
2013-IJ-002

Compensation by Design

Xi Liu, Shaofa Yang, J.W.Sanders

Formal Aspects of Computing 2013

Most of the papers available from this document appear in print, and the corresponding copyright is held by the publisher. While the papers can be used for personal use, redistribution or reprinting for commercial purposes is prohibited.
Compensation by design

Xi Liu¹, Shaofa Yang² and J. W. Sanders³

¹ State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China
² State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China
³ African Institute for Mathematical Sciences, Muizenberg and Stellenbosch University, Stellenbosch, South Africa

Abstract. The current dominance of the service-based paradigm reflects the success of specific design and architectural principles embodied in terms like SOA and REST. This paper suggests further principles for the design of services exhibiting long-running transactions (that is, transactions whose characteristic feature is that in the case of failure not all system states can be automatically restored: system compensation is required). The principles are expressed at the level of scope-based compensation and fault handling, and ensure the consistency of data critical to the business logic. They do so by demanding (a) either the commitment of all of the transaction or none of it, and (b) that compensation is assured in case of failure in ‘parent’ transactions. The notion of scope is captured algebraically (rather than semantically) in order to express design guidelines which ensure that a given transaction satisfies those principles. Transactional processes are constructed by parallel composition of services, and transactions with scopes in a single service are dealt with as a special case. The system semantics is formalised as a transition system (in Z) and the principles are expressed as formulae in linear temporal logic over runs of the transition system. That facilitates the model checking (using SAL) of their bounded versions. Two simple examples are used throughout to illustrate definitions and finally to demonstrate the approach.

Keywords: Service computing, Long-running transaction, Compensation, Temporal logic

1. Introduction

Service-based computation provides a vital computational paradigm whose progress has resulted from the identification of specific design and architectural principles, like service oriented architecture (SOA) and REpresentational state transfer (REST) [FT02, Fie00]. SOA allows the designer to tame complex functionality by integrating existing web services; its principles include XML-based standards (WSDL, SOAP, etc.), service registry and service contract. RESTful services are viewed as URL-identified resources; the principles of REST include stateless client-server architecture and operation through HTTP methods.
The main characteristics that distinguish service-based systems from more traditional distributed systems result from long-running computations, or transactions, which in an open environment cannot simply be rolled-back when a fault occurs. Technologies like Sagas [GMS87] and BPEL [OAS07] provide support for such transactions, but offer no guidance to the designer for ensuring the consistency of data critical to business logic when a fault occurs. More specifically, (1) no proper definition is provided for transactional properties of long-running transactions: either full cancellation is assumed (e.g., [BHF04, GMS87]), or nothing concerning fault-handling and compensation (e.g., [OAS07, OAS09b]); (2) no rules exist for the design of proper compensable transactions: either the designer is required to provide compensation for every activity (e.g., [BHF04, OAS09b]) or the task of what and how to design the transaction is left completely to the designer (e.g., [OAS07]). The purpose of this paper is to propose new ‘transactional principles’ ensuring the atomicity and consistency on critical business data of long-running transactions and to identify design guidelines ensuring that those principles are met.

In formulating the transactional principles and design guidelines it has been found convenient to work at a level slightly more abstract than an implementation language like BPEL. This permits unimportant syntactic detail to be overlooked and allows the results to be applied more readily to a range of target languages. Also, it avoids possible ambiguities and oversights in the BPEL specification [QWPZ05, BHF04].

Transactions that are long-running typically involve partners whose interactions are not able to be monitored by the traditional lock-based methods for ensuring data consistency. Instead, fault handling with compensation is used to restore consistency in the case of failure. When a transaction fails, restoration of all its state to its original value is simply impractical; partners in the environment may lie beyond its control. Environmental variables may already have been committed; and it may have been found expedient to use local variables (e.g., to assist the design or to log process status) that do not pertain to the critical business logic and so their restoration is irrelevant. In this paper the notion of critical variable is introduced to capture those variables whose restoration is required in the case of failure.

‘Scopes’, the building blocks of long-running transactions, provide structure for combining the normal activity of a transaction with its fault handling, compensation and critical variables. Following BPEL, scopes are nested to arbitrary depth. A service is a top-level scope, i.e., one which is not contained in any other scope. A transaction may lie within a single service, or may span several services. In the latter case, the collection of scopes from different services is said to form an inter-service transaction (IST); one scope is designated the main scope, and the others called supporting scopes.

1.1. Contribution

The contributions of this paper are to propose principles for ensuring atomicity and consistency on critical business data of transactions involving one or more services, and to provide guidelines and validation approaches to ensure those principles are met. The design guidelines are sufficient conditions for transactional principles. When a process design falls outside the requirements for design guidelines, model checking techniques can be used to validate the process design against the principles.

The concept of process, and of scope in particular, are formalised algebraically. Standard constructs from sequential and reactive programming are exploited to express transactions, and the semantics is formalised as a transition system in Z [Spi92].
Compensation by design

To ensure the consistency of critical data, each transaction, scope and IST is required to exhibit so-called transactional principles:

**All-or-nothing:** after termination of a transaction, its effect on critical variables is the result of the scope’s normal activity and all scopes of the IST, if no fault occurs. But otherwise,

- the result of the scope’s normal activity with all successfully completed sub-scopes is committed (ignoring the result of failed sub-scopes), if a fault occurs but only within a sub-scope; analogously for the IST case: the result is committed of the normal activities of the IST’s successfully completed participating scope (ignoring the result of failed supporting scopes), if a fault occurs but only within a supporting scope of the IST;
- no changes are committed to critical variables if a fault occurs but not in any of the sub-scopes of the scope, analogously: for an IST, no changes are committed to critical variables if a fault occurs but not in supporting scopes or sub-scopes of the IST’s main scope.

**Compensation:** the critical variables are restored by the scope’s compensation if, after commitment of a transaction, compensation is triggered due to failure or compensation in a parent scope or the main scope of the IST.

Although both principles require recovery of critical variables, the all-or-nothing principle specifies the desired property of fault handlers that, in case of failure, nothing is committed on critical variables (consider commitment as persistence in databases). The compensation principle specifies compensation: affected updates on critical variables are ‘reversed’ (consider persistent updates in a database to be updated by the original). The transactional principles and design guidelines in the present paper do not support ad hoc fault handling and compensation strategies, such as charging penalties instead of recovery when the penalty affects critical variables of other transactions. One of the reasons is that penalty charging is an application-specific strategy.

The principles aim to ensure atomicity and consistency of long-running transactions. Although isolation and durability are also important in database transactions, they are not expected in long-running transaction which are typically open and data not locked. Changes made by uncommitted transactions are visible to the system and to the environment through interactions. Persistence of business data is achieved by the engine, so that durability is not a concern when designing transactions.

A language named \textit{BaT} is introduced (with minimum syntax) for the abstract expression of BPEL-like long-running transactions. The above principles are then stated as formulae of linear temporal logic (LTL) over executions of processes. A formal semantics for \textit{BaT} is provided as foundation for the principles. For validation of the transactional principles against the semantics, the process designer may take advantage of model-checking techniques. In this paper, the model checker SAL [dMOR’04] is used (though it suffices for a model checker to support programs and LTL with variable assignments). To transform the semantics from Z to the specification language of SAL [dMS03] the approach suggested by Smith and Wildman [SW05] is followed, resulting in formulation of the transactional principles as theorems in SAL. As usual, for the purpose of model checking the variable domain is assumed to be bounded.

‘Design guidelines’ are proposed as sufficient conditions for ensuring that a design satisfies the transactional principles. They constrain the form of a transaction and are defined to be as readily checkable as possible. The design guidelines ensure the transactional principles by using a mixture of static and dynamic analysis, with the intention of making the ‘dynamic’ part as simple as possible. The design guidelines are defined in a ‘modular’ and ‘recursive’ form so that they can be used to reduce the problem of checking the transactional principles on complex transactions to the same problem but on simpler sub-scopes or participating scopes (of ISTs).

As a means of ensuring correct compensation, the transactional principles are by no means complete. That is, these principles are sufficient conditions for correct compensation but are not necessary. Optimised, application specific techniques, lie beyond their reach. This is to be expected. An analogy (transactional memory shares much with transactional services, though the topics appear disjoint) might be drawn with correctness principles for the Java memory model: currently it does not seem possible to justify all thread optimizations used in Java. In order to make general progress various forms of optimization have been sacrificed. In the present case, that includes those which change the order of compensation (nonetheless the correctness of such optimized compensations follows only if they can be shown to act as if compensation were achieved in reverse order).

The relationship between the concepts appearing in the paper is illustrated in Fig. 1.
1.2. Examples

This paper carries two running examples. The first, Batch-Atm, exemplifies the theory for transactions within a single service. The second, Bank-transfer, does so for inter-service transactions (IST). An overview is now provided as a means of introducing the fundamental notions of scope, IST and critical variable.

**Batch-Atm**

In Batch-Atm, shown in Fig. 2, a collection of operations is processed on a given bank account. Batch-Atm firstly receives a request from the client that contains the account and a bundle of operations with amounts. After initialization, the back-end work (WorkBack) starts. It consults the bank service (Check) to check the balance of the account. If the balance is non-negative the account is declared valid and Batch-Atm continues to handle the operations in the bundle one by one (DoOp) until all have been committed, in which case Batch-Atm completes by replying to the client with the final balance; otherwise, Batch-Atm ends and replies to the client with an error message pertaining to invalidity of the account. As it handles each operation, Batch-Atm interacts with the bank, updates variable balance, and removes the operation request from bundle.

It is not reasonable to assume Batch-Atm can recover everything when handling faults. For example, the bank, as a business partner service, logs every transaction; Batch-Atm cannot force the log be erased. Naturally, as an ATM service, every transaction operates on the customer's balance. When a transaction fails, the balance must be restored—because the customer would not want to pay for the error of the machine or of the bank. Therefore, variables such as the account balance should be made critical to the transactions in Batch-Atm.

There are four scopes in Batch-Atm. The top-level scope Atm declares variables balance and bundle, interacts with the client and initializes variables such as balance := 1. The scope WorkBack groups all back-end work with critical variable balance and uses scopes Check and DoOp. The scope Check checks the balance from the bank service and has balance as a critical variable. Finally the scope DoOp handles each of the client operation requests and consists of actions ‘invoke BankSys’ and ‘update balance pop bundle’; its variables bundle and balance are critical.

**Bank-transfer**

The process Bank-transfer, shown in Fig. 3, is choreographed from services Payee, Payer and Bank. Payee receives the bank-transfer request btreq from the environment, the client process, and first checks the balance. Then Payer is invoked to transfer the amount of money requested by btreq. After the transfer finishes, Payer notifies Payee which confirms the balance; in case of error a fault is thrown. The final balance is then returned to the client. Bank is a simplified bank system which handles operations of balance enquiry and account transfer.

Regardless of whether or not a fault occurs, and regardless of where it occurs, Payer and Payee naturally require that either (a) the correct amount of money is successfully transferred to the right account, or (b) the balance of Payer and that of Payee are both kept unchanged. This transaction, called Pay, crosses boundaries of all three services, and involves ReqPay (in service Payee), Payer and BankTrans (in service Bank). It has critical variable Bank_bal, which is also the critical variable of scope BankTrans.
1.3. Related work

An overview of the development of fault handling, from exception handling in programming languages, progressing through Sagas compensation [GMS87] and ACID transactions in databases [GR93] and culminating in business-process modelling [OAS07], has been given by Greenfield et al. [GFJK03]. The case is made that compensation, after evolving through those stages, has been prematurely adopted in web services: a long-running transaction may interact with its environment (the real world) triggering changes for which ‘compensation (in the sense in which it has previously been used) is not enough’. A similar view has been expressed by Korth et al. [KLS90]. Subsequently many approaches have been taken to the problems of compensation, but largely addressing theoretical issues.

Butler et al. [BFN05] formalize compensation of BPEL using the process algebra StAC [BF04]. Subsequently, in [BHF04], they extend that work by considering an appropriate form of compensation that merely ‘will restore the world to a state which is an acceptable approximation to the state that it had before the start of the transaction’ and they propose a calculus of compensations in terms of (trace-based) CSP. Operational semantics of compensable CSP is also discussed by Butler et al. [BR05], and the correspondence between the two semantics is checked by PVS in [RB10b] and proved by induction in [RB10a]. Similar work, but arising instead from Sagas [GMS87] and expressed in operational semantics rather than with algebraic laws, is due to Bruni et al. [BMM05]. A comparison of the two approaches is given in [BHF04]. Operational semantics has also been used by Qiu et al. [QWPZ05] in an analysis of BPEL4WS that reveals shortcomings in the standard (the main problem being compensation of a parallel composition, also addressed by [BBF*05]).

The approach of [BHF04] is extended by work of He et al. [He07, LZH07] which uses the ‘unifying theories of programming’ (UTP) approach to derive compensation laws from laws of reactive programs. The notion of a ‘design’ is enriched with new observations corresponding to ‘enabledness in the presence of a preceding exception’, ‘an exception during current execution’, ‘termination without exception’ and ‘an exception requires rollback to the initial state’, and it is shown that those observations suffice for a calculus of compensations that models the forward and reverse flow of business logic characteristic of BPEL-like languages.

Coleman [Col05] examines the compensation mechanism of BPEL, allowing compensation to be called from fault handlers, for greater flexibility. The present paper, by comparison, follows BPEL in restricting the use of compensation to fault handlers. Extension of Bat, the principles and guidelines to incorporate Coleman’s suggestion is routine. Coleman models compensation by procedure call. Recording correct arguments for such calls actually requires non-trivial logic, especially when default handling (history of committed instances) and ISTs (coordination between services and participating scopes) are involved. The method adopted here documents the difference.

A number of approaches extend existing formalisms to describe and reason about transactional services. Process algebra is a popular choice for avoiding state, as already indicated by the work of Butler and of He. The \( \pi \)-calculus has been used by Liao et al. [LTL05] (untimed) and Zhang et al. [ZLTW11] (timed).
Mendling et al. [MS06] extends the event-driven process chain [MN06] with operators to compose web services. None of those specifically handles compensation. More relevant here is the work of De Vries et al. [DKH10a, DKH10b] which extends CCS to obtain TransCCS for the explicit modelling of the combination of rollback recovery (of full state) and coordinated checkpointing in web services. Proof systems, liveness and safety notions have been proposed, but not guidelines and principles for ensuring compensation. In another modification of CCS, Bruni et al. [BMM04] introduce cJoin, the committed join of long-running transactions with the property that all or none of the transactions commit. An alternative in avoiding state whilst remaining at the program level is to use functional programming. Donnelly and Fluet [DF08] present a version of concurrent Haskell as the setting for their ‘transactional events’ which allow all-or-nothing multi-way synchronisation. However compensation is not considered.

The approaches mentioned so far are to be contrasted with the use of database management systems (DBMS) to handle long-running transactions (see, for example, the work of Wang et al. [WSL10]). In that setting, logging and DBMS middleware are used to recover from failure. The approach considered here places emphasis instead on correctness by design, and requires no work on the part of the DBMS to maintain different granularities of transaction. So the two approaches might be considered complementary.

The OASIS standards family of WS-coordination [OAS09c] (WS-C) and WS-businessactivity [OAS09b] (WS-BA) specify the coordination between services in long-running transactions, which is missing in BPEL (cf. Sect. 12.3 in [OAS07]). Sauter et al. [SM05] compare WS-BA and BPEL, and propose a coordination handler to replace the compensation handler of BPEL. This idea is further developed by Pottinger et al. [PML07] who extend WS-BA to implement BPEL’s fault and compensation mechanisms as well as coordination capabilities between services. Influences of the status of ’external’ long-running transactions to localized scope-based transactions in BPEL are briefly investigated by Koop et al. [KML09]. Recent similar work by Sun et al. [SeKA10] integrates BPEL with the WS-coordination protocols family (WS-AT [OAS09a] and WS-BA) and uses policies to define declarative transactions.

However all that work has provided little support for the design of web-services incorporating compensation mechanisms. The present paper addresses that problem drawing, from that more theoretical work, the assumption that fault-handling restores only critical (instead of all) variables. Moreover, it extends the results (as far as possible) to ISTs, reflecting the help they offer the designer as a way of defining long-running transactions that cross service boundaries.

1.4. Outline

The rest of the paper is organized as follows. The language $\Lambda_T$ of transactional processes is introduced in Sect. 2. The execution of transactional processes is given in Sect. 3 for the single-service special case. The semantics is formalised as a labeled transition system, specified using the notation $Z$. Based on the semantics, Sects. 4 and 5 propose transactional principles and design guidelines for transactional processes with a single service. The behavioural properties of transactional processes are presented in Sect. 6 and correctness of the guidelines is proved. Choreographed transactional processes and ISTs are discussed in Sects. 7 and 8, with coordination being described in Sect. 7, and the principles as well as guidelines for IST being given in Sect. 8. The transactional principles (for both single-service transactions and IST) are validated using the model checker SAL in Sect. 9. Complete semantic detail appears in the appendix.

Throughout, the symbol $\overset{\text{def}}{=} \,$ is used to mean equals by definition.

2. A language, $\Lambda_T$

In this section, the language $\Lambda_T$ (for BPEL-like Abstract Transactions) is introduced to model long-running transactions that use scope-encapsulated fault handling and compensation. It is motivated by BPEL but designed to be a little more abstract. $\Lambda_T$ is not proposed as a new transactional language, but to facilitate study. Thus it is compared in this section with other languages for modeling compensable processes.

To facilitate the presentation, the following are assumed.

- A (countably) infinite set, $SName$, of scope names.
- A product type, $Var$, consisting of the tuple of program variables; the positive length of the tuple is unconstrained. The type of variable $x$ is $\text{typ}(x)$. 
A product type, \( Exp \), consisting of all well-typed expressions formed from all required function and constant symbols.

A (meta-) type, \( Type \), containing all types of data and messages.

Figure 4 collects further basic notation used in \( \text{BaT} \).

### 2.1. Syntax

A (choreographed transactional) ‘process’ consists of a finite collection of services which communicate via asynchronous message passing. A service is enabled when the process starts execution. In \( \text{BaT} \), single-service transactions, of which services are a special class, are represented by the scope construct, and ISTs, which span several services, are formed by grouping scopes from different services.

Accordingly, the type \( \text{BaT} \) of processes is defined to consist of all nonempty finite sets of services

\[
\text{BaT} ::= \| F_1 \text{Service} \tag{1}
\]

where the type \( \text{Service} \) of services consists of scopes which are ‘top-level’ (under the notion of order defined in Sect. 3.1). In general, the type \( \text{Scope} \) is defined

\[
\text{Scope} ::= S : \text{scope } x : X . N [f h F, ch C] \gamma \text{ epocs} \tag{2}
\]

where: \( S \) is the unique scope name; \( x \) is a vector of variable names called the scope variables of \( S \); \( X \) is a vector of types (of the variables in \( x \), thus having the same cardinality as \( x \)); \( N \) is the program representing the scope’s normal activity of \( S \); \( F \) is the program representing the fault handler of \( S \); \( C \) is the program representing the compensation handler of \( S \); and \( \gamma \) is a (possibly empty) finite set of variable names, referred to as the critical variables of \( S \). The scope variables of \( S \) are local to \( S \), that is, to the scope’s component programs: \( N, F \) and \( C \).

Use of vector notation and multiple assignment shows that it suffices to restrict notation to just a single variable and to consider guards with a single free variable. For theoretical purposes it is then convenient to overlook the fact that the single variable is of vector type. In examples several variables are employed, as usual.

Record notation is used to denote the components of a scope. Thus for a scope \( S, S.x \) denotes its scope variable \( x \), \( S.N \) denotes its normal activity, \( S.F \) denotes its fault handler, \( S.C \) denotes its compensation handler, \( S.\gamma \) denotes its set of critical variables, and \( S.V \) denotes the set obtained by flattening its vector of scope variable (\( x \) in Eq. 2).
The program expressing a scope’s component program is of type $Prog$, where

$$
Prog ::= Basic \mid Basic;\; Prog
$$

$$
Basic ::= x := e \mid !(snd, rcv, e) \mid ?(snd, rcv, x) \mid if\; b\; then\; P \;fi \mid do\; b\; then\; P \;od \mid
\text{throw} \mid \text{compen}[S] \mid \text{DH} \mid \text{Scope}
$$

There: $x$ is a vector of variables; $e$ is a vector of expressions; $b$ is a vector of boolean expressions; and $P$ is a vector of programs of type $Prog$. The statement $!(snd, rcv, e)$ denotes output of message $e$ from sender $snd$ to receiver $rcv$. Again it is convenient to use record notation and write $(snd, rcv, e)$. The input to variable $x$ of the message from sender $snd$ to receiver $rcv$. The other three well-formedness conditions are standard in sequential or reactive programming. The other three must have been declared in some ‘super’ scope of $S$, that is, in a scope which contain $S$. For instance, $\text{fault}(e)$ is used in place of $e$ where $\text{fault}$ is a special constructor. When the receiving service picks up such a message with content $\text{fault}(e)$, the value of $e$ is assigned to a designated variable and the receiving service considers that it has encountered a fault. Additionally, to facilitate that one service may report to another service a fault together with detailed information such as an error code, $\text{fault}(e)$ is used in place of $e$ where $\text{fault}$ is a special constructor. When the receiving service picks up such a message with content $\text{fault}(e)$, the value of $e$ is assigned to a designated variable and the receiving service considers that it has encountered a fault. For instance, $\text{fault}(-1)$ means that $-1$ is sent to the receiving service $\Psi'$ and at the same time, upon receiving this message, $\Psi'$ acts as if it has encountered a fault with error code $-1$.

Program notation is adapted from the guarded-command language. Recall that $x := e$ denotes multiple assignment. The conditional $if\; b\; then\; P \;fi$ (where $P$ and $b$ have equal arities) executes a program in $P$ whose guard is $true$; the program in $P$ is nondeterministically chosen if more than one guard in $b$ is true. The loop $do\; b\; then\; P \;od$ executes, at each iteration, a program in $P$ whose guard is $true$ (the program in $P$ is nondeterministically chosen, when more than one guard in $b$ is true); if no guard holds the loop terminates. Execution of $\text{throw}$ casts a fault and the corresponding fault handler is triggered. $\text{DH}$, when called in the scope $S$, represents calling the default fault or compensation handler by compensating all committed scopes within $S$ in reverse order of their commitment. A designer may call the compensation of a specific scope using $\text{compen}[S]$. As in BPEL, use of $\text{DH}$ and $\text{compen}[S]$ is confined to fault and compensation handlers. Scope-specific compensation $\text{compen}[S]$ may be mixed with default handling $\text{DH}$ in a scope’s fault handler or compensation.

Reflecting its purpose, BaT does not include various operators for orchestration which would compound its semantics without benefit in the current context. Such operators would facilitate orchestration, but that is not the purpose here and their absence does not decrease the expressive power of BaT. Examples are parallel composition within a service, and input-guarded choice. The former can be expressed as a service without synchronisation by using the laws of parallel composition to express the service in ‘normal form’ without parallel, as in process algebra. The latter, corresponding to BPEL’s $\text{pick}$ activity, can be expressed in terms of conditional, again as in process algebra.

A process $P : \text{BaT}$ of type as Eq. (1) is said to be well-formed if

1. each service in $P$ has no free variables; that is, each variable has been declared;
2. each expression is well-defined and free of type error;
3. for any output $!m$ or input $?m$ appearing in a service $\Psi$ of $P$, $m$.snd $= \Psi[m.rcv = \Psi]$ and $m$.snd $\neq m.rcv$;
4. every scope in $P$ has a unique scope name;
5. for every scope $S$, $S, y \subseteq \text{vars} \; (\text{Recall that vars} \; P \; \text{denotes the set of free variables of program} \; P)$;
6. $\text{DH}$ appear in only fault or compensation handlers, and for every scope $S$, $\text{compen}[S]$ in only the fault or compensation handler of the parent scope of $S$.

The first three well-formedness conditions are standard in sequential or reactive programming. The other three conditions are for scopes. Names are used to identify scopes, so that every scope has a unique name. Each critical variable of a scope $S$ must have been declared in some ‘super’ scope of $S$, that is, in a scope which contain $S$. As a result, the scope variables of $S$ and its sub-scopes are excluded from being critical variables of $S$. The request of compensation is not allowed in normal activity, following implementation languages like BPEL. From now on all processes are assumed to be well-formed.

