

UNIVERSITY OF SOUTHAMPTON

**Precise Modelling of Business Processes
with
Compensation**

CARLA FERREIRA

*A dissertation submitted for the degree
of Doctor of Philosophy*

November 2002

UNIVERSITY OF SOUTHAMPTON

ABSTRACT

FACULTY OF ENGINEERING

ELECTRONICS AND COMPUTER SCIENCE

Doctor of Philosophy

PRECISE MODELLING OF BUSINESS PROCESSES WITH COMPENSATION

by Carla Ferreira

This thesis presents a formal business process modelling language called StAC. The distinctive feature of this language is the concept of compensation, which can be defined as the action taken to correct any errors or when there is a change of plan. The motivation for developing StAC came from a collaboration with IBM concerning the extension of existing notions of compensation for business transactions within the BPBeans enterprise technology.

The StAC language supports sequential and concurrent processes, as well as compensation and early termination. A system specification has two components, the StAC processes that describe the system behaviour and compensation information, and the B specification that describes the system state and its basic operations. The language has two variants: the basic one that supports nested compensation; and an extended one where each process may have multiple compensations. Both StAC variants were applied to several examples and it emerged that both variants have features that make them useful to specify different types of systems. An operational semantics was defined for StAC extended with multiple compensation, and the interpretation of basic StAC was defined in terms of the extended language. An operational approach is also used to justify the integration of StAC processes with B operations.

A strategy for the refinement of StAC specifications is explored in this thesis. This strategy proposes to explicitly embed the behavioural and compensation information into a B machine. The machine obtained is standard B, allowing the use of the B notion of system refinement to prove the refinement of StAC specifications.

An extension to UML activity diagrams is defined as a way of making StAC more accessible to non-formal methods users. In UML a StAC specification is modelled by a class diagram that describes the data, and an activity diagram that describes the behaviour of the system.

StAC has formalised the notion of compensation while extending the notion of transaction compensation in several ways. The most relevant extensions are: the non-atomicity of compensation that allows a compensation to be a complex process and that a compensation itself can be compensated; and multiple compensation that allows a process to have several independent compensations.

Contents

1	Introduction	1
1.1	Transactions and Failure	2
1.1.1	ACID Transactions	3
1.1.2	Compensation	4
1.1.3	BPBeans	4
1.2	Related Work	6
1.2.1	Formal Notations	6
1.2.2	Graphical Notations	10
1.2.3	Combined Notations	13
1.2.4	Transaction Processing Models	15
1.2.5	Business Process Modelling Languages	17
1.3	Overview	20
2	The StAC Language	23
2.1	Introduction	23
2.2	StAC Operators	24
2.2.1	Basic Operators	24
2.2.2	Sequential and Parallel Operators	25
2.2.3	Choice	26
2.2.4	Early Termination	27
2.2.5	Compensation Operators	28
2.2.6	Scoping of Compensation	30
2.3	StAC Examples	30
2.3.1	Raffle	31
2.3.2	E-Bookstore	35
2.3.3	Order Fulfillment	36

3	Extending StAC with Multiple Compensation	39
3.1	Extended Compensation Operators	40
3.2	Selective Compensation: Travel Agency Example	42
3.3	Alternative Compensation: Arrange Meeting Example	46
4	Semantics	50
4.1	Introduction	50
4.2	Operational Semantics for Compensation	51
4.2.1	Normalisation	52
4.2.2	Operational Rules	55
4.3	Operational Semantics for Termination	61
4.3.1	Normalisation	64
4.3.2	Operational Rules	66
4.4	Examples	67
4.5	Executable Semantics	71
4.6	Translation from StAC to StAC _i	73
4.7	Integration of StAC _i and B	76
4.8	Discussion	79
5	Refinement	81
5.1	Refinement Approaches	81
5.2	StAC Refinement	83
5.2.1	Embedding StAC into B	85
5.2.2	Refinement in B	98
5.3	Case Study	99
5.3.1	Dealing with Single Clients	104
5.3.2	Dealing with Multiple Clients	115
5.4	Discussion	117
6	Extending UML for Modelling StAC Specifications	119
6.1	Introduction	119
6.2	Representing StAC in UML	121
6.2.1	Representing Data	121
6.2.2	Representing Behaviour	122

6.3	Examples	124
6.3.1	Arrange Meeting	124
6.3.2	E-Bookstore	129
6.3.3	Order Fulfillment	131
6.4	Discussion	132
7	Discussion	134
7.1	Conclusions	134
7.2	Related Work	136
7.2.1	Compensation	137
7.2.2	Combining B with Process Algebras	138
7.2.3	Representing Formal languages in UML	139
7.3	Future Work	140
A	Examples	141
A.1	Raffle	141
A.2	E-Bookstore	144
A.3	Order Fulfillment	147
A.4	Travel Agency	150
A.5	Arrange Meeting	156
B	Semantics - Auxiliary Functions	159
B.1	Labelling Function	159
B.2	Normalisation Functions	160
B.2.1	<i>norm</i>	160
B.2.2	<i>normalised</i>	161
B.2.3	<i>terminate</i>	161
C	StAC Animator	162
C.1	Operational Rules	162
C.2	Normalisation Functions	164
C.2.1	<i>normalisation</i>	164
C.2.2	<i>norm</i>	164
C.2.3	<i>normalised</i>	165
C.2.4	<i>terminate</i>	166

D Examples modelled in UML	167
D.1 Arrange Meeting	167
D.1.1 Class Diagram	167
D.1.2 Compensation Activity Diagrams	168
D.2 E-Bookstore	170
D.2.1 Class Diagram	170
D.2.2 Compensation Activity Diagrams	170
D.3 Order Fulfillment	173
D.3.1 Class Diagram	173
D.3.2 Compensation Activity Diagrams	173
 Bibliography	 176

List of Figures

1.1	BPBeans example	5
1.2	Abstract machine structure	9
1.3	Statechart example	11
1.4	Activity diagram example	13
1.5	csp2b example	14
1.6	AMBER example (behaviour model)	18
1.7	RAD example	20
4.1	The StAC animator	72
4.2	$[S]$ and $\langle S \rangle$ rules	77
5.1	StAC refinement	84
5.2	STD for <i>Counter</i>	86
5.3	State of the integral B machine	94
5.4	STD for <i>Client0</i>	105
5.5	STD for <i>Client1</i>	107
5.6	AtelierB interactive prover	112
6.1	Class structure and associations between classes	120
6.2	State hierarchy in activity diagrams	120
6.3	Class diagram for the arrange meeting example	125
6.4	Activity diagram for process <i>ArrangeMeeting</i>	126
6.5	Activity diagram for process <i>CheckRoom</i>	127
6.6	Activity diagram for process <i>Decide</i>	128
6.7	Activity diagram for process <i>ChooseBooks</i>	130
6.8	Activity diagram for process <i>ChooseBook</i>	131
6.9	Activity diagram for process <i>FulfillOrder</i>	132

List of Tables

2.1	StAC Syntax	24
3.1	StAC _i Syntax	40

Chapter 1

Introduction

The motivation for this work came from the collaboration of the Declarative Systems and Software Engineering (DSSE) Group with the Transaction Processing Design and New Technology Development Group at IBM UK Laboratories, Hursley. The collaboration has been concerned with approaches and techniques for component-based enterprise system development, in particular, with IBM's BPBeans (Business Process Beans) technology, that is a feature of the IBM WebSphere Application Server Enterprise Edition. We started our work by analysing the underlying semantics of the graphical design language of the Application Builder for Components (ABC) tool [CVG⁺01b], a tool that supports the development of BPBeans applications. The ABC tool has the following basic features:

- The target enterprise solution is built by composing Enterprise Java Beans (EJBs) [MH99].
- The tool allows sequential and parallel composition of behaviours.
- It should be possible for earlier actions to be compensated, whereby the system keeps track of the compensations that need to be executed if part of a process is to be aborted.

The IBM group believes that compensation gives more flexibility than the traditional commit-rollback approach to transaction processing. This flexibility is necessary for the heterogeneous distributed environment on which modern enterprise systems operate. In the case of abnormal events, instead of restoring the system to the state before activities were performed, activities can have a compensation activity associated with them.

