Proc. 2nd Int. Conf. on Telecommunications and Electronic Commerce (ICTEC), 1999

Efficient Support for Decision Flows in E-Commerce Applications

Richard Hull[*], Francois Llirbat[◇], Jianwen Su[†],
Guozhu Dong[‡], Bharat Kumar[*], and Gang Zhou[*]

[*]Bell Laboratories, Lucent Technologies
[◇]INRIA, Rocquencourt, France
[†] University of California, Santa Barbara
[‡]Wright State University
{hull,bharat,gzhou}@research.bell-labs.com
francois.llirbat@inria.fr, su@cs.ucsb.edu, gdong@cs.wright.edu

Abstract

In the coming era of segment-of-one marketing, decisions about business transactions will be quite intricate, so that customer treatments can be highly individualized, reflecting customer preferences, targeted business objectives, etc. This paper describes a paradigm called “decision flows” for specifying a form of incremental decision-making that can combine a myriad of diverse business factors and be executed in near-realtime. Starting with initial input, a decision flow will iteratively gather and derive additional information until a conclusion is reached. Decision flows can be specified in a rules-based manner that generalizes so-called “business rules” but provides more structure than traditional expert systems. In many cases business managers will be able to understand and even modify decision flows.

We also introduce an execution model that supports eager and parallel execution (a.k.a. speculative execution) of decision flow tasks. This paper develops algorithms that can help minimize response time and/or workload. The algorithms detect at runtime three key properties of tasks: *eligible* for eager evaluation, even if the output may not be used in the final output; *unneeded*, i.e., the task is not needed for completing the decision flow instance; and *necessary*, i.e., the task is definitely needed for completing the decision flow instance.

1 Introduction

A variety of technologies will be needed to support the explosive growth of electronic commerce. Research challenges include the development of new frameworks, infrastructures, and protocols that (a) support communication between e-commerce players, and (b) permit individual players to maximize their effectiveness when using e-commerce. The current paper focuses on the second area. Specifically, the paper describes a paradigm called “decision flows” for specifying business policies and executing them in near-realtime. Decision flows support a form of incremental decision-making, that can easily incorporate a myriad of business factors and specify the relative weights they should be given. Decision flows support a rule-based style of specifying decision policies; they are more expressive than decision trees and traditional business rule systems, but can nevertheless be understood and even modified by business and marketing managers. As such, decision flows are ideal for making many different kinds of decisions during e-commerce transactions. This paper describes and illustrates decision flows, and introduces algorithms for minimizing the response time and work load of decision flow execution.

A decision flow consists of a family of attributes which may be evaluated during execution. Some of the attributes will be “target” and embody the output of a decision flow, e.g., what priority of service to give this customer, or what promotional image to display on the next web page. Other attributes correspond to intermediate results of the decision flow. For example, a “promo hit list” attribute might hold a listing of potential promo messages to display, along with scores combining the likelihood that a customer will buy the promo and the potential profit that might be derived. Some intermediate attributes might gather data from external sources, such as databases. Since attribute evaluation can have a real cost, enabling conditions are used to decide which attributes should be evaluated. The set of data flow and control flow dependencies in a decision flow must form an acyclic graph. The “attribute-centric” perspective of decision flows permits a systematic approach for specifying what factors should be incorporated as a decision is being made.

In the decision flow model a variety of mechanisms are provided for specifying how attributes should be evaluated. This includes user-defined functions and database dips. Importantly, attribute evaluation can be specified using simple-to-understand mechanisms for synthesizing and aggregating information. For example, a simple table look-up can be used, e.g. to decide between giving a customer a normal marketing treatment, or a special collections treatment. A set of rules can be specified, each of which potentially contributes a value for the attribute, along with a “combining policy”, which describes how the contributed values should be combined. As a simple example, each rule might contribute a possible integer value for the business priority of a customer contact, and the values contributed by rules with true condition might be combined using aggregate functions, such as maximum or average.

Decision flows are especially useful in customer care applications (e.g., e-commerce, call centers, insurance claims processing). Increasingly, these applications call for “segment-of-one marketing”, i.e., providing very individualized treatment to different customers [9]. Such treatments can cater to the individual tastes and preferences of customers, and can support targeted marketing initiatives and promotions by the host enterprise. This individualized treatment is important in connection with establishing and maintaining a loyal customer base, a cornerstone to success in business in general and e-commerce in particular [11]. A myriad of factors may be involved in differentiating between customers, including their status with the enterprise (e.g., green card, gold card), current “cases” they are involved with (e.g., home-mortgage application, application for insurance), the history of recent interactions with the enterprise, and their potential value to the enterprise. In many cases, the data is widely distributed across an enterprise, and multiple database queries are needed to process each customer contact. Since current e-commerce and customer care applications must support thousands or even millions of contacts per day, there is a tremendous need for optimization of this kind of decision making, in terms of both throughput and response time.

Decision flows were first introduced in [4] as part of a workflow model that permits the specification of workflow schemas supporting highly differentiated treatments. The current paper focuses directly on decision flows and their applicability in e-commerce applications. The paper also introduces algorithms for optimizing the execution of decision flows. Experimental performance analysis has verified the benefits of some of these algorithms.

The execution algorithms presented in this paper focus on minimizing response time and work load, by using parallel and eager (a.k.a. speculative) processing of decision flow tasks. Similar techniques have been applied in various areas, such as pipelined execution of machine level instructions in the field of computer architecture [8]. We develop here algorithms for detecting at runtime three key properties of tasks: *eligible* for eager evaluation, even if the output may not be used in the final output; *unneeded*, i.e., the task is not needed for completing the decision flow instance; and *necessary*, i.e., the task is definitely needed for completing the decision flow instance. Unlike the notion of eligible typical of many parallel processing paradigms, the notion here takes into account the semantics of the decision flow, and in particular, the enabling conditions on tasks. The notions of unneeded and necessary permit two important optimizations, by permitting the task scheduler to delete unneeded tasks from consideration, and to promote necessary tasks to the front of the queue. The framework established in this paper provides the basis for a future study of a broad variety of heuristic optimizations.

A prototype decision flow engine has been implemented that incorporates the algorithms for detecting eligible and unneeded tasks. Experiments verify that identifying the unneeded tasks can dramatically reduce both response time and work performed. We plan future experiments concerning the usefulness of determining necessary tasks.

Organization. §2 presents examples of decision flows, and defines the decision flow model. §3 describes an architecture for a decision flow engine. §4 presents algorithms for identifying eligible and unneeded tasks, and §5 presents the algorithm for identifying necessary tasks. §6 discusses the advantages of these algorithms and describes performance results concerning the unneeded algorithm. §7 offers brief conclusions. Due to space limitations the presentation here is rather terse. For more detail the reader may refer to [5].

Related Work. Decision flows can be used to support near-realtime decision making, and are richer than decision trees and traditional business rules frameworks. A problem with some expert systems, resulting from the possibility of arbitrary length chaining, is a “ripple” effect when individual rules are modified. Because decision flows provide more structure than expert systems, this ripple effect is substantially reduced.

Decision flows are complimentary to decision support and data mining systems. Those systems provide tools to analyze large volumes of data that chronicle previous business transactions, to help develop appropriate policies for future transactions. Decision flows can be used to implement those policies during execution of the future transactions.

Workflow systems such as Flowmark [7], Meteor [6], and others [12] specify work activities (for human agents or computers) using graphs whose nodes are tasks and edges corresponding to enabling conditions. Although decision flows can serve as the basis for a workflow model (see [4]), the current paper focuses primarily on the application of decision flows for near-realtime automated decision-making, where no human agents are involved. Many current workflow and related systems (e.g, [7, 6, 2, 12] can support such decision flows as part of a larger workflow. Alternatively, such workflow systems might use a decision flow engine as an adjunct that is invoked whenever a complex decision must be made.

The use of enabling conditions in decision flows is reminiscent of their use in the ThinkSheet model [10]. Two differences between the models is that decision flows can be executed in a fully automated mode, and decision flows give different handling for side-effect and non-side-effect tasks.

To the best of our knowledge there has been no work on optimizing workflow executions by varying the scheduling of tasks or by using speculative evaluation. Such optimization is possible with decision flows, because of their acyclic form and the use of speculative evaluation of non-side-effect tasks. Coordinating accesses to multiple databases was studied in Carnot [1, 13], but the focus is on scheduling policies in the execution model that ensures consistency rather than optimization.

