, *X*, are forbidden. However, if *A* calls *B* and *B* forwards the call to *X*, then the overall system behavior is undesirable.

The approach we used was to model basic services as commonalities and features as variabilities of a telecommunications product line and then to investigate whether restrictions on feature interactions can be represented as constraints in DECIMAL. Table 6 lists resolution techniques for feature interactions compiled from (Griffeth and Velthuijsen, 1994) and (Homayoon and Singh, 1988) in the left-hand column. The right-hand column of the table shows the DECIMAL rule used to model each constraint on the feature interactions.

For each feature, there is an associated Boolean variability which indicates whether or not to include that feature. For example, Vcid = "True" means that Caller ID feature should be included for that member. Vsig_SC and Vsig_3WC were new variabilities that had to be introduced to represent the constraints. These variabilities are of enumerated type and represent the signal to which the *Service Code* (SC) and *Three Way Call* (3WC) features respond.

Additional variabilities Ven911_prio and Vdcw_prio were also introduced to represent the priority of the associated feature (namely *911* and *Delayed Call Waiting,* respectively).

Feature Interaction Resolution Policy **[Griffeth et al] [Singh et al]**	**Representation in Decimal of the feature arbitration policies**
Mutual Exclusion: e.g., CFU and CFB (CFU-Call Forward Unconditional, CFB – Call Forward Blocking)	(Vcfu = "True" => Vcfb = "False") and (Vcfb = "True" => Vcfu = "False")
Dependency Between Features: e.g**.**, ACB and CID (ACB – Automatic Call Back, CID - Caller ID)	Vacb = "True" => Vcid = "True"
Conditional Dependency Between Features: e.g**.**, CFU, TCS, mCID (TCS – Terminator Call Screening, mCID – a modified Caller ID informing a subscriber the origin of a call)	(V2 = "True" => V1 = "True") => V3 = "True"
Signal Conflicts Between Features: e.g**.**, 3WC and Service Code	(Vsig_3WC = "flashhook" ^ Vsig_SC = "flashhook") => ((V3WC = "True" => Vsig_SC = "False") ^ (Vsig_SC = "True" => V3WC = "False"))
Time precedence among features: e.g., CW and CFB (CW – Call Waiting)	Cannot be represented in Decimal
Dependency on Complexity of Features: e.g., CW and 3WC	Cannot be represented in Decimal
Precedence to Special Features: e.g., Enhanced 911	e.g., Ven911_prio = "HIGH" ^ Vdcw_prio <> "HIGH" => ((Ven911 = "True") => Vdcw ="False")

Table 6. Feature-interaction resolution in DECIMAL.

The table shows how features such as mutual exclusion, dependency and conditional dependency can successfully be represented as constraints in DECIMAL without requiring any additional variabilities. For example, mutual exclusion between CFU (Call Forwarding

Unconditional) and CFB (Call Forward Blocking) can be represented as a simple rule involving Vcfu and Vcfb.

Signaling conflicts (when two features react to the same signal, such as flash-hook signal) and precedence for special features (e.g., that 911 calls have the highest precedence) can also be represented in DECIMAL, but require additional variabilities. Time precedence among features (e.g., that feature A has precedence over feature B for the first call alone, but that for subsequent calls feature B has precedence over feature A) cannot be represented in DECIMAL. Resolving interactions at run time requires a notion of state, which is not part of DECIMAL's requirements-based modeling.

In summary, DECIMAL can adequately capture feature interactions that are resolved at specification time (like mutual exclusion of features, conditional dependency, and signaling conflicts). Resolutions of feature interactions (i.e, dependencies among variabilities) that are enforced at run-time cannot be checked by DECIMAL. This application helped distinguish how feature interactions differ from product-line constraints in that many dependency constraints among features are handled at run-time, whereas most product-line dependencies are resolved at specification time. This need for run-time checking may limit the usefulness of product-line specification approaches for feature-intensive applications.