Compensation is a key concept in this thesis, therefore in this paragraph we will try to describe what is our interpretation of compensation. In the context of business transactions, Gray [GR93] defines a compensation as the action taken to correct any errors or when there is a change of plan. Consider the following example, a client buys some books in an on-line bookstore, the bookstore debits the client's account as the payment for the book order. The bookstore later realises that one of the books in the client's order is out of print. To compensate the client for this problem, the bookstore can credit the account with the amount wrongfully debited and send a letter apologising for their mistake. This example shows that a compensation can be more than just undoing previous actions.

The main goal of our work is to clarify and understand the mechanics of compensation, which is a complex concept. The complexity arises in particular because of the combination of compensation with parallel execution. We have defined a formal business process modelling language based on the ABC design language, called StAC [BF00] (**Str**uctured **A**ctivity **C**ompensation). StAC supports sequential and concurrent processes, as well as compensation. In ABC, the description of a system determines the way to “connect” simple components (EJBs) in order to create a complete system. In StAC, the components are B [Abr96] operations (B is a model-oriented formal notation). We use B instead of EJBs because it provides a higher-level description, and we want the StAC language to have a formal semantics. In our approach a system specification has two components: the StAC specification that describes the execution order of the operations including ordering of compensation operations, and a B specification that describes the state of the system and its basic operations.

In Section 1.1 we briefly describe business transactions and ways of recovery from failures. Section 1.2 presents some related work that could be used in business process modelling.

1.1 Transactions and Failure

The concept of transaction is described in [GR93] as a collection of operations on the application state. A standard example of a transaction is the credit or debit

of money from a bank account. The crucial problems in transaction processing are maintaining state consistency when concurrent transactions are reading and changing the state, and recovering from failure. ACID transactions (which satisfy the atomicity, consistency, isolation, and durability properties) were introduced to deal with consistency and failure in transaction processing.

1.1.1 ACID Transactions

ACID transactions were presented by Gray in 1983 [GR93]. An ACID transaction is a collection of operations that has the following basic properties:

Atomicity Either all operations of the transaction succeed or none of the operations happen. When all operations of a transaction succeed, the transaction ends with commit. Otherwise it ends with rollback.

Consistency The execution of all operations of a transaction results in a consistent state, assuming that the state at the beginning of the transaction was consistent.

Isolation A transaction is executed as if no other transaction is executed at the same time. In practice several transactions are executed at the same time, but it appears to each transaction that the other transactions occurred before or after it.

Durability Once a transaction commits all of its effects will survive any system failures.

An ACID transaction is the basic building unit for organising an application into atomic blocks of operations. ACID transactions are bracketed by a *begin work* instruction and a *end work* instruction. Two different *end work* instructions can be called at the end of the transaction: a commit instruction if the transaction has succeeded or a rollback instruction if the transaction aborted.

The major disadvantage of ACID transactions is that it is not possible to commit or abort parts of such transactions. When considering long running transactions, that involve long and complex operations, the ACID behaviour implies that rolling back will be very expensive. Moreover, a long running transaction is more frequently interrupted by failures because of its long execution time.

1.1.2 Compensation

A compensation is the action taken to correct something that has gone wrong. Generally a compensation undoes the action that occurred. But there are actions that are not reversible, as for example the firing of a missile, which can be compensated by destroying the missile before it reaches its target.

The concept of compensation comes from the transactions described in [Gra81], where these have an associated compensation transaction that corrects any errors of an already committed transaction. Garcia-Molina and Salem [GMS87] used the idea of compensation transaction to overcome the problems of long running ACID transactions, by defining the concept of *sagas*. A saga partitions a long running transaction into a sequence of several smaller subtransactions, where each of the subtransactions has an associated compensation. If one of the subtransactions in the sequence aborts, the compensation associated with those subtransactions is executed in reverse order.

1.1.3 BPBeans

The BPBeans framework is described in [CVG01a] and [MRS⁺00]. A BPBean is a Java class¹ that describes a simple action that needs to be performed by a system. Generally, a BPBean will receive a piece of input data, process it, update some stored data, and produce a result.

The BPBeans framework tool allows an application designer to build an application by defining a nested hierarchy of business processes. The top level of the hierarchy describes the major processes of the system, where each process can be decomposed into several subprocesses. The leafs of the hierarchy are basic operations, described by BPBeans. The tool supports several process combinators, as parallel and sequential, that can either communicate synchronously or asynchronously. The BPBeans runtime is then responsible for combining the necessary middleware to support the application. Also, the runtime will control transactions and advanced error recovery, such as compensation, through properties and constructs added to the business process model.

¹More specifically a BPBean is a combination of a JavaBean with some XML code that describes the services required by the JavaBean. The XML code is generated when the BPBean is loaded into the ABC tool. A JavaBean is a Java class that implements a specific interface.

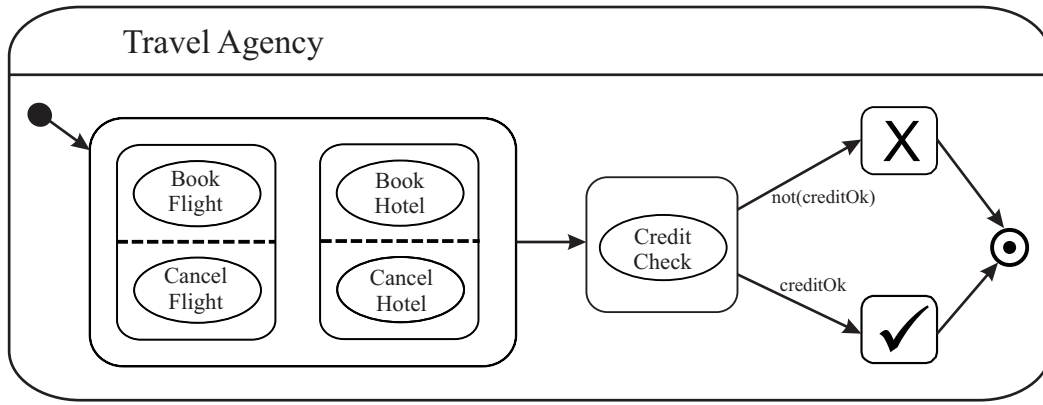


Figure 1.1: BPBeans example

1.1.3.1 ABC Tool

As mentioned before, a BPBeans application is made up of a hierarchy of nested components. At the bottom of this hierarchy are the primitive components, which are referred to as activities. Using the ABC tool, the application designer can connect these primitive components into composite components called processes. Processes may also contain other processes, which is how the hierarchy of components is built up. The ABC tool supplies several process connectors, the more relevant are the following:

Concurrent Where several processes or activities are executed in parallel.

Sequential A sequential process will step through a sequence of tasks according to a predefined order.

Compensation Pair A compensation pair has two elements, a primary task and a compensation task. If the primary task is executed, the compensation task is remembered and can be called later if the primary task needs to be compensated.

We refer to the invocation of compensation activities as *reversal*. If we reach a point where compensation will no longer be required, compensation activities can be forgotten. We refer to this as *acceptance*.

The BPBeans framework uses a graphical representation of the patterns described above. Also, the framework provides for acceptance and reversal of tasks, which

are represented in Figure 1.1. In this figure, the ovals represent activities, while boxes and arrows are used to group and order these. The boxes with the dashed line represent compensation pairs. In this case, *Book Flight* is compensated by *Cancel Flight*. Arrows represent sequencing of activities, so *Book Flight* occurs before *Credit Check*. Processes in the same box not connected by arrows represent concurrent activities, so the compensation pairs with primary processes *Book Flight* and *Book Hotel* take place concurrently. The box with the tick represents transaction acceptance, while the box with the cross represents transaction reversal. The solid circle represents the entry point, while the circle with the dot in the center represents the exit point. A process in BPBeans may consist of several hierarchically structured diagrams.

1.2 Related Work

In this section we briefly discuss notations that could be used in the context of business process modelling. We divided the notations in five categories: formal notations, graphical notations, combined notations, transaction processing models, and specific business process modelling notations.

1.2.1 Formal Notations

Here we will describe some well-known formal notations. We will look at event-based and state-based notations. In a state-based formalism a system is described in terms of state evolutions, while in an event-based formalism a system is described by sequences of events. The first two approaches presented, Hoare's Communicating Sequential Processes (CSP) [Hoa85] and Milner's Calculus for Communicating Systems (CCS) [Mil89], are event-based approaches. In the last subsection we present a state-based approach, B AMN developed by Abrial [Abr96]. From the existing formal languages, we present the ones that are most relevant to this thesis.