BaT programs need not be deterministic, and indeed are closed under the nondeterministic choice combinator which may be defined in terms of conditional:

$$
P \sqcap Q ::= \text{if} \,(\text{true, true}) \; \text{then} \,(P, Q) \,\text{fi}.
$$
Compensation by design

In the examples in the remainder of the paper, syntactical sugar is used to simplify code. A vector assignment \( x := e \), where the vectors \( x \) and \( e \) are for example \((x_0, x_1)\) and \((e_0, e_1)\) respectively, is written as a multiple assignment: \( x_0, x_1 := e_0, e_1 \). As usual, \( \text{skip} \) abbreviates \( x := x \). A conditional \( (\text{if } b \text{ then } P \text{ else } Q) \), in which \( b \) is the vector \((b_0, \neg b_0)\) and \( P = (P_0, P_1) \), is written in ‘if \( \text{else} \)’ form as \((P_0 \text{ if } b_0 \text{ else } P_1)\). A loop \((\text{do } b \text{ then } P \text{ od})\), in which \( b = (b) \) and \( P = (P) \), is written \((\text{do } b \text{ then } P \text{ od})\).

2.2. Language comparison

The language BAT has been designed to abstract BPEL [OAS07] placing emphasis on its notion of scope, and allowing expression of what is achieved by the mechanisms suggested in the more theoretical work on compensation, like compensable CSP (cCSP) [BHF04]. The three languages are briefly compared to identify the origins allowing expression of what is achieved by the mechanisms suggested in the more theoretical work on compensation:

Conditional branches, loops, and message-passing statements, structures which appear in both BPEL and cCSP, derive from the guarded-command language and reactive programming. Statements related to intra-service transactions are borrowed from BPEL. The syntax for scopes closely resembles the definition in BPEL (activity \(<\text{scope}>\)). Nesting of scopes, an extension to Sagas [GMS87], are allowed as in BPEL and cCSP. Also, in Sect. 7, scopes in different services can be organized to cooperate in an inter-service transaction. The BAT statement \( \text{compensate}[S] \) corresponds to the BPEL activity \(<\text{compensateScope target}='S'>\), and the statement \( \text{if} \) \( \text{then} \) \( \text{else} \) \( \text{fi} \) corresponds to \(<\text{compensate}>\). To compare with the notions in cCSP: given a scope \( S \), \( S.N \) and \( S.F \) constitute an interrupt handler; \( S.N \) and \( S.C \) form a compensation pair; and, a scope is a transaction block. The execution of a scope’s normal activity always yields to interruptions (faults). Also for statement \( \text{throw} \), its counterpart in BPEL is activity \(<\text{throw} \text{and in cCSP } \text{THROW}>\).

Critical variables.
BAT has been designed to express the idea from theoretical work such as cCSP that a transaction should exhibit the all-or-nothing property, and the behavior exhibited by an action followed by its compensation is the same as \( \text{skip} \). In that way it expresses what the mechanisms expressed in cCSP achieve. Data are manipulated in BAT via variables, and \( \text{invocation} \) to other applications beyond the process in question are supported. With the complexity caused by realistic data under the open-world assumption it is \textit{impractical} to require a transaction to restore the ‘world to a state before the start of the transaction’ [BHF04]. The reasons are:

- It is not reasonable to assume there is a cancellation for every action: some actions may involve non-trivial calculations whose effect cannot be simply canceled; some actions may use variables whose values cannot be determined statically, such as those depending on messages.
- It is not necessary to restore variables used only to assist the design (such as some email to be sent) or to log process status.

Therefore, certain variables should be \textit{identified by the designer} as being critical to the business logic, and the transactional properties required to hold on only those critical variables.

The idea of making some variables ‘more important’ can also be found in the methodology of artifact-centric business-processes modeling [BHS09]. Data that carry important business logic are regarded as ‘business entities’ in which not only the data structure information is recorded but also the process execution is defined as the evolution of these artifacts. Naturally, business transactions must exhibit the transactional property on these variable, such as atomicity and consistency, while the state of other process data may, from this point of view, be disregarded.

Parallel composition of services.
Parallel compositions are supported in BAT. Although there is no parallelism within each service, transactional processes are constructed by parallel composition of (choreographed) services. This abstraction does not affect the expressiveness of the language, because intra-service parallelism with shared variables can be modeled by inter-service parallelism with message passing. Extra control variables and message exchanges may be necessary. That design decision is taken because parallel composition of services is typical in business processes.
Fig. 5. Service \textit{Atm} defined as a scope in the language \textit{Bat} where () denotes an empty vector.

\begin{align*}
N_{Atm} & := \ ?c1ntreq ; \\
\text{req, balance} & := \ c1ntreq.msg, -1 ; \\
\text{WorkBack} & := \ \textit{scope} () \cdot N_{WB} [fh \ DH, ch \ DH] \{balance\} \textit{epocs} ; \\
\text{Fin if balance} & \geq 0 \ \text{else} \ Err \\
N_{WB} & := \ \textit{Check} : \ \textit{scope} () \cdot N_{Check} [fh \ F_{Check}, ch \ C_{Check}] \{balance\} \textit{epocs} ; \\
\text{Ops if balance} & \geq 0 \ \text{else} \ \text{skip} \\
\text{Opt} & := \ do \ #\text{req.bundle} > 0 \ \text{then} \\
& \ \ \ \ \text{size} := \ #\text{req.bundle} ; \\
& \ \ \ \ \text{DoOp} : \ \textit{scope openant} : Z \cdot N_{Op} [fh \ F_{Op}, ch \ C_{Op}] \{\gamma_{Op}\} \textit{epocs} ; \\
& \ \ \ \ \text{throw if} \ (\text{size} = \ #\text{req.bundle}) \ \text{else} \ \text{skip} \\
\text{Fin} & := \ ![\text{Atm}, c1ntreq.snd, balance] \\
\text{Err} & := \ ![\text{Atm}, c1ntreq.snd, fault(-1)]
\end{align*}

Vectors.
Vector notation for variables is convenient for theoretical purposes (since it results in a ‘single variable’), although in examples traditional notation is used. The use of vectors in guarded conditions can simplify the modeling of non-determinism. Vectors of conditions in loops and conditionals help to make their structure more uniform (since loops are interpreted as conditional recursions).

Minimalist syntax.
In order to highlight the problem in question, the syntax of \textit{Bat} is minimalist, abstracting execution-specific details. Thus some structures in languages such as BPEL are indirectly supported or not considered.

‘Input guarded choices’, activities like \texttt{<pick>} or even the change of installed compensation can be modeled in \textit{Bat} using auxiliary control variables and conditional branches. For example, for ‘input guarded choices’ and \texttt{<pick>}, a received message can be stored in some message variable, then the concrete type and message content can be checked in the guard. The type of message variables is a union of the types of messages received in the \texttt{<pick>}. For example, that type is \((A \times B \times X) \cup (C \times B \times Y))\) for receiving in \(B\) either from service \(A\) or \(C\) with the content of the type \(X\) or \(Y\), respectively. Change of compensation can be modeled similarly with a control variable recording the rules for choice of compensation handler. Such variables can be set, for example, by an input message.

Different iterative structures in BPEL, such as \texttt{<repeatUntil>} and \texttt{<forEach>}, can be modeled as loops. As already explained, intra-service parallelism in BPEL, such as \texttt{<flow>} and \texttt{<eventHandlers>}, can be modeled using inter-service parallelism. Termination in BPEL is treated in \textit{Bat} as interruption, that is, as a special fault which can be handled by fault handlers. Fault handling according to specific fault type is supported, as for \texttt{<pick>}, by using fault-type control variables and conditional branches on the fault types (recorded, e.g., in some control attribute). The \texttt{<rethrow>} activity is supported by control variables set in the fault or compensation handler and conditional branches on these control variables.

Process recursion is not supported, as in BPEL: a scope cannot be defined recursively (in terms of itself). Also time or duration-related features, such as \texttt{<wait>} or \texttt{<onAlarm>}, are not supported because in most cases they are not important to transactions.
2.3. Scope example

**Batch-Atm** has been introduced in Sect. 1.2 and depicted in Fig. 2. Like any single-service process, **Batch-Atm** is defined as a scope. It uses a default handler for both fault handling and compensation, and has no critical variables:

\[
\text{Atm : scope } \begin{aligned}
\text{balance : } & \mathbb{Z} \\
\text{req : } & \text{AtmReq, size : } \mathbb{Z}, \\
\text{clntreq : } & \text{CAReq, bsresp : } \text{BAResp.}
\end{aligned}
\]

Because **Atm** is a service, that is a top-level scope, all variables are defined within the scope. Therefore it cannot have critical variables, which means there is no variable required to be restored when handled by **Atm**. But as critical variables are defined for child scopes of **Atm**, faults must be properly handled by compensation of the child scopes.

Although **Batch-Atm** is a single-service transactional process, it may interact with the environment, consisting of the ATM Client and other services that perform some task (consider the model of service orchestration). Suppose that **Atm** uses a bank service **Bank** to accomplish its operations. The messages are defined as follows, where \(\mathbb{Z}\) denotes the type of integers, \(\text{Env}\) denotes the special singleton set of the environment of the processes constructed from **Atm** and **Bank** (i.e. the client to the process **Batch-Atm**), **AtmReq** is a record type consisting of a character string **account** : Charstr and a bag (multiset) of natural numbers **bundle** : bag \(\mathbb{N}\):

- **CAReq** :: \(\text{Env} \times \{\text{Atm}\} \times \text{AtmReq}\)
- **BAResp** :: \(\{\text{Bank}\} \times \{\text{Atm}\} \times \mathbb{Z}\)
- **AtmReq** :: record[account : Charstr, bundle : bag \(\mathbb{N}\)]

As given in Fig. 5 all back-end work in **Atm**, other than interacting with the client and initiating variables, is grouped in scope **WorkBack**, which in turn uses two scopes, **Check** and **DoOp**. The scope **Check** is used to check the validity of the account and update **balance**. It guarantees that in case of fault (generated either internally or by a faulty reply from the bank due, for example, to an invalid account), **balance** is restored to \(-1\) so that execution can jump to the end and provide an error reply to the client.

Let \(\text{invBank} : \text{Charstr} \times \mathbb{Z} \rightarrow \text{Prog}\) be the program which sends a request to **Bank** and waits for a response.

\[
\text{invBank}(\text{str, } n) ::= !(\text{Atm, Bank, [account = str, amount]}); \text{bsresp}
\]

For scope **Check** (Fig. 5), its normal program is \(\text{NCheck}\), fault handler \(\text{FCheck}\) and compensation \(\text{CCheck}\):

\[
\text{NCheck} ::= \text{invBank}(\text{req,account, 0}); \\
\text{balance} ::= \text{bsresp.msg}
\]

\[
\text{FCheck} ::= \text{balance} ::= -1
\]

\[
\text{CCheck} ::= \text{balance} ::= -1
\]

Scope **DoOp** handles one of the user requests, updates both the balance and the bank then removes the committed request from **bundle**. When \(\text{req.bundle}\) is empty (i.e. all operations in **bundle** have been properly handled), the client receives a reply consisting of the final balance, and the process commits. After each iteration of **DoOp**, if the size of **req.bundle** does not change, the operation of **DoOp** fails and a fault is thrown to **Atm**.
Let \( \text{put} : \text{bag } X \times X \rightarrow \text{bag } X \) be a function that puts an element in the bag, \( \text{pick} : \text{bag } X \rightarrow X \) be a function\(^1\) that returns an arbitrary element from the bag, and \( \text{rem} : \text{bag } X \times X \rightarrow \text{bag } X \) removes an element from the bag if the element is in the bag. For scope \( \text{DoOp} \) (Fig. 5) its normal program is \( N_{Op} \), fault handler \( F_{Op} \) and compensation handler \( C_{Op} \):

\[
N_{Op} :\overset{\text{def}}{=} \text{opamnt} := \text{pick}(\text{req.bundle}) \; ; \\
\text{if } (\text{balance} + \text{opamnt} \geq 0) \; \text{else} \; \text{throw} \\
P_{Op} :\overset{\text{def}}{=} \text{invBank}(\text{req.account}, \text{opamnt}) \; ; \\
\text{balance, req.bundle} := \text{balance} + \text{opamnt}, \text{rem(req.bundle, opamnt)} \\
F_{Op} :\overset{\text{def}}{=} \text{F_op} \; \text{if } \text{size} \neq \#\text{req.bundle} \; \text{else} \; \text{skip} \\
C_{Op} :\overset{\text{def}}{=} \text{invBank}(\text{req.account}, -\text{opamnt}) \; ; \; P_{OpH} \\
\text{gamma} :\overset{\text{def}}{=} (\text{req.bundle} \; \text{balance}) \\
P_{OpH} :\overset{\text{def}}{=} \text{balance, req.bundle} := \text{balance} - \text{opamnt}, \text{put(req.bundle, opamnt)}.
\]

3. Execution of single-service processes

This section presents the formal semantics for \( \text{BaT} \), restricted to processes containing only a single service. First an intuitive and informal description of the execution is provided; then a transition-system semantics is formulated using the notation \( Z \).

3.1. Executing a process

Consider the intuition behind the behaviour of a process with a single service in \( \text{BaT} \).

First, notions for scopes are defined. Scopes can be nested. If \( S \) and \( T \) are scopes with \( T \) defined in component programs of \( S \), namely \( S.N \), \( S.F \) or \( S.C \), then \( T \) is a sub-scope of \( S \), and \( S \) a super-scope of \( T \). Note that a scope is regarded as neither a super-scope nor a sub-scope of itself. \( T \) is said to be a direct sub-scope of \( S \) iff \( T \) is a sub-scope of \( S \) and there does not exist a scope \( T' \) such that \( T \) is a sub-scope of \( T' \) and \( T' \) is a sub-scope of \( S \). The set of all directed sub-scope of \( S \) is written \( \text{sub}(S) \). Scope \( T \) is a child scope of \( S \) if \( T \) is defined in \( S.N \) and \( T \) is a direct sub-scope of \( S \). Let \( \text{child}(S) \) denote the set of child scopes of \( S \). If \( T \) is a child scope of \( S \), then \( S \) is called the parent scope of \( T \). The set of descendant scopes of \( S \) is the image of \( S \) under the non-reflexive transitive closure of the ‘child scope’ relation. A scope may be executed several times, for example in a loop. Each execution is called an instance of the scope. A scope (instance) is said to commit if its execution completes without occurrence of a fault or if the fault occurs only in its sub-scopes.

The semantics of \( \text{BaT} \) is explained as follows. A process starts execution by declaring all its services to be ready for execution, letting each of them execute as a scope (see the next paragraph) (note that a receiving command of the form \( ?\text{snd}, \text{rcv}, x \) in a service is blocked until a message from another service \( \text{snd} \) is received).

When execution comes to a scope, an instance of the scope \( S \) is first created. Uninitialised scope variables are assumed to be non-deterministically initialised to a value of their type. Execution of a scope amounts to execution of its normal activity. If no fault (including explicit faults signified by \( \text{throw} \) statement) occurs within execution of the normal activity \( S.N \), then the scope instance completes successfully and commits. If however a fault occurs in the course of the normal activity, then immediately execution of normal activity is ‘terminated’ and the fault handler \( S.F \) is triggered. Committed instances of descendant scopes \( T \) of the faulted or compensating scope \( S \) may be compensated by default handler (\( \text{DH} \)) or by calling the compensation handler of \( T \). When the fault handler finishes execution, the scope instance of \( S \) is considered to have finished its execution.

A fault or compensation handler of a scope \( S \) may contain a statement \( \text{compen}[T] \) where \( T \) is a child scope of \( S \). When \( \text{compen}[T] \) is encountered, compensation of the last committed but not compensated instance of \( T \) is triggered. More precisely, compensation starts by \( T.V \) being initialized at the time of commitment of the most recently committed but not compensated instance of \( T \). Then the program \( T.C \) is executed and after it \( T.C \) terminates, that instance of \( T \) is said to be compensated. If no instance of \( T \) has been committed or all committed instances of \( T \) are all compensated, then \( \text{compen}[S] \) is \( \text{skip} \).

A fault or compensation handler of \( S \) can also use the default handler \( \text{DH} \). Execution of \( \text{DH} \) in \( S.F \) or \( S.C \) automatically calls the compensation of all committed instances of child scopes of \( S \) in reverse order of their commitment.

\(^1\) Properly, \( \text{pick} \) is a relation. However for ease of coding, suppose the implicit nondeterminism is resolved by a hidden parameter, like the seed of a random-number generator. An alternative is to replace the assignment by a specification statement.
The similarity between the scopes introduced abstractly in $\text{BaT}$ and those in implementation languages like BPEL facilitates the support of BPEL-targeted design whilst overlooking unimportant syntactic detail that would otherwise restrict results to only certain target languages.

### 3.2. Transition-system semantics of $\text{BaT}$

Formal transition system semantics for transactional processes (with single services) are given in this subsection. Readers not interested in the semantic formalism may resume the plot in Sect. 4.

From set theory, the cardinality of a finite set or vector is denoted $\#$. If $R$ is a binary relation its transitive closure is denoted $R^*$, its non-reflexive transitive closure is denoted $R^+$, and the forward relational image of a subset $E$ of its domain is

$$R[E] := \{ y \mid \exists x : E \cdot x R y \}. \quad (3)$$

Then given a scope $S$, the set of its sub-scopes is denoted $\text{sub}^*(\{ S \})$ and its descendant scopes $\text{child}^*(\{ S \})$.

The semantics of a (transactional) process is given in terms of a labeled transition system (LTS). The LTS semantics$^2$ of service $\Psi$ is denoted $\text{TS}(\Psi)$ the complete semantics of processes with multiple services and inter-service transactions appears in Appendix A.

The following semantic artifacts are helpful. They are not intended as an extension to $\text{BaT}$, whose simplicity is advantageous, but in formulating the semantics.

- **Scope**: Each scope $S$ is augmented to make the end of its component programs explicit by appending ‘$\mathcal{g}$ end’ to each component. That is, if

$$S : \text{scope } X : \mathcal{X} \cdot N [\text{fh } F, \text{ ch } C], \text{ epocs},$$

then

$$S^\ddagger : \text{scope } X : \mathcal{X} \cdot (N^\ddagger \mathcal{g} \text{ end}) [\text{fh } (F^\ddagger \mathcal{g} \text{ end}), \text{ ch } (C^\ddagger \mathcal{g} \text{ end})], \text{ epocs}$$

where for any component program $P$ of $S$, $P^\ddagger$ denotes the program by replacing in $P$ any scope $T : \text{sub}^*\{ S \}$ with $T^\ddagger$. The resulting type is written $\text{Scope}^\ddagger$. Thus if $S^\ddagger : \text{Scope}^\ddagger$, then $S^\ddagger.N$ (respectively $S^\ddagger.F$ and $S^\ddagger.C$) denotes $(S.N)^\ddagger \mathcal{g} \text{ end}$ [respectively $(S.F)^\ddagger \mathcal{g} \text{ end}$ and $(S.C)^\ddagger \mathcal{g} \text{ end}$]. The ‘top level’ ends are required because $S$ is not a sub-scope of itself. To generate the transition-system semantics of a given service $\Psi$, all scopes $S$ in $\Psi$ are extended to the corresponding $S^\ddagger$.

- **eop**: symbol $\text{eop}$ is introduced to represent the end of a program.

- **comp[S$^\ddagger$]**: is used to stand for compensating scope instance $S^{\text{id}}$ in the semantic specification.

The end construct is introduced in order to mark explicitly the end of a scope’s component program. As a result the transitions, specified by the schemas $\text{ExitNorm}$, $\text{ExitFH}$ and $\text{ExitCH}$ in the following, do not need to refer to the program still to be executed (i.e. $\wp$ in the state space definition in the following) to see if the scope component program ends. Notation $\text{eop}$ is to mark the end of the execution. Thus in specifying the semantics, there is no need to consider the case that the remaining program to execute is empty and from the following specification of transitions, when $\text{eop}$ is to be executed (head in $\wp$), no defined transition schema is enabled. Furthermore, according to the specification of transitions, $\text{eop}$ is added to the program in case the program to be executed contains only one statement in $\text{Basic}$ or is either end or $\text{eop}$ and when dealing with inter-service transactions (cf. Sect. 7.4).

In the following, the type $\text{Basic}^\ddagger$ denotes the extension of $\text{Basic}$ to include the constructs end, $\text{eop}$, and $\text{comp}[S^{\text{id}}]$, after replacing $\text{Scope}$ by $\text{Scope}^\ddagger$. Analogously,

$$\text{Prog}^\ddagger := \text{Basic}^\ddagger | \text{Basic}^\ddagger \mathcal{g} \text{ end} \mathcal{g} \text{ Prog}^\ddagger.$$  

Since (extended) programs are generated from $\Psi$ only in the semantics, ‘well-formedness’ conditions are not given for $\text{Prog}^\ddagger$. Given an (extended) program $P : \text{Prog}^\ddagger$, all statements of $\text{Basic}^\ddagger$ are called program steps in $P$.

$^2$ The semantics $\text{TS}(\Psi)$ of a single-service process $\Psi$ can be seen as a special case of the semantics $\text{TS}(\| \Psi \|$ of a choreography $\| \Psi \|$, where $\Psi$ is a nonempty set of single-service transactions: $\text{TS}(\Psi) := \text{TS}(\| \Psi \|)$. 
As mentioned in Sect. 3.1, a scope may have several instances during an execution. The identifier of an instance is denoted by a pair consisting of the scope name and an instance numeral. For a single service \( \Psi \), the instance of scope \( S : \text{sub}^+ \mid \Psi \) (where \(^\ast\) denotes reflexive transitive closure) identified by \( \text{id} : \mathbb{N} \) is denoted \( S^\text{id} \), \( (\mathbb{N} \text{ denotes natural numbers) and the type of all scope-instance identifiers is denoted } \text{ScopeIns}:

\[
\text{ScopeIns} := \text{Scope} \times \mathbb{N}.
\]

Different scope instances may have different statuses during an execution. The instance status, \( \text{Status} \), enumerates from the following values (borrowed from BPEL, WS-BA, cCSP and Sagas)

- **active**: the instance has started but not yet completed;
- **committed**: the instance has committed but not yet compensated;
- **failed**: the instance encounters some fault in the execution or in the fault handling;
- **fth-hndled**: the instance has completed but with fault;
- **cmpning**: the compensation handling is triggered and running;
- **compensed**: the instance’s compensation has completed.

The semantics of service \( \Psi \) is again written \( TS(\Psi) \). It is defined on state space \( \text{State} \), with initial values \( \text{Init} \), and transitions \( \text{Transit} \) each described, for convenience, using the specification language \( Z \) [Spi92]. Recall that \( \text{Type} \) denotes the type of all data and message types. Let

\[
\begin{align*}
\text{Val} & := \bigcup \{ S, V \rightarrow \text{Type} \mid S \in \text{sub}^+ \mid \Psi \} \\
\text{Msg} & := S\text{Name} \times S\text{Name} \times \text{Type} \\
\text{Msg}^\dagger & := \text{Msg} \times \text{ScopeIns} \\
\text{ScopeStat} & := \text{ScopeIns} \rightarrow \text{Status} \\
\text{Snapshot} & := \text{ScopeIns} \times \text{Val} \\
\text{SnapHist} & := \text{seq} \text{Snapshot} \\
\text{EncScope} & := \text{Scope} \cup \{ \text{none} \} \\
\text{RunningStat} & := \{ \text{active, failed, cmpning} \}.
\end{align*}
\]

It is assumed that messages are associated with scope instance information. That is, the elements in the message queue, called marked messages, are of type \( \text{Msg}^\\dagger \). For any marked message \( m^\dagger : \text{Msg}^\\dagger \), the corresponding message is written \( \text{Msg}(m^\dagger) \) and the notation \( m^\dagger.x.\text{cv} \) is used for \( \text{Msg}(m^\dagger).x.\text{cv} \). Similar notation applies to \( m^\dagger.x.\text{snd} \) and \( m^\dagger.x.\text{msg} \). Given a message variable \( m \) and a scope instance \( \text{sins} \), the corresponding marked message is denoted \( (m, \text{sins}) \). Analogous to the notion of snapshot in BPEL, the data structure \( \text{Snapshot} \) is intended to store information about a committed scope instance that may be needed when this instance is to be compensated later.

If \( x \) is a sequence or a vector of length \#x, its \( i \)th element, for \( 0 \leq i < \#x \), is denoted \( x_i \). Indexing begins from 0.