6. Discussion

DECIMAL is currently a stand-alone tool that supports automatic checking of a new member's requirements against the specifications of the product line. To be useful in an industrial setting, DECIMAL will need to be integrated into an end-to-end development process. DECIMAL was thus designed to allow extensions for interoperability with other product-line tools. For example, if the stored data from the domain analysis is in tabular (e.g., Excel) or XML form (based on a well known XML schema), the file can be parsed and imported into DECIMAL. Currently no industry-standard schema exists for storing product-line specifications in XML, but this can be expected to change. In particular, such a schema would allow a domain model specified in a DSL (Domain Specific Language) to be input to DECIMAL. The warning messages output from DECIMAL can similarly be readily converted to alternate formats, e.g., for inclusion in printed summary reports.

A product line specified in DECIMAL becomes a reusable asset of the product line. It is populated by domain engineers and used by application engineers to assist in verification and to explore alternatives for new members. As with other product-line assets, the DECIMAL specification for a product line should be placed under Configuration Management. This ensures that as the product line evolves the DECIMAL database (of commonalities, variabilities, and constraints) remains current.

DECIMAL's design makes it extensible to handle subfamilies. A subfamily is a subset of a family consisting of instances that share characteristics (e.g., additional commonalities or variabilities) that distinguish members of the subset from all other members of the family. For example, in the Virtual Reality device drivers, the drivers derived from the class vjInput form a product family (they share methods defined in their base class vjInput) but differ in their functions to get input data. The various positional device drivers form a subfamily because they not only share the base class' functions (vjInput and vjPos) but also have additional methods in them.

From the user's perspective, the product line requirements in DECIMAL are arranged in a hierarchical structure similar to a file system. A subfamily is treated as another product family in that it may have its own set of variabilities, commonalities, and constraints in addition to those it inherits from its parent. By providing a subfamily with its own database table, DECIMAL can perform the checks on the subfamily, but must also make sure that its parent's commonalities, variabilities, and dependency constraints are satisfied.

All the checks that DECIMAL automates can be performed manually. However, since checking dependency constraints is labor-intensive and confusing to non-experts in the domain, automation encourages more thorough verification in practice. DECIMAL supports traceability of the product-line requirements to later development artifacts both by recording changes made to the requirements and by lowering the overhead of maintaining adherence to the constraints through its automated checks.

With regard to scalability, we have not performed experiments regarding the size at which the use of DECIMAL becomes cost-effective for a project. Initial use indicates that the key factor for determining the breakeven point is the number of constraints to be checked. When the number of constraints to be checked (*numConst*) and/or the number of members to be built (*numMem*) are high, the overhead involved in populating the database (which scales linearly with the number of requirements, *numReq*) is acceptable. We hypothesize that use of DECIMAL is cost-effective when $numConst \times numMem > numReq$. The use of SQL for the tool promotes scalability by allowing growth of the database to be linear in the number of requirements (commonalities and variabilities). It is possible for the number of dependency constraints to grow exponentially but in successful product lines they seldom do, since project risk-management tools tend to throttle dependency constraints (Clements and Northrop, 2002).

7. Conclusion

This paper describes DECIMAL, a tool for the requirements specification and verification of a product line. DECIMAL provides automated checking of completeness, consistency, and range and type, between a specified product line and an envisioned new system in the product line. The most significant contribution of the tool is its capability to check that dependency relationships among the values of the variabilities are maintained in the new system being specified. Industrial users of product lines have indicated a strong need for automated tool support for product-line requirements verification. DECIMAL provides a step toward filling that need.

Acknowledgements

We thank Allen Bierbaum, Carolina Cruz-Neira and, in particular, Christopher Just, all of the Virtual Reality Applications Center, Iowa State University, for providing us with information about the VR Juggler device drivers.

References

Ardis, M., Daley, N., Hoffman, D., Siy, H., and Weiss, D. 2000. Software product lines: a case study. *Software Practice and Experience*, 825-847.

