Ensuring Consistency in Long Running Transactions*

Jeffrey Fischer and Rupak Majumdar
Dept. of Computer Science, University of California, Los Angeles
Los Angeles, California, USA
fischer@cs.ucla.edu, rupak@cs.ucla.edu

ABSTRACT

Flow composition languages permit the construction of long-running transactions from collections of independent atomic services. Due to environmental limitations, such transactions usually cannot be made to conform to standard ACID semantics. We propose *set consistency*, a powerful, yet intuitive, notion of consistency for long-running transactions. Set consistency considers the collection of permanent (non-intermittent) changes made by a process, when viewed at the end of its execution. Consistency requirements for such collections of changes are specified as predicates over the atomic actions of a process. Set consistency generalizes *self-cancellation*, a standard consistency requirement for long-running transactions, where failed processes are responsible for undoing any partially completed work. Set consistency can also express strictly stronger requirements, such as mutual exclusion or dependency.

We show that the set consistency verification problem for processes is co-NP complete and present an algorithm for verifying set consistency by reduction to propositional validity. We have implemented this algorithm and demonstrate the value and tractability of our approach on three real-world case studies. In each case, the consistency requirements can be verified within a second, demonstrating the practicality of our approach.

Categories and Subject Descriptors: D.2.4 [Software Engineering]: Software/Program Verification

General Terms: Verification, Languages

Keywords: transactions, semantics, compensation, business process, flow composition, set consistency

1. INTRODUCTION

In Web Services and Service Oriented Architectures, flow composition and orchestration languages [13, 1, 2] are used to combine disparate services into composite applications. Unlike monolithic applications built on a single database, these applications cannot take advantage of atomic transactions to ensure that changes are not partial or lost completely. Instead, they generally recover using *compensation* — actions that are applied to undo other previously committed actions. Flow compensation languages may provide *compensation operators* which associate compensation actions with completed actions and automatically run compensation when a failure occurs. Existing research has focused on formalizing the semantics of compensation [7, 8, 9, 5, 4] or on verifying interactions between processes [6, 15, 14]. Here, we define a notion of correctness appropriate for a long-running transaction, focusing on the permanent changes made by the transaction.

Standard, atomic transactions have a well-defined, application independent, criterion for correctness: the ACID (Atomicity, Consistency, Isolation, and Durability) properties [17]. However, ACID transactions are not suitable for flow composition. First, a transaction which composes web services is usually *long-running* — it takes a substantial amount of time to complete, because of asynchronous interactions across many systems, possibly requiring human intervention. Thus, the holding of locks and tight coordination of the participating systems is not feasible. Second, web services rarely expose distributed transaction interfaces, due to organizational boundaries, protocol limitations, and application limitations. Therefore, one must look for looser notions of correctness.

One such notion is *cancellation semantics* [9]: a long-running transaction should either complete successfully or undo all its observable results. The undo mechanism is a programmer-specified *compensation* action. For example, *sagas* [16] are two-level transactions with compensation actions for each forward action.

Unfortunately, cancellation semantics is insufficient for many real-world scenarios. First, due to application requirements or the limitations of the actual systems, a process may contain non-compensatable actions and may need to continue execution even if certain actions fail. Thus, it is important to define a notion of correctness which accounts for "partial completion" of processes. Second, cancellation semantics cannot represent many application specific requirements, such as the mutual exclusion of two actions.

Alternatively, one may use expressive temporal logics, such as Linear Temporal Logic (LTL), to specify desired process properties. For many business processes, the particular temporal *sequence* of events is less important than the *set* of events that have executed at the end of a run, so LTL

*This research is sponsored in part by the NSF grants CCF-0427202, CCF-0546170, CCF-0702743, and CNS-0720881.

ASE'07, November 5–9, 2007, Atlanta, Georgia, USA.

is often overkill. Moreover, checking temporal properties of compensating processes is undecidable [12], and checking even finite systems is expensive (PSPACE-complete [22]). In practice, the number of states may explode due to compensation, nondeterminism, and parallelism, which frequently occur in business processes.

To address these limitations, we define a simple notion of correctness for long running transactions, called *set consistency*. A set consistency specification captures the set of services which made observable changes, when viewed at the *end* of a process's execution. We show how such a specification can be compactly represented using propositional logic, and how it may be verified by tracking, for all traces, the set of actions which completed successfully and were not undone through compensation. Set consistency specifications can represent cancellation semantics, as well as both stronger and weaker restrictions on processes. For example, they can specify behaviors which relax the self-cancellation requirement.

Set consistency can capture many requirements currently modeled using temporal logic. In most situations, a set consistency specification will be more compact (and, we believe, easier to understand) than the corresponding temporal logic specification. This is because: 1) we treat non-invocations, atomic failure, and compensated invocations as equivalent, avoiding the need to specify each separately, and 2) explicit order dependencies do not need to be specified. Moreover, this restriction in input makes the complexity of verification lower (co-NP complete rather than PSPACE-complete).

We formalize the notion of set consistency using a core calculus for process composition. Our core language composes atomic actions using sequential and parallel composition, choice, compensation, and exception handling constructs, and is an abstraction of common flow composition languages such as BPEL. Then, the *set consistency verification problem* checks that all feasible executions of a process are contained in the set of executions defined by the set consistency specification. We show that this problem is co-NP complete and present an algorithm for verifying set consistency by constructing a predicate representing all feasible process executions. This reduces the verification problem to propositional validity which can be checked using an off-the-shelf SAT solver.

We have implemented the algorithm for set consistency verification and applied it to three case studies, including a real-world CRM system which contained a set consistency bug and was the inspiration of our work (see Section 2). The verification problems resulting from our case studies can each be discharged within a second, showing that the formalism provides an intuitive yet expressive formalism for real-world properties which is also tractable for real-world applications.

2. EXAMPLE

We now informally describe *set consistency* using an example inspired by a bug actually seen in a production system. One of the authors of this paper previously worked for an enterprise applications vendor. Once, when visiting a customer (a large bank), the author reviewed a set of the customer's business processes, which integrated a mainframe-based financial application with a CRM (Customer Relationship Management) application. When an account was created or changed in the financial application, a message containing the account was sent to the CRM system via a transactional queue. The CRM system took the message from the queue and updated its version of the account accordingly. When examining the business process run by the CRM system, the author found a consistency bug, which inspired the work in this paper.

Upon taking a message from the queue, the CRM system executed a business process which transformed the message to its internal account representation, performed other bookkeeping, and saved the account. If the account was saved successfully, the CRM system should commit its transaction with the queue, to avoid duplicate messages. If the account was not saved, the CRM system should abort its transaction with the queue, to avoid losing the update message. Upon the abort of a message "take", the message was automatically put back on the queue, unless a retry threshold was exceeded. In this case, the message went to a special "dead-letter" queue, to be investigated by an administrator. We can model the interactions of the CRM system with the queue as the following process:

$$AcctRecv = \mathbf{TakeMsg}; (body \rhd (\mathbf{LeaveOnQ}; \mathsf{throw})); \mathbf{CommitQ}$$

Here, **bold fonts** represent atomic actions (the implementation of *body* is not shown). The ";" operator composes two actions or processes sequentially: it runs the first process, and if it is successful, runs the second. The "$\rhd$" operator catches exceptions: it runs the first process, and, only if it fails, runs the second. throw is a special built-in action which throws an exception. The **TakeMsg** process takes a message off the queue. The subprocess *body* handles each account. If *body* fails, then the message transaction is aborted (by calling **LeaveOnQ**), putting the message back on the queue. Otherwise, **CommitQ** commits the message transaction, permanently removing it from the queue. Ignoring all the implementation details of *body*, we want to ensure that, if the action **SaveAcct** is called within *body*, then **CommitQ** is called, and if **SaveAcct** is not called, then **LeaveOnQ** is called.

Deep within the *body* subprocess, the author found the equivalent of the following code:

$$\mathbf{SaveAcct} \rhd \mathbf{LogErr}$$

where **SaveAcct** performs the actual account write. Someone had added an exception handler which, in the event of an error in saving, logged a debug message. Unfortunately, they left this code in their production system. The exception handler "swallows" the error status — **CommitQ** will always be called, even if the save fails, violating our correctness requirement. If, due to a data validation error or a transient system problem, the CRM system rejects an account, bank tellers will not be able to find the customer!