1.2.1.1 CSP

CSP is a process algebra where a process communicates with the environment through events. The set of all events that a system can perform is called alphabet. The behaviour of a process is described by an algebraic expression constructed

by atomic events and CSP operators. Atomic events are either basic processes or elements of the process alphabet.

The CSP language includes several basic processes, such as *STOP* that refuses to engage in any event, and *SKIP* that describes successful termination. CSP also provides process constructors for defining structured processes. The event prefixing expression $a \rightarrow P$ represents the process that engages in event a and then behaves as process P . CSP has two different kinds of choice: external and internal. External choice is described by the operator \square : the expression $P \square Q$ describes the process that can behave like either P or Q – this choice is resolved by the environment. The internal choice is described by the operator \sqcap : the expression $P \sqcap Q$ also describes the process that can behave like either P or Q , but in this case the choice is resolved internally by the process.

The parallel composition of processes is represented by $P \parallel Q$. The process $P \parallel Q$ executes P and Q concurrently, and it can only interact by synchronising over common events of both alphabets. The parallel operator can introduce deadlock (where no process can make any progress) if both processes do not have a common next event. A related approach to parallel composition is given by the interleaving operator, written $P \parallel\!\!\parallel Q$. Here, processes P and Q are executed independently of each other.

Processes can be defined recursively. The recursive expression $\mu P \bullet E(P)$ behaves as $E(P)$, where $E(P)$ is a guarded expression containing P . The expression $\mu P \bullet E(P)$ has a unique solution; Hoare [Hoa85] shows that this solution is the weakest fixed point of the function of $P = E(P)$.

The hiding operator is used to hide events from the environment. The expression $P \setminus C$ describes the process that behaves as P with each event in C hidden, where C is a set of events from the alphabet of P .

The most simple semantic model for CSP is the traces model, which describes the process behaviour in terms of sequences of observable events (each sequence of events is called a trace). The traces model is unable to distinguish internal and external choice, and does not model divergence. A divergence is a trace that can

lead to an infinite sequence of internal events. The failures-divergences model is a more complex model that overcomes these limitations of the traces model. In failures-divergences model, a process is modelled by two sets: a set of failures and a set of divergences. A failure is a pair (s, X) , where s is a sequence of events of P , and X is a set of events that P can refuse from that state on. These semantic models can be used to define refinement in terms of inclusion of sets of traces. The *FDR* tool [FDR97] from Formal Systems supports the automated analysis and verification of CSP processes.

1.2.1.2 CCS

CCS is a process algebra similar to CSP. The CCS language has similar constructors to CSP, although it uses different denominations and notations to describe those constructors. The CSP basic processes *SKIP* and *STOP* are represented in CCS as 0 and *NIL*. CCS describes event prefixing as ‘.’, external choice as ‘+’, and parallelism as ‘|’.

The differences between CSP and CCS are not just syntactical, there are significant differences in their interpretation of some operators [But92]. One difference is in the hiding operator, as CCS has a special event τ that represents unobservable behaviour. Hiding an event is just renaming it to τ . Another difference is that CCS does not have an internal choice operator. Internal choice may be expressed in CCS using τ , for example, the internal choice between P and Q can be described in CCS as $\tau.P + \tau.Q$.

The semantics of a CCS process is defined by a set of transition rules based on the structure of the expressions. A process makes a transition by engaging in some event. Each process operator has one or more associated transition rules, and those rules define the meaning of the operator. Milner defines a notion of behavioural equivalence between processes called bisimulation. Bisimulation is a binary relation that is defined by comparing processes transition sequences. In [Mil89] two kinds of bisimulation are presented: a strong bisimulation relation that treats τ as a “visible” event, and a weak bisimulation relation where τ cannot be observed.

MACHINE M
SETS S_s
CONSTANTS C
PROPERTIES P
VARIABLES V
INVARIANT I
INITIALISATION $init$
OPERATIONS
$y \leftarrow op1(x) \hat{=} S$
...
END

Figure 1.2: Abstract machine structure

1.2.1.3 B

B AMN is a model-oriented formal notation that is part of the B-method developed by Abrial [Abr96]. In the B-method, a system is defined as an abstract machine which has the structure presented in Figure 1.2. The sets clause presents user defined sets that can be used in the rest of the machine; those sets can either be enumerated or deferred. The properties are used to define logical properties of the constants or sets of the machine. The variables describe the state of a machine, they are described using set-theoretic constructs, as for example, sets, partial functions, and sequences. The invariant is a set of first-order predicates. The invariant provides typing constraints for the variables of the machine, *e.g.* $basket \in CLIENT \rightarrow \mathcal{P}(BOOK)$, and may also include logical properties that must be preserved by the variables. The initialisation describes the initial values for each variable of the machine. The initialisation must establish the invariant. Operations act on the variables while preserving the invariant and can have input and output parameters. Initialisation and operations are written in the generalised substitution notation of B AMN, which includes constructs such as assignment, guarded statements, and choice. In the assignment statement $x := E$, x is a variable and E is an expression that may use any of the available variables. Simultaneous assignment $x := E \parallel y := F$ is equivalent to $x, y := E, F$. In the guarded statement

SELECT G THEN S END

the guard G is a condition on the state variables and S is an AMN statement. This statement will be enabled only when G holds. The nondeterministic choice between two statements is written

CHOICE S OR T END

that will be enabled when either S or T are enabled. The unbounded choice

ANY x WHERE P THEN S END

nondeterministically chooses some value x satisfying P and then behaves like S . This is a subset of the language, as we only presented the AMN constructs that will be used through this thesis. The B method has two supporting tools, *Atelier-B* [Ate98] from Steria and *B-Toolkit* [Bto96] from B-Core.

1.2.2 Graphical Notations

This section presents two graphical notations, Statecharts and Activity Diagrams, which are part of the Unified Modelling Language (UML) [RJB99]. UML is a object-oriented modelling language that aggregates several graphical notations, each notation can be used to specify different features of the system. We have chosen these notations for their “proximity” to process languages, as they all can represent graphically process operators like concurrency, sequence, and choice.

1.2.2.1 Statecharts

Statechart diagrams are a graphical notation developed by Harel [Har87] for the specification of reactive systems. Statecharts are an extension to the standard state-transition diagrams, including features as hierarchy, concurrency, and communication. A statechart diagram may have several interpretations. To try to overcome this problem, Harel and Naamad [HN96] defined a deterministic semantics for statecharts. However, the notation used to describe data is a low-level programming language without a precise semantics.

A Statechart is composed of states and transitions. Next, we present a brief description of these two main components:

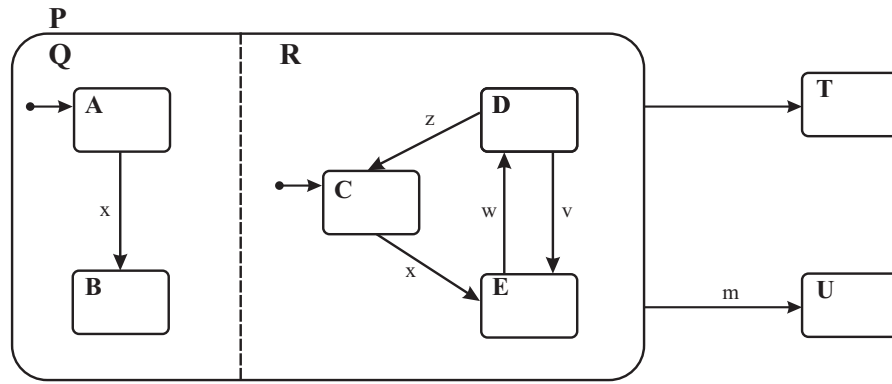


Figure 1.3: Statechart example

States There are three types of states: basic states, and-states, and or-states.

The basic states do not have any substates, they are at the bottom of the state hierarchy. The and-states represent concurrent states: an and-state is composed by several simultaneous substates. The or-states represent sequential states: an or-state is composed by several substates, that will be executed by predefined order.

Transitions A transition represents an evolution in the state of the system: upon the occurrence of an event the system may evolve from a state to another. A transition in Statecharts is composed by five elements: source state, target state, event, action, and condition. The event, action, and condition are said to be the label of the transition. A transition labelled $E[cond]/act$ is triggered by event E when the condition $cond$ holds. Action act is performed when the transition takes place.

