



COVER SHEET

This is the author-version of article published as:

La Rosa, Marcello and van der Aalst, Wil M. and Dumas, Marlon and Ter Hofstede, Arthur H. and Gottschalk, Florian (2006) Generating Interactive Questionnaires From Configuration Models.

Copyright 2006 (The authors)

Accessed from <http://eprints.qut.edu.au>

Generating Interactive Questionnaires From Configuration Models

Marcello La Rosa¹, Wil M.P. van der Aalst^{2,1}, Marlon Dumas¹,
Arthur H.M. ter Hofstede¹, Florian Gottschalk²

¹ BPM Group, Queensland University of Technology, Australia
{m.larosa, m.dumas, a.terhofstede}@qut.edu.au

² Eindhoven University of Technology, The Netherlands
{w.m.p.v.d.aalst,f.gottschalk}@tm.tue.nl

Abstract. Configuration, be it at the level of models or at the level of code, is a recurrent issue in systems engineering. It arises for example in enterprise systems, where modules are adapted and composed to meet the needs of individual customers based on modifications to a reference model. It also manifests itself in the context of software product families, where variants of a system are built from a common code base. Configuration of such generic systems generally involves two phases: (i) collecting data by answering a set of questions; and (ii) performing certain actions on an existing model or code base to produce an individualized model or system. This paper focuses on the first of these phases. The paper proposes a formal foundation for representing configuration models as well as methods to ensure the consistency of these models and to generate questionnaires from them. The generated questionnaires are interactive, in the sense that questions are only posed if and when they can be answered, and the space of allowed answers to a question is determined by previous answers. The approach has been implemented and tested against a reference model from the logistics domain.

Key words: system configuration, configuration model, interactive questionnaire, product family, reference model

1 Introduction

Modeling and building software systems in a configurable manner is a common approach to achieving reuse and adaptability. For example, enterprise systems packages such as SAP provide collections of modules and business objects covering a range of common functions such as invoicing, financial reporting and controlling [6]. Developers adapt and compose these modules to meet the requirements of individual customers. To guide this individualization process, SAP provides a comprehensive collection of *reference models* including more than 4000 entity types and 1000 business process models and inter-organizational business scenarios [16]. These reference models are configured to meet specific needs, and the resulting configured models in turn, drive the individualization of the system

itself [14]. Similarly, software product families are an increasingly popular approach to package related functionality into generic software assets, from which system variants are generated [7]. Configuration is an integral part of the lifecycle of these systems.

Configuration may involve setting a collection of parameters, selecting a set of features, or more generally, making choices by answering a set of questions. These choices determine the actions (e.g. model or code transformations) to be performed to derive an individualized model or system from a generic one. Referring specifically to the configuration of business process models, which is the motivating scenario used in this paper, such actions may correspond to removing a fragment of a process model. For example, the configuration of a procurement process model may involve a choice between “evaluated receipt settlement” versus “payment against invoice”. In the first case a purchaser pays for goods based on data contained in the delivery receipts; in the second case the purchaser waits for an invoice and pays it only after reconciling it against purchase orders and delivery receipts.

The set of questions to be answered during a system’s configuration are often interdependent. For example, once an evaluated receipt mode has been chosen, questions regarding the configuration of the invoice reconciliation sub-process become irrelevant. Instead, other questions become mandatory. Also, answering a question in a given way may restrict the allowed answers to subsequent questions. Indeed, not all combinations of answers may lead to valid configurations.

This paper proposes a formal foundation for defining system configuration models, focusing on the representation of choices. Specifically, configuration models are represented as *questions*, while the space of possible answers to a question is represented as a set of *facts* that can be set to *true* or *false*. Questions and facts can be connected in arbitrary ways through different types of dependencies, so long as they satisfy certain syntactic criteria. These criteria prevent contradictory dependencies that may lead to deadlocks during the configuration process. The paper also proposes a technique to generate interactive questionnaires from configuration models: these questionnaires guide the configuration process by posing relevant questions in an order consistent with the dependencies between questions and facts. The only major assumption made is that questions have a finite or discretized domain of possible answers. This assumption allows the models to be efficiently analyzed so as to prevent the user from entering conflicting responses to successive questions. Beyond its applicability to systems configuration, the proposed foundation can be applied to build interactive questionnaires for configuring products and services for individual customers.

The remainder of the paper is organized as follows. Section 2 presents a motivating example and outlines the approach. Next, Section 3 presents the formal framework, while Section 4 presents the generation of interactive questionnaires, represented as labeled transition systems, from configuration models. This generation technique has been implemented as a tool outlined in Section 5. This section also shows an example of a configuration process. Finally, Section 6 discusses related work and Section 7 draws conclusions.

2 Motivating Example and Approach

We propose to depict choices independently of specific notations or languages, by means of a set of *facts* that represent the space of possible answers to a set of *questions*. At runtime, questions are answered via an interactive questionnaire that guides the configuration, by posing to users only the relevant questions in an order consistent with the dependencies between questions and facts.

We now examine the approach in detail. To us making a choice corresponds to setting a *fact* within a *question*. Facts are simply *statements* such as “Shipping via DHL” or *features* such as “Return Merchandise Claim”. Initially, each fact is *unset* while at runtime it can be configured by setting its value to *true* or *false*. For example, setting “Shipping via DHL” to *false*, would mean that we are not interested in using DHL for shipping, whilst “Return Merchandise Claim” = *true* would mean that we want to support that type of claim. Each fact has a default value (*true* or *false*) and can be marked as ‘mandatory’, if it needs to be set explicitly by users. Under certain restrictions, a non-mandatory fact can be left *unset* at runtime. In this case its default value is used instead.

Facts are grouped in questions according to their content, so that all the facts of the same group can be set at once by answering the associated question. For example, facts “Return Merchandise Claim” and “Loss or Damage Claim” can be grouped under the question “Which Claims have to be handled?”. Each question features at least one fact and the set of questions must cover all the facts. Although a fact can appear in more than one question, its value can be set only the first time, and must be preserved in all the subsequent questions that contain it. However, an implementation of the questionnaire should support the ability of changing the value of a fact previously set, by rolling back the question that contains it.

A *facts setting* is any combination of facts values where all the facts have been set, either explicitly by answering questions or by using their defaults.

In order to illustrate these concepts, an order fulfilment collaborative process model in the area of supply chain management has been devised, featuring a number of variability points to be configured. The process, based in part on the Voluntary Inter-industry Commerce Standard (VICS) EDI Framework,³ involves three roles, Supplier, Buyer and Carrier, and may support one or more business functions among Product Merchandising, Ordering, Logistics and Payment. In particular, Logistics may comprise one or more sub-phases among Freight Tender, Carrier Appointment, Freight in Transit and Freight Delivered. These phases range over the whole logistics sub-process, from making an offer to a Carrier (Freight Tender), through agreeing on the freight pick-up and delivery details (Carrier Appointment) and on the messages to be exchanged during the shipment (Freight in Transit), to the types of claims to be supported after the delivery (Freight Delivered). The planned usage of a Carrier’s supplied trailer can also be decided upon, and thus configured, based on the size of the freight being shipped. It can be “Truckload” (TL), for full usage, “Less-than Truckload”

³ http://www.uc-council.org/ean_ucc_system/stnds_and_tech/vics_edi.html

(LTL), for partial usage, or “Small Package” (SP), when just single packages are to be shipped. This choice has a strong influence on subsequent decisions. For TL or LTL shipments, the roles responsible for fixing the Pickup and the Delivery appointments can be decided, provided Carrier Appointment is included in Logistics. For the pickup, this role can be played by either the Supplier or the Carrier; for the delivery, by either the Buyer or the Carrier. The appointment negotiation is not allowed in case of SP shipments, as the dates of pickup and delivery are imposed by the Carrier. The Carrier’s usage also affects the type of notifications to be sent during the transit, if Freight in Transit is included in Logistics. For TL or LTL, a Supplier’s or Buyer’s inquiry to the Carrier is followed by a shipment-status message for each parcel of the freight, whilst for SP the inquiry is followed only by one package-status message. Also, only in case of TL or LTL, and if Payment is selected, the Carrier can support a module for charging accessorial costs that may be incurred during the transit. Finally, in Freight Delivered, Claims support can be configured, in order to handle a Merchandise Return and/or cases of Freight Lost or Damaged. If the latter type of claim has been selected, then the Claim Manager is to be chosen, between the Supplier and the Buyer.

A possible structure of questions/facts for the above process is depicted in Fig. 1, and will be used throughout the paper as a motivating example. Here questions and facts are assigned a unique id and a description. For example, facts f_1 to f_4 refer to the four business functions the process may support. These facts are grouped in question q_1 asking for the business functions to be implemented. Question q_2 groups the facts relating to the expected Carrier’s usage. Since this choice is rather important as it affects the process overall, these facts are mandatory (labeled with a \textcircled{M} in the picture), so that they have to be explicitly set when answering q_2 . Other questions would allow users to choose the pickup and delivery managers (q_6, q_7), the claims to be handled (q_4) and the manager for Loss or Damage Claims (q_5). The default values assigned to the facts of Fig. 1 (labeled with a \textcircled{T} in the picture when their default is *true*) refer to a process model featuring all the business functions and the Logistics’s sub-phases, and pitched for TL shipments. In this type of shipment, the Supplier is usually in charge of fixing the Pickup appointment while the Buyer is responsible for Delivery. Also, only Loss or Damage Claims are usually supported, and they are managed by the Supplier which acts as intermediary between the Buyer and the Carrier. Therefore in q_1 facts f_1 to f_4 have default *true*, in q_2 only f_5 (TL) is *true* by default, and so on for the defaults of the other facts.

