Verifying Transactional Requirements of Web Service Compositions Using Temporal Logic Templates

Scott Bourne, Claudia Szabo, and Quan Z. Sheng

School of Computer Science
The University of Adelaide, SA 5005, Australia
{scott.bourne,claudia.szabo,michael.sheng}@adelaide.edu.au

Abstract. Ensuring reliability in Web service compositions is of crucial interest as services are composed and executed in long-running, distributed mediums that cannot guarantee reliable communications. Towards this, transactional behavior has been proposed to handle and undo the effects of faults of individual components. Despite significant research interest, challenges remain in providing an easy-to-use, formal approach to verify transactional behavior of Web service compositions before costly development. In this paper, we propose the use of temporal logic templates to specify component-level and composition-level transactional requirements over a Web service composition. These templates are specified using a simple format, configured according to scope and cardinality, and automatically translated into temporal logic. To verify design conformance to a set of implemented templates, we employ model checking. We propose an algorithm to address state space explosion by reducing the models into semantically equivalent Kripke structures. Our approach facilitates the implementation of expressive transactional behavior onto existing complex services, as demonstrated in our experimental study.

1 Introduction

Service-oriented architectures and Web services have been the focus of active research in the past decade [1–3]. Despite significant interest in techniques for designing, deploying, and ensuring reliability of Web services, many existing services experience severe issues such as timeout, dependability and unexpected behavior [4]. Market pressures that require ad-hoc deployment without proper quality assurance contribute to this issue. An important challenge remains verifying the correctness of a Web service composition with respect to fault-handling logic at *design-time*. This will permit developers to identify design flaws before costly development and improve the quality of the service composition [2, 3, 5].

Transactional behavior can be used to contain, handle, and undo the effects of faults in the execution of a Web service composition [3, 5, 6]. Requirements for transactional behavior need to be drawn from application-specific business logic that dictate which faults are acceptable, retriable, or recoverable. For example,

X. Lin et al. (Eds.): WISE 2013, Part I, LNCS 8180, pp. 243–256, 2013.
© Springer-Verlag Berlin Heidelberg 2013

a service to retrieve customer details can be retried safely, but services to commit payment or place orders may require recovery or replacement upon failure. Business logic can also dictate requirements at the composition-level, such as compensatory processes to rollback all execution effects [7].

A formal yet practical process of specifying transactional behavior and requirements, followed by the verification of the Web service composition design can significantly reduce development and maintenance cost, and also increase credibility and reliability in the deployed service. Previous approaches have proposed the detailed specification of failure models [5], composition risk levels [7], or the definition of vital components for the successful completion of Web services [3]. These approaches offer significant improvements towards the verification of transactional requirements, but tend to lead to state space explosion when verified [8] or restrict the transactional requirements that can be verified. A method to specify transactional requirements that, to the best of our knowledge, is yet to be explored, is the adaptation of temporal logic patterns [9–11]. These are frequently used structures of temporal logic properties [12], that can be implemented to specify sophisticated requirements. This allows users to take advantage of the expressive power of temporal logic, while reducing the effort and error-prone nature of pure manual specification.

In this paper, we propose a modeling approach for the design of transactional Web service compositions that allows users to specify transactional requirements formally using temporal logic templates and verify conformance at design time. Our earlier work has proposed a novel model that separates the service behavior into *operational* and *control* behaviors, allowing for flexible design, development, and verification of complex Web services [2]. The operational behavior contains the underlying business logic of the system, while the control behavior maintains the transactional state and guides the execution of the service. A set of pre-defined messages enable conversations between operational and control behavior that trigger the execution of components, indicate faults, specify recovery operations, and signal completion, among other operations [13]. We extend our previous work by enabling the control and operational behaviors to be verified against transactional requirements drawn from the business logic specified by the user with temporal logic templates. The main contributions of our work are:

- A set of temporal logic templates to formally specify component-level and composition-level transactional requirements derived from business logic to facilitate the verification of composite Web services.
- A service verification approach based on model checking that utilizes temporal logic properties obtained from implemented templates to verify transactional requirements, while addressing state space explosion with model reduction measures.
- A prototype implementation that facilitates our verification approach over Web service composition designs as control and operational behavior models.

The remainder of the paper is organized as follows. Section 2 presents an overview of the control and operational behavior approach for Web service modeling. Section 3 outlines our approach for capturing transactional requirements

with temporal logic templates. Section 4 describes how a design can be verified against these requirements in our approach. Section 5 reports the prototype implementation and experimental study. Finally, Section 6 contrasts our work with related work and Section 7 concludes this paper and discusses future directions.

