A Specification-based Data Model*

Munish Gandhi
gandhim@copper.ucs.indiana.edu

Edward L. Robertson
edrbtsn@iuvax.cs.indiana.edu

Abstract

This report presents a conceptual data model for engineered products ranging from software systems to physical objects. The presented model views the hierarchy of components that constitute a system as layers of alternating specification and implementation. If one considers the principles of abstraction and encapsulation, one can see that this viewpoint is quite natural. Abstraction implies that all implementations are implementations of *some* specification. Thus specifications may be regarded as directly "above" implementations. Encapsulation implies that implementations, at least conceptually, use specifications rather than other implementations to fulfill their goals. Thus, implementations may be regarded as directly "above" specifications.

This viewpoint has other advantages. It keeps specifications and implementations consistent with one another, models an evolving system nicely, and avoids version percolation problems naturally. It also suggests a way to separate local and global issues in system design.

*Partially supported by the Indiana Business Modernization and Technology Corporation.

Contents

List of Figures

1 Introduction

Engineered products, ranging from software systems to mechanical appliances, are built from components interacting in complex ways. While there are methods to model configurations in each discipline we do not believe there is any model unifying the different approaches. Clearly there is a need for such an approach if only to handle products which use components from differing engineering technologies. Moreover, in each technology we notice ad-hoc methods being used to model component configurations. These methods, in short, identify objects in the final product and link them with either a part-of relationship or a uses relationship. The chief disadvantage of this approach is that a product specification gets divorced from the final product itself.

In this report, we present a model in which the specification of each component of a product is closely linked to its implementation. The need for such an integration has been felt before. For example, Swartout and Balzer [SB82] argue that even though software process models view specification and implementation as successive steps, in reality they influence one another. In other words, as software evolves both specifications and implementations undergo change [1].

Certain philosohical principles have been used to guide us in developing this model. We outline them here.

1. The model should capture only the structural relationships between the components of the product being developed. Any other interactions between the components, especially those which process oriented, are outside the purview of the model.

2. The model should provide a framework that is applicable across many engineering domains. Thus, the framework should be minimal and offer the least resistance when adopted for a particular domain.

3. Specifications should be pre-eminent in the model. This would allow all major design decisions used in developing a product to be reflected in the specifications.

[1] The spiral model of Boehm [Boe88] recognizes this fact, but does so only at the macro level.

4. Changes in design should occur only under human mediation. For example, redesigning a part should not cause changes in units containing that part except under the designers control. This controlled change may be the result of some automated tool, but the model should not force this change. Thus it is the goal of the model to facilitate such process activities, not to mandate them.

The above principles led us to develop a model with the following central features.

- The development process is conceptually separated into three stages – design, configuration and manifestation.
- During design, development proceeds using alternate layers of specification and implementation. Further modifications are accomodated using the versioning mechanisms for both specifications and implementations.
- During configuration, a choice mechanism is provided to pick those versions of implementation which are most suited for assembling the system.
- During manifestation, the design for each component in the system assembly is materialized. This results in each materialized component getting an identity of its own.

In the next two sections we present the Specification-based Data Model (SBDM) using an E-R diagram. Section 2 reveals the entities and relationships needed to manage an evolving system design. Section 2.1 presents the specification entity, section 2.2 presents the implementation entity, and section 2.3 discusses how a system may be designed using the relationships between these entities. Finally, section 2.4 illustrates the above concepts by modeling a CAD application example, a product database example, and an object-oriented software example.

Section 3 reveals the rest of the entities and relationships in the SBDM E-R diagram. The configuration entity presented in section 3.1 is used to configure the system and the manifestation entity in section 3.2 allows configurations to be instantiated. Finally, section 3.3 develops the examples considered before to illustrate the new concepts.

Section 4 concludes the report by summarizing some implications of the presented model.

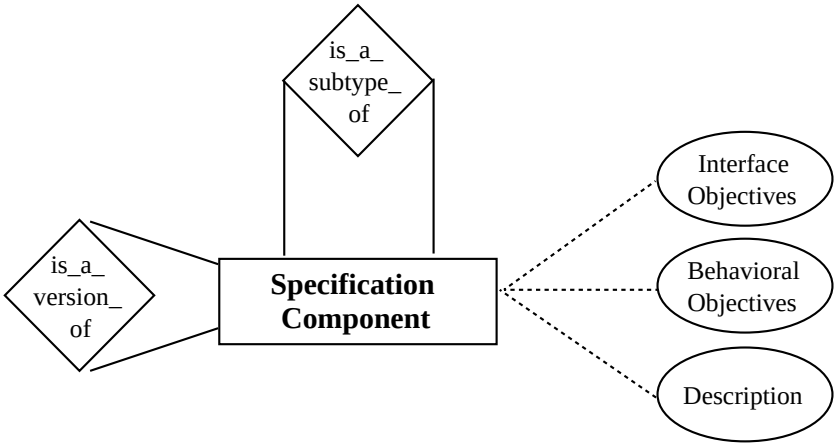


Figure 1: E-R Diagram: Specification Component

2 Design Components in SBDM

This section presents the design-oriented aspects of SBDM. The model provides alternating layers of specifications and implementations. Each specification may have many alternative implementations. A specification, by itself, does not describe the internal structure for the entity it specifies. It is the implementation that details the internal structure.

2.1 Specification Component

A *Specification Component (SC)* is a precise statement of objectives expected to be satisfied by some objects. This statement has three parts – Interface Objectives, Behavioral Objectives and a Description (Figure 1). *Interface Objectives* define the protocol of the interaction between the objects and the external world. *Behavioral Objectives* define the functional characteristics expected of the object. The above two objectives should necessarily be syntactically or semantically verifiable. The *Description* section informally states non-verifiable and other miscellaneous objectives.

A specification may be versioned to reflect a maturing design. An SC *is_a_version_of* another if the former is directly derived from the latter. The graph formed by SCs and the

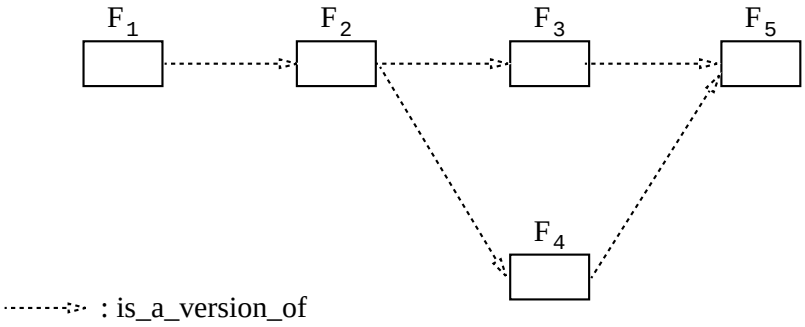


Figure 2: Versioning in Specifications

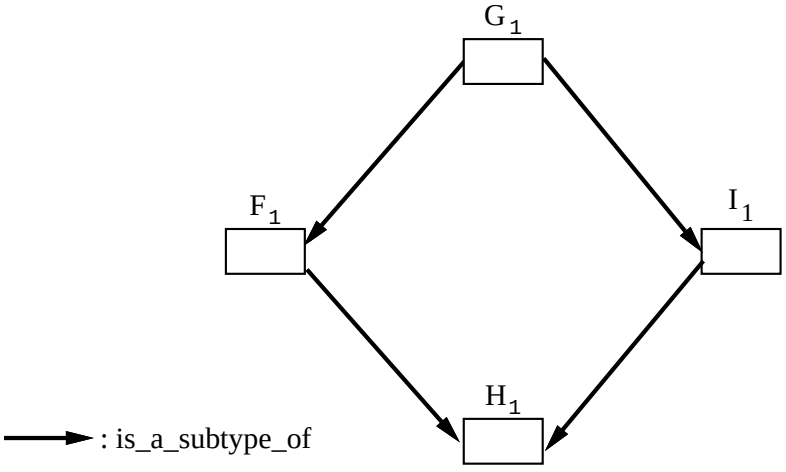


Figure 3: Specification Hierarchy

is_a_version_of link between them is assumed to be a directed acyclic graph [2] (Figure 2). The nodes of a connected component of this graph form a *specification set*. For purposes of this explication, we denote elements in this set by a uniquely subscripted capital letter. This letter, however, is common for all the elements in the set.

SCs may also be used to create a subtype hierarchy. If A_m *is_a_subtype_of* B_n then A_m is a specialization of B_n, or, alternatively, B_n generalizes A_m. The structure formed by considering SCs as nodes and *is_a_subtype_of* relationships as directed edges is again a directed acyclic graph. In Figure 3, F_1 and I_1 are subtypes of G_1, thus G_1 intersects the objectives stated in F_1 and I_1. H_1 is a subtype of both F_1 and I_1, therefore it specifies objectives which are in either F_1 or I_1.

