Published in "Managing Integrity in Design Information Flows", Computer Aided Design, (May, 1996), 28:6/7, pp.551-565.

MANAGING INTEGRITY IN DESIGN INFORMATION FLOWS

Charles M. Eastman

Center for Design and Computation

University of California at Los Angeles

ABSTRACT:

This paper addresses integrity rules that are embedded within engineering design applications and that apply between applications. A representation for integrity rules that are embedded in applications is presented and a set of related methods developed for: (a.) maintaining the integrity condition of application developed data, (b) managing the precedence order between applications, in the context of (c.) changing the schema and the associated mix of applications and (d.) iterated execution of applications and change propagation. Both integrity rules literally embedded within external applications and others required to be embedded within a database are considered. The techniques are demonstrated with an extensive example.

KEYWORDS: integrity, constraints, concurrent engineering, change propagation.

I. INTRODUCTION

Significant efforts are being made to integrate multiple engineering and design applications using a backend database. The general goals are the same as those for other database uses: re-use of data needed in multiple applications, managing the consistency and integrity of data, and support for an extended set of applications. Engineering design, however, involves different conditions from those in other database applications. In response to these different needs, database researchers have focused on such issues as version control [17], dynamic schema extensibility [2], composite objects [18] and long transactions [3].

Motivating recent work on engineering databases is the need to both shorten design and manufacturing times and to improve product quality. Called *concurrent engineering*, this line of work strives to integrate design, production and operation planning processes by carrying them out in parallel and with improved cross communication [20]. Concurrent engineering is only possible with effective data integration and exchange. Partially in response to these needs, a major effort is underway to develop international standards for the representation of products, known as ISO STEP (Standard for the Technical Exchange of

Product Model Data) [28]. The goal of STEP is to provide improved data exchange mechanisms for engineering and design.

In databases, *semantic integrity* is the correctness of data stored in relation to the world being modeled. Integrity can be defined, explicitly or implicitly, by logical rules over subsets of the data [29]. Traditionally, in a database supporting multiple applications, each application must update the database in such a way that the integrity rules of *all* supported applications are not violated. This results, for example, in initially integrated applications possibly having to be revised if a later added application requires new integrity rules. There has been a variety of efforts to represent integrity rules within the database itself, as a more centralized means to manage integrity [13]. Each application then only needs to transmit its integrity rules to the database, which manages the rules of all applications. Triggers and constraints are two mechanisms developed to implement integrity rules. In general, these mechanisms require that each schema modification and data update satisfy all integrity rules [19], [25]. Such integrity preserving actions are called *transactions* [14].

Semantic integrity is particularly important for databases supporting concurrent engineering design. The general purpose of design is to specify a product in sufficient detail for fabrication and to simultaneously allow evaluation of and to make acceptable the various behaviors of the product before it is produced. Concurrent engineering expands the behaviors to be evaluated. Design typically involves a large number of independently developed applications that predict the behavior of the product, such as performance, manufacturability, or operability and/or elaborate the general product description, such as its geometry or material properties. Performance based applications apply mathematical rules of physical behavior to determine that the product description will perform as required and possibly select properties that will achieve the desired performance, i.e., beam sizing of structures.. Geometric modeling applications rely on representations with numerous complex well-formedness rules [26]. Thus most engineering applications apply various rules assessing a products integrity and/or take actions that satisfy some integrity rules. Managing the design's overall integrity condition is part of the task of design. Many integrity rules in engineering are too complex to embed within the database itself. It would still be extremely beneficial, however, if they could be managed and their status guaranteed.

In design and many other database application areas, a precedence order exists over applications. For example, an accounting database that posts payroll might involve three applications: (1.) update all salary changes, (2.) update absences, and (3.) compute payroll checks. They have the execution order shown in Figure 1. That is, both salary changes and absences must be entered before the payroll can be correctly computed. It can be asserted that for the payroll application to execute correctly for any pay period, the following integrity rules must first be satisfied:

a. "salary changes must be defined for the pay period".

b. "employee absences must be defined for the pay period".

The two assertions are associated with the “post payroll” application and specify the conditions in which it can be correctly executed. If because of some clerical error, one or more salary changes were overlooked, the payroll checks would be incorrect and would have to be re-run after the salary changes were updated. Alternatively, a work-around process might be undertaken. The work-around might involve dealing with the specific employee salaries' changed or specific absences that were in error. This suggests that the above integrity conditions can be re-defined to apply not to the set of all salary changes and absences, but for those related to each employee.

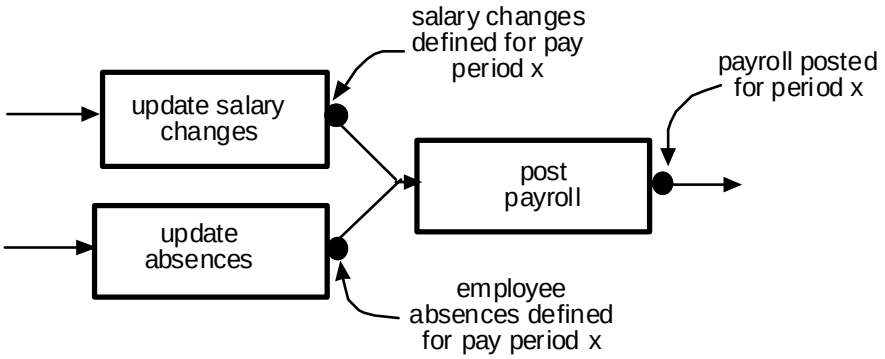

FIGURE ONE: The three tasks are shown in boxes, while the two integrity rules realized by the first two applications are shown filled circles.

Precedence relations are based on both the existence of information used in an application and the state of the information. In the example above, the information existed but was in the wrong state for the intended use. In a similar manner, engineering data undergoes a variety of tests that may not modify the data but only assess conditions that the data is required to satisfy. In a distributed design environment, information may be available but have some integrity conditions satisfied and others not. A major issue in engineering database management is to define and guarantee the integrity condition of the data stored.

Because design is not a post facto recording of facts, but rather an apriori specification of future actions, a design database represents candidate decisions that are subject to iteration and change. Changes apply at two levels:

At the database schema level. Materials and fabrication technologies are sometimes selected during the process of design, which require particular schemas to describe them, e.g., composite material versus steel. Also new performance requirements emerge during the development of a design, for example regarding vibration or due to the interaction between materials. Schema changes have associated integrity rules both applying internally within the added subschema and between it and the rest of the schema. For example, vibrations may be sensitive to weight distribution of other systems, and

composite materials may be sensitive to gaseous emissions of other materials. Schema changes must address both kinds of integrity relations.

Secondly, at the value level. Within any design database schema, value assignments are likely to be iterated. Some aspect of the process may be iterated several times, with each change having ramifications elsewhere. For example, a change in the layout of an automobile dashboard may effect the wiring, fuses and electrical circuits. These changes are caused by complex chains of integrity rules across database values. Currently, the implications of such changes are managed manually.

Both types of change create serious problems for integrity management in engineering design. Schema changes modify the set and applicability of integrity rules. Iteration of value assignments requires managing the validity of integrity rules which may be organized in complex networks of interaction.

This paper refines a general representation for semantic integrity rules associated with complex applications first introduced in [21]. It then introduces rules supporting integrity management during schema evolution and design iteration. Section II presents a detailed example from automobile design and manufacturing, used later to demonstrate the proposed methods. In Section III, a general representation for semantic integrity rules is presented along with predicates defined for characterizing the relations among design applications. In Section IV, this representation is used to address the issue of precedence relations among applications and the effects of design iteration. These two sections show the applicability of the methods presented by applying them to the example. In Section V, implementation and management issues are discussed. Section VI relates the work presented here to the parallel work developing international standards for product models within ISO STEP.

II. THE EXAMPLE

The example is based on recent literature [5,6] to reflect the typical flows encountered in modern product development [10]. It considers the design of a portion of an automobile door, shown in Figure Two. The door is obviously only a small part of a larger design. The example addresses the door's interior and external panels and an interior longitudinal safety beam. Doors typically involve a window mechanism plus other devices: lock, venting ducts and radio speakers, as well as an interior finish panel. These are not considered here.

Development for this aspect of the design takes place in three different departments: *Structures*, *Design* and *Process Planning,* shown across the top of Figure Three. The figure shows design actions in boxes and

the necessary precedence relations between actions as directed edges[1]. Precedences that include information availability and integrity state are shown in solid edges, those involving integrity states alone, i.e., that rules are satisfied, are shown as dashed. Each action is described with a short phrase. The state resulting from an action is shown as a filled circle and labeled ("c" followed by an index). The starting information is delivered to Design, which extracts the door data from the overall design and develops its details. We track the three elements: the outside and interior panels and the structural beam.

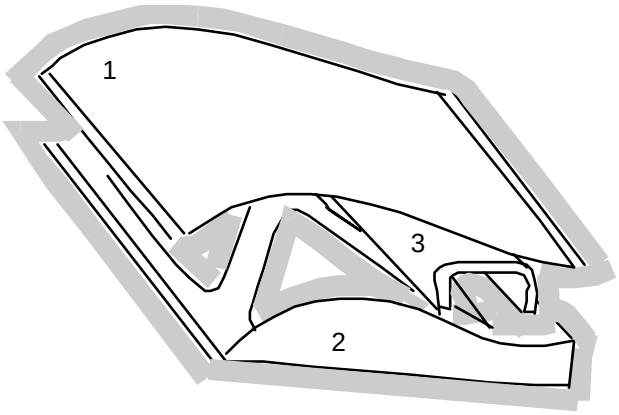

FIGURE TWO: Three parts studied in the example door panel design: (1) exterior panel, (2) interior panel, and (3) safety beam.