The state space \( \text{State} \) is formalised in \( Z \) by schema as in Fig. 6, which consists of a ‘signature’ (declaration of all observables with their types) and a ‘body’ (capturing the invariant relationship between the observables). The observables are:

- \( \varphi : \text{Prog}^\dagger \), representing the remaining program to execute;
- \( \text{val} : \text{Val} \), representing a valuation, assigning a value to each variable;
- \( \text{mq} : \text{seq} \text{Msg}^\dagger \), representing the queue of outstanding marked messages;
- \( \text{ident} : \mathbb{N} \), is the counter of scope instance identifiers, that is the maximum numeral used to identify scope instances;
- \( \text{status} : \text{ScopeStat} \), representing the status of each commenced scope instance;
- \( \text{hist} : \text{SnapHist} \), representing a sequence of all committed scope instances not yet compensated; and
- \( \text{enc} : \text{EncScope} \), representing the enclosing scope responsible for fault handling; but if \( \text{none} \) then the fault is ignored.

The quantification \( \exists_i \) is defined as exactly one in \( Z \), and \( \exists_{\leq 1} \) means at most one:

\[
\exists_{\leq 1} e \cdot p := (\exists_i e \cdot p) \lor (\forall e \cdot \neg p).
\]
Compensation by design

Fig. 6. The state space of the semantics $TS(\Psi)$ of a single-service process $\Psi$. In writing schemas, indentations are used to minimise use of conjunctions and parentheses. Note the difference between the identifier $id : N$ and the identity function $id$.

For $P : Prog^\dagger$, the head program step, $phead(P)$, and tail program, $ptail(P)$, are defined syntactically:

1. if $P : Basic^\dagger$, then $phead(P) ::= P$ and $ptail(P) ::= \text{cop}$;
2. if $P = R ; Q$ for $R : Basic^\dagger$ and $Q : Prog^\dagger$, then $phead(P) ::= R$ and $ptail(P) ::= Q$.

The body of Schema $State$ (shown in the part below the horizontal line, where $\text{dom } f$ denotes the domain of $f$ and $(f \circ g)(x) ::= f(g(x))$) captures the following relationships between observables:

- for any scope there is at most one instance of status active or failed, and there is at most one instance of status cmpning;
- for any scope instance in status and hist, its instance identifier lies in the interval $[0, \text{idcnt})$;
- if there is an instance of a ‘belonging scope’ of $phead(\phi)$ which is of status committed, then $phead(\phi)$ is in the compensation of either the belonging scope itself or one of its sub-scopes. Given a program step $ps$ and scope $S$, $ps$ is said to belong to scope $S$ iff $ps$ is in a component program of $S$ or one of its sub-scopes. Furthermore, $ps$ is said to belong exactly to scope $S$ iff it belongs to $S$ but to no (proper) sub-scope;
- any scope instance whose snapshot lies in hist is of status committed;
- $\text{enc} = \text{none}$ iff $phead(\phi)$ is not in the normal activity program of the exactly belonging scope.

Given a program step $ps$ and scope $S$, $Belong(ps)$ denotes the set of scopes to which $ps$ belongs. Furthermore, the exact-belonging scope of $ps$ is

$$\text{exbelong}(ps) ::= S \in Belong(ps) \land \exists T : \text{sub}(S) \land T \notin Belong(ps).$$

Furthermore, for any program step $ps$, $\text{inprog}(ps) = S.N$ [respectively $\text{inprog}(ps) = S.F$ and $\text{inprog}(ps) = S.C$] means that $ps$ is a program step in $S.N$ and $\text{exbelong}(ps) = S (ps$ exactly belongs to $S$) (respectively $S.F$ and $S.C$).

Let $\Psi$ be the service in a single-service process. Initially, the whole service remains to execute. The valuation $\text{val}$ is initialized so that every variable in $\Psi$ is assigned nondeterministically an arbitrary value of its type. Numeral $\text{idcnt}$ is zero. There is no enclosing scope entered yet, so $\text{enc} = \text{none}$. The initial value of $mq$ is a sequence of arbitrary messages (from the environment) to $\Psi$. All other state observables are initialized to be empty. Let $\text{Init}$ be the set of all such initial states. (Schema $\text{Init}$ with the support of service parallel composition and inter-service transactions appears in Appendix A.)
A transition of $TS(\Psi)$ occurs when the state changes as a result of executing the current program step (i.e., the head program step of $\varphi$) or dealing with a fault. There are more than a dozen cases to consider, as summarised by Schema Transit:

\[
\text{Transit ::= Assign \lor Send \lor Receive \lor Choices \lor Loop \lor Throw \lor FaultEnv \lor AScope \lor CompensateScope \lor CompensateScopeIns \lor DefaultHandler \lor ExitNorm \lor ExitFH \lor ExitCH} \tag{4}
\]

Each of those 14 schemas specifies a kind of transition, resulting from the handling of the various program steps. In specifying the transitions using Z, it is convenient to use if - then - else as shorthand for the appropriate conjoined implications (or disjoined conjunctions).

Schemas Assign, Choices, Loop and Send are from sequential and reactive programming. Consider next schemas for the transitions dealing with other transactions (i.e., scopes for this section).

Recall that the constructor $\text{fault}(e)$ results in fault by $e$. Given expression vector $e$, $\text{fault}(e)$ holds iff some $e_i$ (for $0 \leq i < \#e$) is constructed by $\text{fault}(e_i)$. Let $\text{typ}(x) : \text{TYPE}$ denotes the type of variable $x$. In $Z$, $X \setminus Y$ is the concatenation of sequences $X$ and $Y$; relation $X \oplus Y$ relates everything in dom $Y$ to the same objects as $Y$ does, and everything else in the dom $X$ to the same objects as $X$ does; and given a nonempty sequence $X$, head $X$ is its first element and tail $X$ is its tail sub-sequence $(\text{head } X) \⌢ \text{tail } X = X$. front $X$ is its front sub-sequence and last $X$ is its last element $(\text{front } X) \setminus \langle \text{last } X \rangle = X$.

Schema Receive specifies the receipt of messages from the environment of $\Psi$ (see Fig. 7). The last message in the message queue must match the receiver and type of $m$. Then message queue is updated as its front segment, and valuation of variable $m$ is set to the message content of the last element of the message queue. The only difference from standard reactive programming occurs if the received message is a fault; then throw is inserted at the head of the next program to execute.

To define the schema for handling fault thrown, function afterend : $\text{Prog}^+ \rightarrow \text{Prog}^+$ is defined to return the program after the first end:

\[
\text{afterend}(P) ::= \begin{cases} 
\text{ptail}(P) & \text{if } \text{phead}(P) = \text{end} \\
\text{eop} & \text{if } \text{phead}(P) = \text{eop} \\
(\text{afterend} \circ \text{ptail})(P) & \text{if } \text{phead}(P) \notin \{\text{end}, \text{eop}\}.
\end{cases}
\]

Schema Throw specifies the handling of throw (Fig. 8). If $\text{enc} = \text{none}$, current execution is in the fault or compensation handler of some scope or in an initial state, the fault is ignored and execution continues. Otherwise, $\text{enc}$ is set to none, the normal activity of the scope (as $\text{enc}$) is interrupted and terminated, its fault handler starts. That is, the program-to-execute, $\varphi$, is set to the fault handler of the enclosing scope, $\text{enc}$, and followed by the programs after the enclosing scope. The instance status is changed from active to failed. Outstanding messages sent by the failed scope instance are flushed from the message queue (by using the range anti-restriction operator $R \setminus \{y \mid (x, y) \in R \}$ and $y \notin Y$).

Schema FaultEnv specifies environmental fault by inserting throw at the head of the next program to execute (i.e., $\varphi' = \text{throw} \circ \varphi$). The complete schema is not included in this section but can be found in Appendix A (with support of service parallelism and inter-service transactions).
Compensation by design

**Fig. 8.** Schema *Throw* for handling of a *throw*

<table>
<thead>
<tr>
<th>ΔState</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \exists \text{id} : \mathbb{N}; \ S : \text{Scope} \bullet )</td>
</tr>
<tr>
<td>( S = \text{enc} )</td>
</tr>
<tr>
<td>( \text{phead}(\varphi) = \text{throw} )</td>
</tr>
<tr>
<td>( \text{status}(S^{id}) = \text{active} )</td>
</tr>
<tr>
<td>( \varphi' = S^{1} \cdot N ; ; ; \text{ptail}(\varphi) )</td>
</tr>
<tr>
<td>( \text{status'} = \text{status} \oplus { S^{id} \rightarrow \text{failed} } )</td>
</tr>
<tr>
<td>( \text{enc'} = S )</td>
</tr>
<tr>
<td>( \text{idcnt'} = \text{idcnt} + 1 )</td>
</tr>
<tr>
<td>( \text{val'} = \text{val} \land \text{mq'} = \text{mq} \land \text{hist'} = \text{hist} \land \text{wait'} = \text{wait} )</td>
</tr>
</tbody>
</table>

**Fig. 9.** Schema *AScope*: initiating execution of a new scope instance

<table>
<thead>
<tr>
<th>ΔState</th>
</tr>
</thead>
<tbody>
<tr>
<td>( S : \text{Scope} )</td>
</tr>
<tr>
<td>( \text{phead}(\varphi) = S )</td>
</tr>
<tr>
<td>( \varphi' = S^{1} \cdot N ; ; ; \text{ptail}(\varphi) )</td>
</tr>
<tr>
<td>( \text{status'} = \text{status} \oplus { S^{idcnt} \rightarrow \text{active} } )</td>
</tr>
<tr>
<td>( \text{enc'} = S )</td>
</tr>
<tr>
<td>( \text{idcnt'} = \text{idcnt} + 1 )</td>
</tr>
<tr>
<td>( \text{val'} = \text{val} \land \text{mq'} = \text{mq} \land \text{hist'} = \text{hist} \land \text{wait'} = \text{wait} )</td>
</tr>
</tbody>
</table>

Schema *AScope* specifies entry to a scope (Fig. 9). An instance identifier is assigned to the new scope instance using the numeral *idcnt*; the status of the scope instance is set to *active*; *enc* is set to the scope itself; and on the next state, *φ* is set to be \( S^{1} \cdot N \) followed by (extended) programs following the scope *S*, and *idcnt* is increased by one.

To facilitate the definition of the schemas handling scope compensation, the notation \( \text{hist'}_{\text{Set}} \) is defined for a sequence *hist* of snapshots and a scope set *Set*. \( \text{hist'}_{\text{Set}} \) gives a new sequence of snapshots of scopes in *Set* in the same order as *hist*.

\[
(\emptyset)_{\text{Set}} := \emptyset
\]

\[
(((T^{id}, \text{val})) \cdot \text{hist})_{\text{Set}} := \begin{cases} \text{if } T \in \text{Set} \text{ then } ((T^{id}, \text{val})) \cdot \text{hist} \big|_{\text{Set}} & \text{else } \text{hist} \big|_{\text{Set}} \end{cases}
\]

Schema *CompensateScope* specifies handling of *compen*[S]. If *hist* contains no snapshot of an instance of *S* then *compen*[S] = \( \text{skip} \). Otherwise, the next program to execute is *compen*[*sins*], which is to compensate the first instance *sins* of *S* in *hist*, and then the (extended) program following *compen*[S]. The compensation of a scope instance is specified by schema *CompensateScopeIns*. When compensating a scope instance, \( S^{\text{†}} \cdot C \) is executed; valuations in the first snapshot of *S* are added to *val*; and the instance status is changed from *committed* to *compenning*. As for *Throw*, messages sent by the to-be-compensated scope instance are flushed from the message queue if they have not yet been handled. These two schemas are specified in Fig. 10.

When specifying the default fault/compensation handler, two functions are necessary. The function *callch* generates from the input snapshot the compensation of the corresponding scope instance.

\[
\text{callch} : \text{Snapshot} \rightarrow \text{Basic}^{\dagger}
\]

\[
\text{callch}(\text{snap}) := \text{compen}[\text{first snap}]
\]
Recall that `Snapshot` is a tuple of `ScopeIns` and `Val`, so `first snap` is a scope instance of snapshot. The function `seqlc` gives the transferred sequential programs from the sequence of programs.

\[
\text{seqlc} : (\text{seq}, \text{Prog}) \rightarrow \text{Prog}
\]

\[
\text{seqlc}(ps) := ps(1) \mathbin{; \ldots ;} ps(\#cmp)
\]

`DefaultHandler` specifies default handling: compensate all committed but not-yet-compensated child scope instances of the exact-belonging scope in the reverse order of commitment (Fig. 11). Given an exact-belonging scope \(S\) of the current program step, default handling is achieved by changing \(\varphi\) to start with a sequential composition of the `compensate` whose scope is a child scope of \(S\), and the order of `compensate` is the same as that of the corresponding snapshots in `hist`.

Schemas specify the exit from the current scope component is enabled when \(\text{phead}(\varphi) = \text{end}\). Valuations of the current scope variables are removed. If the exit is from normal activity (Schema `ExitNorm`) then the status of the instance is changed from `active` to `committed` and the instance snapshot is added to the head of `hist`. If the exit is from the fault handler (Schema `ExitFH`) then the status is changed from `failed` to `flt-hndld`. Finally if the exit is from the compensation handler (Schema `ExitCH`) then the status is changed from `cmpning` to `compened` and the snapshot is removed from `hist`. These schemas are in Fig. 12, in which the domain restriction \(X \triangleleft R\) of a relation \(R\) to a set \(X\) relates \(x\) to \(y\) iff \((x, y) \in R\) and \(x \in X\).
### 3.3. Runs of transition systems

Single-step transitions have been defined in the previous subsection. The behaviour of a single-service process \( \Psi \) is given by its semantics, \( TS(\Psi) \), expressed in terms of the runs generated by \( \Psi \). A run is an infinite sequence of states and transitions, where each transition is generated from the schemas of \( Transit \) and when appropriate, together with actual data as in messages received. A transition is said to be enabled at a state \( s \) if its precondition holds at \( s \).

**Definition 3.1 (run).** A run of \( TS(\Psi) \) is an infinite sequence of states and transitions

\[
\begin{align*}
    r &::= (s_i \xrightarrow{t_i} s_{i+1} | i : \mathbb{N}) \\
\end{align*}
\]

where: \( s_0 : Init \) is an initial state, and for each \( i : \mathbb{N}, s_i : State \) is a state, \( t_i : \{ \tau \} \cup Transit \) is a transition, if \( t_i \neq \tau \) then \( s_i \) enables \( t_i \), and if some \( t \in Transit \) is enabled by \( s_i \), then \( t_i \neq \tau \). Thus deadlock corresponds to the situation in which no transitions are enabled from some state \( s_i \); it is represented by the run with constant tail having transition \( \tau \) and state \( s_i \). The set of all runs of \( TS(\Psi) \) is denoted \( Run(\Psi) \). For the run (5) it is convenient to write \( r(i) ::= s_i \) for the \( i \)th state in \( r \), and to call the sequence containing just its states, \( (s_i | i : \mathbb{N}) \), a state run of \( r \).

The notion of a predicate holding at state \( i \) of a state run of \( r \) is defined as usual. Temporal operators in the sense of LTL [Pnu77] are defined over state runs. Let \( \phi, \psi \) be temporal formulae on state run of \( r \). Then \( X \phi \) holds at state \( i \) of a state run of \( r \) iff \( \phi \) holds in the next state, \( r(i + 1) \models \phi \); and \( \phi U \psi \) holds at state \( i \) of a state run of \( r \) iff \( \phi \) holds until \( \psi \) does in the sense that \( \exists j \geq i, \forall k : [i, j), r(k) \models \phi \land r(j) \models \psi \). Other operators \( G \) (always), \( F \) (eventually) and \( W \) (weak until) are defined as in LTL; and \( r \models \phi \) means \( r(0) \models \phi \).
Finally, the transition system of single-service process $\Psi$, $TS(\Psi)$, is said to satisfy a temporal formula $\phi$, written $TS(\Psi) \models \phi$, i. f. $\forall r : Run(\Psi), r \models \phi$.

Given a state $s : State$ (cf. Fig. 6), record notation is used to refer to a specific state observable. For example, $s.val$ is the variable valuation $val$ on state $s$. In the following, use of a state observable without referral to a state is shorthand for a state function. For example, given a scope instance $sins : ScopeIns$, $status(sins) = \text{active}$ is shorthand for $\forall s : State, s.status(sins) = \text{active}$.

4. Transactional principles for scopes

The syntax of (transactional) processes has been defined via the language $\text{BaT}$. The present and following two sections focus on localized transactions, scopes, and processes having just one service. The results are extended, to processes with ISTs and more than one service, in Sects. 7 and 8. This section introduces the transactional principles which such processes should exhibit. The principles are defined in terms of runs (with semantical notions) defined in Sect. 3.

Notation ($\text{restored}$)

It is convenient first to define the notion of $\text{restored}$. The transaction should restore the value of critical variables after fault handling and compensation. Generalized to a given set of variables, for some scope instance, the restoration is about the value of these variables on two pairs of states in a run of the process: (1) the first state that the instance status changes to $\text{active}$ and the first state to $\text{flt-hndled}$; (2) the first state that the instance status changes to $\text{active}$ and the first state to $\text{compended}$. The value of the variables must be the same on either of the state in each pair—the variable value is restored. This notion is formulated as $\text{restored}$.

The formula $\text{restored}$ expresses the property that for some scope instance $S_{id}$, either

1. there is no state with status $stt : \{\text{compended, flt-hndled}\}$, or
2. the variables in $vs$ at the first state with status $stt$ are restored to the values they had at the first state with status $\text{active}$.

Recall that operator $\text{weak until}$, denoted $\text{W}$, is defined $\phi \text{W} \psi := (\phi \cup \psi) \lor G \phi$. For each $S_{id} : \text{ScopeIns}$, $stt \in \{\text{compended, flt-hndled}\}$ and $vs \subseteq \text{Var}$,

$restored(S_{id}, stt, vs) := \land \{ (S_{id} \not\in \text{dom status}) \land start(S_{id}, stt, x, v) \land x \in vs \land v \in \text{typ}(x) \}$

where

$start(S_{id}, stt, x, v) := \text{status}(S_{id}) = \text{active} \land (val(x) = v \Rightarrow \text{until}(S_{id}, stt, x, v))$

$until(S_{id}, stt, x, v) := (\text{status}(S_{id}) \not= stt) \land (status(S_{id}) = stt \land val(x) = v)$.

In the important case $vs = S.\gamma$, $restored(S_{id}, stt, vs)$ is abbreviated $restored(S_{id}, stt)$. \hfill \Box

Recall that Sect. 1 introduced the intuition behind the transactional principles: the $\text{all-or-nothing}$ principle states that in case a fault occurs in scope $S$ (but not in any of its sub-scopes), no changes are committed to critical variables. The $\text{compensation}$ principle states that if, after commitment of $S$, compensation is triggered (due to failure in a parent scope) then its critical variables are restored by $S.C$.

**Definition 4.1 (transactional principles)** Let $\Psi$ be a service. Scope $S : \text{sub}(\emptyset, \Psi)$ is said to satisfy the transactional principles iff for any $id : \mathbb{N}$

$restored(S_{id}, \text{flt-hndled}) \land \text{(all-or-nothing)}$

$\land$

$restored(S_{id}, \text{compended}) \land \text{(compensation)}$.

The service $\Psi$ is said to satisfy the transactional principles iff every scope in $\Psi$ satisfies the transactional principles. \hfill \Box

In Definition 4.1, both of the all-or-nothing principle and the compensation principle are defined using $restored$ (note that when the third parameter of $restored$ is omitted, it lies in the scope’s critical variable set). As explained in Sect. 1, these two principles specify different features of transactions. All-or-nothing
Compensation by design restricts the design of fault handlers. When a fault occurs during execution of a scope’s normal activity program, the fault handler restores the value of the critical variables to the value at the start of the scope. Viewed from outside the transaction, there has been no change to the critical variables. Compensation is triggered after commitment of the transaction, and is achieved by a program reversing the effect of the committed updates on critical variables. Viewed from outside the transaction, some changes are firstly committed and later ‘reset’. The two principles have been separated for clarity, and it is their conjunction which is desired.

The transactional principles do not restrict application-specific treatments of fault and compensation (such as sending emails or notifying other process participants) as long as the critical variables are not affected. Imposing a penalty instead of requiring recovery is another kind of fault-handling and compensation strategy. For example, a fee may be charged in case of transaction failure, as an alternative to keeping the balance unchanged. When such a penalty does not affect critical variables of other scopes, charging a penalty can be modeled as an additional action taken out of the scope. For example, if the Batch-Atm process charges a service fee and balance is not to be recovered (not a critical variable), the fee can be charged after the scope WorkBack finishes (see Sect. 2.3). As in [GMS87, BHF04, He08], the present work on BaT does not support ad hoc and complex penalties inter-related between scopes because penalty charging is mostly application-specific and therefore requires more ingenuity in process design. The present paper focuses on typical features for business transactions, that is recovery of critical variables in case of fault and requested compensation.

5. Design guidelines

In order to facilitate the design of processes that meet the transactional principles, design guidelines (that is, sufficient conditions) for the transactional principles are proposed in this section. The design guidelines are defined to be statically checkable as much as possible. Moreover, they are designed in ‘recursive manner’ in order to reduce the problem of the satisfaction of transactional principles for complex scopes to that of their (simpler) child scopes. Here only guidelines for scopes of single-service processes are considered; ISTs are studied in Sect. 8. Validation of the principles using bounded model checking with SAL is considered in Sect. 9.

Although the guidelines given in this section are presented on transactional processes with single services, they can also be applied to choreographed processes (i.e. those with a parallel composition of services). The design guidelines are ‘modularised’, with separate guidelines for service choreography. Certain assertions are made on message interactions (i.e. on the message queue), and the guidelines tested under those assertions.

The guidelines are given first for default fault and compensation handling and then for user-defined fault and compensation handling provided the compensations of designer-specified scopes lie in a certain ‘preceding’ relationship. They allow the designer to determine whether or not a scope satisfies the principles by reasoning recursively through child scopes until simple conditions on finer scopes are reached. The design guidelines are explained informally, formalized and then their correctness given in Sect. 6 (using the semantics). But first it is helpful to introduce an auxiliary concept.

5.1. Guideline condition

A common notion, that of ‘guideline condition’, is used in following guidelines, and involves semantical checking. However, as the condition is recursive in nature, the checking can be reduced to progressively simpler scopes until the result is readily comprehended at the design stage.

Given service $\Psi$ and scopes $S, U : sub^*(\Psi)$, the guideline condition $Gl(S, U)$ is satisfied iff predicates $Gl_1(U)$ and $Gl_2(S, U)$ both hold:

- $Gl_1(U)$ holds iff for each child scope $T$ of $U$, $T$ satisfies the all-or-nothing principle and, in addition, either $T$ satisfies the compensation principle, or the combined effect of $T.N$ immediately followed by $T.C$ in $\Psi$ leaves critical variables of $T$ unchanged (see skipon in the following).

- $Gl_2(S, U)$ holds iff in scope $U$ each update to some variable $x$ that may affect the value of the critical variables of $S$ (see rhsP in the following) lies in some child scope $T$ of $S$, and $x$ is a critical variable of $T$. 
\(Gl_1(U)\) involves semantically checking the effect of fault and compensation handling in child scopes of \(U\). \(Gl_2(S, U)\) requires all updates that matter to the transactional principles to be organized in child scopes of \(U\). The guidelines may be checked recursively on scope. To formalize \(Gl_1\) and \(Gl_2\), notations \(\text{skipon}\) and \(\text{rhs}_P\) are used.

**Notation (skipon)**

The predicate \(\text{skipon}(\phi, P, vs)\) captures the notion that the effect of executing program \(P\), starting from an initial global state which satisfies a formula \(\phi\), leaves the values of variables in \(vs\) unchanged while imposing no constraints at all on the values of variables outside \(vs\). The formula \(\phi\) is an assertion over the free variables of \(P\) and the contents of the message queues of \(P\); furthermore \(vs\) is a subset of the set of free variables of \(P\). More precisely, the execution of \(P\) under \(\phi\) is captured by a transition system \(TS(P, \phi)\), defined in the same way as \(TS(\Psi)\) (cf. Sect. 2), except that in any initial global state in \(TS(P, \phi)\), the values of free variables of \(P\) and the contents of the message queues in \(P\) satisfy \(\phi\). Expressed semantically, \(\text{skipon}(\phi, P, vs)\) holds if for every run \(r = r(0) \xrightarrow{t_0} r(1) \xrightarrow{t_1} \cdots \xrightarrow{t_{i-1}} r(i)\) in \(TS(P, \phi)\) with \(r(0)\) being an initial global state (recall that \(\phi\) holds on \(r(0)\)), and \(r(i)\) a state immediately after executing \(P\), for any variable \(x\) in \(vs\), the value of \(x\) in \(r(i)\) equals that in \(r(0)\).

