Proc. Int. Conf. on Work Activities Coordination and Collaboration (WACC), 1999

Declarative Workflows that Support Easy Modification and Dynamic Browsing

Richard Hull* Francois Llirbat* Eric Simon† Jianwen Su‡
Guozhu Dong§ Bharat Kumar* Gang Zhou*

ABSTRACT

A new programming paradigm named "Vortex" is introduced for specifying a wide range of decision-making activities including, in particular, workflows. In Vortex workflows are specified declaratively. A particular emphasis is on "object-focused" workflows, i.e., workflows focused on how individual input objects should be processed within an organization. Such workflows arise commonly in practice, including insurance claims processing, and many electronic commerce applications, and in the area of Customer Care, e.g., web-based storefronts. Vortex workflows are "attribute-centric", because they are centered around how the attribute values for an input object are gathered and computed. Initially, only a few attributes of an input object have assigned values. During processing of the object, additional attribute values may be assigned by external modules, or by internal modules, including "decision modules". Decision modules include "attribute rules" that specify contributions to specific attribute values; these are combined with one of a broad family of available semantics. In Vortex, enabling conditions are used to determine what attributes should be evaluated. A novel choice-based execution model provides a general framework for optimization strategies. The use of enabling conditions, attribute rules and declarative semantics makes Vortex workflows easier to modify and refine than traditional, procedurally specified workflows. Vortex supports modularity and permits the natural intermixing of Vortex workflows with traditional, procedural workflows. The paper introduces a novel spreadsheet-like interface for dynamic browsing of Vortex executions.

Keywords

Workflow management, decision-making, declarative semantics, choice-based execution, browsing

1 INTRODUCTION

The workflows used in industry are becoming increasingly complex. This results from new applications, such as electronic commerce, the need for "mass customization" in customer care, and the need that some parts of workflows operate with near-realtime response times. It is also becoming imperative that non-programmers (e.g., business managers) be able to understand the effect of workflows, even complex ones, and in some cases to suggest or even perform modifications to these workflows. This paper presents a new programming paradigm named "Vortex" which can be used to specify a wide range of decision-making activities including, in particular, workflows. In supporting workflows, the Vortex paradigm meets several of the challenges listed above, in part because it supports declarative specification.

The Vortex paradigm focuses on workflow applications that are "object-focused", i.e., that focus on how individual objects (insurance claims, mortgage applications, customer sessions with a web-based catalog) should be processed using a workflow. The Vortex paradigm is said to be "attribute-centric" because workflows specified in Vortex are centered around how the attribute values for an input object are gathered and/or computed during execution of the workflow. Initially, only a few attributes of an incoming object have assigned values. During processing of the object, additional attribute values may be assigned. The decision whether an attribute is to be evaluated is controlled by the "enabling condition" for the attribute (or more properly, the module defining it). There are four fundamental ways of determining attribute values: by "decision modules", by Vortex modules, by modules based on a flowchart language, and by "foreign" modules, i.e., modules outside of the Vortex paradigm (black boxes). A decision module can be used to specify how a single attribute is to be computed. This includes a set of "attribute rules", and a program written in the Combining Semantics Language (CSL) introduced here, that specifies how the contributions of these rules should be aggregated and synthesized. A wide variety combining semantics can be specified in CSL.

The use of foreign modules is common in workflow specifcation languages. For the purposes of runtime optimization, we carefully distinguish here between modules with externally relevant side effects and those without. Foreign modules that gather attribute values (e.g., a database query) may be viewed as having no side effects, and may be invoked in an eager fashion to reduce runtime. Invocation of other foreign modules, either to gather attribute values (e.g., have a human agent fill in entries in a form) or to invoke actions (e.g., issue a check) are viewed as having side effects, and are not invoked in an eager fashion.

In addition to the core aspects of Vortex, the paper also presents tools for complex workflow applications. In particular, the Vortex paradigm supports modularized specification of workflows. It allows a workflow specification to be packaged as a module and subsequently used in defining more complicated workflows. The concept of a module is supported not only in the syntax but also in the semantics. It is noted that Vortex further permits the natural intermixing of Vortex workflows with traditional, procedural workflows. At the conceptual level, the paper introduces DecisionSheets, a novel spreadsheet-like interface for dynamic browsing of Vortex workflows.

At the implementation level this paper sketches an implementation approach which is general and flexible. The execution model is "choice-based" and facilitates optimization of Vortex workflows. In particular, the execution model is based on two things: an engine that can execute specialized parallel, procedural programs in near-realtime, and a framework for mapping declarative Vortex workflows to that engine. The engine is fundamentally interpretive, and a workflow can be modified without recompiling the engine or interrupting the processing on currently active instances of a workflow. The mapping framework supports a variety of optimization strategies that permit trade-offs between system load and improved

* Bell Laboratories, Lucent Technologies, 600 Mountain Ave., Murray Hill, NJ 07974. {hull,llirbat,bharat,gzhou}@research.bell-labs.com.
† INRIA Rocquencourt, Domaine de Voluceau, 78153 Le Chesnay, France. Eric.Simon@inria.fr. Part of work was done while visiting the Bell Labs.
‡ Department of Computer Science, University of California, Santa Barbara, CA 93106. su@cs.ucsb.edu. Part of work was done while visiting the Bell Labs and part of work was supported by NSF grants IRI-9411330 and IRI-9700370.
§ Department of Computer Science and Engineering, Wright State University, Dayton, Ohio 45435. gdong@cs.wright.edu. Part of work was done while visiting the Bell Labs during his sabatical from University of Melbourne, Australia.

Appears in *Proc. ACM International Joint Conference on Work Activities Coordination and Collaboration, WACC '99, Feb. 22–25, San Francisco, CA 1999, pp. 69–78*

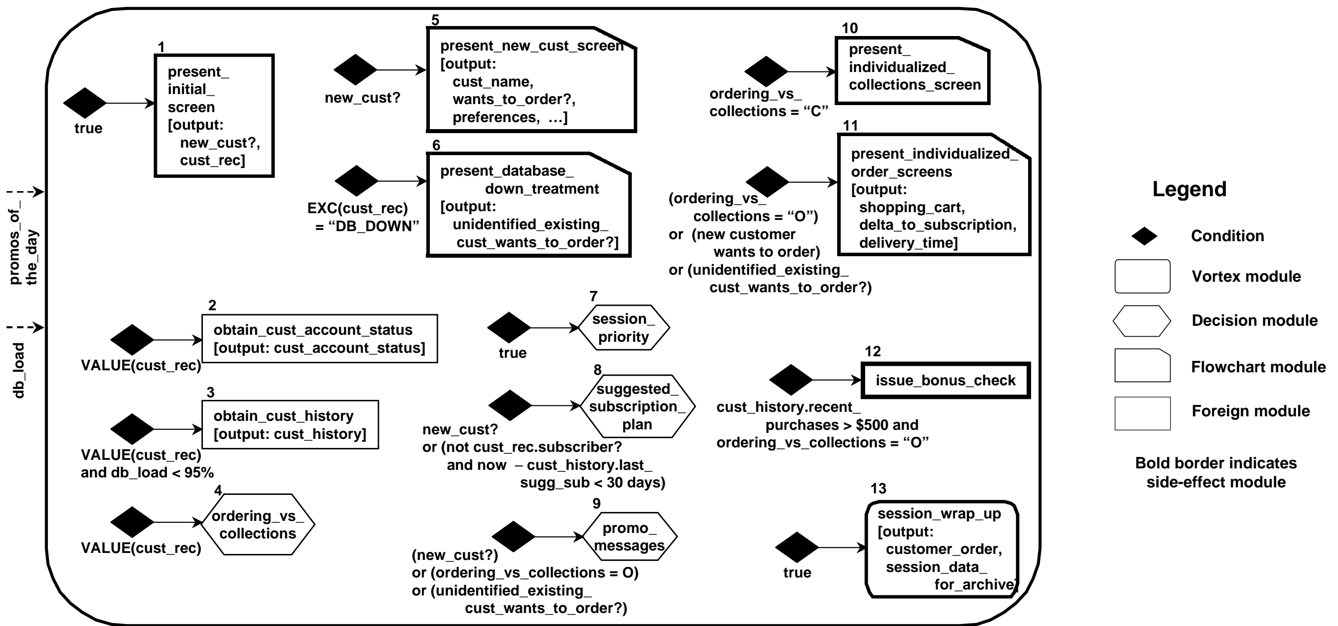

Figure 1: Vortex workflow for web-storefront of grocery store

response times. At Bell Labs we have built a prototype of the engine that works on procedural workflows in near realtime; we are currently extending the prototype to support mapping of declarative workflows to the engine.