2.2 Implementation Component

A Specification Component *is_implemented_by* an *Implementation Component(ImC)* (Figure 4). This object necessarily satisfies the Interface Objectives of its specification. However, it only attempts to satisfy the Behavioral Objectives and the Description sections of its specification. The degree to which the ImC satisfies the Behavioral Objectives is reflected in the *status* of the *is_implemented_by* link. The implementation variants of a specification, represented as superscripted specifications, form an *implementation set* [3]. For example in Figure 5, the implementation set of F_1 consists of F_1^1, F_1^2, F_1^3. Furthermore, objects in the implementation set are related by the *is_a_version_of* link and form a DAG structure similar to specification sets.

A specification set together with the implementation sets for all of its elements together constitute a structured group meeting similar objectives. Elements in this group of plans and

[2] It is expected that most tools that track versioning create trees rather than DAGs; but, in keeping with the philosophical principles, we permit a more flexible structure. It is even arguable that a DAG is too restrictive. For instance, a cycle of design alternatives may record the exploration and rejection of a set of alternatives. However, there are other ways of indicating rejection, and thus we prefer to indicate rejected alternatives simply as "dead-ends".

[3] But the implementation sets of different specifications are disjoint. If an ImC A meets the functionality of two distinct SCs, then those SCs should be abstracted into a single SC which is implemented by A. This makes explicit the "union" of the two specifications.

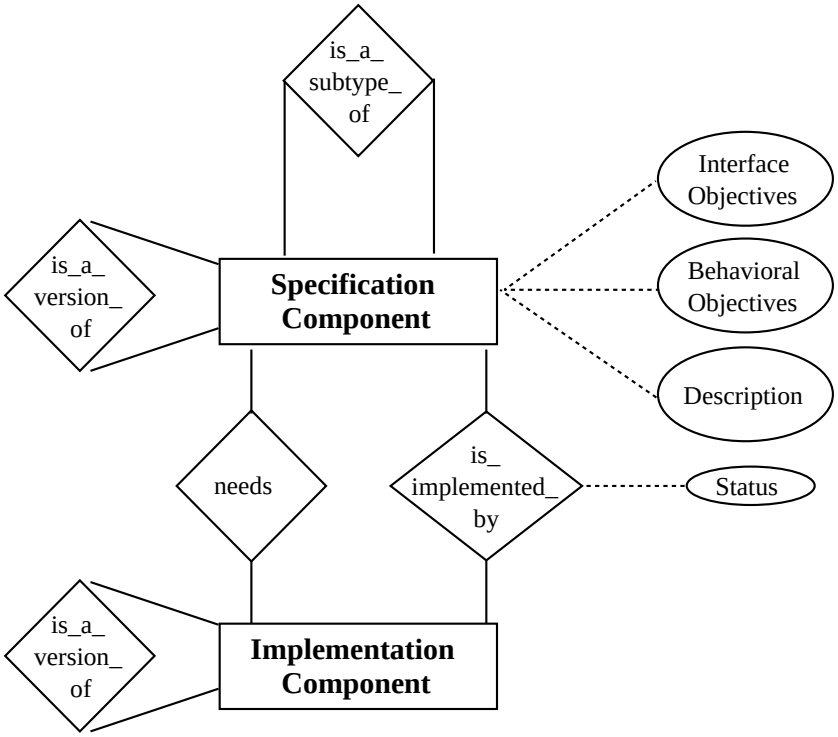


Figure 4: E-R Diagram: adding Implementation Component

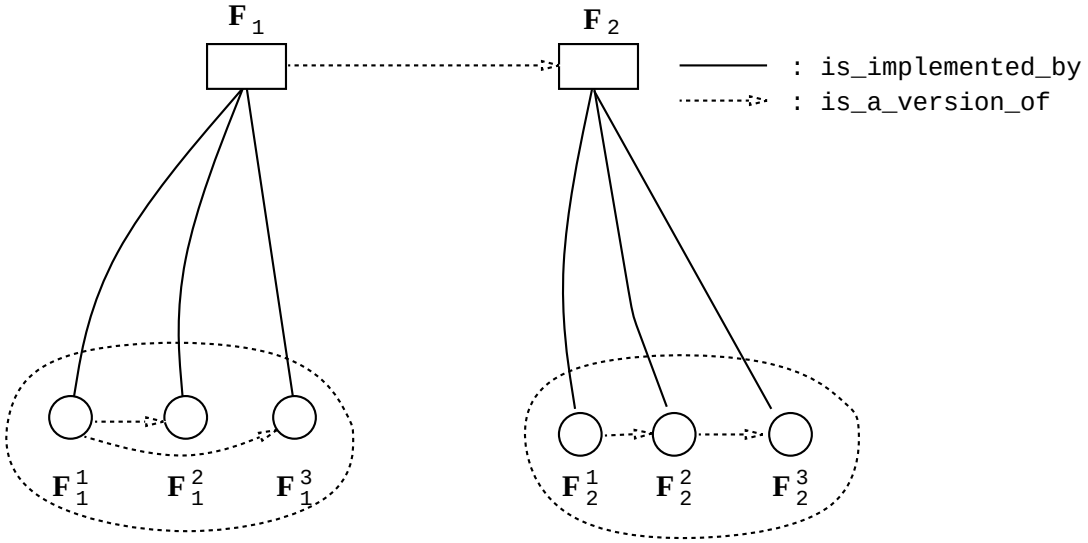


Figure 5: Functionality F

implementations form a cohesive unit called a *functionality*. We shall denote a functionality by the letter used to denote elements in the specification set for that functionality. Continuing our example (Figure 5), F_1 and F_2 and their implementation sets constitute the functionality F. Sometimes we will use the term functionality to refer to just one specification and its implementation set.

2.3 Multi-level Design

A system is usually designed as a component hierarchy with "lower" levels refining or decomposing "higher" ones. In SBDM, SCs and ImCs are fundamental to the design process and we refer to them collectively as *design components*. A single level design organizes these design components to construct various functionalities. A multi-level design, in turn, organizes the functionalities heirarchically to form subsystems. This is done by linking an implementation of a higher functionality with a specification of its constituent functionality. In effect, an ImC of the higher functionality *needs* SCs of the lower functionality (Figure 4).

The *needs* relationship between an implementation and a specification corresponds to

those relationships among modules which define the software architecture. The semantic richness of such relationships necessitates at least two kinds of *needs*. The first, *uses*, indicates that an implementation uses the facilities promised by the specification. A procedure call in a software system is a good example of this. The second, *is_composed_of*, indicates that a module is a part of another. This may occur, for example, in a mechanical assembly. An obvious difference between the two is that *uses* link between functionalities may form a cycle, the *is_composed_of* link may not.

Conceptually, the *needs* relationship could be considered as linking functionalities. However, linking functionalities rather than design components would result in a rigidly structured system. Designers must be free to allow different implementations within [4] a functionality to decompose into different lower-level functionalities. A direct link like the *needs* relationship between ImCs and SCs furnishes the model with such a flexibility. In Figure 6, for example, F_2^1 and F_2^2 need only G_1, F_2^3 needs both G_1 and H_3. We also allow multiple *needs* from an ImC to a SC. These links indicate that several copies of the object specified by the SC are required. For example, H_3^1 requires two copies of I_2. The issue of copy identification is obviously relevant only for some technologies.

Figure 6 also illustrates a few constraints in the relationship between SCs and ImCs.

- *The fan-in of an ImC (via is_implemented_by) is always equal to one.* In other words, an ImC attempts to implement only one specification. A need to generalize two specifications should be manifest in a generalization hierarchy of specifications as in Figure 3.

- *The fan-in of an SC (via needs) is greater than or equal to 1 (except for the root, which specifies the entire system).* For example, I_2 is needed by both G_1^2 and H_3^1.

- *The fan-out of an ImC (via needs) is greater than or equal to 0.* An ImC may use more than one SC or not use them at all. If the fan-out of an ImC is 0, we call that ImC *atomic*.

- *The fan-out of an SC (via is_implemented_by) is greater than or equal to 0.* A specification may have zero or more implementations.