2.1 Dependencies and Constraints

Dependencies can be defined to fix an order among deciding on facts. For example, we use dependencies to impose that the role responsible for Pickup (either f_{16} or f_{17}) is to be chosen only after deciding on the Carrier Appointment (f_9), as the latter includes the pickup details. We express such dependencies by associating a set of preconditions with a fact x , where a precondition is a group of facts that need to be set before x . Fact x can be set only if at least all the facts in

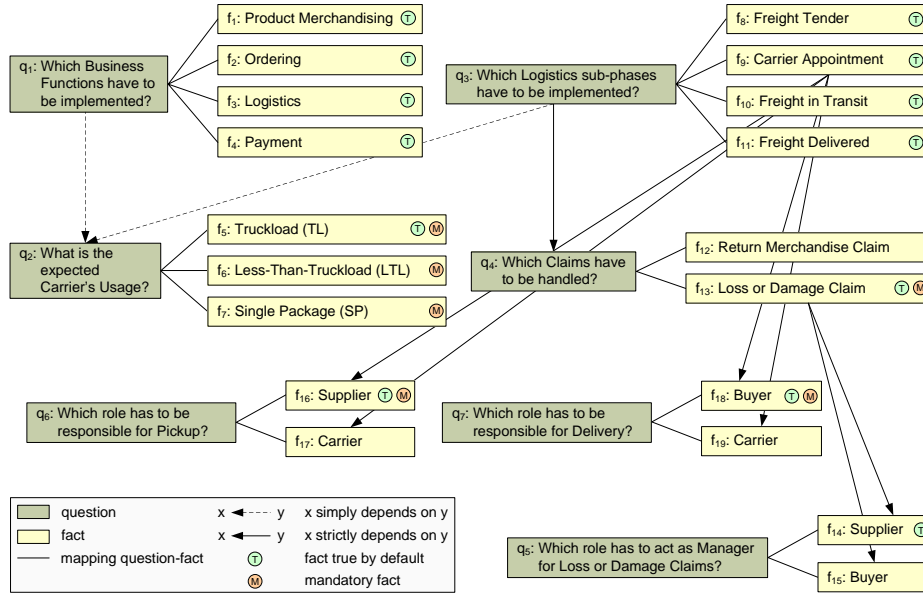


Fig. 1. A possible structure of questions/facts drawn from the VICS EDI Framework.

one of its preconditions have already been set. We say a fact “simply depends” on another fact if the latter belongs at least to one of its preconditions. Also, a fact “strictly depends” on another one if the latter occurs in all its preconditions. A *simple dependency* is represented in Fig. 1 by a dashed arrow connecting a fact to its dependent fact, while a *strict dependency* is depicted by a plain arrow following the same rule. Accordingly, f_{16} and f_{17} strictly depend on f_9 , i.e. they can be set only after f_9 , as they have one precondition containing only f_9 .

Dependencies over facts affect the order of posing questions to users, as questions inherit the dependencies defined for their facts. In our example, since f_{16} in q_6 depends on f_9 in q_3 , then q_6 automatically depends on q_3 , although this dependency is not explicitly shown in Fig. 1. Analogously, q_7 depends on q_3 and q_5 on q_4 . Sometimes, though, it may be more natural to express those dependencies directly at the level of questions, provided the dependencies inherited from facts (if they exist) are not violated. In Fig. 1, q_4 strictly depends (directly) on question q_3 and its facts have no dependencies on other facts, whilst q_2 has a (direct) simple dependency on q_1 and q_3 , so it can be answered after at least one of q_1 and q_3 has been answered.

Dependencies provide a means for ordering questions but do not affect facts values. Answering a question in a given way may restrict the allowed answers to subsequent questions, and not all combinations of answers may lead to valid facts settings. We model interactions over facts values as a set of *constraints* in propositional logic, used to restrict the space of possibilities. The following constraints refer to the facts of Fig. 1:⁴

⁴ \vee indicates the exclusive disjunction (XOR), a commutative and associative relation.

$$\begin{array}{ll}
\text{C1: } f_1 \vee f_2 \vee f_3 \vee f_4 & \text{C2: } f_3 \Leftrightarrow (f_8 \vee f_9 \vee f_{10} \vee f_{11}) \\
\text{C3: } (f_5 \vee f_6 \vee f_7) \Leftrightarrow (f_4 \vee f_9 \vee f_{10}) & \text{C4: } (f_{12} \vee f_{13}) \Rightarrow f_{11} \\
\text{C5: } \neg(f_5 \vee f_6 \vee f_7) \Leftrightarrow \neg(f_4 \vee f_9 \vee f_{10}) & \text{C6: } f_{13} \Leftrightarrow (f_{14} \vee f_{15}) \\
\text{C7: } (f_9 \wedge \neg f_7) \Leftrightarrow ((f_{16} \vee f_{17}) \wedge (f_{18} \vee f_{19})) & \text{C8: } \neg f_{13} \Leftrightarrow \neg(f_{14} \vee f_{15}) \\
\text{C9: } \neg(f_9 \wedge \neg f_7) \Leftrightarrow \neg(f_{16} \vee f_{17} \vee f_{18} \vee f_{19}). &
\end{array}$$

C1 ensures that at least one business function is chosen in q_1 . On the other hand, C3 and C5 state that exactly one type of shipment is to be selected as Carrier's usage in q_2 , if and only if at least one phase among Payment, Carrier Appointment and Freight in Transit is selected, otherwise no shipment type can be chosen. Indeed, as mentioned before, TL, LTL and SP have an influence on the above process phases, so it makes no sense to decide on the shipment type unless a phase affected is selected. Likewise, as per C7 and C9, exactly one role between Supplier and Carrier is to be responsible for Pickup (q_6), and exactly one role between Buyer and Carrier is to be responsible for Delivery (q_7), if and only if Carrier Appointment is selected, and one of TL and LTL is *true*. This is because the process fragments dealing with the management of the appointments are within the TL and LTL fragments of the Carrier Appointment phase.

Dependencies and constraints are not overlapping concepts. Rather, they complement each other, as shown by C4 and the dependency between q_3 and q_4 . Accordingly, claims can be supported only if Freight Delivered 'has been' selected. This is because q_4 strictly depends on q_3 , so Freight Delivered must be set before a claims decision can be made. Similarly, due to C6 and q_5 , which indirectly depends on q_4 , exactly one manager for Loss of Damage Claims is to be selected in q_5 , but only after Loss or Damage Claim 'has been' asserted in q_4 .

In some cases, instead, the sole usage of constraints is sufficient to achieve the desired semantics, as shown by C2. This constraint states that at least one Logistics sub-phase is to be selected in q_3 if and only if this business function 'is' selected in q_1 . Since these two questions are not bound by any dependency, users can start to configure the system either by answering first q_1 or q_3 , and the first answer given will affect the facts values of the other question still unanswered.

Constraints can also be defined over questions (e.g., an *OR* question is a question whose facts are all in *OR* relation). However in the end they need to be traced back to the level of facts. From the above list of constraints it is easy to derive that q_1 is always an *OR* question, while q_3 and q_4 are *OR* questions and q_2 , q_5 , q_6 and q_7 are *XOR* questions, provided some conditions are met. For example, q_5 is an *XOR* question (i.e. exactly one manager can be selected for Loss or Damage Claims), provided $f_{13} = \text{true}$ (i.e. Loss or Damage Claims has been set to *true* in q_4).

A facts setting is a *configuration* if and only if it complies with the constraints over facts values. A configuration is thus the result of answering an interactive questionnaire, where questions are posed to users according to the dependencies, and constraints are dynamically checked in order to prevent users from entering conflicting responses. Although a configuration solely relies on facts values, questions and dependencies are needed to provide a semantically consistent yet simple interface to users, who are only required to fill in a questionnaire, instead of configuring a set of uncorrelated facts.

2.2 Actions

Facts can be associated to sets of actions, i.e. modifications to be performed on the initial model, so as to reflect the effects of a configuration. An action must belong to at least one fact and the set of facts must cover all the actions.

Referring specifically to the configuration of business process models, such an action may correspond, for instance, to removing a process fragment from the initial configurable model, whenever the fact associated to that fragment has been set to *false* in a configuration. Fig. 2 provides an overview of the order fulfilment collaborative process model, represented in the YAWL notation [1].⁵

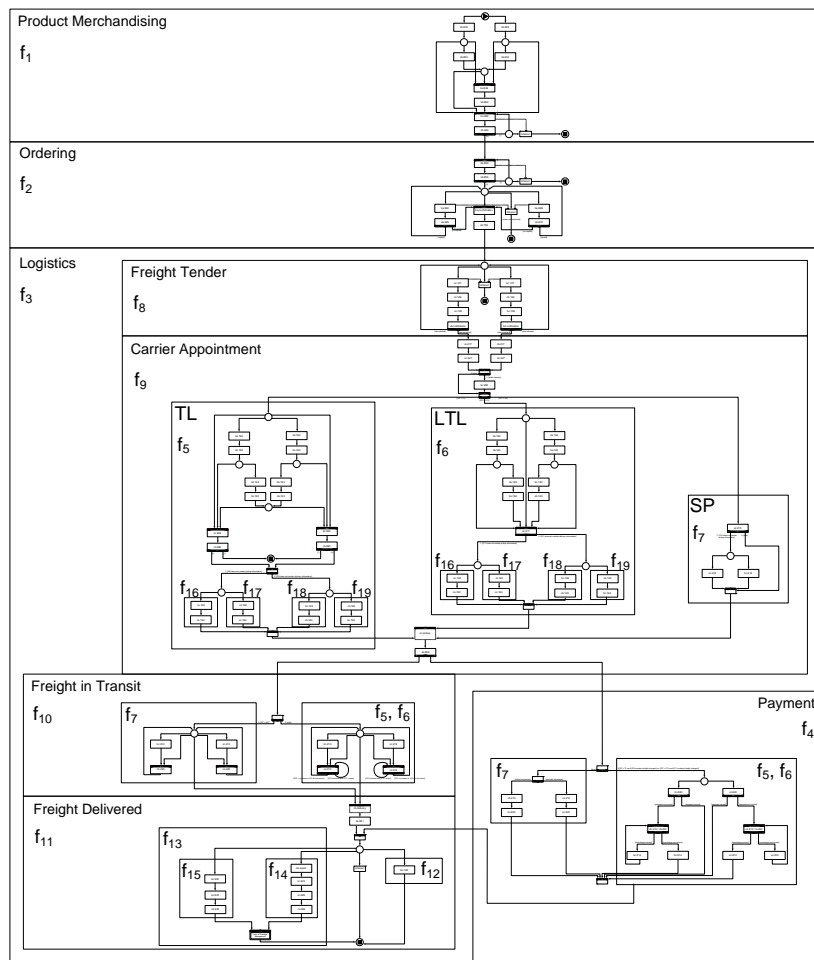


Fig. 2. The order fulfilment collaborative process model, with the facts of Fig. 1.