The two panels each receive two types of analysis. The first is structural deformation under various loads for flexure and bending. The other type of analysis is metal forming, involving the stamping of the parts using dies, anticipating metal deformation and springback. The beam's resistance to lateral crash loads is also analyzed. These analyses are made by the CAE department, using finite element analysis models. The structural analyses are done first because they are used in determining the gauge of metal used, which can affect the deformation during stamping. Process Planning then determines how the elements are to be fabricated. Last, schedules for each element and for the assembly of the door are defined.

The analysis actions are all applied to the design data, generated in $\phi 2$ and $\phi 3$. The structural analyses only check integrity conditions that are required by later actions. The metal deformation analysis generates modified shape definitions used in die design. The assembly, process plans and schedules generate new representations that are used in later actions.

The edges indicate necessary precedence relations and show that many of the actions can be undertaken in parallel. The inner and outer panels and the beam all may be analyzed in parallel, for both types of

---

[1] In conventional design areas, design development generally follows a predefined process. In innovative design, the process itself may not be well-defined beforehand and is structured as design proceeds.

analysis. After the elements' stamping dies have been defined, each may have their process plans generated in parallel. A single action may satisfy a set of rules, different subsets of which are used in later actions. This occurs in action $\phi 2$. Multiple states are identified, each defining different rules realized by the action that are used in later actions. The precedence relations shown are not those imposed by an organization, for example, as in [15], but rather capture the minimal precedences imposed by the information and integrity rules the actions require. Organizations will typically operate much more serially, because of organizational conventions and experience. Additional actions and precedences may be added to reflect organizational policies.

At any point of development, problems may arise. An analysis whose result is unsatisfactory requires iteration and a change in the output of some earlier action, which leads to iteration of successor actions. For example, if any analysis by the CAE department identifies problems, portions of the panels or beam must be redesigned, requiring re-doing most actions on the changed elements. Tasks that generate new information from existing, such as die design, may fail and lead to iteration, if features are found that are incompatible to the generation process.

After the design actions record the base design data, it becomes quite possible to use data incorrectly. The stamping die application might use data not corrected for metal springback. If the assembly test failed and door re-design is required, the re-analysis of the changed design may be forgotten. Errors in managing changes may only show up later, when the automobile goes into prototype production or worse, after an accident shows that some portion of the design did not perform as expected. Automated methods to guarantee that such errors will not occur are presented below[2].

III. REPRESENTATION OF INTEGRITY RULES

The following vocabulary will be used. In design, a *database management system (DBMS)* supports the definition of a *product schema* that is then assigned values by applications to describe a particular *design specification*. Correspondingly, a *data model* is here considered an abstraction of a DBMS, a *product model* is an abstraction of a product schema and a *design model* an abstraction of the fully loaded database. The function of the product model is to represent the state of the design model as it evolves throughout a design process. An *application* is external code that adds to, modifies or evaluates a design model. An *action* is a human activity that includes the execution of an application. An application may be applied to a design model multiple times and over varied sets of objects. We call each execution of an application a *operation instance*.

---

[2] Errors regarding analysis assumptions or due to errors of computation are not addressed here.

A design model consists of both a product model, defined as a set of class entities, and also instances of those classes that are assigned values[3]. A class entity is denoted E and a set of classes as {E}. An instance of entity class E_i is denoted e_i. and a set of instances {e}. Entity classes are organized into composition hierarchies corresponding to various assemblies and sub-assemblies. The components of an assembly are called the assembly's *parts* [7]. More formally, entity class E_i subsumes E_{ij} and E_{ijk} where *subsumption*, denoted ≥, indicates that all components that are members of E_i are also members of E_{ij}, or:

$$E_i \geq E_{ij} \geq E_{ijk} . \qquad (3.1)$$

Associated with the classes of a product model are various integrity rules, defined as constraints, where C_E is a constraint associated with entity class E. Here, constraints may be of two types: *variant* and *invariant*, Their possible values are:

Variant	Invariant	Meaning
True	True	constraint satisfied
False	(not allowed)	constraint not satisfied
Undefined	Undefined	data needed for evaluation missing
NULL	(not allowed)	state unknown - needs to be evaluated

In most database work to date, the focus of attention has been invariant constraints [4,7]. Here, the focus is on the variant constraints, whose allowed values are True, False, Undefined and NULL. True and False are values known by the design model, Undefined denotes that a constraint value cannot be computed because of missing data. NULL indicates a constraint whose value is unknown. The constraints C_E associated with an entity class E are inherited into the set of all instances $\{e_E\}$. Such constraint instances are denoted c_e. Each variant constraint instance has a separate value. Because constraints may have more than two alternative values, they are evaluated explicitly by a predicate, e.g. *state(c_e, True)* [4].

Constraints associated with some class E_i are also associated with classes that E_i subsumes. Thus

---

[3] Classes might be depicted as relations in a relational data model or object classes in an object-oriented data model.

[4] Alternatively, constraint values may be depicted using a four-valued logic. This approach has not been explored.

$$\forall \alpha \mid \alpha \in \{C_{E_i}\} \Leftrightarrow \alpha \in \{C_{E_{ij}}\} \quad (3.2)$$

where $\{C_{E_i}\}$ is the set of constraints associated with class E_i. Constraint definitions include their parameter types, while constraint instances include a set of entity instances as arguments that must match the parameter types of the corresponding constraint definition.

Constraints may or may not have a function body. Those without a function body serve as *shadows* for an external application and their state is treated as a flag by the application interface, which sets the constraint instance states corresponding to the operation taken. Those that have a function body are executable and derive the constraint's state when applied to its arguments.

A composed entity can include both integrity constraints associated with its own definition, constraints across the set of parts it encompasses, as well as the constraints within its parts. The constraints across parts are able to define well-formedness conditions for a composition, such as for a circuit or truss. Constraints can also represent performance rules defining the properties of a composed object in terms of the properties and composition of its parts, e.g. the maximum force in the truss or the timing of the circuit. This definition of composition is similar to most other data models of design information [7], [22], with the extension being the inclusion of the corresponding integrity constraints. There are many subtleties in representing composition hierarchies not discussed here, but the presentation is sufficient for the issues to be developed.

Each application acts upon a set of classes, here called its *transaction set (TS)*. A TS has two subsets which are not necessarily inclusive or exclusive of each other: a *readset* $\{E\}^R$ and a *writeset* $\{E\}^W$, corresponding to the set of classes read as input to the application and the set of classes written as output upon successful completion of the application, where $TS \equiv \{E\}^R \cup \{E\}^W$.

Associated with the TS of an application are two sets of integrity constraints, denoted $\{C\}_{E_b}{}^B$ and $\{C\}_{E_a}{}^A$. They are called the *before-constraint set,* $\{C\}_{E_b}{}^B$, and the *after-constraint set* $\{C\}_{E_a}{}^A$ of the application. A constraint has a scope, corresponding to the classes accessed in its evaluation. We restrict the scope of a constraint to be those classes in its parameter list. The before-constraint set may be of any scope but is assumed to include the readset $\{E\}^R$ as arguments. The after-constraint set is a subset of the constraints whose scope is both the readset and writeset entities:

$$\{E\}_a \geq (\{E\}^R \cup \{E\}^W) \quad (3.3)$$

Taken together, a design application can be represented with the following formulation:

$$\Phi_i = (\{E\}_i^R, \{E\}_i^W, \{C\}_i^B, \{C\}_i^A), \tag{3.4}$$

Any representation of the information in an application would include the readset and writeset. What is new here is the two constraint sets. The constraints represent the two types of integrity relations existing between operations:

(1) before-constraint rules apply to the readset instances. These rules correspond to the well-formedness rules applied to the data input to the operation. Examples include the connectedness of graphs in circuit simulation or structural analysis, or spacing requirements in circuit layout. They also may be rules with global scope, such as timing conditions in a circuit or that no spatial interferences exist for the solids in the readset.

(2) after-constraint rules correspond to the mapping between input and output within the application. These rules correspond to assertions that might be made to define the operations' logical correctness. These rules apply across both the readset and writeset.

Readset entities are the classes read by the application. The writeset is the set of entity classes modified in the application and may overlap with the readset classes.

Design applications may be executed multiple times. Each application invocation, defined as above, is called an *operation instance* and denoted ϕ_{ij}, which is an instance of application Φ_i Each *operation instance* may access and modify a set of entity instances that may or may not overlap with the sets affected by other *operation instances* from this or other applications. *Operation instances* are denoted:

$$\phi_{ij} \equiv (\{e\}_{ij}^R, \{e\}_{ij}^W, \{c\}_{ij}^B, \{c\}_{ij}^A), \tag{3.5}$$

where $\{e\}_{ij}^R$ are instances of $\{E\}_i^R$,

$\{e\}_{ij}^W$ are instances of $\{E\}_i^W$,

$\{c\}_{ij}^B$ are instances of $\{C\}_i^B$,

and $\{c\}_{ij}^A$ are instances of $\{C\}_i^A$.

When an *operation instance* is undertaken, new entity instances are created, deleted or modified. In addition, the state of constraint instances change. It is both the entity instances and the constraint instances that determine the state of the design model and manage the communication between one *operation*

instance and others. The fundamental point here is that in engineering it is not meaningful to assert the absolute correctness of some data. Correctness is always relative to the integrity constraints applied. These may change over time. Thus integrity constraints are relied on to directly define the adequacy of data for executing particular applications. Given this definition of design operations, it is clear that constraints are satisfied incrementally by a sequence of *operation instances*. Each operation typically adds to the set of integrity rules already satisfied.

II.A. The Logic of Design Operations

Given this representation of an application and its execution as an *operation instance*, the use of operations and the structure between them can be defined. An application has a product model interface, defined in terms of the entity classes accessed for reading and writing. Some *operation instances* may start de novo, with an empty readset; for example a CAD program used to define a part shape or building floorplan. In the invocation of an *operation instance* having a non-empty readset, the operation may read all instances of the readset classes and write to all instances of the writeset classes. More often, however, an *operation instance* will apply to a subset of the instances of all those possible.