Organization of the paper: § 2 introduces through intuitive examples the primitives and semantics in the Vortex paradigm. § 3 presents the Combining Semantics Language, which is used to specify alternative semantics for execution of rules specifying how attribute values are to be computed. § 4 presents an operational semantics for Vortex, that supports various optimizations. § 5 introduces the DecisionSheet interface for dynamic browsing. § 6 summarizes relevant work. Finally, § 7 briefly mentions some of the future directions we plan to pursue, including prototyping work on an engine that will implement the Vortex paradigm.

A more detailed presentation of the material in this paper may be found at www-db.research.bell-labs.com/projects/vortex/.

2 VORTEX MODULES

This and the following section introduce key elements of the Vortex workflow model. We present an informal tutorial, along with a handful of formal definitions. This section is focused at a high level, on how Vortex programs specify what tasks should be performed and under what conditions. The next section is focused at a more detailed level, describing a novel approach to specifying the computation of attribute values.

We begin with the high-level syntax for Vortex programs, and then present a *declarative* semantics for them. In the discussion we sometimes refer informally to how Vortex workflows might be executed; the detailed operational semantics is presented in § 4.

Attributes and modules: Vortex is attribute-centric, in the sense that for each input to a Vortex workflow, the primary activity of the workflow instance for this input is to obtain values for some or all of the attributes associated with that input. External actions, such as interacting with an external customer, having a human agent perform a task, or issuing a check, use the attribute values obtained as parameters.

Attribute values are computed by modules. Each module can return values for one or more attributes, and may have associated side-effects. Each module has an *enabling condition*, which indicates whether or not the module should be executed for a given workflow instance. The flow of control in Vortex programs is not explicitly specified; rather, it is inferred from the data flow requirements and enabling conditions of the modules.

There are four kinds of modules in Vortex programs:

Vortex (depicted using a rounded-corner rectangle): These have a declarative semantics, and contain modules along with enabling conditions.

Decision (depicted using a hexegon): These produce a single attribute value, using a novel framework for aggregating and synthesizing information based on CSL (described in § 3).

Flowchart (depicted using rectangle with cut corner): These are specified using a flowchart language using modules.

Foreign (depicted using a rectangle): These are implemented using functions outside of the Vortex paradigm (e.g., database queries, web server routines).

Vortex and decision modules are the focus of the current paper; flowchart modules are included in the overall framework to provide added flexibility to programmers and business managers.

Example: We introduce and illustrate salient features of the Vortex paradigm by describing a Vortex program for controlling a very simplified web-based storefront for a grocery store. (A typical application would involve several Vortex programs, for responding to different types of events that might arise.) We have included a high-level view of that program in Fig. 1 for reference. (The numbering indicated in these visual representations is included for exposition purposes only; they are not part of the formal program.) Note that the program is itself a Vortex module.

In the example, foreign module 1 presents an initial screen to a user, determines whether this is a new or existing customer, and retrieves the customer record, if appropriate. For existing customers data is obtained (using modules 2 and 3), and decision module 4 determines whether the customer will be allowed to order (flowchart module 11) or sent to a special collections treatment (module 10). Special treatment is provided for new customers (module 5) and for existing customers whose customer record was not found because of a database exception (module 6). A "session_priority" (module 7) is determined for all sessions (e.g., high priority sessions might get faster service), and in some cases promotional messages (module 9) and a suggested subscription plan (module 8) are selected. (These might be presented to users in module 11). Some users

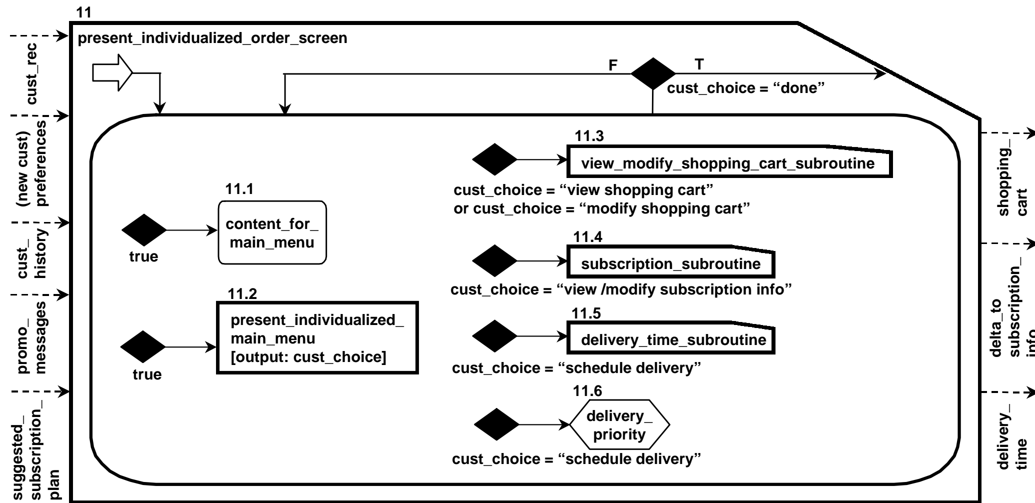

Figure 2: Drill-down for shopping cart

will receive bonus checks (module 12), and in all cases some data archiving is performed (Vortex module 13).

Fig. 2 shows a "drill down" into flowchart module 11. The simple flowchart of this module involes a single loop, which repeatedly calls a Vortex module. (We have not shown the input and output attributes for this Vortex module.) The Vortex module presents a menu page (module 11.2) to the user, allowing him to view or modify his shopping cart and his subscription status (e.g., to have deliveries weekly or bi-weekly), and to select a delivery time for his order. Vortex module 11.1 computes individualized content for the menu page each time that it is shown. (This might include, for example, targeted promotional messages.)

More generally, module 11 illustrates how the Vortex paradigm can be intermixed with other workflow paradigms. As detailed below, Vortex modules satisfy a certain acyclicity condition, and so flowchart modules are useful when cycles are needed. A specific flowchart language has been designed and implemented as part of the Vortex effort, but is not a focus of the current paper.

Vortex modules (e.g., module 13) consist of modules with enabling conditions. When implementing a Vortex workflow, the Vortex modules can be treated as "black boxes", or can be "flattened" so that optimization algorithms can take advantage of the internal structure of these modules.

Text-based specification: There is a natural text-based language for specifying modules. For example, module 7 is declared as follows. (We defer specification of how to compute this module until § 3).

```
Module: session_priority
 Enabling condition: true
 Input attributes:   cust_rec, cust_history,
    cust_account_status, ordering_vs_collections,
    wants_to_order?
 Output attributes:  session_priority: [1..5]
 Type:               decision
 Computation:        ...
 Side-effect:        no
```

This declares that module `M_session_priority` computes attribute `session_priority` whose type is the integers between 1 and 5. In practice, sessions with high priority might be placed on lightly loaded web servers, and given other preferential treatment. (By convention, the name of a decision module is the same as the name of the attribute it produces.)

Attribute and module states: A module is executed only if its enabling condition is true. As a result, some modules may be disabled, and not be executed. More generally, both modules and attributes may enter several different states. Attribute states are UNINITIALIZED, VALUE, EXC, DISABLED, and FAIL. The meaning of these states is as follows: VALUE: a value was obtained for this attribute; EXC: the evaluation of this attribute was attempted but an exception occurred; DISABLED: the enabling condition of the module defining this attribute was false (or that module executed but returned the state DISABLED for this attribute); FAIL: the module defining this attribute was enabled, but it aborted before producing one of the other 3 states for the attribute; UNINITIALIZED: essentially no information is available about this attribute. Module states are UNINITIALIZED, SUCCESS, EXC, and DISABLED. The meaning of these is as follows: SUCCESS: the enabling condition for the module was true, and the module executed successfully; EXC: the module was enabled, but an exception occurred; DISABLED: the enabling condition was false; UNINITIALIZED: essentially no information is available about this module. In general, only those attributes and modules performed externally from the workflow engine will return exception values.