Traces. Executing a process gives rise to a trace. A *trace* is the sequence of successfully executed actions invoked by the process, along with the final status of the run (either ✓ or ×). For example, if the **SaveAcct** action fails, we get the following trace:

$$T = \mathbf{TakeMsg}, \mathbf{Preprocess}, \mathbf{LogErr}, \mathbf{CommitQ}\langle\checkmark\rangle$$

The **Preprocess** action represents the preprocessing which occurs in *body* before **SaveAcct** is called. The error from **SaveAcct** is not propagated to the outer process, and thus the queue transaction is committed. Note that the failed invocation of **SaveAcct** does not appear between **Preprocess** and **LogErr**. We leave out failed invocations as they have no permanent, observable effect on the system.

`Atomic Action`	`A`	`::=`	$\mathsf{skip} \mid \mathsf{throw}$	built-ins
			$\mid$ $\mathtt{A}_i \in \mathcal{A}$	defined actions
`Process`	`P`	`::=`	`A`	atomic actions
			$\mid$ `P; P`	sequence
			$\mid$ $\mathtt{P} \parallel \mathtt{P}$	parallel
			$\mid$ $\mathtt{P} \,\Box\, \mathtt{P}$	choice
			$\mid$ $\mathtt{P} \div \mathtt{P}$	compensation
			$\mid$ $\mathtt{P} \rhd \mathtt{P}$	exception handler

Figure 1: Syntax of process calculus.

Set Consistency. Our first observation is that the bug in *AcctRecv* can be caught if we know that there exists a feasible trace in which **CommitQ** is executed and **SaveAcct** is either not called or is invoked, but failed. That is, we can abstract away the relative order of individual actions, and only consider the *set* of actions that executed successfully. Accordingly, we define an *execution* as the set of actions which appear in a trace. The execution E associated with the above trace is:

$$E = \{\mathbf{TakeMsg}, \mathbf{Preprocess}, \mathbf{LogErr}, \mathbf{CommitQ}\}$$

A *set consistency specification* defines a set of "good" executions. For example, correct executions of our account process either:

1. include both **SaveAcct** and **CommitQ**, or
2. include **LeaveOnQ**, but not **SaveAcct**.

We write set consistency specifications in a predicate notation, where the literal a means that action a is included in the execution, and $\neg a$ means that action a is not included in the execution and literals are combined using the boolean $\wedge$ (and), $\vee$ (or), and $\neg$ (not) operators. A *set consistency predicate* is interpreted over the actions which appear in the process. It represents a set consistency specification which contains exactly those executions that satisfy the predicate. To represent the above two conditions, we can define the specification predicate φ_{q1} as follows:

$$\varphi_{q1} = (\mathbf{SaveAcct} \wedge \mathbf{CommitQ})\ \vee (\neg\mathbf{SaveAcct} \wedge \mathbf{LeaveOnQ})$$

Any actions not included in the predicate are left unconstrained. Other consistency predicates can be specified for our process. For example, we might want to ensure that **LeaveOnQ** and **CommitQ** are never called in the same run:

$$\varphi_{q2} = (\mathbf{LeaveOnQ} \wedge \neg\mathbf{CommitQ}) \vee (\neg\mathbf{LeaveOnQ} \wedge \mathbf{CommitQ})$$

We can check both requirements simultaneously by taking the conjunction of the two predicates: $\varphi_{q3} = \varphi_{q1} \wedge \varphi_{q2}$.

Verification. The *set consistency verification problem* takes as input a process P and a consistency predicate φ and asks if all feasible executions of P satisfy φ. If so, we say that the process *satisfies* the specification. Clearly, the process *AcctRecv* does not satisfy the specification φ_{q1} — a counterexample is the execution E above, which clearly does not satisfy φ_{q1}. However, the process does meet the specification φ_{q2}.

In Section 4, we show that one can check a specification by constructing a predicate ϕ_p that represents all the feasible executions for the process P. Then, P satisfies the specification φ if ϕ_p implies φ, which is a propositional satisfiability check. We have built a verifier, described in Section 6, that verifies set consistency by this method. If we run *AcctRecv* through our verifier, using specification φ_{q1} or φ_{q3}, it returns the execution E above as a counterexample.

Note that we can fix the problem by either removing the exception handler in *body* or re-throwing the exception after calling **LogErr**. With either of these fixes, the process passes our verifier.

Other Features. In practice, business process flows contain, in addition to sequencing and exception handling, parallel composition (where processes execute in parallel) and *compensations* (processes that get executed to undo the effect of atomic actions should a subsequent process fail). Our process language (and verifier) allows us to write both parallel composition and compensation actions.

As an example, consider an alternative version of *body* which saves the account in two steps. First, it writes a header using the action **SaveHdr** and then writes contact information using the action **AddContact**. Both of these actions can fail. If **SaveHdr** succeeds, but **AddContact** fails, we undo the account change through the compensating action **DelHdr**, which never fails. In our process language, we specify a compensation action using the "÷" operator. Our new process is written as:

$$\mathit{AcctRecv}_2 = \mathbf{TakeMsg}; (\mathit{body}_2 \rhd (\mathbf{LeaveOnQ}; \mathsf{throw})); \mathbf{CommitQ}$$
$$\mathit{body}_2 = (\mathbf{SaveHdr} \div \mathbf{DelHdr}); \mathbf{AddContact}$$

A set consistency requirement for $\mathit{AcctRecv}_2$ states that if both the account header and contact actions are successful, then **CommitQ** should be run. Otherwise, **LeaveOnQ** should be run. That is,

$$(\mathbf{SaveHdr} \wedge \mathbf{AddContact} \wedge \mathbf{CommitQ}) \vee$$
$$(\neg\mathbf{SaveHdr} \wedge \neg\mathbf{AddContact} \wedge \mathbf{LeaveOnQ})$$

The negated action $\neg$**SaveHdr** captures the scenarios where either **SaveHdr** is not run, or it is run but subsequently compensated by **DelHdr**. We capture the effect of compensations using a programmer-defined *normalization set* $\mathcal{C} = \{(\mathbf{SaveHdr}, \mathbf{DelHdr})\}$ that encodes that the effect of running **SaveHdr** can be undone by running its compensation **DelHdr**. Given the predicate and the normalization set, our verifier automatically expands the predicate to

$$(\mathbf{SaveHdr} \wedge \neg\mathbf{DelHdr} \wedge \mathbf{AddContact} \wedge \mathbf{CommitQ}) \vee$$
$$((\neg\mathbf{SaveHdr} \vee (\mathbf{SaveHdr} \wedge \mathbf{DelHdr})) \wedge$$
$$\neg\mathbf{AddContact} \wedge \mathbf{LeaveOnQ})$$

This expanded specification predicate is then used by our algorithm, which shows that process $\mathit{AcctRecv}_2$ satisfies this set consistency specification.

3. PROCESS CALCULUS

We use a simple process calculus to describe long-running transactions. The operators in our calculus correspond to similar concrete operations in web service orchestration languages like BPEL.

Syntax. Figure 1 defines the syntax for our language. Processes are constructed from *atomic actions*, using a set of composition operations. Atomic actions are indivisible operations which either succeed completely or fail and undo all state changes. There are two built-in actions: skip, which always succeeds and does nothing, and throw, which throws an exception. We use $\mathcal{A}$ for the set of atomic actions defined by the environment and Σ for the set of all atomic actions: $\Sigma \equiv \mathcal{A} \cup \{\mathsf{throw}, \mathsf{skip}\}$.

A process is either an atomic action or a composition of atomic actions using one of five composition operators. The sequence operator ";" runs the first process followed by the second. If the first fails, the second is not run. The parallel operator "∥" runs two processes in parallel. The choice operator "$\Box$" non-deterministically selects one of two processes and runs it. The compensation operator "$\div$" runs the left process. If it completes successfully, the right process is installed as a compensation to run if the parent process terminates with an error. The exception handler "$\rhd$" runs the left process. If that process terminates with an error, the error is ignored and the right process is run. If the left process terminates successfully, the right process is ignored. Our core language does not include iteration operators —we will add iteration to our language in Section 5.

