Questionnaire-based Variability Modeling for System Configuration

Marcello La Rosa[1], Wil M.P. van der Aalst[2,1], Marlon Dumas[3,1], 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@tue.nl
[3] University of Tartu, Estonia
marlon.dumas@ut.ee

Abstract. Variability management 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 proposes an approach to capture system variability based on questionnaire models that include order dependencies and domain constraints. The paper presents analysis techniques to detect circular dependencies and contradictory constraints in questionnaire models, as well as techniques to incrementally prevent invalid configurations by restricting the space of allowed answers to a question based on previous answers. The approach has been implemented as a toolset and has been used in practice to capture configurable process models for film post-production.

Keywords: variability modeling, system configuration, questionnaire, software product family

1 Introduction

Explicitly modeling the variability of information and software systems in order to enable their configuration is a well-known approach to achieving reuse [22]. In this context, variability refers to "the parts of a [system] development process and its resulting artefacts that is made to differ between products or in certain situations within a single product" [31]. For example, enterprise system packages such as SAP provide modules and business objects covering common functions such as invoicing, financial reporting and controlling. Analysts and developers configure and compose these parts to meet the requirements of individual customers. This individualization may be performed by means of so-called *reference models* that capture the data, functionality and business processes supported by the system [27, 28]. Similarly, software product families

are an approach to package related functionality into generic software assets, from which system variants are derived [14].

Variability of an information or software system may be captured as a collection of parameters [16] as a collection of features [31], or more generally, as a collection of choices. These choices determine the actions (e.g. model or code transformations) that should 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 focus area of 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 choices that need to be made during configuration of a model or system are often interdependent. For example, once an evaluated receipt mode has been selected, choices regarding the configuration of the invoice reconciliation sub-process become irrelevant. Also, making a certain choice may restrict the allowed choices subsequently. Indeed, not all sequences of configuration choices may lead to valid configurations. Here, by valid configuration we mean a configuration that satisfies a collection of constraints inherent to the application domain (i.e. the *domain constraints*).

The work presented in this paper is motivated by the following question: How to proactively guide a user through the configuration space in such a way that choices are presented at a suitable moment, and choices that may lead to invalid configurations are avoided a priori? With respect to this research question, the contribution of this paper is a variability modeling framework with the following characteristics:

- It provides fine-grained control over the order in which choices are presented to users at configuration time.
- It incrementally prunes the space of allowed choices in order to prevent users from making inconsistent choices during configuration. In particular, the framework is able to statically detect inconsistent variability models.

In most existing variability modeling and system configuration tools (e.g. [33], [9] and [3]), inconsistent choices are detected and reported *a posteriori*, with the drawback that users need to backtrack and revise their choices when an inconsistency is found. Also, limited control is given over the order in which choices are presented to users.

As argued by Batory [5], these limitations can be explained by the lack of a formal underpinning for variability models. In this paper, we follow the

lines drawn by [5] to develop a formal framework for variability modeling that addresses the above two limitations.

Figure 1 provides an overview of the approach. *Configuration models* composed of *questions* capture the way in which the variability of a generic system is resolved at configuration time. Each question refers to a set of *facts* that can be set to true or false. Facts encode the variability of the system, e.g. optional features, values of configuration parameters, etc. The individualization of the generic system is captured by means of *actions*. As the questionnaire is answered, values are assigned to facts, and the resulting valuation of facts determines which actions should be performed on the generic system to derive an individualized system. The framework supports the definition of domain constraints in the form of propositional logic expressions over facts. In addition, questions and facts can be connected through precedence/order dependencies in arbitrary ways, so long as these dependencies satisfy some well-formedness rules. These well-formedness rules are shown to be sufficient to prevent contradictory dependencies that lead to deadlocks during configuration.

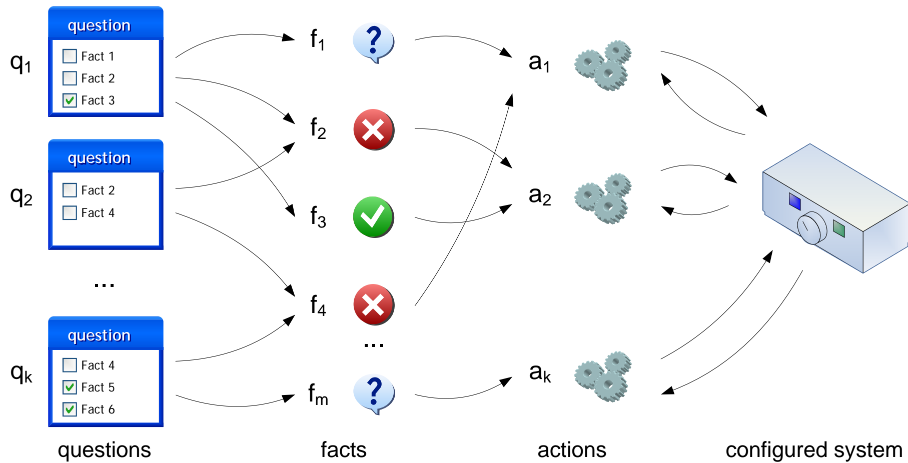

Fig. 1. Overview of the approach.

The proposal includes 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 domain constraints.

A major assumption of the framework 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, thus achieving the second requirement above. The proposed framework does support the use of configuration parameters of arbitrary types (e.g. integer, string), but such parameters can not be used in the expression of domain constraints unless they are discretized.

The framework has been tested by capturing the variability of process models in the film industry. A number of configurable process models and questionnaire-based configuration models have been defined with input from domain experts. Using these configuration models, we have conducted experiments to demonstrate the scalability of the technique for incrementally pruning the configuration space. The experience has also shown that the framework is able to cope with practical variability scenarios involving numerous facts, dependencies and constraints, and that, at least in some domains, the restriction outlined above is not a major impediment.

The rest 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 describes the generation of interactive questionnaires from configuration models. This generation technique has been implemented as a tool outlined in Section 5. Section 6 discusses the evaluation of the proposal through a case study and experiments. Finally, Section 7 compares the proposal with related ones and Section 8 draws conclusions.

2 Overview of 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 configuration time, 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 configuration time its value can be set 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*), which is provided as a suggestion (e.g., it may correspond to the most common choice in that domain). Moreover, a fact can be marked as ‘mandatory’ if it needs to be explicitly set by

the user (e.g., it can be used to refer to an important aspect of the domain that cannot be overlooked).

Facts are grouped into questions according to their content, so that all the facts of the same group can be set at once. For example, the facts “Return Merchandise Claim” and “Loss or Damage Claim” can be grouped under the question “Which Claims have to be handled?”. Questions are thus a structuring mechanism to aid users when assigning values to facts, and are organized in a partial order, such that the user is not posed all the questions at the same time.

A fact can appear in multiple questions. For example, in a screen business project, a fact “DVD” ($true$ if the project releases on DVD, $false$ otherwise) can be seen as a finish format as well as a distribution channel. It can thus be included in the questions “What finish formats have to be supported?” and “What are the distribution channels?”, with thee purpose of facilitating the configuration process. In fact, since questions are organized in a partial order, users can set the same fact by following different answering paths (i.e. by answering different questions), according to their preferences. The value of the fact can only be set the first time, and is preserved in all the subsequent questions that contain it. However, it is possible to change a decision, by rolling back an answered question.

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.

2.1 Working Example

To illustrate these concepts, we consider an order fulfillment process model extracted from the Voluntary Inter-industry Commerce Standard (VICS) [35]. VICS is an industry standard endorsed and used by various large companies that interact with suppliers and logistics providers by means of Electronic Data Interchange (EDI) transactions. This process model includes a number of variability points since it is intended to be adapted by user organizations in order to fit their individual requirements.

The order fulfillment process involves three roles: Supplier, Buyer and Carrier, and may support one or more business functions among Product Merchandising, Ordering, Logistics and Payment. 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" (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 Figure 2 and will be used throughout the paper as a working example. Here questions and facts are assigned a unique identifier 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 Figure 2 (a Ⓣ indicates a fact whose default=$true$, while no symbol means that default=$false$). Selecting the default values leads to a VICS process that implements all the business functions ($f_1, f_2, f_3, f_4 = true$) and all the Logistics' 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 responsible for organizing and scheduling the Pickup (so $f_{16} = true$ and $f_{17} = false$) while the Buyer is responsible for

organizing and scheduling the 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 ($f_{14} = true$, $f_{15} = false$).

