Variability Modeling for Questionnaire-based System Configuration

Marcello La Rosa[1], Wil M.P. van der Aalst[2,1], Marlon Dumas[1], Arthur H.M. ter Hofstede[1]

[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@tm.tue.nl

Abstract. Variability management, 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 configured and composed to meet the requirements 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. This paper presents a formal foundation for representing system variability for the purpose of generating questionnaires that guide users during system configuration. 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: variability modeling, system configuration, questionnaire, software product family, reference model

1 Introduction

Variability is inherent to many information and software systems. Explicitly modeling the variability of such systems in order to enable their configuration, is a well-known approach to achieve reuse [21]. For example, enterprise systems packages such as SAP provide modules and business objects covering a range of common functions such as invoicing, financial reporting and controlling [9]. Analysts and developers configure and compose these modules to meet the requirements of individual customers. To guide this individualization, SAP provides a comprehensive collection of *reference models* encompassing more than 4000 entity types and 1000 business process models and inter-organizational business scenarios [27]. These reference models are configured to meet specific needs, and the resulting configured models in turn, drive the individualization of the system [26]. Similarly, software product families are an approach to package related functionality into generic software assets, from which system variants are generated [11]. Configuration is an integral part of the lifecycle of these systems.

Configuration may involve setting a collection of parameters that capture the system's variability, 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 system 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 framework for modeling variability for the purpose of system configuration. Variability is captured by means of *configuration models* composed of *questions*. The space of possible answers to a question is represented as a set of *facts*, each of which can be set to true or false. These facts encode the variability of the system, e.g. optional features, values of configuration parameters, etc. The individualization of the model or system is captured as *configuration actions*. As the questionnaire is answered, values are assigned to facts, and the resulting facts valuation determines which configuration actions should be performed to derive the individualized model or system. Since some combinations of answers may lead to invalid configurations, the framework supports the definition of propositional logic constraints over facts. Questions and facts can be connected through precedence/order dependencies in arbitrary ways so long as these dependencies satisfy some syntactic criteria. These criteria prevent contradictory dependencies that lead to deadlocks during configuration.

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, and also, in a way that prevents the violation of the propositional constraints defined over facts. The only major assumption is that questions have a finite or discretized domain of possible answers, which essentially means that the space of possible system variants is finite. This assumption allows configuration models to be efficiently analyzed so as to prevent the user from entering conflicting responses to successive questions.

The remainder of the paper is organized as follows. Section 2 outlines the approach by means of a working example. 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 gen-

eration 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 Variability Modeling Approach

We propose to depict variability 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 only the relevant questions in an order consistent with the precedences between questions and facts.

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 into 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?". Questions are thus interfaces that present facts to users in a structured manner. Although the same 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. An implementation of the questionnaire should keep track of the facts previously set, and support the ability of changing the value of a fact already set by rolling back the question that contains it.

A *facts valuation* 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, we consider an order fulfillment collaborative process model in the area of supply chain management featuring a number of variability points. This process, based 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

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

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" (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 incidental 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 working 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 can implement. These facts are grouped in question q_1 that asks 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 Ⓜ in the picture), so that they have to be explicitly set to *true* or *false* when answering q_2. Other questions would allow users to choose the roles responsible for Pickup and Delivery (q_6, q_7), the Claims to be handled (q_4) and the Manager for Loss or Damage Claims (q_5). Default values have been assigned to the facts of Fig. 1 (where with a Ⓣ we indicate a fact whose default=*true*, while no symbol is used to mean a fact whose default=*false*). They have been set in order to capture a VICS process that implements all the business functions ($f_1, f_2, f_3, f_4 = true$) and all the Logistics's sub-phases ($f_8, f_9, f_{10}, f_{11} = true$), and that supports TL shipments ($f_5 = true$, $f_6, f_7 = false$). In this type of shipment, the Supplier is usually in charge of fixing the Pickup appointment (so $f_{16} = true$ and $f_{17} = false$) while the Buyer is responsible for Delivery ($f_{18} = true$, $f_{19} = false$). The process handles only Loss or Damage Claims (thus $f_{12} = false$ and $f_{13} = true$), managed by the Supplier which acts as intermediary between the Buyer and the Carrier ($f_{14} = true$, $f_{15} = false$).

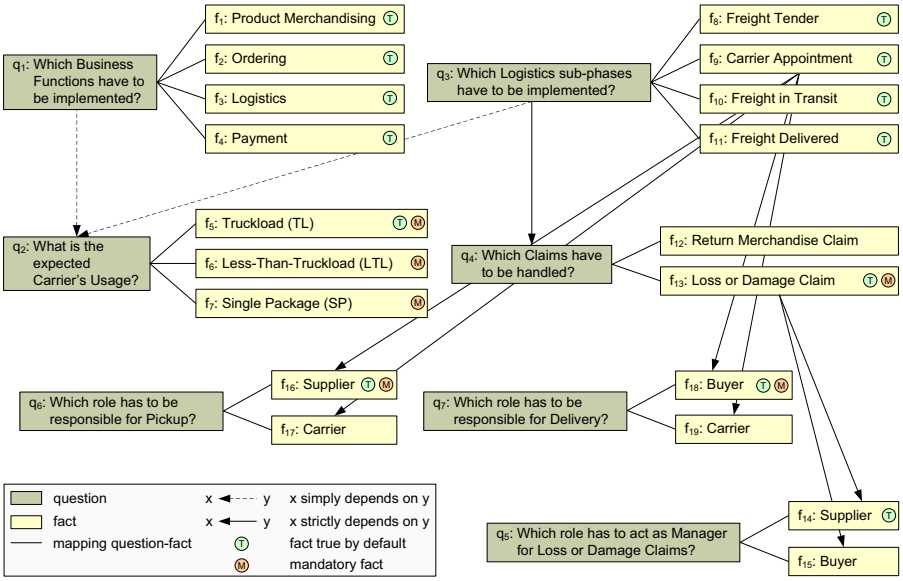

Fig. 1. A possible structure of questions-facts drawn from the VICS EDI Framework.

2.1 Order Dependencies and Constraints

Order dependencies ("dependencies" for short) can be introduced to establish 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. Therefore, fact x can be set only if at least all the facts in 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 in which questions are posed to users, as questions inherit the dependencies defined on 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. Fig. 2 shows the final structure that defines the partial order in which the questions of Fig. 1 will be be posed to users. From the diagram we can see that q_5, q_6 and q_7 have inherited their facts' dependencies.

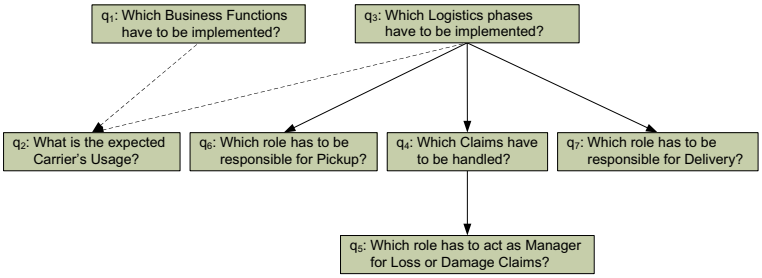

Fig. 2. The partial order over the questions of Fig 1.