2 An Abstract Model of Data-intensive Decision Flows

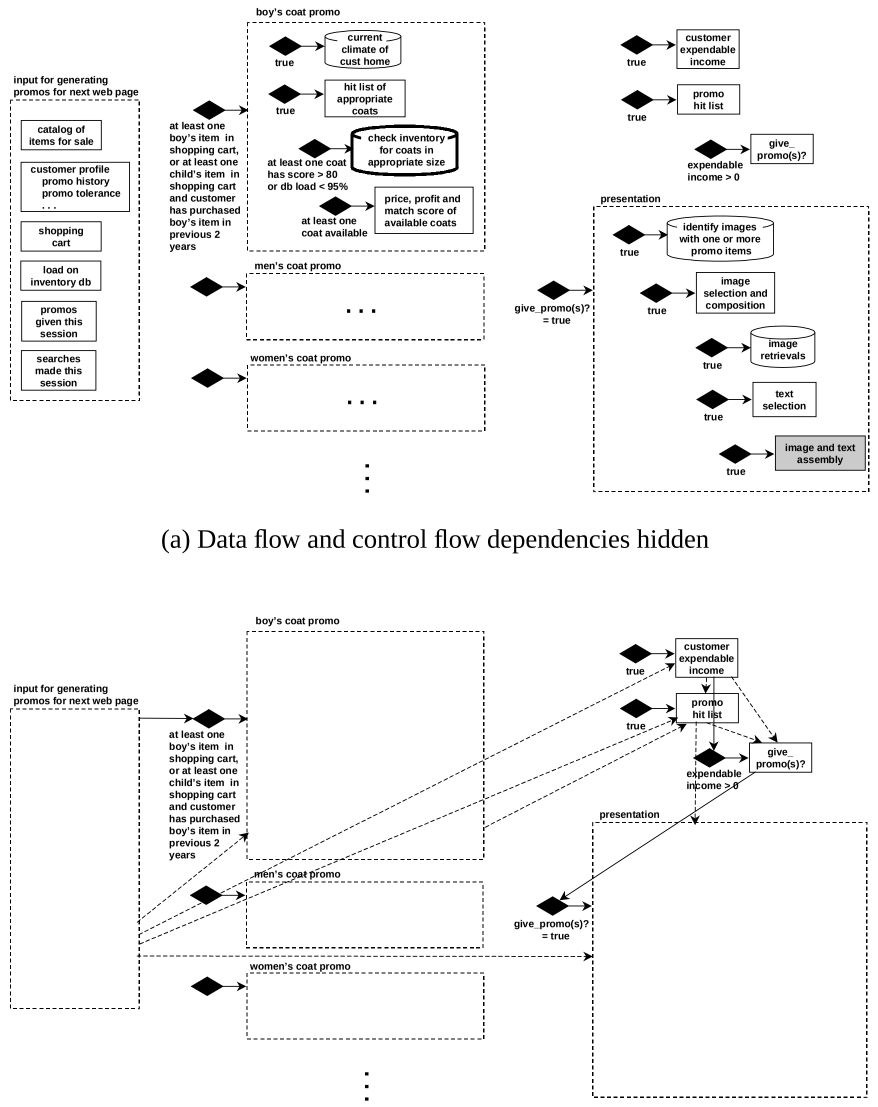
This section presents several examples that illustrate decision flows and their possible applications. The section also presents a formal definition of decision flows, that is used to describe the execution model and optimization algorithms developed in subsequent sections. This model is a subpart of the Vortex model [4], which was developed as a complete language for specifying workflows involving substantial decision-making activities.

We begin with an illustration of decision flows in moderate detail, and briefly describe additional decision flows.

Decision flow for selecting promos when generating web pages. Figure 1(a) shows part of a (simplified) decision flow that could be used to respond to customers interacting with the web-based store front of a hypothetical clothing catalog sales company. The decision flow focuses on selecting items that can be promoted, and might be executed each time a page is generated for a customer. Another decision flow that might be used for each page with a customer is deciding the level of service to provide (e.g., give faster service by moving the interaction to a less loaded web server). Other decision flows might be used when a customer first visits the store-front (e.g., to determine whether to give normal treatment or a collections treatment).

In Figure 1(a), each database and (solid boundary) rectangle corresponds to a *task* which might be performed for a given decision flow instance. Each task has the effect of producing a value for one or more attributes whose values may be used by other tasks of the instance ("intermediate" attributes) or returned as an output value of the instance ("target" attributes). The dashed rectangles (except for the far left one) indicate groupings of tasks into *modules*; this helps support scalability in the specification of decision flows.

The input attributes for this decision flow include the profile of the customer, the current value of the shopping cart, information about promos that the business is especially interested in moving, and information about the web session with the customer so far. Based on different enabling conditions (shown as diamond nodes) different categories of promotions will be considered by the decision flow. For example, if there is already one boy's item in the shopping cart, or if there is a child's item in the shopping cart and the customer has bought something for a boy in the past two years, then a promo for a boy's coat is considered. This involves doing a database dip to get information about the climate at the customer home, deriving a "hit list" of coats that might be appropriate to the customer, checking with inventory for coats in the appropriate size, and then creating a listing of possible coats to promo, along with info on the price, potential profit and degree of confidence that the promo matches customer interest.



input for generating promos for next web page
catalog of items for sale
customer profile promo history promo tolerance . . .
shopping cart
load on inventory db
promos given this session
searches made this session
at least one boy's item in shopping cart, or at least one child's item in shopping cart and customer has purchased boy's item in previous 2 years
boy's coat promo
true
current climate of cust home
true
hit list of appropriate coats
at least one coat has score > 80 or db load < 95%
check inventory for coats in appropriate size
at least one coat available
price, profit and match score of available coats
men's coat promo
. . .
women's coat promo
. . .
true
customer expendable income
true
promo hit list
expendable income > 0
give_ promo(s)?
give_promo(s)? = true
presentation
true
identify images with one or more promo items
true
image selection and composition
true
image retrievals
true
text selection
true
image and text assembly

(a) Data flow and control flow dependencies hidden

(b) Data flow (dashed arrow) and control flow (solid arrows) dependencies shown

Figure 1: Decision flow for selecting and generating promo images in web-based storefront

The decision module will estimate the customer expendable income (based on customer profile, shopping cart, and perhaps other factors), and create a listing of promos obtained so far. Based on the business value of the promos and the likelihood of their success, a decision is then made about whether to give promos or not.

Finally, if a promo will be given, the presentation module identifies images and text that can be used to display the promo(s), and assembles these for inclusion in the generated web page.

Attributes and tasks. A decision flow model is *attribute-centric*: the main objective of the execution is to determine the values of certain attributes, based on other given or derived attribute values. Decisions made by a decision flow are represented in the attribute values.

Attributes are computed in decision flows by two kinds of tasks. *Foreign tasks* are external to the decision flow execution engine (e.g., database queries, web server routines, questions to a human). These can produce one or more attribute value; for brevity here we assume that each produces a single attribute. *Synthesis tasks* produces a single attribute value, using a framework for aggregating and synthesized information based on combining policies. Although not a primary focus of the current paper we illustrate briefly illustrate this framework now.

The framework for specifying synthesis tasks is motivated largely by two factors: (a) there are many different ways to combine business factors, and (b) "business rules" provide a familiar and convenient mechanism for specifying policies. One way to specify a synthesis task is to specify a set of rules along with a *combining policy*. As a very simple example, an attribute business_priority might be computed by specifying rules, each of which contributes a value between 1 and 10. Example rules are: "if the customer is green card, then business_priority ← 4", and "if the customer bought $>$ \$100 in the last month, then business_priority ← 9". A simple combining policy would be to take the maximum value contributed by any rule with true condition. Another combining policy would be to take the average of such contributions. As a second example, suppose that a decision must be made between a small set of alternatives, e.g., to choose the agent skill needed to answer a particular phone call (e.g., sales, service, collections, high-tier). A set of rules can be written, each contributing a (value,weight) pair, e.g., (high-tier, 20). The combining policy could group the contributed pairs according to value, add up the contributed weights, and choose the skill with greatest weight.

Non-rules-based paradigms are also supported. For example, when developing a hit-list of promos, one could use a "weighted dot product" approach. Specifically, in the customer profile one might have data on the customer, stored in the form of a vector of (attribute, confidence level) pairs. For example, Joe might be described using [$<$ "has children", 70% $>$, $<$ "not fashion conscious", 80% $>$, ...]. Each item to be considered might have a vector indicating the importance of different attributes, e.g., for a heavy-duty raincoat the vector might be [$<$ "not fashion conscious", 90 $>$, $<$ "likes yellow", 40 $>$, ...]. A score for each promo relative to Joe can be obtained by doing a dot product (or join) operation on Joe's vector with the vector of the promo.

Data flow and enabling flow: The decision flow model presented to users is modular, to support scalability and levels of abstraction. Basically, a module consists of a set of tasks and submodules, all of which compute values of attributes, the module then specifies enabling conditions for these tasks and submodules. In turn the module can be used in specifying a higher level model.

Definition 2.1 A (*decision flow module*) is a 4-tuple $(Att, Cond, Source, Target)$ where

1. Att is a set of *attributes*. For each non-source attribute A there is a task or module which computes the value of A.
2. $Source$ and $Target$ are disjoint subsets of Att, corresponding to the *source* and *target* attributes, respectively. The target attributes are used outside of the current decision flow. In an execution of the decision flow, a value should be produced for each target attribute that is enabled (see below).
3. $Cond = \{C_m \mid m$ is a task or module computing a non-source attribute $\}$ is the set of enabling conditions, one for task or module used.