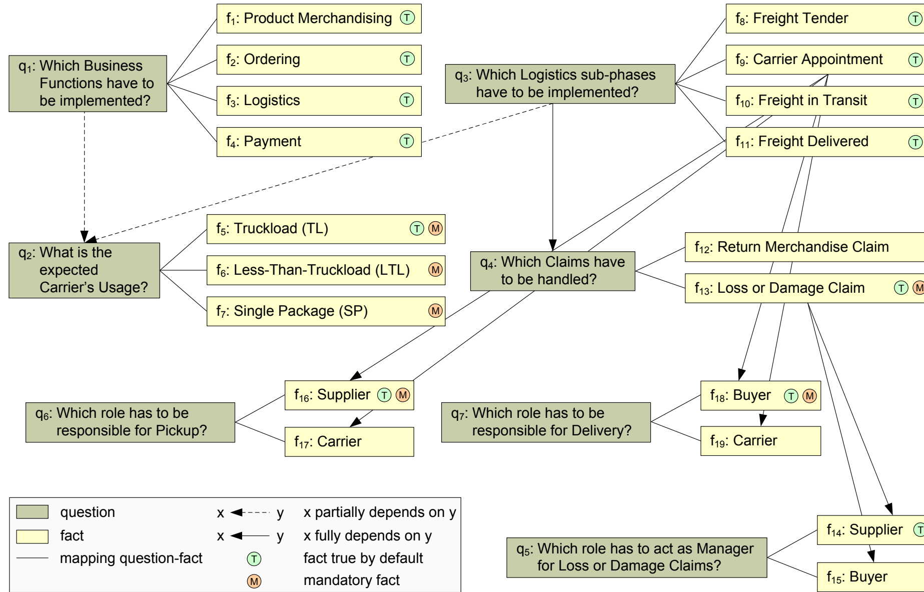

Fig. 2. A possible structure of questions-facts drawn from the VICS EDI Framework.

2.2 Order Dependencies and Constraints

Order dependencies ("dependencies" for short) can be introduced to enforce a partial ordering on facts and on questions. Let us first consider the ordering of facts. For example, we can 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 alternative preconditions with a fact x, where a precondition is a group of facts that all need to be set before x. Only one precondition needs to be satisfied for a dependency to be fulfilled. 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 *partially depends* on another fact if the latter belongs to at least one of its preconditions. On the other hand, a fact *fully depends* on another one if the latter belongs to all its preconditions. A full dependency subsumes a partial dependency.

A partial dependency is represented in Figure 2 by a dashed arrow connecting a fact to its dependent fact, while a full dependency is depicted by a solid arrow. Accordingly, f_{16} and f_{17} fully depend on f_9, i.e. they can be set only after f_9, as they have one precondition containing only f_9. However, if a fact x partially depends on another fact y, it is still possible that x is set before y if a precondition not including y is satisfied.

Dependencies over facts affect the order in which questions are posed to users, since 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 Figure 2. Similarly, q_7 depends on q_3 and q_5 on q_4.

Sometimes though, it may be more natural to express dependencies directly at the level of questions. This is allowed so long as the dependencies defined at the level of questions do not contradict those defined at the level of facts. In Figure 2, q_4 fully depends (directly) on question q_3 and its facts have no dependencies on other facts, whilst q_2 has a (direct) partial 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. Figure 3 shows the final structure that defines the partial order in which the questions of Figure 2 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. Circular dependencies among questions and facts are prevented by means of simple well-formedness rules (further details will be given in Section 3).

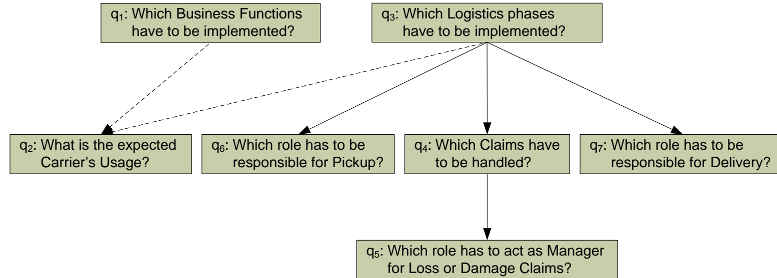

Fig. 3. The partial order over the questions of Fig 2.

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 shipment is to be supported in a configured system. This corresponds to asserting that exactly one fact among f_5, f_6 and f_7 holds 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, 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 model these *constraints* as propositional logic expressions over facts. From an analysis of the VICS EDI Framework, we have derived the following constraints (which refer to the facts shown in Figure 2):[4]

C_1: $f_1 \vee f_2 \vee f_3 \vee f_4$
C_2: $f_3 \Leftrightarrow (f_8 \vee f_9 \vee f_{10} \vee f_{11})$
C_3: $(f_5 \veebar f_6 \veebar f_7) \Leftrightarrow (f_4 \vee f_9 \vee f_{10})$
C_4: $(f_{12} \vee f_{13}) \Rightarrow f_{11}$
C_5: $\neg(f_5 \vee f_6 \vee f_7) \Leftrightarrow \neg(f_4 \vee f_9 \vee f_{10})$
C_6: $f_{13} \Leftrightarrow (f_{14} \veebar f_{15})$
C_7: $(f_9 \wedge \neg f_7) \Leftrightarrow ((f_{16} \veebar f_{17}) \wedge (f_{18} \veebar f_{19}))$
C_8: $\neg f_{13} \Leftrightarrow \neg(f_{14} \vee f_{15})$
C_9: $\neg(f_9 \wedge \neg f_7) \Leftrightarrow \neg(f_{16} \vee f_{17} \vee f_{18} \vee f_{19})$.

C_1 ensures that at least one business function is chosen in q_1. C_3 and C_5 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 affect 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 C_7 and C_9, 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 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 an *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 C_4: $(f_{12} \vee f_{13}) \Rightarrow f_{11}$ and the full 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 set to true before. Similarly, due to C_6: $f_{13} \Leftrightarrow (f_{14} \veebar f_{15})$ and q_5 which indirectly depends on q_4, exactly one

[4] $\veebar$ indicates the exclusive disjunction (XOR), i.e., $f_1 \veebar f_2 \Leftrightarrow (f_1 \vee f_2) \wedge (f_1 \neq f_2)$.

Manager for Loss of Damage Claim is to be selected in q_5, but only after Loss or Damage Claim (f_{13}) 'has been' set to true in q_4.

Because of the domain constraints, answering certain questions can lead to other questions becoming constrained or irrelevant. This suggests that questions should be ordered such that the most discriminating questions are asked first. This way, the configuration space is pruned and the total number of questions that need to be answered is minimized. For example, q_1 and q_3 in the example are highly discriminating questions and it makes sense that they are posed before the other ones. However, dependencies can be defined based on other considerations. For example, in some contexts, dependencies may be defined based on the role of the user(s) that will configure the system. For example, a logistics expert can be first posed questions related to Logistics, while a merchandise expert can be first asked questions related to Product Merchandising. Alternatively, each user can be assigned a subset of questions according to their expertise (distributed configuration).

The framework relies on boolean encodings of the space of possible answers. This encoding enables the use of efficient techniques to incrementally prune the space of answers, so that questions that become constrained or irrelevant due to answers given to previous questions, can be simplified or skipped. On the other hand, this boolean encoding can be a limitation in some scenarios. While enumerated types can be encoded as a collection of boolean values (and this encoding can be made transparent with appropriate tool support), this approach is not applicable for non-enumerated types (e.g. integers, strings). As a tradeoff, the framework supports the definition of questions with non-enumerated types as their space of possible answers. For example, one can define a question "number of carriers?" of type integer and use this as the answer to a question. However, this configuration parameter can not be used in the domain constraints. Alternatively, if a discretization of this configuration parameter is defined (e.g. "one carrier", "between 2 and 5 carriers", and "more than 5 carriers"), each of these discretized values can then be mapped to a fact that can be used to express domain constraints.

2.3 Actions

To propagate configuration decisions to the derivation of an individualized model or system from a generic one, we associate facts to actions. Actions denote modifications to be performed on the system or model to be configured. For example, in the field of software product families, such an action could correspond to removing a code fragment from a software asset. Meanwhile, in the area of business process model configuration, an action could be associated to adding or removing a fragment of the model.

Figure 4 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 dashed boxes and identified by the facts of Figure 2.