Figure 1.3 shows a statechart where state P is composed of two concurrent substates Q and R , represented by a box divided by a dashed line. When entering state P , states A and C become simultaneously enabled. The small horizontal arrows show that A and C are the initial states of Q and R . If event x occurs, A will evolve to B and C will evolve to E , both transitions will occur simultaneously. This means that states Q and R are synchronised over event x . If event w then occurs, a transition from E to D will take place. Event w only affects state R while state Q remains unaltered, in this case states Q and R evolve independently. State Q finishes after the occurrence of x because there are not any outgoing transitions in B . State R may never finish as all its substates have

outgoing transitions. Note that state D has two outgoing transitions for events v and z . In this situation, whichever event occurs first will trigger its associated transition: the occurrence of v causes the state to evolve to E , while the occurrence of z causes the state to evolve to C . There are two transitions exiting state P : the unlabelled transition occurs after both concurrent states Q and R have finished; the labelled transition is triggered by the occurrence of event m and causes process P to terminate immediately – if the concurrent processes Q and R have not finished they will be interrupted.

1.2.2.2 Activity Diagrams

Activity diagrams [FS00] are used to describe the flow of control within a system. They model sequences of activities in a process. Although activity diagrams have similar constructs to statecharts, the two notations are used to specify different abstraction levels of a system. Activity diagrams are intended to specify the overall control flow of a system, while statecharts are intended to specify in detail a specific process of a system.

Activity diagrams consist of activities, transitions between activities, decisions and synchronisations:

Activities Activities describe the execution of a task in a system. Activities have a hierarchical structure, each activity can have a nested activity diagram. Actions are a special kind of activity, they represent tasks that cannot be further decomposed. Actions are atomic, noninterruptible, and instantaneous.

Transitions A transition indicates the evolution in the sequence of activities. When the source activity finishes its tasks, one of its outgoing transitions is triggered automatically.

Decisions A decision describe several alternative paths protected by a boolean guard expression. If more than one guard expression evaluates to true, an activity diagram is nondeterministic. In order to assure that the diagram is deterministic the guard expressions must be mutually exclusive.

Synchronisations Synchronisations model concurrent flows of work. There are two kinds of synchronisations: fork and join. A fork can split a path into

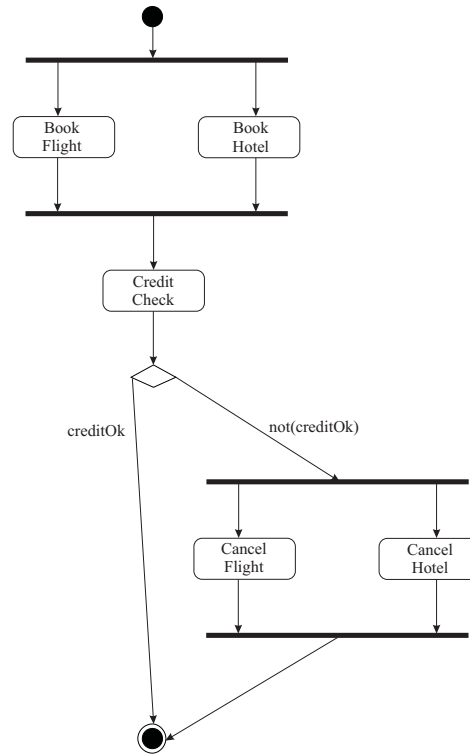


Figure 1.4: Activity diagram example

several concurrent sequences of activities. A join synchronises those concurrent paths initiated by a fork.

Figure 1.4 shows a very simple activity diagram describing the same travel agency example of Figure 1.1. The diagram starts with two concurrent activities, *Book Flight* and *Book Hotel*, where concurrency is represented by the horizontal bar. After both concurrent paths are finished the activity *Credit Check* is executed. After that, the guard *creditOk* is evaluated and depending on the guard value a different path will be executed, which is represented by a diamond where each outgoing arrow describes a alternative path. In our example if *creditOk* is false, the activities *Cancel Flight* and *Cancel Hotel* are executed concurrently.

1.2.3 Combined Notations

In this section we present the csp2B [But00] notation that combines CSP with B. There are several other notations that combine different languages. The csp2B notation will be described in more detail as the approach used to justify the inte-

<p>MACHINE <i>VendingMachine</i></p> <p>ALPHABET <i>Coin Tea Coffee</i></p> <p>PROCESS <i>VM</i> = <i>AwaitCoin</i> WHERE</p> <p style="padding-left: 40px;"><i>AwaitCoin</i> = <i>Coin</i> → <i>DeliverDrink</i></p> <p style="padding-left: 40px;"><i>DeliverDrink</i> = <i>Tea</i> → <i>AwaitCoin</i></p> <p style="padding-left: 80px;">□ <i>Coffee</i> → <i>AwaitCoin</i></p> <p>END</p> <p>END</p>

Figure 1.5: csp2b example

gration of StAC processes with B specifications was based on the csp2b approach of combining CSP and B. Nevertheless, in the end of this section we discuss briefly some different approaches on how to combine notations.

1.2.3.1 csp2B

In [But00] Butler presents the csp2b tool which combines a CSP-like description with B specifications. The B machine is used to describe the system state and operations, while the CSP is used to describe the order in which the operations of a B machine may occur. The tool converts CSP-like specifications into standard B specifications. The resulting B machine can be animated, and appropriate proof obligations can be generated using a B tool.

The csp2B tool supports a process language similar to CSP, including prefixing, choice and the deadlock process *STOP*. Parallel composition and interleaving are supported only at the outermost level. Internal nondeterminism is not supported, but it can be modelled using nondeterministic operations in B.

The tool converts a CSP machine into a B machine containing variables that represent the implicit states of the CSP processes. For each event of the CSP machine, a B operation is created which updates appropriately the variables that represent the implicit state. The CSP machine can be used by itself or to constrain the execution of an existing B machine.

Figure 1.5 shows the CSP machine that describes a vending machine [But00]. This machine has three operations, *Coin*, *Tea*, and *Coffee*. The behaviour of the machine is described by the CSP process *VM*, which can either be on state *AwaitCoin* or on state *DeliverDrink*. *AwaitCoin* is the initial state of process *VM*. In state *AwaitCoin* the only enabled operation is *Coin*, in *DeliverDrink* state both *Tea* and *Coffee* operations may be invoked. The csp2B tool can convert the vending machine CSP specification into a standard B machine, which can be either in state *AwaitCoin* or *DeliverDrink*, and has operations *Coin*, *Tea*, and *Coffee*.

Other Combined Notations

A more general approach from csp2B on how to combine notations is presented in [ZJ93]. Zave and Jackson ([ZJ93] and [ZJ96]) address some problems of composing partial specifications written in different languages. In their approach, heterogeneous specifications are translated into a common specification domain. The composition of partial specifications is just the conjunction of those specifications in the common domain.

UML [RJB99] has unified a collection of software development approaches in order to create a standard notation, applicable to a wide range of applications and domains. This unification is mainly syntactical, although UML also tries to standardize the informal semantics associated with each notation. Furthermore, in UML it is possible to specify different perspectives of a system in a suitable notation. The static structure of a system may be specified using class diagrams, but state machine diagrams are more appropriate to specify the dynamic behaviour of a system. The lack of a formal semantics in any of the UML notations implies that is difficult to verify the consistency between specifications written in different notations.

1.2.4 Transaction Processing Models

The transaction models were introduced to deal with the increasing complexity of the transactions. These models should provide a way of grouping operations within a transaction and at a higher level a way of structuring transactions.

The most elementary transaction model is the ACID model (described in Section 1.1.1), any other model that increases the transactions organization is called an advanced model. Next, we will briefly described the more relevant advanced transaction models that support compensation, such as nested and open nested transactions, compensating transactions, and ConTracts.

Sagas

In the saga construct (already mentioned in Section 1.1.2), transactions are non-hierarchical and purely sequential. Compensation activities are invoked when there is a failure in a system.