The flow of data and enabling in a decision flow is largely implicit. Figure 1(b) shows the data flow (using dashed arrows) and the enabling flow (using solid arrows) for the example decision flow. A data flow edge is included from task/module A to task/module B if an attribute specified by A is used as input for B. For example, the output of promo_hit_list is used as input for identifying images that show the promo items. A enabling flow edge is included from task A to task B if an attribute specified by A is used in the enabling condition for B. For example, customer expendable income is used in determining give_promo(s).

A module can be represented by its corresponding *dependency graph* that highlights the enabling and data dependencies between attributes. Nodes of the graph are Att. The dependency graph contains two kind of edges: *Data flow* edges of form (A, B) where the value of attribute A is used in the module for B and *Enabling flow* edges form (A, B) where attribute A is referred to in the enabling condition C_B for B. Figure 1(b) shows a variant of a Dependency Graph, where enabling conditions are also shown as nodes, and where flow modules are present.

A decision flow module S is *well-formed* if the dependency graph of S is acyclic. The acyclicity condition implies that attribute assignment is *monotonic*: if an attribute value is assigned, then it will never be overwritten. This is because during execution of a decision flow each module will be executed at most once.

Target and side-effect attributes. Intuitively, a target attribute is one that must be stable in order for execution of a decision flow instance to successfully complete. In the example the only target module is the one for image and text assembly (shown in gray). If this module is enabled, then execution will not complete until a value is obtained. If the module becomes disabled, then execution can halt immediately.

A *side-effect* attribute is one whose evaluation has a significant side-effect, such as performing a funds transfer, issuing a confirmation, or incurring a substantial cost (e.g., if a query against an external database has an associated charge.) In the example, the database query concerning coat inventory is a side-effect attribute (shown with bold outline). This might be because the inventory database is heavily loaded, or because the database will be updated to indicate that one or more coats has been "reserved" for this customer, in case they decide to purchase a coat that was promoted. The determination of side-effect vs. non-side-effect queries depends on the application.

During execution non-side-effect attributes might be evaluated speculatively, before their enabling condition is known to be true. This is prohibited for side-effect attributes.

Execution of decision flows. Before presenting the declarative semantics for decision flows we describe intuitively how they can be implemented. During execution, an attribute becomes *stable* if its enabling condition becomes true and the task specifying the attribute has executed and returned a value, or if its enabling condition becomes false, in which the attribute is assigned the value $\perp$, i.e., null value. (In a generalization of this model [4] we could distinguish exception values from other values, to explicitly model situations where a module is launched but unsuccessful.) A module can be executed after all of its input attributes have become stable.

Importantly, tasks in a decision flow must be capable of executing once their input attributes are stable, even if some of them have value $\perp$. This requirement is appropriate in many e-commerce applications, where a decision must be made on the basis of incomplete information. For example, even if the inventory database is unavailable, a decision must be made about whether to include a promo or not.

Declarative semantics of decision flows. In the abstract, during execution an attribute will have one of four states: UNINITIALIZED, ENABLED, VALUE or DISABLED. (Additional states are possible and described when specific execution details are involved; see §3 below.) Source attributes start with state VALUE. An attribute will become ENABLED if its enabling condition becomes true, and it will become DISABLED if its enabling condition becomes false. If ENABLED, an attribute will take a value and will then reach the VALUE state. If DISABLED, the attribute will take the null value $\perp$.

The semantics of decision flows is declarative, and defined using the notion of "complete snapshot".

Definition 2.2 A *complete snapshot* is a pair $s = (\sigma, \mu)$, where

(a) the state function σ maps each non-source attribute into {VALUE, DISABLED},

(b) the value function μ maps each non-source attribute A with state VALUE into the value returned by the module producing A (using the values given by μ for attributes occurring in the module) and maps each non-source attribute with state DISABLED into the null value $\perp$, and

(c) non-source attributes A is in state VALUE if the enabling condition C_A evaluates to true (using the values given for attributes occurring in C_A), and is in state DISABLED otherwise.

The acyclicity assumption guarantees that there is a unique complete snapshot for given source attribute values. An execution of a decision flow instance is *correct* if it produces states and values for the set of target attributes, and is compatible with the unique complete snapshot. (The states and values produced or not produced for other attributes are viewed as irrelevant.)

In this paper we we assume that for each given instance of the decision flow, the data needed by the database queries to compute the attribute values remains fixed during the processing of this decision flow instance. We view this assumption as reasonable for the kinds of decision flows that arising in e-commerce, that execute automatically in near-realtime. This assumption permits flexibility in the timing of launching queries and the use of speculative execution.

Snapshots and reporting. A crucial aspect of any production system used in business is the ability to understand how well it is achieving the objectives of the enterprise. The snapshot construct of decision flows provides a natural mechanism for reporting on how decisions are made. To illustrate, suppose that one is interested to see how well the example decision flow is functioning. A (possibly nested) relation can be created, where each row corresponds to a single execution of the decision flow. Each row will be constructed from the snapshot of the execution, for example, holding selected input attributes (e.g., customer profile, shopping cart,...), selected internal attributes (e.g., promo hit list, give_promos?,...), and the target attributes (in this case, perhaps the list of promo items presented to the user). The row might also include "outcome" attributes, such as whether the customer purchased any of the promoted items. (In some cases the outcome attributes might be available immediately; in other cases these might become available after some days or months have passed.) We shall call this relation a *snapshot history*.

A snapshot history can be used in a variety of ways. Manual and automated data mining techniques can be used to find patterns and relationships. These can be used in turn to determine that the specification of some synthesis tasks can be improved, and that other synthesis tasks have little impact (and can be eliminated). In some cases, the snapshot history could be used to support a feedback loop, that refines the decision flow on a daily basis.

Additional applications of decision flow technology Decision flows can be used to support near-realtime decision-making in a variety of e-commerce applications; we mention some additional examples here. The first example concerns dynamic pricing. For example, in the airlines industry pricing is now dependent on a variety of factors, including time until the flight and seat availability. Over time, customer relationship and other factors will come into the decision. Decision flows can provide a useful mechanism for specifying how the decisions should be made,

Increasingly e-commerce will be conducted by machines, with little human intervention. For example, to rent a car or buy a book, a consumer will own software that can automatically interact with vendor's software to check on availability and pricing, negotiate about different options, and potentially commit to a transaction. Decision flows are useful to both vendor and consumer in this context; we focus on the consumer here. How will the consumer give her preferences to the software? Using current technology, it will be easy for the consumer to indicate bounding parameters for deals (e.g., maximum price she is willing to pay). If decision flows are used inside the software, then the consumer can specify more detailed aspects of her preferences, e.g., the weights she gives to different aspects of a possible options, and intermediate attributes that can be used towards a final decision. Over time, the consumer could easily incorporate modify the weighting functions, and introduce new data sources and intermediate attributes.

3 An Execution Model for Parallel Processing of Decision Flows

In this section we present the Vortex execution model. The key feature of the Vortex execution model is its flexible scheduling of parallel executions of the tasks in the decision flow. With this model, we can incorporate many optimization strategies to reduce response time and/or work performed. These strategies include parallel scheduling, speculative execution and propagation of information. In this section we also introduce three specific optimization strategies useful in the model.

3.1 Decision flow execution schemas

Although the decision flow model presented to users is modular, at the execution level the flow modules are "flattened"; this permits more freedom with regards to the order of task execution, and especially speculative execution. To flatten a flow module M, we combine (with the "and" connective) the enabling condition for M with the enabling condition of each task and submodule within M. For example, for implementation the enabling condition for the boy's coat promo module will be "anded" into each of the enabling conditions for the four tasks inside. In the technical discussion below we focus on decision flows whose flow modules have been flattened.

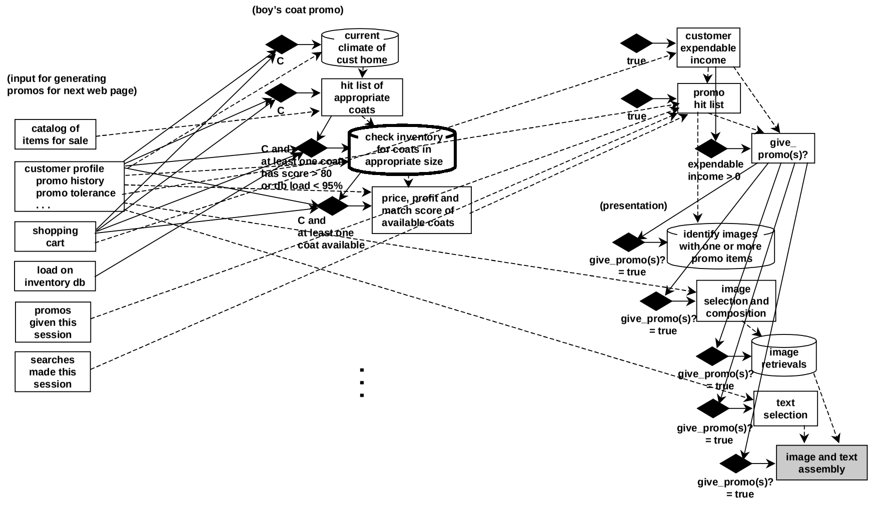

(boy's coat promo)
(input for generating promos for next web page)
catalog of items for sale
customer profile
promo history
promo tolerance
. . .
shopping cart
load on inventory db
promos given this session
searches made this session
C
current climate of cust home
C
hit list of appropriate coats
C and at least one coat has score > 80 or db load < 95%
check inventory for coats in appropriate size
C and at least one coat available
price, profit and match score of available coats
true
customer expendable income
true
promo hit list
expendable income > 0
give_promo(s)?
(presentation)
give_promo(s)? = true
identify images with one or more promo items
give_promo(s)? = true
image selection and composition
give_promo(s)? = true
image retrievals
give_promo(s)? = true
text selection
give_promo(s)? = true
image and text assembly
. . .