In our examples, we also use *named subprocesses* — subprocesses which are defined once and then appear as atomic actions in the overall process — as syntactic sugar to enhance readability. Named subprocesses are not true functions — they are simply inlined into their parent. Thus, recursive calls are not permitted.

We define $|P|$, the size of process P, by induction: the size $|A|$ of an atomic action in Σ is 1, and the size $|P_1 \otimes P_2|$ for any composition operation $\otimes$ applied to P_1 and P_2 is $|P_1| + |P_2| + 1$.

Trace Semantics. We now define a trace-based semantics for processes. A *run* is a (possibly empty) sequence of atomic actions from Σ. A *trace* is a run followed by either $\langle\checkmark\rangle$ or $\langle\times\rangle$, representing successful and failed executions, respectively. For example, if action A_1 is run successfully and then action A_2 fails, the corresponding trace would be $A_1\langle\times\rangle$. In the following, we let the variable A range over atomic actions; the variables P, P', Q, Q', and R range over processes; the variables p, q, and r range over runs, and the variables s and t range over $\checkmark$ and $\times$.

We define an operator "&" which combines two process status symbols ($\checkmark$, $\times$). Given $s\&t$, if both s and t are $\checkmark$, then $s\&t = \checkmark$. Otherwise, $s\&t = \times$. For the parallel composition rules, we introduce an operator $\bowtie$: $\mathcal{R} \times \mathcal{R} \rightarrow 2^{\mathcal{R}}$, where, given two runs p and q, $p \bowtie q$ produces the set of all interleavings of p and q.

We use the symbol Γ to represent an *action type environment*, which maps each action to a set of possible results $2^{\{\checkmark,\times\}} = \{\{\checkmark\}, \{\times\}, \{\checkmark, \times\}\}$ from running the associated action. This allows us to distinguish actions which may fail from actions which never fail. The result set $\{\times\}$ is for the special action throw, which unconditionally throws an error.

For each process form P, we define using mutual induction two semantic functions $\Pi(P)$ and $[\![P]\!]$. The rules for $\Pi : \mathcal{P} \rightarrow 2^{(\mathcal{T},\mathcal{P})}$, found in figure 2, define a set of pairs. Each pair $(p\langle s\rangle, Q) \in \Pi(P)$ consists of a trace $p\langle s\rangle$, representing a possible execution of the process P, and a process Q, representing a *compensation process* associated with the trace that will get run on a subsequent failure.

The function $[\![P]\!] : \mathcal{P} \rightarrow 2^{\mathcal{T}}$, maps a process P to a set of feasible traces of P. To compute the actual traces possible for a top-level process, this function must consider the successful and failed traces independently. Compensation processes are dropped from successful traces. For a failed trace $p\langle\times\rangle$, one computes all possible compensation traces for the associated compensation process P' and appends these to p. $[\![P]\!]$ is defined as follows:

$$
\begin{aligned}
[\![P]\!] = \{(p\langle\checkmark\rangle | (p\langle\checkmark\rangle, P') \in \Pi(P)\} \cup \\
\{pp'\langle\times\rangle | (p\langle\times\rangle, P') \in \Pi(P), p'\langle s\rangle \in [\![P']\!]\}
\end{aligned}
$$

We now describe the rules in Figure 2 in more detail.

Atomic Actions. For atomic actions, we enumerate the possible results for each action. Individual atomic actions use skip as a compensation process, since no compensation is provided by default. Compensation for an atomic action must be explicitly defined using the $\div$ operator.

Sequential Composition. For a sequential composition $P; Q$, we consider two cases. If P succeeds, we have a successful trace $p\langle\checkmark\rangle$ for P, after which we concatenate a trace q from Q. The status for the overall trace is then $\langle s\rangle$, the status from the trace q. The compensation process (if Q fails) invokes the compensation for Q first, followed by the compensation for P. On the other hand, if P fails, we have a failed trace $p\langle\times\rangle$ for P. The process Q is not run at all.

Parallel Composition. For parallel composition, we first consider the case where both sub-processes run. If both are successful, the entire process is successful. If one fails, the process throws an error. We simulate the parallel semantics by generating a possible trace for all interleavings of the two subprocesses. The compensation for the two sub-processes is also run in parallel. Note that, if a sub-process fails, the other sub-process runs to completion, unless it also encounters an error. However, if the second sub-process has not started yet, and the first fails, an implementation can avoid running the second at all. This is handled by the last two sets in the union.

Choice Composition. The traces for $P \Box Q$ are simply the union of the traces for P and Q.

Compensation. The compensation operator $P \div Q$ runs P and then sets up process Q as compensation. If P is successful, Q overrides any previous compensation for P — e.g., P' if $\Pi(P) = (p\langle\checkmark\rangle, P')$. If P fails, then the original compensation process returned by $\Pi(P)$ is instead returned. In this case, the process Q is never run.

Exception Handler. The rule for the exception handling operator has two scenarios. Given $P \rhd Q$, if P is successful, Q is ignored. If P fails, the compensation for P is run and then process Q is run. To express this, we use $[\![P]\!]$ to obtain a full trace for P, including compensation. This makes the $[\![\cdot]\!]$ and Π functions mutually recursive.

Example 1. We consider the set of feasible traces for the *SimpleOrder* process, defined as:

$$
\begin{aligned}
\mathit{SimpleOrder} =& \mathit{Billing}; \textbf{ProcessOrder} \\
\mathit{Billing} =& \textbf{Charge} \div \textbf{Credit}
\end{aligned}
$$

We assume that the **Charge** and **ProcessOrder** atomic actions both can fail, but the **Credit** compensation never fails. Thus, we obtain the following definition for Γ:

$$\langle \textbf{Charge} \mapsto \{\checkmark\ \times\},\ \textbf{Credit} \mapsto \{\checkmark\}, \textbf{ProcessOrder} \mapsto \{\checkmark, \times\}\rangle$$

The set of feasible traces for this process is:

$$\{\textbf{Charge ProcessOrder}\langle\checkmark\rangle, \textbf{Charge Credit}\langle\times\rangle, \mathsf{skip}\langle\times\rangle\}$$

4. SET CONSISTENCY

We can now formalize our notion of consistency with respect to our process calculus.

Executions. For a trace $p\langle s\rangle \in [\![P]\!]$, we define the *execution* for $p\langle s\rangle$ as the set $\pi_p \subseteq \Sigma$ of atomic actions that appear in p, that is, the execution $e(p) = \{a \in \Sigma \mid \exists p_1, p_2. p\langle s\rangle \equiv p_1 a p_2\langle s\rangle\}$. For a process P, let

Process form	Π
$\Gamma \vdash A : \{\checkmark, \times\}$	$\{(A\langle\checkmark\rangle, \mathsf{skip}), (\langle\times\rangle, \mathsf{skip})\}$
$\Gamma \vdash A : \{\checkmark\}$	$\{(A\langle\checkmark\rangle, \mathsf{skip})\}$
$\Gamma \vdash A : \{\times\}$	$\{(\langle\times\rangle, \mathsf{skip})\}$
`P; Q`	$\{(pq\langle s\rangle, Q'; P') \mid (p\langle\checkmark\rangle, P') \in \Pi(P), (q\langle s\rangle, Q') \in \Pi(Q)\} \cup \{(p\langle\times\rangle, P') \mid (p\langle\times\rangle, P') \in \Pi(P)\}$
`P` $\parallel$ `Q`	$\{(r\langle s\&t\rangle, P' \parallel Q') \mid r \in p \bowtie q, (p\langle s\rangle, P') \in \Pi(P), (q\langle t\rangle, Q') \in \Pi(Q)\}$ $\cup \{(p\langle\times\rangle, P') \mid (p\langle\times\rangle, P') \in \Pi(P)\} \cup \{(q\langle\times\rangle, Q') \mid (q\langle\times\rangle, Q') \in \Pi(Q)\}$
`P` $\square$ `Q`	$\{(p\langle s\rangle, P') \mid (p\langle s\rangle, P') \in \Pi(P)\} \cup \{(q\langle t\rangle, Q') \mid (q\langle t\rangle, Q') \in \Pi(Q)\}$
`P` $\div$ `Q`	$\{(p\langle\checkmark\rangle, Q) \mid (p\langle\checkmark\rangle, P') \in \Pi(P)\} \cup \{(p\langle\times\rangle, P') \mid (p\langle\times\rangle, P') \in \Pi(P)\}$
`P` $\rhd$ `Q`	$\{(p\langle\checkmark\rangle, P') \mid (p\langle\checkmark\rangle, P') \in \Pi(P)\} \cup \{(pp'q\langle t\rangle, Q') \mid pp'\langle\times\rangle \in [\![P]\!], (q\langle t\rangle, Q') \in \Pi(Q)\}$

Figure 2: Trace semantics.