An attribute (module) is *stable* if its state is in VALUE, EXC, DISABLED, FAIL (SUCCESS, EXC, DISABLED). (For both attributes and modules, the state UNINITIALIZED is primarily useful in connection with the operational semantics.)

Enabling conditions can refer to attribute values (e.g., the enabling condition for module 11), to the "state" of attributes, e.g., whether it has a value (module 2) or whether an exception occurred when attempting to evaluate it (module 6). (We present the detailed syntax and semantics of conditions below.) The ability to explicitly use exception information is especially useful in workflows that are intended to produce a result in near-realtime, even if some subsystems are inaccessible.

Side-effect modules: Some modules are designed to produce side effects. This is the case for module 12 of Fig. 1, declared below.

```
Module: issue_bonus_check
 Enabling condition:cust_history.recent_purchases
          > $500 and ordering_vs_collections="O"
 Input attributes:  cust_record
 Output attributes: check_issued?: boolean(target)
 Computation:       issue_and_mail_check($50,
          cust_record.name, cust_record.address)
 Side-effect:       yes
```

This module computes attribute `check_issued?` and issues a bonus check to valuable customers. This module has an external and semantically important side effect, namely to make a payment to someone. This contrasts with `session_priority`, which executes a function on a local processor. In the figure we depict side-effect modules by using shapes with a wide border.

During execution, in order to reduce response time it may be useful to compute non-side-effect modules in an eager fashion, be-

fore it is known whether its enabling condition is true. This would not be appropriate for side-effect modules.

Attribute Roles: In a Vortex workflow attributes can have one of three roles: *source*, *target*, or *internal*. Source attributes serve as "input" and target attributes serve as "output". There are two source attributes in our example: `promos_of_the_day` and `db_load`. Attribute `promos_of_the_day` is used during selection of possible promotional messages to be presented to specific customers, and `db_load` is representative of information about system load that a workflow might use. The attributes of nodes 10 to 13 are target attributes. Attributes that are neither source nor target are internal.

Notation for modules: In our formalism, a Vortex program identifies a set Att of (typed) attributes along with a set Mod of modules to obtain the values of these attributes. A *module* is an expression of form "$M(A_1, ..., A_n; B_1, ..., B_k)$", where M is a module name, $\{A_1, ..., A_n\}$ and $\{B_1, ..., B_k\}$ are disjoint sets of input and output (resp.) attributes. A module *defines* each of its output attributes. The semantics of modules is defined so that the attributes having no value or exception value are properly handled. For decision modules, additional requirements will be discussed in § 3.

Vortex workflow schemas: We now present the formal notion of Vortex workflow "schema", which provides an abstract formalism for representing Vortex programs. The formalism makes it easy to describe an acyclicity condition that Vortex workflows must satisfy.

We start with the notion of "pre-schema", which includes all of the components of a schema, but may violate the acyclicity requirement. A *pre-schema* is a tuple $\mathcal{S} = (Att, Src, Tgt, Mod, Cnd, \mathit{Eff})$ where

(a) Att is a set of attributes.
(b) Src and Tgt are disjoint subsets of Att (the *source* and *target* attributes, respectively).
(c) Mod is a set of modules such (i) each input attribute of each module is in Att and (ii) each $A \in (Att - Src)$ is defined by exactly one module. The function names used in different modules are distinct.
(d) $Cnd = \{\gamma_M \mid M \in Mod\}$ is a set of conditions using attributes from Att. (We use γ_M to denote the *enabling condition* for module M.)
(e) Eff is a subset of Mod (the *side-effect* modules).

To define the acyclicity condition, we associate a graph $\mathcal{G}_\mathcal{S}$ to each pre-schema $\mathcal{S} = (Att, Src, Tgt, Mod, Cnd, \mathit{Eff})$. The nodes of $\mathcal{G}_\mathcal{S}$ are $Mod \cup Cnd$; and the edges are $E_c \cup E_d$ where $E_c = \{(M, \gamma_{M'}) \mid M, M' \in Mod$ and some output attribute of M occurs in $\gamma_{M'}\} \cup \{(\gamma_{M'}, M') \mid M' \in Mod\}$ is the set of *control edges*, and $E_d = \{(M, M') \mid M, M' \in Mod$ and some output attribute of M is an input attribute for $M'\}$ is the set of *data edges*.

Definition 2.1 A (*Vortex*) *workflow schema* is a pre-schema $\mathcal{S}$ such that $\mathcal{G}_\mathcal{S}$ is acyclic.

We make the following assumption with regards to side-effect modules: If attributes M and M' are incomparable in $\mathcal{G}_\mathcal{S}$ then executions of M and M' commute (even if one or both of them involves side-effects).

For the remainder of this section we assume that a workflow schema $\mathcal{S} = (Att, Src, Tgt, Mod, Cnd, \mathit{Eff})$ is fixed.

Operational semantics (informal): The Vortex paradigm is fundamentally declarative. Programmers specify a collection of modules that may potentially be executed during workflow instances, along with enabling conditions that govern whether the modules should be executed. The semantics of Vortex workflows is defined in a declarative manner. We shall provide this declarative semantics shortly; but to provide some of the underlying intuition, we first give a brief description of an operational semantics for Vortex workflows.

Speaking loosely, a Vortex program executes as follows: An input object is presented, where the source attributes are either given values or the DISABLED state. Modules will be executed incrementally, producing values for some attributes, and making enabling conditions true or false. A module can be executed when its enabling condition becomes true and when it is known that its input attributes are stable (i.e., their states will not change). A module can produce values for its output attributes one at a time, or all at once. Execution can end after all relevant target attributes have been executed. If a module is DISABLED, then all of its output attributes will be DISABLED and will not take values.

The acyclicity condition on Vortex workflows guarantees a form of *monotonicity* for attribute values. In particular, once an attribute A has value assigned during execution of the workflow, the value associated to A will not change.[1] This implies that if an enabling condition for some module becomes true, it will not subsequently become false. (The restriction mentioned above concerning commutativity of side-effect modules is also needed to ensure this monotonicity.)

Snapshots: A key notion is that of "snapshot", which is used to describe the status of the system at different stages of an execution. We introduce now a simple notion of snapshot that will be used to present the declarative semantics. (A richer notion of snapshot is presented in § 4.)

Definition 2.2 A *snapshot* for workflow schema $\mathcal{S}$ is a 3-tuple $s = (\sigma, \mu, \xi)$ where

(a) The *state function* σ is a total function from $Mod \cup Att$, which maps modules to {UNINITIALIZED, SUCCESS, EXC, DISABLED} and attributes to {UNINITIALIZED, VALUE, EXC, DISABLED, FAIL}.
(b) The *exception function* ξ is a partial function from $Mod \cup Att$ to exception values, where $\xi(X)$ has a value iff $\sigma(X) =$ EXC. In this case, $\xi(X)$ is an exception value for X.
(c) The *attribute value mapping* μ is a partial function from Att into $\cup\{\tau(A) \mid A \in Att\}$, such that for each A, $\mu(A)$ is defined iff $\sigma(A) =$ VALUE. In this case, $\mu(A) \in \tau(A)$.
(d) The state function σ satisfies the following condition for each module M and attribute A defined by M: (i) If M is in SUCCESS or EXC, then A must be stable; (ii) if M is in SUCCESS, then no attribute defined by M can be in state FAIL; and (iii) if M is in DISABLED [UNINITIALIZED], then each attribute defined by M is in state DISABLED [UNINITIALIZED].

A snapshot is *initial* if each source attribute has state VALUE or DISABLED, and all modules (and non-source attributes) have state UNINITIALIZED. A snapshot is *complete* if each attribute and module is in stable state. In the declarative semantics, a Vortex workflow is viewed as an input-output device which essentially maps initial snapshots into complete snapshots.

Syntax and semantics of conditions: The syntax for enabling conditions permits access to the states, values and exception values of attributes and modules.

Conditions are built from atoms. Atoms such as VALUE(A), EXC(M), DISABLED(M) can be used to test the state of attributes and modules. (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 expressions of form A[VALUE] (abbreviated as A), which denotes the value

[1] We assume that source attributes, including attributes such as `db_load`, are read only once during a workflow instance. The attribute value may be read when the workflow instance begins, or at sime time during execution of the workflow instance.

of attribute A (if defined); A[EXC], the exception value of A; and M[EXC], the exception value for module M. 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 and module M referred to by $pred(t_1, ..., t_k)$ has a correct state (i.e. the state VALUE if A[VALUE] occurs, and the state EXC if A[EXC] or M[EXC] occurs), 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., $A > 3 \vee A \leqslant 3$ is not a tautology.) The *truth value* of condition γ in a snapshot s is denoted *TruthVal*(γ, s).

