ProjectLeader: a Constraint-Based Process Support for the Distributed Design of Component Based Products

Marie-José Blin, Françoise Fabret, Olga Kapitskaia, François Llirbat

ProjectLeader: a Constraint-Based Process Support for the Distributed Design of Component-Based Products

Marie-José Blin*, Françoise Fabret†, Olga Kapitskaia‡, François Llirbat†

Résumé

Nous présentons, dans ce papier, un support d'aide à la conception collaborative de produits appartenant à une famille de produits. Le produit est modélisé par un arbre de composants et les contraintes de développement, utilisées pour guider le processus de conception, sont exprimées à l'aide d'un langage spécifique basé sur des expressions logiques. L'objectif principal de ProjectLeader est d'économiser le temps de conception en favorisant le travail simultané et en prévenant les retours en arrière. Ainsi, pour chaque intention de décision émise par un concepteur, ProjectLeader vérifie que la décision ne remettra pas en cause la possibilité d'atteindre un état correct du produit final, c'est-à-dire un état dans lequel toutes les contraintes sont vérifiées.

Mots-clefs : conception collaborative, système à base de composants, modélisation et traitement de contraintes de conception, modélisation et traitement de la variabilité

Abstract

In this paper we present a support that helps organizing distributed design of products belonging to a product family. We model the product to be developed as a component-based tree of object templates, develop a logic-based language that allows expressing diverse development constraints and use these constraints to guide the design process. The main objective of our support is to save time and development effort in increasing parallel work and avoiding roll-backs. For that, it reasons in terms of correctness of the future, *final* state of the product, and verifies that the product state after each operation allows the reacheability of such a state.

* LAMSADE, Université Paris-Dauphine, 75775 Paris, blin@lamsade.dauphine.fr

† INRIA, Domaine de Voluceau, Rocquencourt-B.P. 105, 78153-Le Chesnay Cedex, France, {Francoise.Fabret, Francois.Llirbat}@inria.fr

‡ Ecole supérieure d'ingénierie Léonard de Vinci, 92916-Paris la Défense Cedex, France, Olga.Kapitskaia@devinci.fr

Key words : collaborative design, component-based system, design constraints modeling and processing, variability modeling and processing

(This paper is in Pro. of Fourth International Workshop on Product Family Engineering (PFE-4), Bilbao, Spain, 3-5 october 2001, LNCS 2290, 2002, Springer, pp. 207-223.)

1 Introduction

Some systems have to be built for a lot of different technical and functional environments and responsibles of their development may decide to manage them as a product family. Generally, the products of the family are complex, share some of components between them and contains some other specific ones. We can call them component-based products. In this paper, we are interested by the design of a new product of a family. Construction of complex component-based products belonging to a same family presents a major challenge to modern industry. It requires a strong collaboration between several participants and the different components have to be put together with respect of constraints. The process can be modeled hierarchically: first, the project manager defines the "general architecture" and specifies product development constraints. Then, the work has to be divided between the teams designing the product and conducted in a way that satisfies development constraints and optimizes time and effort.

Currently, project leaders are obliged to manually divide the work between the teams and manually verify that the parallel work on the product by several teams do not violate the constraints. This task is laborious and prone to errors. The existing helps are not well-adapted. For example, distributed transactions in database systems force different teams to synchronise instantiation of objects involved in a constraint. Such synchronisation can be too strong a requirement in a distributed environment. In the field of configuration management and collaborative design, teams make initial choices that are treated like facts that have to be respected. The constraints are verified with respect to these choices and all possible instantiations of the objects steaming from these choices are calculated by a constraint solver and given to the designers. These solutions count to the fact that the number of initial choices is small and lead the designers towards a small number of "good" configurations.

In this paper we describe a more general solution that does not force the designers to choose one of the (small number of) known configurations and leaves them the liberty of instantiation. More precisely, we make the following contributions:

– We model a complex product to be constructed as a partially instantiated *composition tree* that specifies the dependency relationships between the components. The development of the product is modeled as operations on the tree: creating objects, setting values to attributes, etc. We use the real-life example presented in section 2 to illustrate our solution (Section 4.1).
– We propose a logic-based language to specify the product development constraints

(Section 4.2) that should be satisfied by the *final* product. These constraints specify the mandatory, allowed and prohibited configurations, wrt the objects created.

– We develop an efficient execution model that allows instantiation of the product without rollbacks, which helps to save a considerable amount of time and effort. A designer can submit desired operations to verify that the constraints are not violated, cancel them, if he finds out that he made a mistake, and commit them, when he is absolutely sure of his decision (Section 5).

2 Motivating Example

As our running example we choose the environment CORSSE (Component-ORiented System Specification Environment) dedicated to construction of distributed management systems for different types of stores ranging from small shops to hypermarkets. The systems can be installed in different countries with different management rules, different languages, and might require different standards. Each system may be composed of hardware, software, documentation and services. Each of these components are in turn composed of large number of sub-components. For example, hardware configuration of a particular system contains servers, workstations, cash registers and informative tills; a cash register is composed of several cables, one printer, and one or several screens. Software configuration can contain specific programs such as the cash register management program FORTE as well as general programs such as documentation formatter, print tools, test tools, a database management system and a graphical user interface. In addition, the system designer can choose one of the many available tools: for example a small shop might choose MicrosoftAccess while a hypermarket Auchan might choose Oracle or DB2 as DBMS.