Dependencies provide a means for ordering questions but do not affect facts values. For example, with a dependency we cannot capture the restriction on the Carrier's Usage, which implies that only one type of shipments is to be supported in a configured system. This latter restriction corresponds to asserting that exactly one fact among f_5, f_6 and f_7 in q_2. Moreover, answering a question may restrict the allowed answers to subsequent questions, and not all combinations of answers may lead to valid facts valuations. Indeed, if SP (f_7) is asserted in q_2, then no appointment negotiation is allowed for Pickup and Delivery, i.e. f_{16}, f_{17} have to be negated in q_6 and f_{18}, f_{19} have to be negated in q_7.

We capture these restrictions as a set of propositional logic *constraints* over facts. The following constraints capture the requirements of the VICS EDI Framework and refer to the facts of Fig. 1:[4]

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

C1 ensures that at least one business function is chosen in q_1. 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 in q_3, 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 that is affected by the Carrier's Usage is selected. Likewise, as per C7 and C9, exactly

[4] $\veebar$ indicates the exclusive disjunction (XOR).

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 Pickup and Delivery appointments are handled during the Carrier Appointment phase of the VICS process and only in case of TL or LTL shipments.

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 as exactly one Manager is to be chosen for Loss or Damage Claim, provided Loss or Damage Claim has been set to *true* in q_4.

Dependencies and constraints are not overlapping concepts. Rather, they complement each other. An example is shown by C4: $(f_{12} \vee f_{13}) \Rightarrow f_{11}$ and the strict dependency that q_4 has on q_3. Here the behavior we want to capture is that Claims can be handled only if Freight Delivered 'has been' selected, viz., f_{12} and f_{13} can be set to *true* only if f_{11} has been asserted before. Similarly, due to C6: $f_{13} \Leftrightarrow (f_{14} \veebar f_{15})$ and q_5 that indirectly depends on q_4, exactly one Manager for Loss of Damage Claim is to be selected in q_5, but only after Loss or Damage Claim (f_{13}) 'has been' asserted in q_4. In both examples, constraints alone do not provide enough information to get the desired behavior.

In some cases, instead, the sole usage of constraints is required to achieve the desired semantics, as shown by C2: $f_3 \Leftrightarrow (f_8 \vee f_9 \vee f_{10} \vee f_{11})$. This constraint states that at least one Logistics sub-phase is to be chosen in q_3 if and only if this business function 'is' selected in q_1. Since these two questions are not bound by any order dependency, one can answer q_1 or q_3 for first. However, the answer given to one of the two questions will affect the facts values of the other question.

As dependencies are not bound to constraints, they can rely on the context so as to facilitate the configuration process. For example, they can vary based on the usage or on the organization role that is meant to configure the system. To cater for this, multiple sets of dependencies can be associated to the same structure questions-facts. This would not be possible if we automatically derived dependencies from constraints, e.g. by means of a symbolic analysis of boolean expressions. Besides, as shown before, there are situations where dependencies are not needed in order to capture the desired behavior.

A facts valuation is a *configuration* if and only if it complies with the constraints over the facts values. A configuration is thus the result of answering an interactive questionnaire, where questions are posed to users according to the order dependencies, and constraints are dynamically checked so as to prevent users from entering conflicting responses. Although a configuration solely relies on facts values, questions and dependencies are used 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 'unordered' facts.

2.2 Actions

Facts can be associated to sets of actions, i.e. modifications to be performed on the domain model to reflect the effects of a configuration. For example, in the field of software product families, such an action could correspond to removing some code fragment from a software asset, as a result of answering a set of questions. In business process model configuration, an action could be associated to adding/removing a process fragment, whenever the corresponding fact is set to *true*, resp. *false*.

Fig. 3 shows an overview of the order fulfillment process model.[5] For readability purposes, the model has been divided into a set of configurable process fragments, where fragments are delimited by dotted boxes and identified by the facts of Fig. 1.

The four main process fragments refer to the Business Functions – Product Merchandise, Ordering, Logistics and Payment – that the process can implement. As such, their boxes encompass all the other configurable fragments. For example, Logistics (box "f_3") contains the fragments for its sub-phases, i.e. Freight Tender ("f_8"), Carrier Appointment ("f_9"), Freight in Transit ("f_{10}") and Freight Delivered ("f_{11}"). If we associated an action to each of these facts, that corresponds to the removal of the affected process fragments, then setting f_3 to *false* would imply to remove Logistics as well as all the fragments therein. This complies with C2, which has been built right to reflect this relation 'parent-child' that Logistics holds with its sub-phases.

Carrier Appointment, in turn, includes a fragment for handling each type of shipment ("f_5","f_6","f_7") and each role that can be responsible for Pickup (boxes "f_{16}" and "f_{17}") and for Delivery (boxes "f_{18}" and "f_{19}"). The last four fragments occur only within the boxes for "f_5" and "f_6", as only for TL or LTL shipments the Pickup and Delivery details can be decided. Since all the above facts are mapped to fragments within Logistics, if at least one of them is chosen in the configuration process, then Logistics cannot be removed anymore (i.e. f_3 must be set to *true*). At the level of facts, these interactions are described by constraints C3, C5, C7 and C9.

Similar considerations hold for the remaining process fragments and constraints. Constraints can be defined over actions as well (e.g. two actions that are in *XOR* relation), provided in the end they are traced back to the level of facts.

In the next section we will rigorously define the above concepts. The formal definition is used to describe with accuracy the variations that can be identified in a configurable domain (e.g. the VICS EDI Framework represented by its process model). We will then show how to generate an interactive questionnaire from this definition, that can be used to configure such variations.

[5] A full representation of this process using the YAWL notation [1] can be found at `http://www.fit.qut.edu.au/~dumas/ConfigurationTool.zip`

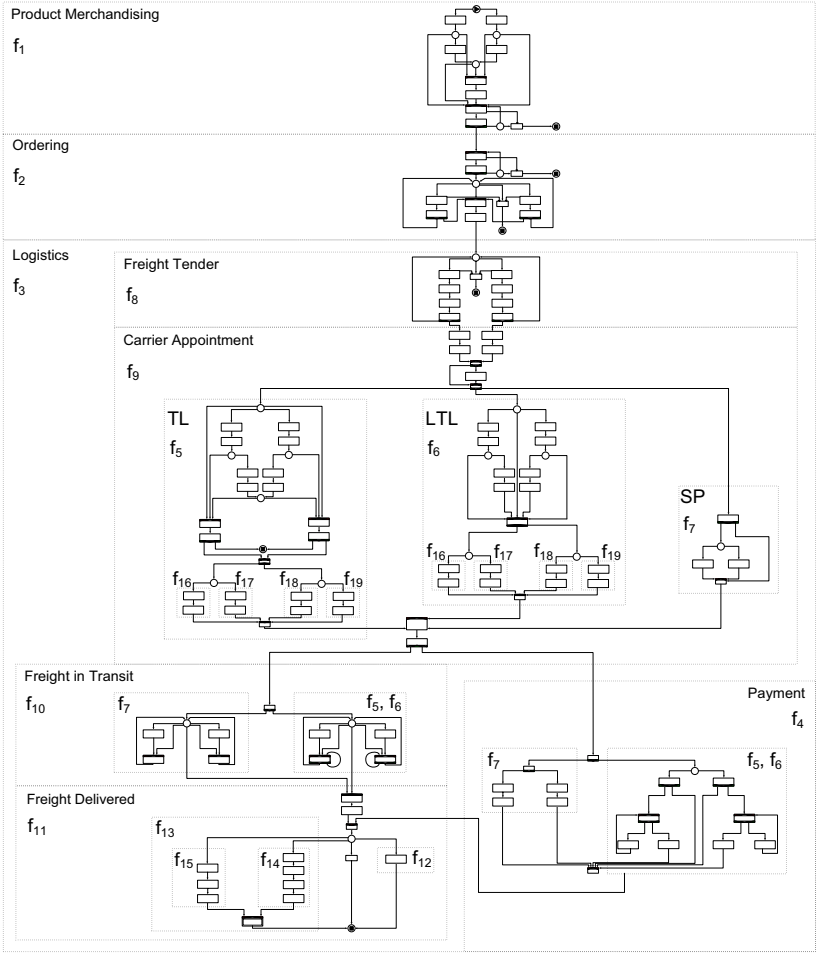

Fig. 3. The order fulfillment collaborative process model, with the facts of Fig. 1.