Consistency of complete snapshots We now present two notions of consistency. (A third notion, involving decision modules, is defined in § 3.) A complete snapshot $s = (\sigma, \mu, \xi)$ is *enabling-consistent* if for each module M, (a) *TruthVal*(γ_M, s) = true iff $\sigma(M) \in$ {VALUE, EXC}, and *TruthVal*(γ_M, s) = false iff $\sigma(A)$ = DISABLED.

The second notion of consistency is about the relationship between the values of the attributes and the modules that define them. To provide an interpretation for the behavior of modules we define an *environment* for workflow schema $\mathcal{S}$ to be a mapping $\mathcal{E}$ such that, for each module $M(A_1, ..., A_n; B_1, ..., B_k)$ in $\mathcal{S}$, $\mathcal{E}(M)$ is a total mapping with the same signature as M. As we shall see, we do not need to specify explicitly the side effects associated with side-effect modules. The use of a static environment in this formalism is a convenience; in practice Vortex workflows will operate in the context of a dynamic world.

A complete snapshot $s = (\sigma, \mu, \xi)$ is *$\mathcal{E}$-consistent* for environment $\mathcal{E}$ if for each attribute A such that $\sigma(A)$ = VALUE, $\mu(A)$ is equal to the A-value computed by $\mathcal{E}(M)(\mu(A_1), ..., \mu(A_n))$.

The notion of environment presented here is "blind" to the contents of modules. After defining how computations are specified for decision modules in § 3, the notion of environment will be refined to reflect the intended semantics of such modules. Further refinement in connection with flowchart modules is made in the full model.

Declarative semantics, ignoring side-effects: We present the semantics of workflow schemas in two phases, first ignoring side-effects and then incorporating them. To this end, suppose for now that no modules in $\mathcal{S}$ have side-effect.

As noted above, workflow schemas are viewed as input-output devices. The input will be an environment $\mathcal{E}$ and an initial snapshot s. The output will be states and values for the target attributes.

We say that a complete snapshot s' *extends* initial snapshot s if the state and value functions of s' agree respectively with those functions of s on every attribute which is stable in s. The following result (proof omitted) is demonstrated by a straightforward induction, using the acyclicity of $\mathcal{G}_\mathcal{S}$.

Proposition 2.3 Let $\mathcal{E}$ be an environment for $\mathcal{S}$. For each initial snapshot s there exists a unique complete enabling- and $\mathcal{E}$-consistent snapshot s' that extends s. In this case, s' is called the *completion* of s for $\mathcal{S}$ under $\mathcal{E}$.

We are not interested in all of the completion of s, only in the states and values of target attributes. Let s be a complete snapshot for $\mathcal{S}$. The *projection* of $s = (\sigma, \mu, \xi)$ onto a set $X \subset Att$, denoted $s|_X$, is the 3-tuple $(\sigma|_X, \mu|_X, \xi|_X)$ consisting of the restrictions of the functions in s to domain X.

Definition 2.4 Suppose workflow schema $\mathcal{S}$ has no side-effect modules. The *declarative semantics* of $\mathcal{S}$ under environment $\mathcal{E}$ is defined as the function that maps each initial snapshot s to $t = s'|_{Tgt}$, where s' is the completion of s for $\mathcal{S}$ under $\mathcal{E}$.

1. if `cust.class = "platinum"` and `cust.subscriber = "true"` then `session_priority` $\leftarrow 4$
2. if `cust.class = "green"` then `session_priority` $\leftarrow 1$
3. if `cust_account_status = "credit"` then `session_priority` $\leftarrow 2$
4. if *match*(`cust_rec`, `promos_of_the_day`) then `session_priority` $\leftarrow 3$

Figure 3: Attribute rules for `session_priority`

Declarative semantics, with side-effects: What about side-effects of module executions? Suppose now that $\mathcal{S}$ has one or more side-effect modules. Let $\mathcal{E}$ be an environment and s a complete snapshot that is enabling- and $\mathcal{E}$-consistent. Suppose $M(A_1, ..., A_n; (B_1, ..., B_k)$ is a side-effect module where $\sigma(M) \in$ {SUCCESS, EXC}. Intuitively, this means that M was executed during the workflow instance represented by s. Note that the side-effect caused by M in s is completely determined by the string $M(v_1, ..., v_n)$, where v_j is the value of $\mu(A_j)$ (or $\perp$) for $j \in [1, n]$. We denote this string as *side_eff*(M, s). More generally, if $Y \subseteq$ *Eff*, the side-effects caused by Y in s are given by the set *side_eff*(Y, s) = $\{$*side_eff*$(M, s) \mid M \in Y$ and $\sigma(M) \in$ {SUCCESS,EXC}$\}$.

Suppose now that s is an initial snapshot and $s' = (\sigma', \mu', \xi')$ is the completion of s for $\mathcal{S}$ under $\mathcal{E}$. The question is: what side-effects from s' should be included in the declarative semantics as part of the output of s? For each module $M \in$ the set Eff_{Tgt} of side-effect modules that define target attributes, if $\sigma(M) \in$ {SUCCESS, EXC} then the workflow execution should produce *side_eff*(M, s'). This is because the workflow instance must produce something for the target attributes defined by M, and so M must have been executed. What about the side-effects of a module M that do not produce target attributes? We argue that there is no fundamental reason that *side_eff*(M, s') be produced. In particular, a "clever" execution of the workflow might be able to produce correct values for the target attributes without executing module M.

Definition 2.5 The *declarative semantics* of workflow schema $\mathcal{S}$ with side-effects under environment $\mathcal{E}$ is defined as the relation containing all pairs of form $(s, (t, e))$ where

(a) s is an initial snapshot,
(b) $t = s'|_{Tgt}$, where s' is the completion of s for $\mathcal{S}$ under $\mathcal{E}$, and
(c) e is a set of side effect expressions such that
side_eff$(Eff_{Tgt}, s') \subseteq e \subseteq$ *side_eff*(Eff, s').

3 DECISION MODULES

This section describes decision modules, one of the novel aspects of the Vortex workflows. Decision modules derive the value of an attribute from the values of other attributes. A decision module consists of a set of "attribute rules", along with a combining function specified using the Combining Semantics Language (CSL) introduced here. Each attribute rule whose associated condition is true suggests a possible contribution to the attribute value. The "combining function" then synthesizes the contributions to provide the final value for the attribute. Decision modules support a broad variety of approaches to aggregating and synthesizing information.

Example 3.1 Module 7 of Fig. 1 is a decision module which determines the value of `session _priority`. This attribute takes an integer value between 1 and 5. Some representative attribute rules for this attribute are shown in Fig. 3. The combining semantics to be used for `session _priority` is "high value wins". Informally, for a given workflow instance, a set of integers are produced by those rules whose conditions are true; then the maximum value in that set is assigned to `session_priority`. For rules

1. if `"cust ordered every week for past month"`
   then `suggested_subscription_plan` $\leftarrow$ ⟨`weekly`, 20⟩
2. if `"cust ordered 2 times per month for last 2 months"`
   then `suggested_subscription_plan` $\leftarrow$ ⟨`bi_weekly`, 15⟩
3. if `cust.class` $\in$ {`"platinum"`, `"gold"`}
   then `suggested_subscription_plan` $\leftarrow$ ⟨`weekly`, 10⟩

Figure 4: Attribute rules for `suggested_subscription_plan`

that produce numeric values it is easy to develop other combining semantics (e.g., min, sum, average). ■

In Vortex, all the rules for an attribute must return values of the same type. The type need not be atomic.