The development of a system starts with an *initial design phase* when a system architect decides on the overall structure of the system and specifies the sub-components of each component. At this point the details (e.g. exact number of screens for each cash register, the type of a DBMS etc.) of components can be ignored. The architect also specifies the constraints of the system development, e.g. the need of a certain type of software ("we must have Netscape"), the incompatibilities between certain components ("we cannot have Eudora and Netscape in the same partition"), etc.

The construction of a system is never accomplished by one person: thus, after the initial design, an *instantiation phase* begins during which the work on different parts of the system is distributed between teams of specialists. The development teams start to "implement" the decisions taken during the specification phase. For example, the team responsible for cash register configurations will decide on how many screens each register gets, which registers get card readers and what kind of printers will be used. This instantiation should respect the dependencies between sub-components and the constraints defined in

the initial design phase. The work on a system is finished when all necessary components are created and all development constraints are satisfied.

Currently, the information about components, sub-components, their incompatibilities and/or successful configurations is not computerized, thus no help is provided in modeling the system that has to be created and monitoring the instantiation process.

3 Related Work

The question we attempt to answer is: how to build a valid complex product belonging to a family in avoiding roll-backs and allowing the maximum of concurrent work? In other words, how to manage the concurrent designing of a complex system configuration? Configuration management and concurrent engineering are well-known by several disconnected research areas : engineering design (car design, mechanical and civil engineering, etc.), software configuration management (SCM), product data management (PDM) and in a certain sense databases. Each of these areas differently considers configuration management and concurrent engineering.

In engineering design, like in our running example, the applications are being worked on by different teams; each team takes care of a sub-configuration (part) of the whole system. The teams make initial choices that are treated like facts that have to be respected. The constraints are verified wrt these choices, and all possible instantiation of the application steaming from these initial choices are calculated by a *constraint solver* and given to designers ([15], [20], [10], [19], [3]). These solutions count on the fact that the number of initial choices is small, and lead designers towards a small number of known “good” configurations. The constraints expressed in the system are *compatibility* constraints stating which components are compatible.

SCM focuses on software development management and particularly on version control and definition of consistent sets of component versions. Composition of software is provided by a list of components and based-rule selection mechanisms search the most appropriate version of each of them in a component repository [6]. Most of the time, no data model is provided (see for example, the industrial tool ClearCase [17]). Version branches, merge and locking mechanisms (check-in/check-out) and workspace managers allow the concurrent modification of a module by several developers (see the industrial tool CVS [5] and [7]). Now, with the development of component-based software, software design comes near to being system design and some researchers think to bring SCM and PDM together [18].

PDM tools provide means to design complex hardware systems (or products) with respect of quality, time and cost constraints [8], [13]. A data model defines the structure of the product as a tree where each node is a component and the edges are composition links.

Each component is designed by a team (or a person or a sub-contractor). Compatibility constraints may exist between components and team collaborations may be necessary to exchange needs and to agree on the choices. Constraints are checked on user request. If violated, work are to be undone.

Database community proposed a solution to collaborative updating of data : transaction concept has been extended to support long-duration activities and to solve the problem of constraint enforcement in software engineering and workflow applications (see [14], [1], [16], [9], [4], [11], [21], [22]). The main idea is to relax the ACID properties which would make data unavailable for a long time (since transactions may run for days or weeks). When concurrency conflicts or failures happen, the compensation concept is used in place of the standard rollback: transactions are associated with compensating transactions which leave the database in a consistent state. In federated databases, the verification of constraints concerning the data from different databases is postponed [2], [12]. However, either the transaction affecting a constraint is suspended until the verification is not done, or the transaction is accepted but risks to be undone, if a constraint is violated.

4 Modeling the Product

A product to be designed with the help of `ProjectLeader` passes through two stages: initial design phase and instantiation phase.

During the initial design phase, the architect of the product proposes the main architecture, without specifing details such as the exact number of objects of each type and the values of the attributes, and specifies the development constraints. For example, a CORSSE environment architect might decide that a supermarket configuration will contain a `CashRegisterConfiguration`, which in turn consists of a `CashRegisterSoftware` and a `CashRegisterHardware`; `CashRegisterHardware` consists of a `BasicCashRegister` which in turns consists of a `Keyboard`, a `Processor`, a `Drawer` and a `Memory`. The previous experience in constructing supermarket configurations, or the components documentation, leads to expressing the development constraints, e.g. the fact that versions 3.0 and 3.1 of the cash register management program FORTE is incompatible with a keyboard of the type FPI, or the fact that the currency of a ChequeReader should be euro. Then, `ProjectLeader` monitors system instantiation and only accepts actions allowing the reacheability of a “good” final product. Below, we present our hierachical model for expressing the initial design phase (section 4.1) and our logic-based language for specifying development constraints (section 4.2). Instantiation phase is treated in section 4.3.

4.1 Specifying the initial design : the composition tree

The general description of a product consists of a *schema* providing the specification of the different components and a hierarchical structure, referred to as *composition tree*, that specifies the subcomponents of each component. The architect of the product can define a new schema and construct a new composition tree based on this schema, or use a previously defined schema and/or composition tree. Product components described in a schema are modeled as object classes: each class is defined by a name and a set of attributes. Below we show class specifications for the CashRegisterConfiguration part of our running example.

```
CashRegisterConfiguration();
FORTE(version, size, interface);
CashRegisterHardware();
Keyboard(type);
Drawer(type, dimensions);
ReaderUnit(type);
Adaptor(type);
CheckReader(type, brand);
ComplementaryHardware();
```

```
CashRegisterSoftware();
TCF(version, size, interface);
BasicCashRegister(type, brand);
Processor(type);
Memory(type, speed, capacity);
Cable (type);
CardReader(speed, brand);
Scanner (type, brand);
```

Components are organized in a composition tree (noted CT). A node n of CT is a *template* of a class C (noted $T(C)$). The presence of $T(C)$ in the composition tree indicates that a number of objects of the class C can be instantiated at this point in the final product, but their exact number and characteristics (the values of attributes) are non-specified at the initial design phase[1]. Any number of templates of a class C can be present in CT (and are differentiated by the id's of the nodes of the tree).