Figure 2: Data and control flow in the execution schema

The flow of data and enabling in the flattened decision flow is then obtained from the tasks and enabling conditions in the same way as in for the modular version. Figure 2 shows the data and enabling flow for the flattened version of the decision flow of Figure 1(a).

To provide grounding for the discussion of algorithms, we give a formal definition of a decision flow execution schema:

Definition 3.1 A *decision flow (execution) schema* is a 4-tuple $(Att, Cond, Source, Target)$ where

1. Att is a set of *attributes*. For each non-source attribute A there is a foreign or synthesis task which computes the value of A.
2. $Source$ and $Target$ are disjoint subsets of Att, corresponding to the *source* and *target* attributes, respectively. The target attributes are used outside of the decision flow. In an execution of the decision flow, a value should be produced for each target attribute that is enabled (see below).
3. $Cond = \{C_A \mid A$ is a non-source attribute $\}$ is the set of enabling conditions, one for each non-source attribute.

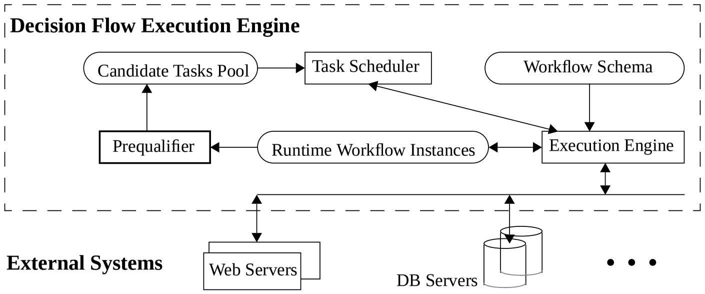
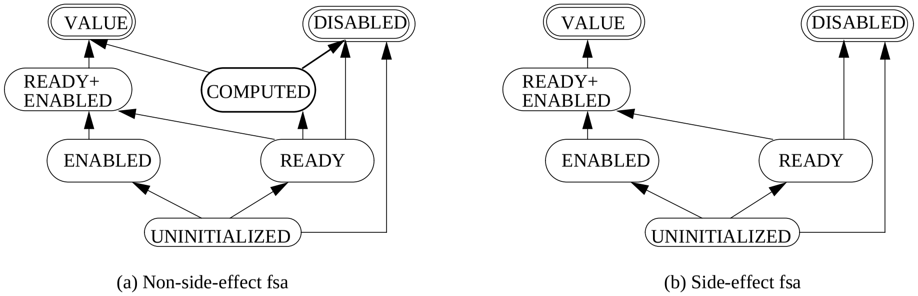


Figure 3: Architecture of the decision flow execution engine



(a) Non-side-effect fsa

(b) Side-effect fsa

Figure 4: Finite state automata for states of attributes

3.2 Parallel execution model

Before describing the execution model, we first give a brief description of the architecture of the decision flow execution engine (see Figure 3). The three round boxes represent data repositories. One contains *decision flow execution schemas*, and another contains *runtime instances* of the decision flows. Whenever a new case, e.g. a new set of promos for a web page needs to be generated, or a new insurance claim, comes in, a new instance of the decision flow is created. We explain the third round box later. The rectangles represent software modules. More specifically, the *execution engine* works on the decision flow instances to execute the tasks in the decision flow and propagate the effects of the executions until the goal is reached. The engine works in a multi-thread fashion, so that parallel processing of multiple decision flow instances, and multiple tasks within one instance is possible. To execute the tasks, the engine consults the *task scheduler* that dynamically chooses one or more tasks from a pool of candidate tasks, i.e., the round box *candidate tasks pool*. The candidate pool is maintained by the *prequalifier*. The main focus of this paper is about novel optimization strategies developed for the prequalifier.

To simplify the exposition, we assume for the remainder of the paper that each decision flow task involves the evaluation of a single attribute. It is straightforward to generalize the algorithms described here to apply to contexts where tasks produce more than one attribute value. Our assumption means that we can identify each task t by the attribute that A_t that it produces. Further, we interchangeably refer to execution of a task t or evaluation of the corresponding attribute A_t.

We now give a sketch of the *execution algorithm*, which summarizes the three important phases of executing decision flows. This algorithm is based on a generalized notion of snapshot, which is described shortly. The execution program is invoked each time a new decision flow instance is initiated, and each time new values of attributes are obtained for a running decision flow instance. We describe now one step of the algorithm. Assuming that a decision flow instance has already been initiated, the algorithm performs the following steps when a new attribute value becomes available.

(1) `Evaluation` phase:

  (a) Construct a new snapshot that incorporates the new attribute value(s).
  (b) If a terminal snapshot (i.e., all the target attributes are stable) is reached, then exit.

(2) Prequalifying phase (*prequalifier*): Identify a set of *candidate attributes* in the decision flow that are ready to be evaluated.

(3) Scheduling phase (*scheduler*): Select one or more attributes out of the candidate attribute set based on scheduling heuristics, and send their corresponding tasks to the external DB server(s).

The execution algorithm constructs a series of snapshots, each one incorporating newly acquired information obtained through the evaluation of attributes. We now describe the extended form of snapshots used. As in §2, the extended snapshots will be ordered pairs of form (σ, μ). However, the set of possible states for attributes is expanded, as indicated in Figure 4. Part (a) of that figure shows the non-side-effect fsa, which is used for non-side-effect attributes. Before describing these fsa's, we stress that these serve a different purpose than the state diagrams found in the Carnot [1] and Meteor [6] workflow models. In those models, the task states are visible to (testable by) conditions in the workflow schema. Here the states are used primarily for optimization purposes, and only states VALUE and DISABLED are testable by conditions in the decision flow schema.

The intuitive meaning of an attribute A being in a state of the fsa is now given. States UNINITIALIZED, ENABLED, VALUE and DISABLED retain the meaning from §2. The states VALUE and DISABLED are shown with double circles because they are terminal states for attributes. An attribute A can move into state ENABLED (DISABLED) if, based on information accumulated so far, the enabling condition for A is determined to have value true (false). The state READY indicates that all of the input attributes for an attribute have stabilized (i.e., their states are DISABLED or VALUE). If an attribute is in state READY, then it can be evaluated speculatively. State READY+ENABLED indicates both that the input attributes are stable and the enabling condition for an attribute has been determined to be true. State COMPUTED (and not enabled), indicates that the value for A has been computed speculatively but the truth value of the enabling condition is not yet determined.

The state COMPUTED is used when non-side-effect attributes are computed before the truth value of their enabling conditions are known. The READY state is useful to the prequalifier, because then it knows that the attribute can be executed and may choose to inform scheduler that the attribute is eligible for execution. The scheduler can then decide to launch the execution of the attribute eagerly (before knowing the result of the evaluation of its enabling condition).

The side-effect fsa is shown in Figure 4(b). The only difference is that for side-effect attributes, the COMPUTED state is omitted. This is because side-effect attributes should not be computed until their enabling conditions are known to be true.

There is a natural partial ordering on the states of the fsa. For example, we write READY $<$ COMPUTED and READY $<$ VALUE.

An execution permitted by the execution algorithm can be described by a sequence (s_0, V_0=$\emptyset$,C_0=$\emptyset$), $\cdots$, (s_i,V_i,C_i), $\cdots$ (s_n,V_n,C_n) where s_0 is the initial snapshot (where all sources attributes are READY+ENABLED, and s_i, $i \in [1, n]$, are snapshots computed by the execution algorithm (step (1) of the algorithm). V_i, $i \in [1, n-1]$ are non-empty sets of pairs of the form (A, v_A) where A is an attribute and v_A is the value for A returned to the evaluation engine C_i, $i \in [1, n]$, contains the attributes selected for computation since the beginning of the execution. Intuitively, there is a transition from (s_i, V_i, C_i) to $(s_{i+1}, V_{i+1}, C_{i+1})$ when new attribute values has been returned to the execution engine. These new values are those in $V_{i+1} - V_i$. These new values are then used to compute the new snapshot s_{i+1} from s_i by applying the evaluation phase (step 1 of the execution algorithm). The obtained snapshot value s_{i+1} is used to determine the new attributes to be launched by the execution by the prequalifying phase and the scheduling phase (step 2 and 3). These newly launched attributes are put in C_{i+1}.