For example, if scope \(S\) is given by:

\[
S.N := (x_1, x_2 := x_1/y_1, x_2/y_2) \quad \text{and} \quad S.C := (x_1, x_2 := x_1 \times y_1, x_2 \times y_2),
\]

where \(x_1, x_2, y_1, y_2\) are real-valued variables, then \(\text{skipon}(y_1 \neq 0), (S.N \cup S.C), \{x_1\}\) holds (where no claim is made about returning \(x_2\) to its initial value because no assumption is made about \(y_2\)).

**Notation (rhs])**

Next, \(\text{rhs}_P\) formulates the intuition that a variable may change the value of another variable in a program. The binary (infix) relation \(\text{rhs}_P\) holds between variables of a program \(P, x \text{ rhs}_P y\), iff \(x\) has ‘direct data dependency’ on \(y\); more precisely iff \(y\) appears in a guard (of a conditional or loop) or on the right-hand side of an assignment in \(P\). Firstly, for an assignment \(P := (x := e)\),

\[
\text{rhs}_P := \{(x_i, y) \mid 0 \leq i < \#x \land y \in \text{vars e}\}.
\]

Secondly for a conditional branch or loop \(P := (\text{if } b \text{ then } Q \text{ else } \text{do } b \text{ then } Q \text{ od})\)

\[
\text{rhs}_P := \bigcup \{\text{rhs}_Q \mid Q \in Q\} \\
\quad \cup \{(x_i, y) \mid 0 \leq i < \#x \land 0 < j < \#b \land y \in \text{vars b}, \text{land } (x := e) \text{ is a sub-program of Q}\}.
\]

Thirdly, if \(P\) is a sequential composition, then \(\text{rhs}_P\) is the union of the \(\text{rhs}_Q\) where \(Q\) is a subprogram of \(P\). Finally if \(P\) is a scope, then \(\text{rhs}_P\) is the union of the \(\text{rhs}_Q\) where \(Q\) is a component program of \(P\). That notation is also lifted, using forward relational image \(\text{rhs}_P(vs)\), to sets of variables, and to sets of programs:

\[
\text{rhs}_P(vs) := \bigcup \{\text{rhs}_P(vs) \mid P \in P\}.
\]

As an example, let \(U\) be a scope with child scope \(S\), such that

\[
U.N := (x := f_1(y)) ; S \triangleright T \\
S.N := (x := f_2(y)) \text{ if } b(z) \text{ else skip} \\
S.F := (x := f_3(y)) ; (y := f_3(v)) \\
S.C := (y := f_3(v)),
\]

and let \(\text{NC}\) be the set of all subprograms of \(U\) except \(S.F\). Then

\[
\text{rhs}_{\text{NC}}(\{x\}) = \{y, z\} \quad \text{and} \quad \text{rhs}_{\text{NC}}(\{x\}) = \{x, y, z, w\},
\]

where (again) * denotes reflexive transitive closure and (\_\_\_) forward relational image; see Eq. (3).

In what follows, a write command of variable \(x\) refers to either an assignment \(x := \text{exp}\) or an input? \((\text{snd}, \text{rcv}, x)\). Moreover, a global assertion is a predicate \(\text{GA}_\Psi^F\) on behaviours (thought of as an assertion on free variables of \(T\) and the contents of the queue of messages received by \(\Psi\)) that holds iff execution of \(T\) has begun in that behaviour of \(\Psi\).
Compensation by design

Now ‘guideline condition’ can be defined.

**Definition 5.1 (guideline condition).** Let $\Psi$ be a service with scopes $S$, $U : sub^* (\Psi)$, and $Sub\ Var (U) := \bigcup {T.V \mid T \in sub^* (U)}$. Recall $GA^\Psi_T$ is a global assertion. Then

- $Gl_1 (U)$ holds iff each $T : child (U)$ satisfies the all-or-nothing principle, and either $skipon (GA^\Psi_T , (T.N \parallel T.C)) \text{ or } T$ satisfies the compensation principle;
- $Gl_2 (S, U)$ holds iff for every $x : rhs^*_NC \parallel \text{sub} \parallel \text{SubVar}(U)$, any write command of $x$ in $U$ lies in some scope $T \in child (U)$ for which $x \in T.y$, where $NC$ is the set of subprograms of $U$ except fault handlers $T.F$ of every scope $T : child (S)$.

The guideline condition $Gl(S, U)$ is said to hold iff both $Gl_1(U)$ and $Gl_2(S, U)$ hold. □

For $Gl_1$, if no assurance can be made that the child scope $T$ satisfies the compensation principle, the designer needs to ensure that the effect of $Q := T.N \parallel T.C$ keeps $T.y$ unchanged, under an assumption that $GA^\Psi_T$ holds prior to execution of $T$. When no such assurance is available, more precisely, $GA^\Psi_T = true$, then $skipon (true, Q, T.y)$ requires the designer to ensure that the effect of $Q$ leaves $T.y$ unchanged no matter what values are assumed by variables in $Q$ and the message queue prior to $Q$’s execution. Guideline condition $Gl_2$ requires restoration of any variable which may change the critical variables of $S$ but imposes nothing on variables $y$ for which given a critical variable $x$ of $S$, $x$ has data dependency on $y$ only in fault handlers of child scopes of $U$. This is because, the valuation of $y$ may change the value of $x$, but only when recovering $x$ when fault occur. As $Gl_2$ is about compensation of $S$, $y$ is not important.

### 5.2. Default fault and compensation handling

In $\text{BPEL}$, executing $\text{DI}$ calls automatic compensation of each committed (but not compensated) scope. $\text{DI}$ corresponds to the $\text{BPEL}$ activity $\text{<compensate>}$, when the scope’s fault or compensation handler is not explicitly defined in a $\text{BPEL}$ $\text{<scope>}$, $\text{<compensate>}$ is automatically executed when a fault occurs or compensation is called (like cancellation in $\text{cCSP}$). Default handling with automatic compensation is supported so that the process designer can focus on realizing business logic rather than handling fault and doing compensation. Therefore, it is important first to study guidelines to assist the process designer in using automatic default handling, i.e., use of $\text{DI}$ by the scope’s fault and/or compensation handler, rather than a user-defined handler calling scope-specific compensation (user-defined handling is studied in Sect. 5.3).

The first guideline is specified in Theorem 5.1 and is based on use of the default handler $\text{DI}$, which compensates committed child scope instances in reverse order. To guarantee the all-or-nothing principle for $S$, that principle must hold for child scopes of $S$, and the normal behaviour of any child scope followed by compensation must either have the same effect as $\text{skip}$ or satisfy the compensation principle ($Gl_1(S)$); all commands that may affect the value of the critical variables of the scope must lie in the child scopes, and any variable updated by that command must also be a critical variable of the child scope ($Gl_2(S, S)$). Then all critical variables of the scope are restored by the default handler.

If the default handler is used as the fault handler or compensation handler of the parent scope, the idea for the guideline matching the compensation principle is similar, but in addition requires that the variables that may affect the value of any critical variable of $S$ must not be modified by other commands out of the parent scope $U$ after the execution of $S$. As in $Gl_2(S, U)$, those variables that affect the value of critical variables only in fault handlers of child scopes of $U$ are of no concern. Such differences stem from the fact that the fault handler is always executed immediately after partial execution of the scope, at a point when a fault occurs; on the other hand, the compensation handler is triggered during execution of the fault or compensation handler of the parent scope. In particular, execution between commitment of the parent scope $U$ and the point at which $S.C$ is triggered by $U.C$ may also touch the critical variables of $S$. Therefore, the additional requirement is necessary.

**Theorem 5.1 (default-handler guideline).** Let $\Psi$ be a service, $S, U$ be scopes in $\Psi$, and $U$ be the parent scope of $S$, i.e., $S \in child(U)$, and $NC$ be the set of subprograms of $U$ except the fault handlers $T.F$ of scopes $T : child(U)$.

1. If $S.F = (P \parallel \text{DI} \parallel Q)$ (where $P, Q : \text{Prog}$ can be $\text{skip}$), then $S$ satisfies the all-or-nothing principle if
   
   (a) no $\text{comp}$ in $T$ for $T : child(S)$ or write commands to variables in $\text{rhs}(S.y)$ appear in $S.F$, and
   (b) $Gl(S, S)$ holds.
2. If \( U.F = (P_F \triangleright DH \triangleright Q_F) \) and \( U.C = (P_C \triangleright DH \triangleright Q_C) \) (where \( P_F, Q_F, P_C, Q_C : Prog \) can be skipp), then \( S \) satisfies the compensation principle iff

(a) no \( \text{compen}[T] \) for \( T : child(U) \) or write commands to variables in \( \text{rhs}_{\text{req}}^S \cdot (S, \gamma) \) appear in \( U.F \) or \( U.C \),
(b) \( Gl(S, U) \) holds, and
(c) there is no write command to any variable in \( \text{rhs}_{\text{req}}^S \cdot (S, \gamma) \) lying outside \( U \) unless such a write command is executed only before \( U \) starts.

Let \( S \) and \( U \) be scopes with \( S : child(U) \). The next result provides a sufficient condition for \( Gl_2(S, U) \), aimed to help designers in establishing those guidelines. It follows by arguments similar to those used in proving Theorem 5.1 (cf. Sect. 6.2).

**Proposition 5.1.** Let \( S \) be a scope in service \( \Psi \). Then \( \text{skipon}(GA^Q_S, (S, \gamma) \triangleright S.C, T, \gamma) \) holds when \( S.C = (P \triangleright DH \triangleright Q) \) (where \( P, Q : Prog \) can be skipp) and

- no \( \text{compen}[T] \) for \( T : child(S) \) or write commands to variables in \( \text{rhs}_{\text{req}}^S \cdot (S, \gamma) \) appear in \( S.C \), and
- \( Gl(S, S) \) holds [the same condition as Theorem 5.1(1b)].

The default-handler guideline is important because it supports the process designer in focusing on business logic rather than fault handling and compensation. Whilst fault and compensation handlers with user-defined programs calling compensations of specific child scopes do increase design flexibility, they also impose a design burden, especially when the design of a transaction evolves, and a user-defined handler has to be re-designed. A default handler is therefore useful. But to cover further design, guidelines for user-defined fault and compensation handlers are also proposed in Sect. 5.3.

**Example**

The purpose of this example is to illustrate the use of the default-handler guideline (Theorem 5.1) by showing that \( DoOp \) in \( \text{Batch-Atm} \) satisfies the compensation principle.

The parent scope of \( DoOp \), scope \( \text{WorkBack} \), uses \( \text{DH} \) as fault and compensation handler [Condition (a) of Theorem 5.1 holds], and \( DoOp \) has no sub-scopes but does have critical variables. Both variables \( \text{balance} \) and \( \text{req.bundle} \) have data dependency (in \( \text{WorkBack} \) other than \( \text{Check.F} \) and \( \text{DoOp.F} \)) only on \( \text{balance} \) and \( \text{req.bundle} \) (see Sect. 2.3). Since the only updates to \( \text{balance} \) and \( \text{req.bundle} \) out of \( \text{WorkBack} \) are immediately before \( \text{WorkBack} \), Condition (c) of Theorem 5.1 holds. And because updates to \( \text{balance} \) and \( \text{req.bundle} \) in \( \text{WorkBack} \) are all in \( \text{Check} \) and \( DoOp, Gl_2(DoOp, \text{WorkBack}) \) holds. To show that the compensation principles are satisfied, it suffices to ensure \( Gl_1 \) on \( DoOp \). That is,

\[
\]

(6)

Variable \( \text{balance} \) is assigned \(-1\) immediately before \( WorkBack \), and \( \text{Check} \) starts execution with \( \text{WorkBack} \). From Sect. 2.3, both \( \text{Check.F} \) and \( \text{Check.C} \) are the assignment \( \text{balance} := -1 \) (note that \( \text{Check.F} = \{ \text{balance} \} \)). Therefore the all-or-nothing principle holds for \( \text{Check} \).

Since an assertion of \( DoOp \)’s free variables cannot be deduced from \( Atm \), to ensure (6) it suffices to ensure

\[
\text{skipon}(\text{true}, (DoOp.N \triangleright DoOp.C), (\text{balance}, \text{req.bundle})).
\]

(7)

Recall that \( \text{put} : (\text{bag} \times X) \times X \rightarrow \text{bag} X \) is the operation that adds an element to a bag. \( \text{pick} : \text{bag} X \rightarrow X \) is the operation that returns an arbitrary element from a bag, and \( \text{rem} : (\text{bag} \times X) \times X \rightarrow \text{bag} X \) is the operation that removes an element from a bag. The design of \( DoOp \), given in Sect. 2.3, is:

\[
DoOp : \text{scope opammt : Z, NOp [fh FOp, ch COp]} \triangleright \{ \text{req.bundle, balance} \} \text{ epochs}
\]
where

\[
\begin{align*}
N_{Op} & \ ::= \ opamnt := \ pick(req.bundle) ; \\
P_{Op} & \ ::= \ \text{if} \ (balance + \ opamnt \geq 0) \ \text{else} \ \text{throw} \\
P_{Op} & \ ::= \ invBank(req.account, opamnt) ; \\
\text{balance, req.bundle := balance + opamnt, rem(req.bundle, opamnt)}
\end{align*}
\]

\[
\begin{align*}
F_{Op} & \ ::= \ P_{Op} \ \text{if} \ \text{size} \neq \ #req.bundle \ \text{else} \ \text{skip} \\
C_{OpH} & \ ::= \ invBank(req.account, -opamnt) ; \ P_{OpH} \\
P_{OpH} & \ ::= \ \text{balance, req.bundle := balance - opamnt, put(req.bundle, opamnt)}
\end{align*}
\]

Note that \text{size} does not need to be critical to \text{DoOp} although \text{balance} has data dependency on \text{size}, because the dependency appears only in the fault handler of \text{DoOp} (in the conditional guard); \text{size} would not affect the compensation of \text{DoOp} in \text{WorkBack} (see \text{Gl2}). Since the only operations changing critical variables of \text{DoOp} are \text{P}_{Op}, \text{P}_{OpH}, and (\text{P}_{Op}, \text{P}_{OpH} = \text{skip}). Property (7) follows.

5.3. User-defined handling

The default-handler guideline requires no scope-specific compensation is used. That is, fault handling and compensation must be handled with the built-in automatic approach. As this facilitates process design, it also restricts flexibility. Consider the case in which user-defined handling calls compensation of specific child scopes. The ‘reverse-compensation guideline’ ensure that if the compensations of specific scopes in the user-defined fault and compensation handlers satisfy a reverse ‘preceding relation’, the principles are guaranteed. And when scope-specific compensations are mixed with the default handler in user-defined fault handling and/or compensation, the guidelines of default-handler and reverse-compensation can be combined and extended to ensure the principles.

**Notation (\text{\prec}_{R})**

First, the preceding relations for scopes and compensation statements are defined as \text{\prec}_{R} for program \text{R}.

Let \text{S} be a scope and \text{R} be \text{S.N} or one of its subprograms. The preceding relation in \text{R} is defined:

\[
\text{\prec}_{R} : \text{child}(S) \leftrightarrow \text{child}(S)
\]

\[
\text{\prec}_{R} := \begin{cases} \text{\prec}_{P} & \text{if } R = \text{if } B \ \text{then } P \ \text{for some } B \not\in \text{Scope} \\
\text{\prec}_{P} \cup \{(T, T) \cup \{T\} \times \text{Child}(S, P)\} & \text{if } R = T \ \text{for some } T \not\in \text{child}(S) \\
\bigcup_{0 \leq i < n} \text{\prec}_{R,i} & \text{if } R = \text{if } (b_{i} \ \text{then } P_{i})_{i=1}^{n} \ \text{for some } P_{i} \in \text{child}(S) \\
\emptyset & \text{otherwise}
\end{cases}
\]

where \text{Child}(S, P) denotes the set of child scopes of \text{S} in \text{P}. Intuitively, if \text{T}_1 \text{\prec}_{R} \text{S.N.T}_2, then \text{T}_1 and \text{T}_2 are child scopes of \text{S} and do not appear in any loops or different conditional branches in \text{S.N}; furthermore \text{T}_2 is a sequent of \text{T}_1. Note that \text{T} \not\prec_{R} \text{T} when \text{T} is not in a do-loop in \text{R}. This is to cover the case when the subset of child scopes in question is a singleton set (see, e.g., \text{ScopePrec} on the next page).

If \text{S} is a scope, a further ‘preceding relation’ is defined syntactically on compensation calls in \text{S.F} as follows. For any \text{T}_1, \text{T}_2 : \text{child}(S), \text{compen}[T_1] \not\prec_{S.F} \text{compen}[T_2] \iff

- there is only one instance of each of \text{compen}[T_1] and \text{compen}[T_2] in \text{S.F} and neither appears in a scope, a conditional branch or a loop; and
- either \text{T}_1 = \text{T}_2, or when they are not the same \text{compen}[T_1] ≠ \text{compen}[T_2] or \text{compen}[T_1] \not\prec_{S.F} \text{compen}[T_2] for some program \text{P} is a subprogram of \text{S.F}, that is, \text{compen}[T_2] is a sequent of \text{compen}[T_1] in \text{S.F}.

A ‘preceding relation’ on compensation calls in \text{S.C} is defined similarly.

**Notation (\text{ScopePrec})**

To be used in specifying the guidelines, a predicate \text{ScopePrec} is defined as follows. Given a set \text{Set} of scopes and a program \text{P}, the predicate \text{ScopePrec} on \text{Set} and \text{P} holds iff for each scope \text{S} in \text{Set}, there is another scope in \text{Set} related by the preceding relation in \text{P}:

\[
\text{ScopePrec}(\text{Set}, P) ::= \forall T_1 : \text{Set} \ . \ \exists T_2 : \text{Set} \ . \ (T_1 \not\prec_{P} T_2 \ \lor \ T_2 \not\prec_{P} T_1).
\]
In another word, \( \text{ScopePrec}(\text{Set}, P) \) holds when none of the scopes in \( \text{Set} \) appears in a loop in \( P \) and no two scopes in \( \text{Set} \) appear in different conditional branches in \( P \).

**Guideline for using compensations of specific scopes \( \text{compen}[T] \)**

If \( \text{DH} \) is not used, but scope-specified compensations in user-defined fault and compensation handlers satisfy the reverse compensation order relations, the principles are again guaranteed. That is, the principles are ensured when there is a preceding relation between every pair of child scopes of \( S \) which may affect the value of variables in \( S, \gamma \), and compensation of these child scopes is in the reverse precedence order in the fault handler or compensation handler compared with the scopes’ precedence order in normal activity of the corresponding scope (\( S \) for all-or-nothing principles and the parent scope of \( S \) for compensation principle). The guideline is as follows.

**Theorem 5.2 (reverse-compensation guideline).** Let \( \Psi \) be a service, \( S, U \) be scopes in \( \Psi \), and \( U \) be the parent scope of \( S \), i.e., \( S \in \text{child}(U) \), and \( \text{NC} \) be the set of subprograms of \( U \) except the fault handlers \( T.F \) of scopes \( T : \text{child}(U) \) (as in Theorem 5.1).

1. Provided that \( \text{DH} \) is not in \( S,F \) (unless in scopes in \( S,F \)), then \( S \) satisfies the all-or-nothing principle if
   
   (a) there is no write command to variables in \( \text{rhs}_{S}^*([S,\gamma]) \) in \( S,F \).
   
   (b) \( \text{Gl}(S, S) \) holds (the same condition as Theorem 5.1(1b)).
   
   (c) \( \text{ScopePrec}(\text{child}(S), S.N) \) holds, and
   
   (d) for any \( T_1, T_2 : \text{child}(S) \) for which there is some write command to some variable in \( \text{rhs}_{S}^*([S,\gamma]) \), respectively, in \( T_1.N \) and \( T_2.N \),

   \[
   (T_1 \prec_{S,N} T_2) \Rightarrow (\text{compen}[T_2] \prec_{S,\gamma} \text{compen}[T_1]).
   \] 

2. Provided that \( \text{DH} \) is not in \( U.F \) and not in \( U.C \) (unless in scopes in \( U.F \) and \( U.C \)), then \( S \) satisfies the compensation principle if

   (a) no write command to variables in \( \text{rhs}_{\text{NC}}^*([S,\gamma]) \) in \( U,F \) or in \( U,C \),
   
   (b) \( \text{Gl}(S, U) \) holds (the same condition as Theorem 5.1(2b)).
   
   (c) there is no write command to any variable in \( \text{rhs}_{\text{NC}}^*([S,\gamma]) \) lying outside \( U \) unless such a write command is executed only before \( U \) start, (the same condition as Theorem 5.1(2c)).
   
   (d) \( \text{ScopePrec}(\text{child}(U), U.N) \) holds, and
   
   (e) for any \( T_1, T_2 : \text{child}(U) \) for which there is some write command to some variable in \( \text{rhs}_{\text{NC}}^*([S,\gamma]) \), respectively, in \( T_1.N \) and \( T_2.N \),

   \[
   \bigwedge \{(T_1 \prec_{U,N} T_2) \Rightarrow (\text{compen}[T_2] \prec_{P} \text{compen}[T_1]) \mid P \text{ is } U,F \text{ or } P \text{ is } U,C \}.
   \] 

**Guideline for mixed compensations of specific scopes and default handler**

The previous guidelines Theorems 5.1 and 5.2 requires the fault handler and compensation to be either the default handler or compensation of specific scopes. The scope-specific compensations can be mixed with the default handler, see the syntax in Sect. 2.1. The previous two guidelines are combined and generalized to the ‘mixed-handler guidelines’.

The mixed-handler guideline considers the fault handler and compensation with \( \text{DH} \) but before that there are also compensations of specific child scopes (while in the default-handler guideline as Theorem 5.1, no such compensations are called). Then these scope-specific compensations must also satisfy reverse compensation order. That is if there is a preceding relation between two of such child scopes in the normal activity program, their compensation must satisfy the reverse preceding relation (similar as reverse-compensation guideline, as Theorem 5.2). Moreover, for any child scope \( S \) whose compensation is not called before \( \text{DH} \), in the normal activity program, \( S \) is not a sequent of (i.e. never executed after) any child scope \( T \) whose compensation is called before \( \text{DH} \), that is, either \( S \) precedes \( T \) in a subprogram of \( S.N \) or \( S \) and \( T \) are in different conditional branches. (Then such \( S \) can be compensated by \( \text{DH} \).) And because after the execution of \( \text{DH} \), all committed child scope instances are compensated, any compensation after \( \text{DH} \) is skips.
Theorem 5.3 (mixed-handler guideline). Let \( \Psi \) be a service, \( S, U \) be scopes in \( \Psi \), and \( U \) be the parent scope of \( S \), i.e., \( S \subseteq child(U) \), and \( NC \) be the set of subprograms of \( U \) except the fault handlers \( T.F \) of scopes \( T : child(U) \) (as in Theorem 5.1).

1. Provided \( S.F = (P \uparrow \text{Dh} \uparrow Q) \), there \( P, Q : \text{Prog} \) and \( Q \) can be \( \text{skip} \) and \( P \) is without \( \text{Dh} \) but contains some \( \text{compen}(T) \) for \( T : child(S) \). Let \( \text{ChildC}(S) \subseteq \text{child}(S) \) be the set of scopes \( T \) where \( \text{compen}(T) \) is in \( P \). Then \( S \) satisfies the all-or-nothing principle if

   (a) no write command to variables in \( \text{rhs}_{S}^\Psi(S,Y) \) in \( S.F \) [the same condition as Theorem 5.2(1a)],

   (b) \( \text{Gl}(S, U) \) holds [the same condition as Theorem 5.1(1b)],

   (c) \( \text{ScopePrev}(\text{ChildC}(S), S.N) \) holds, and

   (d) for any \( T_{1}, T_{2} : \text{ChildC}(S) \) for which there is some write command to some variable in \( \text{rhs}_{S}^\Psi(S,Y) \), respectively, in \( T_{1}.N \) and \( T_{2}.N \),

       \( (T_{1} \prec_{S.N} T_{2}) \Rightarrow (\text{compen}[T_{2}] \prec_{P} \text{compen}[T_{1}]), \)

   (e) for any \( T_{1} : \text{ChildC}(S) \) and \( T_{2} : \text{child}(S) \setminus \text{ChildC}(S) \) for which there is some write command to some variable in \( \text{rhs}_{S}^\Psi(S,Y) \), respectively, \( T_{1} \) is in \( R_{1} : \text{Progs} \) and \( T_{2} \) is in \( R_{2} : \text{Progs} \), and \( R_{1} \downarrow R_{2} \) is not a subprogram of \( S.N \).