Both schema and composition tree are encoded as XML files, so an architect can either edit XML files or use the system GUI.

Figure 1 shows the CashRegisterConfiguration composition tree. The numbers between brackets refer to the associated constraints described below.

The successful fulfilment of the initial design phase demands a good knowledge of the application domain and should be performed by a human. `Project Leader` provides a model to represent an initial design of a product but leaves the liberty of modelisation to the product architect.

1. If the architect wants to enforce certain attribute values he can do it by associating a constraint (see section 4.2) to the template node.

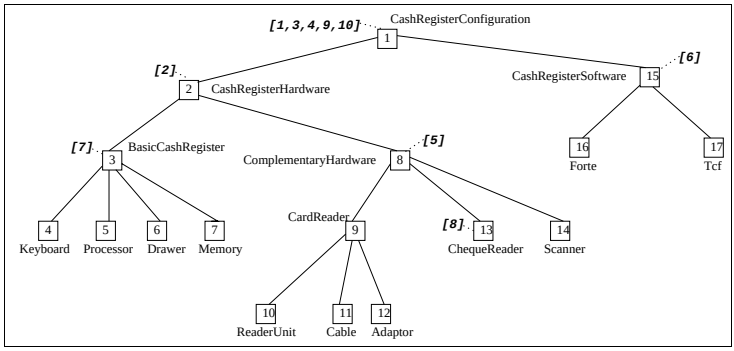

FIG. 1 – *Initial composition tree for the* CashRegisterConfiguration

4.2 Development Constraints

After defining a schema and constructing a composition tree, the architect should specify the *development constraints* of the product. These constraints can reflect the previous experience in construction of similar types of products or be given as a part of a component documentation.

To help a product architect we have developed a high-level language to express development constraints. We distinguish three types of constraints: (1) constraints describing prohibited configurations, i.e. the incompatibilities between different (versions of) components, referred to as INCOMPATIBILITY constraints. (2) constraints describing the mandatory presence of certain (versions of) components in the presence of some other (versions of) components, referred to as DEPENDENCY constraints. (3) constraints stating choice between certain (versions) of components, referred to as CHOICE constraints. Internally, each constraint is represented as a disjunctive logical formula.

Below, we give the high-level definitions of constraints and the equivalent logical formulas. The set of all constraints for the `CashRegisterConfiguration` is given in Appendix A.

Definition 1. Atom. An *atom* is an expression of the form $(C(P))$, where (1) C is a class name, and (2) P is a conjunction of predicates $p_1\&...\&p_k$ where p_i is of the form $a \in V$ or $a \notin V$, a is an attribute occurring in C and V is a set of values.

For example, the atom (Forte.version $\in$ {3.0, 3.1}) concerns versions 3.0 and 3.1 of the software FORTE.

Definition 2. Constraint Formula. A *constraint formula* is a conjunction of atoms.

Definition 3. Incompatibility constraint. An *incompatibility constraint* c is expressed as INCOMPATIBILTY (E), where E is a constraint formula. The equivalent logical formula is $\neg(E)$, that we rewrite as a disjunction of negative atoms.

For our running example, the constraint INCOMPATIBILITY((Forte.version $\in$ {3.0, 3.1}), (Keyboard.type $\in$ {FPI})) indicates that the versions 3.0 and 3.1 of the sofware FORTE are incompatible with the keyboard FPI. The equivalent logical representation of this constraint is:
$\neg$ (Forte.version $\in$ {3.0, 3.1}) $\vee\neg$ (Keyboard.type $\in$ {FPI})

Definition 4. Dependency constraint. A *dependency constraint* c is expressed as: DEPEND(E, E'), where E and E' are constraint formulas. The equivalent logical expression is $\neg(E) \vee E'$.

For our running example, the constraint DEPEND ((Scanner.brand $\in$ {NCR}), (Keyboard.type $\in$ {FPI})) indicates that if the product has a Scanner of the NCR brand it should also have an FPI keyboard. The equivalent logical representation is:
$\neg$ (Scanner.brand $\in$ {NCR} $\vee$ (Keyboard.type $\in$ {FPI})

Definition 5. Choice Constraints. A *choice constraint* c is expressed as: CHOICE(E_1, ..., E_n), where E_i is a constraint formula. The equivalent logical representation is: $E_1 \vee$ ... $\vee E_n$. A choice constraint states that a product should have a set of objects satisfying any one of the given formulas. The choice constraint can have only one formula, expressing the obligation of having a set of objects satisfying this formula.