$\mathsf{execs}(P) \subseteq 2^{\Sigma}$ represent the set of all executions of P, defined by $\mathsf{execs}(P) = \{e(p) \mid p\langle s\rangle \in [\![P]\!]\}$.

Example 2. If we drop calls to skip, the set of feasible executions $\mathsf{execs}(\mathit{SimpleOrder})$ for *SimpleOrder* is

$$\{\{\mathbf{Charge}, \mathbf{ProcessOrder}\}, \{\mathbf{Charge}, \mathbf{Credit}\}, \emptyset\}$$

Set Consistency Specifications. A *set consistency specification* $\mathcal{S} \subseteq 2^{\Sigma}$ is a set of action sets representing the permissible executions for a given process. A process P is *set consistent* with respect to a set consistency specification $\mathcal{S}$ if all executions of P fall within the set consistency specification: $\mathsf{execs}(P) \subseteq \mathcal{S}$.

Set consistency specifications are semantic objects. We use a boolean predicate-based syntax for describing sets of executions. Given a set of atomic actions Σ, a *set consistency predicate* is an expression built from combining atomic predicates Σ with the logical operators $\neg$ (not), $\wedge$ (and), and $\vee$ (or). The size $|\varphi|$ of predicate φ is the defined by induction: $|a| = 1$ for a literal, and $|\varphi_1 \otimes \varphi_2| = |\varphi_1| + |\varphi_2| + 1$ for a logical operator $\otimes$. To evaluate a predicate φ on an execution $e \in 2^{\Sigma}$, we assign the value `true` to all atomic actions that occur in e and `false` to all the atomic actions that do not occur in e. If the predicate φ evaluates to `true` with these assignments, we say the execution e *satisfies* the specification φ, and write $e \models \varphi$. The set consistency specification $\mathcal{S}$ defined by the set consistency predicate φ is the set of satisfying assignments of φ:

$$\mathsf{spec}(\varphi) = \{e \in 2^{\Sigma} \mid e \models \varphi\}$$

Normalization. When defining set consistency specifications, we wish to treat the compensated execution of an action as equivalent to an execution where the action (and its compensation) never occurs. If A° is a compensation action for A, then the term $\neg A$ in the specification predicate should yield two executions in the final, expanded specification: one with neither A nor A°, and one with both A and A°. We call a specification that has been adjusted in this manner a *normalized* specification. Normalization is performed with respect to a programmer-specified *normalization set* $\mathcal{C} \subseteq \{(a, a^{\circ}) | a, a^{\circ} \in \Sigma\}$ of atomic action pairs, where the second action in each pair is the compensation action for the first action. Given the consistency specification predicate φ and a normalization set $\mathcal{C}$, we apply the function $\mathsf{spec\_norm}(\varphi, \mathcal{C})$. This function uses DeMorgan's laws and the normalization set to convert the predicate to a form where (1) the negation operator only appears in front of literals, and (2) given a pair (a, a') from $\mathcal{C}$, each occurrence of $\neg a$ is replaced with $(\neg a \wedge \neg a') \vee (a \wedge a')$ and each occurrence of a is replaced with $a \wedge \neg a'$.

Example 3. We consider a specification for the *SimpleOrder* process. We assume the action typing Γ_{so} of Example 1 (in which the **Credit** action never fails) and the normalization set $\mathcal{C}_{so} = \{(\mathbf{Charge}, \mathbf{Credit})\}$.

We wish to have cancellation semantics, where either both **Charge** and **ProcessOrder** complete successfully, or, in a failed execution, any completed actions are undone. Given the normalization set, we do not need to distinguish between failure cases. Thus, we can write a set consistency predicate φ_{so} for *SimpleOrder* as:

$$(\mathbf{Charge} \wedge \mathbf{ProcessOrder}) \vee (\neg\mathbf{Charge} \wedge \neg\mathbf{ProcessOrder})$$

We now expand φ_{so} to get $\mathsf{spec\_norm}(\varphi_{so}, \mathcal{C}_{so})$. It is already in the form we need for substitution (negation only of literals). We replace all occurrences of $\neg$**Charge** with $(\neg\mathbf{Charge} \wedge \neg\mathbf{Credit}) \vee (\mathbf{Credit} \wedge \mathbf{Charge})$ and all occurrences of **Charge** with $(\mathbf{Charge} \wedge \neg\mathbf{Credit})$:

$$(\mathbf{Charge} \wedge \neg\mathbf{Credit} \wedge \mathbf{ProcessOrder}) \vee$$
$$((\neg\mathbf{Charge} \wedge \neg\mathbf{Credit}) \vee (\mathbf{Charge} \wedge \mathbf{Credit})) \wedge \neg\mathbf{ProcessOrder}$$

Set consistency verification. For a process P, a normalization set $\mathcal{C}$, and a set consistency predicate φ, the set consistency *verification problem* is to check if all the executions of P w.r.t. $\mathcal{C}$ satisfy φ, that is, if $\mathsf{execs}(P) \subseteq \mathsf{spec}(\mathsf{spec\_norm}(\varphi, \mathcal{C}))$.

THEOREM 1. *The set consistency verification problem is co-NP complete.*

Predicate-based verification. We now give an algorithm for verifying set consistency. We define a syntax-directed analysis which takes a process as input and constructs a predicate ϕ whose satisfying assignments precisely represent the feasible execution set. The predicate, ϕ is composed from atomic predicates Σ using logical operators $\neg$, $\wedge$, $\vee$,

Predicate	$\Gamma \vdash A : \{\checkmark\}$	$\Gamma \vdash A : \{\checkmark, \times\}$	$\Gamma \vdash A : \{\times\}$
$\phi_{\checkmark 0}$	A	A	`false`
$\phi_{\checkmark\checkmark}$	A	A	`false`
$\phi_{\checkmark\times}$	`false`	`false`	`false`
$\phi_{\times 0}$	`false`	$\neg A$	$\neg A$
$\phi_{\times\checkmark}$	`false`	$\neg A$	$\neg A$
$\phi_{\times\times}$	`false`	`false`	`false`
ϕ_{00}	$\neg A$	$\neg A$	$\neg A$