A successful *operation instance* is one whose entity instances have all before-constraints and all after-constraints set to True.

> ***Definition***: a *successful operation instance* ϕ_x is one in which all before- and all after- constraint instances are True. That is:

$$\forall\, c_y, \forall\, c_z \mid \phi_x \equiv (\{e\}_x^R, \{e\}_x^W, \{c\}_x^B, \{c\}_x^A) \wedge (c_y \in \{c\}_x^B \wedge \mathit{state}(c_y, \mathit{True})) \wedge$$
$$(c_z \in \{c\}_x^A \wedge \mathit{state}(c_z, \mathit{True})) \Leftrightarrow \mathit{successful}(\phi_x) \qquad (3.6)$$

An *operation instance* for which any before-constraints are not satisfied is considered *infeasible*. However, the state of any before-constraint is based on additional conditions: that the previous before-constraints of earlier operations that set the current operation's before-constraints are also True, and the before-constraints of the operation that set those before-constraints are True, and so on recursively. This condition corresponds to the assumption that the *operation instances* were carried out in precedence order, beginning with operations without before-constraints, then undertaking operations for which the initial ones satisfied their before-constraints, then taking later operations whose before-constraints were satisfied previously and so on. Global integrity fails if the integrity states for the input data used in deriving the output data are not True, or the integrity states required to derive that data, and so on. For example, a finite element structural model includes as its input shape and loads. If the inputs used to derive the loads are inconsistent with the integrity relations used in their derivation, these conditions invalidate the before-constraints on the loads and thus the analysis results.

A constraint's evaluation defines integrity only locally. Additional conditions are required to guarantee that a constraint's integrity state is valid globally.

> ***Definition:*** *predecessor*(ϕ_x, ϕ_y) is the relation between two *operation instances* such that the after-constraints of ϕ_x intersect with the before-constraints of ϕ_y.
>
> $$\exists c_q \mid \phi_i \equiv (\{e\}_i^R, \{e\}_i^W, \{c\}_i^B, \{c\}_i^A) \wedge \phi_j \equiv (\{e\}_j^R, \{e\}_j^W, \{c\}_j^B, \{c\}_j^A) \wedge (c_q \in \{c\}_i^A \wedge c_q \in \{c\}_j^B) \Leftrightarrow predecessor(\phi_i, \phi_j) \quad (3.7)$$

Predecessor defines a relationship between one *operation instance* and others that have to be *successful* for the one to assert its after-constraint states globally. Concatenated, the *predecessor* relation defines a sequence of relations from any one operation back to an operation that has no before-constraints. In the worst case, this is the initial design operation. We call an operation that has no before-constraints an *open operation*. The *predecessor* relation can be applied with its arguments reversed, resulting in a *successor* relation, defining those *operation instances* that rely on the current one as a predecessor.

> ***Definition****:* A constraint instance c_x is *globally_valid* only if its state is assigned by an *operation instance* that is *successful* and that a sequence of *successful* operations exist, defined by the *predecessor* relation from all before-constraints of the state assigning operation to an *open operator*:

$$\exists \phi_i, \exists \phi_j, \forall c_y \mid \phi_i \equiv (\{e\}_i^R, \{e\}_i^W, \{c\}_i^B, \{c\}_i^A) \wedge \phi_j \equiv (\{e\}_j^R, \{e\}_j^W, \{c\}_j^B, \{c\}_j^A) \wedge c_x \in \{c\}_i^A \wedge predecessor(\phi_i,\phi_j) \wedge successful(\phi_i) \wedge c_y \in \{c\}_i^B \wedge globally\_valid(c_y) \Leftrightarrow globally\_valid(c_x) \quad (3.8)$$

Globally_ valid does not mean that a constraint instance evaluates to True. Rather it means that the state of the constraint instance was determined when the database was in a condition allowing a valid evaluation to be made. The constraint state may be any value other than NULL. *Globally_valid* constraint instances are required for meaningful communication between operations.

III.B. Operation Modeling

We can apply these concepts to the example problem. In the example, eleven entity classes and twenty-three different instances are involved, as listed below. Three of the entity classes are output from analyses, used only for review and not used later. The entity instances defined or modified within each *operation instance* are also shown in Figure Three:

ENTITIES	INSTANCES
E1. overall automobile layout dataset	e1,2,3,..... datasets for all outer body panels
E2. door body panel design	e4. initial outer panel data
	e5. inner panel design data
	e5.1. outer panel design data
E3. panel structural data (for review)	e6. inner panel structural model data
	e7. outer panel structural model data
E4. panel stamping data	e8. inner panel deformation dataset
	e9. outer panel deformation dataset
E5. beam design data	e10. safety beam design data
E6. beam structural model dataset (for review)	e11. safety beam structural data
E7. stamping die model	e12. outer panel die design data
	e13. inner panel die design data
E8. beam stamping die model	e14. safety beam stamping design data
E9. assembly plan data model	e15. door assembly test data
E10. process plan data model	e16. outer panel process plan
	e17. inner panel process plan
	e18. safety beam process plan
	e19. overall door assembly process plan
E11. fabrication schedule datamodel	e20. outer panel fabrication schedule
	e21. inner panel fabrication schedule
	e22 safety beam fabrication schedule
	e23. overall door assembly schedule

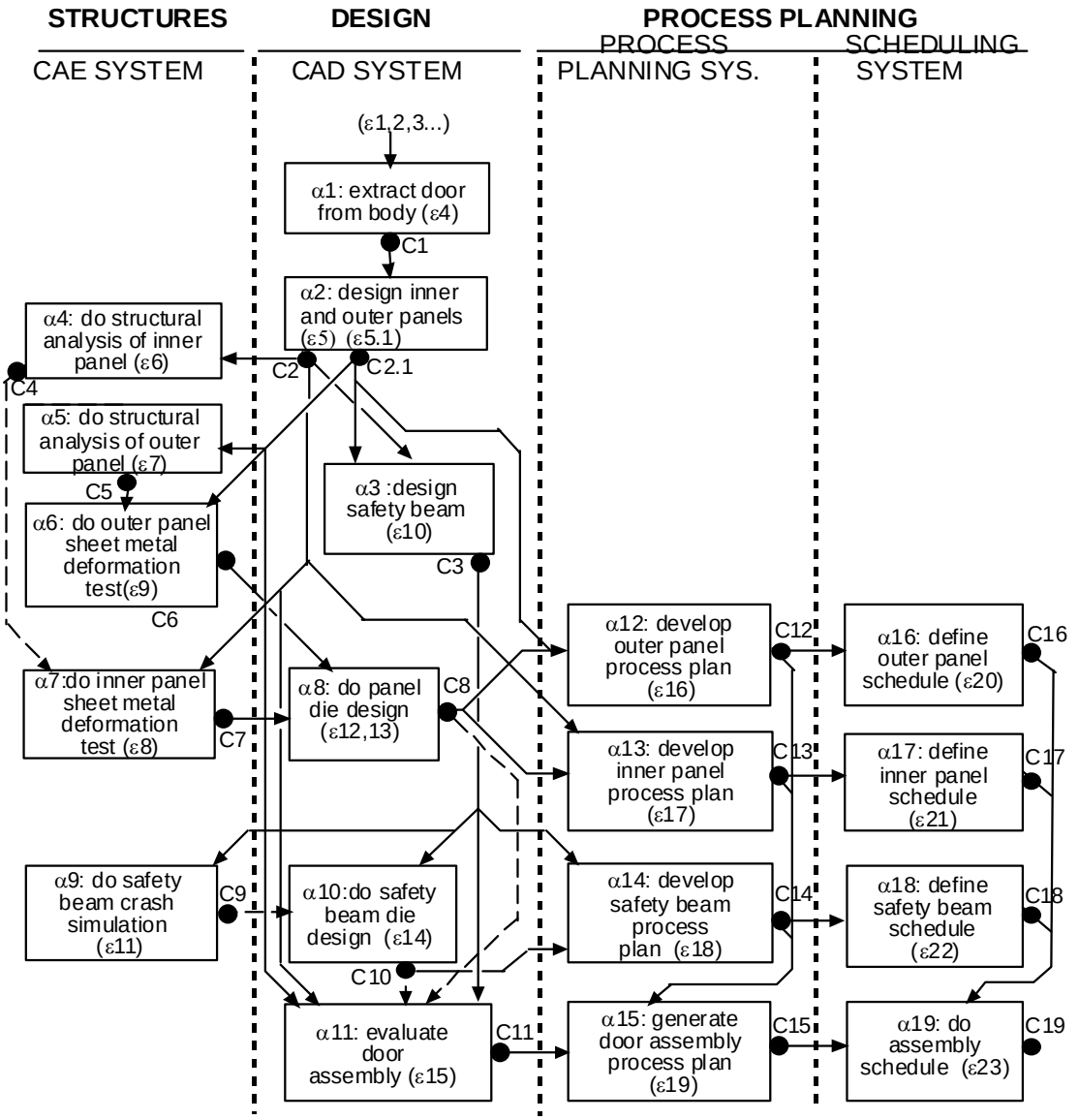

FIGURE THREE: Design operations, states and precedences for the example task. Precedence are relations shown with directed lines.

The design actions incorporating the applications shown in Figure Three can be represented as shown in Figure Four. Figure Four represents each *operation instance* according to eq.(3.5). In some cases, an operation generates new data and the integrity constraints denote the existence of that data. For example, operation a_2 generates the panel designs e_5 and $e_{5.1}$, which are used in $\{\phi_{3,4,5,6,7,11,12}\}$ and ϕ_{13}. In other cases, the operation applies to one or several readset instances and generates analysis results that are not used elsewhere; $\{\phi_{4,5,9}\}$ are examples of this case. Some *operation instances* may modify their readset resulting in the same entity instances being in their writeset. Most operations read data that is shared, thus also read, by multiple operations.