Nested and Open Nested Transactions

In nested transactions [Mos82], a transaction is decomposed into a hierarchy of subtransactions. Each subtransaction can either commit or rollback, and the commit will only take effect when its parent transaction (transaction's predecessor in the hierarchy) commits. The rollback of a transaction causes all of its subtransactions to rollback. With the tree structure of nested transactions, invoking a commit or a rollback instruction will only effect its subtransactions and it is dependent on the outcome of its predecessor in the hierarchy. In open nested transactions [WS92], which are a generalisation of nested transactions, subtransactions can commit or abort independently of their predecessor. Considering that a transaction can abort after several of its subtransactions have already committed, open nested transactions require a compensation function for each subtransaction. The compensation function has to semantically undo the effects of committing its corresponding transaction. In both nested and open nested transactions the invocation of rollback is based on system failure.

Compensating Transactions

A formal approach that attempts to overcome the limitations of ACID transactions is presented in [KLS90]. The authors introduce the notion of compensating transactions, which allows access to uncommitted data and undoing of committed transactions. Compensation is formalised in terms of the properties it has to guarantee: a compensating transaction has to reverse the effects of execution of the associated transaction, so that the state of the system after the compensation

must be identical to the state before the execution of the transaction. This notion of compensation is very restrictive, and for real world actions (*e.g.*, firing a missile, sending a letter) is impossible to achieve. This approach does not provide a specification language, the focus is on properties of compensation.

ConTracts

The ConTract model [WR92, RSS97] has a structured approach to compensation. In ConTracts a system is described as a set of steps (actions or operations), which are executed according to a script (control flow description). Each step must have an associated compensation that will be invoked explicitly by the user within a conditional instruction: if the outcome of a step is false, then the associated compensation is executed. In this approach a compensation step has to semantically revert the effects of the associated step, which can be more than just undoing. Although compensations may be non-atomic, each step can only have a single compensation.

1.2.5 Business Process Modelling Languages

A *business process* is defined in [HC01] as “*a collection of activities that takes one or more kinds of input and creates an output that is of value to the customer*”. Typical examples of those activities are making a claim to a insurance company or applying for a mortgage from a bank. A business process can be performed by a combination of people, machines and computer systems.

In this section we present two approaches specifically developed for business process modelling: Architectural Modelling Box for Enterprise Redesign (AMBER) [EJL⁺99], and Role Activity Diagrams (RADs) [Oul95].

1.2.5.1 AMBER

AMBER has a core language containing some basic modelling concepts. The core language can be tailored to specific purposes by a specialisation mechanism. The language has three aspect domains, that can be seen as three different views of the business process being modelled:

Actor domain It describes graphically the entities present in a business process. The actor domain description can be structured, and also can have

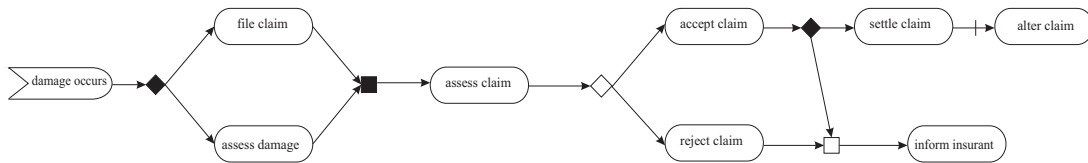


Figure 1.6: AMBER example (behaviour model)

interaction points, indicating physical or logical locations at which the actor can interact with its environment.

Behaviour domain It describes graphically what happens in a business process.

Action is the basic concept in the behaviour domain, modelling an atomic computation. An action can only happen when its enabling condition is satisfied. These conditions are formulated in terms of other actions having occurred. Several behaviours can be grouped in blocks, and also blocks can be nested. Separated blocks can interact, and those interactions are related to the interaction points in the actor domain.

Item domain It models the item (it can be seen as a data structure) on which the behaviour is performed.

The item domain was not yet included in AMBER. Part of AMBER has a formal semantics that can be given by a state automaton. In [JMMS98] the authors describe how to verify a subset of AMBER with SPIN.

Figure 1.6 shows the behaviour model of an insurance company [EJL⁺99]. We do not present here the actor domain nor the item domain, they are described in [EJL⁺99]. The irregular box on the left, *damage occurs*, describes a trigger. The occurrence of an accident triggers two actions: *file claim* and *assess damage*. The black diamond says that the actions are initiated concurrently. The black square that precedes action *assess claim* describes a synchronisation point, the claim will only be assessed after both previous actions have ended. The white diamond describes choice, either *accepted claim* or *reject claim* will be enabled. The white box says that action *inform insurant* can be enabled by either of its previous actions. The cross-arrow disables action *alter claim*, the claim cannot be changed after settlement.

1.2.5.2 Role Activity Diagrams

Role Activity Diagrams (RADs) [Oul95] are a business process modelling notation. RADs describe a business process by using roles, which typically correspond to real-world entities. Different roles can communicate and coordinate through interactions. Next, we describe the main concepts in RADs:

Roles A role involves a set of activities, and it represents a particular responsibility within business process. A role may be a group, a department, or a system. The behaviour of the role is described by a sequence of activities.

State A role has a state, it determines in which point of the activities sequence the role is. When a role executes an activity, the state will evolve to the next state. A role has an initial state and may have a final state. If the final state is also the initial state, the role will iterate.

Activities There are two kinds of activities: action activities and interaction activities. An action is a task that a role executes in isolation. The execution of an action will cause the state to evolve. An interaction is a task that is executed by a set of roles. An interaction can only occur when all roles involved are “ready”, roles synchronise at interaction points. When an interaction is executed all roles involved will move to the next state.

Control Commonly, a role evolves sequentially from one state to the next. In addition to the sequential construct, RADs have constructs to describe choice and concurrency. In a choice there can be several alternative paths, but only one may be chosen. Concurrent paths must join again, denoting that all paths have completed their tasks.

We used the same insurance company example presented in Section 1.2.5.1 to illustrate RADs concepts and notations. Figure 1.7 shows the insurance company specified using RADs. The RAD specification has three roles: *Garage*, *Insurant*, and *Insurance Company*. The trigger for this business process is the occurrence of a car accident by an insurant. This is called an external event and is represented by a solid arrow. The flow of control is represented by the vertical line connecting the activities in a role, and states are represented by ovals. An action activity is represented by a black box, as for example, *assess claim* and *settle claim*. The triangles pointing upwards represent concurrent paths, in our example the

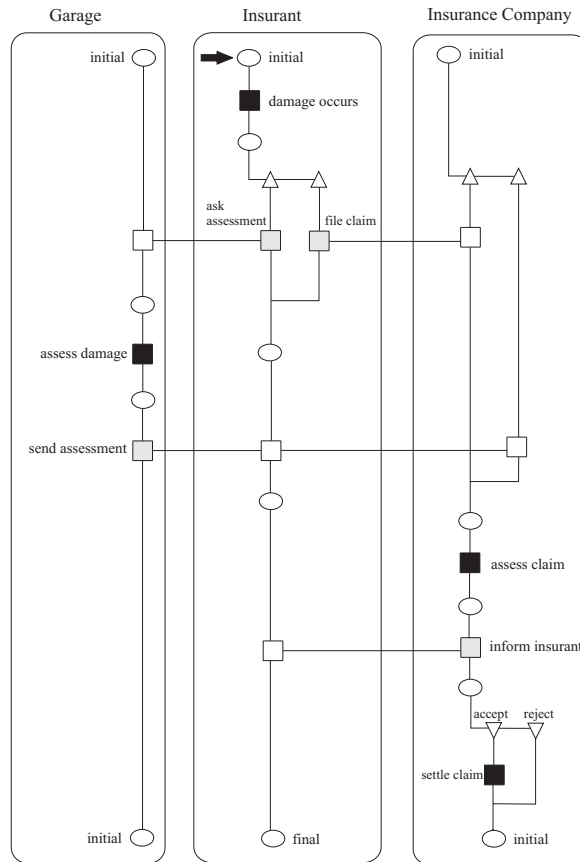


Figure 1.7: RAD example

insurant will execute concurrently the activities *ask assessment* and *file claim*. The interaction activity that initiates an interaction is represented by a shaded box, as for example *file claim*. The other interaction activities are represented by white boxes and do not require a label. Choice is represented by downward pointing triangles, each triangle describes an alternative path. In our example, the triangles labelled *accept* and *reject* are alternative paths.

1.3 Overview

Chapter 2 introduces and gives an overview of StAC, a business process language. Some examples are presented to illustrate different features of the language. The first example, the *raffle*, is a simple example dealing with sequential, parallel and compensation operators. This example will be used to describe how to define a B

machine containing the state and the activities used in the StAC processes. The *e-bookstore* example focuses on compensation scoping. The last one, the *order fulfillment* example, focuses on early termination.