2 Background

To specify transactional Web service compositions at design-time, we adapt the modeling method proposed in our previous work [2], based on the separation of Web service behavior into *control* and *operational* behaviors. The control behavior is an application-independent model that maintains the transactional state of the composition, while the operational behavior contains the application-dependent flow of business tasks. The execution and recovery operations of the service are directed by the control behavior, according to events reported from the operational behavior. Using these models, this design provides a detailed view of the functional and transactional behavior of a composition, moreover, each perspective to be designed and modified independently.

The control and operational behavior models are expressed as a 5-tuple $\mathcal{B} = \langle \mathcal{S}, \mathcal{L}, \mathcal{T}, s_0, \mathcal{F} \rangle$ where $\mathcal{S}$ is a finite set of state names, s_0 is the initial state, $\mathcal{F} \subseteq \mathcal{S}$ is a set of final states, and $\mathcal{L}$ is a set of event-condition-action labels. $\mathcal{T} \subseteq \mathcal{S} \times \mathcal{L} \times \mathcal{S}$ is the transition relation where each $t \in \mathcal{T}$ consists of a source and target state, and a transition label. We can express these models using statecharts, as shown in the online payment composition in Figure 1. The control behavior model contains the transactional states of the composition, while the operational behavior contains the flow of business tasks of the process.

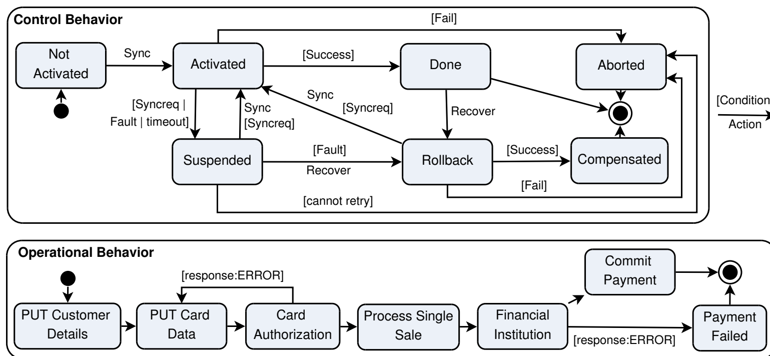

Fig. 1. The control and operational behaviors of a basic online payment composition

To enable communication between the behavior models, we use a set of *inter-behavior messages*. These allow the control behavior to direct execution, and the operational behavior to report events and status. The messages are classified as *initiation messages* that are sent from the control behavior, and *outcome messages* that are responses from the operational behavior. The initiation messages

include `Sync` to initiate or resume execution of the service, `Recover` to trigger recovery operations, `Delay` to force a response from an operational behavior state, and `Ping` to test the liveness of a state. The outcome messages include `Success` to indicate the successful commit of the service, `Fail` to signal an abort, `Fault` to indicate the presence of a fault that requires recovery, `Ack` to report the liveness of a state, and `Syncreq` to request a `Sync` message to retry a component or resume execution following the recovery. These inter-behavior messages enable us to specify transactional behavior over the design. For example, in the online payment example in Figure 1, a `Sync` message could be used to trigger the process from *PUT Customer Data*, and a `Success` or `Fail` message could be sent from *Commit Payment* and *Payment Failed* respectively.

Our previous work [13] proposes a method to ensure well-formed inter-behavior conversations that avoid deadlock, incomplete execution, and prevent inconsistency between the behavior models. However, these properties are insufficient for the designer to ensure that the transactional behavior of the model conforms to their expectations, as the design cannot be verified against application-dependent requirements, such as the success of critical components prior to commit, or acceptable alternative operations given a failed component.

3 Temporal Logic Templates

A developer must have confidence that a potentially long-running composition of distributed and heterogeneous Web services will conform to a set of transactional requirements for containing and handling faults. These requirements could apply to failures of individual components, e.g., required recovery operations to undo the component's effect, or to the scope of the composition, such as components critical for success, or relaxed atomicity conditions for failure. It is crucial to specify these requirements formally, and identify and resolve any compliance issues prior to Web service development.