Predicate	**P;Q**	**P‖Q**	**P□Q**
$\phi_{\checkmark 0}$	$P_{\checkmark 0}Q_{\checkmark 0}$	$P_{\checkmark 0}Q_{\checkmark 0}$	$P_{00}Q_{\checkmark 0}\vee\ P_{\checkmark 0}Q_{00}$
$\phi_{\checkmark\checkmark}$	$P_{\checkmark\checkmark}Q_{\checkmark\checkmark}$	$P_{\checkmark\checkmark}Q_{\checkmark\checkmark}$	$P_{00}Q_{\checkmark\checkmark}\vee\ P_{\checkmark\checkmark}Q_{00}$
$\phi_{\checkmark\times}$	$P_{\checkmark 0}Q_{\checkmark\times}\vee\ P_{\checkmark\times}Q_{\checkmark 0}$	$P_{\checkmark 0}Q_{\checkmark\times}\vee\ P_{\checkmark\checkmark}Q_{\checkmark\times}\vee\ P_{\checkmark\times}Q_{\checkmark 0}\vee$ $P_{\checkmark\times}Q_{\checkmark\checkmark}\vee\ P_{\checkmark\times}Q_{\checkmark\times}$	$P_{00}Q_{\checkmark\times}\vee\ P_{\checkmark\times}Q_{00}$
$\phi_{\times 0}$	$P_{\checkmark 0}Q_{\times 0}\vee\ P_{\times 0}Q_{00}$	$P_{00}Q_{\times 0}\vee\ P_{\checkmark 0}Q_{\times 0}\vee\ P_{\times 0}Q_{00}\vee$ $P_{\times 0}Q_{\checkmark 0}\vee\ P_{\times 0}Q_{\times 0}$	$P_{00}Q_{\times 0}\vee\ P_{\times 0}Q_{00}$
$\phi_{\times\checkmark}$	$P_{\checkmark\checkmark}Q_{\times\checkmark}\vee\ P_{\times\checkmark}Q_{00}$	$P_{00}Q_{\times\checkmark}\vee\ P_{\checkmark\checkmark}Q_{\times\checkmark}\vee\ P_{\times\checkmark}Q_{00}\vee$ $P_{\times\checkmark}Q_{\checkmark\checkmark}\vee\ P_{\times\checkmark}Q_{\times\checkmark}$	$P_{00}Q_{\times\checkmark}\vee\ P_{\times\checkmark}Q_{00}$
$\phi_{\times\times}$	$P_{\checkmark 0}Q_{\times\times}\vee\ P_{\checkmark\times}Q_{\times\checkmark}\vee\ P_{\times\times}Q_{00}$	$P_{00}Q_{\times\times}\vee\ P_{\checkmark 0}Q_{\times\times}\vee\ P_{\checkmark\checkmark}Q_{\times\times}\vee$ $P_{\checkmark\times}Q_{\times 0}\vee\ P_{\checkmark\times}Q_{\times\checkmark}\vee\ P_{\checkmark\times}Q_{\times\times}\vee$ $P_{\times 0}Q_{\checkmark\times}\vee\ P_{\times 0}Q_{\times\times}\vee\ P_{\times\checkmark}Q_{\checkmark\times}\vee$ $P_{\times\checkmark}Q_{\times\times}\vee\ P_{\times\times}Q_{00}\vee\ P_{\times\times}Q_{\checkmark 0}\vee$ $P_{\times\times}Q_{\checkmark\checkmark}\vee\ P_{\times\times}Q_{\checkmark\times}\vee\ P_{\times\times}Q_{\times 0}\vee$ $P_{\times\times}Q_{\times\checkmark}\vee\ P_{\times\times}Q_{\times\times}$	$P_{00}Q_{\times\times}\vee\ P_{\times\times}Q_{00}$
ϕ_{00}	$P_{00}Q_{00}$	$P_{00}Q_{00}$	$P_{00}Q_{00}$

Predicate	**P÷Q**	**P▷Q**	
$\phi_{\checkmark 0}$	$P_{\checkmark 0}Q_{00}$	$P_{\checkmark 0}Q_{00}\vee\ P_{\times\checkmark}Q_{\checkmark 0}$	
$\phi_{\checkmark\checkmark}$	$P_{\checkmark 0}Q_{\checkmark 0}$	$P_{\checkmark\checkmark}Q_{00}\vee\ P_{\times\checkmark}Q_{\checkmark\checkmark}$	
$\phi_{\checkmark\times}$	$P_{\checkmark 0}Q_{\times\checkmark}\vee\ P_{\checkmark 0}Q_{\times\times}$	$P_{\checkmark\times}Q_{00}\vee\ P_{\times\checkmark}Q_{\checkmark\times}$	
$\phi_{\times 0}$	$P_{\times 0}Q_{00}$	$P_{\times\checkmark}Q_{\times 0}$	
$\phi_{\times\checkmark}$	$P_{\times\checkmark}Q_{00}$	$P_{\times\checkmark}Q_{\times\checkmark}$	
$\phi_{\times\times}$	$P_{\times\times}Q_{00}$	$P_{\times\checkmark}Q_{\times\times}\vee\ P_{\times\times}Q_{00}$	
ϕ_{00}	$P_{00}Q_{00}$	$P_{00}Q_{00}$	

Figure 3: Inference rules for computing ϕ

$\rightarrow$, and $\leftrightarrow$. For the moment, we assume that a given action A is referenced only once in a process. Later, we will extend our approach to remove this restriction. A predicate ϕ over atomic predicates in Σ defines a set of executions $\mathcal{E}(\phi) = \{e \in 2^{\Sigma} \mid e \models \phi\}$.

Execution Predicate Construction. We create the predicate ϕ by recursively applying the rules of Figure 3, based on the form of each sub-process. These rules define seven mutually-recursive functions: $\phi_{\checkmark 0}$, $\phi_{\checkmark\checkmark}$, $\phi_{\checkmark\times}$, $\phi_{\times 0}$, $\phi_{\times\checkmark}$, $\phi_{\times\times}$, and ϕ_{00}, all with the signature $\mathcal{P} \times \mathcal{G} \rightarrow \Phi$, where $\mathcal{P}$ is the set of all processes, $\mathcal{G}$ is the set of all action type environments, and Φ the set of all predicates. The two subscripts of each function's name represent the forward and compensation results of running the process, respectively, where $\checkmark$ is a successful execution, $\times$ is a failed execution, and 0 means that the process was not run. For example, $\phi_{\checkmark\times}(P, \Gamma)$ returns a predicate representing all executions of process P where the forward process completes successfully and the compensation process (whose execution must be initiated by the failure of a containing process) fails, given a type environment Γ. As a shorthand, we use terms of the form $P_{\checkmark\times}$ to represent the function $\phi_{\checkmark\times}(P, \Gamma)$. We also leave out the conjunction symbol ("$\wedge$") when it is obvious from the context (e.g. $P_{\checkmark 0}Q_{\checkmark 0}$ for $P_{\checkmark 0} \wedge Q_{\checkmark 0}$).

Given these functions, we compute the predicate ϕ, representing the possible executions of a process, using the function $\mathsf{pred} : \mathcal{P} \times \mathcal{G} \rightarrow \Phi$, defined as:

$$\mathsf{pred}(P, \Gamma) \equiv \phi_{\checkmark 0}(P, \Gamma) \;\vee\; \phi_{\times\checkmark}(P, \Gamma) \;\vee\; \phi_{\times\times}(P, \Gamma)$$

Example 4. We illustrate the predicate generation algorithm by constructing a predicate for the *SimpleOrder* process. We start from the execution predicate definition:

$$\mathsf{pred}(\mathit{SimpleOrder}, \Gamma_{so}) = \phi_{\checkmark 0} \;\vee\; \phi_{\times\checkmark} \;\vee \phi_{\times\times}$$

and iteratively apply the appropriate sub-predicates from Figure 3. After simplification, we obtain the following final result:

$$\begin{array}{l} \textbf{Charge} \wedge \neg\textbf{Credit} \wedge \textbf{ProcessOrder} \vee \\ \textbf{Charge} \wedge \textbf{Credit} \wedge \neg\textbf{ProcessOrder} \vee \\ \neg\textbf{Charge} \wedge \neg\textbf{Credit} \wedge \neg\textbf{ProcessOrder} \end{array}$$

Note that the three conjunctions in the final predicate correspond to the three possible executions of *SimpleOrder*.

Memoization. As usual, one can memoize each computation by giving names to sub-predicates. While generating the execution predicate, we name each non-atomic subpredicate, using the names as literals instead of expanding the sub-predicates of ϕ. The resulting execution predicate is then conjoined with a definition ($n \leftrightarrow \phi_n$) for each name n and its definition ϕ_n. Although the memoized execution predicate can be larger for some processes, the worst-case size of the memoized predicate is polynomial in the size of a process, whereas the size of a non-memoized predicate can be exponential.

Theorem 2. *For any process P and action type environment Γ,*

1. *The execution set obtained from P's execution predicate is equal to the set of all P executions:* $\mathcal{E}(\mathsf{pred}(P,\Gamma)) = \mathsf{execs}(P,\Gamma)$.
2. $|\mathsf{pred}(P,\Gamma)|$ *is polynomial in* $|P|$.

Checking Consistency. We check a specification by checking if the execution predicate $\mathsf{pred}(P,\Gamma)$ implies the normalized specification predicate $\mathsf{spec\_norm}(\varphi,\mathcal{C})$. If the implication is valid, then all executions satisfy the specification and the solution to the consistency verification problem is "yes." Otherwise, there is some execution that does not satisfy the consistency specification. Therefore, to check a process for consistency, we can build the execution and normalized specification predicates and check the implication by a Boolean satisfiability query.