Chapter 3 introduces some extensions to the StAC language leading to StAC_i . In the extended language, a process can have several independent compensation tasks, we called this feature multiple compensation. Within multiple compensation we have defined two compensation mechanisms: selective compensation and alternative compensation. With selective compensation, the reversal invokes the compensation of some activities, leaving the remaining activities unchanged. With alternative compensation, an activity can have several compensations attached and the reversal will choose which of these should be invoked. To illustrate selective and alternative compensation we present two examples: the *travel agency* and *arrange meeting* example. [CVG⁺02] is based on results of Chapter 2 and 3.

In Chapter 4 we define the operational semantics for the StAC_i language. The operational semantics presentation is divided in two parts. The first part excludes the early termination operator, and focus on the remaining StAC_i operators. Restricting the language gives a better understanding of the interpretation of compensation, as it avoids the complexity introduced by early termination. In the second part we present the operational rules for termination, which raises several issues on the “interference” between compensation and early termination. The semantics for StAC is defined in terms of the StAC_i semantics by a translation function, so the interpretation of a StAC process is given in terms of the StAC_i semantics. We formally justify the integration of a StAC process with its associated B machine through an operational approach. [BF00] describes an early version of the operational semantics of StAC excluding early termination.

In Chapter 5 we explore a possible approach to the refinement of StAC specifications. In this approach, the behavioural and compensation information are explicitly embedded in a B machine. The fact that the resulting machine is standard B means that one can apply the B notion of refinement to prove the refinement of StAC specifications. This strategy was applied to the *e-bookstore* example and the *Atelier-B* tool was used for generating and to help prove the

proof obligations.

Chapter 6 proposes some extensions to UML activity diagrams to include representations for compensation and early termination operators. In UML a StAC specification is modelled by a class diagram that describes the data, and a set of activity diagrams that describe the behaviour (including compensation information) of the system. The examples *arrange meeting*, *e-bookstore*, and *order fulfillment* were modelled using the extended UML, which allows the representation of either implicit or multiple compensation tasks.

Chapter 2

The StAC Language

In this chapter we start by presenting StAC syntax, followed by an informal description of the StAC operators. A few examples are introduced in order to illustrate the use of StAC. We will use one of the examples to show how the activities are described in B (Appendix A presents the B machines for all the examples).

2.1 Introduction

We can say informally that in StAC a system is specified as a process, and such process will be decomposed into several sub-processes in a top-down approach. At the bottom level there will only be activities (each activity is an atomic computation), so they cannot be further decomposed. Formally a system is described by a set of equations of the form

$$N = P,$$

where N is a process identifier and P is an expression that can contain several process identifiers, including N , since the equations can have recursion. We have determined, for simplicity reasons, that the first equation describes the overall system being specified. The syntax of StAC is presented in Table 2.1.

The specification of a system is not complete just with the StAC equations, as we might also want to specify the effect of the basic activities on the information structures. Instead of extending StAC to include variables and expressions we

Process	::=	A	(activity label)
		$null$	(null)
		$b \rightarrow P$	(condition)
		$rec(N)$	(recursion)
		$P; Q$	(sequence)
		$P \parallel Q$	(parallel)
		$\parallel_{x \in X(\sigma)} . P_x$	(generalised parallel)
		$P \sqcap Q$	(choice)
		$\sqcap_{x \in X(\sigma)} . P_x$	(generalised choice)
		$\underline{let} \ X = e \ \underline{in} \ P_X$	(let)
		\odot	(early termination)
		$\{P\}$	(termination scoping)
		$P \div Q$	(compensation pair)
		\boxtimes	(reverse)
		\boxdot	(accept)
		$[P]$	(compensation scoping)

Table 2.1: StAC Syntax

used the B notation [Abr96] to specify the effect of activities. With this approach the specification of a system has two components, a set of process equations and a B machine describing the activities. The B machine includes an information state Σ , operations on the state and boolean expressions. In Section 2.3.1 we show how to describe activities in B.

2.2 StAC Operators

The StAC language allows sequential and parallel composition of processes, and the usual process combinators. Besides these, it has specific combinators to deal with compensation. An overview of the language is given in this section.

2.2.1 Basic Operators

Each activity label A (in StAC) has an associated activity \xrightarrow{A} (in B) representing an atomic change in the state: if Σ is the set of all possible states, then \xrightarrow{A} is a relation on Σ .

Note: In the rest of the thesis we consider the capital letters A to D are activities, and the letters P to S are processes.

The process *null* does nothing and completes immediately. This process has a similar interpretation to the CCS inactive agent **0** that does not produce any action.

In the conditional operator, process P is guarded by a boolean function b . This boolean function can consult the state, *i.e.*, $b : \Sigma \rightarrow \mathbf{BOOL}$. Process $b \rightarrow P$ behaves as P if b is evaluated to true. Conversely, if b is false, the conditional process terminates immediately, *i.e.*, process $false \rightarrow P$ behaves as *null*. Notice that the StAC interpretation of the conditional process differs from the interpretation of guarding in B. For example, in B the sequence:

SELECT b THEN P END; T

will deadlock if b is false, while in StAC the process $(b \rightarrow P); T$ will execute T if b is false. This implies that StAC does not model deadlock, while for example B and CSP do.

The recursive operator $rec(N)$ enables the use of a process identifier N of the right-side of an equation to be used in the left-side term of an equation.

2.2.2 Sequential and Parallel Operators

The sequential construct combines two processes, $P;Q$. In process $P;Q$, P is executed first, and only when P terminates Q can be executed.

In parallel process $P \parallel Q$, the execution of the activities of P and Q is interleaved. Generalised parallel extends the parallel operator over a set X , which can either be finite or infinite. For example,

$$\parallel_{x \in \{x_1, x_2, x_3\}} P_x$$

describes three parallel instances of process P , where each instance is indexed by a unique index x belonging to set X . A parallel process completes when all the processes instances complete.

The indexing of generalised processes must be a constant set. Allowing the index to be an expression in the state may create problems, as the activities within the generalised process may alter the state, and consequently alter the indexing expression. An example of this problem is showed in process:

$$P = \parallel_{x \in S}. S := \emptyset; Q(x) \quad (2.1)$$

where $S \in \mathcal{P}(\mathbb{N})$ is a state variable. Because S is assigned a new value, it is unclear how the generalised process should be interpreted.

To overcome this limitation we introduced the let statement that makes explicit at what point the state is being accessed. In the statement let $X = e$ in P_X , the state expression e^1 is assigned to X , so the value of X is fixed before the execution of P_X . After fixing the value of X , P_X represents process P where every occurrence of index X is replaced by the value of e . After rewriting process (2.1) using a let statement, we obtain:

$$P = \text{let } X = S \text{ in } \parallel_{x \in X}. (S := \emptyset; Q(x))$$

The fact that the state variable S is altered within the generalised process does not interfere with index X . Because X was fixed to the value of S at the beginning of the let statement.

Communication Model

StAC does not explicitly model communication between processes. Since activities act on a shared global state, defined by the variables of a B machine, the processes of a parallel composition can communicate indirectly via variables. This will be described in Section 2.3.1.

2.2.3 Choice

The choice $P \sqcap Q$ selects whichever of P or Q is enabled. If both P and Q are enabled, the choice is made by the environment and it is resolved at the first activity. The environment could be a user selecting one of the options in a menu,

¹We are assuming that the outcome of the expression e is a set, *i.e.*, $e \in \Sigma \longrightarrow \mathcal{P}(Y)$, where Y is a set.

for example. Notice that the \parallel operator causes nondeterminism in some cases. If we consider the following example:

$$(A; B) \parallel (A; C)$$

when activity A occurs its not possible to determine which one of the two behaviours $A; B$ or $A; C$ will be chosen. In this case, the choice is made by the system rather than the environment. Generalised choice extends choice over a infinite or finite set of processes. This operator was introduced to avoid having to introduce process parameters in StAC. For example, a process that allows a user to choose a book would be described in StAC as:

$$\parallel_{b \in BOOK} . ChooseBook_b$$

as opposed to having b as parameter, $ChooseBook(b : BOOK)$. In both generalised operators we will use the notation $P(x)$ in place of P_x , as it is a more familiar notation – it resembles process arguments.