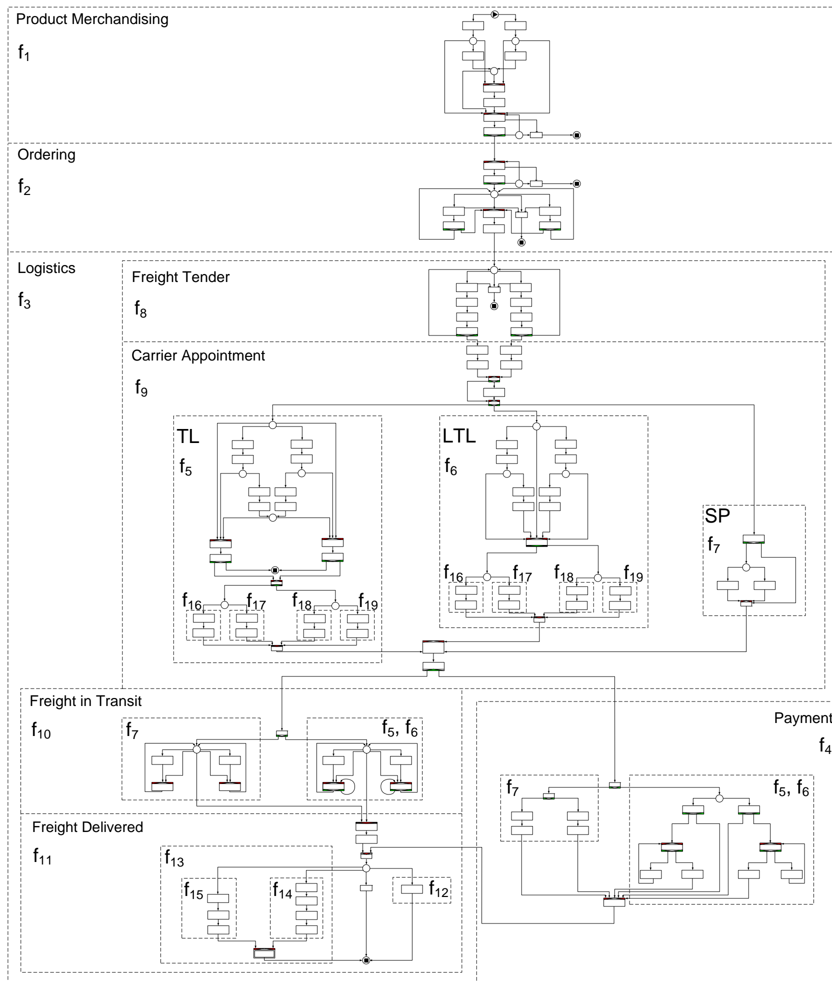

Fig. 4. The order fulfillment process model associated to the facts of Figure 2.

The four main process fragments refer to the Business Functions – Product Merchandise, Ordering, Logistics and Payment – that the process can imple-

[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`

ment. 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}"). By associating to each of these facts an action that corresponds to the removal of the affected process fragments, setting f_3 to $false$ would imply to remove Logistics as well as all the fragments therein. This complies with C_2, which has been built to reflect this 'parent-child' relation 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 C_3, C_5, C_7 and C_9. Similar considerations hold for the remaining process fragments and constraints.

The following section formalizes the notions discussed above. The formalization allows us to convey the ideas in an unambiguous way and will be used as a basis for the implementation presented in Section 5.

3 Configuration Models

We use the concept of *Configuration Model* (CM) to directly capture variants of a system or model in terms of facts, questions and their dependencies. Given a configuration model, a *configuration* is the result of assigning values to each fact by answering the questions. Abstracting away from questions that are not encoded in boolean form (and that are therefore not used to capture domain constraints), we can view a configuration as a valuation of facts that complies with the domain constraints. Below we formally capture this intuition.

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*,*

- $map_{QF} \in Q \rightarrow \mathcal{P}(F) \setminus \{\varnothing\}$[6] *is a function mapping a question onto a set of facts, such that* $\bigcup_{q \in Q} map_{QF}(q) = F$,
- $map_{FA} \in F \rightarrow \mathcal{P}(Act)$ *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* $f \in F$, $pre_F(f)$ *is the set of* preconditions of f, *i.e., each* $F' \in pre_F(f)$ *represents a precondition. A precondition is satisfied if all its facts are set. Only one precondition needs to be satisfied to set* f. *There is always at least one precondition* ($pre_F(f) \neq \varnothing$), *hence a situation with no order dependencies is represented by the empty precondition* ($pre_F(f) = \{\varnothing\}$). *Moreover, function* pre_F *needs to satisfy the following well-formedness rules:*
  1. $\forall_{f \in F} \forall_{r,p \in pre_F(f)} (r \subseteq p \Rightarrow r = p)$, *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. For any* $q \in Q$, $pre_Q(q)$ *is the set of* preconditions of q. *Each precondition corresponds to a set of questions that need to be answered before* q *is answered, but it is sufficient to satisfy at least one precondition. There is always at least one precondition, hence a situation with no order dependencies is represented by the empty precondition. Moreover, function* pre_Q *needs to satisfy the following well-formedness rules:*
  1. $\forall_{q \in Q} \forall_{r,p \in pre_Q(q)} (r \subseteq p \Rightarrow r = p)$, *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. $\forall_{q \in Q} \forall_{Q' \in pre_Q(q)} \forall_{f \in map_{QF}(q)} \forall_{F' \in pre_F(f)} F' \subseteq \bigcup_{q' \in Q'} map_{QF}(q')$, *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).

[6] $\mathcal{P}$ indicates the power set, i.e., each question is mapped onto a non-empty set of facts.

We say a fact f is *meaningful* if it truly represents a variation, i.e. if it allowed by the constraints to assume both values $true$ and $false$. Formally, if there exist $F'_1, F'_2 \in CS$ such that $f \in F'_1$ and $f \notin F'_2$. If a fact is not meaningful, it should not be included in the model, as it would represent a commonality, which is a stable element of the configurable domain.

Actions depend on the type of the system or model to be configured and the language used for its description. For example, if they refer to the configuration of software code trunks/features, the programming language needs to be taken into account. Likewise, if actions refer to process/data models, the modeling notation used for the representation of such models needs to be considered. This is important as actions (and their relations) must not violate the syntactic and semantic rules of the description language. Since we aim at providing a language-independent formalization of variability, a detailed description of actions is left out. In separate work [19] we have explored the use of actions for business process models configuration (further details can be found in Section 7).

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 partially depends *on* f' *iff* $\exists_{F' \in pre_F(f)}\ f' \in F'$,
- f fully depends *on* f' *iff* $\forall_{F' \in pre_F(f)}\ f' \in F'$,
- q partially depends *on* q' *iff* $\exists_{Q' \in pre_Q(q)}\ q' \in Q'$,
- q fully 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 an *AND* relation, while preconditions are in an *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 full dependency. On the other hand, there is a partial dependency from f_1 to f_3 and from f_1 to f_4, as f_3 and f_4 do not belong to each clause of $pre_F(f_1)$.

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 well-formedness rule on pre_F and pre_Q (see Definition 1) 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 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 well-formedness rule on pre_F and pre_Q (see Definition 1) 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 $pre_F(f_3) = \{\{f_1\}\}$ (Figure 5 - a), or a case where $pre_F(f_1) = \{\{f_2\}\}$, $pre_F(f_2) = \{\{f_3\}\}$ and $pre_F(f_3) = \{\{f_1\}, \{f_2\}\}$ (Figure 5 - b) are not allowed according to Definition 1. The reason is that 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. This violates the second well-formedness rule on pre_F given in Definition 1 because of an undesirable cycle. Such undesirable cycles can be caused by both partial and full dependencies.

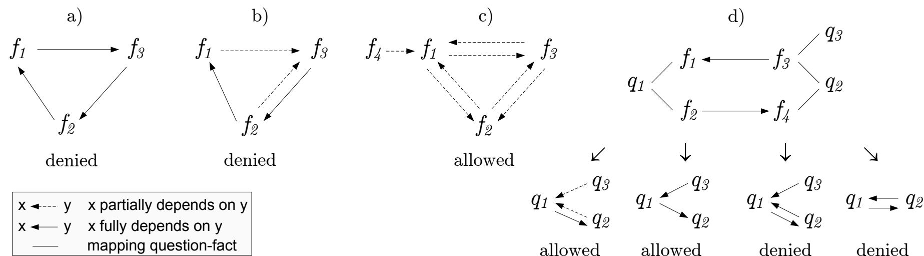

Fig. 5. Examples of circular dependencies over facts and questions.