3 Formal Definition of Configuration Models

We use the concept of *configuration model* (CM) to directly capture variations in terms of facts, questions and their relations. A CM does not incorporate elements of commonalities, such as those aspects of a configurable domain that do not vary. The only difference with the informal description of the approach is that in a CM constraints over facts values are described by means of a true table of their conjunction.

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

- F *is a finite, non-empty set of* facts,
- $F_D \subseteq F$ *is the* default valuation, *i.e. the set of facts whose default 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,
- $map_{QF} \in Q \rightarrow \mathcal{P}(F) \setminus \{\varnothing\}$ *is a function mapping a question onto a set of facts, such that* $\bigcup_{q \in Q} map_{QF}(q) = F$,[6]
- $map_{FA} \in F \rightarrow \mathcal{P}(Act)$ *is a function mapping a fact onto a set of actions, such that* $\bigcup_{f \in F} map_{FA}(f) = Act$,
- $pre_F \in F \rightarrow \mathcal{P}(\mathcal{P}(F)) \setminus \{\varnothing\}$ *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. $\nexists_{G \in \mathcal{P}(F) \setminus \{\varnothing\}}\ \forall_{f \in G}\ \forall_{F' \in pre_F(f)}\ F' \cap G \neq \varnothing$, *i.e. no undesired circular dependencies,*
- $pre_Q \in Q \rightarrow \mathcal{P}(\mathcal{P}(Q)) \setminus \{\varnothing\}$ *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. $\nexists_{G \in \mathcal{P}(Q) \setminus \{\varnothing\}}\ \forall_{q \in G}\ \forall_{Q' \in pre_Q(q)}\ Q' \cap G \neq \varnothing$, *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,*
- $CS \subseteq \mathcal{P}(F)$ *is the set of the* allowed valuations *of the facts in* F, *such that* $F_D \in CS$, *i.e. the default valuation is always allowed.*

Elements of CS are those facts valuations 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 CS, it follows that the fact is negated in that valuation. For example, if $F = \{f_1, f_2, f_3, f_4\}$ and $\{f_1, f_2, f_4\} \in CS$ is a facts valuation, then in the latter all the facts but f_3 are set to *true*.

As the default valuation must always be allowed, set CS is non-empty. If no constraints are defined, $CS = \mathcal{P}(F)$. A situation where $CS = \{F\}$, means that all the facts must be asserted (upper-bound case), while $CS = \{\varnothing\}$ corresponds to negating all the facts (lower-bound case).

We say a fact is *meaningful* if it can be freely set before starting the configuration process, i.e. if $\exists_{F'_1, F'_2 \in CS}\ (f \in F'_1\ \wedge\ f \notin F'_2)$. Such a fact represents a variation in the model. Thus, if a fact is not meaningful it should not be included in the model, as it would represent a commonality.

Actions depend on the type of the domain model and the language used for its description. For example, if they refer to the configuration of software code trunks/features, then the programming language needs also to be taken into account. Likewise, if actions refer to process/data models, the modeling notation used for the representation of such models needs also to be considered.

[6] $\mathcal{P}$ indicates the power set.

This is important as actions (and their relations) must not violate the syntactic rules of the description language. Since here we aim at providing a language-independent formalization of variability, a detail description of actions is left out. In separate work [17] we have explored the use of actions for business process models configuration (further details can be found in Section 6).

The set of preconditions for facts and questions are used to specify the order dependencies as follows.

Definition 2 (Order Dependencies). *Let* $CM = (F, F_D, F_M, Q, Act, map_{QF}, map_{FA}, pre_F, pre_Q, CS)$ *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* $\forall_{Q' \in pre_Q(q)}\ q' \in Q'$.

For a fact or question, its set of preconditions represents the disjunction of preconditions being conjunctions of the dependencies. In other words, a fact can be set (to *true* or *false*), or a question can be answered, only if at least all the facts in one of its preconditions have already been set (to *true* or *false*), or all the questions in one of its preconditions have already been answered. Thus facts (questions) in the same precondition are in *AND* relation, while preconditions are in *OR* relation.

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 $pre_F(f_1)$. This is a strict dependency. On the other hand, f_1 can depend either on f_3 or f_4, as these facts do not belong to each clause of $pre_F(f_1)$. These are simple dependencies. A strict dependency always implies a simple one.

As per 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 or question q, then $pre_F(f)$ or $pre_Q(q)$ should contain only 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 it cannot contain other sets, since all the sets would include the empty one.

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. Since all the preconditions are in *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'. These occur whenever, for each fact (or question) of a given set, all its preconditions contain at least one 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. 4 - 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. 4 - b) are denied since all the preconditions share the same set of facts. By applying the second requirement to both the above cases, 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 in G. Such undesirable circles can be caused by simple and strict dependencies, as in the last case.

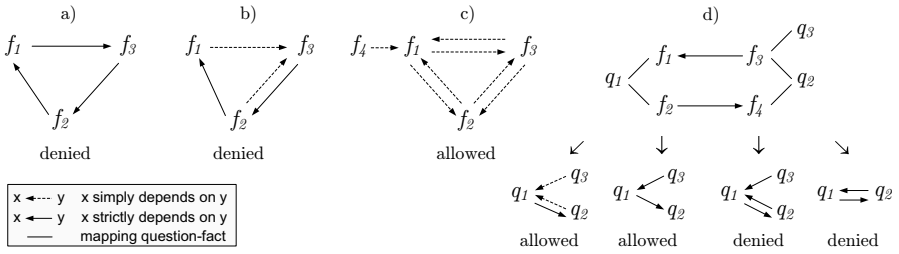

Fig. 4. Examples of circular dependencies over facts and questions.

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 $pre_F(f_1) = \{\{f_2\}, \{f_3\}, \{f_4\}\}$, $pre_F(f_2) = \{\{f_1\}, \{f_3\}\}$, $pre_F(f_3) = \{\{f_1\}, \{f_2\}\}$, $pre_F(f_4) = \{\varnothing\}$ (Fig. 4 - 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 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 in all the preconditions F'. These dependencies that 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 $pre_F(f_4) = \{\{f_2\}\}$ (Fig. 4 - d). Here f_3 is a shared fact between q_2 and q_3. If we lift facts dependencies to the level of questions, we see that q_3 does not inherit any dependencies as it is mapped to f_3 only, q_2 strictly depends on q_1 by means of f_4, 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 for 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 completes the formal description of the approach presented so far by defining the "runtime behavior", i.e. the configuration process for a *CM*. In a configuration process questions are dynamically posed to users according to the order dependencies, and answers can be given only if they comply with the constraints.