Theorem 3. *For any process P and specification predicate* φ*, with action type environment* Γ *and normalization set* $\mathcal{C}$*,*

1. $\mathsf{execs}(P,\Gamma) \subseteq \mathsf{spec}(\mathsf{spec\_norm}(\varphi,\mathcal{C}))$ *iff* $\mathsf{pred}(P,\Gamma) \rightarrow \mathsf{spec\_norm}(\varphi,\mathcal{C})$ *is valid.*
2. *The consistency verification problem can be solved in time exponential in some polynomial function of* $|P|$, $|\Gamma|$, $|\varphi|$, *and* $|\mathcal{C}|$.

Example 5. The verification problem for process *SimpleOrder* may be reduced to checking the validity of the predicate $\mathsf{pred}(\mathit{SimpleOrder},\Gamma_{so}) \rightarrow \mathsf{spec\_norm}(\varphi_{so},\mathcal{C}_{so})$.

Multiple Calls to an Action. So far, we have assumed that each action in $\mathcal{A}$ is called at most once in a given process. If an action may be called multiple times by a process, we do not distinguish the individual calls. Given an action A, which appears more than once in the text of the process P, the specification predicate A is `true` if A is called at least once, and `false` if A is never called. This follows directly from our definition of a process execution, which is a *set* of called actions, rather than a *bag*. These semantics are useful for situations where an action may be called in one of several situations, and we wish to verify that, given some common condition, the action is called. For example, in *OrderProcess* of Section 6.1, the action `MarkFailed` must be called if the order fails, either due to a failed credit check or to a failed fulfillment subprocess.

If an action is called more than once, $\mathsf{pred}(P,\Gamma)$ may produce an unsatisfiable predicate. For example, $\phi_{\checkmark 0}(A \,\square\, A) = (A \wedge \neg A) \;\vee\; (\neg A \wedge A)$, which is clearly unsatisfiable. We solve this by applying the function $\mathsf{trans\_calls} : \mathcal{P} \rightarrow \mathcal{P} \times \Phi$, which takes as input a process P and returns a translated process P' along with a predicate ϕ_{tc}. Given a set of actions $\{A^1, ...A^n\} \subseteq \mathcal{A}$, which occur more than once in P, each occurrence of these actions the translated process P' is given a unique integer subscript. For example, given the process $P = (A;B) \,\square\, (A;B)$, $\mathsf{trans\_calls}(P)$ will return the process $P' = (A_1;B_1) \,\square\, (A_2;B_2)$. The predicate $\phi_{tc} = \phi^1_{tc} \wedge ...\phi^n_{tc}$, where ϕ^i_{tc} uses the boolean $\leftrightarrow$ operator to associate the atomic predicate A^i with the disjunction of the predicates for each subscript. For example, ϕ_{tc} for $P = (A;B)\square(A;B)$ will be $(A \leftrightarrow (A_1 \vee A_2)) \wedge (B \leftrightarrow (B_1 \vee B_2))$. We now re-define our predicate function pred, to combine $\mathsf{trans\_calls}$ with $\mathsf{pred}_{\mathsf{mem}}$, our original predicate generation function:

$$\mathsf{pred}(P,\Gamma) = \mathtt{let}\ (P',\phi_{tc}) = \mathsf{trans\_calls}(P)\ \mathtt{in}$$
$$\mathsf{pred}_{\mathsf{mem}}(P',\Gamma)\ \wedge \phi_{tc}$$

Lemma 1. *For any process P, which may contain multiple calls to the same action, and atomic action typing* Γ*, the execution set obtained from applying* pred *to P and* Γ *is equal to the set of all P executions:* $\mathcal{E}(\mathsf{pred}(P,\Gamma)) = \mathsf{execs}(P,\Gamma)$.

Counterexample traces. As a consequence of Theorem 3, if a process does not satisfy a specification, we can obtain a model for the execution predicate which violates the specification predicate. This model corresponds to a counterexample *execution*. Using the process definition and a model for the memoized execution predicate, we can also obtain a counterexample *trace* — an ordered list of actions called by the process, annotated with the success/failure status of each action.

This is accomplished by walking the process definition, simulating its execution per the trace semantics of Section 3. When a choice process is encountered, the model is checked to see which subprocess of the choice has a true value for its ϕ_{00} predicate, indicating that it was not run. The traversal continues with the other subprocess, ignoring the not-chosen subprocess. When an atomic action is reached, the model is checked to see if that action completed successfully. If so, it is added to the trace as a successful action. Otherwise, it is added to the trace as a failed action, and result propagated up through its parent processes.

The model is ambiguous — both non-execution and failure of atomic actions are represented by mappings to `false`. We resolve this ambiguity by ensuring that the traversal only reaches an atomic action if it was attempted. For example, if the first action of a sequence fails, then the rest of the sequence is not traversed. There are a few cases where either failure or non-execution will cause the same future behavior of a process (e.g., the first action of the overall process). For these cases, we assume failure rather than non-execution, unless a given action cannot fail. The computation of a counterexample trace is polynomial with respect to the process size — each subprocess and action is traversed only once.

5. ITERATION

We now extend our core process language to support iteration by introducing two new process composition operators, "$**$" and "$*|$", with the following syntax:

Process	P	::=	...	
			| $**$P	sequential iteration
			| $*\|$P	parallel iteration

The sequential iteration operator runs a process one or more times, one copy at a time. The parallel iteration operator runs one or more copies of a process in parallel. For both operations, the number of copies is not known a priori — this is determined at runtime.

We define the trace semantics of these operators by taking the fixpoint of the following two equations:

$$\Pi(**\,P) = \Pi(P) \;\cup\; \Pi(P;(**\,P))$$
$$\Pi(*|P) = \Pi(P) \;\cup\; \Pi(P \parallel (*|P))$$

The sequential iteration operator generates a sequence of traces from $\Pi(P)$, where all but the last trace must correspond to successful executions. As with pairwise sequential composition, the compensation process for sequential iteration runs the compensation processes for each action sequentially, in reverse order from the forward execution. Parallel

Process	Spec	Proc size	Spec size	Pred size	Time (ms)
AccountReceive	φ_{q1}	13	8	124	70
OrderProcess	φ_{o1}	26	21	247	112
OrderProcess	φ_{o2}	26	20	236	130
BrokenOrder	φ_{o2}	22	20	201	90
Travel	φ_{t1}	13	24	181	90
Travel	φ_{t2}	13	39	210	86

Figure 4: Experimental Results. "Spec" is the consistency specification, "Spec size" is the size of the specification, "Pred size" is the size of the execution predicate.

set iteration interleaves traces of P arbitrarily. If any of these traces fails, the parent has a failed status. As with parallel composition, processes may not be interrupted in the event of a failure, but may be skipped if they have not started. Compensation is run in parallel as well.

We wish to generalize set consistency predicates to processes with iteration. We encounter two problems with iteration. First, the set of potential traces for a given process now becomes infinite. Second, consider two atomic actions A and B, which occur within an iterated subprocess. The specification $A \wedge B$ will be true for all traces where both A and B are called at least once, even if A and B are never called in the same iteration. This is usually too weak. In an executable process, an iterated subprocess would likely be parameterized by some value (dropped when abstracting to our core language), which changes each iteration. Thus, one is more likely interested in checking if, whenever A is called, B is also called within the same iteration. We will see an example of such a process in Section 6.1.

Verification. We generalize specification predicates to cover iterative processes by universally quantifying specifications over all iterations. Such specifications are satisfied if and only if each individual iteration meets the specification. Thus, we can verify an iterative process by replacing iteration operators with their enclosed subprocess and checking the resulting non-iterative process.

6. EXPERIENCES

To demonstrate the viability of our approach, we have implemented a verifier for our process language, based on the predicate-generation algorithm of Section 4. We then modeled the example of Section 2 and two additional case studies in our language and verified them using our tool. The source code for our verifier and examples may be found at `http://cs.ucla.edu/~fischer/code/trans`.

We implemented the predicate generation algorithm in Objective Caml. To determine the validity of verification predicates, we use the MiniSat satisfiability solver [**?**].