[4] Note the terminology. ImCs implementing the same SC are implementations *of* the SC *within* a functionality

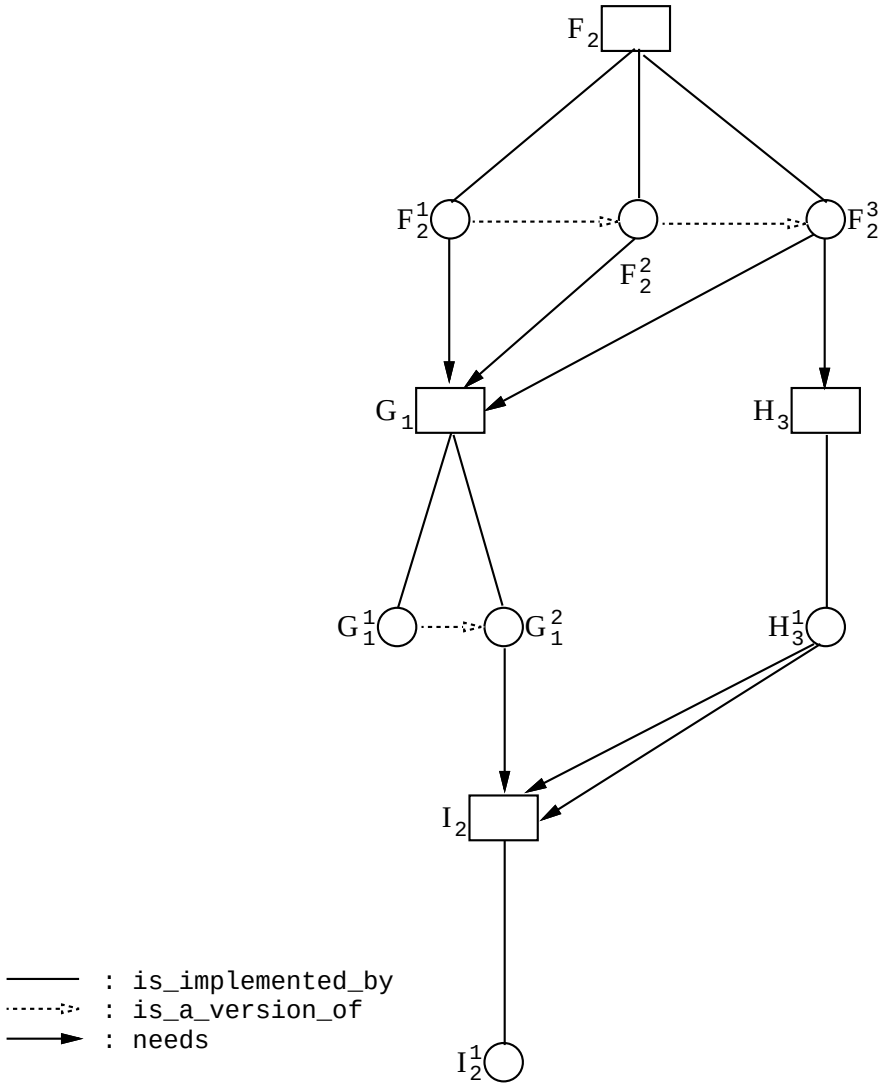


Figure 6: Multi-level Design

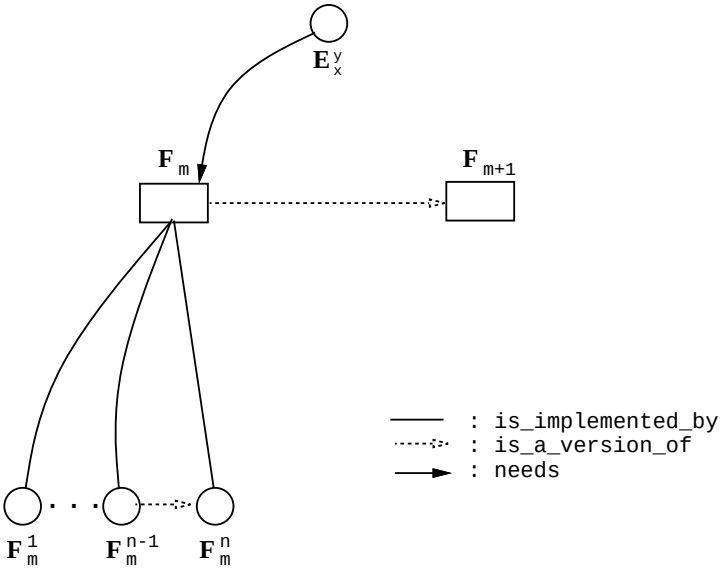


Figure 7: Version Percolation

The structure of a multi-level design is similar to an AND-OR graph [5]. In Figure 6, the assembly of F_2^3 requires both G_1 and H_3. At G_1 and H_3 exactly one of the implementations is chosen, say, G_1^2 and H_3^1 respectively. Thus, SCs in multi-level design correspond to OR nodes and ImCs correspond to AND nodes.

Systems which are built using alternating implementation and specification layers have significant advantages.

- Version percolation is effectively controlled. An SC succinctly describes what is expected of its ImCs. In Figure 7, a new version F_m^n has been derived from F_m^{n-1}. Since E_x^y requires F_m and F_m^n still satisfies F_m there is no need for versions to be percolated above. However, when a new version of an SC, say F_{m+1}, is derived, the situation is different. Since the contract between E_x^y and F_m is still valid, there is no need for another version, say E_x^{y+1}, to be derived from E_x^y. In fact, creating the new version E_x^{y+1}

[5] The structure is not exactly an AND-OR graph because a subtype hierarchy may be present.

linked to F_{m+1} is an invalid operation because the objectives of E_x^{y+1} may be incompatible with F_{m+1}. Thus, in both cases, versioning is essentially a local phenomenon and, hence, controllable.

The designer of E_x^y may feel the need to be notified when F_{m+1} is created. A design method using SBDM may easily incorporate such a notification process. However, SBDM does not mandate it since it attempts to define only structural relationships among components.

- A decision as to whether a new object is just a version of a previous implementation or belongs to a new functionality is made rather trivially. If the new object satisfies the objectives stated in a SC then it should be in the implementation set of that specification. Otherwise, it constitutes a new functionality (and hence a new SC should be created to accommodate the new object).

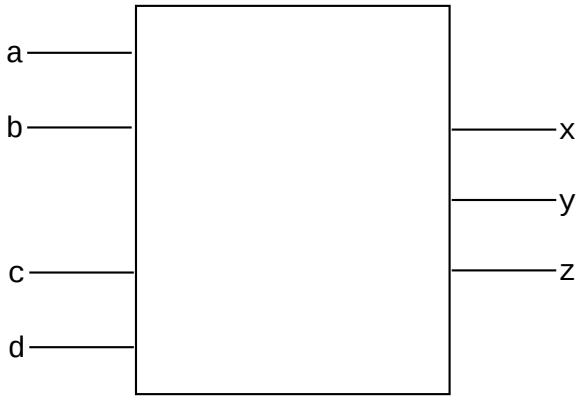
- The Description section attempts to capture uncertain human factors associated with engineering a system. The isolation of the fuzzy aspects allows one to maximize the potential for automating the design process. In early stages of design, SC nodes high in the tree are expected to have a large Description which would be transformed into Behavioral Objectives as the requirements are refined.

- The design process becomes disciplined simply because new ImCs are not allowed in the model before their specifications are defined.

2.4 Example applications

SCs, ImCs and the structural relationships between them constitute an important part of SBDM. This section illustrates the wide applicability of these concepts by structuring systems from different engineering disciplines. We will return to the explication of SBDM in Section 3.

The first example is a detailed one and illustrates the versioning mechanism. In this example, we also specify the internals of a SC and an ImC, so as to clarify the relationship between the design components. The next example, which considers the design of mechanical objects, illustrates the subtyping mechanism. The last example demonstrates the power of



Interface Objectives
input : a b c d
output: x y z

Behavioral Objectives
x = (a+c) MOD 2
y = ((a+2b+c+2d) MOD 4) DIV 2
z = (a+2b+c+2d) DIV 4

Description
The input lines a,b and c,d encode two 2-bit numbers. The adder outputs on x,y,z the 3-bit encoded sum of these numbers

Figure 8: Adder Specification: $Adder_1$ (step a)

the model in designing products in a domain with a rich set of structural relationships, namely the object-oriented domain.

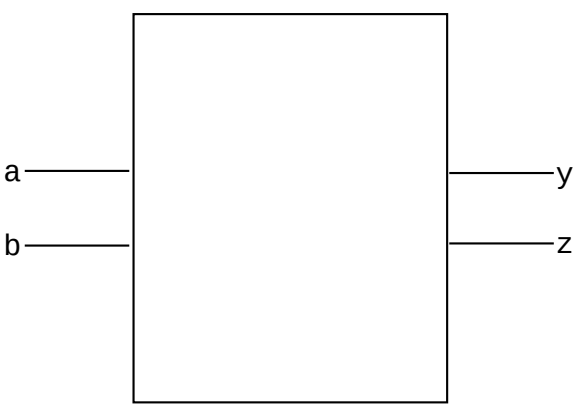
2.4.1 CAD Database

We model a CAD application using an example similar to that used in [AN91]. Consider the design of an adder which outputs the sum of two 2-bit numbers. In order to illustrate the versioning mechanisms of our model, we will assume a hypothetical scenario consisting of the following steps

a) We begin by specifying our objectives for the adder (Figure 8). That we begin by defining a SC reflects the importance of specifications in our model.

b) Assuming that a half adder would be useful in implementing the 2-bit adder, we create the specifications for an adder slice (Figure 9). Since this is the first version of the adder slice we call it $AdderSlice_1$.

c) $Adder_1$ is now implemented using the $AdderSlice_1$ as a subcomponent (Figure 10). Since this is the first implementation of $Adder_1$ the ImC is named $Adder_1^1$.