2. Provided \( U.F = (P_{F} \uparrow \text{Dh} \uparrow Q_{F}) \) and \( U.C = (P_{C} \uparrow \text{Dh} \uparrow Q_{C}) \), there \( P_{F}, Q_{F}, P_{C}, Q_{C} : \text{Prog} \) and \( Q_{F}, Q_{C} \) can be \( \text{skip} \) and \( P_{F}, P_{C} \) are without \( \text{Dh} \) but contain some \( \text{compen}(T) \) for \( T : child(S) \). Let \( \text{ChildC}(U) \subseteq \text{child}(U) \) be the set of scopes \( T \) where \( \text{compen}(T) \) is in \( P_{F} \); \( \text{ChildC}(U) \subseteq \text{child}(U) \) be the set of scopes \( T \) where \( \text{compen}(T) \) is in \( P_{C} \). Then \( S \) satisfies the compensation principle if

   (a) no write command to variables in \( \text{rhs}_{NC}(S.Y) \) in \( U.F \) or in \( U.C \) (the same condition as Theorem 5.2(2a)),

   (b) \( \text{Gl}(S, U) \) holds (the same condition as Theorem 5.1(2b)),

   (c) there is no write command to any variable in \( \text{rhs}_{NC}(S.Y) \) lying outside \( U \) unless such a write command is executed only before \( U \) start, (the same condition as Theorem 5.1(c)),

   (d) both \( \text{ScopePrev}(\text{ChildC}(U), U.N) \) and \( \text{ScopePrev}(\text{ChildC}(U), U.C) \) hold,

   (e) for any \( T_{1}, T_{2} : \text{ChildC}(U) \) [and resp., \( T_{1}, T_{2} : \text{ChildC}(U) \)] for which there is some write command to some variable in \( \text{rhs}_{NC}(S,Y) \), respectively, in \( T_{1}.N \) and \( T_{2}.N \).

       \( (T_{1} \prec_{U.N} T_{2}) \Rightarrow (\text{compen}[T_{2}] \prec_{P} \text{compen}[T_{1}]), \)

   [and respectively, \( (T_{1} \prec_{U.N} T_{2}) \Rightarrow (\text{compen}[T_{2}] \prec_{P} \text{compen}[T_{1}]) \).]

   (f) for any \( T_{1} : \text{ChildC}(U) \) and \( T_{2} : \text{child}(S) \setminus \text{ChildC}(U) \) [and respectively, \( T_{1} : \text{ChildC}(U) \) and \( T_{2} : \text{child}(S) \setminus \text{ChildC}(U) \)] for which there is some write command to some variable in \( \text{rhs}_{NC}(S,Y) \), respectively, \( T_{1} \) is in \( R_{1} : \text{Progs} \) and \( T_{2} \) is in \( R_{2} : \text{Progs} \), and \( R_{1} \downarrow R_{2} \) is not a subprogram of \( U.N \).

Example

This example shows how to apply the mixed-handler guideline for user-defined handlers.

The Batch-Atm process is extended, more specifically, change the scope of WorkBack as

\[
\text{WorkBack}^\prime : \text{scope changed} : \exists N_{WB} \cdot N_{WB}^\prime [\text{fh} F_{WB}^\prime, \text{ch} C_{WB}^\prime](\text{balance}) \text{ epocs}
\]

where its normal activity program is

\[
N_{WB}^\prime ::= \text{Check} : \text{scope} () \cdot N_{Check}^\prime [\text{fh} F_{Check}^\prime, \text{ch} C_{Check}^\prime](\text{balance}) \text{ epocs} \uparrow
\text{Ops if balance} \geq 0 \text{ else skip} \uparrow
\text{F more if balance} > 0 \text{ else skip}
\]

\[
P_{more} ::= \text{DoOp2} : \text{scope opmnt} : \exists N_{Op} \cdot N_{Op}^\prime [\text{fh} F_{Op}^\prime, \text{ch} C_{Op}^\prime](\text{opmnt}) \text{ epocs} \uparrow
\text{charged} ::= \text{false} \uparrow
\text{Fee} : \text{scope} () \cdot N_{Fee}^\prime [\text{fh} F_{Fee}^\prime, \text{ch} C_{Fee}^\prime](\text{balance}) \text{ epocs}
\]
Recall that scope $DoOp$ is called in a do-loop in $Ops$. $WorkBack$. $N$ is appending program $P_{more}$ when the balance is positive to the end of $WorkBack$. $P_{more}$ is a sequence of $DoOp$ which tries to deal a user request one more time; initializing the new scope variable $charged$ to $false$; then charges service fee by scope $Fee$. The new scope $DoOp$ is the same as $DoOp$ except having a different scope name (due to the well-formedness conditions).

For scope $Fee$, its normal activity charges 1 unit for service fee and its fault handler and compensation returns this fee:

$$
N_{Fee} := \text{charged, balance} := \text{true, balance} - 1
$$

$$
F_{Fee} := (\text{charged, balance} := \text{false, balance} + 1) \text{ if charged = true else skip}
$$

$$
C_{Fee} := (\text{charged, balance} := \text{false, balance} + 1)
$$

Recall that the fault handler and compensation of $WorkBack$ are both $DH$ (Fig. 5). As an example to the mixed-handler guideline, the fault handler and compensation of the extended $WorkBack'$ are designed as:

$$
F'_{WB} ::= \text{compen[Fee]} \text{; compen[DoOp2];}
$$

$$
\text{DH; !}(Atm, clntreq.snd, \ldots) /*send (email) notification*/
$$

$$
C'_{WB} ::= \text{compen[Fee]} \text{; compen[DoOp2];}
$$

$$
\text{DH}
$$

It is illustrated that scope $WorkBack'$ satisfies the all-or-nothing principle by applying Theorem 5.3. From $F'_{WB}$, Condition 1.a holds. $ChildC(WorkBack')$, $\vdash (\text{Fee, DoOp2})$, $ScopePrec(ChildC(WorkBack'))$, $N'_{WB}$ holds (Condition (1.e)); $DoOp2 \prec_{N'_{WB}} Fee$ and $\text{compen[Fee]} \prec_{f} \text{compen[DoOp2]}$, Condition (1.d) holds; besides, $Check \prec_{N'_{WB}} DoOp2$ and $Check \prec_{N'_{WB}} Fee$, and $DoOp2$ and $Fee$ are both in a sequent of the while loop $DoOp$ is in, Condition (1.e) holds. Note that in $Fee$, $balance$ depends on $charged$ only in $Fee.F$. Following similar arguments in the example in Sect. 5.1, $Gl(ChildC, WorkBack')$, $Gl(DoOp, WorkBack')$, $Gl(DoOp2, WorkBack')$ and $Gl(Fee, WorkBack')$ hold. Therefore, all 5 conditions for all-or-nothing principle in the mixed-handler guideline are satisfied. $WorkBack'$ satisfies the all-or-nothing principle on critical variable $balance$.

6. Behaviour and proof of guidelines

The purpose of this section is to establish, in Theorems 5.1, 5.2 and 5.3 that the design guidelines are indeed sufficient for the transactional principles to hold. To facilitate the proofs, some basic properties of the behaviour of a (single-service) process are established. As in the previous two sections, this section considers processes containing only one service. Extension to service choreography with ISTs is given in Sect. 7. Readers not interested in the proofs may continue by reading Sect. 7.

6.1. Behavioural properties

Runs are defined in Sect. 3.3. In this subsection, behavioural properties are given as lemmas to prepare for the proof of the design guidelines.

Let $\Psi$ be a service (more precisely, a single-service transactional process). Lemmas 6.1 to 6.3 explicate the semantics. Their proofs follow from the invariants of the Z schemas given in Sect. 3.2. Again, use of a state observable without referral to a state is shorthand for a state function. For example, given $sins : ScopeIns$, the shorthand $status(sins) = \text{active}$ means $\lambda s : \text{State}. s.status(sins) = \text{active}$.

Firstly, the identifier of each new scope instance is unique.

**Lemma 6.1 (unique id).** The semantics $TS(\Psi)$ satisfies: for any scope instances $sins, sins' : ScopeIns$ and any $id : \mathbb{N}$,

$$
\text{newins}(sins, id) \Rightarrow X \text{G} \neg(\text{newins}(sins', id))
$$

where, given $sins : ScopeIns$ for a scope instance and $id : \mathbb{N}$ representing the instance numeral of a scope identifier,

$$
\text{newins}(sins, id) ::= (sins \not\in \text{dom status}) \land X (\text{status}(sins) = \text{active} \land id = \text{second sins}).
$$
Lemma 6.2 (status property). The semantics $TS(\psi)$ satisfies: for any scope instance $S : sub^*(\psi)$ and any id : $\mathbb{N}$,

$$\text{undef } W (\text{stat}(active) W fhorch)$$

where

$$\text{undef } ::= S^{id} \notin \text{dom } \text{status}$$

$$\text{stat}(stt) ::= \text{status}(S^{id}) = stt$$

$$fhorch ::= (\text{stat}(\text{failed}) W (G \text{stat}(\text{flt-handled})))$$

$$\lor (\text{stat}(\text{committed}) W \text{runch})$$

$$\text{runch} ::= \text{stat}(\text{compn}) W (G \text{stat}(\text{compn}))$$

Proof

Only transitions originating from Schema $AScope$ create a new instance in $\text{status}$, and the newly created instance is labeled $\text{active}$ in the next state. Each newly created instance is assigned the instance numeral $\text{ident}$, and $\text{ident}$ is then incremented. Furthermore, $\text{ident}$ is modified only in $AScope$. $\square$

The second lemma describes the evolution of the status of a scope instance. Initially the instance has not started and so the status is undefined. Immediately after the scope instance has been initialized, its status is active. Then, if the instance terminates, it either completes successfully, i.e. its status changes to $\text{committed}$, or completes with fault, i.e. its status changes to failed. When fault handling of the failed instance finishes, the status changes to $\text{flt-handled}$. If the instance is committed, the status changes to $\text{committed}$, and if its compensation is then triggered the status changes to $\text{compn}$ and when it terminates, the status of the instance changes to $\text{compn}$. For states other than those above, the status remains the same as in its previous state.

**Lemma 6.2 (status property).** The semantics $TS(\psi)$ satisfies: for any scope instance $S : sub^*(\psi)$ and any id : $\mathbb{N}$,

$$\text{undef } W (\text{stat}(active) W fhorch)$$

Proof

Recall that $\phi W \psi$ is defined to be $(\phi U \psi) \lor G \phi$, which equals $\neg G \phi \Rightarrow (\phi U \psi)$, or $F \neg \phi \Rightarrow (\phi U \psi)$. To show that $\phi W \psi$ holds on a state $s_0$ of a run $\mathcal{I}$, let $m : \mathbb{N}$ be the smallest index $m \geq n$ for which $\neg \phi$ holds on state $s_m$. It is required to show that $\phi$ holds on states from $s_n$ to $s_{m-1}$ and that $\psi$ also holds on $s_m$. To do so, note that only transitions specified by Schema $AScope$ can create new instances, and only transitions specified by $\text{FaultEnv}$, $\text{Throw}$, or $\text{ExitNorm}$, $\text{ExitFH}$ or $\text{ExitCH}$ can change the status of instances.

Firstly, $\text{undef}$ holds on $s_0$ because $s_0.status = \emptyset$. Let $i : \mathbb{N}$ be the smallest positive index for which $\neg \text{undef}$ holds on $s_i$. It remains to show that $\text{stat}(active) W fhorch$ holds on $s_i$. Suppose $\neg \text{stat}(active)$ holds on $s_j$ where $j$ is the smallest index $j \geq i$. It must be shown that $\text{stat}(active)$ holds on states from $s_i$ to $s_{j-1}$, and that $fhorch$ holds on $s_j$.

Because only transitions specified by $AScope$ can create new instances, $\neg \text{undef}$ holds on $s_i$, so $\text{stat}(active)$ holds on $s_i$. So it remains to show $fhorch$ holds on $s_j$. Since $fhorch$ is the disjunction of two formulae, both constructed from the $W$ operator, the proof that $fhorch$ holds on $s_j$ can be done using similar arguments. It suffices to observe:

- status changes from $\text{active}$ to $\text{failed}$ only due to Schema $\text{Throw}$
- status changes from $\text{active}$ to $\text{committed}$ only due to Schema $\text{ExitNorm}$
- status changes from $\text{failed}$ to $\text{flt-handled}$ only due to Schema $\text{ExitFH}$
- status change from $\text{compn}$ to $\text{compn}$ only due to Schema $\text{ExitCH}$ (cf. Sect. 3.2). $\square$

Recall that the observable $\text{hist}$ of $TS(\psi)$ records the snapshot history of committed, but not yet compensated, scope instances in their order of completion. That property is guaranteed by the third lemma in which, for convenience, it is divided into three phases:

$\phi_1$: *No compensated instance.* For any instance, if in the run there is a state at which the instance is of status $\text{compn}$, then the snapshot of this instance is not in $\text{hist}$ of this or any subsequent state.
\( \phi_2 \): Committed valuation. For any instance, if \( s \) is the first committed state of the instance, then for any state \( t \) after \( s \) and before the compensated state of the instance, there is a snapshot of the instance in \( t.hist \), and the valuation \( s.val \) of the scope variable of the instance is the same as the valuation on the snapshot of the instance in \( t.hist \).

\( \phi_3 \): Snapshots order. For any two instances, if the first instance commits before the second, then for any state \( s \) following the committed state of the second instance until the compensated state of any of the two instances, the snapshot of the second instance lies before the snapshot of the first instance in \( s.hist \).

**Lemma 6.3** (History property). The transition system \( TS(\Psi) \) satisfies each of \( \phi_1, \phi_2 \) and \( \phi_3 \): for any scope instances \( sins, sins' : ScopeIns, S : Sub\{\Psi\} \), set of scope instances \( Set : F \ ScopeIns \), and valuation \( assign : S.V \rightarrow \text{Type} \), the following hold:

\[
\begin{align*}
\phi_1 &:= G(\text{compened}(sins) \Rightarrow G(\neg\text{inhist}(sins))) \\
\phi_2 &:= (\neg\text{comitted}(sins)) \lor (\text{commitval}(sins, assign) \Rightarrow (\text{histval}(sins, assign) \lor \text{compened}(sins))) \\
\phi_3 &:= G(\text{commitord}(sins, sins') \Rightarrow \text{histord}(sins, sins'))
\end{align*}
\]

where

\[
\begin{align*}
\text{compened}(sins) &:= \text{status}(sins) = \text{compened} \\
\text{inhist}(sins) &:= \text{hist} \upharpoonright s sins = \langle \rangle \\
\text{comitted}(sins) &:= \text{status}(sins) = \text{comitted} \\
\text{commitval}(sins, assign) &:= \text{commit}(sins) \land (\text{val} \land S.V = assign) \\
\text{histval}(sins, assign) &:= \text{inhist}(sins) \lor \text{valmatch}(sins, assign) \\
\text{valmatch}(sins, assign) &:= assign = ((\text{second}\ \text{order}(\text{hist} \upharpoonright sins)) \land S.V) \\
\text{commitord}(sins, sins') &:= (\text{commit}(sins) \land \neg\text{compened}(sins) \land \neg\text{comitted}(sins')) \\
\text{histord}(sins, sins') &:= (\text{histfilter}(\langle sins', sins' \rangle) \in \{\langle sins', sins' \rangle \land \neg\text{compened}(sins) \land \neg\text{comitted}(sins')\} \\
\text{histfilter}(Set) &:= \text{first}(\text{hist} \upharpoonright Set).
\end{align*}
\]

**Proof**

Consider a run \( (5) \) of \( TS(\Psi) \).

**Phase \( \phi_1 \)**

Schema \( \text{ExitCH} \), is the only schema specifying transitions that changes the status of a scope instance to \text{compened}, and removing the snapshot instance from \text{hist}. So there is a transition \( t_i \) after which \text{compened}(sins) holds in state \( s_{j+1} \) but \text{inhist}(sins) does not. By Lemma 6.2, \( s_{j+1} \models G \text{compened}(sins) \). For \( \phi_1 \), it suffices to show \text{sins} is not added to \text{hist} after state \( s_{j+1} \). But an instance snapshot is added to \text{hist} only by transitions specified by Schema \( \text{ExitNorm} \). And by Lemma 6.1 each new instance has a fresh identifier. Therefore, \( s_{j+1} \models G(\neg\text{inhist}(sins)) \), and \( \phi_1 \) holds.

**Phase \( \phi_2 \)**

Suppose \( s_j \) is a state with smallest index \( i > 0 \) on which \text{comitted}(sins) holds. So at any preceding state, the stronger predicate \text{commitval}(sins, assign) does not hold. It must be shown that if \text{commitval}(sins, assign) also holds on \( s_j \), then \text{inhist}(sins) and \text{valmatch}(sins, assign) hold on any state after \( s_j \) until \text{comitted}(sins) holds. The transition \( t_{i-1} \) results from Schema \( \text{ExitNorm} \), so \text{inhist}(S^0) holds on \( s_j \) (since snapshots are added to \text{hist} in the schema). But instance snapshots are removed only by transitions specified by Schema \( \text{ExitCH} \), which are also the only transitions that can change the status of an instance to \text{comitted} thus ensuring that \text{comitted}(sins) holds on the next state. Suppose \( s_j \) is a state having smallest index \( j > i \) on which \text{comitted}(sins) holds. Then \text{inhist}(sins) holds on any state from \( s_j \) to \( s_{j-1} \). The valuation of scope variables is the second element of the added snapshot by transitions specified by Schema \( \text{ExitNorm} \). Therefore, \text{valmatch}(sins, assign) holds on all states from \( s_j \) to \( s_{j-1} \), and \( \phi_2 \) holds.

**Phase \( \phi_3 \)**

Let \( s_j \) \( (i > 0) \) be a state on which \text{commitord}(sins, sins') holds. It has been proved: \text{histfilter}(\langle sins, sins' \rangle) = \langle sins \rangle \) holds on \( s_j \). Let \( s_j \) be a state with smallest index \( j > i \) on which \text{comitted}(sins) \lor \text{comitted}(sins') holds. It remains to prove that from \( s_j \) to \( s_{j-1} \), \text{histfilter}(\langle sins, sins' \rangle) \in \{\langle sins \rangle, \langle sins', sins \rangle\}. \) If there is a state from \( s_j \) to \( s_{j-1} \) for which \text{comitted}(sins') does not hold, then \text{histfilter}(\langle sins, sins' \rangle) = \langle sins \rangle. \) If
there is $s_k$ with smallest index $k \in [i, j)$ on which \textit{committed}($s_k$) holds, by ExitNorm, a snapshot of $s_k$ is added as head of the sequence.hist. So, for any state from $s_k$ to $s_{j-1}$, histfilter($s_j$, $s_k$) = $\langle s_k$, $s_j$), and $\phi_3$ holds.

6.2. Proof of design guidelines

The design guidelines proposed in Sect. 5 prescribe sufficient conditions for scopes to satisfy transactional principles. This section contains proofs of the theorems.

Proof of Theorem 5.1, the default-handler guideline

Proof

1. Consider all-or-nothing. The proof is based on the fact that due to the use of default handling, compensation of child-scope instances that are committed but not-yet-compensated are triggered in reverse order after the state at which a fault occurs. It is shown that each compensation restores the value of critical variables of the child scope instance, from the last committed child scope instance to the first. It follows that the critical variable of $S$ is restored by fault handling.

Consider a run (5) of $T(S)$ for all-scope instances $S$ committed but not-yet-compensated are triggered in reverse order after the state at which a fault occurs. It is shown that each compensation restores the value of critical variables of the child scope instance, from the last committed child scope instance to the first. It follows that the critical variable of $S$ is restored by fault handling.

By Lemma 6.2, the index of $\text{end}$ must be larger than the index of $\text{start}$. But Schema $\text{Throw}$ is the only one that changes the status of an instance from active to failed, so there is a state $\text{ft}$ between $\text{start}$ and $\text{end}$ which is the final state of a $\text{Throw}$ transition. Therefore, the program $\text{executes}$ $\phi$, and state $\text{ft}$ is prefixed by $S.F$. Because $\text{Di}$ is the only statement calling compensation in $S.F$ based on Schema $\text{DefaultHandler}$ (cf. Fig. 11), on the next state of $\text{ft}$, the program $\phi$ starts with $\text{seqle}$ $\text{cmpnins}$ $\text{hist}$ $\text{child}(S)$. If there are no child scopes of $S$ committed until state $\text{ft}$, $x$ is restored even if there is a failed child scope of $S$ that updates $x$, since $x$ must also be critical to the child scopes of $S$ (by $G\text{I}_2(S, S)$) and by $G\text{I}_1(S)$. Because of $G\text{I}(S, S)$ the value of $x$ does not change from $\text{start}$ to $\text{ft}$, and $\text{ft.hist}$ $\text{child}(S)$ = $\langle \text{hist} \rangle$. Then, by Schema $\text{DefaultHandler}$, it follows that $\text{ft}$ is the state immediately preceding $\text{end}$ and $\text{start}.val(x)$ = $\text{end}.val(x)$.

Otherwise, due to Lemma 6.2, for any child scope $T : \text{child}(S)$ and $\text{idv} : \mathbb{N}$, let $\text{start}(T^{idv})$, $\text{commit}(T^{idv})$, $\text{cmppning}(T^{idv})$, and $\text{cmpned}(T^{idv})$ denote respectively states in the run having the smallest indexes for which instance $T^{idv}$ is of status active, committed, cmppned, and cmmpned. Let $T^{idv}$ be the last executed instance of $\text{child}(S)$ (i.e. there are no scope instances executed after commitment of $T^{idv}$ and before $\text{ft}$). It is to be shown that, when restricted to the variables in $\text{vs} := \text{rhs}_{\text{NC}}(x) \cup \text{SubVar}(S)$, $\text{start}(T^{idv}).val$ is the same as $\text{cmpned}(T^{idv}).val$:

\begin{equation}
\text{start}(T^{idv}).val =_{\text{vs}} \text{cmpned}(T^{idv}).val.
\end{equation}
Similarly, if $M^{idm}$ is the first committed instance of a child scope of $S$,

\[ \text{start}(M^{idm}).\text{val} \stackrel{v}{=} \text{cmpned}(M^{idm}).\text{val}. \]

But no commands change the critical variable of $S$ between $\text{start}$ and $\text{start}(M^{idm})$, nor between $\text{cmpned}(M^{idm})$ and $\text{end}$. So $\text{start}.\text{val}(x) = \text{end}.\text{val}(x)$ and the all-or-nothing principle holds.

2. Consider compensation. For $U.F = \text{dh}$ and $S \in \text{child}(U)$, the proof follows that showing that the values of critical variables are the same on the start state and compensated state of child scope instances. Note, because of Condition (2.e), that the values of variables in $\text{rhs}^\text{NC}(S.\gamma)$ are the same on states between state $\text{commit}(T^{idt})$ and state $\text{cmpned}(T^{idt})$ for any $T : \text{child}(U) \setminus \{ S \}$ and $idt : \mathbb{N}$.

\[ \square \]

**Proof of Theorem 5.2, the reverse-compensation guideline**

**Proof**

The proof resembles that of Theorem 5.1. Since $\text{ScopePrec}(\text{child}(S), S.N)$ [respectively $\text{ScopePrec}(\text{child}(U), U.N)]$ holds, there is only one instance for any child scope $T \in \text{child}(S)$ [respectively $T \in \text{child}(U)$].