Example 3.2 Consider attribute `suggested_subscription_plan`, whose range is an enumerated set { `weekly`, `bi_weekly`, ...}. The rules of Fig. 4 each return an ordered pair, with first coordinate a subscription plan and second coordinate an integer serving as a weight. Under the "weighted-sum-semantics", the contributed values of rules with true conditions are grouped according to the first coordinate. The weights within each group are summed, and the subscription plan with maximum sum is chosen. (A mechanism for breaking ties must also be specified.) Again, aggregates other than sum can be used. ■

Combining functions are expressed using the *Combining Semantics Language* (*CSL*). CSL is based on an algebra of simple operators that permit us to manipulate collections of values. The choice of the basic operators for our language was motivated by the examples of combining semantics we have encountered in practice. CSL is very close to database programming languages [BTS91, OBBT89, BBKV88, PSV92], and contains some additional operators for list manipulation that appeared to be very useful in practice. CSL permits us to define families of combining functions, and it can even be used by a naive user to construct new combining functions.

CSL values: The data produced by the attribute rules and manipulated by the combining functions are based on a type system that defines the following *value types* (called *CSL value types*). Each CSL value type is extended with the value $\perp$ (standing for "No value") as a specific occurrence.

- atom. Examples of atom types are integer, char or string.
- $\langle a_1 : \tau_1, ..., a_n : \tau_n \rangle$ is a tuple type, if each τ_i is a CSL value type and each a_i is the name of the attribute.
- $[\tau]$ is a homogeneous list type, if τ is a CSL value type.
- $\{\!|\tau|\!\}$ is a homogeneous bag type, if τ is a CSL value type.

In the following, we assume that all the workflow attributes have CSL value types.

Attribute rules: Let Att be a set of attributes and A an attribute in Att. An *attribute rule* r for A has the form "if *cond* then $A \leftarrow$ *a-term*", where *cond* is a condition as defined in the previous section, *a-term* (attribute term) is an expression that specifies a CSL value. More precisely, *a-term* is either (i) a constant value, (ii) the name of an attribute B in Att, different from A or, (iii) the call of a user defined function that applies on the extended values of some attributes in Att and produces a CSL value. In this last case, *a-term* has the form $f(e_1, ..., e_k)$ where f is the name of the user defined function and each e_i ($i \in [1..k]$) is either a constant value or the name of an attribute B different from A.

Definition 3.3 Let Att be a set of attributes and $r =$ "if *cond* then $A \leftarrow$ *a-term*" an attribute rule. Let $s = (\sigma, \mu, \xi)$ be a snapshot. Let $s = (\sigma, \mu, \xi)$ be a complete snapshot. The result of the execution of r is a CSL value, denoted $\text{EXEC}(r, s)$, obtained as follows: $\text{EXEC}(r, s) = \perp$ if the condition of r is evaluated to false. If the condition of r is true, $\text{EXEC}(r, s)$ is the result of the evaluation of the expression *a-term* on s where each reference to an attribute $B \in Att$ is replaced by its value $\mu(B)$ if $\mu(B)$ is defined and $\perp$ otherwise.

Combining functions: A combining function applies on homogeneous collections of CSL values and produces a new collection of CSL values. The body of these functions are expressed using the following operators: (i) Constructors (like tupling (denoted$\langle\rangle$), bagging (denoted $\{\!|\,|\!\}$) or listing (denoted $[]$) that respectively allows to construct tuples, bags or lists; (ii) De-constructors like the $\text{proj}[a_i]$ operator that takes as input a tuple (or list) and returns the value of the attribute of name a_i. (iii) Operators parameterized by combining functions such as $\text{map}[f]$ and $\text{collect}[id, f]$. The $\text{map}[f]$ operator takes as input a collection of CSL values and applies the combining function f on each value of the collection. The $\text{collect}[id, f]$ operator (analogous to the collect operator in [PSV92], and the "fold" or "reduce" of many functional languages) is parameterized by a CSL value id and a binary combining function of type $T \times T \rightarrow T$ (T means any CS type). This operator permits computation of aggregates over collections. It takes as input a collection of CSL values and returns a CSL value of the same type that is obtained by recursively applying the function f over the elements of the collection. If the collection is empty it returns the value id. If the collection contains a single element, it returns that element. For example, $\text{collect}[0, +]$ computes the sum of a collection of integers: $\text{collect}[0, +]([1, 3, 6, 7]) = +(+(+(1, 3), 6), 7) = ((1 + 3) + 6) + 7 = 17$. Finally, factor and dot are binary operators that allow to combine collections with other CSL values. factor takes as input a list (or a bag) and any CSL value. It associates a value to each element of the list as follows: $\text{factor}([x_1, ..., x_n], y) = [\langle x_1, y\rangle, ..., \langle x_n, y\rangle]$. dot operator takes as input two lists and associates elements with the same position as follows: $\text{dot}([x_1, ..., x_n], [y_1, ..., y_n]) = [\langle x_1, y_1\rangle, ..., \langle x_k, y_k\rangle, \langle x_{k+1}, \perp\rangle, ..., \langle x_n, \perp\rangle]$. The CS operators can be combined in a CSL expressions using sequences and conditionnals. CSL can also use any arbitrary user-defined function over atomic values. We call such functions *atomic mappings*. The signatures of these functions are of the form $\tau_1 \times \cdots \times \tau_n \rightarrow \tau$ where $\tau_1, ..., \tau_n$ and τ are atom types.

A *CSL function* $f(x_1, ..., x_k)$ is an expression in CSL whose free variables are $\{x_1, ..., x_k\}$.

Example 3.4 The "high value wins" combining semantics, informally presented in Example 3.1, can be expressed by the following CSL function (HVW) that takes as input a list of integers:

$$\text{HVW}(x) = \text{collect}[\perp, \text{sup}_{int}](x)$$

Here $\text{sup}_{int}(x, y)$ is an atomic mapping that returns x if $y = \perp$ or x is greater than y, and returns y otherwise. By replacing the function sup_{int} with other functions, we can specify many variations such as "low value wins", "non null values wins". ■

In CSL, by using the CSL operators, we can define "macro" functions that are parameterized by other combining functions. An example of such a macro is $\text{group}[f, g]$; it is parameterized by two combining functions f and g. $\text{group}[f, g]$ subsumes the group operator in the nested relational algebra: It takes as input a collection S of CSL values and returns a collection of tuples of the form $\langle hash : v, vals : S'\rangle$ where v is an atom value and S' is a bag that contains all the values $g(x)$ where x belongs to S and is such that $f(x) = v$.

Combined with the CSL operators, macros allow easy specification of a large variety of combining semantics.

Example 3.5 We can now express in CSL the "weighted-sum" combining semantics, informally presented in Example 3.2. The (WSS) function takes as input a bag of 2-tuples $\langle v : atom, w : int\rangle$ where

v is the contributing value and w is the weight value associated with v. It is defined as follows:

$$\mathsf{WSS}(x) = \text{let } res = \mathsf{group}[\mathsf{proj}[v], \mathsf{proj}[w]](x) \text{ in } \mathsf{collect}[\bot, \mathsf{wsup}](\mathsf{map}[\mathsf{valssum}](res))$$

In the specification of WSS, we first group the values according to their contributing value (res). Then we compute the sum of the weights for each contributing value (valssum) and finally we take the value with the maximal weight (wsup). The CSL function valssum takes as input a set of tuples of the form $\langle hash : atom, vals : int\rangle$ and returns a set of tuple of the form $\langle v : atom, w : int\rangle$. The CSL function wsup takes as input two tuples of the form $\langle v : atom, w : int\rangle$ and returns the one with the higher value for w.

$$\mathsf{valssum}(x) = \langle v := \mathsf{proj}[hash](x), w := \mathsf{collect}[0, +](\mathsf{map}[\mathsf{proj}[vals]](x))\rangle$$
$$\mathsf{wsup}(x, y) = \text{ if } \mathsf{proj}[w](x) \geqslant \mathsf{proj}[w](y) \text{ then } x \text{ else } y.$$

Here again, we can specify many variations of "Weighted-Sum Semantics" by replacing the functions valssum and wsup. ∎

Due to space limitations, we do not include more intricate kinds of combining semantics which we have encountered in practice and which can be expressed using CSL.

Decision-consistent environments: A decision module M for an attribute A is a set of attribute rules for A together with a combining function specified in CSL. The combining function takes as input the set of values produced by the execution of the rules and returns a value for A. We can now define the notion of an environment being consistent with decision modules.

Definition 3.6 An environment $\mathcal{E}$ is *decision-consistent* for $\mathcal{S}$ if the semantics assigned by $\mathcal{E}$ to each decision module M in $\mathcal{S}$ agrees with the specification of M.

In the sequel we consider only decision-consistent environments.