⁵ A detailed picture can be found at <http://www.fit.qut.edu.au/~dumas/ConfigurationTool.zip>

The process model is divided in a set of configurable process fragments, identified by frames in the picture, which have been associated with the fact(s) of Fig. 1. The four business functions labeled with facts f_1 to f_4 , refer to the four main process fragments and as such, they encompass all the other sub-fragments. Logistics (“ f_3 ”) contains the frames for Freight Tender (“ f_8 ”), Carrier Appointment (“ f_9 ”), Freight in Transit (“ f_{10} ”) and Freight Delivered (“ f_{11} ”). Payment comprises of a sub-fragment for charging accessorial costs in case of TL or LTL shipments (labeled with “ f_5, f_6 ”), and of another one for standard invoicing, used for SP shipments (“ f_7 ”). Carrier Appointment, in turn, includes a sub-fragment for handling each type of shipment independently (“ f_5 ”, “ f_6 ”, “ f_7 ”) and each role that can be responsible for Pickup (“ f_{16} ” and “ f_{17} ”) and for Delivery (“ f_{18} ” and “ f_{19} ”). The last four sub-fragments occur only within frames “ f_5 ” and “ f_6 ”, as only for TL or LTL the pickup and delivery details can be configured. Freight in Transit contains the sub-fragments for handling shipment-status notifications in case of TL or LTL (frame “ f_5, f_6 ”) and package-status notifications in case of SP (frame “ f_7 ”). Freight Delivered contains the sub-fragments for handling claims (“ f_{12} ” and “ f_{13} ”) and the ones related to managing Lost or Damage Claims (“ f_{14} ” and “ f_{15} ”) within the frame labeled with “ f_{13} ”.

For example, removing the Logistics fragment from the process model (i.e. setting f_3 to *false* in q_1) would imply to remove also all the sub-fragments within Logistics. On the other hand, if at least one of those sub-fragments is chosen to be present in the process model, then the Logistics fragment cannot be removed anymore (i.e. f_3 must be asserted in q_1). At the level of facts, these interactions correspond to constraints C2, C3, C5, C7 and C9. Analogous considerations hold for the remaining facts and constraints. Therefore constraints can also be defined over actions, provided in the end they are traced back to the level of facts.

3 Formal Definition of Configuration Models

A *Configuration Model* which directly captures the concepts presented so far is formally defined in this section. The only difference is that we represent constraints over facts values by means of a true table of their conjunction.

Definition 1 (Configuration Model). *A configuration model is a ten-tuple $CM = (F, F_D, F_M, Q, Act, map_{QF}, map_{FA}, pre_F, pre_Q, C)$ where:*

- F is a finite, non-empty set of facts,
- $F_D \subseteq F$ is the default setting, i.e. the set of facts whose default value is true,
- $F_M \subseteq F$ is the set of mandatory facts,
- Q is a finite (non-empty) set of questions,
- Act is a finite set of actions,
- $map_{QF} \in Q \rightarrow \mathcal{P}(F) \setminus \{\emptyset\}$ is a function mapping questions onto sets of facts, such that $\bigcup_{q \in Q} map_{QF}(q) = F$,⁶

⁶ \mathcal{P} indicates the power set.

- $map_{FA} \in F \rightarrow \mathcal{P}(Act)$ is a function mapping facts onto sets of actions, such that $\bigcup_{f \in F} map_{FA}(f) = Act$,
- $pre_F \in F \rightarrow \mathcal{P}(\mathcal{P}(F)) \setminus \{\emptyset\}$ is a function mapping a fact onto a set of sets of facts, where for any $f \in F$, $pre_F(f) \subseteq \mathcal{P}(F \setminus \{f\})$ is the set of preconditions of f , satisfying the following requirements:
 1. $\forall r, p \in pre_F(f) (r \subseteq p \Rightarrow r = p)$, i.e. no redundancies,
 2. $\nexists G \in \mathcal{P}(F) \setminus \{\emptyset\} \forall f \in G \forall F' \in pre_F(f) F' \cap G \neq \emptyset$, i.e. no undesired circular dependencies,
- $pre_Q \in Q \rightarrow \mathcal{P}(\mathcal{P}(Q)) \setminus \{\emptyset\}$ is a function mapping a question onto a set of sets of questions, where for any $q \in Q$, $pre_Q(q) \subseteq \mathcal{P}(Q \setminus \{q\})$ is the set of preconditions of q , satisfying the following requirements:
 1. $\forall r, p \in pre_Q(q) (r \subseteq p \Rightarrow r = p)$, i.e. no redundancies,
 2. $\nexists G \in \mathcal{P}(Q) \setminus \{\emptyset\} \forall q \in G \forall Q' \in pre_Q(q) Q' \cap G \neq \emptyset$, i.e. no undesired circular dependencies,
 3. $\forall Q' \in pre_Q(q) \forall f \in map_{QF}(q) \forall F' \in pre_F(f) F' \subseteq \bigcup_{q' \in Q'} map_{QF}(q')$, i.e. facts dependencies must be preserved at the level of questions,
- $C \subseteq \mathcal{P}(F)$ is the set of the allowed settings of the facts in F , such that $F_D \in C$, i.e. the default setting is always allowed.

Elements of C are those facts settings that satisfy all the constraints, where only the facts asserted are present in each element. Hence, if a fact is not contained in a clause of C , it follows that the fact is negated in that setting. Also, as the default setting must always be allowed, set C is non-empty. If no constraints are defined, $C = \mathcal{P}(F)$. If $C = \{F\}$, all the facts must be asserted (upper-bound case), while $C = \{\emptyset\}$ corresponds to negating all the facts (lower-bound case).

The two types of dependency for facts and questions are formally defined as follows.

Definition 2 (Dependency). Let $CM = (F, F_D, F_M, Q, Act, map_{QF}, map_{FA}, pre_F, pre_Q, C)$ be a configuration model and f, f' and q, q' pairs of facts, resp. questions:

- f simply depends on f' iff $\exists F' \in pre_F(f) f' \in F'$,
- f strictly depends on f' iff $\forall F' \in pre_F(f) f' \in F'$,
- q simply depends on q' iff $\exists Q' \in pre_Q(q) q' \in Q'$,
- q strictly depends on q' iff $\forall Q' \in pre_Q(q) q' \in Q'$.

Example 1. Let $pre_F(f_1) = \{\{f_2, f_3\}, \{f_2, f_4\}\}$ be the set of preconditions of fact f_1 . Then either f_2 and f_3 or f_2 and f_4 have to be set before f_1 can be set. We can observe that f_2 must be set in any case before f_1 , since it appears in all the clauses of $pre_F(f_1)$. On the other hand, f_1 can depend either on f_3 or f_4 , as these facts are not elements of each clause of $pre_F(f_1)$. Note that a strict dependency implies a simple one.

According to the definition, for any fact f and question q , both $pre_F(f)$ and $pre_Q(q)$ are not the empty set. Thus, if we want to model a situation where no dependencies are defined for a fact f , resp. question q , $pre_F(f)$, resp. $pre_Q(q)$,

should contain just the empty set.

The first requirement of pre_F and pre_Q is used to avoid redundancies among preconditions. Accordingly, if a precondition contains the empty set, then it cannot contain other sets, since all the sets would include the empty one. As mentioned before, a fact (question) can be set only if at least all the facts (questions) in one of its preconditions have already been set (answered). This corresponds to saying that facts (questions) in the same precondition are in an *AND* relation, as all of them must be set (answered) before, whilst preconditions are in an *OR* relation, as at least one of them must be satisfied before.

Example 2. A situation where $pre_F(f_1) = \{\{f_2\}, \{f_2, f_3\}\}$ is not allowed since the first clause is a subset of the second. In fact, as all the elements of pre_F (resp. pre_Q) are in an *OR* relation, it does not make sense for f_1 to depend on f_2 *OR* on (f_2 *AND* f_3), as the latter set of dependencies implies the former. In such cases only one clause should be selected.

The second requirement on preconditions avoids ‘undesirable circular dependencies’. Such cases may be caused by both *simple* and *strict* dependencies whenever there exists a set of facts, resp. questions, such that for each element x of the set, all the preconditions of x contain at least an element of the set itself.

Example 3. A case where $pre_F(f_1) = \{\{f_2\}\}$, $pre_F(f_2) = \{\{f_3\}\}$ and $pre_F(f_3) = \{\{f_1\}\}$ (Fig. 3 - a), or a case where $pre_F(f_1) = \{\{f_2\}\}$, $pre_F(f_2) = \{\{f_3\}\}$ and $pre_F(f_3) = \{\{f_1\}, \{f_2\}\}$ (Fig. 3 - b) are excluded as all the preconditions share the same set of facts. By applying the second requirement we see there exists a $G = \{f_1, f_2, f_3\} \subseteq F$ such that for all $f \in G$, all the clauses in $pre_F(f)$ contain at least a fact belonging to G . Note that in the second case both types of dependencies are involved.