We first define some concepts to work with facts valuations, such as *set of facts valuations*, *answer*, *state* and *state space*. These concepts are needed to specify when a question can be posed to users. In particular, an answer is any facts valuation where only a subset of facts (the ones that relate to a question) are set, while a state of *CM* is identified by a facts valuation and a set of answered questions.

Definition 3 (Set of fact valuations, Answer, State, State space). *Let* $CM = (F, F_D, F_M, Q, Act, map_{QF}, map_{FA}, pre_F, pre_Q, CS)$ *be a configuration model:*

- $V = F \to \{true, false, unset\}$ *is the* set of facts valuations*,*
- $a \in V$ *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* $CM = (F, F_D, F_M, Q, Act, map_{QF}, map_{FA}, pre_F, pre_Q, CS)$ *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,*

- $f(s) = f(vs) = \{f \in F \mid vs(f) = \mathit{false}\}$ *is the set of facts that are* false *in state* s,
- $u(s) = u(vs) = \{f \in F \mid vs(f) = \mathit{unset}\} = F \setminus (t(s) \cup f(s))$ *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* $a \in V$:
- $t(a) = \{f \in F \mid a(f) = \mathit{true}\}$, *is the set of facts set to* true *by answer* a,
- $f(a) = \{f \in F \mid a(f) = \mathit{false}\}$, *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,
- $\mathit{compl}(s) = \mathit{compl}(vs) = \{f \in F \mid vs(f) = \mathit{true} \ \vee \ (f \in F_D \wedge vs(f) \neq \mathit{false})\}$ *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* $f \in F$:

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

For each state a set of valid questions is presented to users. For a question to be valid in a state ($\mathit{valid}(q, s)$), two conditions must hold: i) the question has not been answered yet, and ii) at least one of its preconditions is satisfied.

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

Definition 5 (Valid answer). *Let* $CM = (F, F_D, Q, Act, map_{QF}, map_{FA}, pre_F, pre_Q, F_M, CS)$ *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:*

- $\mathit{valid}(q, s) = q \notin qs \ \wedge \ \exists_{Q' \in pre_Q(q)} \ Q' \subseteq qs$, *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,*
- $\mathit{outcome}(a, q, s) = (vs \oplus a, qs \cup \{q\})$, *i.e. the state resulting after answering* a *to question* q *in state* s,
- $\mathit{valid}(s) = \exists_{F' \in CS} \ (t(s) \subseteq F' \ \wedge \ f(s) \cap F' = \varnothing)$, *i.e. the facts valuation of the state has to comply with the constraints on facts,*
- $\mathit{valid}(a, q, s) = \mathit{valid}(q, s) \ \wedge \ t(a) \cup f(a) = map_{QF}(q) \ \wedge \ \forall_{f \in map_{QF}(q) \setminus u(s)} \ a(f) = vs(f) \wedge \mathit{valid}(\mathit{outcome}(a, q, s))$, *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 CS, so as to verify if it complies with the constraints defined on facts values. In this way

[7] This function sets to *true* those facts left *unset* whose default was *true*.

we ensure it is always possible to complete the current facts valuation by setting any remaining fact still unset.

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

Definition 6 (Labeled Transition System of CM). *Let* $CM = (F, F_D, F_M, Q, Act, map_{QF}, map_{FA}, pre_F, pre_Q, CS)$ *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* $target(t) = s'$*,*
- $s_{init} = (\{(f, unset) \mid f \in F\}, \varnothing) \in S_v$ *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 defaults. As shown in the definition of final state of the labeled transition system, this is possible only if the facts valuation that results after applying the defaults complies with the constraints on facts values, i.e. if it does not violate the configuration process so far.

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 $f_1 \Rightarrow ((f_2 \wedge f_4) \veebar (f_3 \wedge f_5))$. It follows that $CS = \{\{f_1, f_2, f_4\}, \{f_1, f_3, f_5\}, ...\}$, where the remaining elements of *CS* 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 valuation cannot be applied for the remaining *unset* facts in q_2, since only either f_2 and f_4 or f_3 and f_5 can assume value *true*. Hence we cannot find an $F' \in CS$ such that $\{f_1, f_2, f_3\} \subseteq F'$. On the other hand, if we set f_1 to *false* we get straightaway to a final state, where all the mandatory facts have been set and the remaining ones can take their default.

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

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* $CM = (F, F_D, F_M, Q, Act, map_{QF}, map_{FA}, pre_F, pre_Q, CS)$ *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:*

- $\sigma = (t_1, ..., t_n) \in T^+$ *is a* trace *of LTS iff* $target(t_i) = source(t_{i+1})$ *for each* $1 \leqslant i \leqslant n-1$*, 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. the facts valuation reached with the last state of a configuration trace, completed with default values. Therefore, a configuration always complies with the constraints.

Definition 8 (Configuration of *CM*, Configuration Space of *CM*). *Let* $CM = (F, F_D, F_M, Q, Act, map_{QF}, map_{FA}, pre_F, pre_Q, CS)$ *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.*

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 following theorem is used to prove that the definition of pre_Q and *CS* are sufficient to avoid any deadlock during the configuration process. This is because undesirable circular dependencies are excluded a priori in pre_Q, and only those facts valuations that comply with the constraints are represented in *CS*.

The theorem is followed by a corollary that shows the results.

Definition 9 (Trace Notation). *Let* $CM = (F, F_D, F_M, Q, Act, map_{QF}, map_{FA}, pre_F, pre_Q, CS)$ *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* $CM = (F, F_D, F_M, Q, Act, map_{QF}, map_{FA}, pre_F, pre_Q, CS)$ *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: i) we show that for all valid non-final states there always exists at least one valid question; ii) we show that for all valid questions in a valid state there always exists at least one valid answer.

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

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

Hence $valid(q,s)$.

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

- $[valid(q,s)]$. *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* $vs(f) = true$, *then* $f \in t_s(q)$,
    2) *if* $vs(f) = false$, *then* $f \in f_s(q)$,
    3) *if* $vs(f) = unset$,
       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) $a(f) = true$ *and* $f \in t_s(q)$ *or* $a(f) = false$ *and* $f \in f_s(q)$, *hence* (*definitions of* $t_s(q)$ *and* $f_s(q)$) $a(f) = true$ *and* $vs(f) = true$ *or* $a(f) = false$ *and* $vs(f) = false$, *hence* $a(f) = vs(f)$.
- $[valid(outcome(a,q,s))]$. *Let* $s' = outcome(a,q,s) = (vs \oplus a, qs \cup \{q\})$.
  - $[t(s') \subseteq F']$. $t(s') = \{f \in F \mid a(f) = true \vee (vs(f) = true \wedge a(f) = unset)\}$ (*definition of* $x \oplus y(f)$). *Let* $f \in t(s')$,
    1) *if* $a(f) = true$, *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* $vs(f) = true$ *and* $a(f) = unset$, *then* $f \in t(s)$ *and* $f \in F \setminus map_{QF}(q)$, *hence* $f \in F'$ *as* $t(s) \subseteq F'$.
  - $[f(s') \cap F' = \varnothing]$. $f(s') = \{f \in F \mid a(f) = false \vee (vs(f) = false \wedge a(f) = unset)\}$ (*definition of* $x \oplus y(f)$). *Let* $f \in f(s')$,
    1) *if* $a(f) = false$, *then* $f \in f_s(q) \cup f_u(q)$, *hence* $f \notin F' = \varnothing$ *given that* $f_s(q) \cap F' = \varnothing$ *and* $f_u(q) \cap F' = \varnothing$.