Interface Objectives
inputs : a b
outputs: y z

Behavioral Objectives
y = (a+b) MOD 2
z = (a+b) DIV 2

Description
The sum of the inputs on a and b is encoded as a 2-bit number on the lines y and z.

Figure 9: Half Adder: $AdderSlice_1$ (step b)

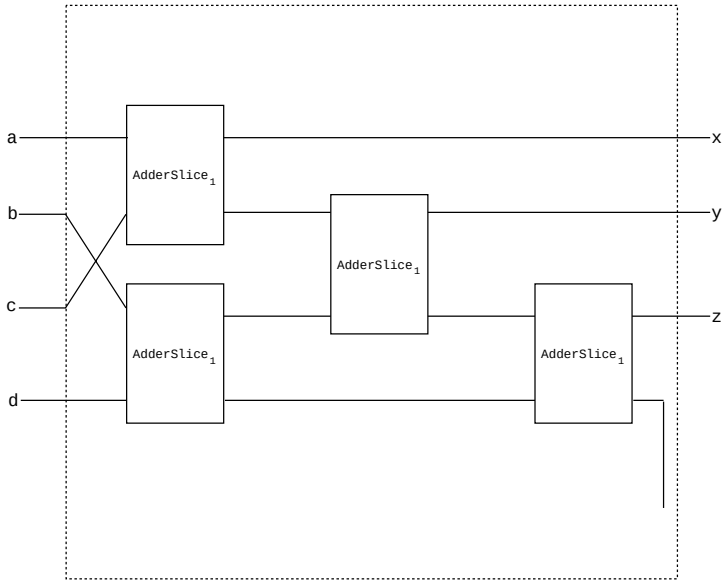
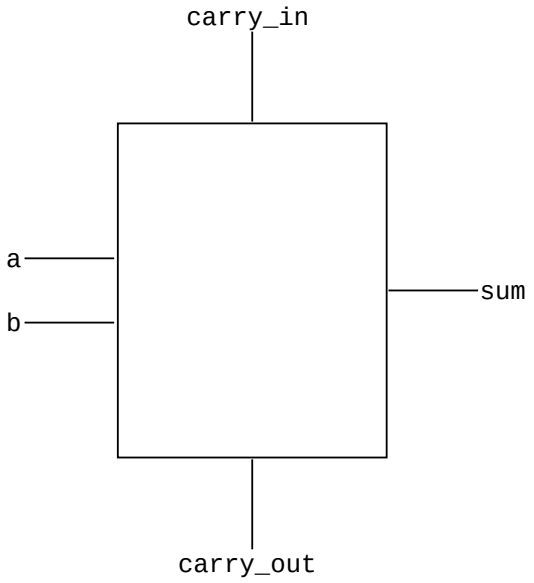


Figure 10: Adder Implementation: $Adder_1^1$ (step c)



Interface Objectives
inputs : a b carry_in
outputs: sum carry_out

Behavioral Objectices
sum = (a+b+carry_in) MOD 2
carry_out= (a+b+carry_in) DIV 2

Description
The circuit adds the individual inputs on a, b and carry_in. The unit place of the total is output on sum. The carry is presented on carry_out.

Figure 11: Full Adder: $AdderSlice_2$ (step d)

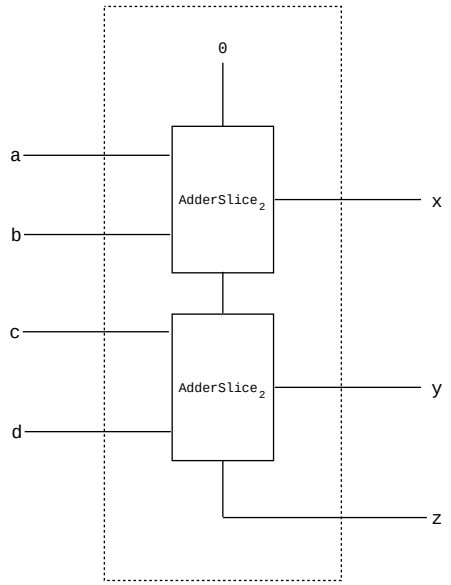


Figure 12: Adder Implementation: $Adder_1^2$ (step e)

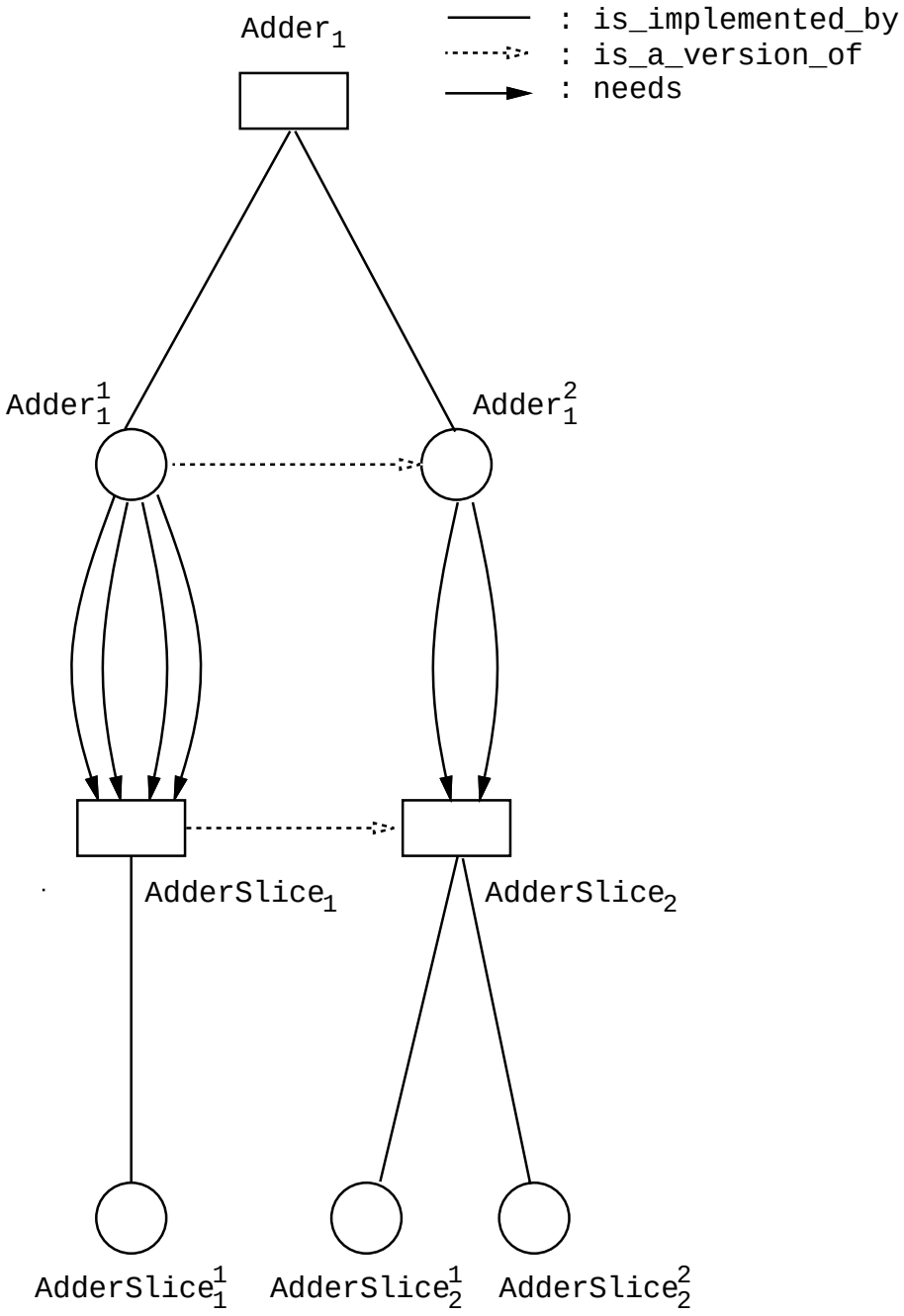


Figure 13: SBDM Design of the Adder

d) $AdderSlice_1$ may be improved to handle an extra carry input. The new adder slice, a full adder, is a version of $AdderSlice_1$ and named $AdderSlice_2$ (Figure 11).

e) $Adder_1$ is now implemented using the full adder (Figure 12).

Figure 13 summarizes the relationship between the components used in the adder design. The figure also shows the implementations for the adder slices. These may be completed any time after the specifications for the adder slices have been finalized.