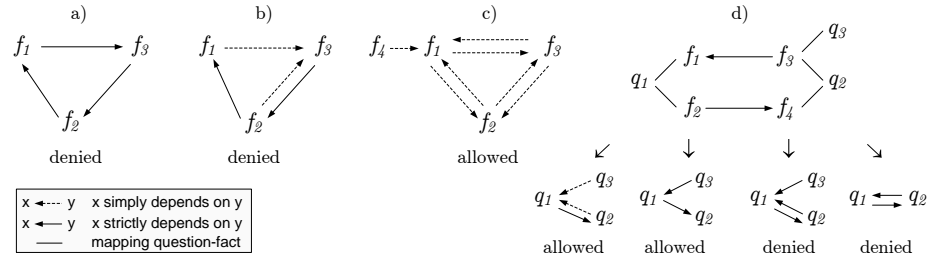


Fig. 3. Examples of circular dependencies over facts and questions.

Not all circular dependencies are undesirable, though. For example, a loop created by a set of facts (questions) can be allowed if there exists an entry point to the loop, i.e. an element of the given set which progressively satisfies all the preconditions. This entry point is a fact (question) with at least one precondition that does not contain any element of the given set.

Example 4. A combination where $pre_F(f_1) = \{\{f_2\}, \{f_3\}, \{f_4\}\}$, $pre_F(f_2) = \{\{f_1\}, \{f_3\}\}$, $pre_F(f_3) = \{\{f_1\}, \{f_2\}\}$, $pre_F(f_4) = \{\emptyset\}$ (Fig. 3 - c) is allowed as f_4 does not have dependencies on the set $\{f_1, f_2, f_3\}$ and thus it first enables f_1 , and then f_2 and f_3 in any order. We cannot actually find a $G \subseteq F$ such that the second requirement on preconditions does not hold.

The only difference between the definitions of pre_F and pre_Q is the addition of a third requirement to the latter, in order to move dependencies over facts to the level of questions, without violating them. Given a question q , the requirement checks for the existence of preconditions F' on the facts of q . If these exist, it forces each precondition Q' of q to contain a set of questions whose facts cover at least all the facts of all the preconditions F' . These dependencies q inherits from its facts, can be extended by adding further dependencies directly at the granularity of questions, provided they comply with the first two requirements. This is possible since $\bigcup_{q' \in Q'} map_{QF}(q')$ is defined as a superset of all the F' .

Example 5. Consider a situation where $map_{QF}(q_1) = \{f_1, f_2\}$, $map_{QF}(q_2) = \{f_3, f_4\}$, $map_{QF}(q_3) = \{f_3\}$, $pre_F(f_1) = \{\{f_3\}\}$ and $pre_F(f_4) = \{\{f_2\}\}$, as shown Fig. 3 - d (where f_3 is a shared fact between q_2 and q_3). By lifting facts dependencies to the level of questions, we see q_3 does not inherit any dependencies, q_2 strictly depends on q_1 , while there are four possible sets of preconditions for q_1 , i.e. $pre_Q(q_1) = \{\{q_2\}, \{q_3\}\}$ or $\{\{q_3\}\}$ or $\{\{q_2, q_3\}\}$ or $\{\{q_2\}\}$. All these sets meet the third requirement, as f_3 – the only fact f_1 depends on – is contained in at least one question $q' \in Q'$ for each $Q' \in pre_Q(q_1)$. However, according to the second requirement, only the first two alternatives are valid, as they do not create undesirable circular dependencies between q_1 and q_2 .

4 Generation of Interactive Questionnaires

This section rounds off the formalization presented so far by defining the “runtime behavior”, i.e. the actual configuration process for a *configuration model*. Here questions are generated dynamically according to the dependencies, and answers can be given only if they comply with the constraints.

Definition 3 formally defines the concepts of *fact valuation*, *answer*, *state* and *state space*.

Definition 3 (Fact valuation, Answer, State, State space). Let $CM = (F, F_D, F_M, Q, Act, map_{QF}, map_{FA}, pre_F, pre_Q, C)$ be a configuration model:

- $V = F \rightarrow \{true, false, unset\}$ is the set of facts valuations (i.e. facts settings),
- $a \in V$ is an answer, i.e. a facts valuation where all $f \in F$ for which $a(f) \neq unset$ are set,
- $s = (vs, qs)$ is a state of CM if and only if $vs \in V$ and $qs \subseteq Q$, where qs is the set of questions answered and vs is the valuation of the facts thus far,
- $S_{CM} = V \times \mathcal{P}(Q)$ is the state space of CM .

Elements of V are thus “parts of state” (vs) as well as “answers” (a). Hereafter S_{CM} is shortened to S whenever the configuration context is clear.

In order to perform operations on facts valuations, we define the following notation.

Definition 4 (Facts Valuation Notation). *Let $CM = (F, F_D, F_M, Q, Act, map_{QF}, map_{FA}, pre_F, pre_Q, C)$ be a configuration model and let $s = (vs, qs) \in S$ be a state of CM and $a \in V$ an answer:*

- $t(s) = t(vs) = \{f \in F \mid vs(f) = true\}$ is the set of facts that are true in state s ,
- $f(s) = f(vs) = \{f \in F \mid vs(f) = false\}$ is the set of facts that are false in state s ,
- $u(s) = u(vs) = \{f \in F \mid vs(f) = unset\} = F \setminus (t(s) \cup f(s))$ is the set of facts that are unset in state s . Note that $t(vs)$, $f(vs)$ and $u(vs)$ can be applied to any valuation $vs \in V$, thus to any answer $a \in V$:
- $t(a) = \{f \in F \mid a(f) = true\}$, is the set of facts set to true by answer a ,
- $f(a) = \{f \in F \mid a(f) = false\}$, is the set of facts set to false by answer a ,
- $u(a) = F \setminus (t(a) \cup f(a))$, is the set of facts left unset by answer a ,
- $compl(s) = compl(vs) = \{f \in F \mid vs(f) = true \vee (f \in F_D \wedge vs(f) \neq false)\}$ is the set of facts set to true through answers, merged with those facts left unset, which were true by default,⁷
- for $x, y \in V$ and $f \in F$:

$$x \oplus y(f) \begin{cases} true, & \text{if } y(f) = true \vee (x(f) = true \wedge y(f) = unset), \\ false, & \text{if } y(f) = false \vee (x(f) = false \wedge y(f) = unset), \\ unset, & \text{otherwise.} \end{cases}$$

For each state a set of *valid* questions are presented to users. For a question to be valid in a state ($valid(q, s)$), two conditions must hold: a) it must not have been answered yet, b) at least one of its preconditions needs to be satisfied (i.e. all the questions within an element of $pre_Q(q)$ must have been answered before).

Users can answer one valid question at a time. However, an answer to a question in a certain state is valid ($valid(a, q, s)$) if and only if all the facts within that question are set. Besides, since facts can appear in more than one question, those of them already set in previous questions must keep their values in the answer, i.e. it is possible to reconfirm answers, and the outcome of the answer ($outcome(a, q, s)$) must result in a valid state ($valid(s)$), i.e. a state whose facts valuation complies with the constraints on facts.

Definition 5 (Valid answer). *Let $CM = (F, F_D, Q, Act, map_{QF}, map_{FA}, pre_F, pre_Q, F_M, C)$ be a configuration model and let $s = (vs, qs) \in S$ be a state of CM , $q \in Q$ a question, and $a \in V$ an answer:*

- $valid(q, s) = q \notin qs \wedge \exists_{Q' \in pre_Q(q)} Q' \subseteq qs$, i.e., question q may be asked if it has not been answered yet and at least a group of preceding questions has been answered,

⁷ This function is used to replace the truth value of *unset* facts to their default value.

- $outcome(a, q, s) = (vs \oplus a, qs \cup \{q\})$, i.e. the state resulting after answering a to question q in state s ,
- $valid(s) = \exists_{F' \in C} (t(s) \subseteq F' \wedge f(s) \cap F' = \emptyset)$, i.e. the facts valuation of the state has to comply with the constraints on facts,
- $valid(a, q, s) = valid(q, s) \wedge t(a) \cup f(a) = map_{QF}(q) \wedge \forall_{f \in map_{QF}(q) \setminus u(s)} a(f) = vs(f) \wedge valid(outcome(a, q, s))$, i.e. a valid answer to a valid question has to set all the facts of the question without changing the value of the facts already set, and the given valuation must result in a valid state.

The valuation resulting from an answer has to be checked against set C , so as to verify if it complies with the constraints defined on facts values. In this way we ensure it is always possible to complete the current facts valuation by setting the remaining facts still *unset* (if they exist).

By joining the possible states of a configuration process, we can now build a labeled transition system on top of a configuration model. This is used later on to formally define the concept of *configuration*.

Definition 6 (Labeled Transition System of CM). Let $CM = (F, F_D, F_M, Q, Act, map_{QF}, map_{FA}, pre_F, pre_Q, C)$ be a configuration model and let S be the state space of CM and V the set of facts valuations. The labeled transition system of CM is a five-tuple $LTS = (S_v, L, T, s_{init}, S_F)$ where:

- $S_v = \{s \in S \mid valid(s)\}$ is the set of states of LTS , corresponding to the valid states of CM ,
- $L = \{(a, q) \in V \times Q \mid t(a) \cup f(a) = map_{QF}(q)\}$ is the set of transition labels of LTS , where each element of L is a pair composed of an answer and a question of CM ,
- $T = \{(s, (a, q), s') \in S_v \times L \times S_v \mid valid(a, q, s) \wedge s' = outcome(a, q, s)\}$ is the set of transitions of LTS , where for each $t = (s, (a, q), s') \in T$ $source(t) = s$ and $target(t) = s'$,
- $s_{init} = \{(f, unset) \mid f \in F\}, \emptyset \in S_v$ is the initial state of LTS , i.e. the state in which all the facts are *unset* and all the questions are *unanswered*,⁸
- $S_F = \{(vs, qs) \in S_v \mid (f \in F_M \Rightarrow vs(f) \neq unset) \wedge valid(s^*)\}$ is the set of final states of LTS , where $s^* = (vs^*, qs) \in S$ with $t(vs^*) = compl(vs)$ and $f(vs^*) = F \setminus t(vs^*)$. A final state is a state where all the mandatory facts have been set, and the facts still *unset*, if these exist, can take their default value without violating the constraints on facts.