4 OPERATIONAL SEMANTICS

In this section we describe a framework for specifying an operational semantics for Vortex workflows. The framework is based on the use of "choice" functions, as found in active database systems (Starburst, Chimera [WC95]), expert systems (OPS5 [For81]), and elsewhere. By using proper choice functions we can obtain a variety of dynamic optimizations (e.g., to minimize response time) during the execution.

In the framework developed here, execution of a Vortex workflow consists in producing a sequence of "operational snapshots" (these are generalizations of the notion of snapshot developed in § 2). New snapshots are produced in two possible ways, namely, choice: based on current information the choice function may select new modules that should be "launched", and inference: new information about module and attribute states and values is incorporated into the current snapshot. The primary contribution of this section is to develop conditions on the choice and inference steps that are sufficient to guarantee that executions agree with the declarative semantics for Vortex.

In [HLS+99] we have used the framework presented here to develop correct choice and inference functions that identify, during execution, modules which are "useless" or "necessary" for completion of the execution. The framework also provides the basis for the development of additional optimizations. The framework presented here is using choice on modules in a Vortex workflow. It may be useful to support the use of choice for other aspects of a Vortex workflow, e.g., to permit eager evaluation of attribute rules within decision modules. This is briefly explored in § 5 below.

For this discussion we assume that a Vortex schema $\mathcal{S}$ = (*Att, Src, Tgt, Mod, Cnd, Eff*) is fixed.

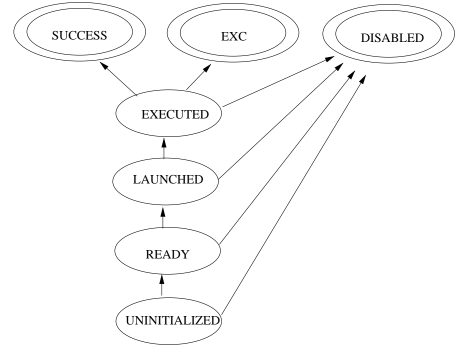

Figure 5: The states and transitions of a unit of processing

Operational snapshots: In the framework presented here, the choice function uses information about the current execution. To represent such dynamic information, we use an extended form of snapshot called "operational snapshot". Such snapshots reflect the execution states of all the modules in the workflow schema, the values and the states of all attributes at a given step of the execution, and all the side-effects that were produced since the beginning of the execution. It may also contain some auxiliary information that can be used by the choice and inference functions.

Definition 4.1 Let $\mathcal{S}$ be a workflow schema. An *operational snapshot* over $\mathcal{S}$ is a tuple $s = (\sigma, \mu, \xi, Side, Aux)$ where μ and ξ are as defined in § 2. The state function σ maps Mod to the set of execution states shown in the fsa in Fig. 5 and maps Att to the attribute states described in § 2. *Side* is a set of side-effect expressions (as defined in § 2), and Aux can be any auxiliary information.

Intuitively, the meaning of the states described in Fig. 5 is the following. (A double circle represents a possible terminal state.) States UNINITIALIZED, DISABLED, SUCCESS and EXC have the same meaning as in § 2. State READY indicates that all the input attributes of the module are stable. State LAUNCHED indicates that the module has been selected by the choice function and has started its execution. We assume that all side-effects of a module occur as soon as it is launched. State EXECUTED indicates the module has terminated its execution (succesfully or with an exception), in which case some attribute values may have been returned. At the begining of the execution all the modules are in state UNINITIALIZED. Also, in an operational snapshot s, the set *Side* corresponds intuitively to the set of side-effects that occurred before or when s was produced.

Choice-based operational framework: An instance of the workflow $\mathcal{S}$ is modeled as a sequence of operational snapshots. Importantly, we do not allow back-tracking (or rollback) in the execution sequences. This implies that executions always make "progress". We formalize this notion of progress by the notion of "extends":

Definition 4.2 Let $s = (\sigma, \mu, \xi, Side, Aux)$ and $s' = (\sigma', \mu', \xi', Side', Aux')$ be operational snapshots over $\mathcal{S}$. Then s' *extends* s if

(a) For each M in Mod, $\sigma(M) \leqslant \sigma'(M)$ in the ordering of states given by the fsa in Fig. 5.
(b) For each A in Att, $\sigma(A)$ = UNINITIALIZED or $\sigma(A) = \sigma'(A)$.
(c) The partial mapping μ (resp. ξ) agrees with μ' (resp. ξ') when it is defined.
(d) $Side \subseteq Side'$.

A (non-operational) complete snapshot s' (as defined in § 2) *extends* s if conditions (a), (b), and (c) hold.

Since execution sequences cannot be rolled-back, we have to avoid intermediate execution steps that could lead to a "dead-end" or a final result inconsistent with the declarative semantics. One kind of problem arises if an incorrect output for a decision module is used. (This is prevented by condition (i) in the definition below.)

Another kind of problem arises if the value of an enabling condition is determined incorrectly. A final kind of problem arises if some undesirable side-effects are produced. This can occur if a side-effect module is launched prematurely. (The latter two kinds of problems are prevented by condition (ii) in the definition below.) We say that a (decision-consistent) environment $\mathcal{E}$ is *compatible* with operational snapshot s if it agrees with all defined values of μ and ξ in s.

Definition 4.3 An operational snapshot $s = (\sigma, \mu, \xi, Side, Aux)$ over $\mathcal{S}$ is *permissible* if (i) there exists an environment $\mathcal{E}$ that is compatible with s, and (ii) for each environment $\mathcal{E}$ compatible with s, if s' is the completion of $s|_{Src}$ for $\mathcal{S}$ under $\mathcal{E}$, then s' extends s and $Side \subseteq side_eff(Eff, s')$.

To illustrate item (ii) above, suppose that in s side-effect module M is in state LAUNCHED, but some attributes used in γ_M are not stable. Since M has been launched, the side-effect expression associated with M is in *Side*. Let $\mathcal{E}$ be an environment compatible with s and s' the associated completion of $s|_{Src}$. Suppose that under $\mathcal{E}$, γ_M is false. Then the side-effect expression for M is not in $side_eff(Eff, s')$, and s violates (ii).

In general, it is undecidable whether an operational snapshot is permissible [HLS+99]. However, as discussed below tractable sufficient conditions can be developed.

We now develop conditions that ensure that a sequence of snapshots provides a correct execution of a Vortex workflow. An *execution sequence* is a sequence $\vec{s}$ of operational snapshots $s_0 = (\sigma_0, \xi_0, \mu_0, Side_0, Aux_0), \cdots, s_n = (\sigma_n, \xi_n, \mu_n, Side_n, Aux_n)$. Part (d) of the following definition requires that if a module M is launched during $\vec{s}$, then there is a pair s_{i-1}, s_i such that M is READY in s_{i-1} and M is LAUNCHED in s_i. In this case, i is called the *launch-point* of M in $\vec{s}$, denoted $launch_point(M, \vec{s})$.

Definition 4.4 Execution sequence $\vec{s}$ is *proper* if

(a) s_0 is an initial snapshot.
(b) Each target attribute is stable in s_n.
(c) Each s_i is permissible.
(d) For each $i \in [1..n]$ and module M, if $\sigma_i(M) \in \{$LAUNCHED, EXECUTED, SUCCESS, EXC$\}$ and $\sigma_{i-1}(M) \in \{$UNINITIALIZED, READY$\}$, then $\sigma_i(M)$ = LAUNCHED and $\sigma_{i-1}(M)$ = READY.

We recall that the declarative semantics of a Vortex workflow is defined using environments. Condition (d) above helps us identify the environment corresponding to a proper execution sequence. Specifically, suppose that time stamps are associated with each operational snapshot in $\vec{s}$, and let $\text{EXEC}(\vec{s})$ be the sequence of module invocations corresponding to the launch-points occurring in $\vec{s}$. If M is launched in s_i, we assume the input attributes for M are populated from s_{i-1}. We assume further that if a value for attribute A first occurs in s_i, then the module defining A produced this value sometime between the time stamps of s_{i-1} and s_i. The *environment* of $\text{EXEC}(\vec{s})$, denoted $\mathcal{E}_{\text{EXEC}(\vec{s})}$, is defined so that for each module $M(A_1, ..., A_n; B_1, ..., B_k)$, $\mathcal{E}_{\text{EXEC}(\vec{s})}(M)$ is the function defined by module M at timestamp $launch_point(M, \vec{s})$ (and is irrelevant for those M not launched in $\vec{s}$).