2) *if* $vs(f) = false$ *and* $a(f) = unset$, *then* $f \in f(s)$ *and* $f \in F \setminus map_{QF}(q)$, *hence* $f \notin F'$ *as* $f(s) \cap F' = \varnothing$.

Hence $valid(outcome(a, q, s))$.

Hence $valid(a, q, s)$.

Corollary 1 (Configuration processes always terminate). *For any configuration model* $CM = (F, F_D, F_M, Q, Act, map_{QF}, map_{FA}, pre_F, pre_Q, CS)$ *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 has begun, 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, such a question can take only one possible answer. We call these questions *skippable*, as they can be automatically answered and thus skipped by a supporting implementation (e.g. a questionnaire tool).

Definition 10 (Skippable Question). *Let* $CM = (F, F_D, F_M, Q, Act, map_{QF}, map_{FA}, pre_F, pre_Q, CS)$ *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:*

- $forceable(f, s) = f \in u(s) \wedge \forall_{F_1, F_2 \in CS} [(t(s) \subseteq F_1 \cap F_2 \wedge f(s) \cap (F_1 \cup F_2) = \varnothing) \Rightarrow F_1(f) = F_2(f)]$, *i.e. f assumes the same value in all the facts valuations still possible,*
- $skippable(q, s) = valid(q, s) \wedge \forall_{f \in map_{QF}(q)} [forceable(f, s) \vee f \notin u(s)]$, *i.e. a question can be skipped iff none of its facts is mandatory, and all its unset facts can have exactly one value or all its facts have been previously set.*

If a question is skippable the only possible answer is valid, since this valuation always complies with the constraints. Precisely, the forceability of a fact is determined by set CS, while if all the facts have been previously set, then the answer is already included in the last state s, which is valid by assumption.

5 Tool Support

In order to establish the practical feasibility of our approach, we have implemented a tool for the dynamic generation of interactive questionnaires. The features of this tool,[9] called *Quaestio*, are introduced in the first part of this section. The second part shows how the tool is used to configure the order fulfillment example of Section 2.

[9] Downloadable from `http://sky.fit.qut.edu.au/~dumas/ConfigurationTool.zip`

5.1 Prototype Implementation

Quaestio is a Java GUI which produces a set of ordered questions given a configuration model as input.

The interface 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's details: the list of facts for the question, the dependencies on other questions, and guidelines in natural language to configure the question. In a separate window, a Fact Inspector shows detailed information for each facts: its default value, if mandatory, the constraints that binds the fact in natural language (derived from facts' descriptions), the dependencies on other facts, the level of impact on the domain model, and specific guidelines to configure the fact.

The input format for a configuration model is described by an XML schema, which captures the structural requirements defined in Definition 1. In this way non-well-formed models are avoided a priori, e.g. those models where a fact is not associated to any question or where the questions do not cover all the facts.

Once a model is loaded, Quaestio shows the set of initial valid questions. Next, for each answer given, the tool dynamically calculates the next valid state so as to update the lists of valid and answered questions. The configuration process completes when 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.

A (partial) configuration can be exported to XML as a list of facts, keeping track of the values that have been set and whether they deviate from the defaults.

The implementation adheres to the formalization presented in Section 3 and 4. The only differences are in the internal representation of the constraints and of the state space.

Checking constraints based on CS – a representation of all the valid facts valuations – would be an NP-complete problem [14]. To overcome this issue, we opted to embody an existing calculator[10] based on Shared Binary Decision Diagrams (SBDDs) [7, 19]. SBDDs are a concise representation of boolean formulas for which there are efficient constraint-checking algorithms. They are based on the classical Binary Decision Diagrams(BDDs) [2] with the advantage of being always cheaper in size and time computation than classical BDDs. Regarding scalability, algorithms based on SBDDs can efficiently deal with systems made up of around one million of possibilities [19]. We use SBDDs to check the satisfiability of the constraints and the meaningfulness of the facts. In this way the tool can signal potential issues before starting a configuration. SBDDs are also used during the configuration process, to evaluate the type of relations among the facts of a question (e.g. an XOR question), and to verify the forceability of facts and the validity of answers.

As a result, we decided not to preemptively build the state space with an LTS, as the complexity of this operation would heavily depend on the size of

[10] Downloadable from `http://www-verimag.imag.fr/~raymond/tools/bddc-manual`

CS. Besides, the efficiency of the algorithm required to search for the next state would be affected by the size of the graph itself. Therefore, we opted for a dynamic generation of the state space, composed of the traversed states only. For each answer given, the next state is calculated by scanning only those questions that are neither answered nor valid. For each of these questions, the algorithm checks if at least one precondition can be satisfied (i.e. if all the questions in a precondition have been answered). If so, a question is put in the Valid Questions list only if it is not skippable, otherwise it goes straight to the Answered Questions list. These lists are kept in memory by means of hash sets.

The main features of the tool are:

- decision support: by means of guidelines, constraints and impact-level;
- dynamic checking of answers: answers can be given only if they comply with the constraints;
- default answer: default values can be given to all the facts of a question, if:
  - the value of facts set or forceable do not deviate from their default,
  - the resulting valuation is valid given the current state;
- fact value preservation: facts that occur in more than one question are set the first time and then preserve their value in subsequent questions they appear in;
- forceable facts: such facts are disabled and show their forced value;
- skippable questions: such questions are automatically answered;
- automatic completion: upon request the system can automatically complete the configuration process whenever all the mandatory facts have been answered and default values can be used for the remaining ones.
- question rollback: each answered question can be rolled back to the state before the answer.[11]

5.2 Sample Configuration Process

This section shows a sample configuration process for the order fulfillment process model of Fig. 3. Assume, for example, that we want to configure the model to handle SP shipments and to support only Loss or Damage Claims managed by the Supplier, and that we are not interested in the Payment phase of the process as it will be outsourced. These can be common choices among the stakeholders of a supply-chain management company interested in supporting the VICS EDI Framework.

Once the corresponding configuration model has been loaded into Quaestio, the valid questions are shown in the Valid Questions list. These are q_1 and q_3, since they have no dependencies (Fig. 5). The initial state is s_1 where no answers have been given, i.e. $qs(s_1) = \varnothing$. We decide, for example, to answer q_3 – *Which Logistics phases have to be implemented?* with its default answer. This corresponds to give answer $a_1(q_3) = \{(f_8, \mathsf{T}), (f_9, \mathsf{T}), (f_{10}, \mathsf{T}), (f_{11}, \mathsf{T})\}$, since all the facts of q_3 are *true* by default (shown by a green Ⓣ next to the fact description).

[11] In this case all the questions answered thereafter are also rolled back. Alternatively, a selective roll back can be easily implemented.