The constraint CHOICE(Forte) indicates that the `CashRegisterConfiguration` must have a Forte software. The equivalent logical representation is: Forte()

Definition 6. Constraint Scope. Development constraints expressed during the initial design phase have different scope of application. Consider constraints 9 (Forte()) and 10 (Tcf()) (see Appendix A) that require that a `CashRegister Configuration` contains both TCF and FORTE software, and the constraint 6 ($\neg$ (Forte) $\vee\neg$ (Tcf)) that prohibits the coexistence of these programs. These constraints seem contradictory. However, the `CashRegisterSoftware` template at node 15 can be instantiated to two `CashRegisterSoftware` objects: one with a `Forte` object and the other with a `Tcf` object (see Fig.2). Thus, if constraint 6 is applied at the node 15 and constraints 9 and 10 are applied at the node 1 (`CashRegisterConfiguration`), all constraints are satisfied.

To express the scope of a constraint the designer attaches constraints to the nodes of the composition tree, thus each node n has a set of constraints $c_1 ... c_n$ associated with it. Thus, for the product to function correctly, each constraint c_i attached to n should be

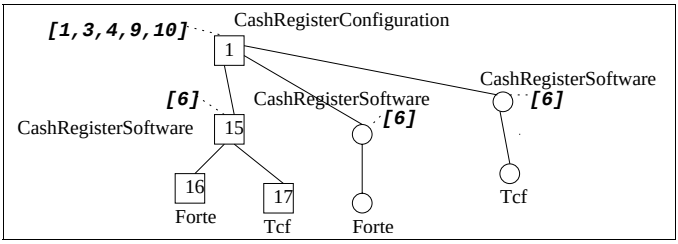

FIG. 2 – *Example of instantiation of the* CashRegisterSoftware *template. Objects are shown as circles.*

satisfied by n and by all nodes in the subtree of n. The mapping of the constraints defined in Appendix to the nodes of the composition tree is given in Fig.1.

Definition 7. Satisfied constraints. The satisfaction of a constraint is defined with respect to the current state of a composition tree, referred as a composition tree instance I. Recall that our constraints are disjunctions of positive or negative atoms. Thus, a constraint c is satisfied on I if at least one of its atoms is satisfied.

A positive atom a=($C(p_1\&...\&p_n)$ at node n is *satisfied* on I if there exists an object o in I of class C in the scope of n such that for i = 1..n the value v_i of the attribute a_i in o satisfies the corresponding predicate p_i. We say that o satisfies a. Note, that if the predicate of a is not specified, any object of the class C in the scope of n satisfies it.

A negative atom a=($\neg C(p_1\&...\&p_n)$) at the node n is *satisfied* if there is no template of class C under n and each object of class C created under n does not satisfy some p_i.

4.3 Instantiation

During the second phase of the application life cycle (the instantiation phase) our support monitors product instantiation and only accepts actions allowing the reacheability of a "good" final configuration. The instantiation proceeds as the sequence of the following operations:

– *Create Object.* An object $o(C)$ can be created by instantiating a template $T(C)$ at the node n. The new node n_{new} is created in the tree and is attached to the same father node as n. The subtree originating at n is copied as the subtree of n_{new} [2].

2. Copy of the sub-tree includes the copy of the constraints attached to its nodes

`ProjectLeader` allows the creation of an object o at the node n only if the father node of n is an object (and not a template).
- *Set a value of an object attribute.* With `ProjectLeader` values can be set only on not instantiated attributes. To change the value on an object attribute a the developper must first cancel the previous set value operation on a to obtain a not instantiated attribute.
- *Delete Template.*
  Node n of $T(C)$ and the subtree originating at n is deleted.

Note that most of the time, product family development supposes the creation, using and management of component versions. In a composition tree, a component version is an object. The choice to instantiate a template with one version of the component or another one is the liberty of the designer.

5 Cooperative Execution Model

5.1 Overview

The implementation of the product is modeled as instantiation of the initial composition tree, using the operations described in Section 4.3. Different teams of designers work on different parts of the product in parallel, communicating independently with our support. A designer issues one of the three commands: submit(op), cancel(op) and commit(op). After each command the designer receives an acknowledgment, if the command is accepted by `ProjectLeader`, or a refusal.

Submit(op) indicates the intent to perform the operation op and allows `ProjectLeader` to verify that applying op puts the product into a state that allows the reachability of a correct final state. A submitted operation can be canceled by issuing cancel(op) and our support guarantees that canceling op does not affect the execution of operations submitted by other designers. Commit(op) is issued for the operations that were already submitted and checked by `ProjectLeader`; committed operations cannot be canceled. Since several teams can work in parallel on the instantiation of the product it is important to check that operations submitted by all the teams always allow the reachability of a final correct state.

Example 5.1 *Consider the initial composition tree depicted in Fig.1 and assume that only two constraints are defined:*

Constraint 1:$\neg$ *(Forte.version* $\in$ *{3.0, 3.1})* $\vee\neg$*(Keyboard.type* $\in$ *{FPI})*
Constraint 2: $\neg$ *(Scanner.brand* $\in$ *{NCR}* $\vee$ *(Keyboard.type* $\in$ *{FPI}*

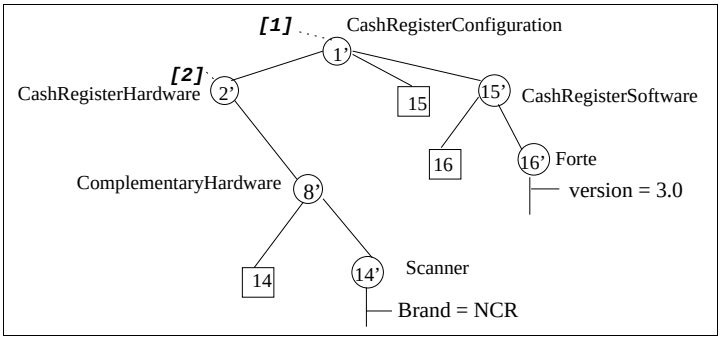

FIG. 3 – *Example of instantiation of composition tree leading to a DeadEnd state*