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

Example 4. A combination where $pre_F(f_1) = \{\{f_2\}, \{f_3\}, \{f_4\}\}$, $pre_F(f_2) = \{\{f_1\}, \{f_3\}\}$, $pre_F(f_3) = \{\{f_1\}, \{f_2\}\}$, $pre_F(f_4) = \{\varnothing\}$ (Figure 5 - 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 well-formedness rule on preconditions does not hold.

The only difference between the definitions of pre_F and pre_Q is the addition of a third well-formedness rule to the latter (see Definition 1), so as to move dependencies over facts to the level of questions without violating them. Given a question q, the formula 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 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 conditions. This is possible since $\bigcup_{q' \in Q'} map_{QF}(q')$ is defined as a superset of all preconditions 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 $pre_F(f_4) = \{\{f_2\}\}$ (Figure 5 - 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 fully 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 well-formedness rule 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 rule, 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 configuration process for a CM. This way we provide executable semantics for the configuration model defined in Definition 1. In a configuration process questions are dynamically posed to users according to the order dependencies, and answers can be given only if these 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:*

- $V = F \rightarrow \{true, false, unset\}$ *is the set of all* facts valuations*, independently of set* CS*,*
- $a \in V$ *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 facts valuations, i.e. “answers” (a) as well as “parts of state” (vs). 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*,*
- $f(s) = f(vs) = \{f \in F \mid vs(f) = false\}$ *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. $t(vs)$, $f(vs)$ *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 are* true *by default,*

– *for* $x, y \in V$ *and* $f \in F$:

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

For each state a set of valid questions is presented to the user. For a question to be valid in a state ($valid(q, s)$), 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 ($valid(a, q, s)$) if and only if all the facts within that question are set and the outcome of the answer ($outcome(a, q, s)$) results in a valid state ($valid(s)$), i.e. a state whose facts valuation complies with the constraints on facts. Also, since facts can appear in more than one question, those 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:*

– $valid(q, s) = q \notin qs \ \wedge \ \exists_{Q' \in pre_Q(q)} \ Q' \subseteq qs$*, 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,*
– $outcome(a, q, s) = (vs \oplus a, qs \cup \{q\})$*, i.e. the state resulting after answering* a *to question* q *in state* s*,*
– $valid(s) = \exists_{F' \in CS} \ (t(s) \subseteq F' \ \wedge \ f(s) \cap F' = \varnothing)$*, i.e. the facts valuation of the state has to comply with the constraints on facts,*
– $valid(a, q, s) = valid(q, s) \wedge t(a) \cup f(a) = map_{QF}(q) \wedge \forall_{f \in map_{QF}(q) \setminus u(s)} \ a(f) = vs(f) \ \wedge \ valid(outcome(a, q, s))$*, 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 CS, 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 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 later used 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,*[7]
- $S_F = \{(vs, qs) \in S_v \mid (f \in F_M \Rightarrow vs(f) \neq unset) \wedge valid(s^*)\}$ *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 reach a final state straightaway, where all the mandatory facts have been set and the remaining ones can take their default.

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

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

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 $f(cf_\sigma) = F \setminus t(cf_\sigma)$,*
- *$Cf_{CM} = \{cf_\sigma \in V \mid (\sigma \in T^+) \wedge valid(\sigma)\}$ 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 i) the state space is finite, and ii) 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 proves 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 application of the result. Before presenting the theorem, we first introduce a shorthand notation.

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. See Appendix.

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.*

Although a fact is meaningful at the beginning, once the configuration process has begun, at a certain state it may turn out from the constraints that such fact can only take one value of the two. In this case an user does not have the freedom to choose, as the value to be given is inferred 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 a fact can appear in multiple questions, 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 (Forceable Fact, 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:*

- $\mathit{forceable}(f, s) = f \in u(s) \wedge \forall_{F_1, F_2 \in CS}\, [(t(s) \subseteq F_1 \cap F_2 \wedge f(s) \cap (F_1 \cup F_2) = \varnothing) \Rightarrow F_1(f) = F_2(f)]$, *i.e. f assumes the same value in all the facts valuations still possible,*
- $\mathit{skippable}(q, s) = \mathit{valid}(q, s) \wedge \forall_{f \in map_{QF}(q)}\, [f \notin F_M \wedge (\mathit{forceable}(f, s) \vee f \notin u(s))]$, *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. In fact the forceability of a fact is determined by the set CS, whereas if all the facts have been previously set, 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, called *Quaestio*, are introduced in the first part of this section.[8] The second part shows how the tool is used to configure the order fulfillment example of Section 2.

5.1 Prototype Implementation

Quaestio is a Java application that guides users through a set of questions given a configuration model as input. The graphical interface comprises 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 fact: its default value, whether it is mandatory, the constraints that bind the fact, the dependencies on other facts, the level of impact on the system or model to be configured, and specific guidelines to configure the fact. A screenshot of the tool with the Fact Inspector is shown in Figure 6.

The input format is described by an XML schema. Figure 7 shows the representation of the schema as UML class diagram.[9] Here each class maps to a complex type in the schema, the aggregation relation maps to the composition of complex types, and the occurrence constraints map to the element cardinality.

We encoded each of the sets in Definition 1 with a complex type, with the exception of sets F_D and F_M – encoded with a boolean element contained in the fact's type (`FactType`) – and set CS. Preconditions for facts and questions are encoded with complex types (`preFElementType` and `preQElementType`) consisting of a list of references (possibly empty) to facts or questions. Each precondition is an element of the respective set of preconditions (`PreFType` and `PreQType`), which at least contains one precondition. Similarly, questions are mapped to facts via a complex type (`MapQFType`). A string element (contained in the root element's type `CModelType`) is used to store constraints expressed as conjunctions of boolean formulas. Other features of Quaestio not detailed here include the ability to present questions whose answers may be integers,

[8] The tool distribution can be downloaded from `http://sky.fit.qut.edu.au/~dumas/ConfigurationTool.zip`

[9] The diagram has been generated with the Eclipse Modeling Framework, `http://www.eclipse.org/modeling/emf`

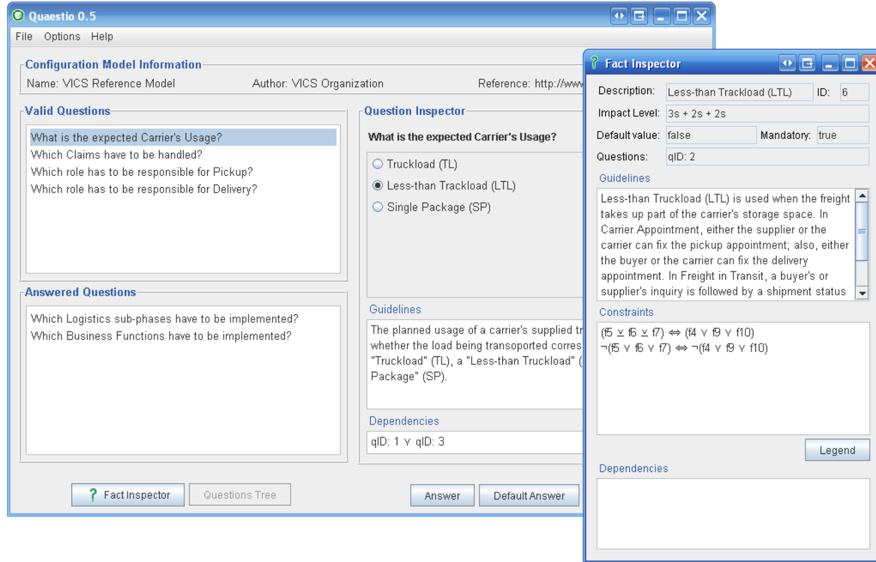

Fig. 6. A screenshot of Quaestio with the Fact Inspector.

floats or strings, though no checks are performed on such questions and they are orthogonal to any constraint.

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 and updates 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.

A separate XML format is used to map, in a language-independent way, an action to a set of facts and to a set of variation points of the system or model to be configured. The XML schemas for the input, output and mapping format, and the files for the application example, are available in the tool distribution.

The formal framework proves that a few syntactic checks on the dependencies between facts and questions are enough to ensure deadlock-free configuration processes (see Corollary 1). This result ultimately enables an efficient implementation of the tool, which only needs to check that the configuration model satisfies the well-formedness rules.