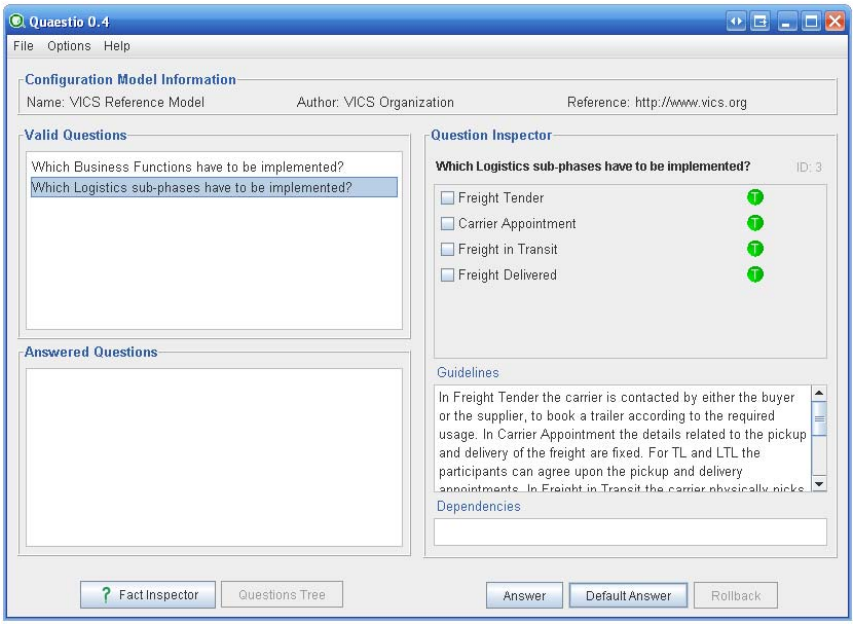

Fig. 5. State s_1: the only valid questions are q_1 and q_3.

With a_1 we reach state s_2 with $qs(s_2) = \{q_3\}$. q_2 is added to the valid questions due to its simple dependency on q_1 or q_3. Assume we choose q_1 from the Valid Questions. From the Question Inspector we can see that f_3 has been forced to *true* and grayed out (Fig. 6). The system has reacted to a_1 by setting f_3 in order to comply with C2. We answer q_1 with $a_2(q_1) = \{(f_1, \mathsf{T}), (f_2, \mathsf{T}), (f_3, \mathsf{T}), (f_4, \mathsf{F})\}$ so as to exclude Payment.

After a_2, we get to 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. Assume we pick q_2 – *What is the expected Carrier's Usage?*. Due to C3 and to the answers given so far, this question can only be answered if exactly one of its facts is set to *true* (the answer button is disabled). Also, this question needs to be explicitly answered as all its facts are mandatory (indicated by a red Ⓜ next to the fact description). We select Single Package and $a_3(q_2) = \{(f_5, \mathsf{F}), (f_6, \mathsf{F}), (f_7, \mathsf{T})\}$ is given.

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, since all their facts can take only value *false* due to C9. Thus $a_4(q_6) = \{(f_{16}, \mathsf{F}), (f_{17}, \mathsf{F})\}$ and $a_5(q_7) = \{(f_{18}, \mathsf{F}), (f_{19}, \mathsf{F})\}$ are automatically given by the system, which moves from s_4 to s_5 with a_5, and from s_5 to s_6 with a_6. q_6 and q_7 are added to the set of answered ones (shown in blue in Fig. 7) and $qs(s_6) = \{q_3, q_1, q_2, q_6, q_7\}$. Next we answer the only valid question remaining, q_4 – *Which claims have to be handled?*, with its default answer $a_6(q_4) = \{(f_{12}, \mathsf{F}), (f_{13}, \mathsf{T})\}$ as it complies with our requirements.

After a_6 we reach s_7 with $qs(s_7) = \{q_3, q_1, q_2, q_6, q_7, q_4\}$. q_5 – *Which role has to act as Manager for Loss or Damage Claims?* is now valid as it depends on

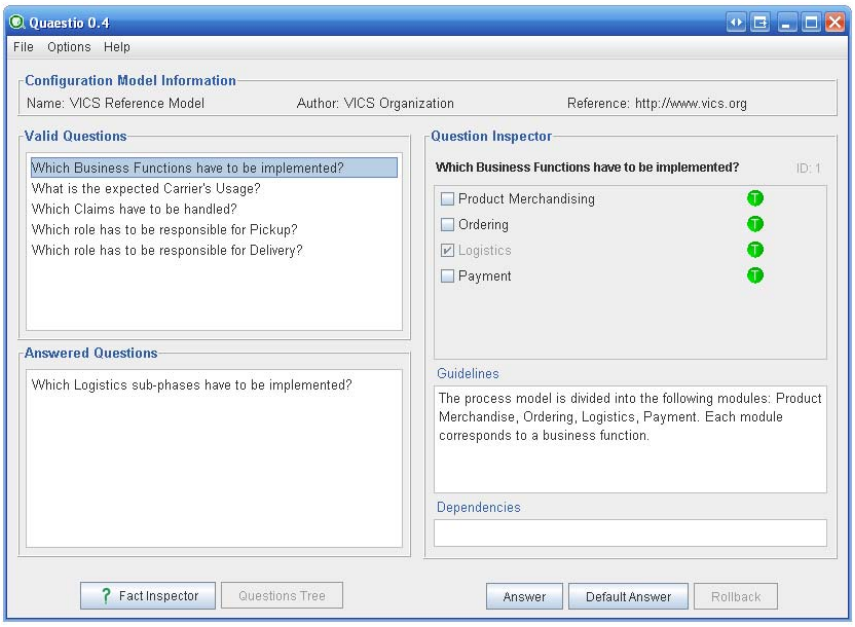

Fig. 6. State s_2: f_3 has been forced to *true* in q_1 in order not to violate C2.

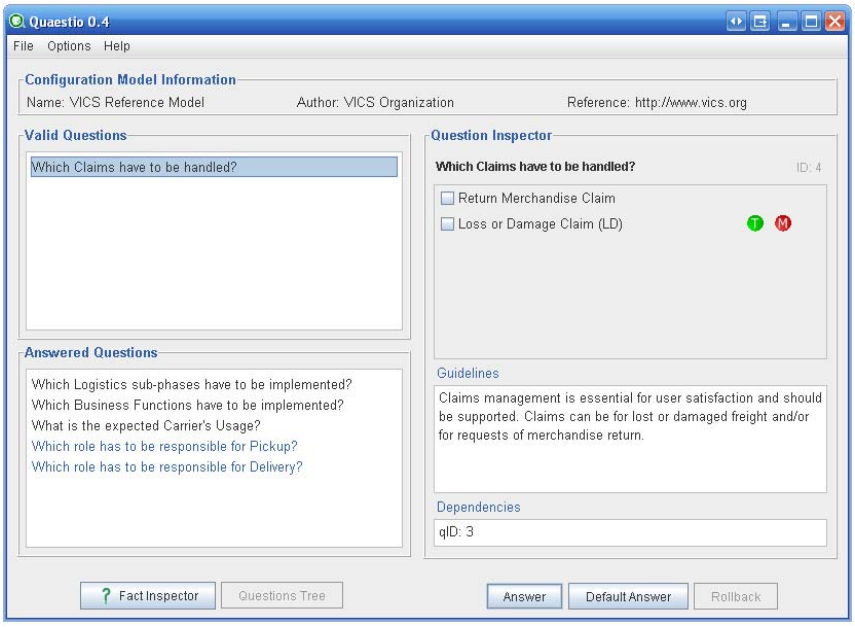

Fig. 7. State s_6: q_6 and q_7 have been skipped as their facts can only be negated.

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 use the automatic completion and answer $a_7(q_5) = \{(f_{14}, \mathsf{T}), (f_{15}, \mathsf{F})\}$ is given.