We propose to formally specify transactional requirements with *temporal logic templates* that are filled by a Web service designer to specify transactional requirements. This approach adapts previous work in temporal logic patterns [9], which simplify property specification by identifying common structural patterns. In contrast, our work constructs templates specialized for transactional requirements, which allow detailed business logic to be defined to this domain. Similar to temporal logic patterns, templates do not require expert knowledge in temporal logic to use, reducing human error and effort. The templates use Linear Temporal Logic (LTL) and Computational Tree Logic (CTL) [12], which specify properties of a system over two different timeline representations. We employ both languages since the properties they can specify are not equivalent [14].

Each template is defined with fields for *description*, *design prerequisites*, *required variables*, *scope*, *cardinality*, *temporal logic* and an *example use*. The *Scope* field allows the user to limit when the transactional requirement should be applied, while the *Cardinality* field reduces template ambiguity by allowing users to customize the relationship between variables. Since transactional requirements

can apply to the behavior of the whole composition, or to specific components, we group our templates into two categories. *Component-level* templates specify requirements specific to components, and *composition-level* templates apply to the transactional behavior of the entire composition. The separation of control and operational behaviors in our design model means the components of interest to component-level templates are the operational behavior states. In contrast, the composition-level templates utilize control behavior states, as they present a transactional view of the composition. The descriptions of the component-level and composition-level templates are shown in Table 1 and 3 respectively. The full specifications of all our templates are omitted for space, but considerators and readers are referred to the author's website for the full list[1]. A formal proof of completeness for our template set is difficult to obtain, but in this paper we provide a foundation of examples that can be extended easily.

Table 1. Names and descriptions of the component-level temporal logic templates

Template Signature	**Description**
`CompensateFailure <Component,Recovery, Card,Scope>`	Specifies a component and a condition. The failure of the component requires the condition to be satisfied in the future, to recover from the failure.
`CompensateSuccess <Component,Recovery, Card,Scope>`	Specifies a component and a condition. When the composition must be undone, the condition must be satisfied to undo the effect of the component.
`Alternative <Component,Recovery, Card,Scope>`	Following the failure of a component, one or several alternative operations, expressed as a condition, are considered acceptable replacements.
`NonRetriable <Component,Scope>`	Following failure of a component, retrial is either not possible, or the user is not interested in it.
`RetriablePivot <Component,Scope>`	A component that may be retried, but not undone. Following its success, the service must commit.
`NonRetriablePivot <Component,Scope>`	A component that may not be retried or undone, and leads to commit or abort depending on success.

3.1 Component-Level Templates

Component-level templates specify transactional requirements for handling failures of individual components in a Web service composition. For example, a component-level transactional requirement of the online payment composition could specify that `Card Authorization` cannot be retried without first reattempting `PUT Card Data`. Other common transactional requirements applied to components include *compensatable*, *retriable*, *pivot*, *replaceable*, or similar labels [3, 5–7]. However, using templates to specify these requirements, instead of applying labels to components, is a more expressive method, as it allows scope and

[1] `www.adelaide.edu.au/directory/scott.bourne?dsn=directory.file;field=data;id=24812;m=view`

other options to be adjusted, and enables complex requirements to be specified, such as satisfactory recovery conditions for specific failures.

We propose six component-level templates, as shown in Table 1. `Compensate Failure`, `CompensateSuccess`, and `Alternative`, require both the component and the compensatory or alternative operations as variables. The `Card` attribute specifies the cardinality relationship between these variables. Templates `Non Retriable`, `RetriablePivot` and `NonRetriablePivot` contain only a single component and specify how that component may be treated following failure or success. All templates contain a variable to define the `Scope` of the requirement.

Table 2 contains a complete specification of the `CompensateFailure` template. The template is implemented by specifying an operational behavior state as the *component*, and a boolean *Recovery* condition that when satisfied, reflects satisfactory failure recovery. The cardinality field allows the user to specify whether *Recovery* only applies to a single failure, or to several failures of the same component. The `Scope` can be global (G), or a function over a condition P. Cardinality and scope determine the LTL property to be used. As shown in the example, the LTL property of cardinality `1:1` and global scope can be informally translated as: it is always the case that if there is a *FAULT* message sent from the component, the component will not be executed until its *Recovery* operation is performed; nevertheless, a fault of the component will always be followed by *Recovery*. This template can be implemented as shown by the example row, which specifies the required fault handling at `Card Authorization` in the online payment design.

3.2 Composition-Level Templates

Composition-level templates differ from component-level templates by specifying requirements over the entire composition, such as preconditions, triggers, or reachability conditions for entering control behavior states. For example, in the online payment design, the success of `Card Authorization` and `Commit Payment` could be preconditions for commit. Our templates can capture these requirements by using control behavior states.