Figure 4 shows the results of running our verifier on the example of Section 2 and the two case studies below. The run times are for a 1.6Ghz Pentium M laptop with 512 MB of memory, running Windows XP. The runs all complete in less than 130 milliseconds. The verification predicates are approximately 10 times larger than the original processes. Since process languages are intended for composing lower-level services, the size of real-world processes are usually quite small (in our industrial experience, not more than an order of magnitude larger than our examples). This is well within the capabilities of current SAT solvers.

$$\begin{aligned}
OrderProcess =& \textbf{SaveOrder}; CreditCheck;\\
& \textbf{SplitOrder}; FulfillOrder; CompleteOrder\\
CreditCheck =& (\textbf{ReserveCredit} \rhd (\textbf{Failed}; \mathsf{throw}))\\
& \div\, OrderFailed\\
OrderFailed =& \textbf{RestoreCredit}; \textbf{Failed}\\
FulfillOrder =& *|(ProcessPO_i)\\
ProcessPO_i =& (\textbf{FulfillPO}_i \rhd (\textbf{MarkPOFailed}_i; \mathsf{throw}))\\
& \div\, \mathbf{CancelPO_i}\\
CompleteOrder =& \textbf{BillCustomer}; \textbf{Complete}
\end{aligned}$$

Figure 5: Order management process

6.1 Case Study: Order Process

Butler *et al* [9] presents a simple order fulfillment transaction which has cancellation semantics, and thus compensation is sufficient for expressing the required error handling. Figure 5 shows a more complex order fulfillment transaction, inspired by the example application in [21], which does not have cancellation semantics. The process *OrderProcess* receives a customer's order and makes a reservation against the customer's credit (**ReserveCredit**). If the customer does not have enough credit, the order is marked *failed* (**Failed**) and processing stopped. Otherwise, if the credit check was successful, the subprocess *OrderFailed* is installed as compensation for the credit check. Then, the order is broken down into sub-orders by supplier, and these sub-orders are submitted to their respective suppliers in parallel (subprocess $FulfillOrder$). If all sub-orders complete successfully, the subprocess *CompleteOrder* finalizes the credit transaction with a call to **BillCustomer** and marks the order as *complete* (**Complete**).

Failure semantics. There are two types of errors that may occur in the order transaction. If the *ReserveCredit* call fails (e.g., due to insufficient credit), the order is marked as failed and execution is terminated before submitting any purchase orders. Alternatively, one or more purchase orders may be rejected by the associated suppliers. If any orders fail, the credit reservation is undone and the order marked as failed. Note that neither error scenario causes the entire transaction to be undone. This is consistent with real world business applications, where many transactions have some notion of partial success and records of even failed transactions are retained. We assume the normalization set

$$\{(\textbf{ReserveCredit}, \textbf{RestoreCredit}), (\textbf{FulfillPO}, \textbf{CancelPO})\}$$

and that the actions **SaveOrder**, **RestoreCredit**, **CancelPO**, **Failed**, and **Complete** never fail.

The order should always be saved. If the order process is successful, the customer should be billed, all the purchase orders fulfilled, and the order marked complete. If the order process fails, the order should be marked as failed, the customer should not be billed, and no purchase orders fulfilled. These requirements are written as the following set consistency predicate φ_{o1}:

$$\begin{aligned}
&\textbf{SaveOrder} \wedge\\
&((\textbf{BillCustomer} \wedge \textbf{FulfillPO} \wedge \textbf{Complete} \wedge \neg\textbf{Failed}) \vee\\
&(\neg\textbf{BillCustomer} \wedge \neg\textbf{FulfillPO} \wedge \neg\textbf{Complete} \wedge \textbf{Failed}))
\end{aligned}$$

When checked with our verifier, *OrderProcess* does indeed satisfy this specification.

Next, we consider an alternative, orthogonal specification. Assume that the **ReserveCredit**, **RestoreCredit**, and **BillCustomer** actions all belong to an external credit service. We wish to ensure that our process always leaves the service in a consistent state: if **ReserveCredit** succeeds, then either **RestoreCredit** or **BillCustomer** (but not both) must eventually be called. Also, if **ReserveCredit** fails, neither should be called. We model these requirements with the predicate φ_{o2}:

$$(\neg\mathbf{ReserveCredit} \rightarrow (\neg\mathbf{RestoreCredit} \wedge \neg\mathbf{BillCustomer})) \wedge$$
$$(\mathbf{ReserveCredit} \rightarrow (\mathbf{RestoreCredit} \oplus \mathbf{BillCustomer}))$$

where $\rightarrow$ and $\oplus$ are syntactic sugar for logical implication and logical exclusive-or, respectively. Since we are referencing **RestoreCredit** directly in our specification, we remove the pair $(\mathbf{ReserveCredit}, \mathbf{RestoreCredit})$ from our compensation set. Our verifier finds that *OrderProcess* satisfies this specification.

Finally, we consider the process *BrokenOrder*, a variation of *OrderProcess* where the *OrderFailed* compensation is left out of the *CreditCheck* subprocess:

$$\mathit{CreditCheck} = \mathbf{ReserveCredit} \rhd (\mathbf{Failed}; \mathsf{throw})$$

When checking this process, our verifier finds that the φ_{o2} specification is not satisfied and returns the following counter-example execution:

$$\{\mathbf{SaveOrder}, \mathbf{ReserveCredit}, \mathbf{SplitOrder}, \mathbf{Failed}\}$$

This execution corresponds to a trace where the process runs successfully until it reaches **FulfillPO**, which fails. The exception handling for **FulfillPO** runs **Failed**, but **RestoreCredit** is never run to undo the effects of **ReserveCredit**.

6.2 Case Study: Travel Agency

Many real world applications involve *mixed transactions*. A mixed transaction combines both compensatable and non-compensatable subtransactions [11]. Frequently, these processes involve a *pivot action* [20], which cannot be compensated or retried. To obtain cancellation semantics, actions committing before the pivot must support compensation (backward recovery) and actions committing after the pivot must either never fail or support retry (forward recovery).

Set consistency specifications can capture these requirements and our verifier can check these properties. To illustrate this, we use a travel agency example from [18]. We model it in our core language as follows:

$$\begin{aligned}\mathit{Travel} = ((&\mathbf{BookFlight} \div \mathbf{CancelFlight};\\ &(\mathbf{RentCar} \div \mathbf{CancelCar})) \rhd \mathbf{ReserveTrain});\\ &(\mathbf{ReserveHotel1} \rhd \mathbf{ReserveHotel2})\end{aligned}$$

In this transaction, a travel agent wishes to book transportation and a hotel room for a customer. The customer prefers to travel by plane and rental car. These reservations can be canceled. If a flight or rental car is not available, then the agent will book a train ticket to the destination. Once made, the train reservation cannot be canceled. There are two hotel choices at the destination. The first hotel may be full and a reservation may fail, but the second hotel reservation is always successful. We can model the failure and compensation properties of these services as follows:

$$\begin{aligned}\Gamma_t = \langle &\mathbf{BookFlight} \mapsto \{\checkmark, \times\}, \mathbf{CancelFlight} \mapsto \{\checkmark\},\\ &\mathbf{RentCar} \mapsto \{\checkmark, \times\}, \mathbf{CancelCar} \mapsto \{\checkmark\},\\ &\mathbf{ReserveTrain} \mapsto \{\checkmark, \times\},\\ &\mathbf{ReserveHotel1} \mapsto \{\checkmark, \times\}, \mathbf{ReserveHotel2} \mapsto \{\checkmark\}\rangle\\ \mathcal{C} = \{&(\mathbf{BookFlight}, \mathbf{CancelFlight}), (\mathbf{RentCar}, \mathbf{CancelCar})\}\end{aligned}$$

The **ReserveTrain** action is a pivot action, as it has no compensation or exception handler. From inspection, we see that the requirements for cancellation semantics are met:

- If **ReserveTrain** is called, then the actions **BookFlight** and **RentCar** have been compensated by **CancelFlight** and **CancelCar**, respectively.
- If **ReserveHotel1** fails, we recover forward by calling the alternate action **ReserveHotel2**, which cannot fail.