The following result (proof omitted) shows that proper execution sequences produce outputs that are compatible with the declarative semantics for Vortex. (The notion of $s|_{Tgt}$ for operational snapshot s is defined in the natural manner.)

Proposition 4.5 Let $\vec{s} = s_0, \cdots, s_n$ be a proper execution sequence for $\mathcal{S}$. Let $t = s_n|_{Tgt}$. Then $(s_0, (t, Side_n))$ is an element of the declarative semantics for $\mathcal{S}$ under environment $\mathcal{E}_{\text{EXEC}(\vec{s})}$.

What restrictions does this imply on choice and inference functions? We focus on execution sequences $\vec{s}$ where each step is either a launch step or an inference step. For an operational snapshot s and set X of modules in state UNINITIALIZED, $launch(X, s)$ denotes the snapshot s' constructed from s by changing the state of each module in X to LAUNCHED. An execution sequence $\vec{s} = s_0, \cdots, s_n$ is *choice-based* using functions *Choice* and *inference* if for each $i \in [1..n]$, either

(a) $s_i = launch(Choice(\mathcal{S}, s_{i-1}), s_{i-1})$, or
(b) $s_i = inference(s_{i-1}, info)$ and i is not the launch-point of any module,

where *info* can be anything, but intuitively corresponds to new information about module and attribute states and values.

We now present sufficient conditions on choice and inference functions that will ensure compliance with the declarative semantics. A choice function *Choice* is *proper* if for each permissible snapshot s:

(a) If no module has state LAUNCHED and at least one module has state READY, then $Choice(\mathcal{S}, s) \neq \emptyset$.
(b) All modules in $Choice(\mathcal{S}, s)$ have the state READY.
(c) Let SE be the set of side-effect expressions associated with modules in $Choice(\mathcal{S}, s)$ in s. Then for each environment $\mathcal{E}$ compatible with s, $SE \subseteq side_eff(Eff, s')$ where s' is the completion of $s|_{Src}$ for $\mathcal{S}$ under $\mathcal{E}$.

Condition (a) helps to prevent starvation, by requiring that if no module is executing but some module is READY, then the choice function selects at least one module for launching. Function *inference* is *proper* if for each permissible operational snapshot s and value of *info*, $inference(s, info)$ is permissible.

Because the notions of proper choice and inference functions rely on the notion permissible, testing proper is undecidable. However, it is relatively easy to develop tractable sufficient conditions for proper. For example, in algorithms of [HLS+99] a side-effect module M is launched by a choice function only if an eager evaluation of the enabling condition for M yields true.

The following (proof omitted) shows that executions based on proper choice and inference functions are compatible with the declarative semantics.

Proposition 4.6 Let *Choice* and *inference* be proper. Let s_0 be an initial snapshot, and let $\vec{s} = s_0, \cdots, s_n$ be a sequence of operational states corresponding to an execution using *Choice* and *inference* that cannot be extended. Then $\vec{s}$ is a proper sequence, and so s_n is compatible with the declarative semantics.

5 DYNAMIC BROWSING USING DECISIONSHEETS

This section presents a brief introduction to the DecisionSheet interface for Vortex workflows. This interface uses elements of the SpreadSheets paradigm to support a form of dynamic browsing of the operation of a Vortex workflow on hypothetical inputs. The interface is useful for understanding and debugging both the semantics of a workflow, and for understanding how optimization strategies affect the processing of different kinds of inputs. We introduce the DecisionSheet interface with an example, and then make some general remarks.

Fig. 6 shows an instance of the DecisionSheet interface for the workflow described in Section 2.

The DecisionSheet interface can be used to show information about snapshots of a workflow instance. Figure 6 shows a snapshot that might arise in a workflow instance involving an existing customer, when using an operational semantics that is very eager. (Indeed, the example shows execution of some modules before their enabling conditions are known, and shows execution of some attribute rules in decision modules before all of the input attributes for those modules are known.)

Each column of the DecisionSheet corresponds to an attribute of $\mathcal{S}$. The attribute names are listed in Row -1, and the attribute states and values are indicated in Row 0. The attributes must be listed in an order that is compatible with the edges of $\mathcal{G}_{\mathcal{S}}$. As

	A	B	C	D	E	F	G	L	M	N
-3	input		present_initial_screen (module 1)		obtain_ cust_account_ status (module 2)	obtain_ cust_history (module 3)	ordering_ vs_collections (module 4)	session_ priority (module 7)	suggested_ subscription_ plan (module 8)	promo_ messages (module 9)
-2			foreign module		foreign module	foreign module	O wins	high value wins	weighted sum	selection by category
-1	promos_of_ the_day	database_ load	new_cust?	cust_rec	cust_account_ status	cust_history	ordering_ vs_collections	session_ priority	suggested_ subscription_ plan	promo_ messages
0	{ <peas, 10%>, <apples, 5%>, <lamb, 15%>}	97%	FALSE	<Joe, green card, not subscrib, loves lamb>		DISABLED		3		
1			SUCCESS		executing	DISABLED	⊥	⊥	⊥	
2								1	⊥	
3							⊥	(contributes value 2)	<weekly, 10>	
4								3		

Figure 6: Example of DecisionSheet interface

with SpreadSheets some columns may be hidden; in our example columns H through K (for the attributes generated by modules 5 and 6) are hidden, as well as all columns beyond column N.

Rows -3 to -1 of the DecisionSheet provide information about how the attributes are computed. Row -3 indicates the name of the module computing the attribute. We generalize the typical notion of cell found in SpreadSheets by permitting the cells of the first row to span a number of columns equaling the number of output attributes of the node. The second row indicates the kind of module, and in the case of decision modules, it indicates the name of the combining semantics function used.

In Row 0, which holds the states and values of attributes, light gray indicates that the attribute value has been established; dark gray that the attribute has state DISABLED; striped gray that the attribute is known to be enabled but the value is not yet known; and white means that no information about enabling or disabling is known yet. (Although not illustrated here, a distinct color might be used for attributes whose state is EXC.)

Figure 6 depicts a snapshot that might arise after a customer has identified himself. The two input attributes have values, as do the attributes from module 1. Module 2 is enabled and executing, and module 3 is disabled because the database load is over 95%.

For decision modules such as modules 4, 7, 8, and 9, the cells in positively numbered rows correspond to the attribute rules in the modules. To illustrate, module 4 is enabled, and based on information already known the conditions of the first and third rules could be evaulated, and were false. No information can be inferred about the second rule. With module 7 (that corresponds to the decision module described in Example 3.1), the first rule has false condition, the second and fourth rules have true condition and returned values 1 and 3. Although we have no information about the value of the condition of the third rule, we do know that if true then the rule would contribute the value 2. Based on the combining semantics the system has inferred that the final value for `session_priority` is 3.

We have made a prototype implementation of the DecisionSheet interface on top of ThinkSheet [PYLS96]. The prototype is currently being used to experiment with different functionalities and presentation schemes.

6 RELATED WORK

Recent overviews on workflows and related work include [GHS95, MAGK95, CHRW98, KR96, BT98]. Several commercial workflow products are available, including Flowmark [LR97] which is general-purpose, and InConcert's TEOSS system [InC] which is focused on telecommunication applications. Traditional workflow management systems (WFMS) emphasize on the control and coordination of tasks that may be performed by software systems or humans. To this end, workflows are typically represented using some form of directed graph [GHS95, Ell79], often based on variations of Petri nets. In contrast, recent work in the area of scientific workflows emphasize the need to promote a data flow view of the workflow at the specification level. In [AIL98], the authors advocate for an "object view" of workflow where the focal point is the data used and generated during workflow execution. There, workflows are considered as graphs of objects with the processes that created them being expressed through the links between them. A mixed view is proposed in [MVW95], wherein both data and control flow are expressible and the user can navigate from an activity to its input data structure.

Vortex offers a complementary view through its object-focused approach. In Vortex, the focus is on determining attribute values, and the means by which these values are obtained are modeled essentially as side-effects of this. However, attributes in Vortex can either model the execution status of some task or the output data of a task. This approach offers a lot of flexibility, which enables to alternatively focus on tasks or data as needed by the decision process incarnated by the workflow.