2.4.2 Design of Mechanical Objects

We apply SBDM to a mechanical design by modeling a (highly simplified!) decomposition of a car. Figure 14 displays a SBDM design of a car. The CAR is composed of an ENGINE and a BODY. The BODY in turn is composed of a CHASSIS, AXLE and WHEELs. As before, the SCs for these entities consist of Interface Objectives, Behavioral Objectives and a Description. In case of a physical component, the Interface Objectives specify the dimensions and positions of the connected components. For example, the interface for the axle may specify the radius at which the bolts for the wheel are located. Behavioral Objectives and Description sections may state those details of a part that will be used in the design of a product using this part as a subcomponent.

In our CAR design, each specification is implemented by at least one ImC. The ENGINE specification has two implementations. The first is a `Low Cost Engine` and uses cheaper materials than the next which is a `High Cost Engine`. The CHASSIS also has two implementations – a `2 Door Chassis` and a `4 Door Chassis`. The ImC for a product database will generally include a detailed design of the component. In addition, it may specify manufacturing details or vendor information if the part is supplied from outside.

When two constituent parts of a product interact with one another then the interaction should be specified in the implementation object that contains the interacting parts. For instance, even though the AXLE and the WHEEL are closely tied together, we do not link them with the *uses* relationship. The exact relationship between them is contained in the implementation of BODY. Thus, the *is_a_component_of* relationship between parts is the only *needs* relationship of relevance in a parts database.

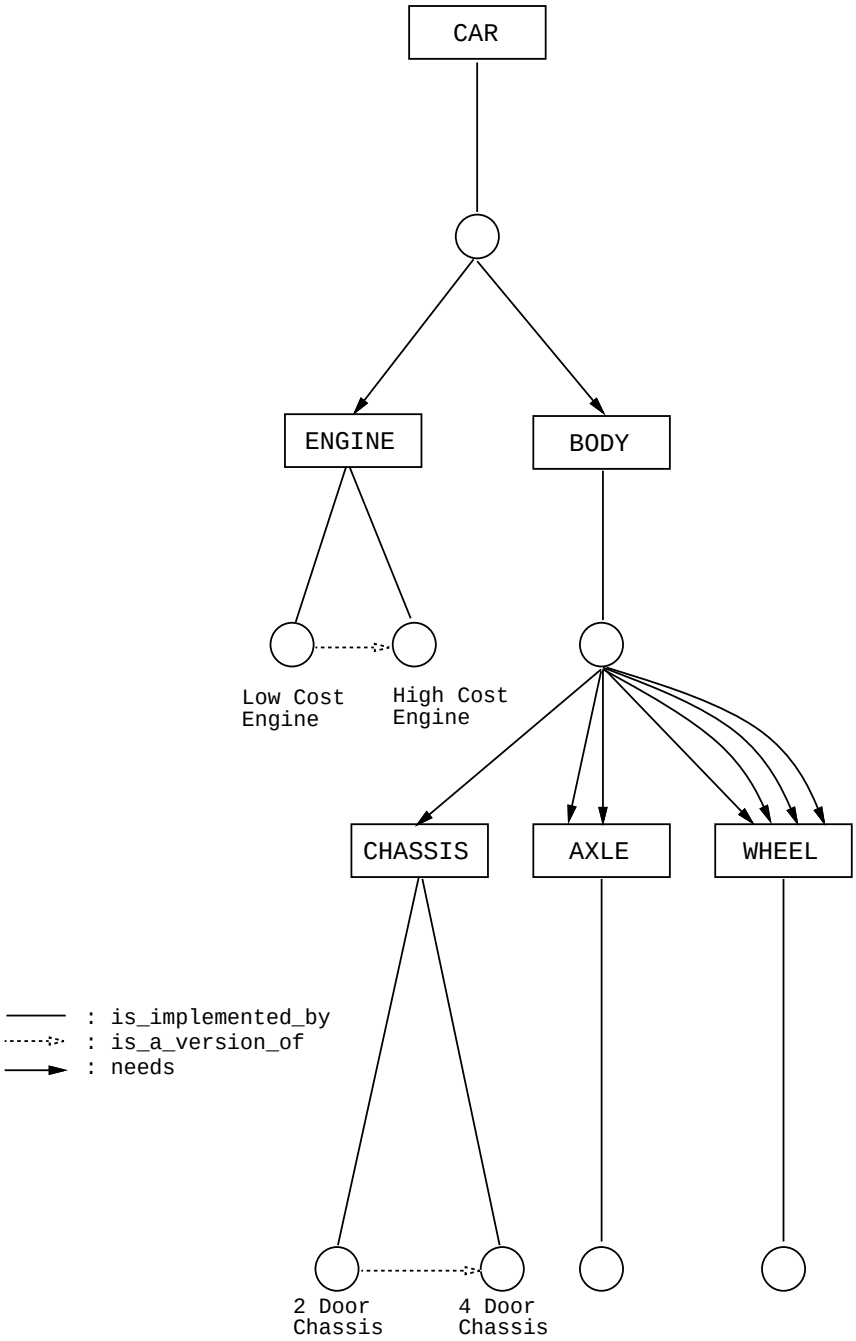


Figure 14: CAR: Design

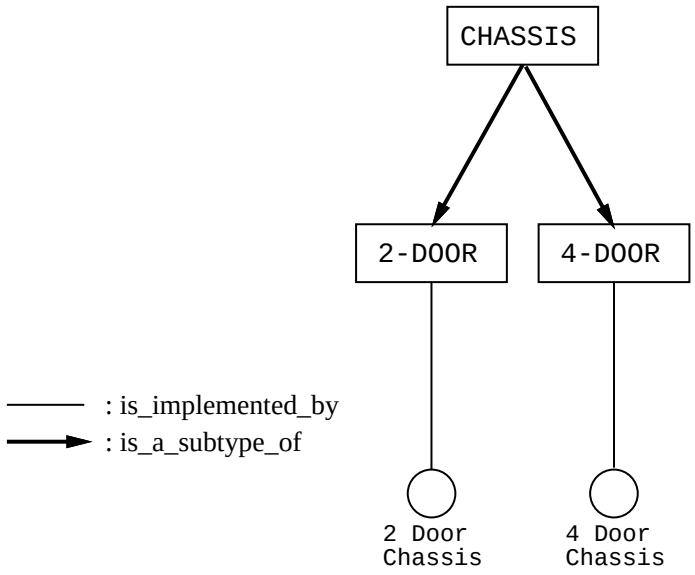


Figure 15: Chassis subtype hierarchy

A product database may also benefit from using a subtyping hierarchy. For instance, consider CHASSIS and its implementations. First we differentiate the 2 Door Chassis and the 4 Door Chassis at the specification level by creating SCs for them. These SCs may now be considered subtypes of CHASSIS (Figure 15). One may incorporate this change in figure 14 by substituting the subtree at CHASSIS with figure 15. Note that in figure 15 we have an example of a design which is not exactly an AND-OR graph. Specifically, both CHASSIS and the nodes below it correspond to OR nodes.

2.4.3 Object Oriented Software Systems

We illustrate the applicability of SBDM to object oriented software systems in this subsection. Our examples use principles represented in the design approaches of Booch [Boo91] and Meyer [Mey88].

A specification for a family of objects in SBDM is very close to the notion of a class interface in object-oriented systems. It defines the communication protocol used to interact with the objects, the behavior expected of the objects, and other important information

regarding the objects. And since most design methods define how exactly the class interface is specified, we can reuse the class interface as the SC for a class in SBDM.

For example, Booch defines a "class template" as a "means of documenting the meaning of each class". Among other elements, the class template contains fields and operations that can be reached by any client of the class (public interface) or a subclass of the class (protected interface). Furthermore, for each operation the parameters, result type, preconditions, post-conditions, action, documentation, etc. are defined. An examination of the above elements indicates that it does include the necessary elements required in a SC. Hence, if one follows the Booch model of object oriented design, the creation of a SC for each class requires no additional effort. In fact, this equivalence of a class template and a SC has some advantages. First, as shown before, the presence of specifications controls version percolation. Second, the integration forces the class documentation to be in tune with the implementations. Third, it represents a more realistic view of the design process as it closely ties specifications and implementations. This is one more example of how our model forces explicit recognition of design decisions (with the hope of capturing documentation on such decisions).

An object oriented system is usually a complex structure embedding in itself at least two hierarchies. The first, a structural hierarchy, defines the structural composition of the system. The second, a class hierarchy, defines the inheritance relationships among classes. In SBDM, the structural hierarchy is represented by the term is_implemented_by and the term needs relationships and the class hierarchy by the *is_a_subtype_of* relationship. That there is no subtype relationship between ImCs reflects the level of abstraction at which a class hierarchy should be defined.