For efficiency reasons, we decided not to build the state space of the questionnaire (as defined in Section 4). Instead, we opted for a dynamic generation

of the state space. For each answer given, the next state is calculated by scanning only those valid questions that are still unanswered. For each of them, we check if at least one precondition can be satisfied (we know there will be at least one). If so, a question is put into the Valid Questions list if it is not skippable, otherwise it is added straight to the Answered Questions list.

Fig. 7. The UML representation of the XML schema for the input file.

In order to check domain constraints, to detect skippable questions, and to prune the space of answers, the tool implementation embeds a propositional logic calculator[10] based on Shared Binary Decision Diagrams (SBDDs) [11, 21]. SBDDs are canonical forms of boolean formulas for which there are efficient analysis algorithms. They are based on the classical BDDs with the advantage of being always cheaper in size and time computation.

The calculator builds an SBDD in memory from the conjunction of constraints c, and returns a polynomial representation $pr(c)$ in conjunctive normal

[10] The calculator can be downloaded from `http://www-verimag.imag.fr/~raymond/tools/bddc-manual`

form. If the negation of c is a tautology, the "always-false formula" is returned as output (i.e. $pr(c) = \mathit{false}$), meaning the constraints are not satisfiable. This output is interpreted by Quaestio to check for satisfiability of the constraints before starting the configuration. Similarly, the tool detects if a fact is not meaningful. The output is also used to evaluate the type of questions. For example, $(pr(c) \Rightarrow xor(map_{QF}(q)) = true$ indicates that q is an XOR question.

At configuration time, an SBDD is constructed from the conjunction of $pr(c)$ and each potential answer given by the user: so long as the conjunction yields $false$, the answer is not valid and the Answer button is kept disabled. In this way the tool prevents users from entering responses which would violate the constraints. For each valid answer that is fed in Quaestio, a new SBDD is constructed by updating $pr(c)$ with the values of the facts that have been set. This avoids to compute the canonical form of the constraints every time from scratch, improving the overall response-time. After an answer, some facts may become forceable and thus some questions may be skippable. This condition is tested against $pr(c)$ in a way similar to the evaluation of the type of questions. For example, $(pr(c) \Rightarrow \neg f) = true$ indicates that f is forceable to $false$.

Performance measurements of the tool's scalability are provided in Section 6.2. The main features of Quaestio 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 those facts that have already been set or that are forceable, does not deviate from the default, and
  - the resulting valuation is valid given the current state;
- fact's 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.

The rollback operation was implemented with the purpose of preserving the answering order. When a question is rolled back, the current state is set to the one before answering the question, and hence, all the questions that were

answered thereafter are rolled back too. Moreover, a fact occurring in multiple questions is kept forced to the value it was set the first time, until all its questions are rolled back.

It is possible to implement a selective rollback that ignores the answering order, by suppressing the dependencies of the answered questions. In this way we could, e.g., roll back q_4 without rolling back q_5, although the latter has a dependency on the former. The new state, with q_5 in qs but not q_4, is still a valid state, since q_5 has no dependencies anymore.

If a question being selectively rolled back had an interplay with some question already answered, its facts might be forced to assume an exact value. For example, if q_5 is answered with at least one of its facts set to true (i.e. by choosing a Manager for Loss or Damage Claim), q_4 will be rolled back with f_{13} forced to true, as per C_6.

5.2 Sample Configuration Process

This section shows a sample configuration process for the order fulfillment process model of Figure 4. For convenience, we introduce the notation $a_m^{q_n}$ to indicate the valuation that is given by the answer m to the facts of question n (where the remaining facts that are not set by the answer are left out). Also, the symbols T and F are shorts for a *true*, resp. for a *false*, valuation.

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 implementing 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 (Figure 8). 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 giving 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's description).

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 partial 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 has been grayed out (Figure 9). The system has reacted to a_1 by setting f_3 in order to comply with C_2. 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.

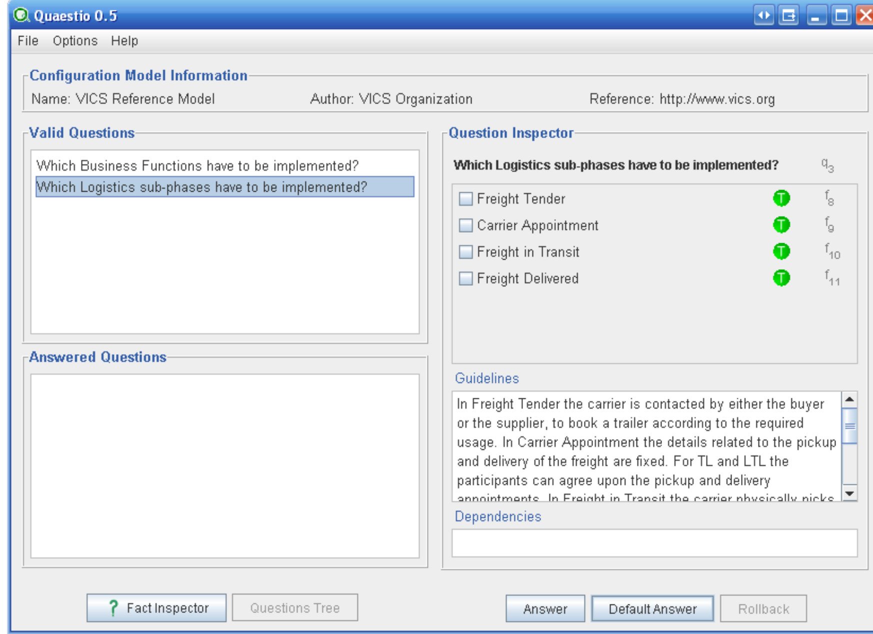

Fig. 8. State s_1: the only valid questions are q_1 and q_3.

After a_2, we reach 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 C_3 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's description). We select Single Package and $a_3^{q2} = \{(f_5, \mathsf{F}), (f_6, \mathsf{F}), (f_7, \mathsf{T})\}$ 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 C_9. Thus $a_4^{q6} = \{(f_{16}, \mathsf{F}), (f_{17}, \mathsf{F})\}$ and $a_5^{q7} = \{(f_{18}, \mathsf{F}), (f_{19}, \mathsf{F})\}$ 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 gray in Figure 10) 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 $a_6^{q4} = \{(f_{12}, \mathsf{F}), (f_{13}, \mathsf{T})\}$ 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 q_4. s_7 is a final state as all the mandatory facts have already been set and the

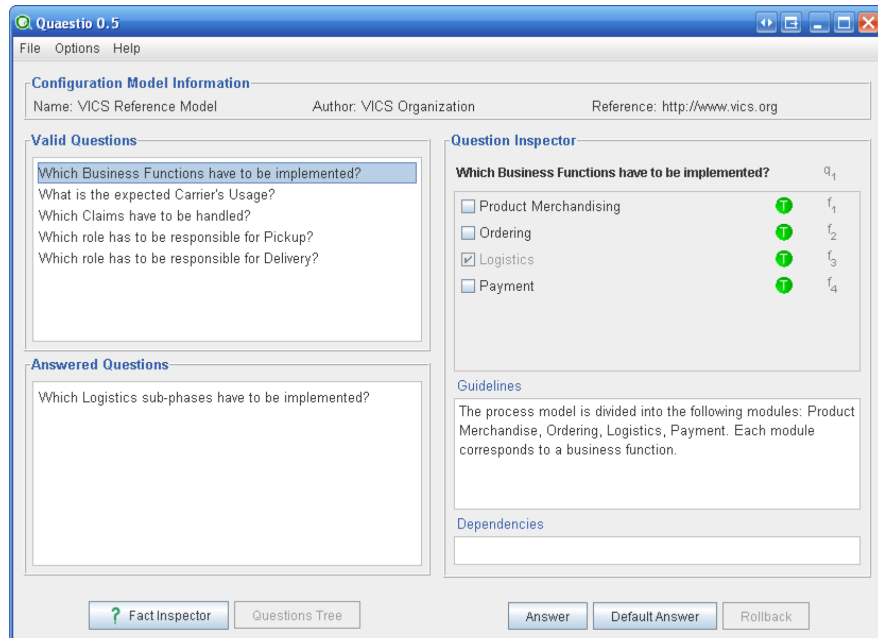

Fig. 9. State s_2: f_3 has been forced to $true$ in q_1 in order not to violate C_2.