$\phi_1=(\{e\}_{1,2,3...}{}^{R},\{e\}_4{}^{W},\varnothing^{B},\{c_1\}^{A})$ (extract door design from body)

$\phi_2=(\{e\}_4{}^{R},\{e\}_{5,5.1}{}^{W},\{c_1\}^{B},\{c_{2,2.1}\}^{A})$ (design inner and outer door panels)

$\phi_3=(\{e\}_{5,5.1}{}^{R},\{e\}_{10}{}^{W},\{c_{2,2.1}\}^{B},\{c_3\}^{A})$ (develop design of safety beam)

$\phi_4=(\{e\}_5{}^{R},\{e\}_6{}^{W},\{c_2\}^{B},\{c_4\}^{A})$ (do inner panel structural analysis)

$\phi_5=(\{e\}_{5.1}{}^{R},\{e\}_7{}^{W},\{c_{2.1}\}^{B},\{c_5\}^{A})$ (do outer panel structural analysis)

$\phi_6=(\{e\}_{5.1}{}^{R},\{e\}_9{}^{W},\{c_{2.1,5}\}^{B},\{c_6\}^{A})$ (do outer panel sheet metal deformation test)

$\phi_7=(\{e\}_5{}^{R},\{e\}_8{}^{W},\{c_{2,4}\}^{B},\{c_7\}^{A})$ (do inner panel sheet metal deformation test)

$\phi_8=(\{e\}_{8,9}{}^{R},\{e\}_{12,13}{}^{W},\{c_{6,7}\}^{B},\{c_{8,8.1}\}^{A})$ (do inner and outer panel die designs)

$\phi_9=(\{e\}_{10}{}^{R},\{e\}_{11}{}^{W},\{c_3\}^{B},\{c_9\}^{A})$ (do safety beam crash simulation)

$\phi_{10}=(\{e\}_{10}{}^{R},\{e\}_{14}{}^{W},\{c_{3,9}\}^{B},\{c_{10}\}^{A})$ (do safety beam die design)

$\phi_{11}=(\{e\}_{5,5.1,10}{}^{R},\{e\}_{15}{}^{W},\{c_{2,2.1,3,8,8.1,10}\}^{B},\{c_{11}\}^{A})$ (eval. door panels and safety beam assembly)

$\phi_{12}=(\{e\}_{5.1,12}{}^{R},\{e\}_{16}{}^{W},\{c_{2.1,8}\}^{B},\{c_{12}\}^{A})$ (develop outer panel process plan)

$\phi_{13}=(\{e\}_{5,13}{}^{R},\{\{e\}_{17}{}^{W},\{c_{2,8.1}\}^{B},\{c_{13}\}^{A})$ (develop inner panel process plan)

$\phi_{14}=(\{e\}_{10,14}{}^{R},\{e\}_{18}{}^{W},\{c_{3,10}\}^{B},\{c_{14}\}^{A})$ (develop safety beam process plan)

$\phi_{15}=(\{e\}_{15,16,17,18}{}^{R},\{e\}_{19}{}^{W},\{c_{11,12,13,14}\}^{B},\{c_{15}\}^{A})$ (generate door assembly process plan)

$\phi_{16}=(\{e\}_{16}{}^{R},\{e\}_{20}{}^{W},\{c_{12}\}^{B},\{c_{16}\}^{A})$ (define manuf.schedule for outer panel)

$\phi_{17}=(\{e\}_{17}{}^{R},\{e\}_{21}{}^{W},\{c_{13}\}^{B},\{c_{17}\}^{A})$ (define manuf.schedule for inner panel)

$\phi_{18}=(\{e\}_{18}{}^{R},\{e\}_{22}{}^{W},\{c_{14}\}^{B},\{c_{18}\}^{A})$ (define manuf. schedule for safety beam)

$\phi_{19}=(\{e\}_{19,20,21,22}{}^{R},\{e\}_{23}{}^{W},\{c_{15,16,17,18}\}^{B},\{c_{19}\}^{A})$ (do manuf.schedule for compl.door assembly)

FIGURE FOUR: Formal representation of the design operations and precedence relations shown in Figure Three.

The overall arrangement of design operations is that one set of *operation instances* define entities and/or satisfy integrity conditions within its after-constraints that are then required as readset instances and before-constraints of other operations. The *operation instances* incrementally build up the design model and its integrity along a frontier, behind which all constraint instances are *globally_valid* and True and in front of which the integrity states are NULL or Undefined. This frontier expands as design proceeds. Design is complete when a state of total integrity is achieved for all instantiated constraints.

IV. OPERATIONS ON THE PRECEDENCE STRUCTURE OF APPLICATIONS

The example presented here is part of a much larger engineering process, involving possibly hundreds of applications within dozens of departments. In the future, these departments may be physically dispersed so

that informal management methods will not be effective and formal methods will be required. It is assumed that all engineering data is electronically accessible from a shared server or distributed set of servers. Thus we cannot assume that the design department is in proximity to the structures or process planning departments.

Suppose that at some point in the design process (using the acttions defined in Figure Three), the door design has been extracted, structural and sheetmetal deformation and springback analyses of both panels have been carried out and approved. Dies for all three parts have been defined and the process plan for the outer and inner panels are defined.. The assembly test, however, indicates that the door cannot be assembled as designed, using current assembly tools. The constraint state result of all these actions on the door design is shown in Table One.

c1= *True*,	c2= *True*,	c2.1= *True*	c3= *True*,	c4= *True*,	c5= *True*,
c6= *True*,	c7= *True*,	c8= *True,*	c8.1= *True*	c9= *True*,	c10= *True*,
c11= *False,*	c12= *True*,	c13= *True*,	c14= *NULL*,	c15= *NULL*,	
c16= *NULL*,	c17= *NULL*,	c18= *NULL*,	c19 =*NULL*.		

TABLE ONE: The state of the design after the example sequence of operations.

IV.A. What Operations Can be Executed?

The proposed method of communication suggests that any *operation instance* is feasible for which all of its before-constraints are True.

$$\forall\ c_p \mid \phi_i = (\{e\}_i^R, \{e\}_i^W, \{c\}_i^B, \{c\}_i^A), (c_p \in \{c\}_i^B) \wedge \text{state}(c_p, \textit{True}) \Leftrightarrow \textit{feasible}(\phi_i) \qquad (4.1)$$

Given the design state shown in Table One, and the *operation instances* defined in Figure Four, operations $\{\phi_{1\text{-}14,16,17}\}$ would evaluate to True.

All operations previously completed as well as all those at the frontier would be flagged as feasible. This indicates that any earlier operation can be iterated as well as new operation positioned along the frontier. In order to distinguish the newly feasible *operation instances*, we add another predicate to eliminate operations that are in a *successful* state:

$$\forall\ \phi_x \mid \text{feasible}(\phi_x) \wedge \neg\, \textit{successful}(\phi_x) \ \Leftrightarrow\ \textit{newly\_feasible}(\phi_x) \qquad (4.2)$$

newly_feasible selects *operation instances* whose before-constraints are True and whose after-constraints are not True. Any of these operations may be executed, and if successfully completed, record its effects according to the definition of successful operation, eq.(3.6). Such an operation advances the frontier

defined by the *newly_feasible* operations by adding their writesets and setting their after-constraints to True. Given the design model state shown in Table One, the *newly_feasible* operations are $\{\phi_{11,14,16,17}\}$.

A designer may believe that an *operation instance* should be newly feasible, but it is not. In such cases, a useful query is to identify which operations must be successfully completed in order for some given operation ϕ_i, that is not feasible, to become newly feasible. Given, $\neg$feasible (ϕ_i) then:

$$\forall\phi_x \; \forall\, c_q \; \forall\, c_p \mid \phi_i = (\{e\}_i^R, \{e\}_i^W, \{c\}_i^B, \{c\}_i^A), (c_q \in \{c\}_i^B \wedge \text{state}(c_q, \neg \mathit{True}) \Rightarrow c_q \in Q) \wedge$$
$$(\phi_x = (\{e\}_x^R, \{e\}_x^W, \{c\}_x^B, \{c\}_x^A), (c_q \in \{c\}_x^A \wedge (c_p \in \{c\}_x^B \wedge \text{state}(c_p, \neg \mathit{True}) \Rightarrow c_p \in Q) \qquad (4.3)$$

This is a recursive query. It identifies in Q all operations whose after-constraints are those of the desired *operation instance* ϕ_i but not True, and all other predecessor operations to an earlier identified operation that are not True.

IV.B. Communication Between Operations

Given the structure defined for a *globally_valid* constraint instance, eq.(3.8), there are several methods by which an operation can be checked to determine its feasibility. One method would be to check if all its before-constraints are *globally_valid* and True, applying the test recursively as defined. This is likely to result in the same precedence sequences being checked many times for different operations. Another method would be for each operation, when it executes, to propagate its effects so that the design model is maintained in a *globally valid state*.

Maintaining a design model to be in a *globally valid state* requires that the constraint instances whose states are True or False are *globally_valid*. No such condition is required for constraint instances that are NULL or Undefined. When an *operation instance* successfully completes, it stores its writeset instances (if not defined previously) and records the constraint instances the operation satisfies. These actions can result in two different conditions:

1. the data for some entity instances are written for the first time, possibly allowing some constraints that previously were Undefined, to now be evaluatable;

2. the values for some entity instances are re-written over previous values. The previous values may have been accessed in the evaluation of other constraints not part of this *operation instance*. The value states of these other constraints can no longer be guaranteed and must be set to NULL.

It is assumed that operations will be executed incrementally, possibly over a long period of time. It is common that multiple constraints be associated with a single design variable. Thus it is likely that some new *operation instance* will modify a variable after it has been set to satisfy other design constraints. Maintaining a design model in a *globally_valid* state requires that an *operation instance* that modifies the variables accessed by a constraint instance set the constraint instance to NULL, forcing the re-evaluation of all other constraint instances whose parameter values have changed. Previously executed operations that now have NULL before-constraints are no longer successful and thus also must have their after-constraints set to NULL. This method of communication between *operation instances* updates all affected constraint instances at the time any action is taken. Thereafter, for a design model in a globally valid state, any operation's feasibility can be determined by only checking its before-constraints. These definitions identify strong conditions that require maintenance whenever an *operation instance* is executed.