2.2.4 Early Termination

A important feature in business processing is the possibility of terminating processes before they have concluded their execution. For this reason we have included in StAC two termination constructs. The early termination \odot that forces a process to terminate, and the termination scoping brackets $\{\dots\}$ that delimit the scope of the early termination. For example, the process

$$\{P; \odot; Q\}; R$$

will first execute P , then the early termination will force the process Q , that is within the braces, to terminate. The overall process will then continue by executing the processes outside the braces. In the case of parallel processes, a termination instruction within one of the parallel process also applies to the other process. For example, in the process

$$\{(P; \odot; Q) \parallel R\} \parallel S$$

the early termination causes R to terminate. The early termination does not cause S to terminate since S is outside the termination scope. Furthermore, it may be the case that R does not terminate immediately on invocation of the early termination but at some later stage. This is because termination of concurrent processes would be implemented by sending messages to the processes instructing them to terminate, and these messages will not be transmitted nor acted upon instantaneously. The informal rules for the early termination are:

- Invocation of an early termination within a sequential process causes that process to terminate immediately.
- Processes within the scope of an early termination that are running concurrently to the early termination may continue to execute for several steps after invocation of the termination instruction before terminating either prematurely or to completion.

2.2.5 Compensation Operators

The next set of operators is related to the compensation concept. The compensation pair $P \div Q$ expresses that P is the primary process (or primary task) and Q is the compensation process (or compensation task). When a compensation pair runs, it runs the primary task, and once the primary process has completed, the compensation process is remembered. The compensation process is constructed in the reverse order to the primary process execution, for example:

$$(A \div A'); (B \div B')$$

behaves as $A; B$ and has the compensation task $B'; A'$. A compensation task can be viewed as a stack where compensation processes are pushed into the top of the stack.

The reverse operator \boxtimes causes the compensation process to be executed. The process

$$(A \div A'); (B \div B'); \boxtimes$$

behaves as $A; B$, and then the \boxtimes operator causes the compensation task to be executed, so the overall behaviour is $(A; B); (B'; A')$ which we write as $A; B; B'; A'$.

The execution of parallel compensation pairs is interleaved, implying that the execution of their compensation task should also be interleaved. The parallel process

$$(A \div A') \parallel (B \div B')$$

executes A and B concurrently and the resulting compensation process is $A' \parallel B'$.

The accept operator \boxtimes indicates that currently remembered compensations should be cleared, meaning that after an accept the compensation task is set to *null*. The process

$$(A \div A'); (B \div B'); \boxtimes; \boxtimes$$

executes A and B , when the \boxtimes operator is called the compensation task B' ; A' has already been cleared by the \boxtimes operator. Next we will consider the combination of compensation with choice. The process

$$(A \div A') \sqcup (B \div B')$$

behaves as either A or B , the choice between A and B is made by the environment. The compensation task in the case that A is chosen is A' and in the other case is B' .

The StAC language permits nested compensation pairs, meaning that compensation can itself be compensated. The following process has two levels of compensation:

$$A \div (B \div C).$$

Initially the above process behaves as A and the compensation task $B \div C$ is remembered as the compensation for A . When the reverse operator is appended to the previous process

$$(A \div (B \div C)); \boxtimes$$

after the execution of activity A the reversal will cause compensation pair $B \div C$ to be executed, by executing B and adding C to the compensation. Activity C can be invoked later by a reversal to compensate for activity B . The nested compensation pair states that A is compensated by B , and B is compensated by C .

2.2.6 Scoping of Compensation

The compensation scoping brackets $[\dots]$ are used to delimit the scope of the acceptance and reversal operators. The start of scope creates a new compensation task, and invoking a reversal instruction within that scope will only execute those compensation activities that have been remembered since the start of the scope. In the process

$$(A \div A'); [(B \div B'); \boxtimes],$$

the overall process would behave as $A; B; B'$. Compensation A' is not invoked because its outside the scope of the reversal instruction. An acceptance instruction, within a scope, will only clear the compensation activities that have been recorded since the start of the scope. For example, the process:

$$(A \div A'); [(B \div B'); \boxtimes]; (C \div C')$$

after A , B and C have been executed, has $C'; A'$ as compensation. Since the acceptance instruction is inside the brackets, it just clears the compensation process B' that is within the brackets. Another feature of the compensation scoping operator is that compensation is remembered if a reversal instruction is not performed, as in the example:

$$(A \div A'); [(B \div B')]; (C \div C').$$

Here, the compensation process is $C'; B'; A'$, which includes the compensation process B' defined inside the brackets. B' is retained because there is no acceptance instruction within the brackets.

2.3 StAC Examples

To illustrate and clarify the applicability of StAC, we present in this section a few examples. The raffle example, is a simple example dealing with sequential, parallel and compensation operators. The e-bookstore example illustrates the use of compensation scoping, and the order fulfillment example illustrates the use of early termination of concurrent processes.

2.3.1 Raffle

In this example, raffle tickets are distributed by several agents. Each agent can sell tickets until a predefined deadline. When that deadline is reached, in order to perform the draw, the total amount of tickets sold by all agents must exceed a defined threshold. In that case, the draw is performed and the prize delivered to the winner. Otherwise, every ticket sold will be refunded by the agent where the ticket was bought.

The top level process is defined as a sequence, first the tickets are distributed and sold by several agents, then the raffle can be drawn or cancelled based on the number of tickets sold.

$$Raffle = SellTickets; DrawOrCancel$$

To illustrate how the StAC processes and its associated B machine interact, we will interleave the presentation of the raffle processes with the state and operations of the *Raffle* machine. The B machines of all the examples are described in Appendix A. We will start by presenting the clauses that define the machine state. The clause SETS presents the sets, *AGENT* and *TICKET*, used in the machine. Set *AGENT* represents all agents that will be selling the raffle tickets, and set *TICKET* represents all tickets available. The constant *threshold* describes the minimum number of tickets that have to be sold in order to perform the draw. The clause VARIABLES names the variables of the abstract machine such as *sold*, *unsold*, and *winner*. In the INVARIANT part we specify the types of the variables introduced in the previous clause. The variables *sold* and *unsold* are functions that for each agent return, respectively, the set of sold and unsold tickets. The variable *winner* is defined as a set that after the draw will contain the winner ticket. The last clause of the invariant states that the set *winner* can either have a single element or be the empty set. We decided to model the variable *winner* as a set because if the draw is cancelled, we can use the empty set to model the fact that there was not a winning ticket. The INITIALISATION describes the initial values of the variables. Both variable *sold* and *unsold* are initialised to the function that assigns to each agent an empty set of tickets.

Variable *winner* is initialised to the empty set.

MACHINE <i>Raffle</i>
SETS
<i>AGENT; TICKET</i>
CONSTANTS
<i>threshold</i>
PROPERTIES
<i>threshold</i> $\in \mathbb{N}$
VARIABLES
<i>sold, unsold, winner</i>
INVARIANT
<i>sold</i> $\in AGENT \rightarrow \mathcal{P}(TICKET) \wedge$
<i>unsold</i> $\in AGENT \rightarrow \mathcal{P}(TICKET) \wedge$
<i>winner</i> $\subseteq TICKET \wedge$
<i>card</i> (<i>winner</i>) ≤ 1
INITIALISATION
<i>sold</i> $:= \lambda agent. (agent \in AGENT \mid \emptyset) \parallel$
<i>unsold</i> $:= \lambda agent. (agent \in AGENT \mid \emptyset) \parallel$
<i>winner</i> $:= \emptyset$

In order to sell the raffle tickets, the tickets must be distributed among the agents. After the tickets have been distributed, process *SellTicketsAgent* describes the tickets being sold by each agent.

$$SellTickets = \mathbf{DistributeTickets}; SellTicketsAgent$$

Notice that some processes are written with a bold font, e.g., **DistributeTickets**, this means that those processes are activity labels, so they cannot be further decomposed and will be specified as B operations in the *Raffle* machine.

In operation **DistributeTickets** the ANY expression selects a function f that assigns to each agent a set of tickets, where all the tickets will be assigned to some agent ($union(ran(f)) = TICKET$), but the same ticket cannot be assigned

to different agents ($\forall a_1, a_2 \in AGENT . a_1 \neq a_2 \Rightarrow f(a_1) \cap f(a_2) = \emptyset$).