Let us suppose a scenario in which two teams work in parallel on the supermarket configuration: Team1 is working on CashRegisterHardware and Team2 is working on CashRegisterSoftware. First, Team1 decides to use a scanner of brand NCR and submits this choice to `ProjectLeader` *by issuing* submit(set(brand = NCR(create(new Scanner)))). *This choice affects Constraint* 2. *Since the type of the keyboard is not fixed yet, constraint* 2 *is not violated and operation is accepted; Team1 receives* ack(op1). *Now, Team2 proposes to use version 3.3 of the Forte software, by issuing* submit(set(version=3.3(new Forte())))*. This choice does not affect any constraint and is thus accepted; Team2 receives* ack(op2). *It turns out however that Forte version 3.3 is not available and Team2 has to use another version of the software. Team2 cancels the operation (*cancel(op2)*) and issues* submit(set(version=3.0(new Forte()))). *The resulting composition tree is depicted in Fig.3. In this figure squares represent template nodes and circles represent created object nodes. Because of the constraint* 1, *Forte version 3.0 can be accepted if no keyboard of type FPI is used in the CashRegisterHardware configuration. This is inconsistent with the fact that one FPI keyboard has to be used to satisfy constraint* 2. *No operation can satisfy both constraints and thus no correct state can be reached. We call such state a* DeadEnd *state. The only solution to avoid DeadEnd state is to refuse the operation of Team 2.*

5.2 State of the product

The product can have four different states (see Fig.4). The product is in *Correct* state if all the constraints are satisfied; it is in *DeadEnd* state if any correct state cannot be reached

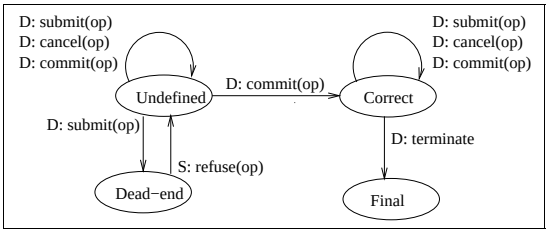

FIG. 4 – *Product's states in* `ProjectLeader`. `D` *shows designer's commands;* `S ProjectLeader` *responses.*

whatever the future operation sequences are. It is in *Final* state when the development has terminated. It is in *Undefined* state otherwise. The execution starts in Undefined state. A command issued by a designer from the Undefined state always triggers the checking of DeadEnd state. If the DeadEnd state is detected `ProjectLeader` refuses the command and the product returns to the Undefined state. The product enters the Correct state only when all constraints are satisfied. Once the product has reached a Correct state, a designer can send any command to update the configuration tree without affecting its correctness. He (she) can decide to terminate the work at any moment by issuing a $terminate()$ command.

Figure 5 summarizes the *execute_command* procedure executed at each state transition. This procedure maintains the current value of the composition tree instance I and the set C of constraints to be verified. At each operation submission (`submit(`op`)`) the procedure calls the $CheckDeadEnd$ algorithm. $CheckDeadEnd$ returns ACK if the Correct state can be reached from the composition tree I' obtained by applying op on I. It also guarantees that (i) once accepted the operation op will never be invalidated due to other operations and (ii) that a future cancelation of op cannot invalidate any concurrent operations. As a consequence op can be cancelled or committed without additional checks. Thus, `cancel(`op) command consists simply in reconstructing the configuration tree from its state before `submit(`op) by re-applying all operations submitted after op (except op). At commit command (`commit(`op`)`) the execute_command procedure computes the set of constraints which are satisfied by op. Since op is a committed operation and since a committed operation cannot be canceled, the constraints satisfied by op cannot be invalidated any more and the product can safely remove these constraints from the set of constraints to be verified. The product reaches a Correct state when no constraint remains. Subsection 5.3 describes how we compute the set of satisfied constraints. Subsection 5.4 describes how we detect DeadEnd states.

Execute_command:
Input:
cmd a command sent by a designer
Global Variables:
I: the current configuration tree instance, C : the set of constraints to check on I
Body :
Switch(cmd)
CASE cmd = submit(op):
  $result$ = CheckDeadEnd(op); if ($result$ = ACK) I = Apply(I,op)
CASE cmd = cancel(op):
  Let cmd_After be the sequence of commands sent to the product after submit(op)
  Let I_before be the configuration tree before submit(op)
  $I := I\_before$
  foreach command op in cmd_After do I= Apply(I,op) od
  $result$:= ACK
CASE cmd = commit(op):
  S := ComputeSatisfiedConstraints(op);$C = C - S$;if(C is empty) GO TO $Correct$ state
  return $result$

FIG. 5 – *Execute_command algorithm in* `ProjectLeader`

5.3 Computing satisfied constraints

A constraint is satisfied by an operation op if at least one of its atom is satisfied by op. We use the following rules to compute satisfied atoms. Let a be an atom at node n and op a committed operation.

1. If op is a $createObject$ of an object o of class C and a is an atom of the form $C()$ and n is above o (or equal to) then a is satisfied.
2. If op sets to v an attribute A of an object o of class C and a is an atom of the form $C(p)$, n is above o (or equal to) and $(A = v) \Rightarrow p$ then a is satisfied.
3. If op is the deletion of a template node T of class C, a is an atom of the form $\neg C(p)$, n is above T and all other nodes of class C under n are object nodes whose value contradicts p then a is satisfied.