For the all-or-nothing guideline, let state $\text{start}(T^{idt})$ and $\text{cmpned}(T^{idt})$ be defined as in the proof of Theorem 5.1. Then similarly, Eq. (10) holds. If there is a scope $T$ that is not executed or fails during its execution, then $TS(\Psi)$ satisfies $G(hist \mid \{ T \} = \{ \})$. By Schema $\text{CompensateScopeIns}$, execution of $\text{compen}(T^{idt})$ is equivalent to that of $\text{skip}$. By Eq. (8) and following similar steps to those in the proof of Theorem 5.1, the guideline follows.

Proof of the compensation guideline is similar.

\[ \square \]

**Proof of Theorem 5.3, the mixed-handler guideline**

**Proof**

Because of Condition 1.e of Theorem 5.3, for any child scope $T$ whose compensation is called before the first DH, the satisfaction of Eq. (10) can be proved following the proof of Theorem 5.2 (the reverse-compensation guideline). For instances of other child scopes, the satisfaction of Eq. (10) can be proved following the proof of Theorem 5.1 (the default-handler guideline). Therefore, mixed-handler guidelines are sufficient conditions for transactional principles.

\[ \square \]

7. **Choreography**

In the preceding sections, scopes have been transactional blocks. In BPEL, transactions are local to each service but there is no distributed transactional coordination among services. Business processes are typically carried out by different participants. The processes are modeled as choreographies of services interacting concurrently. Moreover, actions in business processes are often coordinated to execute in a transactional manner. For example, when payment fails, not only the online shopping system but also the bank system, inventory management and delivery service must take actions to deal with the failure. It is necessary to allow business transactions to cross trust boundaries.

It has been shown (for example [SM05, KML09]) that languages with only localized transactional scopes are insufficient for specifying the transactional properties of such inter-service transactions (ISTs). As stated in the BPEL standard:

*There is no distributed coordination necessary regarding an agreed-upon outcome among multiple-participant services. The achievement of distributed agreement is an orthogonal problem outside the scope of this specification* [OAS07], Sect. 12.3.
There appears to be no common consent on the design and properties of business transactions involving distributed participants. Rather than treating the whole process as one transaction, the proposed inter-service transactions (ISTs) are defined upon the localized scopes and `cross-cut' the services [KML09]. The failure and invocation of compensation of ISTs are also defined on scopes. With this attempt of defining the ISTs, no additional assumption, for example voting [OAS09a] or a (centralized) referee [OAS09b], is required.

The language BaTi from Sect. 2.1, and its semantics in Sect. 6, are extended to include service choreography and inter-service transactions. The extension is mostly at the semantic level and hence transparent to the designer who needs be aware predominantly of parallel composition.

Section 7.1 defines the notion of IST, supported in the following section by the Bank-transfer example. Section 7.3 shows how services composed in parallel coordinate to form an IST. The semantics given in Sect. 7.4 extends that in Sect. 6.

### 7.1. Service choreography and ISTs

Services synchronize by message passing. The type of processes constructed by parallel composition of services has been introduced by Eq. (1) in Sect. 2.1. Thus given a process \( P : \text{BaTi} \) there is a nonempty finite set \( E \) of service scopes with \( P = \| E \). Define \( P, \psi := E \) and extend the star notation for sub to sub*\( P \) :\( ::= \bigcup \{ \text{sub}\}\psi |\psi \in P, \psi\} \).

Transactions may span different services. Assume a countably infinite set of IST names. Given a process \( P : \text{BaTi} \) and \( S \notin H \) satisfying \( |(S) \cup H| \in \text{sub}^*(P) \), an IST \( T \) in \( P \) is declared by

\[
T : \text{transact}[S \text{ with } H]
\]

There: \( T \) is the name of the IST; \( T.m \) is called the main scope; \( T.s \) is the set of supporting scopes;

\[ T.p \ := \{ T.m \} \cup T.s \quad (= \{ S \} \cup H) \]

is the set of participating scopes; and \( \gamma \in \text{Var} \) is the finite set of critical variables. The set of all participating scopes and their sub-scopes is written

\[ \text{sub}^{*}(T) := \bigcup\{ \text{sub}^{*}(S) | S \in T.p \}. \]

Given a choreographed process \( P \), the set of all ISTs defined in \( P \) is denoted \( \text{IST}(P) \).

Let \( P \) be a choreographed process. Then IST \( T \) is said to be well-formed iff

1. any participating scope does not appear in a fault or compensation handler:

\[ \forall S : T.p \ . \exists \psi : P, \psi \ . S \in \text{child}^{*}(\psi) \]

2. participating scopes lie in different services

\[ \forall S_1, S_2 : T.p \ . S_1 \neq S_2 \Rightarrow \exists \psi_1 \neq \psi_2 : P, \psi_1 \ . S_1 \in \text{child}^{*}(\psi_1) \land S_2 \in \text{child}^{*}(\psi_2) \]

3. every critical variable of \( T \) is a critical variable of some participant scope of \( T \)

\[ T.\gamma \subseteq \bigcup\{S.\gamma | S \in T.p \} \]

4. participating scopes of different ISTs in \( P \) have no common sub-scopes

\[ \forall \mathcal{U} : \text{IST}(P) : T \neq \mathcal{U} \Rightarrow \text{sub}^{*}(| T |) \cap \text{sub}^{*}(| \mathcal{U} |) = \emptyset \]

5. critical variables of any scopes not in \( T \) have no data dependency on variables written in participating scopes of \( T \)

\[ \forall S : \text{sub}^{*}(| P |) \ \text{wrt } T \cap \text{rhs}_{P}^{*}(| S.\gamma |) = \emptyset \]

where \( \text{wrt}(T) \) denotes the set of variables being assigned-to by write commands in scopes \( \text{sub}^{*}(| T |) \), and \( \text{rhs}_{P} \) is extended from \( \text{rhs}_{P} \) (cf. Sect. 5.1): \( \text{rhs}_{P} := \bigcup\{ \text{rhs}_{\psi} | \psi \in P, \psi \} \).
The independence of scopes not in an IST (well-formedness Condition 5) ensures that the design of single-services and guidelines for localized scopes are independent of the design of the ISTs. That is important because choreography is typically defined after the services themselves, particularly when reuse of (third-party) services is common. In the theory to follow, it is assumed that all ISTs in any choreographed process are well-formed and with unique names.

Some work on transactions spanning several services assume each service is part of the such transaction as a whole (for example, in KML09 [KML09] and implicitly in WS-BA [OAS09b]). This special case, where every service scope is a participating scope of the IST, restricts the flexibility: every service in the choreography becomes private to one transaction. However, it is necessary to support the case that some part of a service (or a service-based business process) do not serve as participant of any IST, or different parts of a service are for different business transactions. The proposed IST supports such cross-cutting design. The notion of main (and support) scopes is introduced to (1) support a more general notion of mixed-outcome (see WS-BA [OAS09b]) that every scope in the transaction can be committed or failed or compensated; (2) relate the fault and compensation of the IST to localized scopes without additional assumptions (e.g. voting or referees).

7.2. Example for choreographies and ISTs

The choreographed process Bank-transfer has been introduced in Sect. 1.2. The Transfer process and the Pay IST are given in Fig. 13. The process is a parallel composition of services Payee, Payer and Bank, whose structures are given in Fig. 13. The IST Pay starts with a request for payment, i.e. scope ReqPay, and terminates when the transfer operation is finished. The scopes involved are therefore ReqPay, Payer and BankTrans, there ReqPay is a scope in Payee, BankTrans is a scope in Bank with the balance being critical. The critical data of this transaction is the balance at Bank. The BąT code for the process is then given.

The messages used in the interaction between the three services are of the following types, where Env stands for the singleton set of the environment of the process Transfer:

\[
\begin{align*}
\text{BTReq} & ::= \text{Env} \times \{\text{Payee}\} \times \text{PayReq} \\
\text{PtoB} & ::= \{\text{Payer}, \text{Payee}\} \times \{\text{Bank}\} \times \text{OpReq} \\
\text{BtoP} & ::= \{\text{Bank}\} \times \{\text{Payee}, \text{Payer}\} \times \text{Z} \\
\text{RtoE} & ::= \{\text{Payer}\} \times \{\text{Payee}\} \times \text{Charstr}
\end{align*}
\]

there

\[
\begin{align*}
\text{PayReq} & ::= \text{record}[\text{acc} : \text{Charstr}, \text{amnt} : \text{Z}] \\
\text{OpReq} & ::= \text{record}[\text{from} : \text{Charstr}, \text{to} : \text{Charstr}, \text{amnt} : \text{Z}]
\end{align*}
\]
Compensation by design

**Service Payer:** The service Payee receives a request from the client and then invokes the service Payer to transfer the money from Bank. It is defined

\[
\text{Payee} : \text{scope } balee : Z, \text{payreq} : \text{PayReq}, btreq : \text{BTReq}, \text{brespee} : \text{BtoP}.
\]

where

\[
N_{pee} [\text{fh } F_{pee}, \text{ch skip}] \circ
\]

Payee uses scope ReqPay to request payment from the Payer, in which

\[
N_{rp} ::= \text{Pbal} \downarrow
balee, \text{payreq} := \text{brespee}.msg, btreq.msg \downarrow
!(\text{Payee, Payer, payreq}) \downarrow
?\text{payresp} \downarrow
\text{Pbal} \downarrow
\text{throw if brespee}.msg \neq \text{balee} + \text{payreq}.amnt \text{ else balee} := \text{brespee}.msg
\]

P_{bal} is the program invoking Bank to check the balance of the to account.

\[
P_{bal} ::= !(\text{Payee, Bank, 0}) \downarrow
?\text{brespee}
\]

**Service Payer:** The service Payer starts by receiving a payment request from Payee and then calls Bank to transfer the requested amount from the payer’s account, denoted by constant per, to the payee’s account. In case of a fault when processing payment, a fault message ‘fail’ is returned to Payee, which results in failure in the scope ReqPay of Payee.

\[
\text{Payer} : \text{scope } \text{toreq} : \text{PayReq}, \text{bresp} : \text{BtoE}. N_{per} [\text{fh } F_{per}, \text{ch skip}] \circ \text{scope}
\]

where

\[
N_{per} ::= \text{?toreq} \downarrow
!(\text{Payer, Bank, [from = per, to = (toreq}.msg).acc, amnt = (toreq}.msg).amnt)) \downarrow
?\text{bresper} \downarrow
!(\text{Payer, toreq.snd, ‘ack’})
\]

\[
F_{per} ::= !(\text{Payer, toreq.snd, fault(‘fail’)})
\]

**Service Bank:** The bank system Bank handles each client (payer or payee) request for money transfer in each iteration of scope BankTrans. When the operation finishes, it returns the final balance of the from account. In particular, when the from account is the same as the to account, it works as a balance checker. The critical variable of BankTrans is bal, a function of the balance of each account.

\[
\text{Bank} : \text{scope } \text{bal} : \text{Charstr} \mapsto Z, \text{mark} : \mathbb{B}. N_{bank} [\text{fh } F_{bank}, \text{ch skip}] \circ \text{scope}
\]

where

\[
N_{bank} ::= \text{do true then}
\text{mark} ::= \text{false} \downarrow
\text{BankTrans} : \text{scope } clntreq : \text{PtoB}, \text{req} : \text{OpReq}.
N_{BT}[\text{fh } F_{BT}, \text{ch } C_{BT}][\text{bal}] \circ \text{epocs}
\text{od}
\]
Scope BankTrans is used in Bank to carry out each transaction started by a client request. The following abbreviation is used

\[ \text{updBal}((bal, \text{from}, \text{to}, \text{amnt}) := bal + \{\text{from} \mapsto (bal(from) - \text{amnt}), \text{to} \mapsto (bal(to) + \text{amnt})\}. \]

Then,

\[ N_{BT} := \text{?clntreq}. \]
\[ \text{req} := \text{clntreq}.\text{msg}. \]
\[ P_{Tr} \text{ if } \{\text{req}.\text{from}, \text{req}.\text{to}\} \subseteq \text{dom} bal \text{ else throw} \]
\[ F_{BT} := \text{skip} \text{ if } \text{mark} \text{ else } P_{Tfc} \]
\[ C_{BT} := P_{Tfc} \]

\[ P_{Tr} := (\text{mark}, bal) := (\text{true}, \text{updBal}(bal, \text{req}.\text{from}, \text{req}.\text{to}, \text{req}.\text{amnt})) \]
\[ ! (\text{Bank}, \text{clntreq}.\text{snd}, bal(\text{req}.\text{from})) \]
\[ P_{Tfc} := (\text{mark}, bal) := (\text{false}, \text{updBal}(bal, \text{req}.\text{to}, \text{req}.\text{from}, \text{req}.\text{amnt})) \]
\[ ! (\text{Bank}, \text{clntreq}.\text{snd}, \text{fault}, bal(\text{req}.\text{from})) \]

In Transfer, services Payee and Payee are both client services of the Bank. Payee pays a certain amount of money from its bank account (denoted by constant \text{per}) to some other bank account, as requested by Payee. Payee receives a payment request from the environment (the client to the process).

### 7.3. Coordination for ISTs

Since ISTs cannot be captured using local scopes alone, they must be achieved by coordination of services. The form of coordination chosen here has been motivated by the community of web services including standards WS-C [OAS09c] and WS-BA [OAS09b].

A (logically central) coordinator is introduced to the system to coordinate the scopes, distributed in different services, to ensure certain transactional behaviour. Each instance of the transaction—either a localized scope or an IST—is registered at the coordinator, which therefore knows and can track the status of each transaction. When a scope \(S\) starts it registers at the coordinator:

- If \(S = \text{main} \text{ for some } T : \text{IST}\), a new instance of \(T\) also starts, the status of the instance of \(T\) and \(S\) are both set to active in the coordinator registry, and the main scope instance is marked as a participant of the IST instance of \(T\).
- Otherwise, the instance is unregistered in the coordinator.

When a scope receives a message, it first checks to see if the scope is a supporting scope of some active IST instance. If so, it is registered in the coordinator as a participant of the IST instance.

If the main scope is committed, the IST instance is marked as committed after all participating scope instances are either committed, \text{flt-hndled} or \text{compened}. The failure of an IST instance is due to the failure of its main scope instance, and the compensation of an IST instance is caused by compensation of the main scope instance.

If the main scope instance fails (a fault in execution), the status of the corresponding IST instance is changed to \text{failed}, all of the active instances of supporting scopes are interrupted (as if encountering a fault from the environment) immediately or after termination of already-running fault or compensation handlers of descendant scopes. Then all of the committed instances of supporting scopes are compensated in reverse order of commitment. When the fault handler of the main scope instance finishes, and all supporting scope instances are of status either \text{compened} or \text{flt-hndled}, the transaction instance is marked as \text{flt-hndled}.

If the main scope instance is requested to be compensated (by its parent scope), the status of the corresponding IST instance is changed to \text{cmpning}, all of the committed instances of supporting scopes are compensated in reverse order of commitment and, when the compensation of the main scope instance finishes and all supporting scope instances are of status either \text{compened} or \text{flt-hndled}, the transaction instance is marked as \text{compened}.

Recall that \(\text{RunningStat} ::= \{\text{active, failed, cmpning}\}\). When a main scope instance is to change its status from either undefined or not in \(\text{RunningStat}\) to a status in \(\text{RunningStat}\), this step is required to wait until all IST instances with this scope as main scope exit \(\text{RunningStat}\).
7.4. Extending the semantics

The semantics of Sect. 6 is now extended to choreographed processes. A complete semantics is given in Appendix A.

Firstly, a new semantic artifact `interrupt[T]` is defined for interrupting the running supporting scope when the IST fails. Interruption of any scope of an IST `T` starts the fault handling of the enclosing scope `S` if `S` is a participating scope of `T` or a child scope of a participating scope. And `Basic^I` is modified to include `interrupt[T]`.

Then, some of the abbreviations introduced in Sect. 6 are modified, and more introduced. Let `P : Bat` be a (choreographed transactional) process. `Val` is extended to all scope variables in `P`; `SnapHist` (and so is state observable `hist`) is extended to every services in `P`, so is `EncScope` (and state observable `enc`). Recall `P : Ψ` is the set all service scopes in `P`. New abbreviations are for ISTs and coordination.

\[
\begin{align*}
Val & ::= \cup \{ S. S \implies \text{Type} \mid S \in \text{sub}^*(P) \} \\
SnapHist & ::= P : \Psi \implies \text{seq} \text{Snapshot} \\
EncScope & ::= P : \Psi \implies \text{Scope} \cup \{ \text{none} \} \\
ISTIns & ::= \text{IST} \times \mathbb{N} \\
WaitFun & ::= \text{Scope} \implies \text{Prog} \\
Registry & ::= \text{ISTIns} \implies (\exists \text{ScopeIns}) \times \text{Status} .
\end{align*}
\]

The transition-system semantics of `P` is denoted `TS(P)`. The state space of `TS(P)` extends that of Sect. 6, by extension and introduction of

- `hist : SnapHist`, as the extension to `SnapHist`, the snapshot sequence of committed but not compensated scope instances of each service in the execution of `P`;
- `enc : EncScope`, as the extension of `EncScope`, the enclosing scope (to handle faults) of each service in the execution of `P`;
- `wait : WaitFun`, the program waiting to execute immediately, after the fault or compensation handler of the corresponding scope; and
- `reg : Registry`, the coordination register which records, for each IST instance, its participating scope instances and status. The instance numeral used to identify the IST instance is the same as its main-scope instance, and so is less than `ident`.

The initial state of `TS(P)` is as follows: `φ` equals `P^I`; `reg` is the empty set; for each service of `P`, `hist` and `enc` are the empty sequence and `none` respectively; message queues are arbitrary as long as the message is from the environment to services in `P`; and for any scope `S` in `P`, `wait(S)` equals `eop`. Other state observables `val` and `idcnt` are initialized as the single-service process. The choreographed process `P`, with each `S : sub^*(P)` replaced by `Si`, is denoted `P^I`.

Some of the transition schemas are also extended from the single-service case to support choreographed processes and IST following the idea of coordination introduced in Sect. 7.3, and one more transition Schema `Interrupt` is added.

- In Schema `Receive`, if the receive lies in a supporting scope `S` of an IST `T` such that both of the instances of `S` and `T` are of status `active`, then the scope instance of `S` is registered in `reg` as a participating scope instance of the instance of `T`.
- The extension to `Throw` is more complicated. Firstly, if the failed scope instance `sink` is a main scope instance of an IST instance `istins`, interruption is inserted at the head of each thread of supporting scopes of the next `P`. Secondly, the IST instance `istins` is marked to have status `failed`. Thirdly, for any supporting scope `Si` of `istins`, `wait(S)` is updated (if `wait(S)` is originally `eop`) or appended at the end (otherwise) by the compensation of all supporting scope instances of `istins`.
- The interruption (denoted `interrupt[T]`, and specified by Schema `Interrupt` in Appendix A) acts like occurrence of a fault in every descendant of every supporting scope until there is no such instance running. That is achieved by inserting `throw` at the head and `interrupt[T]` immediately after the end of the current running scope component program if the enclosing scope is a supporting scope of the specified IST itself or its descendant scopes; otherwise it is treated like `skip`. And if the interruption is executed in the fault or compensation handler, it awaits termination of the fault or compensating handler by updating `wait` of the corresponding scope.
Theorems 5.1, 5.2 and 5.3 and Proposition 5.1.

Although the lemmas in Sect. 6 are stated in terms of transition systems of single-service processes, they remain true for transition systems of choreographed services and IST (with extension to IST). Details of the extended semantics are given in Appendix A.

Notation (rest)

For any \( T^{id} \) \( \in \) ISTIns, \( stt \in \{ \text{compended, flt-hndled} \} \) and \( vs \in \mathcal{V} \ar\)

\[
\text{restist}(T^{id}, stt, vs) := \bigwedge \left\{ (T^{id} \not\in \text{dom reg}) \mid \text{startist}(T^{id}, stt, x, v) \mid x \in vs \text{ and } v \in \text{typ}(x) \right\}
\]

where

\[
\text{startist}(T^{id}, stt, x, v) := (\text{second oreg}(T^{id}) = \text{active} \land (\text{val}(x) = v \Rightarrow \text{untilist}(T^{id}, stt, x, v)))
\]

\[
\text{untilist}(T^{id}, stt, x, v) := ((\text{second oreg}(T^{id}) \neq stt) \mid \text{W}((\text{second oreg}(T^{id}) = stt \land \text{val}(x) = v)).
\]

Intuitively, \( \text{restist}(T^{id}, stt, vs) \) asserts (similarly to \( \text{restored} \) in Sect. 4 for scopes): either there is no state with status \( stt \), or the value of variables in \( vs \) at the first state with status \( stt \) is the same as (restored to) the value at the first state with status \( \text{active} \).

In case \( vs = T\gamma \), \( \text{restist}(T^{id}, stt, vs) \) is abbreviated \( \text{restist}(T^{id}, stt) \).

Now the transactional principles for IST are defined in terms of \( \text{restist} \).

Definition 8.1 (Transactional principles for ISTs). Let \( \mathcal{P} \) be a (choreographed transactional) process. An IST \( T : \text{IST}(\mathcal{P}) \) is said to satisfy the transactional principles iff for any \( id : \mathbb{N} \)

\[
\text{restist}(T^{id}, \text{flt-hndled}) \quad \text{(all-or-nothing)}
\]

\[
\land
\]

\[
\text{restist}(T^{id}, \text{compended}) \quad \text{(compensation)}.
\]

A process \( \mathcal{P} \) is said to satisfy the transactional principles iff every service and IST in \( \mathcal{P} \) satisfies the transactional principles.

Like the principles for scopes, the all-or-nothing principle for IST states that if a fault occurs in the main scope of \( T \) (but not in any of its sub-scopes), no changes to critical variables are committed. The compensation principle for IST states that if after commitment of \( T \) (the main scope instance is committed and all supporting scope instances are not running), its compensation is triggered (due to failure in a parent scope of the main scope), the critical variables are restored by compensating the participating scope instances.
Compensation by design

8.2. The guidelines for IST

As in Sect. 5, guidelines are introduced to ensure that a (choreographed transactional) process satisfies the principles for IST. Again, they provide sufficient conditions for processes with IST to satisfy the transactional principles. By applying the guidelines for IST, the problem of checking the principles for an IST is reduced to checking those for the participating scopes (to which previous results apply).

The design guidelines for principles for IST follow the intuition that when all supporting scopes of an IST are ‘started’ by messages from the main scope (see $MsgDepend$), the effect on the critical variables by the supporting scopes is taken after the start of the main scope. (Thus the relation between the main scope and the supporting scopes would be analogous to the relation between the parent scope and the child scopes). Then if the critical variables of the IST depend only on commands in the participating scopes of the IST (see $Gl_{ist_2}$), ensuring the transactional principles on IST can be localized: to ensuring the transactional principles for participating scopes locally and $skipon$ of normal activity of the participating scopes followed by the compensations ($Gl_{ist_1}$).

So first are defined the notations for message dependency between supporting scopes and the main scope ($MsgDepend$), and then the guideline condition for IST.

**Notation ($MsgDepend$)**

Let $P : BA$ be a choreographed process. Given a scope $S : sub^*(P)$, $MsgDepend(S)$ denotes the set of scopes $T$ lying in some service other than $S$ for which the head of $T.N$ receives messages from either $S$ or other scopes in $MsgDepend(S)$. That is, $MsgDepend(S)$ is the set of scopes satisfying:

- there are services $\Psi_T \neq \Psi_S$ such that $T \in sub^*(P)$ and $S \in sub^*(\Psi_S)$;
- $phead(T.N)$ is of the form $?m$ where $m$ and $L = \Psi_S$ and $m$ is sent by some $ps$ in $\Psi_S$ satisfying
  \[
  inprog(ps) = (T.main).N \lor \exists U : MsgDepend(S) \land inprog(ps) = U.N.
  \]

The scopes in $MsgDepend(S)$ are said to have message dependency on $S$.

**Guideline condition of IST: $Gl_{ist}$**

Given $T : IST$ and $P : BA$, the guideline condition $Gl_{ist}(T, P)$ is said to hold iff both $Gl_{ist_1}(T, P)$ and $Gl_{ist_2}(T, P)$ hold, where

- $Gl_{ist_1}(T, P)$ holds iff every $S : T.parts$ satisfies the all-or-nothing principle (for scopes), and either the compensation principle (for scopes) or $skipon(GA^S, (S.N \cap S.C), S.N)$ holds;
- $Gl_{ist_2}(T, P)$ holds iff for every $x : rhs_{T.C}(T.N) \setminus SubVartst(T)$, any write command of $x$ in $P$ lies in some scope $S : T.parts$ and $x \in S.y$, where $NC$ is the set of subprograms of $P$ except those fault handlers $S.F$ of every scope $S : T.parts$, and $SubVartst(T) := \bigcup \{W.V | W \in sub^*(T)\}$.