The *operation instance* just successfully completed is ϕ_i, and the instance set $\{e\}_k^W$ are those written. A predicate is introduced, called the *scope* of a constraint instance. It is defined as the set of entity instances in the constraint instance's argument list, e.g., $\forall e_k \mid c_p \equiv f(\{e\}_{pj}) \wedge e_k \in \{e\}_{pj} \Leftrightarrow scope(c_p, e_k)$. Then:

$$
\begin{aligned}
&\text{(a.) } \forall e_p, \forall c_q \mid \phi_i \equiv (\{e\}_i^R, \{e\}_i^W, \{c\}_i^B, \{c\}_i^A) \wedge e_p \in \{e\}_i^W \wedge scope(c_q, e_p) \Rightarrow \\
&\qquad\qquad (state(c_q, NULL) \wedge c_q \in Q) \\
&\text{(b.) } \forall \phi_j, \forall c_q, \forall c_r \mid \phi_j \equiv (\{e\}_j^R, \{e\}_j^W, \{c\}_j^B, \{c\}_j^A) \wedge (c_q \in Q \wedge c_q \in \{c\}_j^B) \wedge \\
&\quad (c_r \in \{c\}_j^A \wedge state(c_r, True) \Rightarrow (state(c_r, NULL) \wedge c_r \in Q)) \vee (state(c_r, False) \Rightarrow state(c_r, NULL)) \\
&\text{(c.) } \forall c_s \mid c_s \in \{c\}_i^A \Rightarrow state(c_s, True) \qquad (4.4)
\end{aligned}
$$

The first step, (a.) identifies all constraints that access the changed instances and sets them to NULL. It includes not only the constraints satisfied by this *operation instance*, but also all others that have evaluated the changed data. The second step, (b.) checks all successor operations to the changed ones and, if they include after-constraints whose value is True or False, it sets them to NULL. Recursive application of the successor relation is applied to the True constraint instances. The last step, (c.) sets the after-constraints of the original successful *operation instance* to True. With this method of updating, all data modified by an operation has all constraints that access it set to NULL and the conditions for a globally valid database state are maintained. In the above, the temporary set Q is used to delimit the set of constraints to be considered. Updates must be done in the order (a,b,c). Many of the constraint instances updated in (c.) are accessed in the writeset defined in (eq.4.4(a)). A different order would either result in the successful completion of the *operation instance* not being recorded, e.g., the after-constraints not being set to True, and/or the effects of the changed values in (a.) not being recorded. An example of the use of this update mechanism follows.

IV.C. Iteration of Operations

An important use of integrity constraints is for dealing with issues arising from design iteration. When some design *operation instance* cannot be successfully completed, the immediate cause is that the current combination of instances in the failed operation's readset do not allow generation of a writeset that can satisfy the operation's after-constraints[5]. Which readset instances to modify and what application to use to effectuate the modification involves backtracking.

Whatever application is iterated, it will assign different values to some part of its writeset than those written previously. Even though a successful application iteration results in the after-constraints remaining True, the changed data may violate other integrity constraints that reference the modified data. Since the constraint logic over all operations is serial and conjunctive, the most specific response possible to operations applied iteratively would be if each operation explicitly represented the constraints it possibly violates upon completion, as well as the constraints it satisfies. It could then set the potentially failed after-constraints to False and propagate them forward, in order to maintain global consistency. However, such a capability would require each application to have global knowledge of all constraints, including those associated with all the other operators. This would be extremely difficult in the context of dynamically changing operations, as described in Section I. In practice, it is desirable for iterated operations to be managed using only local knowledge.

Without knowledge of the specific constraints violated by an operation, a coarser propagation of iteration effects is defined. Any iterated application that writes new values to its writeset entity instances may violate other constraints that access them. (Eq. 4.4(a)) sets these constraint instances to NULL, forcing them to be re-evaluated. Instead of treating iterated applications as special, the effects of ANY successfully completed *operation instance* is elaborated to deal with updating required to maintain the design model in a globally consistent state. All *operation instances* that successfully complete and assign values to instances also reset the values to NULL of the all constraints whose scope includes the writeset instances. Then, it accesses all *operation instances* that do not satisfy the conditions of a successful operation and sets to NULL their after-constraints (eq. 4.4(b)). Last, the constraint instances satisfied by the *operation instance* are set to True (eq. 4.4(c)).

In our example in Figure Three, evaluation of the door assembly (ϕ_{11}) failed. Failure here requires changing the design of the outer or inner panels or the structural beam. Regardless of how the re-design is undertaken, it is important for the changes to be communicated to affected operations and to maintain the

---

[5] Infeasibility is only one cause of iteration in design. Others include recognition of better performing alternatives, or poor performance in terms of some externality not included in the formal design model.

design model in a globally valid state. Let us suppose that resolution of the conflict resulted in iterating ϕ_2 and denoting the iterated operation instance as $\phi_{2,1}$ and updating entity instance e_5, the inner panel. e_5 is within the scope of c_2 which would be set to NULL, based on (eq. 4.4(a)) and the *operation instance* would be defined as $\phi_{2,1} \equiv (\{e\}_4^R, \{e\}_5^W, \{c\}_1^B, \{c\}_2^A)$. Then (eq. 4.4(b)) will set to NULL all operations' after-constraints that are no longer successful and that have been set to True or False, $\{c_{3,4,7,8,9,10,11,12,13,14,15,16,17,18,19}\}$. Last, c_2 is set to True, as the result of (eq. 4.4(c)). The state representation after this iteration would be as shown in Table Two. The iteration of $\phi_{2,2}$ had a different writeset than the original *operation instance*, ϕ_2 . The outer panel analyses are still valid. The beam design must be re-considered, however, because it was based on the earlier inner panel design and all subsequent *operation instances* must be repeated.

c1= *True*,	c2=*True*,	c2.1= *True,*	c3= *NULL*,	c4= *NULL*	c5= *TRUE*,
c6= *TRUE*,	c7= *NULL*,	c8= *NULL,*	c8.1= *NULL,*	c9= *NULL*,	c10= *NULL*,
c11= *NULL*,	c12= *NULL*,	c13= *NULL*,	c14= *NULL*,	c15= *NULL*,	
c16= *NULL*,	c17= *NULL*,	c18= *NULL*,	c19 =*NULL*.		

TABLE TWO: The state of the design after iterating ϕ_2 and updating instance e_5.

The effect of coarsely defined operations can be seen in this iteration. *Operation instance* ϕ_8, panel die design, groups the definition of both inner and outer panels. As a result, all before-constraints apply to both panel designs, with the result that a change to the inner panel invalidates the outer panel die design. A better set of *operation instance* definitions would separate the two operations, resulting in only invalidating the inner panel die design. This change would also leave as valid the outer panel process plan, ϕ_{12}. This issue is taken up in Sec. IV.E.

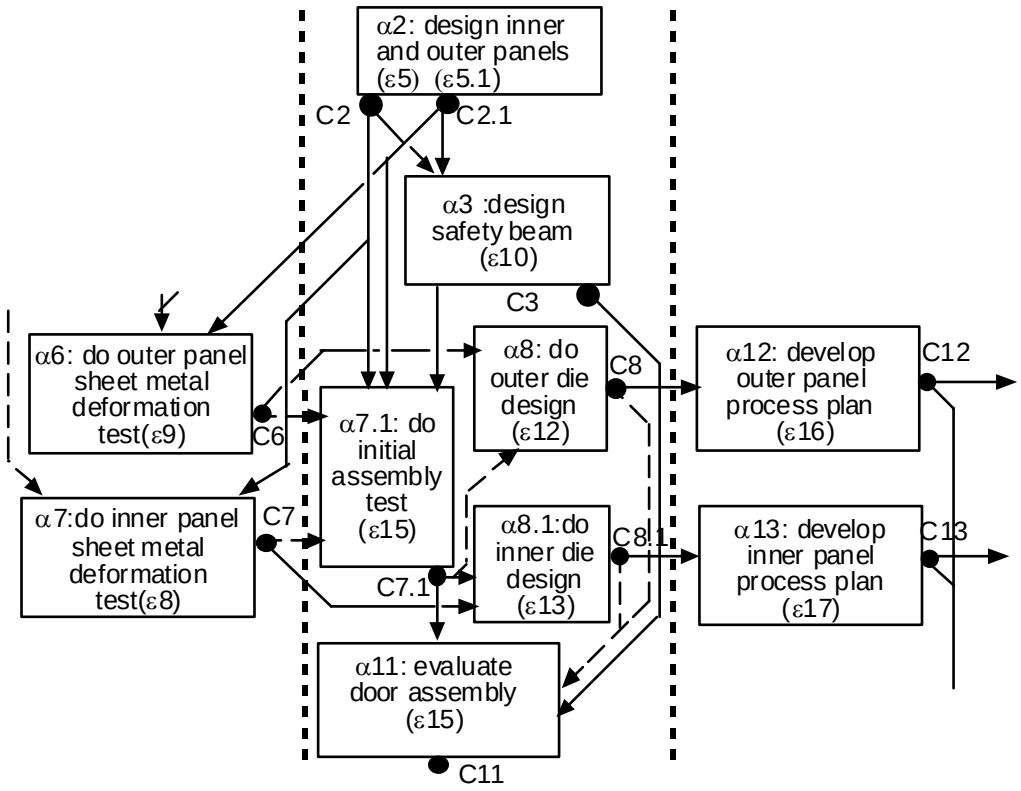

FIGURE FIVE: A revised design sequence and set of operations incorporating a preliminary door assembly test and separate operations for die design.