Vortex enables a declarative specification of workflows, which matches the need for high-level specification languages for designing and prototyping workflows usable by non computer trained users (e.g., a natural scientist) emphasized by existing work on scientific workflows. Such high-level specification languages are currently lacking in existing systems. Reference [DKRR98] introduces a declarative paradigm for specifying workflows that is based on Concurrent Transaction Logic, and study the complexity of decision problems involving consistency, verification and scheduling.

The use of enabling conditions in Vortex is reminiscent of their use in the ThinkSheet model [PYLS96]. A key difference between the models is that Vortex permits side-effects.

Another contribution of Vortex at the specification level is to

introduce rules to specify the computation of an attribute in a node of the workflow. In traditional workflows, code blocks (that may invoke external components) are associated with the nodes of the graph, i.e., the tasks. However, the if-then rules used in Vortex are different from the existing three broad categories of rules-based specification paradigms, i.e., (a) logic programming [Llo87] and datalog (see [AHV95]), (b) expert systems (e.g., OPS5 [For81]), and (c) active databases [WC95], primarily because the execution semantics for rules in Vortex is very different.

Last, Vortex gives the ability to specify partial workflows, i.e., workflows whose complete specification depends on the result of the current execution of the workflow. This is not possible with existing systems, which cannot dynamically launch the excution of a workflow as a result of the execution of another one and share their states. As explained in [MVW95], there is a substantial need for this capability in scientific workflow applications.

From the perspective of fundamentals the area of workflow is still in its infancy. There is little consensus on models and languages appropriate for specifying workflows, and only a little research on theoretical properties. The Mentor model [WWWD96] is based on a variant of statecharts [Har88]. In [MWW+98] techniques from model-checking are used to develop static analysis algorithms for verifying various properties of Mentor workflows. Reference [WW97] develops an approach for converting a logically centralized workflow into a workflow that can be distributed across multiple processing engines.

7 FUTURE DIRECTIONS

A key direction is to develop an engine that can process Vortex workflows for near-realtime workflow applications, and support modifications of workflows In work related to this goal we recently implemented a prototype execution engine for *flowchart-based sequential* workflows even as an interpreter of the workflow schema. More specifically, a workflow schema is represented as a collection of objects (corresponding to the different nodes of the flowchart), and the engine interprets these objects as it executes one or more instances of the workflow. Multiple workflow schemas (i.e., flowcharts) can be maintained simultaneously. Adding a new workflow schema involves only the addition of new set of objects, and can be performed at any time without bringing down the engine nor interrupting the execution of workflow instance/s that are already underway. Secondly, to provide near-realtime processing, the engine uses a multi-threading approach, allocating one or more threads to each workflow instance/. An event handler is used to react to the different kinds of events that may arise, either as modules return values, or as a modified version of the workflow schema has been introduced.

The engine will be centered around the use of choice functions as discussed in § 4. Some further work in this direction is described in [HLS+99], where we develop algorithms for identifying at runtime modules that are "useless" for completing a workflow instance and modules that are "necessary" for completing a workflow instance.

There are many interesting issues that deserve further investigation. One is the development of a GUI to support the specification and browsing of Vortex workflows. Another is to develop a way to respond automatically to violations of logical constraints. The third is to develop support for different granularities of Vortex workflows. The fourth is to explore the incorporation of hard time constraints into enabling conditions. The fifth concerns exploitation of the mathematical properties of Vortex schemas. Preliminary results reported in [HLS+99] show that most decision problems for Vortex workflows are intractable or undecidable. Nevertheless, because Vortex workflows are declarative, the development of sufficient conditions for checking properties of Vortex schemas may be easier than for procedural workflow paradigms. The final direction concerns the possibility of developing an algebra for combining Vortex workflow schemas.

REFERENCES

[AHV95] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, Reading, MA, 1995.

[AIL98] A. Ailamaki, Y. Ioannidis, and M. Livny. Scientific workflow management by database management. In *Proc. Int. Conf. on Scientific and Statistical Database Management*, 1998.

[BBKV88] F. Bancilhon, T. Briggs, S. Khoshafian, and P. Valduriez. FAD, a powerful and simple database language. In *Proc. Intl. Conf. on Very Large Data Bases*, pages 97–105, 1988.

[BT98] G.A. Bolcer and R.N. Taylor. Advanced workflow management technologies. Tech. Rep., Info. & Comp. Sci., UC Irvine, 1998.

[BTS91] V. Breazu-Tannen and R. Subrahmanyam. Logical and computational aspects of programming with sets/bags/lists. In *Proc. Int. Colloq. on Automata, Languages and Programming*, Madrid, Spain, July 1991.

[CHRW98] A. Cichocki, A. Helal, M. Rusinkiewicz, and D. Woelk. *Workflow and Process Automation: Concepts and Technology*. Kluwer Academic Publishers, Boston, 1998.

[DKRR98] H. Davulcu, M. Kifer, C.R. Ramakrishnan, and I.V. Ramakrishnan. Logic based modeling and analysis of workflows. In *Proc. ACM Symp. on Principles of Database Systems*, pages 25–33, 1998.

[Ell79] C. A. Ellis. Information control nets: A mathematical model of office information flow. In *ACM Proc. Conf. Simulation, Modeling and Measurement of Computer Systems*, pages 225–240, August 1979.

[For81] C. L. Forgy. OPS5 user's manual. Technical Report CMU-CS-81-135, Carnegie Mellon University, 1981.

[GHS95] D. Georgakopoulos, M. Hornick, and A. Sheth. An overview of workflow management: From process modeling to workflow automation infrastructure. *Distributed and Parallel Databases*, 3(2):119–154, April 1995.

[Har88] D. Harel. On visual formalisms. *Comm. of the ACM*, 31(5), 1988.

[HLS+99] R. Hull, F. Llirbat, J. Su, G. Dong, B. Kumar, and G. Zhou. Adaptive execution of workflow: Analysis and optimization. Tech. Rep., Bell Labs, 1999.

[InC] Inc. InConcert. Inconcert web site (www.inconcert.com).

[KR96] M. Kamath and K. Ramamritham. Correctness issues in workflow management. *Distributed Systems Engineering (DSE) Journal*, 3(4), December 1996.

[Llo87] J. W. Lloyd. *Foundations of Logic Programming (Second Edition)*. Springer-Verlag, Berlin, 1987.

[LR97] F. Leymann and D. Roller. Workflow-based applications. *IBM Systems Journal*, 36(1), 1997.

[MAGK95] C. Mohan, G. Alonso, R. Guenthoer, and M. Kamath. Exotica: A research perspective on workflow management systems. *Data Engineering Bulletin, Special Issue on Infrastructure for Business Process Management*, 18(1), March 1995.

[MVW95] C. Medeiros, G. Vossen, and M. Weske. Wasa: a workflow-based architecture to support scientific database applications. In *Proc. 6th DEXA Conference*, 1995.

[MWW+98] P. Muth, D. Wodtke, J. Weissenfels, G. Weikum, and A. Kotz Dittrich. Enterprise-wide workflow management based on state and activity charts. In A. Dogac, L. Kalinichenko, T. Özsu, and A. Sheth, editors, *Workflow Management Systems and Interoperability*. Springer-Verlag, 1998.

[OBBT89] A. Ohori, P. Buneman, and V. Breazu-Tannen. Database programming in Machiavelli: A polymorphic language with static type inference. In *Proc. ACM SIGMOD*, 1989.

[PSV92] S. Parker, E. Simon, and P. Valduriez. Svp, a model capturing sets, streams, and parallelism. In *Proc. Int. Conf. on Very Large Data Bases*, Vancouver, Canada, Aug 1992.

[PYLS96] P. Piatko, R. Yangarber, D. Lin, and D. Shasha. Thinksheet: A tool for tailoring complex documents. in *Proc. ACM SIGMOD*, 1996.

[WC95] J. Widom and S. Ceri. *Active Database Systems: Triggers and Rules for Advanced Database Processing*. Morgan-Kaufmann, Inc., San Francisco, California, 1995.

[WW97] D. Wodtke and G. Weikum. A formal foundation for distributed workflow execution based on state charts. In *Proc. Int. Conf. on Database Theory*, pages 230–246, 1997.

[WWWD96] D. Wodtke, J. Weissenfels, G. Weikum, and A. K. Dittrich. The Mentor project: Steps towards enterprise-wide workflow management. In *Proc. IEEE Int. Conf. on Data Engineering*, 1996.