The guidelines for IST are similar to the guidelines for scopes but with restrictions due to choreography and the fact that local transactions (i.e. scopes) affect the result of IST. The idea is that the principles hold if:

- all participating scopes of the IST satisfy the transactional principles (for scopes, cf. Sect. 4), and any variable that may change the value of the critical variables of the IST is also critical to the participating scopes ($Gl_{ist}$, Condition 1 of Theorem 8.1); and
- any supporting scope has message dependency on the main scopes (Condition 2 of Theorem 8.1).

**Theorem 8.1 (coordination guideline).** Let $P$ be a choreographed process and $T$ an IST in $P$. Then $T$ satisfies the transactional principles (both the all-or-nothing principle and compensation principle) if

1. $Gl_{ist}(T, P)$, and
2. $T.supps \subseteq MsgDepend(T.main)$. 


Proof

Consider a run \( r (S) \) of \( TS(P) \). As in the proof of Theorem 5.1, consider first the all-or-nothing principle: \( r \) satisfies \( \text{restist}(T^{id}, \text{flt-hndled}) \).

Let \( id : \mathbb{N} \), \( x : T, y \) and \( v : \text{typ}(x) \) be arbitrary. Let \( \text{start} \) and \( \text{end} \) be states with smallest index, respectively, on which \( \text{startist}(T^{id}, x, v) \) and \( \text{status}(T^{id}) = \text{flt-hndled} \) hold. Then \( \text{start.val}(x) = v \), and it must be proved that \( \text{end.val}(x) = \alpha \). If \( x \in (T \text{main}), y \), due to \( \text{Ghist}(T, P) \), \( \text{start.val}(x) = \text{end.val}(x) = v \). Otherwise, there exists \( S \in T.\text{supps} \) with \( x \in S.y \). By \( \text{Ghist}(T, P) \), \( x \) can be changed only by instances of \( S \). But for any \( id_{S} : ID \), \( S \in \text{MsgDepend}(T.\text{main}), \) so if there is an execution \( s \rightarrow s' \) between \( \text{start} \) and \( \text{end} \) satisfying Schema \( \text{Receive} \), then \( s.\text{status}(S^{ids}) = \text{active} \), and the thread of \( S \) is prefixed by \( S.N \) (with head a receive message from the main scope).

Furthermore second \( (s.\text{reg}(T^{id})) = \text{active} \), because when the main scope instance fails or is requested to be compensated, all messages sent by the main scope instance but not yet received are flushed from the message queue (cf. Schema \( \text{Trans} \) and \( \text{CompensateScopeIns} \) in Appendix A).

By Schema \( \text{Receive} \) (cf. Appendix A), \( S^{ids} \) is added as a participating scope instance of \( T^{id} \), otherwise, although \( S^{ids} \) is not a participating scope instance, \( x \) is not changed in \( S.N \).

The compensation of supporting scope instances caused by interruption and compensation of main scope runs, as for the default handler, in reverse order of commitment. It follows that \( \text{start.val}(x) = \text{end.val}(x) = v \) by \( \text{Ghist}(T, P) \), using steps similar to those in the proof of Theorem 5.1.

The proof for the compensation principle follows similarly, with result \( \text{restist}(T^{id}, \text{compened}) \) where now \( \text{end} \) holds in a successor state of the first state on which \( \text{status}(T^{id}) = \text{compened} \) holds.

Theorem 8.1 reduces the problem of reasoning about the satisfaction of transactional principles of IST to localized scopes, enabling use of the results in Sects. 4 and 5.

8.3. IST guideline example

The purpose of this subsection is to demonstrate use of Theorem 8.1 on the example Bank-transfer.

As in Sect. 1.2 and the example of Sect. 7.2, \( \{ \text{BankTrans, Payer} \} \subseteq \text{MsgDepend}(\text{ReqPay}) \), and none of the three participating scopes writes to variables that may change critical variables of scopes not in \( \text{Pay} \). So it suffices to show that \( \text{Ghist}(\text{Pay, Transfer}) \) holds. But \( \text{Ghist}(\text{Pay, Transfer}) \) holds because Bank\text{bal} is written only in BankTrans. Bank\text{bal} has data dependency only on Bank\text{bal} in programs of Bank-transfer other than Bank\text{Transfer}\text{F} (the other two variables affecting Bank\text{bal} are Bank\text{Trans}\text{req} and Bank\text{Trans}\text{chreq} which are scope variable of BankTrans). The problem has thus been reduced to showing that the three localized scopes \( \text{ReqPay}, \text{BankTrans} \) and \( \text{Payer} \) individually satisfy the transactional principles.

9. Validation of transactional principles

The design guidelines provide sufficient conditions to ensure a design satisfies the transactional principles. As designs may be distinguished by their applications, the guidelines cannot be expected to cover all cases of design. If a design does not conform to the guidelines, in order to ensure it is ‘correct’, that is conform to the principles, the designer can turn to other methods; one choice is model checking.

The principles in Definitions 4.1 and 8.1 are based on LTL formula using the until operator, \( W \). In some model checkers, checking formulas with until operators (\( U, W, \ldots \)) is much more expensive than checking formulas without until operators [PPS09]. Recall that the property stated in Lemma 6.2 is: ‘the status of each scope instance is not initially defined, then starts with active and stays in that status until changing to committed on termination, or to failed otherwise in which case it stays in failed until the fault is handled and the status changes to flt-hndled.’

If \( \text{until} \) operators are not supported or expensive, Lemma 6.2 can be used to cast the principles in terms of only \( G, F \) and \( X \). Given a scope \( S \), the all-or-nothing principle for \( S \) without until is that for any \( id : \mathbb{N} \),

\[
\text{restNOU}(S^{id}, \text{flt-hndled}) := \bigwedge_{x:S,y,v:\text{typ}(x)} G (S^{id} \notin \text{dom status}) \lor F (S^{id} \notin \text{dom status} \land X \text{startNOU}(S^{id}, \text{flt-hndled}, x, v))
\]
where

\[
\text{startNOU}(S^id, \text{flt-hndled}, x, v) := S^id \in \text{dom status} \land \text{status}(S^id) = \text{active} \land \\
(\text{val}(x) = v \Rightarrow \text{untilNOU}(S^id, \text{flt-hndled}, x, v))
\]

\[
\text{untilNOU}(S^id, \text{flt-hndled}, x, v) := G(\text{status}(S^id) \neq \text{flt-hndled}) \lor \text{F(\text{status}(S^id) \neq \text{flt-hndled} \land \\
X(\text{status}(S^id) = \text{flt-hndled} \land \text{val}(x) = v))}
\]

Formulas without until operators of compensation principle for S and principles for IST are analogous.

The model checker SAL \cite{dMOR04} has been used to validate the process Batch-Atm. Generally, the transition system of \text{BaT} in Z is translated to the SAL specification language \cite{dMS03} following the approach given by Smith and Wildman \cite{SW05}. SAL supports specification of the transition system non-deterministically and allows user-defined data structures and functions. In the translation, each \text{BaT} statement is given a label and \varphi is represented by a sequence of labels. Since union types are not directly supported by SAL, message queues with different endpoint services and message content types are modelled separately. Each transition of the translation into SAL denotes the handling of the statement with the first label in sequence \varphi (in the choreographies, the statement with the first label in each service thread). The principles are written as LTL theorems in SAL, in which the domain of each variable is bounded.

After the process Batch-Atm has been implemented in SAL, the SAL bounded model checker is used to validate the principle. A path bound of 40 steps is sufficient for Batch-Atm. The result given shows that the implementation passes checking; but if the second line of \text{P_{OpH}} in Sect. 5.2 is changed to \text{balance} := (\text{balance} - \text{opamnt}), a counter-example path is given, because balance will not be restored in a fault or compensation handler of DoOp.

The approach for validating the principles for IST Pay in Bank-transfer is similar. A path bounded by 80 steps is sufficient to check all possible executions. Validation of both the principles returns a positive answer if the implementation of BankTrans satisfies the transactional principles (because the critical variable sets for ReqPay and Payer are empty).

SAL implementation for both processes Batch-Atm and Bank-transfer can be found at the following url: http://seg.nju.edu.cn/~liux/pub/fac11/.

10. Conclusion

Much of service design rests on standard principles: single services are (reactive) processes; and choreographed services are a parallel composition of single services, so the theory of distributed systems applies. However compensation lies beyond that, requiring new techniques. Any theory of fault handling with compensation must be based on this idea: when compensation is required, invoke, in either reverse order or in user-defined order, procedures that cancel the effects of those previously executed. As usual, the real interest lies in the details, and those have been the purpose of this paper.

The notion of ‘critical variable’ has been introduced to identify state which must be restored, and ‘transactional principles’ defined to ensure such restoration in spite of nesting and choreography. An abstract programming language, \text{BaT}, has been introduced for the purposes of analysing scopes and stating and analysing the transactional principles. Use of \text{BaT} has the substantial advantage of ’hiding’ the semantic underpinnings from the designer. Even so, simpler criteria for fault handling with compensation are required, and supplied here in the form of ‘design guidelines’. ISTs have been modelled in terms of scopes in single-services, which has the advantage of offering simpler appreciation of the concepts first.

The semantics of \text{BaT} must of course be studied, here if not by the service designer. State and transitions on state have been given in Z, with behaviours expressed in temporal logic over state transitions. To help with validation of a design, it has been shown how the semantics of \text{BaT} can be translated into SAL and the transactional principles encoded and model checked using SAL.

The abstract language \text{BaT} of scopes has played an important rôle, facilitating the expression of otherwise semantic properties. The state-based semantics is complicated but that seems inevitable. An important feature, therefore, has been simplicity of the guidelines and the use of model checking.

Future work consists of a translation from \text{BaT} to SAL and automating validation of the transactional principles. Of further interest may be the extension of the transactional principles to cover more technical detail in various implementation languages for long-running transactions, and finding further guidelines for both scopes and ISTs, and other (distributed) long-running transaction models. Validation of generic compensation templates, using the transactional principles, is also proposed.
11. Acknowledgments

The referees are thanked for their wide ranging suggestions which have led to a substantial improvement, from an appreciation of the paper’s contribution and corrections, to improvements in notation and references. The ‘mixed-handler guideline’ was suggested by a referee, as was the observation that the change of compensation may be modelled with conditional branches in the compensation handlers. Professor Michael Butler is thanked, too, for suggesting amendments and clarifications.

Appendix

Since the body of the paper has focused on developing the main ideas, some details have been relegated to this appendix.

A. Semantic details

A.1. Semantic notation

In the following, it is convenient to let \( \text{ParProg} \) denote \( \| \text{EPSet} \), where \( \text{EPSet} \) is a nonempty finite subset of \( \text{Prog}^\dagger \). Given a \( P : \text{ParProg} \), \( \text{Threads}(P) \) denotes all the parallel program threads in \( P \):

\[
P = \| \text{Threads}(P) \).
\]

Evidently for \( P : \text{Prog}^\dagger \), \( \text{Threads}(P) = \{ P \} \).

The functions \( p\text{head} \) and \( p\text{tail} \) from Sect. 3.2 are now augmented with

\[
p\text{tail} : \text{ParProg} \times \text{Prog}^\dagger \rightarrow \text{ParProg}
\]

\[
p\text{tail}(P, \text{thr}) := P[p\text{tail}(\text{thr})/\text{thr}]
\]

where, as usual, \( P[Q/\text{thr}] \) is the result of substituting \( Q \) for \( \text{thr} \) throughout \( P \). Moreover

\[
n\text{extrmp} : \text{ParProg} \times \text{Prog}^\dagger \times \text{Prog}^\dagger \rightarrow \text{ParProg}
n\text{extrmp}(\varphi, \text{thr}, Q) := \varphi[Q \; p\text{tail}(\text{thr})/\text{thr}] .
\]

The main and supporting scope instances of any IST instance \( \text{istins} : \text{ISTIns} \) are obtained from \( \text{reg} : \text{Registry} \) by the functions \( \text{mainins} \) and \( \text{suppins} \) respectively:

\[
\text{mainins} : \text{Registry} \times \text{ISTIns} \rightarrow \text{ScopeIns}
\]

\[
\text{suppins} : \text{Registry} \times \text{ISTIns} \rightarrow \mathbb{P} \text{ScopeIns}
\]

\[
\text{mainins}(\text{reg}, \text{istins}) := \{ (\text{first istins}).\text{main} \} \triangleright (\text{first } \circ \text{reg})(\text{istins})
\]

\[
\text{suppins}(\text{reg}, \text{istins}) := (\text{first istins}).\text{supps} \triangleleft (\text{first } \circ \text{reg})(\text{istins}),
\]

where \( x \in E \) is a computation resulting in a nondeterministic choice for \( x \) over all elements of the nonempty set \( E \). The former, requiring a single element, is well defined. It is convenient to combine the results in one set:

\[
\text{partins} : \text{Registry} \times \text{ISTIns} \rightarrow \mathbb{P} \text{ScopeIns}
\]

\[
\text{partins}(\text{reg}, \text{istins}) := \{ \text{mainins}(\text{reg}, \text{istins}) \} \cup \text{suppins}(\text{reg}, \text{istins}).
\]

A.2. State space, initial state and overview of transitions

Given a choreography process \( P : \text{BaT} \), its semantics \( TS(P) \) specifies the system from a global viewpoint. That is, each state is a global state constructed from the localized states of each service; the transition characterizes the global state change in an interleaving manner.
**Compensation by design**

Fig. 14. State space of $TS(P)$. In writing schemas, indentations are used to minimise use of conjunctions and parentheses. Note the difference between the identifier `id` and the identity function $id$.

For a choreographed process $P : BaT$ the state space of $TS(P)$ is specified by Schema `State` in Fig. 14. There $\varphi$ is the program remaining for execution, $val$ is the valuation of variables, $mq$ is the message queue shared between the services, $status$ is the status of each commenced instance, $hist$ is the history (sequence of snapshots) of committed instances, $enc$ is the enclosing scope responsible for fault handling, $reg$ is the coordination register recording the participating scope instance and status of each IST instance. Given a thread $thr$ in $\varphi$, $\Psi(thr)$ is the service of the thread $thr$.

The predicates in `State` specify that the following properties hold on all states in $TS(P)$.

1. For any scope there is at most one instance of status `active` or `failed`, and there is at most one instance of `status cmpning`.
2. At most one IST instance has status in `RunningStat` (because participating scopes of an IST are disjoint).
3. For any scope instance the instance numeral lies in the interval $[0, idcnt)$, and for any IST instance, its instance numeral is the same as the instance numeral of its main scope instance.
4. For any parallel thread $thr$ of $Threads(\varphi)$,
   (a) instances with snapshot in $hist(\Psi(thr))$ are of status `committed` or `cmpning`;
   (b) snapshots with different index in $hist(\Psi(thr))$ are of different instances;
   (c) $enc(thr) = none \iff phead(thr)$ is not in the normal activity program of the exactly belonging scope.
5. If an IST instance $T^{id}$ is not of status in `RunningStat`, then none of its participating scope instances is of status in `RunningStat`.

The initial state of $TS(P)$ is specified by Schema `Init`.

---

**Table 14. State space of $TS(P)$**

<table>
<thead>
<tr>
<th>State</th>
<th>Init</th>
</tr>
</thead>
</table>
| $\varphi: ParProg$ | $\varphi = P^\dag \land status = \emptyset \land reg = \emptyset \land idcnt = 0$
| $val: Val$ | $\forall x: \text{dom } val \bullet \text{val}(x) \in \text{typ}(x)$
| $mq: \text{seq } Msg^+$ | $\forall m: \text{ran } mq \bullet m.\text{snd} \not\in P.\Psi \land m.\text{rcv} \in P.\Psi$
| $\text{idcnt}: N$ | $\forall thr: \text{Threads}(\varphi) \bullet \text{hist}(\Psi(thr)) = \langle \rangle \land enc(thr) = none$
| $\text{status}: \text{ScopeStat}$ | $\forall S: \text{Scope} \bullet \text{wait}(S) = eop$
| $\text{hist}: \text{SnapHist}$ | $\forall T^{id}: \text{dom } reg \land \text{dom } id = \text{second }\circ \text{reg}(\text{istins}) \land (\text{reg}, T^{id})$
| $\text{enc}: \text{EncScope}$ | $\forall \Psi: \text{Threads}(\varphi) \bullet$
| $\text{reg}: \text{Registrar}$ | $\forall i: [0, \# \text{hist}(\Psi(thr))] \bullet (\text{status} \circ \text{first}^2)(\text{hist}(\Psi(thr))) \in \{\text{committed}, \text{cmpning}\}$
| | $\forall i, j: [0, \# \text{hist}(\Psi(thr))] \bullet i \neq j \Rightarrow (\text{first hist}(\Psi(thr)))_i \neq (\text{first hist}(\Psi(thr)))_j$
| | $\forall S: \text{Scope} \bullet \text{wait}(S) = \text{eop}$
In the initial state, the whole (extended) system remains to execute \((\varnothing = \mathcal{P}^\dagger)\), and therefore \(\text{Threads}(\varnothing)\) is the set of services of \(\mathcal{P}\). Every variable \(x\) in \(\mathcal{P}\) is initialized to an arbitrary value of its type denoted by \(\text{typ}(x)\). Initially, the message queue is an arbitrary sequence of messages from the environment of \(\mathcal{P}\) to any of the services in \(\mathcal{P} \cdot \Psi\). Since no scopes have started yet, \(\text{status}\) is empty, and for each service thread \(\text{enc}\) is \(\text{none}\), \(\text{hist}\) is the empty sequence. Because no scope instance has started, no IST instance has started, so \(\text{reg}\) is also empty.

The existence of an initial state ensures consistency of the state schema; it is confirmed informally as follows. In the initial state \(\text{status} = \varnothing\), so predicates 1 hold. Since \(\text{reg} = \varnothing\), predicates 2 hold. Both \(\text{reg}\) and \(\text{status}\) are \(\varnothing\), so predicates 3 hold. Since \(\varnothing = \mathcal{P}^\dagger\), \(\text{Threads}(\varnothing)\) is the set of extended service scopes in \(\mathcal{P}^\dagger\); but for each extended service scopes, \(\text{hist}\) is the empty sequence, so predicate 4 hold. Finally, because \(\text{reg} = \varnothing\), predicate 5 holds.

Transitions of \(\text{TS}(\Psi)\) describe the state change after executing the current program step (i.e. the head of the program remaining to execute, \(\varnothing\)) or dealing with fault. Recall the specification of \(\text{Transit}\) by Eq. (4):

\[
\text{Transit} ::= \text{Assign} \lor \text{Send} \lor \text{Receive} \lor \text{Choices} \lor \text{Loop} \lor \text{Throw} \lor \text{FaultEnv} \lor \text{Interrupt} \lor \\
\text{ASCope} \lor \text{CompensateScope} \lor \text{CompensateScopeIns} \lor \text{DefaultHandler} \\
\text{ExitNorm} \lor \text{ExitFH} \lor \text{ExitCH} .
\]

A.3. Steps of sequential programs

Schema \textit{Assign} specifies transitions resulting from assignment.

\[
\begin{array}{ll}
\text{Assign} & \Delta \text{State} \\
\text{x : Var} & \text{e : Exp} \\
\exists \ thr : \text{Threads}(\varnothing) \bullet \\
\ & \ phead(thr) = (\text{x := exp}) \\
\ & \ #\text{x} = #\text{e} \\
\ & \ \varnothing' = \varnothing\cdot\text{tail}(\varnothing, \ thr) \\
\ & \ \text{val}' = \text{val} \oplus \{\text{x} \mapsto \text{e}, \ 0 \leq i < \#\text{x}\} \\
\ & \ \text{mq}' = \text{mq} \land \text{idcnt}' = \text{idcnt} \land \text{status}' = \text{status} \\
\ & \ \text{hist}' = \text{hist} \land \text{enc}' = \text{enc} \land \text{wait}' = \text{wait} \land \text{reg}' = \text{reg} \\
\end{array}
\]

Transitions resulting from conditional and loop are specified by Schemas \textit{Choices} and \textit{Loop}, respectively, following standard programming.

\[
\begin{array}{ll}
\text{Choices} & \Delta \text{State} \\
\text{b : Exp} & \text{P : Prog}^\dagger \\
\exists \ thr : \text{Threads}(\varnothing); \ n : \mathbb{N} \bullet \\
\ & \ phead(thr) = (\text{if } \text{b then } \text{P } \text{fi}) \\
\ & \ #\# = \#\text{P} \\
\ & \ \varnothing' \in \{\text{nextrmp}(\varnothing, \ thr, \ P_i) \mid \text{b, } 0 \leq i < \#\text{b}\} \\
\ & \ \text{val}' = \text{val} \land \text{mq}' = \text{mq} \land \text{idcnt}' = \text{idcnt} \land \text{status}' = \text{status} \land \\
\ & \ \text{hist}' = \text{hist} \land \text{enc}' = \text{enc} \land \text{reg}' = \text{reg} \land \text{wait}' = \text{wait} \\
\end{array}
\]
A.4. Messages

The transitions dealing with message sending are specified by Schema Send.

\[\text{Send} \]
\[\Delta \text{State} \]
\[e : \text{Exp} \]
\[\exists \; \text{thr} : \text{Threads}(\phi) \; \bullet \]
\[\text{phead}(\phi, \text{thr}) = (\text{do } e \text{ then } \mathcal{P} \text{ od}) \]
\[#b = \#\mathcal{P} \]
\[\text{if } \forall (b_i | 0 \leq i < #b) \]
\[\text{then } \phi' = [\phi[(P_i, \text{thr})/\text{thr}] | b_i, 0 \leq i < #b] \]
\[\text{else } \phi' = \text{phead}(\phi, \text{thr}) \]
\[\text{val}' = \text{val} \land \text{mq}' = \text{mq} \land \text{idcnt}' = \text{idcnt} \land \text{status}' = \text{status} \land \]
\[\text{hist}' = \text{hist} \land \text{enc}' = \text{enc} \land \text{wait}' = \text{wait} \land \text{reg}' = \text{reg} \]

Transitions corresponding to message receipt are specified by Schema Receive. If for the received message \(m\) \(\text{faulty}(m.\text{msg}) = \text{true}\), then the received message carries a fault from the sender. When \(\text{enc} \neq \text{none}\), the next program remaining to execute starts with \(\text{throw}\), otherwise the receipt succeeds.

When receiving messages, state observable \(\text{reg}\) is updated using \(\text{curins}\) and \(\text{updfirst}\). Function \(\text{curins}\) gives the belonging scope instance of the current program step such that the instance is of status \(\text{active}\) and is a supporting scope of the IST.

\[\text{curins} : \text{Basic}^\dagger \times \text{ScopeStat} \times \text{IST} \to \text{ScopeIns} \]
\[\text{curins}(ps, \text{status}, T) := \]
\[\text{if } \exists \; S : \text{belong}(ps) \cap T.\supps \land \text{id} : \mathbb{N} \; \bullet \; \text{status}(S^{id}) = \text{active} \]
\[\text{then } \{S^{id}\} \]
\[\text{else } \emptyset \]

Suppose \(T : \text{IST}, \text{id} : \mathbb{N}, \text{status} : \text{ScopeStat}, \text{reg} : \text{Registry}\) and the current program step is \(ps\). Function \(\text{updfirst}\) gives a new tuple where the first element of \(\text{reg}(T^{id})\) is updated by adding the \(\text{curins}(ps, \text{status}, T)\), and the second element of \(\text{reg}(T^{id})\) stays the same.

\[\text{updfirst} : \text{ISTIns} \times \text{Basic}^\dagger \times \text{ScopeStat} \times \text{Registry} \to (\mathcal{P} \text{ScopeIns}) \times \text{Status} \]
\[\text{updfirst}(T^{id}, ps, \text{status}, \text{reg}) := ((\text{first} \circ \text{reg})(T^{id}) \cup \text{curins}(ps, \text{status}, T), (\text{second} \circ \text{reg})(T^{id})) \]

The function \(\text{actIST}\) returns, from the state observable \(\text{reg}\), the only IST instance of status \(\text{active}\); if such an instance does not exist, the result is \(\text{none}\).
\[actIST \colon Registry \rightarrow \{\text{none}\} \cup ISTIns\]

\[actIST(reg) \coloneqq \begin{cases} \text{if } \exists istins \colon ISTIns \bullet (second \circ reg)(istins) = \text{active} & \text{then } istins \\ \text{else } \text{none} & \end{cases}\]

Due to the well-formedness conditions, a service can receive only messages whose receiver matches the service name. Given a message \(m : Msg\) and a message queue \(mq : \text{seq} \ Msg\), \(lastind(mq, m)\) is defined to return the index of the last message in \(mq\) that matches the receiver and message type of \(m\):

\[\text{lastind} : \text{seq} \ Msg \times Msg \rightarrow \mathbb{Z}\]

\[\text{lastind}(\langle \rangle, m) \coloneqq -1\]

\[\text{lastind}(mq \langle x \rangle, m) \coloneqq \begin{cases} m.rcv = m.rcv \land \text{typ}(x.\text{msg}) = \text{typ}(m.\text{msg}) & \text{then } #mq \\ \text{else } \text{lastind}(mq, m) & \end{cases}\]

Finally message receipt can be specified as follows.