We can check this with our verifier using the following specification predicate:

$$\begin{aligned}\varphi_{t1} = (((&\mathbf{BookFlight} \wedge \mathbf{RentCar}) \vee \mathbf{ReserveTrain}) \wedge\\ &(\mathbf{ReserveHotel1} \vee \mathbf{ReserveHotel2})) \vee\\ &\neg(\mathbf{BookFlight} \vee \mathbf{RentCar} \vee \mathbf{ReserveTrain} \vee\\ &\mathbf{ReserveHotel1} \vee \mathbf{ReserveHotel2})\end{aligned}$$

The process does indeed satisfy this specification. We can use consistency specifications to check stronger properties as well. For example, we can alter the specification predicate to check that the process does not book both the flight/car and the train and that it only books one hotel:

$$\begin{aligned}\varphi_{t2} = (((&\mathbf{BookFlight} \wedge \mathbf{RentCar}) \oplus \mathbf{ReserveTrain}) \wedge\\ &(\mathbf{ReserveHotel1} \oplus \mathbf{ReserveHotel2})) \vee\\ &\neg(\mathbf{BookFlight} \vee \mathbf{RentCar} \vee \mathbf{ReserveTrain} \vee\\ &\mathbf{ReserveHotel1} \vee \mathbf{ReserveHotel2})\end{aligned}$$

7. RELATED WORK

Flow Composition Languages. Many formalizations of flow composition languages that support composition and compensation have been proposed in the literature [7, 8, 5]. These formalisms such as StAC [7], the saga calculus [5], and Compensating CSP [9] formalize process orchestration using atomic actions and composition operations similar to ours. They differ mostly in the features supported (e.g., whether recursion is allowed, whether parallel processes can be synchronized, or whether there are explicit commit mechanisms), in assumptions on atomic actions (whether or not they always succeed), and in the style of the semantics (trace based or operational).

We chose a core language that includes only the features relevant to our exposition. However, we borrowed extensively from the above languages and believe that similar results hold in the other settings. Like the Saga Calculus, we assume that atomic actions can succeed or fail, as this more closely matches the semantics of industrial languages such as BPEL [13]. We support all the composition operators of the Saga Calculus and Compensating CSP, except that we automatically apply compensation in the event of an error, rather than requiring a transaction block. Our sequential and parallel iterations are inspired by StAC's *generalized parallel* operator. However, our core language does not support interruptible recovery of parallel processes or recursion.

Notions of Correctness. The usual notion of correctness is *cancellation semantics* [9]. One can ensure that a process is self-canceling by restricting processes to have a compensation action for each forward action where compensations cannot fail and are independent of any other compensations running in a parallel branch [9]. Although order-independence between compensations is a realistic restriction, requiring a compensation for each action seems limiting. Verification becomes more involved when this restriction is relaxed. [18] describes an algorithm which checks that an OPERA workflow, potentially containing non-compensatable actions and exception handling, satisfies cancellation in $O(n^2)$ time. In [23], cancellation is checked on collections of interacting processes by creating an atomicity-equivalent abstraction of each process and checking the product of the abstractions.

Set consistency specifications can capture cancellation semantics. In addition, such specifications can model scenarios where self-cancellation is not desired (e.g., the order case study of Section 6.1) and can capture stronger requirements than cancellation (e.g., mutually exclusive actions in the travel agency case study of Section 6.2).

Other specification approaches have been suggested for composing web services, independent of compensation and transactional issues. For example, [3] proposes *consistency interfaces*, which define, for a method m of the service and result o of calling that method, the methods called by m and their associated results which lead the result o. The specification language for method calls includes union and intersection, thus providing similar capabilities as a set consistency specification. Consistency interfaces do not treat non-execution of an action the same as atomic failure, and there is no compensation. This precludes the use of negation in specifications and the interpretation of satisfying assignments as executions of the process. Our algorithm can be applied to check processes against consistency interfaces.

Finally, temporal logic specifications, frequently used by model checking tools, can also be used for compensating processes. While temporal logic is a more powerful specification language, set consistency can already model many properties of interest, and provides a more compact representation.

The problem of checking recursive processes against regular sets of traces is undecidable [12]. Note that any program with compensation and potentially infinite traces (e.g., due to loops or recursion) can have an infinite state space, even when no program variables are modeled. Thus, model checkers usually bound recursion depth and loop iterations (e.g., XTL for StAC [19]). This is less of an issue for set consistency verification, since the number of loop iterations can be abstracted without sacrificing soundness. In addition, as discussed in the introduction, the complexity bound for the verification of finite systems is lower for set consistency (co-NP complete vs. PSPACE complete).

Several verification tools have focused on verifying the interactions between processes. Legal process interactions may be specified as automata (e.g., WSAT [6, 15]) or message sequence charts (e.g., [14]). We view such interaction specifications as orthogonal to specifying the internal behavior of a process. In future work, we plan to combine both types of specifications, with set consistency used to represent internal process requirements and automata used to represent interaction requirements. We believe this approach will facilitate modular specifications and verification.

8. REFERENCES

[1] Business process modeling language (BPML). http://www.bpmi.org.

[2] Web services conversation language (WSCL) 1.0. http://www.w3.org/TR/wscl10.

[3] D. Beyer, A. Chakrabarti, and T. Henzinger. Web service interfaces. In *WWW '05*, pages 148–159. ACM, 2005.

[4] R. Bruni, M. Butler, C. Ferreira, C.A.R. Hoare, H. Melgratti, and U. Montanari. Comparing two approaches to compensable flow composition. 3653:383–397, 2005.

[5] R. Bruni, H. Melgratti, and U. Montanari. Theoretical foundations for compensations in flow composition languages. In *POPL '05*, pages 209–220. ACM, 2005.

[6] T. Bultan, X. Fu, R. Hull, and J. Su. Conversation specification: a new approach to design and analysis of e-service composition. In *WWW '03*, pages 403–410. ACM, 2003.

[7] M. Butler and C. Ferreira. A process compensation language. In *IFM '00: Integrated Formal Methods*, pages 61–76. Springer, 2000.

[8] M. Butler and C. Ferreira. An operational semantics for StAC, a language for modelling long-running business transactions. In *Coordination '04*, volume 2949 of *LNCS*, pages 87–104. Springer, 2004.

[9] M. Butler, C. A. R. Hoare, and C. Ferreira. A trace semantics for long-running transactions. In *Communicating Sequential Processes: The First 25 Years*, volume 3525/2005 of *LNCS*, pages 133–150. Springer, 2004.

[10] N. Een and N. Sörensson. An extensible SAT-solver. In *SAT '03*, pages 502–518, 2003.

[11] A. Elmagarmid, Y. Leu, W. Litwin, and M. Rusinkiewicz. A multidatabase transaction model for interbase. In *VLDB '90*, pages 507–518. Morgan Kaufmann, 1990.

[12] M. Emmi and R. Majumdar. Verifying compensating transactions. In *VMCAI '07*. Springer, 2007.

[13] T. Andrews et al. Business process execution language for web services, May 2003. http://dev2dev.bea.com/webservices/BPEL4WS.html.

[14] H. Foster, S. Uchitel, J. Magee, and J. Kramer. Model-based verification of web service compositions. In *ASE '03*. IEEE, 2003.

[15] X. Fu, T. Bultan, and J. Su. Analysis of interacting bpel web services. In *WWW '04*, pages 621–630. ACM, 2004.

[16] H. Garcia-Molina and K. Salem. Sagas. In *SIGMOD '87*, pages 249–259. ACM, 1987.

[17] J. Gray and A. Reuter. *Transaction processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[18] C. Hagen and G. Alonso. Exception handling in workflow management systems. *IEEE Trans. Softw. Eng.*, 26(10):943–958, 2000.

[19] J.Augusto, M. Leuschel, M. Butler, and C. Ferreira. Using the extensible model checker XTL to verify StAC business specifications. In *AVoCS '03*, 2003.

[20] H. Schuldt, G. Alonso, C. Beeri, and H.-J. Schek. Atomicity and isolation for transactional processes. *ACM Trans. Database Syst.*, 27(1):63–116, 2002.

[21] I. Singh, S. Brydon, G. Murray, V. Ramachandran, T. Violleau, and B. Stearns. *Design Web Services with the J2EE 1.4 Platform*. Addison-Wesley, 2004.

[22] A. Sistla and E. Clarke. The complexity of propositional linear temporal logics. *J. ACM*, 32(3):733–749, 1985.

[23] C. Ye, S.C. Cheung, and W.K. Chan. Publishing and composition of atomicity-equivalent services for B2B collaboration. In *ICSE '06*, pages 351–360. ACM, 2006.