Ascension Technology Corporation. http://www.ascension-tech.com/

Atkinson, C., Bayer, J., Bunse, C., Kamsties, E., Laitenberger, O., Laqua, R., Muthig, D. Paech, B., Wust, J., and Zettel, J. 2002. *Component-based Product Line Engineering with UML*, London: Addison-Wesley.

Bass, L., Clements, P., Donohoe, P., McGregor, J., and Northrop, L. 1999. *4th Product Line Practice Workshop Report*, CMU/SEI-2000-TR-002, Software Engineering Institute, CMU.

Bierbaum, A. 2000. *VR Juggler: A Virtual Platform for Virtual Reality Application Development.* MS Thesis, Iowa State University, Ames, IA.

Clements, P. and Northrop, L. 2002. *Software Product Lines.* Boston: Addison-Wesley.

Doerr, J. 2002. *Requirements Engineering for Product Lines: Guidelines for Inspecting Domain Model Relationships*. Diploma thesis, University of Kaiserslautern.

Feather, M. S. 1998. Rapid application of lightweight formal methods for consistency analyses. *IEEE Trans. Software Eng*. 24. pp. 949-959.

Ghezzi, C., Jazayeri, M., and Mandrioli, D. 2003. *Fundamentals of Software Engineering*, 2nd ed. Upper Saddle River, NJ: Prentice-Hall.

Gomaa, H. 1995. Reusable software requirements and architectures for families of systems, *Journal of Systems and Software*, 28:189-202.

Griffeth, N. D. and Velthuijsen, H. 1994. The negotiating agents approach to runtime feature interaction resolution. *Feature Interactions in Telecommunications Systems*, ed. Bouma, L. G. and Velthuijsen, H. IOS Press, Amsterdam, 217-235.

Homayoon, S., Singh, H. 1988. Methods of addressing Interaction of Intelligent network Services with Embedded Switch Services *IEEE Communications Magazine,* 26(12): 42-70

Jaring, M. and J. Bosch. 2002. Representing variability in software product lines: a case study. *Software Product Line Conf (SPLC2),* ed. Chastek, G. LNC2379, Berlin Heidelberg: Springer Verlag, 15-36.

Just, C., Bierbaum, A., Baker, A., and Cruz-Neira, C. 1998. VR Juggler: a framework for virtual reality development. *2nd Immersive Projection Technology Workshop,* Ames, IA.

Lam, W. 1998. A case study of requirements through product families. *Annals of Software Engineering*: 253-277.

Lee, K., Kang, K. C., Chae, W. and Choi, B. W., Feature-based approach to object-oriented engineering of applications for reuse. *Software—Practice and Experience,* 30: 1025-1046.

Li, H., Krishnamurthi, S. and Fisler, K. 2002. Verifying cross-cutting features as open systems. *10th ACM SIGSOFT Symp. Foundations of Software Eng.* Charleston, SC, pp. 89-98.

Lutz, R. 2000. Extending the product family approach to support safe reuse*. Journal of Systems and Software*, 53: 207-217.

Lutz, R. and Gannod, G. 2003. Analysis of a software product line architecture: an experience report. *Journal of Systems and Software,* 66: 253-267.

Parnas, D. 1976. On the design and development of program families, *IEEE Trans Software Eng*, 2.

Padmanabhan, P. 2002. DECIMAL: A *Requirements Engineering Tool for Product Families*. M.S. thesis, Iowa State University, Ames, IA.

Padmanabhan, P. and Lutz, R. 2002. DECIMAL: a requirements engineering tool for product families. *Int'l. Workshop Requirements Eng. For Product Lines* (REPL'02), Essen, Germany, pp. 39-44.

VRCO TrackD http://www.vrco.com/products/trackd/trackd.html

Weiss, D. and Lai, C. 1999. *Software Product Line Engineering: A Family-Based Software Development Process.* Reading, MS: Addison Wesley.

Zave, P. 2001. Requirements for evolving systems: a telecommunications perspective. *5th Int'l Symp. Requirements Eng.*, pp. 2-9.