5.4 DeadEnd state Checking

Given a composition tree CT and its current state S, the problem is to check if S is (or is not) a DeadEnd state. S is in a DeadEnd state if and only if there is some constraint c such that each atom of c is unsatisfiable in S. The problem can be reformulated as follows: S is not a DeadEnd state if and only if each constraint contains at least one atom that is satisfiable in S. To answer this problem we use an exhaustive approach that consists in enumerating all possible composition trees obtained by selecting one atom from each constraint. We call $atomic$ image of CT such a composition tree. CT is in DeadEnd state if and only if all its atomic images are in a DeadEnd state.

Example 5.2 *Let us consider the example 1 of Section 5.1 with the corresponding composition tree depicted in Fig.3. There are four possible atomic images where nodes 1 and 2 are respectively attached to the following atoms:*

atomic tree 1:*node 1': ¬Forte(version* $\in \{3.0,3.1\}$*) node 2': ¬Scanner(Brand=* NCR*)*
atomic tree 2:*node 1': ¬Forte(version* $\in \{3.0,3.1\}$*) node 2': Keyboard(type =* FPI*)*
atomic tree 3:*node 1': ¬(Keyboard(type =* FPI*)) node 2': Keyboard(type =* FPI*)*
atomic tree 4:*node 1': ¬(Keyboard(type =* FPI*)) node 2': ¬Scanner(Brand=* NCR*)*

To check if an atomic image CT_A is in DeadEnd state we use the following *DeadEnd rules* which give necessary and sufficient conditions: CT_A corresponds to a DeadEnd state if and only if there exists a node n of CT_A, and an atom a at node n such one of the following condition is true:

- R1 a is a negative atom of the form $a = \neg C(p)$ and there is a traversal path p of the tree rooted at n such that the conjunction of all atoms of class C in p is unsatisfiable.
- R2 a is a positive atom of the form $C(p)$ and each traversal path p rooted at n is such that one of the following conditions holds: (i) the conjunction of all atoms of class C in p is unsatisfiable or (ii) p contains only one node of class C and this node is attached to a positive atom a' that contradicts a (a' is of the form $C(p')$ with $p \wedge p' = false$).

where the conjunction of atoms is computed using the following rules:

- S1 Let $C(p)$ and $\neg C(p')$ be two atoms where C is the class and p and p' represent predicates. Then $C(p) \wedge \neg C(p')$ is equal to $C(p \wedge \neg p') \wedge \neg C(p')$.
- S2 Let $C(p)$ be an atom. If p is false then $C(p)$ is equal to *false*.
- S3 Atom $C()$ is equal to $C(true)$
- S4 Let A be a conjunction of atoms. Then $A \wedge false$ is equal to $false$, and $A \wedge A$ is equal to A
- S5 Let A be a conjunction of atoms; A is unsatisfiable if A is equal to $false$.

Let us consider the atomic tree 3 described in example 2. This tree is clearly in a DeadEnd state. Indeed considering the traversal path rooted at node 1' and the atom ¬(Keyboard(type $= FPI$)), we obtain the following atom conjunction: C = ¬(Keyboard(type $= FPI$)) ∧ Keyboard(type $= FPI$). By S1 C = Keyboard(type $\neq FPI \wedge$ type = FPI) = Keyboard(false), and by S2 C = false the DeadEnd rule R1 is verified.

Impact of operations. Besides modifying the composition tree topology, an operation also restricts the set of possible future states. We model these restrictions by setting additional constraints. We then apply the criteria described above to check if the increased set of constraints does not lead to a DeadEnd state. Operation impact depends on the type of the operation:

1. Creation of an object o from a template node n of class C adds new paths in the composition tree since it performs the copy of n's subtree. It also copies the associated constraints. Moreover, we model the fact that each new state has to contain object o by attaching an additional atomic constraint of the form $C()$ at node o.
2. Setting a value v to an attribute a of an object o of class C restricts the possible values of attribute a of o. We model this restriction by transforming the additional atom generated at the object creation by an atomic constraint of the form $C(a = v)$ at node o.
3. Deleting a template node n of class C deletes existing paths under n. It also forbids the creation of new objects of class C at this level. We model this restriction by adding a new atomic constraint of the form $\neg C()$ at the father node of n[3]. Note that by adding such constraint we forbid the deletion of the template node n before the objects created (if they are) from n are committed

Example 5.3 *Let us consider again the example 1 of Section 5.1. To take into account restrictions due to the operations performed by Team 1 and 2 we need to add the following atomic constraints. We add an atom of the form Forte(version* $\in \{3.0\}$*) at node* $16'$ *to indicate that the attribute version is set to* 3.0*. We add an atom of the form Scanner(Brand=* NCR*) at node* $14'$ *to indicate that attribute brand is set to* NCR*. This impacts the set of atomic trees depicted in example 5.2 as follows:*

atomic tree 1:*node 1': ¬Forte(version* $\in \{3.0,3.1\}$*) node 2': ¬Scanner(Brand=* NCR*)*
node 14': Scanner(Brand= NCR*), node 16': Forte(version* $\in \{3.0\}$
atomic tree 2:*node 1': ¬Forte(version* $\in \{3.0,3.1\}$*) node 2': Keyboard(type =* FPI*)*
node 14': Scanner(Brand= NCR*), node 16': Forte(version* $\in \{3.0\}$
atomic tree 3:*node 1': ¬(Keyboard(type =* FPI*)) node 2': Keyboard(type =* FPI*)*
node 14': Scanner(Brand= NCR*), node 16': Forte(version* $\in \{3.0\}$
atomic tree 4:*node 1': ¬(Keyboard(type =* FPI*)) node 2': ¬Scanner(Brand=* NCR*)*