Conventionally, the structural hierarchy of components in a system is drawn separately from the class hierarchies that the components participate in. This may incorrectly suggest that the hierarchies are independent of one another, which of course they are not. Since class hierarchies are defined on specifications and specifications are integrated in the structural hierarchy, SBDM models the two hierarchies as complementing one another.

The ImC and the *is_implemented_by* relationship have obvious parallel in object oriented systems. An Implementation Component may simply be the module that implements the class. For example, in ADA if each class is placed in a separate package, the package implementation may be considered an ImC for that class (and the specification package a

part of the SC).

We now adapt an example from [Boo91] to illustrate the above concepts. A home heating system provides and controls heat to individual rooms in a home. The structure of one system is modeled in SBDM in Figure 16. The rooms have temperature sensors which allow the `RoomControl` to message the `HeatFlowRegulator` whether it `NeedsHeat` or `NoLongerNeedsHeat`. The `HeatFlowRegulator`, which controls the furnace and the water valves in each room, responds by requesting the water valves to either `OpenValve` or `CloseValve`. Thus the functionalities `RoomControl` and `HeatFlowRegulator` have a cyclical *uses* relationship between them. The `HeatFlowRegulator` also has to `ReportFurnaceStatus` regularly and `ReportFault` in case of some other failure.

The `OperatorInterface` *is_composed_of* a `FaultResetSwitch`, which may be set by the `HeatFlowRegulator` or reset by the operator, a `HeatSwitch`, to `SwitchOn` and `SwitchOff` the heating system, and a `FurnaceStatus` indicator. The `FaultResetSwitch` and the `HeatSwitch` are both specialization of the class `ToggleSwitch`. A root class is also used in the example to initialize and start up the system.

3 Configuration and Manifestation Components

In the previous section we introduced and detailed the role of components needed for system design. This section introduces the remaining components in SBDM. Figure 17 presents the E-R diagram with added entities. The new components – Configuration Component and Manifestation Component – are required to configure the system and instantiate the configured system.

3.1 Configuration Component

A configuration for a system may be defined as the relative functional arrangement of its subsystems. In other words, a configuration mechanism must not only identify the relevant subsystems but also specify their placement in the system structure. In SBDM, each functionality may have multiple implementations and each implementation needs other functionalities to fulfill its objectives. Thus, configuring a system would require a marking of an

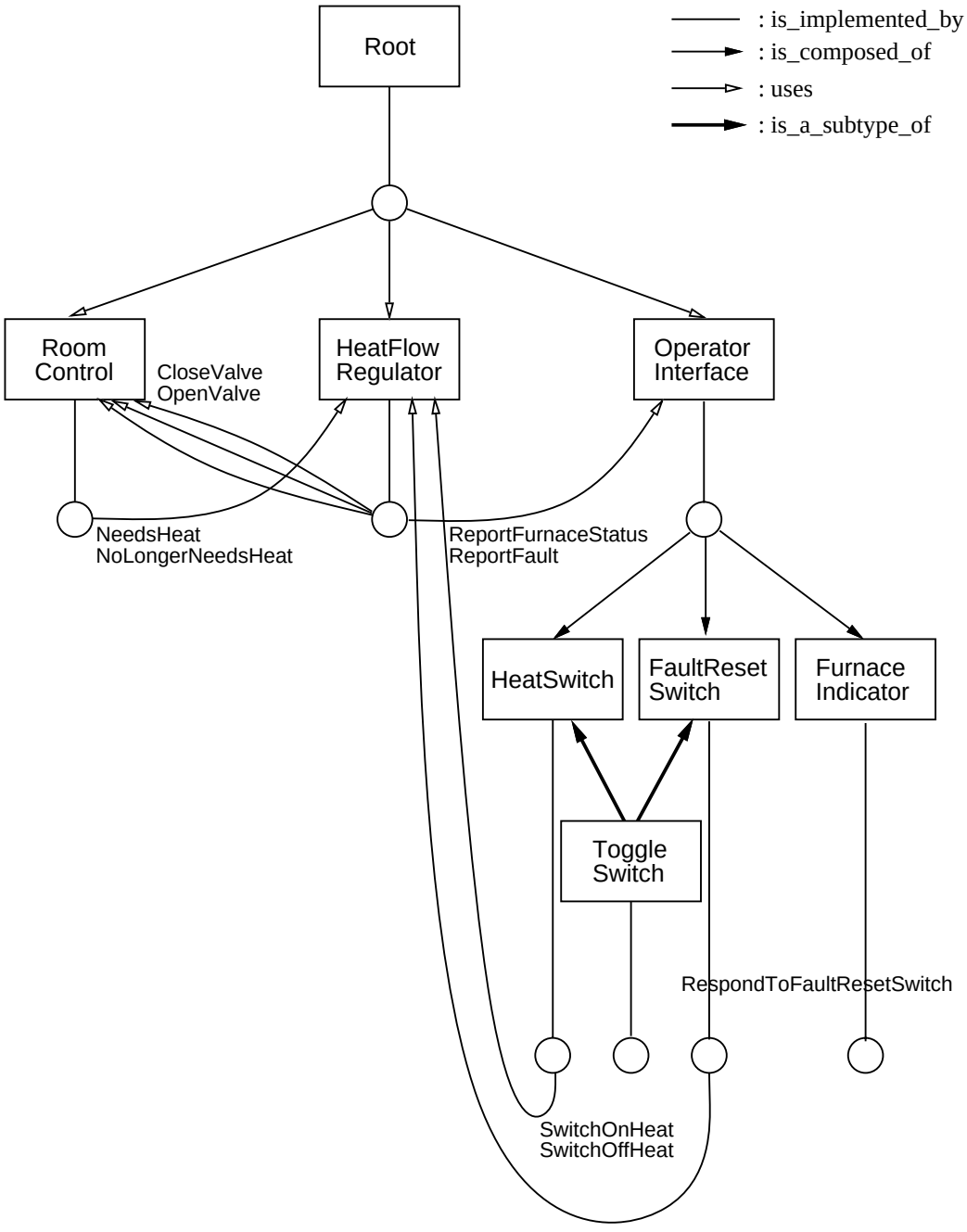


Figure 16: Home Heating System

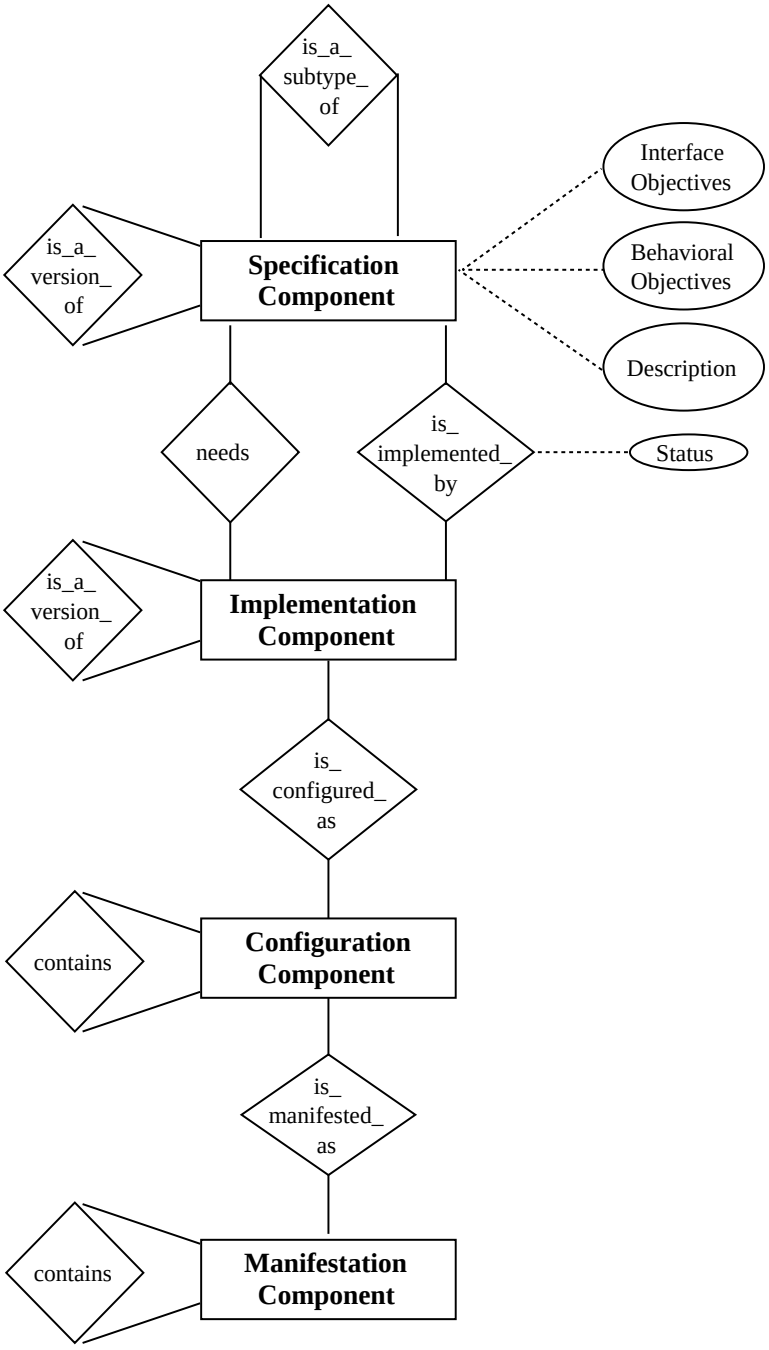


Figure 17: E-R Diagram: adding Configuration and Manifestation Components

ImC and, recursively, the ImCs from each functionality it needs. This marking is done using the *Configuration Component (CC)*. The CCs themselves link to other CCs such that one *contains* another. This results in a structure which parallels that of the ImCs in the system design.