State s_8 is the next state with $qs(s_8) = \{q_3, q_1, q_2, q_6, q_7, q_4, q_5\}$. Assume that now we want to change q_4 in order to support only Return Merchandise Claims. In this case we can rollback q_4 and re-answer it. The system restores the current state to s_6, i.e. the state before answering q_4. We then answer $a_6(q_4) = \{(f_{12}, \mathsf{T}), (f_{13}, \mathsf{F})\}$ and reach s_7 again. This time, though, q_5 is skippable since a Manager can be chosen only for Loss or Damage Claims. The only valid answer is $a_7(q_5) = \{(f_{14}, \mathsf{F}), (f_{15}, \mathsf{F})\}$. With this we reach s_8 and complete.
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 $cf(\sigma) = \{(f_1, \mathsf{T}), (f_2, \mathsf{T}), (f_3, \mathsf{T}), (f_4, \mathsf{F}), (f_5, \mathsf{F}), (f_6, \mathsf{F}), (f_7, \mathsf{T}), (f_8, \mathsf{T}), (f_9, \mathsf{T}), (f_{10}, \mathsf{T}), (f_{11}, \mathsf{T}), (f_{12}, \mathsf{T}), (f_{13}, \mathsf{F}), (f_{14}, \mathsf{F}), (f_{15}, \mathsf{F}), (f_{16}, \mathsf{F}), (f_{17}, \mathsf{F}), (f_{18}, \mathsf{F}), (f_{19}, \mathsf{F})\}$.

The above configuration leads to the order fulfillment process model pictured in Fig. 8.

6 Related Work

Variability modeling has been widely studied in the field of Software Product Line Engineering (SPLE) [21, 8]. Among others, two research streams have emerged in SPLE, namely Software Configuration Management (SCM) [23] and Feature-Oriented Domain Analysis (FODA) [16].

Work on SCM has led to models and languages to capture how a set of available options impact upon the way a software system is built from a set of components. For example, the Adele Configuration Manager [13] supports the definition of constraints among artifacts composing a software family (e.g. "only one realization of an interface should be included in any instance of the family"). Such constraints are expressed as first-order logic expressions over attributes defined on *objects* that represent software artifacts. Building a configuration in Adele involves selecting a collection of objects that satisfy all constraints.

Similarly, in the Proteus Configuration Language (PCL) [28], software entities are annotated with *information attributes* and *variability control attributes*. The former provide stable information about an entity, i.e. commonalities, while the latter capture variability in the structure and in the process of building the entities. Variability attributes determine which actions are performed to build a variant of an entity. For example, one can capture that a sub-system maps to different sets of program files depending on the value of a variability attribute. However, only simple rules of the form "if-then-else" can be specified.

Another example is the Options Configuration Modeling Language (OCML) of the CoSMIC configurable middleware [29]. OCML allows developers to cap-

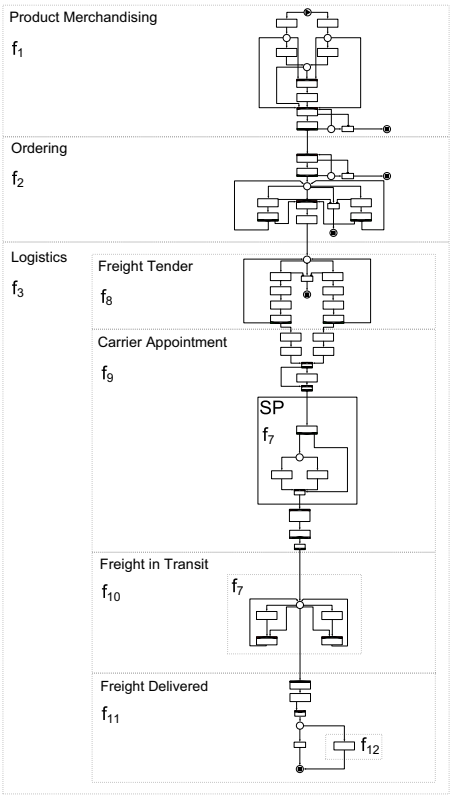

Fig. 8. The configured order fulfillment process model.

ture hierarchical *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. 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 skip options that are no longer relevant.

More generally, none of these approaches deals with guiding the configuration process through interactive questionnaires. Moreover, constraints are usually expressed in first-order logic, making their analysis computationally impractical (e.g. in Adele and OCML). This contrasts with our approach based on propositional logic, for which we can apply efficient analysis techniques to discard invalid answers to questions based on answers to previous questions.

FODA is a set of techniques for describing software product families in terms of their features. A number of feature modeling languages have been proposed [16, 11, 12]. In these proposals, feature models are represented as tree

structures called *feature diagrams*, with high-level features being decomposed into sub-features. A *feature* represents a system property that is relevant to a stakeholder and it is used to capture commonalities or to discriminate among systems in a family [11]. Constraints are expressed as expressions over features, specified by means of a proper grammar [18, 6] (e.g. a limit in the number of sub-features a feature can have).

Feature modeling languages have been embodied in a number of tools [5, 4]. These tools rely on SAT solvers to determine a valid configuration. But unlike our proposal, these tools do not guide the configuration process through interactive questionnaires. An exception is FeaturePlugin [4], which provides a wizard to traverse a feature model in a predetermined order only (depth-first).

Another approach related to FODA is presented in [15]. Here, the authors introduce the concept of *feature variability patterns* as collections of roles and associations that need to be bound to artifacts (e.g. component implementations) to produce a configured system. Constraints are defined over feature variation patterns using a scripting language. A configuration tool guides the developer through a number of tasks corresponding to the binding of a role to an artifact. Still, the tool does not support the definition of order dependencies between tasks. Moreover, 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 tool preemptively avoids constraint violations.

The major difference between our approach and the above research work is related to the representation of domain variability (e.g. in a software artifact or in a conceptual model). We propose to capture variability separately from commonality, while SCM and FODA combine these two in a single model. For example, in a feature diagram, there is no clear separation between a variation and a stable software asset, since both are represented by features. This lack of separation hinders the communication of variability [22].

In this respect, our approach is closer to the principles of the Orthogonal Variability Models (OVMs) [21, 22]. An OVM represents only the variable features, called *variation points*. *Variations* are then linked to a separate domain model, where both variability and commonalities are captured. In our approach we explicitly model this relation by means of *actions*, that are used to reflect the effects of a configuration on the domain model. Our facts can be compared to variations, and questions to variation points. Having said that, our proposal offers more flexibility than OVMs, as we can express non-hierarchical dependencies among features. Also, a question can refer to more than a single variation point and facts can appear in more than one question. The relations between our approach (CM), SCM, FODA and OVM are depicted in Fig. 9.

As shown in the picture, configuration models can also be seen as decision models [3], since context-based order dependencies are exploited to offer decision support through an interactive configuration process. Besides, our tool supports the evaluation and comparison of alternative answers to questions by means of guidelines, constraints and by providing information on the impact of facts on the domain model. In this respect, our proposal shares commonalities with

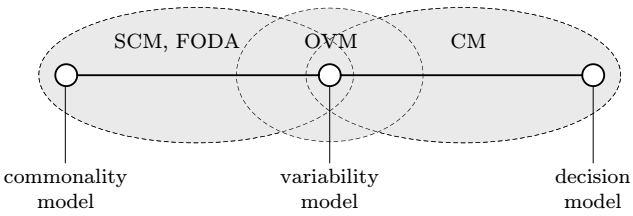

Fig. 9. Comparison among SCM, FODA, OVM and CM.

the CML2 language which was designed to capture configuration processes for the Linux kernel [25]. Like Quaestio, CML2 supports the definition of validity constraints based on propositional formulas over so-called *symbols* (which may be three-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 hierarchy. 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 symbol being set, while our questions can be used to set multiple inter-related facts at once.