The proposed set of composition-level temporal logic templates is shown in Table 3. `ControlStateCritical`, `ControlStateTrigger`, `ControlStateReachable`, and `ControlStateUnreachable` allow users to specify pre-conditions, triggers and reachability conditions for entering control behavior states. `Compensation` and `ConditionalCompensation` allow users to verify that the compensation actions of the composition meet requirements. The underlying properties of these compensation templates partially overlap some component-level templates, but the Web service designer can determine which template type is appropriate. For example, if one operation undoes the effect of several components, it would be simpler to implement a single composition-level template. Conversely, if several components each have a corresponding rollback operation, verifying each relationship individually through component-level templates is preferable.

Table 4 shows the full template specification of `ControlStateCritical`, which specifies a precondition for entering a control behavior state. To express this property in temporal logic, LTL with past-time operators is applied [12]. The O

Table 2. Template specification for CompensateFailure

Name	CompensateFailure <Component,Recovery,Card,Scope>		
Type	Component-level		
Variables	*Component*	An operational behavior state that requires recovery upon failure.	
	Recovery	A condition that undoes the effect of the failure. This can be a single component or a set of components structured with $\wedge$ and $\vee$ operators.	
	Card	One of the cardinality options below.	
	Scope	One of the scope options below.	
Description	The failure of *Component* leaves an impact an effect, which must be compensated by *Recovery* becoming true in the future.		
Prerequisite	A Fault message originating from *Component* in the operational behavior is necessary for this requirement to be verified.		
Cardinality	1:1	*Recovery* undoes one failure of *Component*.	
	Many:1	*Recovery* can undo many failures of *Component*.	
Scope	G	The template applies in all executions.	
	P	Applies during the satisfaction of a condition P.	
	$\neg P$	Applies during the negation of a condition P.	
	Before P	*Recover* must precede the satisfaction of P.	
LTL	1:1	G	$G(Component.FAULT \rightarrow ((\neg(Activated \wedge Component) \cup Recovery) \wedge F(Recovery))$
		P	$F(P) \rightarrow G(Component.FAULT \rightarrow ((\neg(Activated \wedge Component) \cup Recovery) \wedge F(Recovery))$
		$\neg P$	$G(\neg P) \rightarrow G(Component.FAULT \rightarrow ((\neg(Activated \wedge Component) \cup Recovery) \wedge F(Recovery))$
		Before P	$G(Component.FAULT \rightarrow (((\neg(Activated \wedge Component) \wedge \neg P) \cup Recovery) \wedge F(Recovery))$
	Many:1	G	$G(Component.FAULT \rightarrow F(Recovery))$
		P	$F(P) \rightarrow G(Component.FAULT \rightarrow F(Recovery))$
		$\neg P$	$G(\neg P) \rightarrow G(Component.FAULT \rightarrow F(Recovery))$
		Before P	$G(Component.FAULT \rightarrow ((\neg P \cup Recovery) \wedge F(Recovery)))$
Example	CompensateFailure <Card Authorization,PUT Card Data,1:1,G>		

operator is used to specify a property that must have occurred previously, while the H operator defines a property that must hold in *all* previous states. The example specifies the critical condition for entering the Done state.

Table 3. Names and descriptions of the composition-level temporal logic templates

Template Signature	Description
`ControlStateCritical <ControlState,Condition,Scope>`	A condition required for entering a control behavior state.
`ControlStateTrigger <ControlState,Condition,Scope>`	A condition that must trigger a control behavior state in the future.
`ControlStateReachable <ControlState,Condition,Scope>`	A condition that indicates a control behavior is reachable.
`ControlStateUnreachable <ControlState,Condition,Scope>`	A condition that indicates a control state should not be reachable in the future.
`Compensation <CompCondition>`	Specifies a condition that must be met during any compensation process.
`ConditionalCompensation <ExecCondition,CompCondition>`	Given a condition that can be satisfied during successful execution, specifies a second condition for compensation.

4 Proposed Verification Approach

We apply model checking [14] to verify that a Web service composition designed using our control and operational behavior model conforms to a set of transactional requirements specified with temporal logic templates. Model checking is a method to formally verify a system against a set of properties by exhaustively exploring the system state space. If a contradiction is found, a stack trace demonstrating the violation is produced. We employ NuSMV [15] for model checking, since it provides support for properties specified in LTL and CTL.