We illustrate the mechanism by considering the scenario in Figure 18. To assemble a system with specification F_m, we mark an implementation F_m^n as being of interest by using a CC, say $C : F_m^n$ [6] . This is denoted by saying F_m^n *is_configured_as* $C : F_m^n$. For each specification G_x needed by F_m^n, a design version G_x^y is chosen and $C : F_m^n$ is linked to the CC for G_x^y, say $C : G_x^y$. Thus, $C : F_m^n$ *contains* $C : G_x^y$. If G_x^y *needs* other functionalities we repeat the above for specification G_x, where G_x^y is the ImC of interest for G_x. Else, we are done.

Of course, it may not be necessary to explicitly create CCs and link them to get the desired configuration. A method of defaults may achieve this in a more implicit manner. The idea here subsumes that used in [CK86]. For each specification one may designate a distinguished object from its design set as *current*. To configure an object, we recursively construct it using the current versions of each functionality linked by the *needs* link. An atomic ImC, is configured using the ImC itself. Consider Figure 6 again. If F_2^3, G_1^2, H_3^1, and I_2^1, are the current default implementations for their specifications, then the current configuration for F_2 may be implicitly structured as in Figure 19.

Obviously, configurations resulting from this method depend on the designations of current versions at configuration time. If we assume that the current versions are the best versions in their design sets, then we may consider the implicit configuring process as achieving a global best from purely local decisions. However, we do recognize that the best may not always be so easily distinguished and hence needs a more explicit configuring mechanism.

3.2 Manifestation Component

[Kat90] refers to the need for distinguishing an instance hierarchy and a definition hierarchy. The *Manifestation Component (MC)* entity in SBDM together with the *contains* relationship

[6] Note that there may be many CCs for an ImC. Thus, $C : F_m^n$ does not represent a unique CC for the ImC F_m^n. The same holds true for the ImC G_x^y.

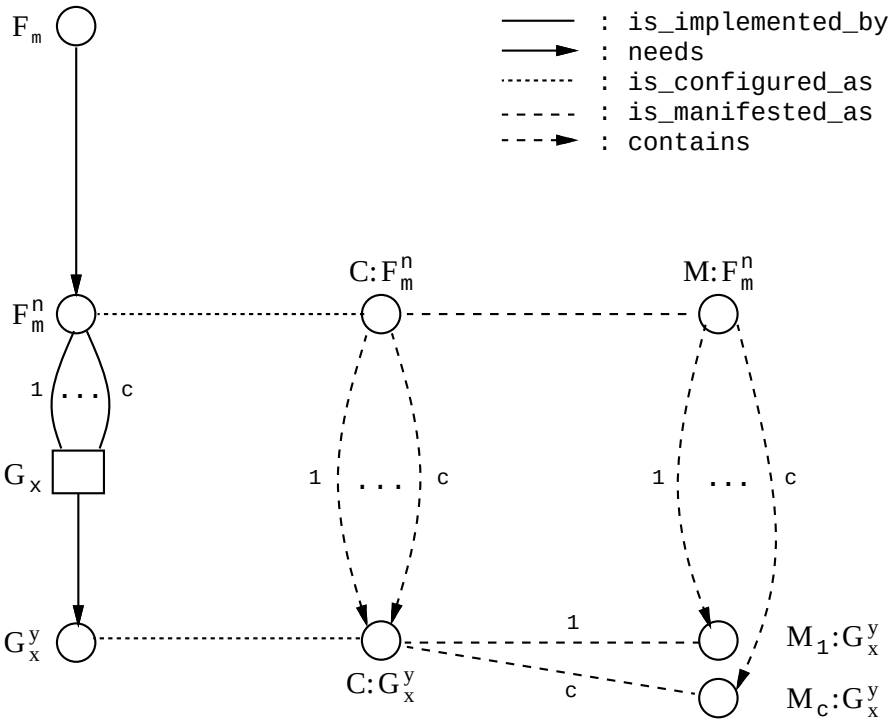


Figure 18: Configuring and Instantiating a system

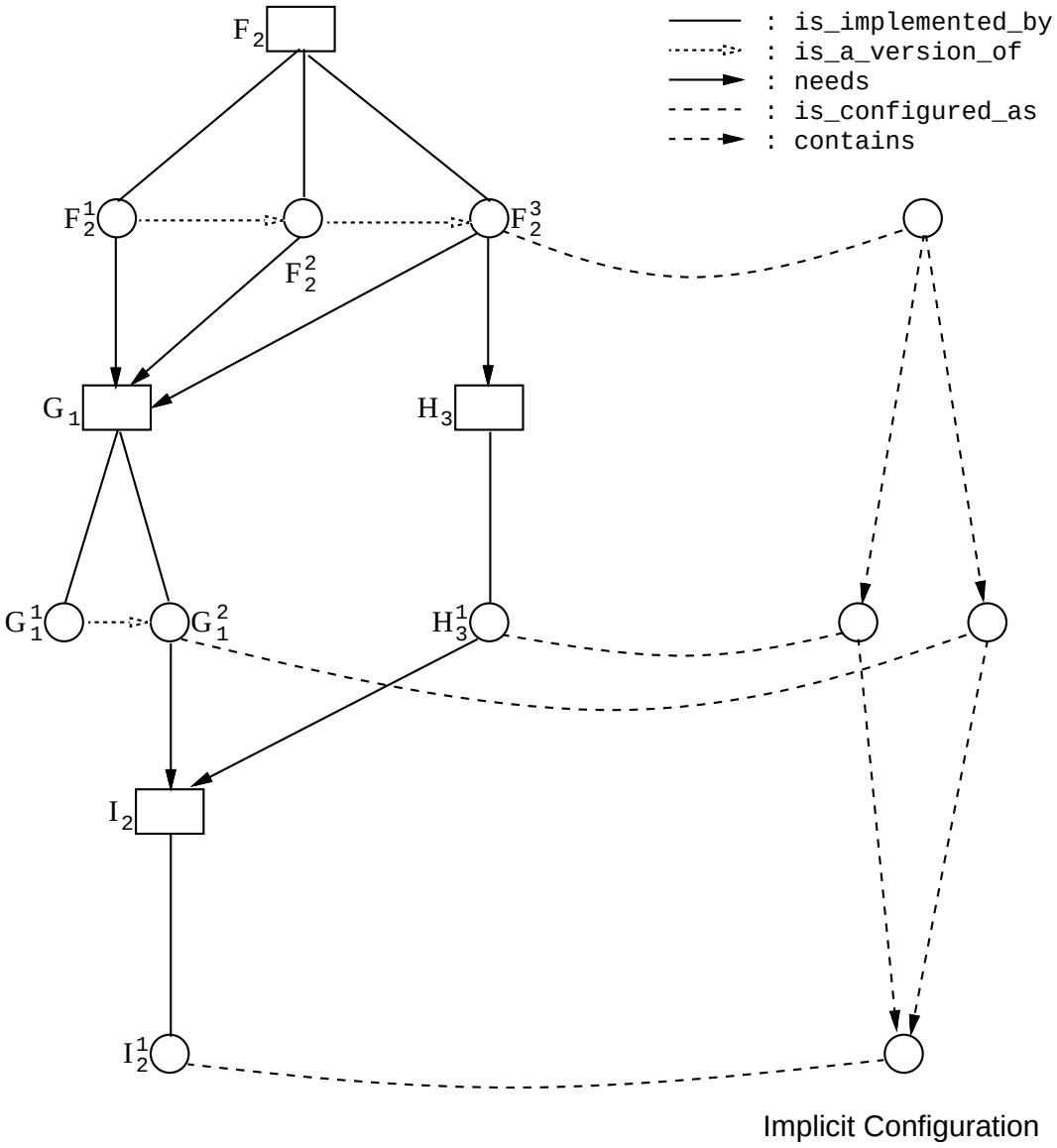


Figure 19: Implicit Configuration using defaults

enables us to explicitly create an instance hierarchy. Furthermore, the *is_manifested_as* relationship links the instance hierarchy with the definition hierarchy, which is built using the CC entity.