3. The rationale for adding atom $\neg C()$ at the father node of n is to avoid future situation where some constraint above in the path imposes the creation of an object of class C after the template node is deleted.

node 14': Scanner(Brand= NCR*), node 16': Forte(version* $\in \{3.0\}$

It is easy to check that all these atomic trees are in a DeadEnd state. Indeed the DeadEnd rule $R1$ *applies on the traversal path rooted at node* $1'$ *and ending at node* $16'$ *in atomic trees* 1 *and* 2*, on the traversal path rooted at node* $1'$ *and ending at node* $2'$ *in atomic tree* 3 *and on the traversal path rooted at node* $2'$ *and ending at node* $14'$ *in atomic tree* 4*.*

6 Conclusion

In this paper we describe a support designed to help the constraint-based design of complex component-based products involving a lot of different teams. Our support provides a powerful language for expressing development constraints including both negations and disjunctions. We propose an efficient execution model which allows different teams to work in parallel, and instantiate a product without rollbacks. Teams propose operations to our support which in turn can accept or refuse it. An accepted operation cannot compromise the final state of the product under construction. Indeed, our support reasons both on the current state and the future states, (i.e. on the states reachable from the current state), and is able to detect DeadEnd states. We exhibit necessary and sufficient conditions for a product under construction to be in a DeadEnd state. The DeadEnd check algorithm allows not only to check inconsistencies in the designer's constraints and wrt to the composition tree but also to add constraints and/or template at run-time, i.e. during the instantiation of the composition tree: after upading the composition tree and/or the set of atomic trees, the set is checked for a DeadEnd state.

We are currently working on the implementation of the algorithms that checks DeadEnd state. As future work we plan to work in two directions. The first one concerns the collaborative aspect. `ProjectLeader` can be extent in order (1) to facilitate collaboration between concurrent teams by automatically deriving collaboration groups from the evolution of constraints satisfaction, (2) to provide possible instanciation of objects to the partners involved in a submit operation, and (3) to integrate existing collaboration protocols as notification of a submit operation to all the involved partners and processing of partners responds including votes, vetos, etc... The second direction of future work concerns the constraints and the primitives provided by `ProjectLeader`. We think to extend the constraint language to represent cardinality as "`CashRegisterHardware has to be composed of n1 to n2 BasicCashRegister`". Additional primitives would allow to instantiate several objets from one template node in one single operation and to create an object by copy of another one.

References

[1] P. Bernstein, E. Newcomer. Principles of Transaction Processing. *Morgan Kaufmann Publishers*, 1998.

[2] S.S. Chawathe, H. Garcia-Molina, J. Widom. A ToolKit for Constraint Management in Heterogeneous Information Systems. In *12th Int. Conference on Cooperative Information Systems*, Brussels, pp : 38-47, 1996.

[3] C.T. Culbreth, M. Miller and P. O'Grady. A Concurrent Engineering System to Support Flexible Automation in Furniture Production. In *Robotics and Computer Integrated Manufacturing*, Vol. 12, no. 1, pp : 81-91, 1996.

[4] P. Chrysanthis, K. Ramamritham. A Formalism for Extended Transaction Model. In *Proceedings of the Seventeenth International Conference on Very Large Data Bases*, 1991.

[5] Concurrent Versions System. http://cvshome.org/

[6] R. Conradi, B. Westfechtel. Versions Models for Software Configuration Management In *ACM Computing Surveys*, 30(2) pp : 232-282, June 1998

[7] J. Estublier. Workspace Management in Software Engineering Environment In *SCM-6 Workshop*, Springer LNCS 1167, Berlin, Germany, March 1996

[8] J. Estublier , JM. Favre, P. Morat. Towards SCM/PDM Integration In *Lecture Notes in Computer Science*, 1439, pp : 75-94, Springer, July 1998

[9] A. K. Elmagarmid, Y. Leu, W. Litwin, M. Rusinkiewicz. A Multidatabase Transaction Model for Interbase. In *Proceedings of the Sixteenth International Conference on Very Large Data Bases*, 1990.

[10] B. Faltings, R. Weigel. Constraint-Based Knowledge Representation for Configuration Systems. *Technical report no. TR-94/59, EPFL, Dept. Info.*, 1994.

[11] H. Garcia-Molina, D. Gawlick, J. Klein, K. Kleissner, K. Salem. Modeling Long-Running Activities as Nested Sagas. In *Proceedings IEEE Spring Compcon*, 1991.

[12] P. Grefen, J. Widom. Protocols for Integrity Constraint Checking in Federated Databases. In *International Journal of Distributed and Parallel Database*, 5(4), pp :327-355, October 1997.

[13] G. Hedin, L. Ohlsson and J. McKenna. Product Configuration Unsing Object Oriented Grammars In *Lecture Notes in Computer Science* no. 1439, pp : 107-134, Springer, July 1998

[14] M. Kamath and K. Ramamritham. Correctness issues in Workflow Management. In *Distributed System Engineering Journal : Special issue on Worflow Management System*, Vol. 3, no. 4, pp : 213-221, December 1996.