3.3 Optimization strategies

We now sketch the optimizations for the evaluation and prequalifying phases of the execution algorithm that we present in this paper. Many kinds of scheduler have been developed for parallel processing. For instance, the *speculative execution* technique relies on heuristics to choose tasks from a pool of candidate ones for (parallel) execution. Since most of the scheduling algorithms rely on heuristics, a good prequalifier can improve the results of these algorithms by maintaining a "good" pool of candidates using the information about the execution accumulated so far. This means the pool should have the following three qualities: (1) identify *eligible* tasks and add them to the pool as soon as possible, so that certain eager evaluations (precomputation) can be performed; (2) minimize the number of *unneeded* tasks in the pool, so that the scheduler does not waste resources for performing these tasks; (3) identify *necessary* tasks in the pool, so that the scheduler can always schedule these tasks if there are doubts about the usefulness of the other tasks. This will reduce the randomness associated with some of the existing schedulers, such as those used for speculative execution. Although we develop our optimization strategies in the context of the Vortex execution model, we feel that some of the principles can be generalized to a other areas.

We use the example decision flow schema shown in Figure 1 in §2 to show how the three qualities of a candidates task pool could be achieved. More detailed discussion about the algorithms are in the following sections.

(1) **Eligible tasks.** Consider the enabling condition for the hit list of appropriate coat (boy's coat promo). If the shopping cart includes a child's item, then this condition is true, regardless of the customer previous purchases. As a result, execution of the task is enabled, and an access to the customer previous purchases does not have to be performed.

(2) **Unneeded tasks.** Suppose that the customer expendable income is computed and has value 0. Then the attribute give_promos? is disabled. In this case the tasks in the presentation module, and the target attribute, will be disabled. Through a chain of inference based on backwards propagation, it can be determined that tasks in the boy's coat promo (and all other promos) do not need to be executed for successful completion of the decision flow.

(3) **Necessary tasks.** Assume now that the expendable income is bigger than 0, and so give_promos? is enabled. The value of this attribute is needed to determine whether or not the target attribute will be enabled. Further, give_promos? will need the value of promo_hit_list, and this is enabled, so promo_hit_list is necessary. Suppose now that the enabling condition for boy's coat promo true, and that the enabling condition for the database dip in that module is true. It can be inferred that the database dip is necessary.

4 Eager Detection of Eligible and Unneeded Tasks

In this section, we describe the Eager_and_Unneeded Algorithm that supports detection of eligible and unneeded tasks. For eligibility, the algorithm computes the

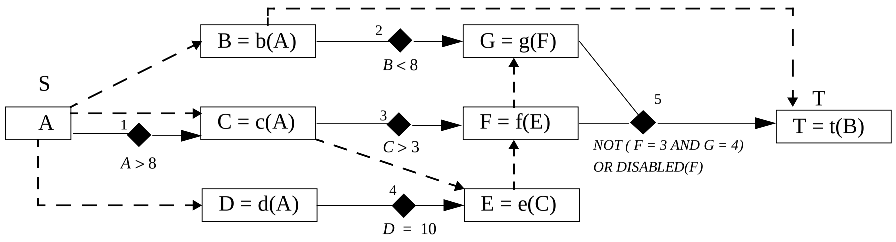


Figure 5: Dependency graph example for illustrating optimization algorithms

values of enabling conditions eagerly, which sometimes permits the prequalifier to determine in advance that attributes will eventually move into the states ENABLED, READY, DISABLED or READY+ENABLED. For unneeded, the algorithm uses a generalization of Dependency Graph of a decision flow schema and a combination of forward and backward propagation.

Importantly, the cumulative running time of the Eager_and_Unneeded Algorithm for the entire execution of a decision flow is linear in the size of the decision flow. Thus, for each execution step it is at most linear, and is typically sub-linear. Moreover, the upper bound on the total cost of the algorithm is independent from the number of steps taken by the execution engin to reach the terminal snapshot. This means that changing scheduling heuristics (in the scheduling phase) will not affect the global performance of the prequalifier phase.

To illustrate our algorithms we are using the following "abstract" example.

Example 4.1 Consider the Vortex workflow schema whose dependency graph is given in Figure 5. The attribute A is the source attribute. At the beginning of the execution all the attributes have the state UNINITIALIZED. When a value for A enters into the system, the execution algorithm can start its first cycle to determine the tasks to execute. Depending on the value of A our optimization algorithm produces different information. For example if A has the value 10, the algorithm detects that the states of the attributes B, C, and D are READY+ENABLED and thus eligible for execution. However no information about the new states of the other attributes and no unneeded attributes are discovered. On the other hand, if A has a value less than 8, say 0, our optimization algorithm discovers much more information. First C and F are inferred to be DISABLED, and T is inferred to be ENABLED. Moreover, B is inferred to be READY+ENABLED (and thus eligible for execution) and D , E and G are inferred to be unneeded. (Example 4.6 below details the reasoning behind these inferences.) This information permits the scheduler to launch the execution of B and to save processing by avoiding the execution of D , E, G, F, and C. ∎

This section focuses on the core components of the Eager_and_Unneeded algorithm, namely a generalization of the Dependency Graph for decision flows and rules that propagate information along that graph. A detailed specification of the algorithm may be found in [5].

In the first part of this section we give a precise syntax and semantics for enabling conditions in decision flows. Next, we describe a data structure and propagation rules for eager evaluation of enabling conditions. Then we present the generalized Dependency Graph and propagation rules for eligibility. The section concludes with rules for unneeded.

4.1 Syntax and semantics for enabling conditions

We present here a specific syntax and semantics for enabling conditions in decision flows. While the algorithms presented below depend in part on the syntax and semantics chosen here, the algorithms can easily be adapted to work with different choices for the syntax and semantics.

Recall that we work with "flattened" decision flows, where each task specifies a single attribute. The syntax for enabling conditions permits access to the values of attributes, and to their (stable) states, (i.e., whether they are VALUE or DISABLED).

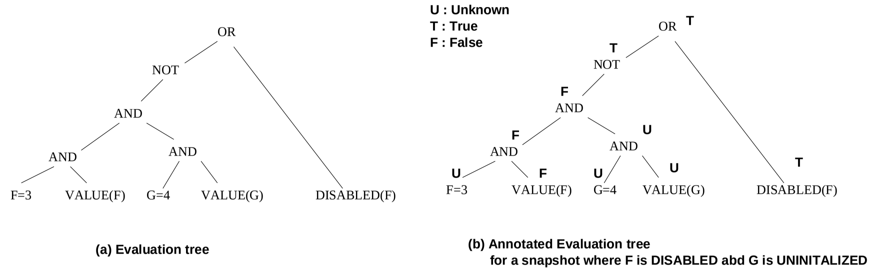
Conditions are built from atoms. The atoms VALUE(A) and DISABLED(A) can be used to test the state of attribute A. (Conditions cannot test for state UNINITIALIZED.) Predicate atoms are boolean functions of form $pred(t_1, \ldots, t_n)$, where each t_i is a term. Terms are built up from constants and attribute names (attribute name A in a term is interpreted as the value $\mu(A)$, if that is defined) Terms can also use arithmetic operators, etc.; the development here is independent of the particular operators used, so we leave that unspecified.

The truth value of conditions, when the involved attributes are stable, is given by the standard two-valued logic with the following exception: $pred(t_1, ..., t_k)$ is true if (a) each attribute A referred to by $pred(t_1, ..., t_k)$ is in state VALUE, and (b) $pred(t_1, ..., t_k)$ is true in the standard sense; the predicate is false otherwise. (This logic does not always behave in the standard way, e.g., $A > 3 \vee A \leq 3$ is not a tautology.)

4.2 Rules for eager evaluation of enabling conditions

To compute eagerly eligible and unneeded attributes, our algorithm combines the technique of eager evaluation of enabling conditions with the propagation of information about attribute states along the dependency graph.

To represent partial computations of enabling conditions we use a three valued logic (`False`, `Unknown`, `True`). Furthermore, instead of considering each condition as one indivisible three-valued logic formula, we represent it by its evaluation tree, so that we can perform short circuit evaluation. The motivation for this is to have a more precise knowledge of what sub-formula is true and what sub-formula is



(a) Evaluation tree

(b) Annotated Evaluation tree for a snapshot where F is DISABLED abd G is UNINITALIZED

Figure 6: Evaluation tree for node 5

false. Such information will be very useful to discover unneeded attributes.

Definition 4.2 The *evaluation tree* of an enabling condition P is obtained from the syntax tree of P by replacing each leaf node p of the form $pred(t_1, \cdots, t_k)$ with a tree $T(p)$ defined as: The root node of $T(p)$ is an AND operator node; $pred(t_1, \cdots, t_k)$ is a leaf node of $T(p)$; VALUE(A) is a leaf node of $T(p)$ if A occurs in $pred(t_1, \cdots, t_k)$; All the leaf nodes are directly connected to the root node.