The instance for a CC, say $C : F_m^n$, is manifest as a manifestation component, say $M : F_m^n$. Because F_m^n (and hence $C : F_m^n$) is a composite, $M : F_m^n$ contains other manifestation components. These satisfy the following constraints:

- For each *contains* link from $C : F_m^n$ to $C : G_x^y$, $C : G_x^y$ *is_manifested_as* $M_1 : G_x^y$, ..., $M_c : G_x^y$, where c is the number of *contains* links. Each manifestation of $C : G_x^y$ is then related to $M : F_m^n$ by *contains*.

- Given a manifestation $M_a : G_x^y$, the CC from which the manifestation's parent in the instance hierarchy is manifest, is the same as, the parent of the CC in the definition hierarchy from which $M_a : G_x^y$ is manifest.

3.3 Example Application

We extend the CAR design example in section 2.4.2 to illustrate configuration and manifestation mechanisms in SBDM.

The right side of Figure 20 uses CCs to configure two types of cars. The 2-DOOR COUPE configuration uses the low cost engine and the 2-DOOR chassis. On the other hand, the 4-DOOR SEDAN uses the high cost engine and the 4-DOOR chassis. Other subcomponents are the same for the two models. Note that we needed to link a version of an engine to a specific chassis. This is a constraint on the structure of the car and appears at a global level. Since the design stage handles essentially local concerns, we need a mechanism to handle the global constraints. The process of configuring a design using CCs provides such a mechanism.

After the local and the global concerns of the design have been handled in the previous stages, one may physically realize the product. Continuing our example (Figure 21), we manifest the CC for the 4-DOOR SEDAN. Manifestation provides an identity to each element. This is especially clear if we consider the wheels of the car. The design and configuration

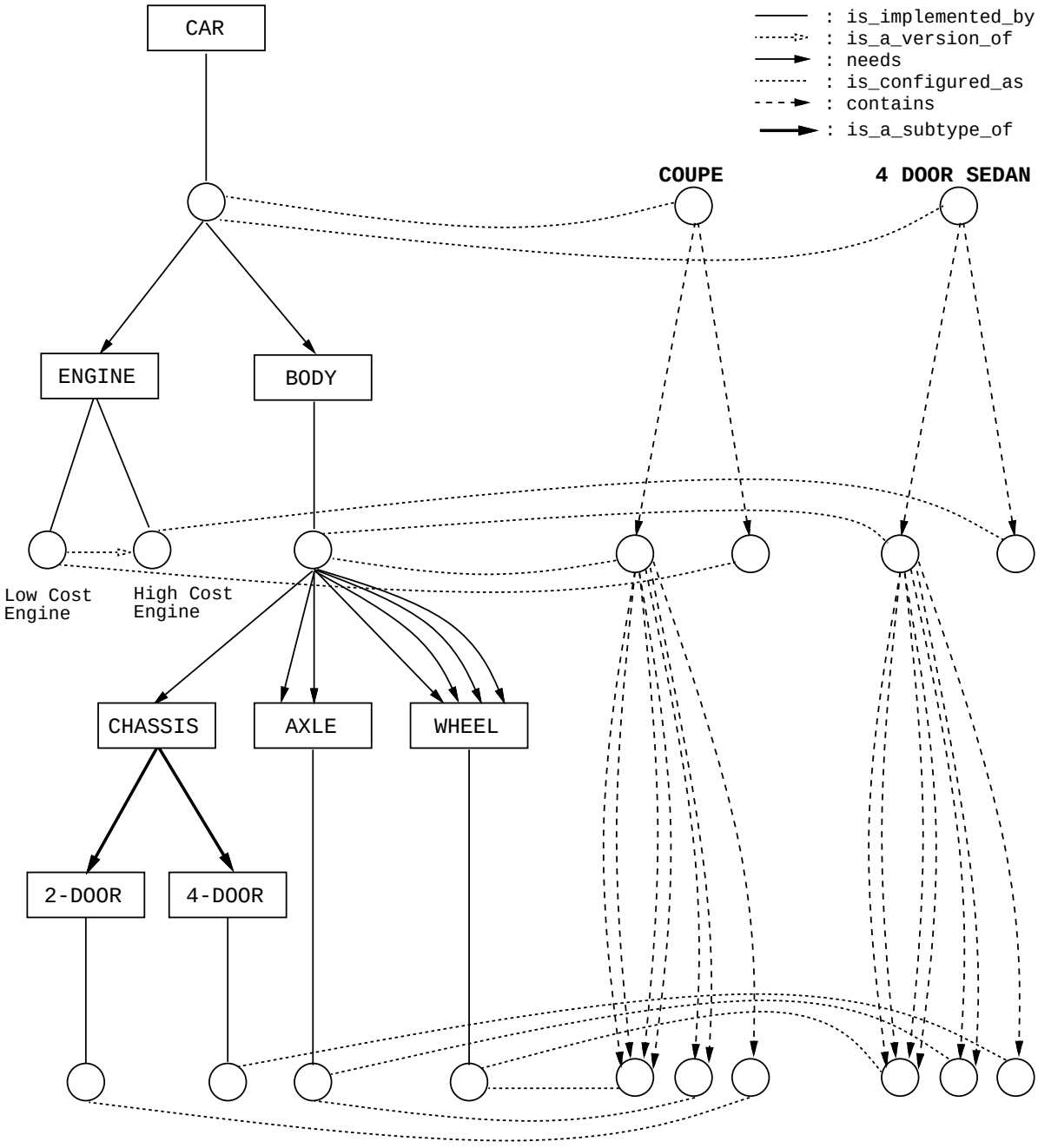


Figure 20: CAR: Design and Configurations

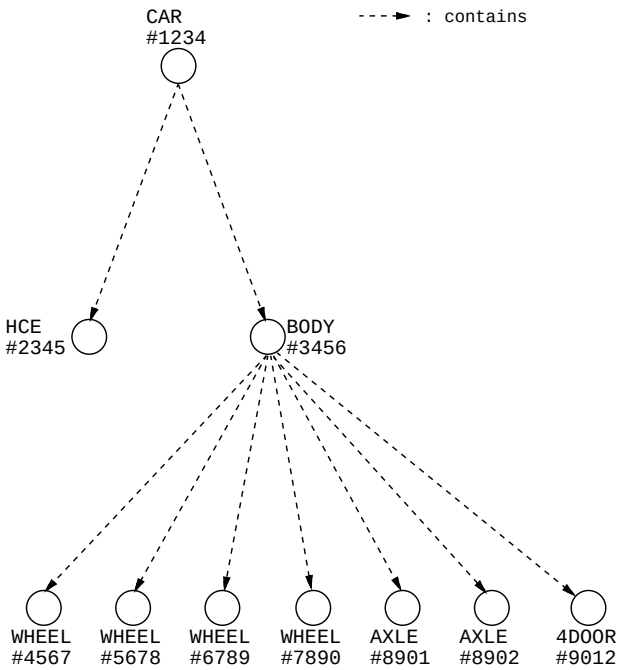


Figure 21: 4-Door Sedan: Manifestation

stages contained multiple references to the same design object. At the manifestation stage, however, each wheel is different and thus has its own identity.

4 Conclusion

We have presented a model which handles issues of system evolution, configuration, subtype management and system architecture in a unified framework. The following are what we believe to be important consequences of viewing products as incorporating specifications, implementations, configurations and manifestations. First, issues of local concern are handled at the level of a functionality and issues of global concern are handled using configurations. Second, version percolation is easily controlled by linking implementations to their specifications. Third, the design process proceeds in a manner which is realistic and keeps the specifications and implementations synchronized.

References

[AN91] R. Ahmed and S.B. Navathe. Version Management of Composite Objects in CAD Databases. In *Proceedings of the 1991 ACM SIGMOD International Conference on the Management of Data*, pages 218–227, May 1991.

[Boe88] B. W. Boehm. A spiral model of software development and enhancement. *IEEE Computer*, 21(5):61–72, May 1988.

[Boo91] G. Booch. *Object Oriented Design with Applications.* Benjamin/Cummings, Redwood City, California, 1991.

[CK86] H. T. Chou and W. Kim. A Unifying Framework for Version Control in a CAD Environment. In *Proceedings of the 12th VLDB conference, Kyoto, Japan*, pages 336–346, August 1986.

[Kat90] R. H. Katz. Toward a Unified Framework for Version Modeling in Engineering Databases. *ACM Computing Surveys*, 22(4):375–407, December 1990.

[Mey88] B. Meyer. *Object-oriented Software Construction.* Prentice Hall International (UK) Ltd., Hertfordshire, HP2 4RG, 1988.

[SB82] W. Swartout and R. Balzer. On the inevitable intertwining of specification and implementation. *Communications of the ACM*, 25(7):438–440, July 1982.