A configuration process always starts from an initial state where no questions are answered and all the facts are *unset*, and terminates in a final state where all the questions have been answered or, all the mandatory facts have been set and the remaining *unset* facts can take their default values. As shown in the definition of final state, this is only possible if the facts valuation resulting after applying the defaults complies with the constraints on facts values, i.e. if it does not violate the configuration process so far.

⁸ s_{init} is valid by definition, since $t(s_{init}) = f(s_{init}) = \emptyset$.

Example 6. Consider a configuration model where $map_{QF}(q_1) = \{f_1\}$, $map_{QF}(q_2) = \{f_2, f_3, f_4, f_5\}$, $F_D = \{f_2, f_3\}$, $F_M = \{f_1\}$, and the constraint $f_1 \Rightarrow ((f_2 \wedge f_4) \vee (f_3 \wedge f_5))$. It follows that $C = \{\{f_1, f_2, f_4\}, \{f_1, f_3, f_5\}, \dots\}$, where the remaining elements of C are the elements of $\mathcal{P}(\{f_2, f_3, f_4, f_5\})$, thus including F_D . If f_1 is set to *true* by answering q_1 , although all the mandatory facts have been set, the default setting cannot be applied for the remaining unset facts, since only either f_2 and f_4 or f_3 and f_5 can assume value *true*, hence we cannot find an $F' \in C$ such that $\{f_1, f_2, f_3\} \subseteq F'$. On the other hand, by setting f_1 to *false*, we arrive straightaway at a final state, as the remaining facts can take their default value.

A *configuration trace* of CM is a sequence of transitions of LTS linking the initial state to a final state.

Definition 7 (Configuration Trace of CM). Let $CM = (F, F_D, F_M, Q, Act, map_{QF}, map_{FA}, pre_F, pre_Q, C)$ be a configuration model, V the set of facts valuations, S the state space of CM and let $LTS_{CM} = (S_v, L, T, s_{init}, S_F)$ be its labeled transition system:

- $\sigma = (t_1, \dots, t_n) \in T^+$ is a trace of LTS iff $target(t_i) = source(t_{i+1})$ for each $1 \leq i \leq n-1$, where $first_s(\sigma) = source(t_1)$ and $last_s(\sigma) = target(t_n)$,
- $valid(\sigma) = (first_s(\sigma) = s_{init} \wedge last_s(\sigma) \in S_F)$, i.e. a trace is valid iff it joins the initial state with a final state. Each valid trace is a configuration trace of CM .

A *configuration* of CM is the result of any configuration trace of CM , i.e. a facts valuation corresponding to the facts valuation of the last state of a configuration trace of CM , completed with default values.

Definition 8 (Configuration of CM , Configuration Space of CM). Let $CM = (F, F_D, F_M, Q, Act, map_{QF}, map_{FA}, pre_F, pre_Q, C)$ be a configuration model, V the set of facts valuations, S the state space of CM , $LTS_{CM} = (S_v, L, T, s_{init}, S_F)$ its labeled transition system, and let $\sigma \in T^+$ be a configuration trace of CM :

- $cf(\sigma) \in V$ is a configuration of CM resulting from σ , iff $t(cf(\sigma)) = compl(last_s(\sigma))$ and $f(cf(\sigma)) = F \setminus t(cf(\sigma))$,
- $Cf_{CM} = \{cf(\sigma) \in V \mid valid(\sigma)\}$ is the configuration space of CM , i.e. the set of all the possible configurations of CM .

A configuration is thus a facts setting that complies with the constraints.

We now show that a configuration process can always terminate in a final state, since for all the valid non-final states, there always exists at least one valid question whose answer leads to another valid state, taking the process closer to a final state. In particular, the purpose of the following theorem is to prove that the definition of pre_Q and C are sufficient to avoid any deadlock during the configuration process, since undesirable circular dependencies are excluded a priori and set C is a representation of those facts settings that comply with the constraints.

Definition 9 (Trace Notation). Let $CM = (F, F_D, F_M, Q, Act, map_{QF}, map_{FA}, pre_F, pre_Q, C)$ be a configuration model, V the set of facts valuations, S the state space of CM and let $LTS_{CM} = (S_v, L, T, s_{init}, S_F)$ be its labeled transition system. Given two valid states of LTS s and s' , we write $s \xrightarrow{\sigma} s'$ iff $\sigma \in T^+$ is a trace of LTS such that $first_s(\sigma) = s$ and $last_s(\sigma) = s'$.

Theorem 1. Let $CM = (F, F_D, F_M, Q, Act, map_{QF}, map_{FA}, pre_F, pre_Q, C)$ be a configuration model, V the set of facts valuations, S the state space of CM and let $LTS_{CM} = (S_v, L, T, s_{init}, S_F)$ be its labeled transition system. For any $s \in S_v$, either $s \in S_F$ or $\exists q \in Q \exists a \in V \exists s' \in S_v s \xrightarrow{(s, (a, q), s')} s', (s, (a, q), s') \in T$.

Proof. We prove the theorem in two steps: first, we show for all valid non-final states there always exists at least one valid question; second, we show for all valid questions in a valid state, there always exists at least one valid answer.

Valid question $[\forall_{s \in S_v \setminus S_F} \exists_{q \in Q} \text{valid}(q, s)]$. Let $s = (vs, qs) \in S_v \setminus S_F$. Let $G = Q \setminus qs$, then $G \neq \emptyset$ as $s \notin S_F$. According to the 2nd requirement of pre_Q , there is a $q \in G$ and a $Q' \in pre_Q(q)$ such that $G \cap Q' = \emptyset$.

- $[q \notin qs]$. True by definition of G and pre_Q .
- $[Q' \subseteq qs]$. $G \cap Q' = \emptyset$, that is $(Q \setminus qs) \cap Q' = \emptyset$, thus $(Q \cap Q') \setminus qs = \emptyset$, $(Q' \subseteq Q) Q' \setminus qs = \emptyset$, hence $Q' \subseteq qs$.

Hence $\text{valid}(q, s)$.

Valid answer $[\forall_{s \in S_v \setminus S_F} \forall_{q \in Q, \text{valid}(q, s)} \exists_{a \in V} \text{valid}(a, q, s)]$. Let $s = (vs, qs) \in S_v \setminus S_F$. Since $s \in S_v$, we can find $F' \in C$ such that $t(s) \subseteq F'$ and $f(s) \cap F' = \emptyset$. Let $q \in Q$ such that $\text{valid}(q, s)$. We define $t_s(q) = \{f \in map_{QF}(q) \mid vs(f) = \text{true}\}$, $f_s(q) = \{f \in map_{QF}(q) \mid vs(f) = \text{false}\}$, $t_u(q) = (F' \cap map_{QF}(q)) \setminus t_s(q)$ and $f_u(q) = map_{QF}(q) \setminus (F' \cup t_s(q))$. We choose $a = \{(f, \text{true}) \mid f \in t_s(q) \cup t_u(q)\} \cup \{(f, \text{false}) \mid f \in f_s(q) \cup f_u(q)\} \cup \{(f, \text{unset}) \mid f \in F' \setminus map_{QF}(q)\}$, then $a \in V$.

- $[\text{valid}(q, s)]$. True by assumption.
- $[t(a) \cup f(a) = map_{QF}(q)]$. $t(a) \cup f(a) = t_s(q) \cup t_u(q) \cup f_s(q) \cup f_u(q)$.
 - $[\subseteq]$ Let $f \in map_{QF}(q)$,
 - 1) if $vs(f) = \text{true}$, then $f \in t_s(q)$,
 - 2) if $vs(f) = \text{false}$, then $f \in f_s(q)$,
 - 3) if $vs(f) = \text{unset}$,
 - a) if $f \in F'$, then $f \in t_u(q)$ as $f \notin t_s(q)$,
 - b) if $f \notin F'$, then $f \in f_u(q)$ as $f \notin t_s(q)$,
 hence $f \in t_s(q) \cup t_u(q) \cup f_s(q) \cup f_u(q)$.
 - $[\supseteq]$ Follows from the definitions of $t_s(q), t_u(q), f_s(q)$ and $f_u(q)$.
- $[\forall_{f \in map_{QF}(q) \setminus u(s)} a(f) = vs(f)]$. Let $f \in map_{QF}(q)$ and $f \notin u(s)$, then $f \in t_s(q)$ or $f \in f_s(q)$, hence (definition of a) $a(f) = \text{true}$ and $f \in t_s(q)$ or $a(f) = \text{false}$ and $f \in f_s(q)$, hence (definitions of $t_s(q)$ and $f_s(q)$) $a(f) = \text{true}$ and $vs(f) = \text{true}$ or $a(f) = \text{false}$ and $vs(f) = \text{false}$, hence $a(f) = vs(f)$.
- $[\text{valid}(\text{outcome}(a, q, s))]$. Let $s' = \text{outcome}(a, q, s) = (vs \oplus a, qs \cup \{q\})$.