A fault is thrown if any one of the elements in the message vector is a fault:

\[\exists i : [0, \#m.\text{msg}) \bullet \text{faulty}(m.\text{msg}_i)\].

### A.5. Fault

Dealing with fault is more complicated, so several functions are useful.

A function \(\text{beforeend} : \text{Prog} \rightarrow \text{Prog}\) is now defined, to return the program before and including the first end.

\[\text{beforeend}(P) \coloneqq \begin{cases} \text{phead}(P) & \text{if } \text{phead}(P) \in \{\text{end}, \text{eop}\} \\ \text{phead}(P) \triangleright (\text{beforeend} \circ \text{ptail})(P) & \text{otherwise}. \end{cases}\]

The function \(\text{intrcmpn}\) gives either \(\text{throw}\) (an interruption) or compensation of the scope instance according to the status of the instance of the input snapshot.

\[\text{intrcmpn} : \text{Snapshot} \times \text{ScopeStat} \rightarrow \text{Basic}\]

\[\text{intrcmpn}(\text{snap}, \text{status}) \coloneqq \begin{cases} \text{throw} & \text{if } (\text{status} \circ \text{first})(\text{snap}) = \text{active} \\ \text{callch}(\text{snap}) & \text{else} \end{cases}\]

The function \(\text{cmpnmi}\) gives the program to compensate a set of instances in the order of \(\text{hist}(\Psi(\text{thr}))\) for a given \(\text{thr} : \text{Prog}\).
Compensation by design

\[
\text{cmpnmi} : (\text{Prog}^+ \times \mathcal{F} \times \text{ScopeIns} \times \text{ScopeStat}) \rightarrow \text{Prog}^+
\]

\[
\text{cmpnmi}(\text{thr}, \text{InsSet}, \text{status}) := \\
\text{if} \quad \#(\text{hist}(\mathcal{F}(\text{thr})) \setminus \text{InsSet}) > 0 \\
\text{then} \quad (\text{seq} \cdot \text{intcmpn})(\text{hist}(\mathcal{F}(\text{thr})) \setminus \text{InsSet}) \\
\text{else} \quad \text{ptail}(\text{thr})
\]

The function \(fhIST\) returns the next \(\phi\) when handling a fault of an IST in the case that the input scope instance is the main scope instance of the input IST. It is used when a fault occurs in the main scope or the main scope is required to be compensated. The returned program thread of the main scope is denoted \(nxt\).

\[
fhIST : (\text{ParProg} \times \text{Prog}^+ \times \text{Registry} \times \text{ISTIns} \times \text{ScopeIns} \times \text{ScopeStat}) \rightarrow \text{ParProg}
\]

\[
fhIST(\phi, \text{thr}, \text{reg}, \text{istins}, \text{nxt}, \text{status}) := \\
\{\left([\text{nxt}] \cup \{\text{interrupt}\} [\text{istins}] \right) \setminus \{\text{oth} \mid \text{oth} \in \text{Threads}(\phi) \setminus \{\text{thr}\})
\]

The function \(Thf\) is used to prepare the next program remaining to execute. If the input scope instance is the main scope of some IST, then return the program given by \(fhIST\) which is a parallel composition of the fault handler followed by the program following the first end (which is the program following the main scope) and compensation of supporting scope instances followed by the original program of each thread; otherwise, the input thread is changed to the scope’s fault handler followed by the program following the first end.

\[
Thf : (\text{ScopeIns} \times \text{ParProg} \times \text{Prog} \times \text{ScopeStat} \times \text{Registry}) \rightarrow \text{ParProg}
\]

\[
Thf(\text{sins}, \phi, \text{thr}, \text{status}, \text{reg}) := \\
\text{if} \exists \text{istins} : \text{dom reg} \cup \text{mainins}(\text{reg}, \text{istins}) = \text{sins} \\
\text{then} \quad fhIST(\phi, \text{thr}, \text{reg}, \text{istins}, ((\text{first ins}) \downarrow \text{afterend}(\text{thr})), \text{status}) \\
\text{else} \quad P((\text{first ins}).F \downarrow \text{afterend}(\text{thr})/\text{thr})
\]

A fault may arise explicitly from \text{throw} or implicitly from the environment. The transition dealing with environmental faults is specified by the Schema \text{FaultEnv}. (It is assumed that when the program of the scope component finishes, i.e. head program step is \text{end}, environmental fault is ignored).

\[
\text{FaultEnv}
\]

\begin{align*}
\Delta \text{State} \\
\exists \text{thr} : \text{Threads}(\phi) \bullet \\
\text{phead}(\text{thr}) & \neq \text{end} \\
\phi' & = \text{nexttmp}(\phi, \text{thr}, \text{throw}) \\
\text{val}' & = \text{val} \land \text{mq}' = \text{mq} \land \text{idcnt}' = \text{idcnt} \land \text{status}' = \text{status} \land \\
\text{hist}' & = \text{hist} \land \text{enc}' = \text{enc} \land \text{wait}' = \text{wait} \land \text{reg}' = \text{reg}
\end{align*}

\[
\text{Throw}
\]

\begin{align*}
\Delta \text{State} \\
\exists \text{thr} : \text{Threads}(\phi); \text{id} : \mathbb{N} \bullet \\
\text{phead}(\text{thr}) & = \text{throw} \\
\text{status}(\text{enc}(\text{thr})^{id}) & = \text{active} \\
\phi' & = Thf((\text{enc}(\text{thr})^{id}), \phi, \text{thr}, \text{status}, \text{reg}) \\
\text{status}' & = \text{status} \sqcup \{\text{enc}(\text{thr})^{id} \mapsto \text{failed}\} \\
\text{mq}' & = \text{mq} \mapsto \{\text{m, sins} : \text{Msg} | \text{sins} = \text{enc}(\text{thr})^{id}\} \\
\text{reg}' & = \text{if} \exists \text{istins} : \text{ISTIns} \bullet \text{enc}(\text{thr})^{id} = \text{mainins}(\text{reg}, \text{istins}) \\
& \text{then} \quad \text{reg} \sqcup \{\text{istins} \mapsto \text{partins}(\text{reg}, \text{istins}), \text{failed}\} \\
& \text{else} \quad \text{reg} \\
\text{enc}' & = \text{enc} \sqcup \{\text{thr} \mapsto \text{none}\} \\
\text{wait}' & = \text{wait} \sqcup \{\text{S} \mapsto \text{seqneop}(\text{wait}(\text{S}), \text{cmpnmi}(\text{oth}, \suppins(\text{reg}, \text{istins}), \text{status})) | \\
& \text{enc}(\text{thr})^{id} = \text{mainins}(\text{reg}, \text{istins}) \land \text{S} \in (\text{first ins}).\text{supps}\} \\
\text{idcnt}' & = \text{idcnt} \land \text{val}' = \text{val} \land \text{hist}' = \text{hist}
\end{align*}
Transitions dealing with `throw` are specified by Schema *Throw*, in which \( \varphi \) on the next state is changed to fault handling, the failed scope instance is changed to status `failed`, the IST instance with the scope instance as a main scope is also changed to status `failed`, and for any supporting scope \( S \) of the failed IST instance, `wait(S)` is changed to (if `wait(S) = eop`) or appended with compensation of all participating scope instances, which is to be executed immediately after the already-running supporting scope's fault or compensation handler (cf. Schemas *ExitFH*, *ExitCH*). Note that for any \( W, P : \text{Prog}^\dagger \),

\[
\text{seqneop}(W, P) \triangleq \begin{cases} (W = eop) & \text{then } P \text{ else } (W ; P). \end{cases}
\]

Moreover, before the fault handler of the scope instance begins to execute, it is necessary to flush the messages sent by the failed scope instance, that is, to remove all (marked) messages sent by such instance from the message queue. This is also done before the compensation of a scope instance (cf. Schema *CompensateScopeIns*).

The interruption `interrupt[T]` is specified by Schema *Interrupt*, in which \( \varphi \) on the next state is changed by inserting `throw` at the head and `interrupt[T]` immediately after the first end in the corresponding thread only if the enclosing scope `enc(thr)` is a descendant scope of a supporting scope of \( T \). Command `interrupt[T]` is repeated because all scopes in \( T \) are required to be interrupted no matter the depth of the scope enclosing the thread. If the interruption is required in the middle of fault or compensation handling of a descendant scope of \( S \), it waits until the fault or compensation handling finishes execution (by `seqneop`).

\[
\text{Interrupt}
\]

\begin{align*}
\Delta \text{State} \\
T : \text{IST} \quad \exists \ thr : \text{Threads}(\varphi) \cdot \\
\quad \text{phead}(\text{thr}) = \text{interrupt}[T] \quad \varphi' = \begin{cases} \varphi[\text{throw} \ beforeend(\text{thr}) ; \text{interrupt}[T] ; afterend(\text{thr})] & \text{if } \exists S \in T.supps \cdot \text{enc}(\text{thr}) \in \text{child}^\ast(S) \\
\quad \text{else } \varphi[\text{tail}(\varphi, \text{thr})] \\
\text{wait'} = \begin{cases} \varphi[\text{wait} \ \text{seqneop}(\text{wait}(S), \text{interrupt}[T])] & \text{if } \exists S \in T.supps ; T \in \text{child}^\ast(S) ; \text{idt} : \mathbb{N} \cdot \text{status}(T{idt}) \in \{\text{failed}, \text{cmpning}\} \\
\quad \text{else } \text{wait} \\
\text{idcnt'} = \text{idcnt} \wedge \text{val'} = \text{val} \wedge \text{mq'} = \text{mq} \wedge \text{status'} = \text{status} \wedge \\
\text{hist'} = \text{hist} \wedge \text{enc'} = \text{enc} \wedge \text{wait'} = \text{wait} \wedge \text{reg'} = \text{reg}
\end{cases}
\end{align*}

**A.6. Compensation**

Taking advantage of the above two functions, the transition dealing with scope compensation is specified by Schema *CompensateScope*. The next program remaining to execute starts with the compensation of the last instance of the scope.
Compensation by design

The transitions for default fault and compensation handler `DH` are specified by Schema `DefaultHandler`, which compensates child scope instances in the same order as in `hist` (reverse to the order of commitment, see Lemma 6.3).

The transitions to compensate one specific scope instance are specified by Schema `CompensateScopeIns`. The next program to execute starts from the compensation handler of the scope, and the valuation recorded in `hist` is retrieved. All messages that are sent by the scope instance are flushed from the message queue. The instance status is changed to `compning`. If the scope is a main scope of some IST, all participating scope instances are also compensated using function `chIST`, which is similar to `fhIST`, but instead of interruption, compensation of participating scopes is inserted on other threads.

```
chIST : (ParProg × Prog† × Registry × ISTIns × ScopeIns × ScopeStat) → ParProg
chIST(φ, thr, reg, istins, nxt, status) :=
∥(nxt) ∪ \{compnmi(oth, suppins(reg, istins), status) | oth ∈ Threads(φ) \ {thr}}
```

The transitions to execute the next program specified by Schema `CompensateScope`.

```
CompensateScope

\[
\Delta State
\]
S : Scope

\[
istNotRun(S)
\]
\begin{align*}
\exists thr : Threads(φ) \bullet \\
phead(thr) &= \text{comp}[S] \\
\text{if } & \#(hist(ψ(thr)) \upharpoonright_{\text{hist}} S) > 0 \\
\text{then let } & S' := (\text{first} \circ \text{head})(hist(ψ(thr)) \upharpoonright_{\text{hist}} S) \text{ in } \text{nextrmp}(φ, thr, \text{comp}[S']) \\
\text{else } & \varphi' = \text{p.tail}(φ, thr) \\
val' &= \text{val} \& \text{mq}' = \text{mq} \& \text{idcnt}' = \text{idcnt} \& \text{status}' = \text{status} \& \text{hist}' = \text{hist} \& \text{enc}' = \text{enc} \& \text{wait}' = \text{wait} \& \text{reg}' = \text{reg}
\end{align*}
```

```
DefaultHandler

\[
\Delta State
\]
\begin{align*}
\exists thr : Threads(φ) \bullet \\
phead(thr) &= \text{DH} \\
\text{let } & S := (\text{exbelong} \circ \text{phead})(thl) \text{ in } \varphi' := \\
\text{if } & \#(hist(ψ(thl)) \upharpoonright_{\text{hist}} S) > 0 \\
\text{then } & \text{nextrmp}(φ, thr, \text{seqle} \circ \text{callch})(hist(ψ(thl)) \upharpoonright_{\text{hist}} S) \\
\text{else } & \varphi' = \text{p.tail}(φ, thr) \\
val' &= \text{val} \& \text{mq}' = \text{mq} \& \text{idcnt}' = \text{idcnt} \& \text{status}' = \text{status} \& \text{hist}' = \text{hist} \& \text{enc}' = \text{enc} \& \text{wait}' = \text{wait} \& \text{reg}' = \text{reg}
\end{align*}
```

```
CompensateScopeIns

\[
\Delta State
\]
S : Scope

\[
id : \mathbb{N}
\]
thr : Prog†

\[
phead(thr) = \text{comp}[S'] \\
\text{if } \#(hist(ψ(thr)) \upharpoonright_{\text{hist}} S) = 0 \\
\text{then } & \varphi' = \varphi \& \text{val}' = \text{val} \& \text{mq}' = \text{mq} \& \text{status}' = \text{status} \& \text{reg}' = \text{reg} \\
\text{else } & \text{val}' = \text{val} \circ (hist(ψ(thr)) \upharpoonright_{\text{hist}} S) \\
\text{status}' &= \text{status} \circ \text{S} \leftarrow \text{compnmi} \\
\text{mq}' &= \text{mq} \circ \{(m, \text{ins}) : \text{Msg} \circ \text{ins} = S\} \\
\text{if } & \exists \text{istins : ISTIns } \bigodot S = \text{mainins}(\text{reg}, \text{istins}) \\
\text{then } & \varphi' = \text{chIST}(φ, thr, reg, T, (S.\hat{C} \circ \text{ptail}(thl)), \text{status}) \\
\text{reg}' &= \text{reg} \circ \{(\text{istins} \mapsto (\text{partins}(\text{reg}, \text{istins}), \text{compnmi})) \} \\
\text{else } & \varphi' = \text{nextrmp}(φ, thr, (S.\hat{C} \circ \text{ptail}(thl))) \& \text{reg}' = \text{reg} \\
idcnt' &= \text{idcnt} \& \text{hist}' = \text{hist} \& \text{enc}' = \text{enc} \& \text{wait}' = \text{wait}
```

A.7. Scopes

Predicate \textit{istNotRun}(S) is defined on scopes \(S\):
\[
\text{istNotRun}(S) \iff \forall T : \text{IST}; \ idt : \mathbb{N} \bullet \quad (T^{id} \in \text{dom reg} \land S = T.\text{main}) \Rightarrow (\text{second} \circ \text{reg})(T^{id}) \notin \text{RunningStat}.
\]

<table>
<thead>
<tr>
<th>AScope</th>
</tr>
</thead>
<tbody>
<tr>
<td>ΔState</td>
</tr>
<tr>
<td>( S : \text{Scope} )</td>
</tr>
<tr>
<td>istNotRun(S)</td>
</tr>
<tr>
<td>( \exists \text{thr} : \text{Threads}(\varnothing) \bullet )</td>
</tr>
<tr>
<td>phead(\text{thr}) = S \land</td>
</tr>
<tr>
<td>( \varnothing' = \text{nexttmp}(\varnothing, \text{thr}, S', N) )</td>
</tr>
<tr>
<td>status' = status \oplus {S^{idcnt} \mapsto \text{active} }</td>
</tr>
<tr>
<td>reg' = \text{if } \exists_1 T : \text{IST} \bullet T.\text{main} = S</td>
</tr>
<tr>
<td>then reg \oplus {T^{idcnt} \mapsto ((S^{idcnt}), \text{active}) }</td>
</tr>
<tr>
<td>\text{else reg}</td>
</tr>
<tr>
<td>enc = S</td>
</tr>
<tr>
<td>idcnt' = idcnt + 1</td>
</tr>
<tr>
<td>val' = val \land mq' = mq \land hist' = hist \land wait' = wait</td>
</tr>
</tbody>
</table>

The transitions handling a new scope are specified by Schema AScope. The next program remaining to execute in the current thread starts from the normal program of the scope. A new instance of the scope is added and the next status is active. If this scope is the main scope of some IST, a new IST instance is registered with status active. Note that a scope variable may take an arbitrary value before being assigned to.

Transitions handling end in scope’s normal activity program, fault handler and compensation handler are respectively specified by Schemas ExitNormal, ExitFH and ExitCH.

A boolean function \(\text{partstatus}\) (or a predicate) is defined to give true iff all participating scope instances of the input IST instance are not of status in \(\text{RunningStat}\) and the IST instance is of status \(\text{stt}_1\) while the main scope instance is of status \(\text{stt}_2\).

| partstatus : ISTIns, Registry, ScopeStat, Status, Status → \mathbb{B} |
| partstatus(istins, reg, status, \text{stt}_1, \text{stt}_2) := |
| \forall sin : \text{partins}(reg, istins) \bullet \text{status}(sin) \notin \text{RunningStat} |
| (\text{second} \circ \text{reg})(istins) = \text{stt}_1 |
| (\text{status} \circ \text{mainins})(reg, istins) = \text{stt}_2 |
Compensation by design

\[\text{ExitNorm}\]
\[\Delta\text{State}\]
\[S : \text{Scope}\]
\[x : \mathbb{N} \to \text{Var}\]
\[\text{thr} : \text{Prog}\]

\[\exists \text{thr} \in \text{Threads}(\varnothing) \quad \bullet\]
\[\text{phead}(\text{thr}) = \text{end}\]
\[\text{improg} \circ \text{phead}(\text{thr}) = S.N\]
\[s' = \text{var}(s, \text{thr})\]
\[\text{val} = \text{val}(s', \text{thr})\]
\[\text{end} = \text{end} \oplus \{\text{thr} \mapsto \text{if } \exists T : \text{Scope } \bullet S \in \text{child}(T) \text{ then } T \text{ else } \text{none} \}\]
\[\text{let id } = \{\text{where status(S')} = \text{active}\]
\[\text{in status'} = \text{status}\oplus\{S' = \text{committed}\}
\[\text{hist}' = \text{hist}\oplus\{\text{thr} \mapsto ((S', S.V \leftarrow \text{val}) \leftarrow \text{hist}(\Psi(\text{thr}))\}
\[\text{reg}' = \text{if } (\exists \text{istins} : \text{findISTIns}(\text{reg}, S') \bullet
\text{partstatus(istins, reg, (status } \cup \text{ status'), active, committed)})
\text{then reg } \oplus \{\text{istins} \mapsto ((\text{first } \text{reg})(\text{istins}), \text{committed})\}
\text{else reg}
\]
\[\text{idcnt'} = \text{idcnt} \land \text{mq} = \text{mq} \land \text{wait}' = \text{wait}\]

When ending the normal activity program of a scope (cf. \text{ExitNorm}), the scope instance is changed from active to committed, instance snapshot is added as head of hist and when the next status of all participating scope instances are not in \text{RunningStat} and the main scope instance is committed, the whole IST instance is changed to committed. (Recall that any scope has only one parent scope, unless it is a top-level scope in which case it has none, where a top-level scope is a scope that is not a child scope of any other scope; for example, services are top-level scopes).

Then ending of fault and compensation handling [respectively cf. \text{ExitFH} and \text{ExitCH}] is similar to the ending of a normal activity program, the difference being in the current and next status, and the next \varnothing depends on wait(S).

In Schema \text{ExitCH}, predicate \text{val} = S.V \leftarrow \text{val} removes valuation of scope variable of S from val. Predicate \text{hist}' = \text{hist} \oplus \{\text{thr} \mapsto ((S' = \text{val}) \leftarrow \text{hist}(\Psi(\text{thr}))\} requires the removal of the instance snapshot, which is of status cmpning, from hist. Snapshots in \text{hist}(\Psi(\text{thr})) with different indices are of different instances, so the transition specified by \text{ExitCH} removes one and only one snapshot from \text{hist}(\Psi(\text{thr}))

Note that no transition is enabled in state s : State if, for any thr : Threads(s), phead(thr) = eop.
ExitFH

$\Delta \text{State}$

$S : \text{Scope}$

$x : \mathbb{N} \rightarrow \text{Var}$

$
\exists \text{thr} : \text{Threads}(\phi) \bullet
\begin{align*}
&\text{phead}(\text{thr}) = \text{end} \\
&(\text{inprog} \circ \text{phead})(\text{thr}) = S^\delta . F \\
&\phi \equiv \text{execwait}(\phi, \text{phead}(\text{thr})) \\
&v' = S, V \equiv v \\
&\text{enc} = \text{enc} \oplus (\text{thr} \mapsto \text{if } \exists T : \text{Scope} \bullet S \in \text{child}(T) \text{ then } T \text{ else } \text{none})
\end{align*}

$\begin{align*}
\text{let } &id : \mathbb{N} \text{ where } \text{status}(S^{id}) = \text{failed} \\
\text{in } &\text{status}' = \text{status} \oplus \{S^{id} \mapsto \text{flt-hndled}\} \\
\text{reg}' = &\text{if } \exists \text{istins} : \text{findISTIns}(\text{reg}, S^{id}) \bullet \\
&\text{partstatus}(\text{istins}, \text{reg}, (\text{status} \cup \text{status}'), \text{failed}, \text{flt-hndled}) \\
&\text{then } \text{reg} \oplus \{\text{instin} \mapsto ((\text{first oreg})(\text{istins}), \text{flt-hndled})\} \\
&\text{else } \text{reg}
\end{align*}$

wait' = wait \oplus \{S \mapsto \text{eop}\}

ident' = ident \land \text{hist}' = \text{hist} \land \text{mq'} = \text{mq}

ExitCH

$\Delta \text{State}$

$S : \text{Scope}$

$\exists \text{thr} : \text{Threads}(\phi) \bullet
\begin{align*}
&\text{phead}(\text{thr}) = \text{end} \\
&(\text{inprog} \circ \text{phead})(\text{thr}) = S^\delta . C \\
&\phi \equiv \text{execwait}(\phi, \text{phead}(\text{thr})) \\
&v' = S, V \equiv v \\
\text{let } &id : \mathbb{N} \text{ where } \text{status}(S^{id}) = \text{cmpning} \\
\text{in } &\text{status}' = \text{status} \oplus \{S^{id} \mapsto \text{cmpning}\} \\
\text{hist}' = &\text{hist} \oplus \{\text{thr} \mapsto ((S^{id} \_)) \mapsto \text{hist}(\Psi(\text{thr}))\} \\
\text{reg}' = &\text{if } \exists \text{istins} : \text{findISTIns}(\text{reg}, S^{id}) \bullet \\
&\text{partstatus}(\text{istins}, \text{reg}, (\text{status} \cup \text{status}'), \text{cmpning}, \text{cmpning}) \\
&\text{then } \text{reg} \oplus \{\text{instin} \mapsto ((\text{first oreg})(\text{istins}), \text{cmpning})\} \\
&\text{else } \text{reg}
\end{align*}$

wait' = wait \oplus \{S \mapsto \text{eop}\}

ident' = ident \land \text{mq'} = \text{mq} \land \text{enc'} = \text{enc}

References


Compensation by design


Received 21 November 2011
Revised 18 January 2013
Accepted 30 January 2013 by M.J. Butler