Our tool is also related to questionnaire systems. A range of commercial products, such as Vanguard Software's Vista,[12] 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 [20], wherein questions are related by a fixed precedence order, while branching operators are used to capture conditional questions. This paradigm is procedural: the developer of the questionnaire needs to determine the points in time at which branching occurs. Additionally, constraints are expressed at the granularity of questions and only used to skip questions. This makes it difficult to capture scenarios where questions can be (partially) answered on the basis of previous answers (e.g. f_3 in q_1 that has been forced to *true* by answering q_3).

The idea of capturing variability in process models by annotating model fragments with boolean conditions and removing fragments whose conditions evaluate to false, has been explored in previous work [10, 24]. In [24] the authors extend UML Activity Diagrams (ADs) and BPMN diagrams with stereotypes to accommodate variability points. A variability point is linked to a feature and is evaluated with respect to a feature configuration (e.g. to activate/deactivate model elements). The approach, however, lacks a formalization, leaving room for ambiguities. In [10] UML ADs are annotated using *presence conditions* (PCs) and *meta-expressions* (MEs), that are then linked to elements of a feature diagram. PCs indicate if the model element they refer to should be present in the model. MEs are used to compute attributes of model elements (e.g. name,

[12] `http://www.vanguardsw.com/vista/online-questionnaires.htm`

return type). However, the approach only supports simple mapping of features to standard variability mechanisms provided by UML (e.g. decision nodes).

In separate work [17], we applied a similar approach to a notation for configurable process modeling, namely Configurable Event-driven Process Chains (C-EPCs) [27]. The idea is that each variability point and its alternatives (variations) captured in a C-EPC, can be associated with boolean expressions over the facts of a configuration model. Thus, a variation is selected whenever the corresponding boolean expression evaluates to true, triggering the execution of an action to configure the variation point. As a result of configuring all the variation points, the process model is transformed into a lawful EPC.

7 Conclusion

This paper has put forward a formal framework for representing system variability for the purpose of supporting configuration through interactive questionnaires. The framework has been embodied as a tool that guides the user through a set of questions, in an order consistent with the established order dependencies between questions and facts, and in such a way that violations of constraints over facts are preemptively avoided. Also, the tool is able to automatically skip questions whose answers are fully determined by previous ones, and it allows users to seamlessly rollback previous answers.

The proposed framework is independent of the notation(s) used to represent the system itself. It can be applied to support the configuration of data models, process models, or software artifact's in general, so long as appropriate types of configuration actions are defined for the corresponding notation. In separate work, we have shown how to apply the proposed framework to a process modeling notation, namely C-EPC.

In future work, we plan to empirically test the proposal in the field of process modeling for screen post-production. The goal is to show that domain experts with little to no knowledge of the notation in which the system is represented, are able to drive the configuration of a reference model for post-production. Also, we would like to measure the perceived usefulness and ease of use of the various features of the framework and of the tool.

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. S. B. Akers. Binary Decision Diagrams. *IEEE Trans. Computers*, 27(6):509–516, 1978.
3. S. L. Alter. *Decision support systems: current practice and continuing challenges*. Addison-Wesley, 1980.
4. M. Antkiewicz and K. Czarnecki. Featureplugin: Feature modeling plug-in for eclipse. In *OOPSLA'04, Eclipse technology eXchange (ETX) Workshop*, 2004.
5. D. S. Batory. Feature-Oriented Programming and the AHEAD Tool Suite. In *ICSE*, pages 702–703. IEEE Computer Society, 2004.

6. D. S. Batory. Feature Models, Grammars, and Propositional Formulas. In J. H. Obbink and K. Pohl, editors, *SPLC*, volume 3714 of *Lecture Notes in Computer Science*, pages 7–20. Springer, 2005.
7. R. E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Trans. Computers*, 35(8):677–691, 1986.
8. P. C. Clements. Managing Variability for Software Product Lines: Working with Variability Mechanisms. In *Software Product Lines, 10th International Conference, SPLC 2006, Baltimore, Maryland, USA, August 21-24, 2006, Proceedings*, pages 207–208. IEEE Computer Society, 2006.
9. T. Curran and G. Keller. *SAP R/3 Business Blueprint: Understanding the Business Process Reference Model*. Upper Saddle River, 1997.
10. K. Czarnecki and M. Antkiewicz. Mapping Features to Models: A Template Approach Based on Superimposed Variants. In R. Glück and M. R. Lowry, editors, *GPCE*, volume 3676 of *Lecture Notes in Computer Science*, pages 422–437. Springer, 2005.
11. K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.
12. K. Czarnecki, S. Helsen, and U. W. Eisenecker. Formalizing cardinality-based feature models and their specialization. *Software Process: Improvement and Practice*, 10(1):7–29, 2005.
13. J. Estublier and R. Casallas. The Adele Software Configuration Manager. In *Configuration Management*, pages 99–139. John Wiley & Sons, 1994.
14. M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, 1983.
15. I. Hammouda, J. Hautamäki, M. Pussinen, and K. Koskimies. Managing Variability Using Heterogeneous Feature Variation Patterns. In *FASE*, pages 145–159, 2005.
16. 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.
17. M. La Rosa, J. Lux, S. Seidel, M. Dumas, and A. H. M. ter Hofstede. Questionnaire-driven Configuration of Reference Process Models. In *Proceedings of the 19th Conference on Advanced Information Systems Engineering (CAiSE 2007)*, Trondheim, Norway, 11-15 June 2007. To appear. Preprint available at QUT ePrints, `http://eprints.qut.edu.au/archive/00005786`.
18. M. Mannion. Using first-order logic for product line model validation. In *Software Product Lines, Second International Conference*, volume 2379 of *Lecture Notes in Computer Science*, pages 176–187. Springer, 2002.
19. 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.
20. 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.
21. K. Pohl, G. Bckle, and F. van der Linden. *Software Product-line Engineering – Foundations, Principles and Techniques*. Springer, Berlin, 2005.
22. K. Pohl and A. Metzger. Variability management in software product line engineering. In *28th International Conference on Software Engineering (ICSE 2006), Shanghai, China, May 20-28, 2006*, pages 1049–1050, 2006.

23. R. S. Pressman. *Software Engineering: A Practitioner's Approach.* Higher Education. Mc Graw Hill, New York, 6^{th} edition, 2005.
24. F. Puhlmann, A. Schnieders, J. Weiland, and M. Weske. Variability Mechanisms for Process Models. PESOA-Report TR 17/2005, Process Family Engineering in Service-Oriented Applications (PESOA). BMBF-Project, 30 June 2005.
25. E. S. Raymond. The CML2 Language. `http://catb.org/esr/cml2/cml2-paper.html`, 2000.
26. 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.
27. M. Rosemann and W. M. P van der Aalst. A Configurable Reference Modelling Language. *Information Systems*, 32(1):1–23, 2007.
28. 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.
29. E. Turkay, 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.