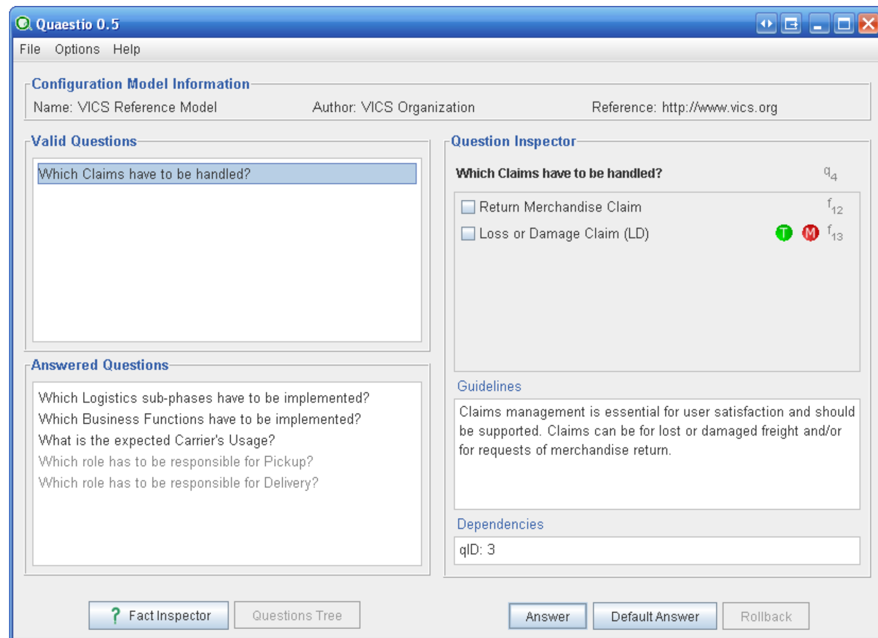

Fig. 10. State s_6: q_6 and q_7 have been skipped as their facts can only be negated.

remaining ones still unset (f_{14} and f_{15}) can take their defaults without violating the constraints. q_5 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 configured order fulfillment process model pictured in Figure 11. This model is the result of performing certain actions on the initial model to remove the irrelevant process fragments, i.e. those process fragments whose facts have been set to false.

6 Evaluation

This section describes a case study in which we applied the framework to capture variability in process models for the film industry. It also describes a set of performance measurements to assess the scalability of the tool.

6.1 Film Industry Case Study

This case study was conducted with domain experts from the Australian Film Television & Radio School (AFTRS).[11] This school has engaged, with other stakeholders, in an initiative to capture business process models in the film industry. However, it was quickly noted that process models in this industry have a high degree of variability. Basically, each production project works different from the others.

In this case study, we focused specifically on process models for the post-production phase. Post-production starts after the shooting phase and deals with the design and edit of the picture, music and soundtrack of a screen project.

[11] The school's web site can be accessed at `http://aftrs.edu.au`

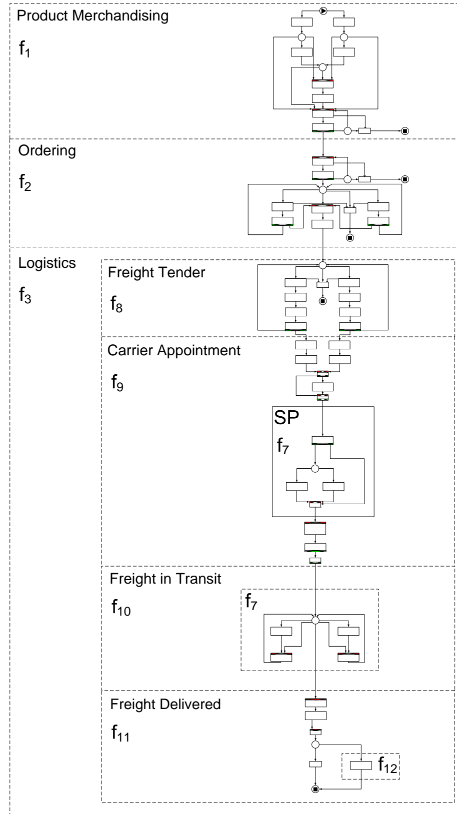

Fig. 11. The configured order fulfillment process model.

Creativity is a distinguishing feature of this domain: indeed, a single decision made by the director, such as that of not having any music, can radically change the whole post-production. This necessarily leads to a great deal of variability: for this reason, the domain in question was deemed suitable to evaluate our framework.

Typical choices in post-production relate to the edit equipment, the format of the release, the addition of visual/audio effects, the presence of artificially or recorded music, etc. All these choices depend on the type of project (movie, documentary, etc.), the distribution channel (cinema, TV, home, etc.), and above all the available budget.

A number of configurable process models were defined to describe the post-production phase and its variations. In this phase, we received input from a producer and a sound editor. To model these processes, we chose the Configurable

Event-driven Process Chain (C-EPC) notation [28], as it allows one to capture both variability and commonality in a process model. C-EPC is an extension of a popular modeling notation, namely EPC, that includes constructs for defining variation points (more details on C-EPC are provided in Section 7.

The complete C-EPC process model for film post-production consists of 568 process elements (spread across different diagrams), of which 98 are variation points, each allowing a number of process variants for a total of around 230,000 valid process configurations.

Having defined the C-EPC model for film post-production, we identified a set of facts to capture the variability of the reference model, and we grouped them into suitable questions. In this phase, a screen composer and a picture editor were also involved. Firstly, we defined one fact for each factor that yields a high number of process variations. Such factors, like the budget level, correspond to domain decisions which are usually not captured in the process model. Secondly, we encoded each fine-grained decision with one fact. Such decisions have little or no impact on the rest of the system. For example, the type of editing suite only affects the medium format, while both the type of suite and the format are determined by the available budget. Thirdly, we defined a system of constraints to encode the interplay among these facts, and we used the SBDD calculator to check for satisfiability. With the help of the calculator, we also realized that some fine-grained facts were redundant, as the variants they captured could be determined by the configuration of other facts. For example, a Telecine suite is a piece of equipment that is only required in particular situations, such as when the shooting is on film and the release is on tape. Thus, it was sufficient to encode all the shooting and release formats, to indirectly capture the Telecine options.

Order dependencies were set in a way to pose the most discriminating questions first, and then fine-tuned according to the indications of the domain experts, to better respond to their needs. Sample questions were: “What is the expected budget for the project” (high impact) and “Are any color adjustments to be performed during tape finishing?” (low impact). We added contextual information to the majority of questions and facts, in the form of textual guidelines, as an aid during the configuration process. These guidelines were derived from the constraints and enriched by information taken from the literature.

The complete configuration model consists of three sub-questionnaires (Picture edit, Sound edit and Screen composition), and an introductory questionnaire which links them together, for a total of 46 questions and 148 facts. An excerpt of this configuration model can be found in the tool distribution. The mapping between configuration models and C-EPCs has been reported in separate work [19].

This experience has shown that the framework is able to cope with practical variability scenarios involving numerous facts, dependencies and constraints, and that, at least in some domains, the restriction to boolean variables is not a major impediment. In this scenario, it was not necessary to introduce questions with non-enumerated domains.

At the time of writing, the configuration model is being used by the AFTRS staff to generate configured process models. The questionnaires and the generated process models are expected to be used for teaching purposes in post-production subjects, and for communicating requirements and decisions in student projects. It is also expected that in future, these models will be used in production projects outside the school.

6.2 Performance Experiments

To demonstrate the scalability of our tool, we measured the time (t_0) taken by the embedded SBDD calculator to transform the boolean function corresponding to the conjunction of all the domain constraints, into a canonical form that is then used for subsequent steps. This was found to be the critical factor to ensure scalability of the tool when the number of facts and constraints increases.

The complexity of a boolean function can be determined by the number of nodes of the corresponding SBDD constructed in memory. As the SBDD size increases, the complexity of the function increases. This in turn entails that the calculator will take more time to reduce and manipulate the function.

The boolean function from the above case study, which ranges over 148 facts, was transformed to an SBDD of 2048 nodes with a response time of 28.3 ms (on average). This is the time needed by the calculator to check the satisfiability of the function, and it represents the upper bound time of any (subsequent) evaluation of the same function. In fact, as users enter answers to the tool, the configuration space is pruned and thus the complexity of the boolean function decreases, lowering down the response time. In other words, an user has to wait at most a time equal to t_0 to see if an answer is correct. Therefore, t_0 is an important yardstick of the efficiency of a pruning technique.