To address the state explosion problem inherent in model checking [14], we automatically reduce the state space of the control and operational behavior model as much as possible, by removing states and messages not relevant to the requirements being verified. To this end, we generate a Kripke structure [16] based on the temporal relations between the variables specified in the templates. Template variables can be defined as $\mathcal{V} \subseteq \mathcal{S}_{co} \cup \mathcal{S}_{op} \cup \mathcal{M}$ where S_{co} and S_{op} are control and operational behavior states and $\mathcal{M}$ is the set of inter-behavior messages. The Kripke structure will capture all instances of elements from $\mathcal{V}$ in the design, and the transition relation between those instances.

A Kripke structure is a finite-state system model defined as $\mathcal{K} = \langle \mathcal{S}_k, \mathcal{I}, \mathcal{T}_k, \mathcal{L} \rangle$, where $\mathcal{S}_k$ is a finite set of states, $\mathcal{I} \subseteq \mathcal{S}_k$ is the set of initial states, $\mathcal{T}_k \subseteq \mathcal{S}_k \times \mathcal{S}_k$ is the transition function, and $\mathcal{L}$ is the labelling function that assigns *atomic propositions* to each state. The atomic propositions are the unique set of properties that hold at a given state. We identify a set of three atomic propositions for each state in the Kripke structure; *i*) the control behavior state; *ii*) the operational behavior state; and *iii*) the most recent inter-behavior message, represented as a pair (s_{op}, m) of related operational behavior state and message type. Since we aim to reduce the model to the properties we wish to verify, the Kripke structure only contains the states with atomic propositions with elements from $\mathcal{V}$.

Table 4. Template specification for `ControlStateCritical`

Name	`ControlStateCritical<ControlState,Condition,Scope>`	
Type	Composition-level	
Variables	*ControlState*	The control behavior state this critical condition applies to.
	Condition	The precondition for entering this control behavior state. This can be a single component or a set structured with $\wedge$ and $\vee$ operators.
	Scope	One of the scope options below.
Description	*Condition* denotes the precondition for entering *ControlState*. When *ControlState* is entered, *Condition* must have been met previously on the execution path.	
Scope	G	The template applies in all executions.
	P	Applies during the satisfaction of a condition P.
	$\neg P$	Applies during the negation of a condition P.
	Before P	*ControlState* is entered before P is met.
LTL	G	$G(\mathit{ControlState} \rightarrow O(\mathit{Condition}))$
	P	$F(P) \rightarrow G(\mathit{ControlState} \rightarrow O(\mathit{Condition}))$
	$\neg P$	$G(\neg P) \rightarrow G(\mathit{ControlState} \rightarrow O(\mathit{Condition}))$
	Before P	$G(\mathit{ControlState} \rightarrow (O(\mathit{Condition}) \wedge H(\neg P)))$
Example	`ControlStateCritical <Done,Card Authorization,Commit Payment,G>`	

To build the Kripke structure, the control and operational behavior model must be exhaustively traversed, so the Kripke states can be created and linked as elements from $\mathcal{V}$ are encountered. Due to space constraints, we omit the detailed algorithm, but provide a description of the verification process. The control and operational behavior are explored with a depth-first traversal, starting from the control behavior state `Not Activated` and the operational behavior yet to be initialized. From this state, the control behavior activates and explores each possible way to trigger operations in the operational behavior through inter-behavior messages. The traversal explores every possible execution path within the operational behavior, and every reachable inter-behavior message. When an element from $\mathcal{V}$ is encountered, a Kripke state with the atomic properties currently true in the traversal is either created, or a transition to an existing Kripke state containing those properties is added. The traversal handles cycles in the model by backtracking when an exiting Kripke state is linked to, a set of atomic properties are revisited since the last addition to the Kripke structure, or the control behavior terminates through `Done`, `Abort` or `Compensated`. The Kripke structure is complete once the traversal returns to `Not Activated`.