IV.D. Changing Operations and the Precedence Relations Among Operations

In the representation of applications and operations presented here, each is defined independently of the others. What is required is that an *operation instance* specify the readset and writeset instances it requires and the before-constraint conditions and after-constraints accomplished by the *operation instance* that are possibly used by others. Because precedence relations are abstracted, new operations need not know what other specific *operation instances* have created the integrity state it requires for proper execution.

Changes to a design development plan are frequently required, because of the introduction of new technologies or tests, or in response to inadequate earlier planning. Such capabilities are especially required for innovative or creative design. An example modification within part of the process shown in Figure Three is presented in Figure Five. This change might be defined in response to concern about the rejection of door designs in the assembly test, ϕ_{11}, after dies have been defined. Die generation is an expensive part of production setup. In the revision, an assembly test for the door is added before the final door assembly approval is made, allowing the process plans for the inner and outer elements to be defined with more confidence that they will not be later rejected. The literature indicates that Japanese car makers begin die making before all testing of the panel design is complete, thereby significantly reducing total development

time [5]. This previous design process reflected such parallelism. The proposed change reduces the risk of error in such a parallel process. In addition, the change treats the stamping die definition for the inner and outer door panels separately. This change is offered as an example of possible evolution within a product model. The effected *operation instances* and transitions are:

$\phi_{7.1}=(\{E_{5,5.1,10}\}^{R},\{E_{15}\}^{W},\{c_{2,2.1,6,7,10}\}^{B},\{c_{7.1}\}^{A})$(do initial door assembly test)

$\phi_{8}=(\{E_{5.1}\}^{R},\{E_{12}\}^{W},\{c_{2.1,7.1}\}^{B},\{c_{12}\}^{A})$ (do outer panel die design)

$\phi_{8.1}=(\{E_{5}\}^{R},\{E_{13}\}^{W},\{c_{2,7.1}\}^{B},\{c_{13}\}^{A})$ (do inner panel die design)

$\phi_{11}=(\{E_{10,15}\}^{R},\{E_{15}\}^{W},\{c_{3,7.1,8.1}\}^{B},\{c_{11}\}^{A})$ (evaluate assembly of door panels and safety beam)

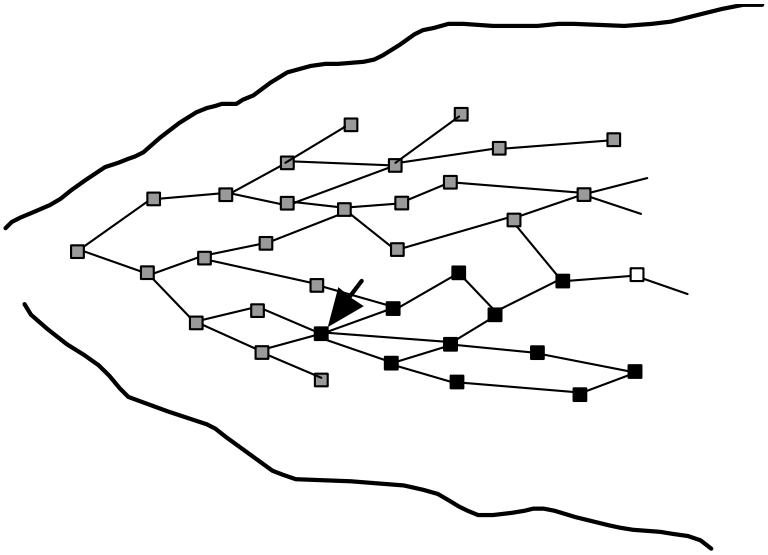

FIGURE SIX: A application sequence, as part of a design process, proceeding from left to right. Successful ones are shown filled in greyscale. The white application is infeasible and backtracking results in the flagged operation being iterated. As a result, the applications filled in black are set to NULL for re-evaluation.

Two operations have been revised and two new ones added. With these changes, communication between the existing and new operations is provided. It can be seen that integrity constraints provide a general representation of the precedence structure of *operation instances* and allows identification of when operations may be executed. It should be noted, that the precedence graph need never be drawn or fully conceptualized. The *operation instances* can be defined with only local knowledge about what other operations they must communicate with. Associated with these changes in precedence relations would be corresponding changes in interfaces to the design model and translators.

IV.E Iterations at Varied Levels of Granularity

Iterating an application involves executing a new instance of an application executed previously with some or all of the entity instances used as arguments being the same as the previous instance. The effect of

iteration is to propagate forward and set to NULL all constraints that accessed the modified data or that relied on a constraint that is now set to NULL to derive other data. Conceptually, the effect is to define a cone of NULL values that is oriented toward and intersects the frontier of *newly feasible* operations. Such behavior is shown diagrammatically in Figure Six. The application shown as a white filled box is infeasible and the application noted with an arrow is iterated as a new *operation instance*. As a result of the new value assignments, all the application instances filled in black are set to NULL, as a result of (eq. 4.4), forcing them to be re-evaluated.. In practice, the effects of this forward propagation is conservative, setting to NULL all constraints that can possibly be affected by the change. If operations and their corresponding constraints are defined coarsely, then the forward propagation will define a wide cone that flags as NULL constraints that do not require change. Integrity management is more effective to the degree that it flags as NULL just those constraints that may actually require their data to be modified to satisfy the effects of the iteration.

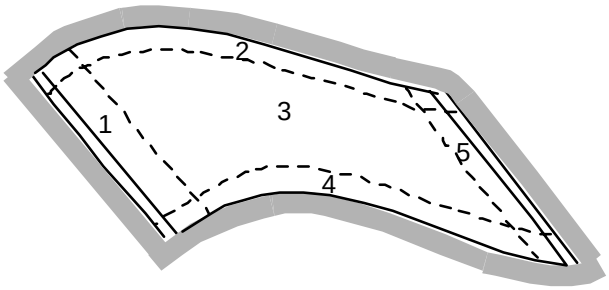

FIGURE SEVEN: An example partitioning of a design element to support change propagation at a finer level of granularity.

One way of narrowing the effects of iteration is by more precisely defining the nature of each operation. The level of composition to which an operation applies can be called its *granularity*. Propagation of changes operate at the level of the object instances in the scope of the operator's before- and after-constraints. If *operation instances* and their corresponding constraints can have their scope defined at a more detailed level of granularity, the effects of propagation will be more localized. Structural analysis, for example, may be applied to a complete structure, to an assembly or to a single component. An operation may be initiated that has as its input scope a coarse level of granularity, while the actual change executed may be at a very detailed level of granularity.

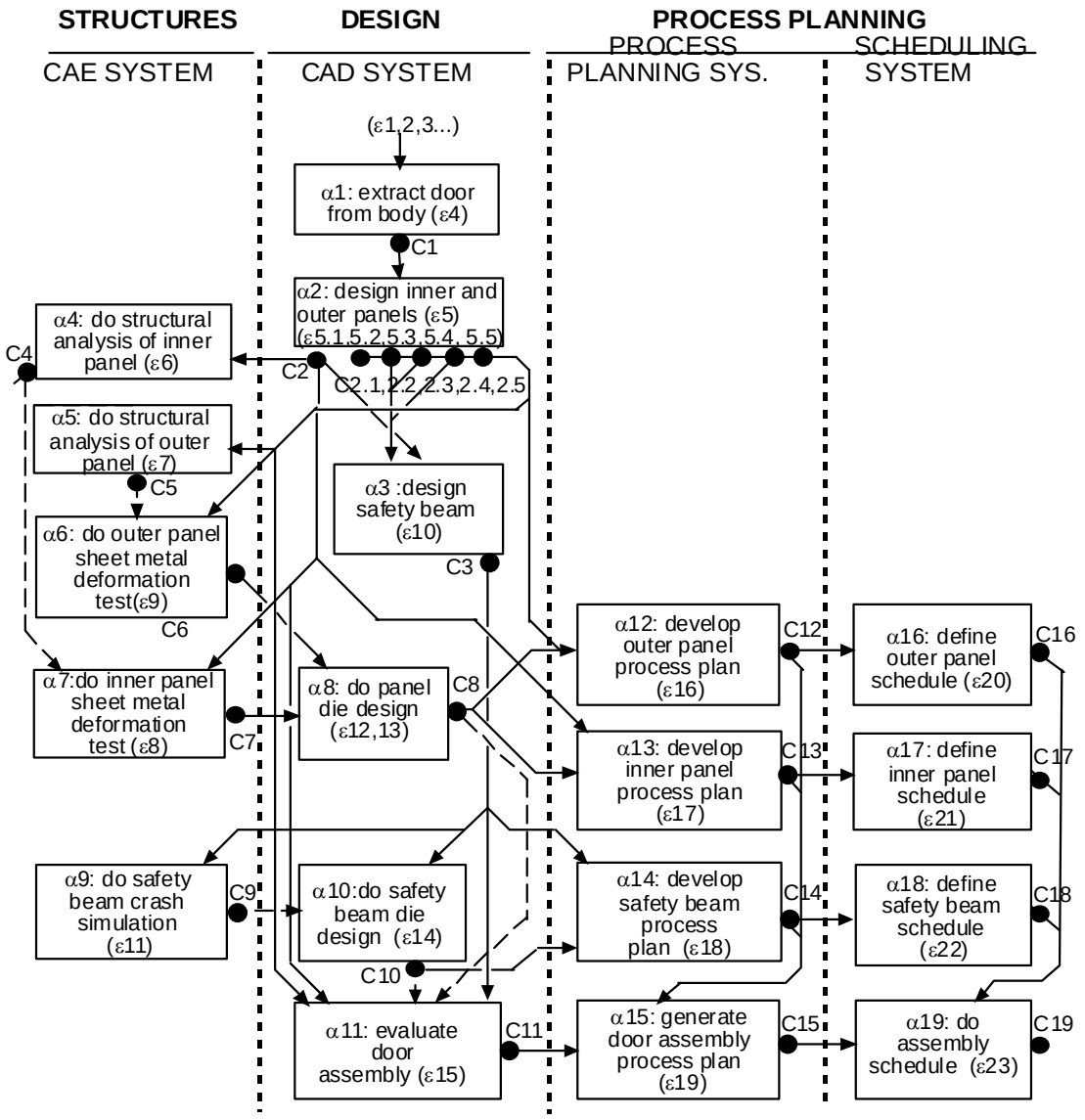

FIGURE EIGHT: Design operations, states and precedences modified by decomposing the design of the outer door panel for one operation.