To simulate complex functions, we measured t_0 as we doubled the number of facts starting from 25, and randomly replicated the constraints over the new facts, yielding a proportional increase of the SBDD size. The experiments were conducted on an Intel Pentium M processor (2.0 GHz, 533 MHz FSB, 2 MB L2 cache), 1GB RAM (DDR2 533), 100GB HDD Serial ATA; the results are shown in Table 1.

Although t_0 increases exponentially, we can observe that the SBDD size must be over 21,000 nodes for performance to start degrading significantly (t_0

Facts	25	50	100	200	400	800	1600	3200
SBDD Nodes	338	679	1361	2725	5453	10909	21821	43645
t_0 (ms)	3.2	12.3	22	67.2	457.8	3040.6	23859.4	164400

Table 1. Performance measurements.

is around 24 s). This size corresponds to 1600 facts, which represents a configuration space of considerable size [5].

In the above experiments, pseudo-constraints were generated with an interplay granularity of 25–50 facts. Although this can seen as a limitation, it reflects the fact that in practical scenarios, most configurable variables interact with a limited number of other variables. This allowed us to increase the size of the SBDD proportionally to the increase of the number of facts, and to draw conclusions based of the latter.

7 Related Work

7.1 Software Product Line Engineering

Variability modeling has been widely studied in the field of Software Product Line Engineering (SPLE) [22, 12]. Among others, two research streams have emerged in SPLE, namely Software Configuration Management (SCM) [24] and Feature Diagrams (FDs) [30].

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 [16] 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) [32], 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 [33]. OCML allows developers to capture 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 prevent the user from entering inconsistent values, nor is it able to skip options that are no longer relevant.

More generally, none of these approaches provides fine-grained control over the order in which choices are presented to users at configuration time. 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 preemptively discard invalid answers to questions based on answers to previous questions.

FDs are a family of techniques for describing software product lines in terms of their features [7, 14, 20, 15]. A number of feature modeling languages have been proposed since FDs were first introduced as part of the FODA (Feature Oriented Domain Analysis) method [18]. In general, 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 [14]. Constraints can be expressed as plain text [7] or as formulas over features [20, 5], specified by means of a proper grammar (e.g. a limit in the number of sub-features a feature can have).

Feature modeling languages provide rich mechanisms for capturing advanced variability patterns, such as for example “iterator variability”, whereby an a priori unknown number of instances of an asset need to be configured and each of these instances requires a separate configuration. Such variability patterns are useful for example when configuring component-based software product lines, which are beyond the scope of our proposal.

Feature modeling languages have been embodied in a number of tools. Some of them rely on a boolean encoding of features. This is the case of the Guidsl module in AHEAD [6] and of Pure::Variants [25]. Other tools can check constraints over non-boolean variables but validate a configuration only a posteriori [3, 9, 10].

In contrast to our proposal, none of these approaches provides fine-grained control over the order in which choices are presented to users at configura-

tion time. In particular, FeaturePlugin [3] provides a wizard to traverse a feature model in a predetermined order (depth-first), but does not support other orderings. Also, the above tools are not able to incrementally prune the configuration space, so that users are preemptively stopped from entering configuration values that would lead to inconsistent configurations (viz. the domain model). Pure:variants comes close to providing such support. It includes an auto-resolving feature that, given a constraint "A requires B", if feature A is selected then feature B becomes automatically selected. However, Pure::variants does not support arbitrary constraints between features (i.e. "facts" in our framework). The possibility of using satisfiability solvers to incrementally prune the configuration space has been raised by Batory [5] but to the best of our knowledge, Quaestio provides the first concrete realization of this idea.

Another approach related to FDs is presented in [17]. 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.

A major difference between our approach and the above body of research lies in the representation of domain variability. SCM and FD approaches capture variability and commonality in the same model, i.e. there is no clear separation between what can vary and what is always stable. For example, in a feature diagram, features can be *mandatory* – if they must be included in the system or model, or *optional* – if they can be excluded. Although the latter always represents a variability, the former does not always represent a commonality. In fact, if a mandatory feature, or any of its parents, has, e.g., an XOR relation with other features, it can still be excluded from the system depending on the configuration of the other features. Another situation is when a mandatory feature is child of an optional feature, and as such can be excluded if its parent is excluded. This lack of separation between stable and variable aspects, can increase the model complexity and hinder the communication of variability to stakeholders [23].

In our proposal, variability and commonality are captured separately: our configuration models focus on resolving variability, while the commonalities are captured in a generic model (e.g. a C-EPC). In this respect, our approach is closer to the principles of Orthogonal Variability Models (OVMs) [22, 23]. An OVM documents the variability of a product line in a dedicated tree-structure

diagram, by means of so-called *variation points*. The *variants* for each variation point are linked to a separate conceptual model, representing the product line to be configured, where both variability and commonalities are modeled. In our approach we explicitly model this relation by means of *actions*, that are used to reflect the effects of a questionnaire-driven configuration on the system or model to be configured. Our facts can be compared to variants, and questions to variation points, i.e. answering a question would allow one to determine which variants are chosen for a given variation point. Having said that, a configuration model offers more flexibility than a tree-like structure, as we can express non-hierarchical dependencies among features, and questions can refer to multiple variation points/features. Moreover, our approach documents the process through which a system or model is to be configured. For this reason, it can be used to complement an FD or OVM. To the best of our knowledge, this is novel in SPLE.

The relations among our approach, SCM, FD and OVM are depicted in Figure 12.

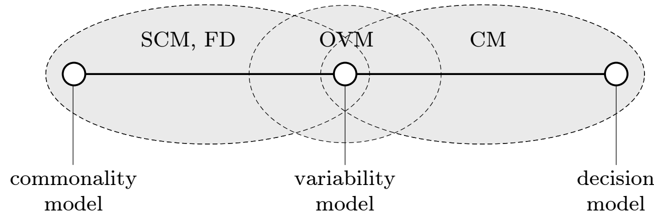

Fig. 12. Comparison among SCM, FD, OVM and CM.

Bayer et al. [8] propose a consolidated meta-model for variability modeling in SPLE. This proposal also follows the principle of separating variability from commonality. In addition, it makes a distinction between variability itself and its resolution. The meta-model classifies variability constraints into three categories: “requires constraints” (if a model element is selected other model elements must be selected too), “excludes constraints (the opposite), and general constraints. Our proposal covers general constraints so long as these can be expressed in propositional logic. The meta-model of Bayer et al. also includes a classification of variability specifications (called *transformers*). Examples of transformers include *property transformers*, whereby a decision needs to be made regarding the value of a property, and *alternative transformers*, whereby a value needs to be selected among a set of alternatives. Our framework can

capture the transformers outlined by Bayer et al., with the restriction that constraints can not be defined over non-discretized space of possible answers. Also, our framework does not address the specification of resolution mechanisms (i.e. *actions*) which Bayer et al. address in details.

7.2 Decision Models and Questionnaire Systems

Configuration models can also be seen as decision models [2], since order dependencies are exploited to offer decision support through an interactive configuration process. Moreover, the questionnaire tool allows users to evaluate and compare alternative answers by means of guidelines, constraints, and by providing information on the impact of facts on the system or model to be configured. In this respect, our proposal shares commonalities with the CML2 language which was designed to capture configuration processes for the Linux kernel [26]. 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.

The KobrA method [4] also relies on decision models to capture configuration choices in SPLE. In KobrA, decision models (called *decision libraries*) are composed of *decisions*. A decision is associated to an enumerated set of possible *resolutions*, and each of these resolutions is linked to one or many variation points. For example, in [4] the authors describe an application of the KobrA method in which variation points are defined in an activity diagram (e.g. to indicate that a certain sub-process is optional). These variation points are then mapped to decisions. For example, a decision could be the “type of payment”, and depending on the type of payment chosen, certain activities in the diagram are dropped. Unlike our proposal, the decision models in KobrA do not include dependencies and constraints.

Our tool is also related to questionnaire systems. A range of commercial products, such as Vanguard Software’s Vista [34], support the definition of online questionnaires and the collection and analysis of responses. Such systems rely on the notion of *question flows*, 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 in advance the points 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).

7.3 Configurable Process Models

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 [29, 13]. In [29] 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 [13] 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, return type). However, the approach only supports simple mapping of features to standard variability mechanisms provided by UML (e.g. decision nodes).