Figure 2 shows a Kripke structure of the design in Section 2 and the example template inputs of Tables 2 and 4, such that $\mathcal{V} = \{$`Card Authorization, PUT Card Data, Commit Payment, Done`$\}$. Each state is labeled with three atomic propositions as described above. While a Kripke structure of all reachable atomic

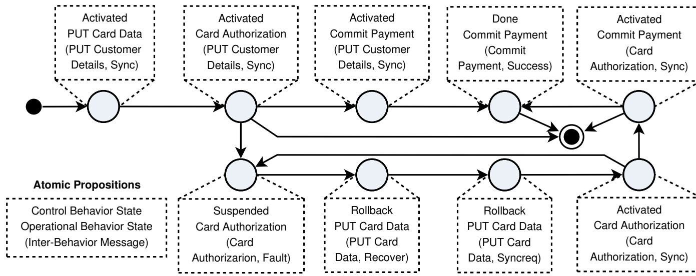

Fig. 2. An example Kripke structure generated from the online payment composition

propositions of the model of Figure 1 would contain 31 states, the reduction measures minimize this structure to 11, creating less model checking overhead.

5 System Implementation and Experiments

We have implemented our verification approach in a prototype tool, with an interface for specifying control and operational behaviors as shown in Figure 3. The prototype reduces the model to the Kripke structure, and writes it to an SMV (Symbolic Model Verifier) file with the temporal logic representations of the implemented templates. NuSMV uses this file to exhaustively verify the model against the set of temporal properties, and lists the properties found to be true, plus any violating state sequences. To further help the designer, our future work will interpret these sequences to diagnose design flaws within the control and operational behavior.

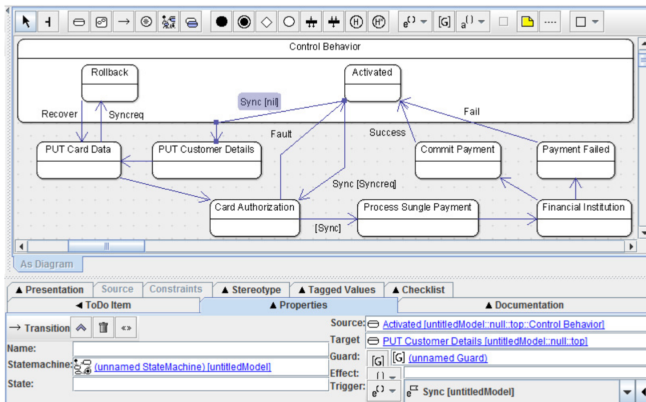

Fig. 3. Specifying a Web service composition as control and operational behaviors

We demonstrate our proposed approach with an extension to the online payment example of Section 2. This design uses the PayLane Web service API[2], and

[2] `http://devzone.paylane.com`

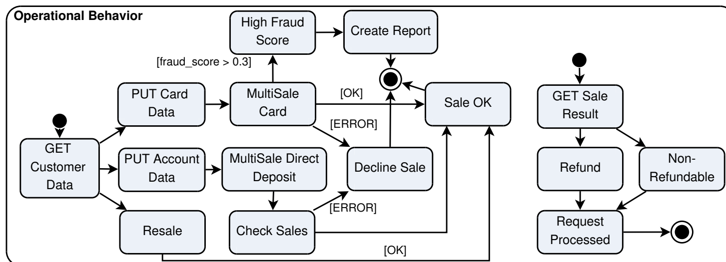

Fig. 4. The operational behavior model of the complex online payment composition

extends the online payment example from Section 2 by incorporating multiple payment options, namely, `card charge` and `direct debit`. While the control behavior model remains the same as the earlier example, the operational behavior and inter-behavior messages of the complex online payment design are shown in Figure 4 and Table 5 respectively. The composition enables users to pay by credit card or direct deposit, either by entering new data or retrieving details from a previous transaction. When a card payment is processed, a `fraud score` is returned to indicate the likeliness of fraud being committed. In cases where this score is above a given threshold, a report of the transaction is made. When a direct deposit is made, the result of the transaction is not immediately available and must be checked. The design also contains a compensatory process that determines whether to process a refund based on retrieved sales details.

Table 5. Inter-behavior messages specified over the online payment design

Message	Source	Target	Guard
Sync	Activated	GET Customer Data	[no message]
Sync	Activated	PUT Card Data	[Resale.SYNCREQ]
Sync	Activated	PUT Account Data	[Resale.SYNCREQ]
Syncreq	Resale	Activated	-
Delay	Activated	MultiSale Card	-
Fault	MultiSale Card	Activated	-
Fault	Check Sales	Activated	-
Success	Sale OK	Activated	-
Success	Request Processed	Rollback	-
Fail	Decline Sale	Rollback	-
Fail	Create Report	Rollback	-
Recover	GET Sale Result	Rollback	[Sale OK.SUCCESS]
Recover	High Fault Score	Rollback	[MultiSale Card.FAULT]
Recover	Decline Sale	Rollback	[FAULT]