- $[t(s') \subseteq F']$. $t(s') = \{f \in F \mid a(f) = \text{true} \vee (vs(f) = \text{true} \wedge a(f) = \text{unset})\}$ (definition of $x \oplus y(f)$). Let $f \in t(s')$,
 - 1) if $a(f) = \text{true}$, then $f \in t_s(q) \cup t_u(q)$, hence $f \in F'$ given that $t_s(q) \subseteq F'$ and $t_u(q) \subseteq F'$.
 - 2) if $vs(f) = \text{true}$ and $a(f) = \text{unset}$, then $f \in t(s)$ and $f \in F \setminus \text{map}_{QF}(q)$, hence $f \in F'$ as $t(s) \subseteq F'$.
- $[f(s') \cap F' = \emptyset]$. $f(s') = \{f \in F \mid a(f) = \text{false} \vee (vs(f) = \text{false} \wedge a(f) = \text{unset})\}$ (definition of $x \oplus y(f)$). Let $f \in f(s')$,
 - 1) if $a(f) = \text{false}$, then $f \in f_s(q) \cup f_u(q)$, hence $f \notin F' = \emptyset$ given that $f_s(q) \cap F' = \emptyset$ and $f_u(q) \cap F' = \emptyset$.
 - 2) if $vs(f) = \text{false}$ and $a(f) = \text{unset}$, then $f \in f(s)$ and $f \in F \setminus \text{map}_{QF}(q)$, hence $f \notin F'$ as $f(s) \cap F' = \emptyset$.

Hence $\text{valid}(\text{outcome}(a, q, s))$.

Hence $\text{valid}(a, q, s)$.

Corollary 1 (Configuration processes always terminate). *For any configuration model $CM = (F, F_D, F_M, Q, Act, \text{map}_{QF}, \text{map}_{FA}, \text{pre}_F, \text{pre}_Q, C)$ and its LTS $CM = (S_v, L, T, s_{init}, S_F)$, and for any state $s \in S_v \setminus S_F$ for which there exists a trace $\sigma \in T^+$ such that $s_{init} \xrightarrow{\sigma} s$, there exists a $\tau \in T^+$ and an $s' \in S_F$ such that $s \xrightarrow{\tau} s'$, i.e. each configuration process can reach a final state.*

In general, before starting the configuration process, a fact can assume both the values *true* and *false*. However once the configuration process begins, at a certain state it may turn out from the constraints that a fact can take only one value of the two. In this case users do not have the freedom to choose, as the value to be given is imposed by the constraints. We call this type of fact *forceable*.

When this situation occurs for all the facts of a question, the question can have only one answer. Moreover, since facts can appear in more than one question, it may happen at a certain state, that all the facts of a valid question have already been answered. Again, this question can take only one possible answer. We call these questions *skippable*, as they can be automatically answered by the questionnaire and thus skipped. These concepts are formally defined as follows.

Definition 10 (Skippable Question). *Let $CM = (F, F_D, F_M, Q, Act, \text{map}_{QF}, \text{map}_{FA}, \text{pre}_F, \text{pre}_Q, C)$ be a configuration model, and let $s \in S$ be a valid state of CM , $f \in F$ a fact and $q \in Q$ a question:*

- $\text{forceable}(f, s) = f \in u(s) \wedge \forall_{F_1, F_2 \in C} [(t(s) \subseteq F_1 \cap F_2 \wedge f(s) \cap (F_1 \cup F_2) = \emptyset) \Rightarrow F_1(f) = F_2(f)]$, i.e. f assumes the same value in all the facts valuations still possible,
- $\text{skippable}(q, s) = \text{valid}(q, s) \wedge \forall_{f \in \text{map}_{QF}(q)} [\text{forceable}(f, s) \vee f \notin u(s)]$, i.e. a question can be skipped if all its unset facts can have exactly one value or if all its facts have been previously set.

Whether all the facts of a question are forceable to a value or have already been set, the only possible answer to that question will be valid, since its valuation always complies with the constraints. In fact, whether a fact is forceable or not is determined by using C , while if all the facts have been set previously, then the answer is already included in the last state s , which is valid by assumption.

5 Tool Support

A prototype implementation of a tool for the dynamic generation of interactive questionnaires has been developed during the course of this research. The features of this tool, called *Quaestio*, are introduced in the first part of this section. The second part shows how the tool is used to configure the order fulfilment example of Section 2.

5.1 Prototype Implementation

Quaestio is a simple Java GUI which produces a set of ordered questions given a configuration model as input.⁹ An XML-Schema file has been defined to describe the input format. The rules imposed in the formalization of *CM* (see Definition 1) have been coded as syntactical rules in the schema itself, in order to avoid non-well-formed configuration models, i.e. models where a fact is not associated to any question, where the questions do not cover all the facts, etc.

The interface of the tool is made up of a main window showing a list of Valid Questions, a list of Answered Questions and a Question Inspector. When a question is picked from one of these lists, the Question Inspector shows the question id, a list of facts (each of them featuring a button for the selection of the fact value and a description), guidelines for configuring the question and the dependencies on other questions. A Fact Inspector can be opened as a separate window, so that whenever a fact is selected from a question, all the information about that fact can pop up. These include the fact description and id, the impact level in the configuration process, the default value, whether the fact is mandatory, the questions the fact appears in, the constraints that bind the fact, the dependencies on other facts and specific guidelines for configuring the fact.

Quaestio loads the configuration model and shows the set of initial valid questions. Next, for each answer given, it calculates the next valid state and updates the Valid Questions list and Answered Questions list. The configuration process completes once all the questions have been answered, or at least all the mandatory facts have been set and the remaining ones can take their defaults without violating the constraints.

The result of a configuration process can be exported as configuration, which is an XML file featuring for each fact the id and description, the value that has been set and whether the fact deviates from its default. A summary of the tool's main features is given hereafter:

- answering a question: an answer can be given to a valid question only when the valuation results in a valid state,
- default answer: the default answer can be given to a valid question (default values are used for all the facts in the question), if:
 - the value of facts already set or forceable do not deviate from their default, and

⁹ The tool can be downloaded from <http://sky.fit.qut.edu.au/~dumas/ConfigurationTool.zip>

- the resulting valuation is a valid state,
- occurrence of facts: when facts appearing in more than one question are set, they preserve their value and are shown as disabled in subsequent questions they appear in,
- forceability of facts: those facts which turn out to be forceable according to the constraints, are disabled and show their forced value;
- skippable questions: these questions are automatically answered by the system and put in the list of answered questions;
- automatic completion: the system can automatically complete the configuration process whenever all the mandatory facts have been answered and the default value can be used for the remaining facts. This is given as an option to users, who can decide whether or not to continue the configuration manually;
- question rollback: each answered question can be rolled back to the state before answering that question. This implies to roll back also all the questions answered thereafter (if they exist).

The tool adheres to the formalization presented in Section 3 and 4. The only difference is in the internal representation of the constraints on facts. Indeed, checking constraints based on C – which is a representation of all the facts settings that comply with the constraints – would not be an efficient operation. To overcome this issue, the constraints checking module of Quaestio embodies an existing calculator¹⁰ based on Shared Binary Decision Diagrams (SBDDs) [3, 11]. SBDDs are a concise representation of a boolean formula for which there are efficient constraint checking algorithms. It has been proven that algorithms based on SBDDs can efficiently deal with systems made up of around one million of possibilities [11].

Accordingly, Quaestio can scale with configuration scenarios made up of thousands of facts and around one million of possible configurations.

5.2 Sample Configuration Process

Assume we want to configure the order fulfilment process model of Fig. 2 for handling SP shipments. Assume also we want to support only Return Merchandise claims and we are not interested in the Payment phase of the process.

Once the corresponding configuration model has been loaded into Quaestio, the valid questions are shown in the Valid Questions list (Fig. 4). They are q_1 and q_3 , as only these questions have no dependencies. The initial state is s_1 where no answers have been given, i.e. $qs(s_1) = \emptyset$. We then answer q_3 – *Which Logistics phases have to be implemented?* with its default answer, by pressing the Default Answer button. This operation corresponds to give answer $a_1(q_3) = \{(f_8, \top), (f_9, \top), (f_{10}, \top), (f_{11}, \top)\}$, as all the facts of q_3 are *true* by default. When the default value of a fact is *true*, a green \oplus is shown next to each fact description (Fig. 4).

¹⁰ Downloadable from <http://www-verimag.imag.fr/~raymond/tools/bddc-manual>

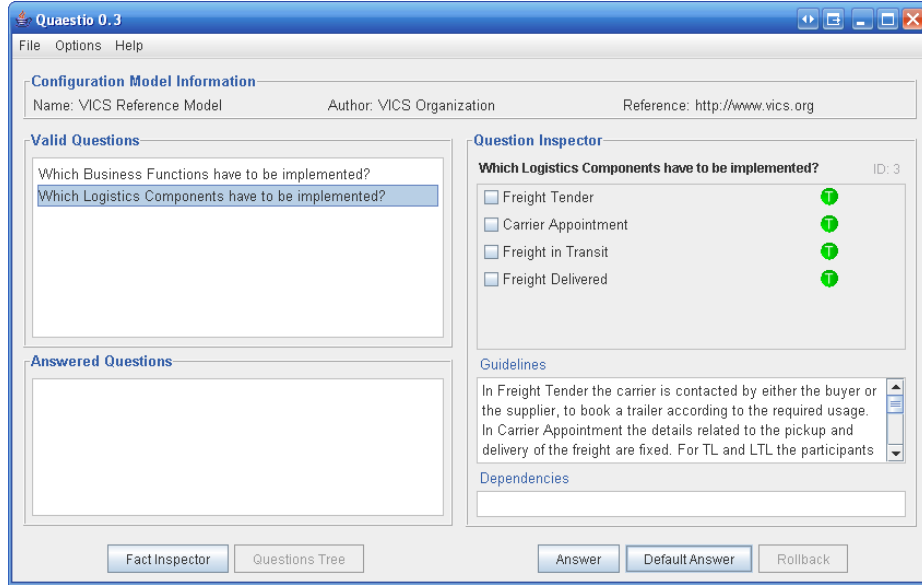


Fig. 4. State s_1 : the only valid questions are q_1 and q_3 .