[15] C. Lottaz. Collaborative Design using Constraint Solving. *http://liawww.epfl.ch/∼lottaz/iccs/collaboration*, 1997.

[16] C. Pu. Superdatabases for Composition of Heterogeneous Databases. In *IEEE Proceedings of The Fourth Conference on Very Large Data Bases*, 1988.

[17] Rational ClearCase. http://www.rational.com/products/clearcase/index.jsp.

[18] B. Westfechtel and R. Conradi. Software Configuration Management and Engineering Data Management : Differences and Similarities. *Lecture Notes in Computer Science*, no. 1439, pp : 95-106, July 1998.

[19] S. Willmott, B. Faltings. Explicit Representation of Choice in a Content Language: Standardised Communication for Constraints and Constraint Satisfaction Problems. In *FIPA meeting*, Seoul, http://liawww.epfl.ch/CCL, January 1999.

[20] S. Willmott, B. Faltings, M. Calisti, S. Macho-Gonzalez, O. Belakhdar and M. Torrens. Constraint Choice Language (CCL). *Technical report 99/320, EPFL, Dept.Info.*, 1999.

[21] H. Waechter, A. Reuter. The ConTract Model. In *Database Transaction Models for Advanced Applications*, A. K. Elmagarmid, editor, Morgan Kaufmann Publishers, 1992.

[22] G. Weikum, H.-J. Schek. Concepts and Applications of Multilevel Transactions and Open Nested Transactions. In *Transactions Models for Advanced Applications*, A. K. Elmagarmid, editor, Morgan Kaufmann Publishers, 1992.

A Constraints for CashRegisterConfiguration

Below, we give incompatibility, dependency and choice constraints for our running example.

1. The versions 3.0 and 3.1 of the sofware FORTE is incompatible with the keyboard FPI:
   INCOMPATIBILITY((Forte.version $\in$ {3.0, 3.1}), (Keyboard.type $\in$ {FPI})).
   The equivalent internal representation of this constraints is:
   $\neg$ (Forte.version $\in$ {3.0, 3.1}) $\vee\neg$ (Keyboard.type $\in$ {FPI})

2. If the system has a Scanner of the NCR brand it should also has an FPI keyboard:
   DEPEND ((Scanner.brand ∈ {NCR}), (Keyboard.type ∈ {FPI})).

   The equivalent internal representation is:
   ¬ (Scanner.brand ∈ {NCR}) ∨ (Keyboard.type ∈ {FPI})

3. The versions AU33_SD, AU33_IF and AU32_N of the software Tcf is incompatible with a scanner of the brand different from ICL:
   INCOMPATIBILITY ((Tcf.version ∈ {AU33_SD, AU33_IF, AU32_N}), (Scanner.brand ∉ {ICL})).
   The equivalent internal representation is:
   ¬ (Tcf.version ∈ {AU33_SD, AU33_IF, AU32_N}) ∨¬ (Scanner.brand ∉ {ICL})

4. The versions AU33_ND and AU32_S of the software Tcf is incompatible with a scanner of a brand different from NCR:
   INCOMPATIBILITY ((Tcf.version ∈ {AU33_ND, AU32_S}) (Scanner.brand ∉ {NCR})).
   The equivalent internal representation is:
   ¬ (Tcf.version ∈ {AU33_ND, AU32_S}) ∨¬ (Scanner.brand ∉ {NCR})

5. Scanner NCR, ChequeReader different from DASSAULT or ICL and a CardReader with the speed different from 1200 bauds do not work together:
   INCOMPATIBILITY ((Scanner.brand ∈ {NCR}), (ChequeReader.brand ∉ {DASSAULT, ICL}), (CardReader.speed ∉ {1200 bauds}))
   The equivalent internal representation is:
   ¬ (Scanner.brand ∈ {NCR}) ∨ ¬ (ChequeReader.brand ∉ {DASSAULT, ICL})
   ∨¬ (CardReader.speed ∉ {1200 bauds}))

6. The software Forte is incompatible with the software Tcf:
   INCOMPATIBILITY ((Forte), (Tcf)).
   The equivalent internal representation is:
   ¬ (Forte) ∨¬ (Tcf).

7. If the system possedes a BasicCashRegister (instantiates an object of this class), it should also instantiate a Keyboard, a Processor, a Drawer, and a Memory:
   DEPEND ((BasicCashRegister), (Keyboard, Processor, Drawer, Memory)). The equivalent internal representation is:
   ¬ (BasicCashRegister) ∨ (Keyboard & Processor & Drawer & Memory). However, since we require each constraint to be a disjunction of atoms, we rewrite this expression as the following set:
   {¬ (BasicCashRegister) ∨ Keyboard, ¬ (BasicCashRegister) ∨ Processor, ¬ (BasicCashRegister) ∨ Drawer, ¬ (BasicCashRegister) ∨ Memory }.

8. If the system has a ChequeReader, the value of the `currency` attribute should be `euro`:
   DEPEND ((ChequeReader), (ChequeReader.currency ∈ {Euros})).

   The equivalent internal representation is:
   ¬ (ChequeReader) ∨ (ChequeReader.currency ∈ {Euros})

9. The supermarket must have a Forte software: CHOICE(Forte).

   The equivalent internal representation is: Forte()

10. The supermarket configuration must have a Tcf software: CHOICE(TCF).

    The equivalent internal representation is: Tcf().