The integrity management process defined in the example operates at a coarse level of granularity. Each *operation instance* typically has only one writeset instance and one after-constraint. More detailed levels of granularity are usually possible. Consider the case for the example where the outer door panel is decomposed into features that interact with other parts of the design, possibly as shown in Figure Seven. Instead of one object instance, the outer door panel now consists of five: each edge and the center. The corners are considered as overlaps of the two edge features. *Operation instance* ϕ_2 is modified to $\phi_{2,2}$ to reflect this decomposition. and the operations that read object instance 5.1 are now modified to read all or a subset of the five features and the corresponding constraints. The revised set of *operation instances* is shown graphically in Figure Eight and textually in Figure Nine.

$\phi_1=(\{e\}_{1,2,3...}{}^{R},\{e_4\}^{W},\varnothing^{B},\{c_1\}^{A})$ (extract door design from body)

$\phi_{2,2}=(\{e\}_4{}^{R},\{e\}_{5,5.1,5.2,5.3,5.4,5.5}{}^{W},\{c_1\}^{B},\{c_{2,2.1,2.2,2.3,2.4,2.5}\}^{A})$ (design inner and outer door panels)

$\phi_3=(\{e\}_{5,5.2,5.3,5.4}{}^{R},\{e\}_{10}{}^{W},\{c_{2,2.2,2.3,2.4}\}^{B},\{c_3\}^{A})$ (develop design of safety beam)

$\phi_4=(\{e\}_5{}^{R},\{e\}_6{}^{W},\{c_2\}^{B},\{c_4\}^{A})$ (do inner panel structural analysis)

$\phi_5=(\{e\}_{5.1,5.2,5.3,5.4,5.5}{}^{R},\{e\}_7{}^{W},\{c_{2.1.2.2,2.3,2.4,2.5}\}^{B},\{c_5\}^{A})$ (do outer panel structural analysis)

$\phi_6=(\{e\}_{5.1,5.2,5.3,5.4,5.5}{}^{R},\{e\}_9{}^{W},\{c_{2.1,2.2,2.3,2.4,2.5,5}\}^{B},\{c_6\}^{A})$

(do outer panel sheet metal deformation test)

$\phi_7=(\{e\}_5{}^{R},\{e\}_8{}^{W},\{c_{2,4}\}^{B},\{c_7\}^{A})$ (do inner panel sheet metal deformation test)

$\phi_8=(\{e\}_{8,9}{}^{R},\{e\}_{12,13}{}^{W},\{c_{6,7}\}^{B},\{c_{8,8.1}\}^{A})$ (do inner and outer panel die designs)

$\phi_9=(\{e\}_{10}{}^{R},\{e\}_{11}{}^{W},\{c_3\}^{B},\{c_9\}^{A})$ (do safety beam crash simulation)

$\phi_{10}=(\{e\}_{10}{}^{R},\{e\}_{14}{}^{W},\{c_{3,9}\}^{B},\{c_{10}\}^{A})$ (do safety beam die design)

$\phi_{11}=(\{e\}_{5,5.1,5.2,5.4,5.5,10}{}^{R},\{e\}_{15}{}^{W},\{c_{2,2.1,3,8,8.1,10}\}^{B},\{c_{11}\}^{A})$

(eval. door panels and safety beam assembly)

$\phi_{12}=(\{e\}_{5.1,5.2,5.3,5.4,5.5,12}{}^{R},\{e\}_{16}{}^{W},\{c_{2.1,2.2,2.3,2.4,2.5,8}\}^{B},\{c_{12}\}^{A})$ (develop outer panel process plan)

$\phi_{13}=(\{e\}_{5,13}{}^{R},\{e\}_{17}{}^{W},\{c_{2,8.1}\}^{B},\{c_{13}\}^{A})$ (develop inner panel process plan)

$\phi_{14}=(\{e\}_{10,14}{}^{R},\{e\}_{18}{}^{W},\{c_{3,10}\}^{B},\{c_{14}\}^{A})$ (develop safety beam process plan)

$\phi_{15}=(\{e\}_{15,16,17,18}{}^{R},\{e\}_{19}{}^{W},\{c_{11,12,13,14}\}^{B},\{c_{15}\}^{A})$ (generate door assembly process plan)

$\phi_{16}=(\{e\}_{16}{}^{R},\{e\}_{20}{}^{W},\{c_{12}\}^{B},\{c_{16}\}^{A})$ (define manuf.schedule for outer panel)

$\phi_{17}=(\{e\}_{17}{}^{R},\{e\}_{21}{}^{W},\{c_{13}\}^{B},\{c_{17}\}^{A})$ (define manuf.schedule for inner panel)

$\phi_{18}=(\{e\}_{18}{}^{R},\{e\}_{22}{}^{W},\{c_{14}\}^{B},\{c_{18}\}^{A})$ (define manuf. schedule for safety beam)

$\phi_{19}=(\{e\}_{19,20,21,22}{}^{R},\{e\}_{23}{}^{W},\{c_{15,16,17,18}\}^{B},\{c_{19}\}^{A})$ (do manuf.sched. for compl.door assembly)

FIGURE NINE: Representation of the modified design operations and precedence relations shown in Figure Seven, using the door decomposed into features.

Seven operation instances have been revised. Previously, given the design state presented in Table One, a revision to $\phi_{2,2}$ resulted in fifteen constraints being set to NULL: $\{c_{3,4,7,8,9,10,11,12,13,14,15,16,17,18,19}\}$ and requiring iteration of fifteen operations. Suppose that the change only involved feature $e_{5.1}$. Given the revised level of granularity for the operations, revision of the outer panel now would result in re-setting only one constraint, $C_{5.1}$, which would propagate according to (eq. 4.4) to set constraints $\{c_{5,6,8,11,12,13,15,16,17,19}\}$ to NULL. Now, only ten operations would be set to NULL. The change is in a location not effecting the beam design and all iteration checks for the beam are now eliminated.

Decomposition of other operations, especially *operation instance* ϕ_8, would allow their effects to be further localized. Also note that the remaining updates now identify a localized feature that has been changed, facilitating evaluation and reducing the effect of propagation.

This one example suggests the benefits of defining constraints and entity instances at a fine level of granularity, allowing information management and change propagation to be very specific. The decomposition was done for only one operation, though it was a crucial one depended upon by many later operations. Similar decompositions could be defined for the analyses and tests, but these would have little effect, because other applications do not use their results. (Detailed granularity of analysis results may help in backtracking, which is not addressed here.)

A fine level of granularity is also accomplished by decomposing the scope of an operation instance into multiple small ones. If these can be defined with only limited data and integrity constraint relations, they may be executed in parallel, with corresponding benefits for concurrent engineering. The granularity and specific decomposition used in one application need not correspond to those of other applications, such as those used in finite element modeling. They serve only to localize communication between preceding and succeeding applications. Fine level granularity supports revision management, similar to region management schemes used in computer graphics and geographical information systems [27].

V. MANAGEMENT AND IMPLEMENTATION

Here, integrity constraints have two alternative implementations The basic implementation is simply as *a state flag*. Upon successful completion, an application's interface to a design model sets the state flags to True that correspond to the *operation instance*'s after-constraints. That is, the applications set flags, which correspond to assertions regarding the rules realized and guaranteed within the code of the application. In this regard, the values asserted upon termination of an *operation instance* by after-constraints are similar to the assertions made regarding program correctness [1].

It is assumed that many existing engineering and manufacturing applications, many of which have long-standing use and validation, will not be re-written, but will have surrounding code or "wrapper" that includes translation and serves the purpose and has the general form shown in eq.(3.5). Prior to extracting readset data, the wrapper will check the value of before-constraints, to determine if the application can be executed. Alternative scopes of the *operation instance* might be considered and only allow application of the *operation instance* on some entity instances. The data for those instances is extracted and the operation is executed. A successful *operation instance* results in both new or modified data being assigned and the after-constraints set to True for the instances being written. As new data is written, the effects of the

changes are propagated. The changed constraint flags are then available to be read by the before-constraints of other applications.

Granularity is a property of an application's interface. If it allows extraction of a partial dataset, then the possible subsets are combinations defined by the granularity of the interface. Development of application interfaces that support partial iteration and that can check before-constraints for subsets of a dataset is an important general capability, made more valuable with the methods defined here. The author is working on examples of interfaces with these capabilities.

In some cases, the rules guaranteed by a previous *operation instance* may not be sufficient to match the before-constraints of another operation. Such cases arise when no *operation instance* guarantees some condition and a user of the program is required to put the design in a particular state. An example might be an interactive program for process planning that requires all welds to have their type specified, which is added in a general purpose CAD system. Another example is the requirement that all bending radii in a design be greater than some (small) value, for fabrication reasons, which is guaranteed by the user. In all such cases, a rule may not be guaranteed by an application but can be checked procedurally. This may be done in two ways: (1) with an explicit test, or (2) guaranteed by a user. In the first case, code must be written that can access the data model and apply the constraint rule, then set the flag to the appropriate condition. This case is similar to the normal use of integrity constraints in databases, except that they are *variant* constraints, which may or may not always be True. In the second case, a condition is guaranteed by a user, requiring an electronic sign-off.