The reason for the replacement is to take into account the specific semantics for $pred(t_1, \cdots, t_k)$ explained in §4.1. In the following, we call the leaf nodes of the tree *predicate nodes*, and the other nodes *operator nodes*. Figure 6(a) shows the evaluation tree for node of the workflow schema in Figure 5. To represent the partial evaluation of a condition we annotate each node of the tree by its value in {`False`, `Unknown`, `True`}. The value of a given node is computed by applying the three valued logic on the values of its predecessors in the tree using the following propagation rules (at the beginning of the execution all the values are $unknown$): Let s be a snapshot and V a set of attribute values that are computed so far, and let T be an evaluation tree of an enabling condition. We have the following propagation rules.

(4.1) If p is a predicate node of the form $pred(t_1, \cdots, t_k)$ and all the attributes involved in p are in V, then the value of the predicate node is equal to its evaluation of p (according to the two valued logic) on the values of the attributes in V.

(4.2) If p is a predicate of the form VALUE(A) and $\sigma(A) \geq$ ENABLED in s, then the value of p is *true*. The value of p becomes *false* if $\sigma(A) =$ DISABLED.

(4.3) If p is a predicate of the form DISABLED(A) and $\sigma(A) \geq$ ENABLED in s, then the value of p is *false*. The value of p becomes *true* if $\sigma(A) =$ DISABLED.

(4.4) The value of an AND node is the minimal value of its direct predecessors in the tree.

(4.5) The value of an OR node is the maximal value of its direct predecessors in the tree.

(4.6) The value of a NOT node is *true* if the value of its predecessor is *false*, *false* if the value of its direct predecessor is *true*, and *unknown* otherwise.

Example 4.3 Figure 6(b) shows the annotated evaluation tree for the enabling condition of T (node 5) for a snapshot where the state of F is DISABLED and G is UNINITIALIZED. Since F is DISABLED, the value of VALUE(F) is $false$ (rule 4.2), the value of DISABLED(F) is true (rule 4.3), and the value of $F = 3$ remains unknown (rule 4.1). The value $false$ is propagated along the AND nodes using propagation rule 4.4. Then using rule 4.6 we can conclude that the NOT node has the value $true$. Finally using rule 5 we can determine the value of the root node to be $true$. ∎

A nice property of the three-valued evaluation of the evaluation tree is the fact that once a value is discovered to be either *true* or *false* for a node, this value will never change. We say that this value is *stable*.

4.3 Rules for eligibility

We now develop rules for inferring eligibility of attributes. To this end, we use the Figure 7, which incorporates evaluation trees into Dependency Graphs.

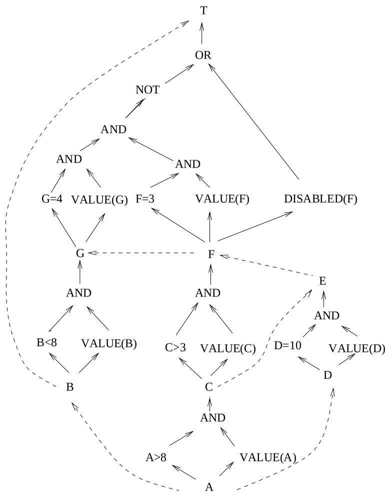


Figure 7: Extended Dependency Graph

Definition 4.4 Let $\mathcal{S} = (\mathcal{A}, S, T, \mathcal{F}, \mathcal{M})$ be a workflow schema. The *extended dependency graph* of $\mathcal{S}$, denoted $D_{\mathcal{S}}$, is a directed acyclic graph that is constructed as follows:

- Each enabling condition in the decision flow schema is represented by its evaluation tree in $D_{\mathcal{S}}$.
- Each attribute in $\mathcal{A}$ is a node in $D_{\mathcal{S}}$.
- There is an edge from the root node of each enabling condition evaluation tree to the attribute node attached to this enabling condition in $\mathcal{S}$.
- There is an edge from an attribute A to a predicate node p iff A occurs in p.
- There is an edge from an attribute A to an attribute B iff A is an input attribute of B.

In $\mathcal{S}$, a node is a *condition node* if it belongs to an evaluation tree, and is an *attribute node* otherwise. An edge between attributes is a *data* (*flow*) *edge*, and each other edge is a *control* (*flow*) *edge*.

For an attribute A, $Enabling\_Root(A)$ is the root of the enabling condition of A in the extended dependency graph.

Figure 7 shows the extended dependency for the decision flow depicted in Figure 5. Control edges are shown as solid arrows, and data edges are shown as dashed arrows.

We now illustrate how information can be propagated from and to evaluation trees occurring within extended dependency graphs.

Example 4.5 Recall the decision flow of Example 4.1, and suppose that the input value of A is 0. (a) To illustrate forward propagation, consider again attribute F. Since A has value 0, the enabling condition of C is $false$ and C is DISABLED. Hence the enabling condition of F is $false$, F is DISABLED, finally by applying partial evaluation on the precondition of T we can conclude that T is ENABLED. ■

We now present the propagation rules for state values. Taken individually, these rules are not deterministic. But in combination, they will determine a specific state value for each affected attribute.

Let s be a snapshot and V be a set of attribute values that are computed so far, and let D_S be the extended dependency graph of a decision flow we have the following propagation rules.

(4.7) If root of enabling condition for A (i.e., the predecessor of A in D_S through a control flow edge) is *false*, then $\sigma(A)$= DISABLED

(4.8) If (A, V_A) is in V, then $\sigma(A)$ $\geq$COMPUTED (For side-effect A, this means that $\sigma(A) \in$ {VALUE, DISABLED}).

(4.9) If root of enabling condition for A is true and (A, v_A) is in V, then $\sigma(A) =$ VALUE

(4.10) If each data input for non-side-effect attribute A (i.e., each predecessor of A in D_S through a data-edge) is in state VALUE or DISABLED, then $\sigma(A)$ $\geq$READY.

(4.11) If each data input for side-effect attribute A is in state VALUE or DISABLED, and if enabling condition for A is true, then $\sigma(A) \geq$ READY+ENABLED

4.4 Rules for unneeded

We now describe how unneeded attributes are detected. This may arise from forward propagation of information, because attributes are discovered DISABLED, from backward propagation, or from a combination of the two.

Example 4.6 Recall the decision flow of Example 4.1, and suppose that the input value of A is 0. (i) To illustrate forward propagation, consider again attribute F. As showed in 4.5 F is discovered DISABLED and and so F does not need to be computed. (ii) Backward propagation is more complex. This involves inferring that although an attribute is or may become enabled, its value is not needed for successful completion of the decision flow instance. For example, suppose again that the input value of A is 0. Since attributes E is only used to compute the value of F and F is DISABLED the value of E is unneeded. Recursively, since D is only used to compute E, D is also unneeded. (iii) To illustrate a combination of forward and backward propagation, recall that the enabling condition of T is already (eagerly) evaluated, without using the value of G, Since the only use of G is to evaluate this condition, G is unneeded. ■

Now we give the (backward) propagation rules for discovering unneeded attributes: Let s be a snapshot and let D_S be the extended dependency graph of a decision flow.

(4.12) If $\sigma(A)$ = DISABLED, then A is unneeded.

(4.13) If n is a condition or an attribute node in D_S and all the successors of n in D_S are either unneeded or stable then n is unneeded. Recall that an attribute is stable if $\sigma(A)$ $\in${ VALUE, DISABLED} and a condition node is stable if its values is in { *true*, *false* }.

We summarize by stating the following result concerning the complexity of detecting eligible and unneeded tasks (see [5] for the proof).

Proposition 4.7 There exists an algorithm that can compute eager snapshots and unneeded attributes, whose cumulative complexity for the entire execution is linear with the size of the workflow schema. More precisely, the running time of this algorithm is (at worst) $O(P) + O(E)$ for the entire execution, where P is the number of predicates of the form $pred(t_1, \cdots, t_k)$ and E is the number of edges in the extended dependency graph.

5 Eager Detection of Necessary Tasks

In this section we concentrate on detecting necessary attributes at runtime. We describe the core of an algorithm that dynamically computes at each step of the execution a set of attributes inferred to be necessary. The algorithm uses as input the information contained in the extended dependency graph, along with the rules of the previous section for detecting eligible and unneeded attributes. The algorithm considers each node A with states READY or READY+ENABLED and propagates some "necessary" properties from A to the top of the graph. If a target attribute is reached then the algorithm concludes that A is necessary. The cumulative cost of the algorithm for the complete execution of a workflow instance is quadratic and independent from the number of execution steps needed.

The section begins with a motivating example. Next, §5.1 introduces four kinds of "necessary property". §5.2 presents a family of rules for correctly propagating these properties along an extended dependency graph.