Another approach to capture variability in process models is represented by C-EPCs [28]. C-EPCs extend the EPC notation by identifying a set of *configurable nodes* in the model, to which *alternatives* are assigned to restrict their behavior. A configurable node can be any process function (activity) or connector, depicted with a thicker border in the model. For example, a configurable function can be set to the alternative 'OFF' to be disabled, and a configurable OR-split can be set to the alternative 'XOR-split', to restrict the outgoing flow to a single branch. Constraints over alternatives, called *requirements*, can be captured via boolean expressions. Once all the configurable nodes are assigned an alternative that complies with the requirements, the C-EPC is transformed into a syntactically-correct EPC.

In [19], we discuss the issue of configuring process models by domain experts, who usually lack a specialized modeling background, and propose the questionnaire-based approach as a solution. For illustration, we also show how the approach can be applied to C-EPC. The idea is that each configurable node of a C-EPC and its alternatives, can be associated with boolean expressions over the facts of a configuration model. Such expressions embody the requirements of the configurable process and the constraints of the domain. Thus, an alternative is selected whenever the corresponding boolean expression evaluates to true, triggering the execution of an action to configure the node. As a result of

filling out the questionnaire, an EPC is produced by means of a tool that interfaces with Quaestio.

8 Conclusion

This paper presented a formal framework for representing system variability. The framework relies on configuration models composed of questions and (boolean) facts, which encode possible answers to questions. These answers determine which actions should be performed on a generic model or system in order to derive an individualized version of it. These questionnaire-based models support the definition of dependencies, both at the level of questions and at the level of facts, as well as domain constraints expressed in propositional logic. Simple well-formedness criteria ensure that no circular dependencies may occur, while satisfiability solving techniques are used to ensure the consistency of domain constraints and to incrementally prune the space of allowed answers at configuration time. The framework is independent of the notation(s) used to represent the system itself. While the development of the framework was motivated by the need to support the configuration of business process models, it may be applied to support the configuration of other types of models (e.g. data models) or software artifacts in general, so long as appropriate types of actions are defined to match the notation(s) used.

An embodiment of the framework was presented in the form of a configuration tool that guides users through a set of questions in an order consistent with the dependencies between questions and facts, and in such a way that violations of domain constraints 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 rollback previous answers.

The framework was illustrated using a standard reference model for collaborative B2B processes. It has also been applied to capture reference process models in the field of film post-production. In this application domain, processes are not fixed, but instead they vary from one project to another depending on a range of factors. By capturing this variability using the proposed framework, stakeholders can then individualize a generic process model and obtain a tailor-made process model for a given production project.

The current version of the configuration tool only supports the configuration phase, i.e. guiding a user through the process of making configuration choices and applying the resulting configuration on a generic model to obtain an individualized one. In future work, we plan to extend the tool to support a wider spectrum of the system configuration lifecycle. In particular, we intend to implement an editor of configuration models supporting the definition of different

types of questions. For example, a user should be able to define the range of answers to a question to be an enumerated type, and the tool should then generate a boolean encoding of this domain and hide the details of this encoding in the domain constraints. We also foresee that the configuration model editor will be able to suggest an "optimized" ordering of questions, such that more discriminating questions are given precedence over less discriminating ones, while still allowing the modeler to override the suggested ordering. The tool should also support the maintenance of configuration models, such as estimating the impact of removing a fact from an existing configuration model.

Acknowledgments. The authors would like to thank Katherine Shortland, Mark Ward and the other members of the AFTRS, for their valuable contribution in the evaluation of the framework.

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. L. Alter. *Decision support systems: current practice and continuing challenges*. Addison-Wesley, 1980.
3. M. Antkiewicz and K. Czarnecki. FeaturePlugin: Feature Modeling Plug-In for Eclipse. In *Proceeings of the 5th Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'04), Eclipse technology eXchange (ETX) Workshop*, 2004.
4. C. Atkinson, J. Bayer, and D. Muthig. Component-based product line development: The KobrA approach. In *Proceedings of the 1st Software Product Line Conference*, pages 289–309, Denver, Colorado, 2000. Kluwer.
5. D. Batory. Feature Models, Grammars, and Propositional Formulas. In J. H. Obbink and K. Pohl, editors, *Software Product Line Conference*, volume 3714 of *Lecture Notes in Computer Science*, pages 7–20. Springer, 2005.
6. D. Batory. AHEAD Tool Suite, `http://www.cs.utexas.edu/users/schwartz/ATS.html`. Accessed: January 2008.
7. D. Batory and B. J. Geraci. Composition Validation and Subjectivity in GenVoca Generators. *IEEE Transactions on Software Engineering*, 23(2):67–84, 1997.
8. J. Bayer, S. Gerard, O. Haugen, J. Mansell, B. Moller-Pedersen, J. Oldevig, P. Tessier, J.-P. Thibault, and T. Widen. Consolidated product line variability modeling. In T. Käkölä and J.C. Dueñas, editors, *Software Product Lines – Research Issues in Engineering and Management*, pages 195–241. Springer, 2006.
9. T. Bednasch, C. Endler, and M. Lang. CaptainFeature, `http://sourceforge.net/project/captainfeature`. Accessed: January 2008.
10. Big Lever Software Inc. Gears, `http://www.biglever.com`. Accessed: January 2008.
11. R. E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Trans. Computers*, 35(8):677–691, 1986.
12. P. C. Clements. Managing Variability for Software Product Lines: Working with Variability Mechanisms. In *Proceedings of the 10th International Conference on Software Product Lines (SPLC'06)*, pages 207–208, Baltimore, Maryland, USA, August 21–24 2006. IEEE Computer Society.

13. 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.
14. K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.
15. 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.
16. J. Estublier and R. Casallas. The Adele Software Configuration Manager. In *Configuration Management*, pages 99–139. John Wiley & Sons, 1994.
17. I. Hammouda, J. Hautamäki, M. Pussinen, and K. Koskimies. Managing Variability Using Heterogeneous Feature Variation Patterns. In *Fundamental Approaches to Software Engineering (FASE'05)*, pages 145–159, 2005.
18. K. C. 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. Available at `http://www.sei.cmu.edu/publications/documents/90.reports/90.tr.021.html`.
19. 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 International Conference on Advanced Information Systems Engineering (CAiSE'07)*, Trondheim, Norway, 11-15 June 2007.
20. 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.
21. S. Minato, N. Ishiura, and S. Yajima. Shared Binary Decision Diagram with Attributed Edges for Efficient Boolean function Manipulation. In *Proceedings of the 27th ACM/IEEE Conference on Design Automation (DAC'90)*, pages 52–57, 1990.
22. K. Pohl, G. Böckle, and F. van der Linden. *Software Product-line Engineering – Foundations, Principles and Techniques*. Springer, Berlin, 2005.
23. K. Pohl and A. Metzger. Variability Management in Software Product-line Engineering. In *Proceedings of the 28th International Conference on Software Engineering (ICSE'06), Shanghai, China, May 20-28, 2006*, pages 1049–1050, 2006.
24. R. S. Pressman. *Software Engineering: A Practitioner's Approach*. Higher Education. Mc Graw Hill, New York, 6th edition, 2005.
25. Pure-Systems. Pure::variants, `http://www.pure-systems.com`. Accessed: January 2008.
26. E. S. Raymond. The CML2 Language. `http://catb.org/esr/cml2/cml2-paper.html`, 2000, Accessed: January 2008.
27. 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.
28. M. Rosemann and W. M. P van der Aalst. A Configurable Reference Modelling Language. *Information Systems*, 32(1):1–23, 2007.
29. A. Schnieders and F. Puhlmann. Variability Mechanisms in E-Business Process Families. In *Proceedings of the 9th International Conference on Business Information Systems (BIS'06)*, pages 583–601, Klagenfurt, Austria, 2006.
30. Pierre-Yves Schobbens, Patrick Heymans, and Jean-Christophe Trigaux. Feature Diagrams: A Survey and a Formal Semantics. In *Proceedings of the 14th IEEE International Conference on Requirements Engineering*, pages 136–145, Minneapolis/St.Paul, Minnesota, USA, September 11–15 2006. IEEE Computer Society.

31. M. Svahnberg, J. van Gurp, and J. Bosch. A Taxonomy of Variability Realization Techniques. *Software Practice & Experience*, 35:705–754, 2005.
32. 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.
33. 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.
34. Vanguard Software Co. Vista, `http://www.vista-survey.com`. Accessed: January 2008.
35. Voluntary Inter-industry Commerce Standard (VICS). *EDI Framework*. `http://www.vics.org`. Accessed: January 2008.

Appendix

Proof of Theorem 1 from Section 4:
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 2nd well-formedness rule on 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 $valid(a, q, s)$.