With a_1 we reach state s_2 with $qs(s_2) = \{q_3\}$. q_2 adds to the valid questions due to its simple dependency on q_1 or q_3 . We then select q_1 in the Valid Questions list. We see that f_3 has been forced to *true* and cannot be selected anymore (Fig. 5). The system has in fact reacted to a_1 and set f_3 to *true* in order to comply with C2. We answer q_1 with $a_2(q_1) = \{(f_1, T), (f_2, T), (f_3, T), (f_4, F)\}$, so as to exclude Payment. In this case we do not use the Default Answer button, as we want to deviate from the default setting of this question. Guidelines for questions and facts can be consulted by users to gather information on how to configure a question or a single fact, as shown in Fig. 4 and 5.

After a_2 the current state becomes s_3 with $qs(s_3) = \{q_3, q_1\}$. Questions q_4, q_6 and q_7 are added to the valid ones as they depend on q_3 . We pick q_2 – *What is the expected Carrier's Usage?* from the list of valid questions. Panel Dependencies in the Question Inspector shows that this question depends on $q_1 \vee q_3$ and the Answer button is disabled. According to C3 and to the answers given so far, q_2 can only be answered if exactly one of its facts is asserted. As q_2 is an *XOR* question in s_3 , Quaestio shows radio buttons for its facts. Moreover, this question needs to be explicitly answered as all its facts are mandatory. In Quaestio a mandatory fact is depicted with a red \textcircled{M} next to its description. As we select Single Package, the Answer button enables itself. Thus by pressing it, we give answer $a_3(q_2) = \{(f_5, F), (f_6, F), (f_7, T)\}$.

The next state is s_4 with $qs(s_4) = \{q_3, q_1, q_2\}$. Although no questions depend on q_2 , after answering a_3 , both q_6 and q_7 become skippable, as all of their facts can take only value *false*, due to C9. Thus $a_4(q_6) = \{(f_{16}, F), (f_{17}, F)\}$ and $a_5(q_7) = \{(f_{18}, F), (f_{19}, F)\}$ are automatically given by the system, which moves

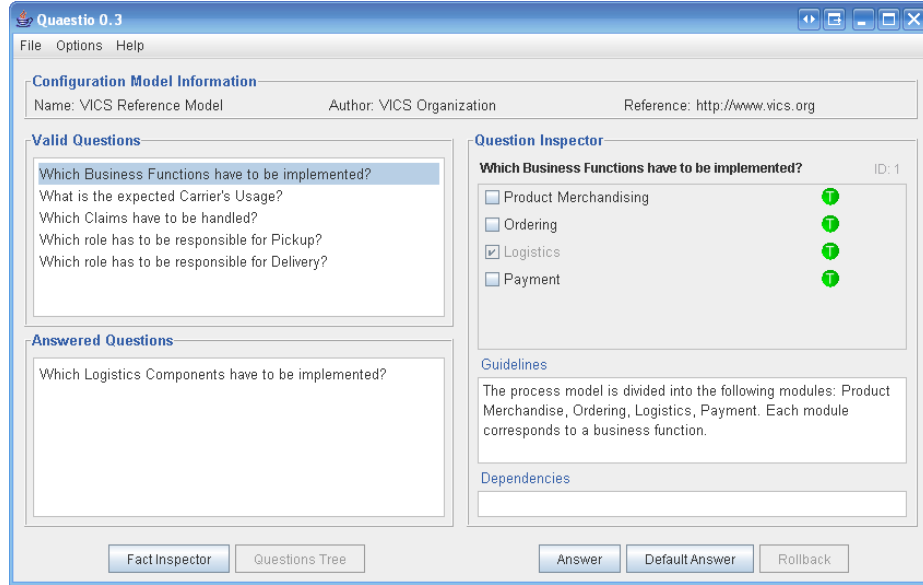


Fig. 5. State s_2 : f_3 has been forced to *true* in order not to violate C2.

from s_4 to s_5 with a_5 , and from s_5 to s_6 with a_6 . Questions q_6 and q_7 are added to the set of answered ones ($qs(s_6) = \{q_3, q_1, q_2, q_6, q_7\}$). In Quaestio skipped questions are shown in blue within the Answered Questions list (Fig. 6). We then answer the only valid question remained, q_4 – *Which claims have to be handled?* with $a_6(q_4) = \{(f_{12}, F), (f_{13}, T)\}$.

After a_6 we reach s_7 with $qs(s_7) = \{q_3, q_1, q_2, q_6, q_7, q_4\}$ and q_5 – *Which role has to act as Manager for Loss or Damage Claims?* becomes valid as it depends on q_4 . s_7 is a final state as all the mandatory facts have already been set and the remaining ones still unset (f_{14} and f_{15}) can take their defaults without violating the constraints. q_4 can thus be answered automatically with defaults. At this point users can decide whether to continue or to complete the configuration automatically. We decide to complete the configuration automatically, and the system answers with $a_7(q_5) = \{(f_{14}, T), (f_{15}, F)\}$, according to the default values.

State s_8 is the next state with $qs(s_8) = \{q_3, q_1, q_2, q_6, q_7, q_4, q_5\}$. The configuration process could terminate here, however, since we are interested only in Return Merchandise claims, we decide to rollback q_4 in order to re-answer it. In fact we configured this question to support Loss or Damage claims and not Return Merchandise claims. To rollback a question, we simply select it from the list of the answered questions and press the Rollback button, that is became enabled. The system restores the current state to s_6 , i.e. the state before answering q_4 . We then answer $a_6(q_4) = \{(f_{12}, T), (f_{13}, F)\}$ and reach s_7 again. This time, though, q_5 is skippable as the only valid answer is $a_7(q_5) = \{(f_{14}, F), (f_{15}, F)\}$. With a_7 we reach again s_8 and complete.

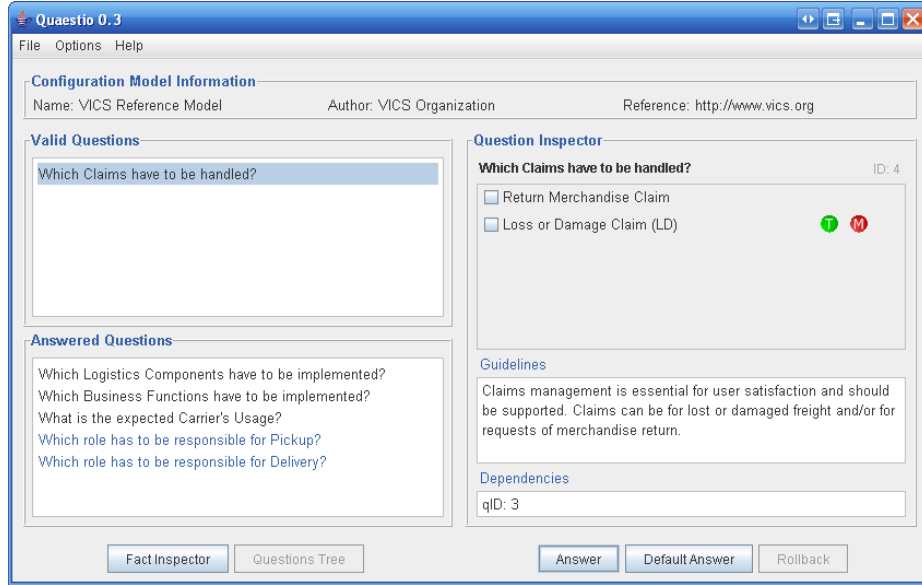


Fig. 6. State s_6 . q_6 and q_7 have been skipped as they facts can only be negated.

The corresponding configuration trace is $\sigma = \{(s_1, (a_1, q_3), s_2), (s_2, (a_2, q_1), s_3), (s_3, (a_3, q_2), s_4), (s_4, (a_4, q_6), s_5), (s_5, (a_5, q_7), s_6), (s_6, (a_6, q_4), s_7), (s_7, (a_7, q_5), s_8)\}$, and the configuration is $cf(\sigma) = \{(f_1, T), (f_2, T), (f_3, T), (f_4, F), (f_5, F), (f_6, F), (f_7, T), (f_8, T), (f_9, T), (f_{10}, T), (f_{11}, T), (f_{12}, T), (f_{13}, F), (f_{14}, F), (f_{15}, F), (f_{16}, F), (f_{17}, F), (f_{18}, F), (f_{19}, F)\}$.

The above configuration leads to the order fulfilment process model pictured in Fig. 7.

6 Related Work

Configuration modeling has been widely studied in the field of Software Configuration Management (SCM). Work in this field has resulted in models and languages to capture how a collection of available options impact upon the way a software system is built from a set of components. This is the case for example of the Adele Configuration Manager [8] and the Proteus Configuration Language (PCL) [17]. Adele supports the definition of dependencies between artefacts composing a software family, such as all-or-none or exclusion dependencies between interfaces and realizations thereof (e.g. “only one realization of an interface should be included in any instance of the family”). Such dependencies are expressed in a first-order logic language using attributes defined on objects, where an object represents a software artefact. Building a configuration in Adele involves selecting a collection of objects that satisfy all constraints. Similarly, PCL supports the representation of variability in the structure and in the process of building a software system using *variability control attributes*.