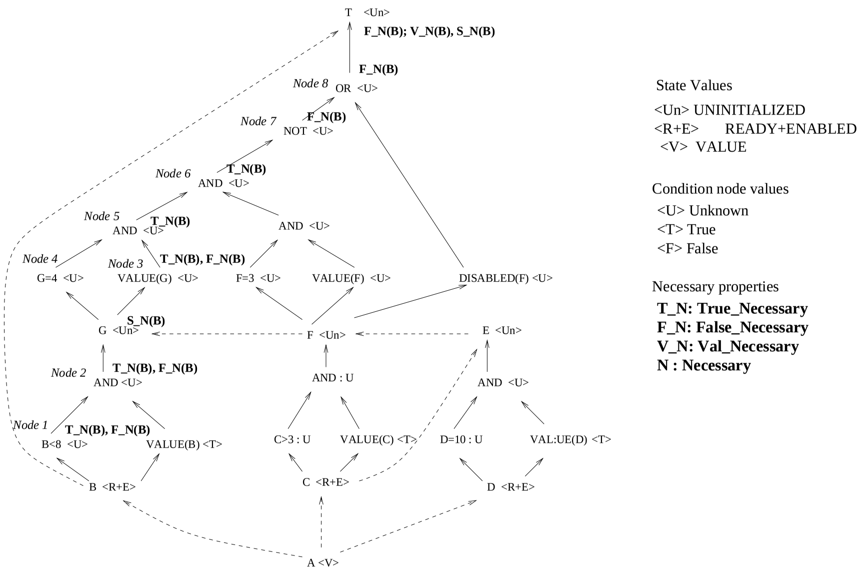


Figure 8: Propagations of "necessary properties" along the dependency graph

Example 5.1 Let us consider again the workflow depicted in Figure 5 and assume now that the input attribute A has the value 10. Figure 8 shows an extended dependency graph with various annotations. The non-bold annotations correspond to the result of the execution of the Eager_plus_Unneeded algorithm where A has value 10. Each condition node is annotated by its value in the eager snapshot and each attribute is annotated by its state in the eager snapshot. The graph shows that the attributes B, C, and D areREADY+ENABLED and thus eligible for execution.

The bold annotations of Figure 8 indicate necessary properties of various nodes relative to attribute B. The complete sequence of inferences used to obtain these annotations is presented in Example 5.4, and we present a brief hint of that reasoning now. The attribute B is determined to be "Value_necessary" for target attribute T, because B is enabled, and it is used as a data input when computing the value for T. The attribute B is also determined to be "False_necessary" for T, because a value for B must be used in any execution sequence where the enabling condition for T turns out to be false. Thus, regardless of whether T is ultimately enabled or disabled, a value for B will be used. We conclude that B is "Stable_necessary" for T. Since T is a target attribute, we conclude that B is *necessary*.

The fact that B is necessary can be used by the scheduler. For example, the scheduler might choose to evaluate B as soon as possible, since C and D may become unneeded at a later point in time. ∎

5.1 Necessary properties

Our approach to detect necessary attributes is to propagate properties that might make an ENABLED or READY+ENABLED node n necessary, proceeding "bottom-up" from n to the target attributes of the graph. (A "top-down" strategy would lead to a greedy algorithm that will be much more expensive.) Let us first explain why our algorithm considers only ENABLED or READY+ENABLED nodes. In fact, only such nodes can be detected to be necessary at the current step of the execution. Indeed, if a node A is not ENABLED in an eager snapshot it means that the result of its enabling condition is uncertain and can become false, making the value of A unneeded. Thus, being at least ENABLED is a necessary condition.

In this subsection we first define the necessary properties we want to propagate. Then we propose inference rules that are sufficient conditions to propagate these properties along a reduced dependency graph. These rules form the core of our algorithm to detect necessary tasks. There are four properties that are useful to discover necessary attributes: "True_necessary", "False_necessary", "Value_necessary" and "Stable_necessary".

True_necessary and False_necessary properties give information about the necessary relationship between an attribute and a predicate. Intuitively, A is True_necessary (False_necessary) for a condition node p if the value of A is needed to make the value of p true (false).

Definition 5.2 (*False_necessary and True_necessary*) Let D be a dependency graph, s be an eager snapshot. An attribute A is *True_necessary* (*False_necessary*) for an evaluation node p, if it is impossible to obtain an eager snapshot s' from s (by applying the execution algorithm) where the value of p is true (resp. false), and the state of A is not in {VALUE, COMPUTED}.

Value_necessary and Stable_necessary properties give information about the necessary relationship between two attributes. Intuitively, A is Value_necessary for an attribute node B if the value of A is needed to obtain a value for B. It is Stable_necessary for B if the value of A is needed to obtain a stable state for B.

Definition 5.3 Let D be a dependency graph and s be an eager snapshot. An attribute A is *Value_necessary* for an attribute node B, it is impossible to obtain an eager snapshot s' from s where the state of B is in $\{$VALUE, COMPUTED$\}$, and the state of A is not in $\{$VALUE, COMPUTED$\}$. A is *Stable_necessary* for B if it is impossible to obtain an eager snapshot s' from s where the state of B is in $\{$VALUE, DISABLED$\}$, and the state of A is not in $\{$VALUE, COMPUTED$\}$.

5.2 Rules for detecting necessary tasks

We now present a set of rules for propagating necessary information along the extended dependency graph. The rules are grouped into three sets.
Sufficient propagation conditions for True_necessary and False_necessary properties:
Let s be an eager snapshot and D_S the extended dependency graph for s. Let A be a attribute node and p a predicate node in D_S. A is False_necessary and/or True_necessary for p, if $\sigma(A) \geq$ ENABLED, p has a value $unknown$ in s and p verifies the following recursive conditions.

(5.1) Suppose p is an OR node. Then A is True_necessary for p, if A is True_necessary for all the direct predecessors of p in D_S whose value is $unknown$ in s. It is False_necessary for p, if A is False_necessary for at least one direct predecessor of p in D_S whose value is $unknown$ in s.

(5.2) Suppose p is an AND node. Then A is True_necessary for p, if A is True_necessary for at least one direct predecessor of p in D_S whose value is $unknown$ in s. It is False_necessary for p, if A is False_necessary for all direct predecessors of p in D_S whose value is $unknown$ in s.

(5.3) Suppose p is a NOT node. Then A is True_necessary for p, if A is False_necessary for the direct predecessor of p. It is False_necessary for p, if A is True_necessary for the direct predecessor of p.

(5.4) Suppose p is a VAL(B) predicate. Then A is True_necessary (resp. False_necessary) for p, if A is True_necessary (resp. False_necessary) for $Enabling\_Root(B)$.

(5.5) Suppose p is a DISABLED(B) predicate. Then A is True_necessary (resp. False_necessary) for p, if A is False_necessary (resp. True_necessary) for the $Enabling\_Root(B)$.

(5.6) Suppose p is a predicate of the form $pred(t_1, \cdots, t_k)$. Then A is True_necessary and False_necessary for p, if it is Value_necessary for at least one direct predecessor of p.

Sufficient propagation conditions for Stable_necessary property:
Let s be a snapshot and D_S the extended dependency graph for s. Let A and B be two attribute nodes in D_S. A is Stable_necessary for B, if $\sigma(A) \geq$ ENABLED , B is not unneeded in s and one of the three conditions holds:

(5.7) B is enabled in s and A is Value_necessary for B.

(5.8) B is not enabled in s and one of these conditions is true: (a) A is Value_necessary for B and False_necessary for $Enabling\_Node(B)$, or (b) A is both True_necessary and False_necessary for the $Enabling\_Node(B)$.

The rationale for these sufficient conditions is the following. For rule (5.7), if the attribute node B is already enabled in s it means that its value has to be computed to reach a stable state. Thus A must be Value_necessary for B. Rule (5.8) deals with the case where B is not enabled. In this case, B can reach the DISABLED stable state or the VALUE stable state. In rule (5.8a) the fact that A is False_necessary for the enabling node of B implies that the value of A is necessary to make A DISABLED. The fact that A is also Value_necessary for B implies that A is needed to obtain a stable value for B. rule (5.8b) exploits the fact that to have the VALUE state B needs to have its enabling condition true. Thus, the fact that A is True_necessary for the enabling node of B guarantees that A is necessary to make B reach the VALUE state.
Sufficient conditions for Value_necessary property:
Let s be an eager snapshot and D_S the extended dependency graph for s. Let A and B be two attribute nodes in D_S. A is Value_necessary for B, if $\sigma(A) \geq$ ENABLED, B is not unneeded and one of the following two conditions holds:

(5.9) A is an input attribute of B, or,

(5.10) A is Stable_necessary for at least one of the input attributes of B.

Items (5.9) and (5.10) rely on the fact that the computation of B needs the stable values of all its input attributes.