We identify seven critical transactional requirements to be verified over this design, as shown in Table 6. Requirement `TR1` specifies that when a committed sale needs to be undone, a refund should be processed, excepting non-refundable

Table 6. Requirements implemented with temporal logic templates and verified

ID	Requirement	Result
TR1	CompensateSuccess <Sale OK, Refund | Non-Refundable, 1:1, G >	Passed
TR2	Alternative <Resale, PUT Account Data | PUT Card Data, 1:1, G>	Passed
TR3	RetriablePivot <Check Sales, G>	Passed
TR4	ControlStateCritical <Aborted, Create Report, High Fraud Score>	Passed
TR5	ControlStateTrigger <Done, Sale OK & (MultiSale Card | MultiSale Direct Deposit | Resale), G >	Passed
TR6	ControlStateReachable <Done, Resale.SYNCREQ, G>	Passed
TR7	ControlStateReachable <Done, MultiSale.DELAY, G>	Failed

sales. TR2 requires that the failure of historical transaction data sales should lead to card or account data being updated before the sale is retried. Requirement TR3 asserts that the result of direct deposit sales should be requested until one is obtained, which then leads to commit or abort. In the event that a high fraud score is detected, requirement TR4 specifies that a report must be generated before the composition aborts. TR5 requires that the composition should always commit following the successful processing of card or direct deposit payment. Requirement TR6 specifies that commit should be reachable following the failure of resale, while TR7 requires that commit should be reachable even if the response of MultiSale Card is delayed. NuSMV took 0.03 seconds to verify these requirements and determined that the design violates TR7, since Done is unreachable following a Delay message to MultiSale Card. To satisfy TR7, the design must be refined to enable MultiSale Card to be retried or replaced following a delay. The remaining requirements were satisfied for all states in the design.

6 Related Work

Ensuring and verifying transactional requirements in Web service composition has been an area of increasing research interest in recent years. Bhiri et al. [5] and Montagut et al. [6] present Web service composition methods to ensure failure atomicity properties defined as an Accepted Termination States (ATS) [17] model. While an ATS model allows detailed specifications of failure atomicity, it is an exhaustive method that increases exponentially in size as the composition grows, whereas our template set allows requirements to be flexibly defined only to components the designer wishes to verify. Montagut et al. [8] limit the ATS model to zones of the composition where transactional requirements are considered critical. Despite this, all valid termination state combinations must be exhaustively defined, while in our approach the designer can determine the level of requirement detail. The FACTS framework [3] enables users to verify that

the transactional behavior and exception handling of a design model supports components specified critical for success. Our composition-level templates enables more detailed composition-level requirements with the definition of critical preconditions, triggers, and reachability conditions for commit, abort, and other transactional states. Another approach proposed by El Haddad et al. [7] uses *risk levels* provided by the user to specify whether the resulting composition should be compensatable, but reduces user control over the transactional behavior of their composition. In contrast, our approach allows compensatory activities to be explicitly specified and verified against business requirements.

Our use of templates draws inspiration from existing work in business rule compliance with temporal logic patterns. Dwyer et al. [9] defined a set of patterns in LTL and CTL with various scope options, based on commonly recurring temporal logic property structures found across surveyed specifications. This initial pattern set has been adapted and expanded in subsequent research. Smith et al. [10] extended this set into templates, allowing them to be customized according to cardinality and other fine-grained options. Elgammal et al. [11] produced an expanded pattern set with a framework that enables atomic patterns to be composed together, and includes a set of basic composite patterns. Since our focus is on transactional requirements instead of general compliance, we produce a set of temporal logic templates highly specialized and more appropriate towards that use. Furthermore, our template set applies to a specific design model with control and operational models, which allows our templates to specify requirements at the component-level and composition-level, increasing granularity and expressiveness at no additional computational cost. Finally, Yu et al. [18] apply temporal logic patterns towards verifying Web service compositions. However, their work analyzes WS-BPEL schemas for conformance to general functional properties, while our approach can be applied prior to any development.