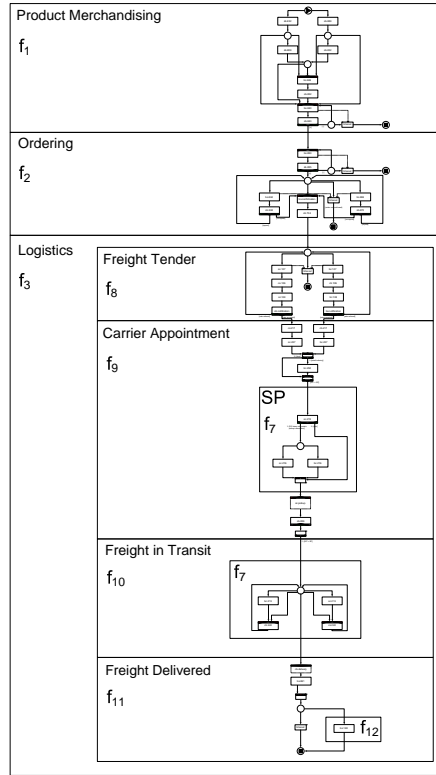


Fig. 7. The configured order fulfilment process model.

These attributes determine which actions will be performed to build a variant of a system. For example, one can capture in PCL that a sub-system, such as a graphical interface, is optional, or that a sub-system maps to different sets of program files depending on the value given to certain variability attributes.

However, neither Adele nor Proteus deal with guiding the configuration process through interactive questionnaires. The same remark applies to other SCM environments, a notable exception being the CoSMIC configurable middleware [18]. A key component of CoSMIC, namely the Options Configuration Modeling Language (OCML), allows developers to capture *options* that affect the way middleware services are configured. Options are similar to variability attributes in PCL, but OCML goes beyond PCL by allowing constraints to be defined over individual options or groups thereof. Like in Adele, constraints in OCML are expressed in a first-order logic language. OCML expressions are fed to an interpreter that prompts users to enter values for each option and raises error messages when the entered values violate a constraint. But unlike our proposal, the OCML interpreter does not preemptively flag incompatible values to remaining options based on values previously given to other options, nor is it able to determine skippable questions.

Another research stream, namely *Feature-Oriented Domain Analysis* (FODA), has led to techniques for modeling software product families in terms of the features they support. A number of feature modeling languages have been proposed [10, 7, 5]. These languages view feature models as tree-like structures with high-level features being decomposed into sub-features. Support for capturing multiplicity constraints (how many sub-features of a given type can a feature have?) is typical of these languages. In contrast, support for capturing arbitrary constraints on combinations of features is limited [5]. Our proposal subsumes feature modeling languages as far as capturing constraints goes, as it supports arbitrary propositional logic constraints over simple facts. In addition, it exploits these constraints to guide the configuration process. On the other hand, some may argue that features are more appropriate for modeling configuration alternatives than questions and facts.

Another approach to software product lines closely related to FODA is presented in [9]. In this paper, the authors introduce the concept of *feature variability patterns*: collections of roles and associations that need to be bound to specific artifacts (e.g. component implementations) in order to produce a configured system. Constraints can be defined over feature variation patterns using a scripting language. A configuration tool guides the developer through a number of tasks: each one corresponding to the binding of a role to an artifact. As in the previously reviewed approaches, constraints are only evaluated after a task is completed, and if the constraint is violated, the developer is left with the burden of repairing it. In contrast, our approach preemptively avoids constraint violations. Also, the approach in [9] does not support the definition of dependencies between configuration tasks (i.e. questions in our terminology). This support is important when certain questions can lead to other questions becoming constrained or irrelevant: in which case the most discriminatory questions can be given higher precedence (e.g. questions 2 and 3 in our working example).

Our proposal has commonalities with the CML2 language which was designed to capture configuration processes for the Linux kernel [13]. Like Quaestio, CML2 supports the definition of validity constraints based on propositional formulae over so-called *symbols* (which may be tri-valued in CML2). A configuration model in CML2 is composed of questions which lead to a given symbol being given a value. Questions can be grouped into menus which are arranged in a hierarchical mode. In CML2, questions within a menu are arranged sequentially while menus are visited from top to bottom. This is in contrast with our approach where questions (and facts) can be arranged in any partial order. Also, questions in CML2 only lead to one fact being set, while our questions can be used to set multiple inter-related facts at once.

Our work is also related to questionnaire systems. A range of commercial products, such as Vanguard Software's Vista,¹¹ support the definition of online questionnaires and the collection and analysis of responses. Such systems rely on the notion of *question flows* as defined in [12], wherein questions are related by precedence dependencies, while branching operators are used to capture condi-

¹¹ <http://www.vanguardsw.com/vista/online-questionnaires.htm>

tional questions. This paradigm is procedural: the developer of the questionnaire needs to determine the points in time at which branching occurs. Additionally, dependencies are expressed at the granularity of questions. This makes it difficult to capture scenarios where the possible answers to a question depend on answers to previous questions.

Our work can be related to form definition languages, e.g. XForms [2], InfoPath,¹² or FB-WIS [4]. These languages support the definition of fields, constraints over fields (e.g. range of values, multiplicity constraints), and dependencies between fields such as “a field is mandatory if another field has been given a certain value” or a “field is only visible if another field has been given a certain value”. Constraints are defined in a first-order logic language, making their analysis computationally impractical. This contrasts with our approach based on propositional logic, for which we can apply relatively efficient analysis techniques to determine if and when should a question be answered, or what are the valid answers to that question given the answers to previous questions.

In separate work [15], we have explored the issue of linking a set of facts identified in a configuration model to a specific notation for describing reference process models, namely Configurable Event-driven Process Chains (C-EPCs) [16]. The central idea is that each variability point and its alternatives captured in a C-EPC can be associated with boolean functions over the set of facts captured in a configuration model. Thus, a facts setting obtained from a configuration process such as the one outlined in this section, can be used to configure the variation points of a C-EPC, yielding a lawful EPC.

7 Conclusion

The framework presented in this paper, as well as the technique for generating interactive questionnaires from configuration models, is a first step towards a broader framework for model-driven configuration of software systems in general, and enterprise systems in particular. One necessary extension to the framework, is to integrate the concept of actions (e.g. model or code transformations) resulting from choices made during configuration. In realistic scenarios, some actions may be incompatible with others. For example, an action that modifies a fragment of a model is incompatible with another action which removes this fragment altogether. Integrating such dependencies between actions into the proposed framework, and designing techniques for propagating these dependencies between actions into constraints over facts, is a direction for future work. We plan to investigate this and other extensions to the framework in the specific context of configuration of business process models. An initial approach aimed at mapping (configurable) business process models to facts settings obtained from a configuration model is reported in [15].

¹² <http://www.microsoft.com/office/infopath>

References

1. W. M. P. van der Aalst and A. H. M. ter Hofstede. YAWL: Yet Another Workflow Language. *Information Systems*, 30(4):245–275, 2005.
2. J. Boyer, D. Landwehr, R. Merrick, T. Raman, M. Dubinko, and L. Klotz. XForms 1.0 second edition, W3C Recommendation, 2006. <http://www.w3.org/MarkUp/Forms>.
3. R. E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Trans. Computers*, 35(8):677–691, 1986.
4. T. Calders, S. Dekeyser, J. Hidders, and J. Paredaens. Analyzing Workflows Implied by Instance-dependent Access Rules. In *Proceedings of the 25th ACM SIGMOD (PODS)*, pages 100–109, Chicago IL, USA, 2006. ACM.
5. V. Cechticky, A. Pasetti, O. Rohlik, and W. Schaufelberger. XML-Based Feature Modelling. In *International Conference on Software Reuse (ICSR)*, pages 101–114, Madrid, Spain, 2004.
6. T. Curran and G. Keller. *SAP R/3 Business Blueprint: Understanding the Business Process Reference Model*. Upper Saddle River, 1997.
7. K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.
8. J. Estublier and R. Casallas. The Adele Software Configuration Manager. In *Configuration Management*, pages 99–139. John Wiley & Sons, 1994.
9. I. Hammouda, J. Hautamäki, M. Pussinen, and K. Koskimies. Managing Variability Using Heterogeneous Feature Variation Patterns. In *FASE*, pages 145–159, 2005.
10. K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh PA, USA, 1990.
11. S. Minato, N. Ishiura, and S. Yajima. Shared Binary Decision Diagram with Attributed Edges for Efficient Boolean function Manipulation. In *DAC*, pages 52–57, 1990.
12. K. Morton, C. Carey-Smith, and K. Carey-Smith. The QUEST Questionnaire System. In *Proceedings of the 2nd ANNES*, pages 214–217. IEEE Computer Society, 1995.
13. E. S. Raymond. The CML2 Language. <http://catb.org/esr/cml2/cml2-paper.html>, 2000.
14. J. Recker, J. Mendling, W.M.P. van der Aalst, and M. Rosemann. Model-Driven Enterprise Systems Configuration. In *Proceedings of the 18th International Conference on Advanced Information Systems Engineering (CAiSE'06)*, pages 369–383, Luxembourg, 2006. Springer.
15. M. La Rosa, J. Lux, S. Seidel, M. Dumas, and A.H.M. ter Hofstede. Questionnaire-driven Configuration of Reference Process Models. 2006. QUT ePrints, <http://eprints.qut.edu.au/archive/00005786>.
16. M. Rosemann and W. M. P van der Aalst. A Configurable Reference Modelling Language. *Information Systems*, 32(1):1–23, 2007.
17. E. Tryggeseth, B. Gulla, and R. Conradi. Modelling Systems with Variability using the PROTEUS Configuration Language. In *Software Configuration Management, ICSE SCM-4 and SCM-5 Workshops*, pages 216–240. Springer, 1995.
18. E. Turkey, A.S. Gokhale, and B. Natarajan. Addressing the Middleware Configuration Challenges using Model-based Techniques. In *Proceedings of the 42nd ACM Southeast Regional Conference*, pages 166–170, Huntsville AL, USA, 2004. ACM.