Example 5.4 The bold annotations of Figure 8 show how the necessary properties are propagated when the rules are applied on the attributes B and C. Let us show how the rules detect that B is Stable_necessary for T. In the figure, the mark `T_N(B)` (resp. `F_N(B)`) represents the True_necessary property of B (resp. False_necessary property of B), `V_N(B)` the Value_necessary property of B and `S_N(B)` the Stable_necessary property of B. First B is both False_necessary and True_necessary for node 1 according to the propagation rule (5.6). These properties are propagated to node 2 by applying the rule (5.2). Rule (5.8b) can be used to infer that B is Stable_necessary for attribute G. Applying rule (5.4) in connection with node 2 propagates the True_ and False_necessary properties to node 3. Unfortunately there is no rule to propagates these properties to node 4. From node 3, rule (5.2) infers the True_necessary property for node 5. The False_necessary property cannot be propagated by rule (5.2), since the node 4 does not have the False_necessary property. Then applying rule (5.2) again and rule (5.3) we can conclude that B is False_necessary for node 8. Moreover applying rule (5.9) makes B Value_necessary for T. Thus by rule (5.8a) we can conclude that B is Stable_necessary for T. ∎

We summarize by stating the following result concerning the complexity of detecting necessary tasks (see [5] for the proof).

Proposition 5.5 There exists an algorithm that can compute necessary attributes, whose cumulative complexity for the entire execution is quadratic with the size of the workflow schema. More precisely, the running time of this algorithm is (at worst) $O(P)+O(E)+O(N^2)$ for the entire execution, where P is the number of predicates of the form $pred(t_1, \cdots, t_k)$, E is the number of edges in the extended dependency graph, and N is the number of edges in the (non-extended) dependency graph.

6 Impact of the Optimization Algorithms

In §4 and §5 we introduced algorithms to eagerly identify eligible, unneeded, and necessary attributes. In this section we analyze the impact of these algorithms on achieving the two optimization goals: minimizing *response time* and *work load*. Work load is defined as the total number of units of processing being performed for each decision flow instance. It is clear that the impact of the algorithms depends on the characteristics of the decision flows. We show how the characteristics of the decision flows may affect the effectiveness of the algorithms. We analyze the algorithms in turn.

We first consider eligible attributes. For a given attribute, we want to make it eligible for evaluation as early as possible. This can improve the performance in the following ways: If tasks can be executed in parallel, more eligible attributes at an early point of time means more tasks can be executed early in parallel. That results in shorter response time. If there is no parallel task execution, more eligible tasks early means that the task scheduler has more freedom in employing heuristics to choose tasks for execution. This may lead to better response time and less work load. In fact, this was confirmed in experiments we reported in reference [3], where some scheduling heuristics resulted in up to 50% performance gain over simple first-in-first-serve scheduling when tasks can be executed in parallel.

When comparing our algorithm with a naive algorithm in terms of identifying eligible attributes early, we consider attributes with side-effect and non-side-effect evaluation tasks separately. In both cases a naive algorithm identifys an attribute to be eligible when its state is READY+ENABLED. If an attribute is evaluated by a task with side-effect, our algorithm makes the attribute eligible when its state is READY+ENABLED, just like the naive algorithm. The difference is that our algorithm performs eager evaluation of the enabling conditions, i.e., we perform short circuit style evaluation whenever possible without waiting for all the input attributes to become stable. This technique make bigger gains if the enabling condition is conjunction or disjunction of larger number predicates. A non-side-effect attribute becomes eligible if its state is READY. Clearly, this state is reached no later than the state of READY+ENABLED. The gain is signified when the enabling condition of the attribute can only be evaluated at a very late stage.

With regards to the unneeded attributes, early detection of unneeded attributes will reduce both response time and work load. Our algorithm performs best compared to a naive algorithm, which does not identify unneeded attributes at all, with the following two kinds of decision flows: (1) Decision flows with many of its enabling conditions evaluated to false. (2) Decision flows with long shortest source to target paths. An attribute can be marked as unneeded if the following two cases are true: (a) The attribute is an input to enabling condition(s) of unneeded attributes and/or computation of unneeded attributes; (b) If the attribute is an input to enabling condition(s) of attribute(s) that have not been marked as unneeded, the enabling condition(s) can be short-circuit evaluated without referring to the attribute. For long decision flows, if attributes close to the target attributes become unneeded, the unneeded status may be propagated backwards to many other attributes.

Early detection of necessary attributes allows the task scheduler to schedule the tasks for the evaluation of these attributes first to reduce speculative executions of other tasks in parallel processing environments. However, this is just a heuristic. It is not always optimal. For example, we can imagine sometimes the execution of an attribute other than a necessary one may lead to early detection of unneeded attributes. In general, our algorithm for detecting necessary attributes has the most significant impact on the performance for the decision flows that also produce large number of eligible attributes at early point of time.

We implemented an execution engine prototype with Eager_Plus_Unneeded Algorithm built in. Using this prototype, we conducted a set of benchmarking experiments (see [3] for a detailed description of the prototype and experiments). We found out that the Eager_Plus_Unneeded Algorithm can reduce the response time and work load between 30% to 70% in the context of the experiments. The experiments confirmed that decision flows with large number of DISABLED attributes (determined at run time) benefit the most from the algorithm. They take advantage of detecting DISABLED attributes early so that it can detect many unneeded attributes. Hence, the more potential disabled nodes there are, the more attributes can be eliminated for evaluation. We also found out that more enabling edges in the decision flow amplifies the effect of the algorithm. This is because that more enabling edges will allow more rapid propagation of the results of the eager evaluation.

7 Conclusions

We described a decision flow model that is useful in expressing business logic in E-commerce applications. The model provides a high level specification language with declarative semantics. As a result, decision flows can be understood by managers from different parts of an enterprise, and can serve as the "contract" between them. In addition, the contract itself is executable since an operational semantics of the decision flow model is easily derived from the declarative semantics.

We developed algorithms for eager detection of eligible, unneeded, and necessary tasks to support efficient execution of decision flow. We show analytically that the optimization algorithms have very significant impact of the performance on the execution of workflows, which was also confirmed by experiments reported in reference [3].

Many related issues are remain to be studied. One issue concerns migration of this technology into practical workflow systems. How can we specify that a subworkflow of a workflow has the properties of a decision flow? Once identified, how would a practical workflow system apply the optimizations of this paper to the subworkflow?

References

[1] P. C. Attie, M. P. Singh, A. Sheth, and M. Rusinkiewicz. Specifying and enforcing intertask dependencies. In *Proc. of Intl. Conf. on Very Large Data Bases*, pages 134–145, 1993.

[2] C. A. Ellis and G. J. Nutt. Office information systems and computer science. *ACM Computing Surveys*, 12(1):27–60, March 1980.

[3] R. Hull, F. Llirbat, B. Kumar, G. Zhou, J. Su, and G. Dong. Optimization techniques for data-intensive decision flows. Technical report, Bell Laboratories, Lucent Technologies, Murray Hill, NJ, 1999. http://www-db.research.bell-labs.com/projects/vortex/benchmark.ps.

[4] R. Hull, F. Llirbat, E. Simon, J. Su, G. Dong, B. Kumar, and G. Zhou. Declarative workflows that support easy modification and dynamic browsing. In *Proc. of Intl. Joint Conf. on Work Activities Coordination and Collaboration (WACC)*, pages 69–78, February 1999.

[5] R. Hull, F. Llirbat, J. Su, G. Dong, B. Kumar, and G. Zhou. Efficient support for decision flows in e-commerce applications (long version). Technical report, Bell Laboratories, Lucent Technologies, Murray Hill, NJ, 1999. http://www-db.research.bell-labs.com/projects/vortex/ictec-full.ps.

[6] N. Krishnakumar and A. Sheth. Managing heterogeneous multi-systems tasks to support enterprise-wide operations. *Distributed and Parallel Databases*, 3(2), 1995.

[7] F. Leymann and D. Roller. Business process management with FlowMark. In *Proc. of IEEE Computer Conference*, pages 230–234, 1994.

[8] D. Patterson, J. Hennessy, and D. Goldberg. *Computer Architecture : A Quantitative Approach*. Morgan-Kaufmann Publishers, Inc., 1996.

[9] D. Peppers and M. Rogers. *The One to One Future*. Doubleday, New York, 1993.

[10] P. Piatko, R. Yangarber, D. Lin, and D. Shasha. Thinksheet: A tool for tailoring complex documents. In *Proc. ACM SIGMOD Symp. on the Management of Data*, page 546, 1996.

[11] P. B. Seybold and R. T. Marshak. *customers.com*. Random House, New York, 1998.

[12] D. Wodtke, J. Weissenfels, G. Weikum, and A. K. Dittrich. The Mentor project: Steps towards enterprise-wide workflow management. In *Proc. of IEEE Intl. Conf. on Data Engineering*, New Orleans, 1996.

[13] D. Woelk, P. Attie, P. Cannata, G. Meridith, A. Sheth, M. Singh, and C. Tomlinson. Task scheduling using intertask dependencies in Carnot. In *Proc. ACM SIGMOD Symp. on the Management of Data*, pages 491–494, 1993.