7 Conclusion

The verification of transactional requirements of Web service compositions using information derived from business logic remains an important challenge despite increasing research interest in recent years. Identifying and resolving compliance issues to transactional requirements early in development is desirable. Furthermore, the formal specification of transactional requirements is error-prone and an approach to hide the specification complexities from the Web service designer will increase the reliability of the design. In this paper, we propose a set of temporal logic templates to formally verify component-level and composition-level transactional requirements of Web service compositions at design time. Each template is defined by the Web service designer according to a specification that contains a description, variables, and scope and cardinality variants. The implemented templates are automatically translated into temporal logic properties that are verified using a model checker. The proposed approach has been successfully implemented and verified with several example scenarios. Our future work includes supporting parallel workflow patterns, enabling more sophisticated guard conditions, and scalability tests with increasingly complex scenarios.

References

1. Papazoglou, M.P., Traverso, P., Dustdar, S., Leymann, F.: Service-Oriented Computing: State of the Art and Research Challenges. IEEE Computer 40(11) (2007)
2. Sheng, Q., Maamar, Z., Yahyaoui, H., Bentahar, J., Boukadi, K.: Separating Operational and Control Behaviors: A New Approach to Web Services Modeling. IEEE Internet Computing (3), 68–76 (2010)
3. Liu, A., Li, Q., Huang, L., Xiao, M.: FACTS: A Framework for Fault-tolerant Composition of Transactional Web Services. IEEE Transactions on Services Computing 3(1), 46–59 (2010)
4. Domingue, J., Fensel, D.: Toward a Service Web: Integrating the Semantic Web and Service Orientation. IEEE Intelligent Systems 23(1), 86–88 (2009)
5. Bhiri, S., Perrin, O., Godart, C.: Ensuring Required Failure Atomicity of Composite Web Services. In: Proceedings of the 14th International Conference on World Wide Web, pp. 138–147. ACM (2005)
6. Montagut, F., Molva, R., Golega, S.: Automating the Composition of Transactional Web Services. International Journal of Web Services Research (IJWSR) 5(1) (2008)
7. El Hadad, J., Manouvrier, M., Rukoz, M.: TQoS: Transactional and QoS-Aware Selection Algorithm for Automatic Web Service Composition. IEEE Transactions on Services Computing 3(1), 73–85 (2010)
8. Montagut, F., Molva, R., Tecumseh Golega, S.: The Pervasive Workflow: A Decentralized Workflow System Supporting Long-Running Transactions. IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews (2008)
9. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in Property Specifications for Finite-State Verification. In: Proceedings of the 1999 International Conference on Software Engineering, pp. 411–420. IEEE (1999)
10. Smith, R.L., Avrunin, G.S., Clarke, L.A., Osterweil, L.J.: PROPEL: an Approach Supporting Property Elucidation. In: Proceedings of the 24th International Conference on Software Engineering, pp. 11–21. ACM (2002)
11. Elgammal, A., Turetken, O., van den Heuvel, W.-J., Papazoglou, M.: Root-Cause Analysis of Design-Time Compliance Violations on the Basis of Property Patterns. In: Maglio, P.P., Weske, M., Yang, J., Fantinato, M. (eds.) ICSOC 2010. LNCS, vol. 6470, pp. 17–31. Springer, Heidelberg (2010)
12. Emerson, E.: Temporal and Modal Logic. In: Handbook of Theoretical Computer Science, vol. 2, pp. 995–1072 (1990)
13. Bourne, S., Szabo, C., Sheng, Q.Z.: Ensuring Well-Formed Conversations Between Control and Operational Behaviors of Web Services. In: Liu, C., Ludwig, H., Toumani, F., Yu, Q. (eds.) ICSOC 2012. LNCS, vol. 7636, pp. 507–515. Springer, Heidelberg (2012)
14. Baier, C., Katoen, J.P., et al.: Principles of Model Checking. MIT Press (2008)
15. Cimatti, A., Clarke, E., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: NuSMV 2: An Opensource Tool for Symbolic Model Checking. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 359–364. Springer, Heidelberg (2002)
16. Kripke, S.: Semantical Considerations on Modal Logic. Acta Philosophica Fennica 16, 83–94 (1963)
17. Kim, W.: Modern Database Systems: The Object Model, Interoperability, and Beyond. ACM Press/Addison-Wesley Publishing Co. (1995)
18. Yu, J., Manh, T.P., Han, J., Jin, Y., Han, Y., Wang, J.: Pattern Based Property Specification and Verification for Service Composition. In: Aberer, K., Peng, Z., Rundensteiner, E.A., Zhang, Y., Li, X. (eds.) WISE 2006. LNCS, vol. 4255, pp. 156–168. Springer, Heidelberg (2006)