Previously, techniques for change management have been developed for levels of granularity corresponding to a file [17] with change propagation based on timestamps. The procedures defined here show that a changed design, with a newer timestamp may or may not require changes to a associated entity defined earlier, depending upon the constraints affected. Semantic based relations are used here instead, which can work at any level of granularity defined by operations and constraints. Thus integrity constraints can be associated with an assembly, such as a body panel, or at a fine level, such as a feature or bolt and changes propagated at this level of granularity.

The integrity methods presented here have been defined in a data model [9] and implemented in a database implementation of the data model called EDM-2 [11,12]. Extension and testing of the definitions and meta-rules defined in this paper and also higher level management functions are currently being undertaken.

VI. INTEGRITY CONSTRAINTS IN INTERCHANGE STANDARDS

This paper demonstrates that the state of product design information includes the rules that are satisfied in the information describing that product. The state of these rules are of equal importance as the data in defining the information needed by an application. Any backend database or data interchange program or

standard needs to provide integrity state information as part of the exchange data, if the database is to be able to guarantee the integrity of the data as required for a specific application.

As flags, constraint values provide communication between applications regarding the state of data within the database. Such communication depends upon apriori explicit definition of constraints. That is, applications that might possibly use data in some integrity state made available by another application must communicate through a shared ontology of integrity rules regarding the data. Conditions may be realized by another application. However, unless the other application makes an assertion corresponding to the before-constraints of the successor operation, communication will not be realized.

It is the developers of engineering and design applications that know the internal requirements of their programs and the data well-formedness conditions embedded within and assumed by them. The issue is how are these conditions to be expressed as integrity constraints? In many areas, the integrity rules are fairly general and well understood, such as in geometric modeling [26]. Most visualization applications compute face normals from the order of vertices, while some CAD systems do not maintain order consistency. Some applications require arcs to be defined counterclockwise, while others allow angles to have both positive and negative rotations. These differences are known because CAD application vendors have discussed these differences in developing previous exchange standards, such as IGES and STEP. The same coordination is needed across the range of integrity issues associated with geometrical objects. It is also needed for higher level design objects, such as building, mechanical and process plant components. Standards efforts such as ISO STEP should include this type of information in its specification.

VII. RELATED WORK AND FUTURE EXTENSIONS

We have outlined some operations for the management of partial integrity. These include update methods that maintain a database in a globally valid state, and for querying constraint state. These only outline a full management system for partial integrity management in engineering design. An important consideration in the representation and methods presented is that *operation instances* are depicted as data that can be changed within live processes. The procedures are based on only limited global knowledge, which resides in a shared ontology of semantic integrity rules.

The method of communication developed here relies heavily upon constraints serving as assertions between independently developed applications. The use of pre-condition and post-condition assertions for program verification was introduced by Hoare [16], to define the state of data required for a particular algorithm or process. As a means to characterize the semantics of an algorithm or process, they have incorporated in several axiomatic approaches to programming [23]. For similar reasons, they have been adopted in some object-oriented programming styles, specifically in “programming by contract” [8, 24].

Here, they are used to characterize the logical conditions defining precedence relations among applications, and the effect of iteration.

Several extensions to the base concepts presented here are not covered. The predecessor and successor relation of operations and the propagation scheme presented does not address situations where there are alternative parallel paths available to complete the same integrity relation, that is, where rules are related by inclusive disjunction. Also not addressed is the use of estimated data; a common method of speeding up and allowing parallel activities is to use estimated values for critical variables. Much design work can be undertaken based on the estimated data, which results in reduced time as long as the estimates are close to the final result. But eventually, the estimated values have to be verified. Estimated data requires extension of the current integrity management scheme. The detailed precedence relation between before- and after-constraints within an application interface demands language capabilities with rich logical capabilities. The development of a constraint language for this purpose is distinct from existing efforts in this area.

Acknowledgment:

This work has benefited from the contributions and discussions of the EDM development team, including Tay Sheng Jeng, Moon Cho, Scott Chase, Hing Chan, and Hisham Assal. Discussions with Stott Parker refined my use of logic and provided new insights into the problem. Several refinements were suggested by the referees. This work was partially supported by a National Science Foundation grant, No. IRI-9319982.

REFERENCES

[1] Aho, A. J. Hopcroft and J. Ullman, The Analaysis and Design of Computer Algorithms, Addison-Wesley, Reading, Mass, 1974.

[2] Banerjee, J. W. Kim, H. Kim and H. Korth [1987] "Semantics and implementation of schema evolution in object-oriented databases",Proc. ACM SIGMOD 1987 Annual Conf. 16:3, pp.311-322.

[3] Barghouti, N.S., and G. Kaiser [1991], "Concurrency control in advanced database applications", ACM Comp. Surveys, 23:3, pp. 269-318.

[4] Buchmann, A.P., R.S. Carrera, M.A. Vasquez-Galindo, [1991], "Handling constraints and their exceptions: an attached constraint handler for object-oriented CAD databases, in K. Dittrich, U. Dayal, and A. Buchmann (eds.) On Object-Oriented Database Systems, Springer-Verlag, New York, pp.65-83.

[5] Clark, J.B. and T. Fujimoto,[1989] "Overlapping Problem Solving in Product Development", in K. Ferdows (ed.), Managing International Manufacturing, 1989, Amsterdam: Elsiver Press.

[6] Clark, K.B., and T. Fujimoto, [1989a]"Lead Time in Automobile Product Development Explaining the Japanese Advantage", J. Engr and Tech. Manag. 6:1, (September), pp.25-58

[7] DeBloch, S. T. Harder, N. Mattos and B. Mitscheng, [1989], "KRISYS: Support for Better CAD Systems" Conf. on Data and Knowlege Sys. for Manuf. and Engr. IEEE, pp.172-182.

[8] Desfray, Philippe, [1994] Object Engineering: The Fourth Dimension, Addison Wesley, Reading Mass.

[9] Eastman, C.M., [1994] "A Data Model for Design Knowledge", in Knowledge-Based Computer-Aided Architectural Design, G. Carrera and Y. Kalay (eds.), Elsevier Science Press, pp. 95-122.

[10] Eastman, C.M. and G. Shirley, [1994], "The management of design information flows", in S. Dasu and C. Eastman (eds.) Management of Design: Engineering and Management Processes, Kluver Press, N.Y. (1994).

[11] Eastman, C.M., M.S. Cho, T.S. Jeng and H.H. Assal, [1995] "A Data Model and Database Supporting Integrity Management Across Heterogeneous Applications", 2nd. International Congr. of Civil Engineering, ASCE, Atlanta GA, June.

[12] EDM Group, [1995] EDM-2 Reference Manual, Version 1.01, Design and Computation Research Report, Center for Design and Computation., University of California, Los Angeles, CA. 90024

[13] Formica, A. and M. Missikoff, [1993], "Integrity Constraints Representation in Object-Oriented databases", Information and Knowledge Management, First int. Conf. Proc. T. W. Finin, C. Nicholas and Y. Yesha, eds. Springer-Verlag Lecture Noes in Computer Science, No. 752.

[14] Gray, J. [1981], "The transaction concept: Virtues and limitations", *Proc. Seventh VLDB Conf., Cannes France*, (Sept.), pp. 144-154.

[15] Harhalis, G., Lin, C.P., Mark, L., Muro-Medrano, P. [1994], "Implementation of Rule-Based Information Systems for Integrated Manufacturing", Trans Knowl. and Data Engr. 6:6, pp. 892-908.

[16] Hoare, C.A.R. [1969], "An Axiomatic Basis for Computer Programming", Communications of the ACM, 12:10, (October, 1969), pp. 576-580.

[17] Katz, Randy, [1990], "Toward a unified framework for version modeling in engineering databases", ACM Comp. Surveys, 22:4, (December, 1990), pp. 375-408.

[18] Kim, W. E. Bertino and J. Garza, [1989], "Composite objects revisited", Proc. 1989 ACM SIGMOD Conf. SIGMOD Record, 18:2 (June, 1989), pp. 337-347.

[19] Kotz, A.M., Dittrich, K., Mulle, J. [1988], "Supporting Semantic Rules by a Generalized Event/Trigger Mechanism", Proc. Int. Conf. Extensing Database Technology, pp.77-91.

[20] Kusiak, A.,ed. [1993], Concurrent Engineering: Automation, Tools, and Techniques, J. Wiley, N.Y.

[21] Kutay, A. and C.M. Eastman [1983] "Transaction management in engineering databases", in SIGMOD Conf. on Engineering DB, IEEE, paper 31.3.pp. 73-80.

[22] MacKeller, B.,K. and J. Peckham, [1992], "Representing design objects in SORAC: a data model with semantic objects, relationships, and constraints", 2nd International Conf. on Artificial Intelligence and Design, June, 1992.

[23] Meyer, Bartrand, [1988] Object-oriented Software Construction, Prentice-Hall, N.Y.

[24] Meyer, Bartrand, [1990] Introduction to the Theory of Programming Languages, Prentice-Hall, N.Y.

[25] Motro, A. [1989], "Integrity = Validity + Completeness", ACM Trans. on Database Sys., 14:4, pp. 480-502.

[26] Requicha, A. [1980] "Representations of rigid solids: theory, methods and systems", ACM Comput. Surv. 12:4, (December), pp. 437-466.

[27] Samet, H. [1990], The Design and Analysis of Spatial Data Structures, Addison -Wesley, Reading MA.

[28] Smith, B., G. Rinaudat, [1988], Product Data Exchange Specification: First Working Draft, NISTIR 88-4004, National Institute of Science and Technology, U.S. Dept. of Commerce.

[29] Ullman, Jeffrey D. [1988], Principles of Database and Knowledge-Base Systems, Vol.1, Computer Science Press.Springer-Verlag, New York,pp.65-83.

REVISED VARIABLES USED

E ::= a class entity
{E} ::= a set of class entities
e ::= an entity instance
{e}::= a set of entity instances
C ::= constraint class
{C} ::= a set of constraint classes
c ::= a constraint instance
{c}::= a set of constraint instances
Φ ::= an operation class
ϕ ::= an operation instance