DistributeTickets \triangleq

ANY f **WHERE** $f \in AGENT \rightarrow \mathcal{P}(TICKET) \wedge$

$\forall a_1, a_2 \in AGENT . a_1 \neq a_2 \Rightarrow f(a_1) \cap f(a_2) = \emptyset \wedge$

$union(ran(f)) = TICKET$

THEN

$unsold := f$

END

Before presenting *SellTicketsAgent* process, we will introduced an abbreviation for a construct frequently used in StAC processes, instead of

$$P = (Q; P) \parallel A$$

which executes process Q until activity A occurs, we write

$$P = Q \star A$$

that can be described as “ Q until A ”.

In the parallel process *SellTicketsAgent* each agent iteratively sells a single ticket until activity **timeOut** is executed.

$$SellTicketsAgent = \parallel_{a \in AGENT} . SellOneTicket(a) \star \mathbf{timeOut}(a)$$

Operation **timeOut**² is used in process *SellTicketsAgent* to exit its recursive definition, this operation does not need to perform any explicit action.

$$\mathbf{timeOut}(a : AGENT) \triangleq skip$$

Process *SellOneTicket* is a generalised choice over the set of unsold raffle tickets of agent a . The generalised choice describes the selection of a single ticket from all unsold tickets. In the compensation pair that is inside the choice, the primary activity sells a single ticket, with the compensation action being to refund the value of that ticket. Note that parameters a and t are stored along **RefundTicket**

²In all B operations we will use the notation $A(x : X) \triangleq S$ as an abbreviation for $A(x) \triangleq \mathbf{PRE} x : X \mathbf{THEN} S \mathbf{END}$.

in the compensation.

$$\text{SellOneTicket}(a) = \underline{\text{let}} \ U = \text{unsold}(a) \ \underline{\text{in}} \\ \quad \prod_{t \in U} . (\mathbf{SellTicket}(a, t) \div \mathbf{RefundTicket}(a, t))$$

Operation **SellTicket** removes ticket t from the unsold tickets and puts it into the sold tickets of agent a . It is not necessary to verify if ticket t is an unsold ticket of agent a , because this is guaranteed by the let statement. Operation **RefundTicket** is similar, but instead it removes ticket t from the sold tickets of agent a and places t into the unsold tickets of a .

SellTicket($a : \text{AGENT}, t : \text{TICKET}$) $\hat{=}$
BEGIN
 $\text{unsold}(a) := \text{unsold}(a) - \{t\} \parallel$
 $\text{sold}(a) := \text{sold}(a) \cup \{t\}$
END

DrawOrCancel makes the choice of doing or cancelling the draw. If the number of tickets sold exceeds the threshold, the process *PerformDraw* will be executed. Otherwise, the reversal instruction is executed refunding all tickets sold.

$$\text{DrawOrCancel} = (\mathbf{overThreshold} \rightarrow \text{PerformDraw}) \\ \quad \square \\ \quad (\neg \mathbf{overThreshold} \rightarrow \boxtimes)$$

overThreshold is used as a guard of a conditional process, so it will be specified in B as a boolean definition (process guards do not change the machine state).

overThreshold ==
 $\text{card}(\bigcup (\text{agent}).(\text{agent} \in \text{AGENT} \mid \text{sold}(\text{agent}))) \geq \text{threshold}$

PerformDraw starts by doing the draw, followed by delivering the prize to the raffle winner. After delivering the prize, the acceptance instruction is executed, clearing all refunding information.

$$\text{PerformDraw} = \mathbf{Draw}; \mathbf{DeliverPrize}; \boxtimes$$

Operation **Draw** chooses nondeterministically a ticket from all the sold tickets,

and this ticket is placed in the *winner* set.

Draw \triangleq
ANY a, t **WHERE** $a \in AGENT \wedge t \in TICKET \wedge t \in sold(a)$ **THEN**
 $winner := \{t\}$
END

Similarly to operation **TimeOut**, **DeliverPrize** does not perform any action – it is used as an abstraction of delivering the prize to the winner.

DeliverPrize $\triangleq skip$

2.3.2 E-Bookstore

The e-bookstore is a typical example of an e-business and we will use it to illustrate nested compensation. In this example each client defines a limited budget and has an e-basket where the selected books are kept. Every time the client selects a book, the budget is checked to see if it was exceeded, in this case the book is returned to the e-shelf. When the client finishes shopping s/he can either pay or abandon the bookstore, in the later case all selected books have to be returned to the shelf.

The e-bookstore is defined as a finite set of parallel *Client* processes:

$$Bookstore = \parallel_{c \in CLIENT} . Client(c)$$

The set *CLIENT* represents all possible on-line clients. Since all instances of on-line clients are run concurrently, each instance of process *Client* has an independent compensation task:

$$\begin{aligned}
 Client(c) = & \textbf{Arrive}(c); \\
 & ChooseBooks(c); \\
 & (\textbf{Quit}(c); \boxtimes \\
 & \quad \parallel \\
 & \quad Pay(c); \boxtimes); \\
 & \textbf{Exit}(c)
 \end{aligned}$$

The first activity in process *Client* is **Arrive** that creates and initialises the client information, setting the budget to a value determined by the client. The next process is *ChooseBooks*, followed by a choice between paying the books in the basket or abandoning the bookstore without buying any books. If the client chooses to quit, the reverse instruction is invoked causing the return of all books in the client's basket to the shelf. If the client decides to pay for his/her order, the accept instruction is executed and all compensation information is discharged. The last process **Exit** finishes the on-line session with the bookstore, clearing all the information related to that client.

To select books the client iterates over the selection of individual books (*ChooseBook*) until **Checkout** is invoked:

$$ChooseBooks(c) = ChooseBook(c) \star \mathbf{Checkout}(c)$$

By using scoping brackets *ChooseBook* creates a new compensation process for each book selected. This new compensation is only related to the selected book.

$$ChooseBook(c) = \llbracket b \in BOOK . [(\mathbf{AddBook}(c, b) \div \mathbf{ReturnBook}(c, b)); \mathbf{overBudget}(c) \rightarrow \boxtimes] \rrbracket$$

Within *ChooseBook* there is a compensation pair, **AddBook** compensated by **ReturnBook**, and the compensation process is only executed if adding that book to the basket exceeds the budget. In this case executing the compensation task implies returning the book that has just been added to the basket, rather than all books in the basket. If the budget is not exceeded, the compensation is preserved. In process *Pay* the client's card is processed and if the card is rejected, the compensation is executed returning all selected books to the shelf:

$$Pay(c) = \mathbf{ProcessCard}(c); \neg \mathbf{accepted}(c) \rightarrow \boxtimes$$

2.3.3 Order Fulfillment

To illustrate the use of early termination and compensation, we use an order fulfilment example described in [CVG01a] and [CVG⁺02]. ACME Ltd distributes goods which have a relatively high value to its customers. When the company receives an order from a customer, the first step is to verify whether the stock

is available. If not available the customer is informed that his/her order can not be accepted. Otherwise, the warehouse starts preparing the order for shipment, and a courier is booked to deliver the goods to the customer. Simultaneously with the warehouse preparing the order, the company does a credit check on the customer to verify that the customer can pay for the order. The credit check is performed in parallel because it normally succeeds, and in this normal case the company does not wish to delay the order unnecessarily. If the credit check fails the preparation of the order is stopped.

At the top level the application is defined as a sequence as follows:

$$\begin{aligned}
 ACME &= \text{AcceptOrder} \div \text{RestockOrder}; \\
 &\quad \text{FulfillOrder}; \\
 &\quad \text{okFulfillOrder} \rightarrow \boxtimes \\
 &\quad \square \\
 &\quad \neg \text{okFulfillOrder} \rightarrow \boxtimes
 \end{aligned}$$

The first step in the *ACME* process is a compensation pair. The primary action of this pair is to accept the order and deduct the order quantity from the inventory database. The compensation action is simply to add the order quantity back to the total in the inventory database. Following the compensation pair, the *FulfillOrder* process is invoked. Finally if the order has been fulfilled correctly, the order is accepted, otherwise the order is reversed.

The order is fulfilled by packaging the order at the warehouse while concurrently doing a credit check on the customer. If the credit check fails, the *FulfillOrder* process is terminated:

$$\begin{aligned}
 FulfillOrder &= \{ \text{WarehousePackaging} \parallel \\
 &\quad (\text{CreditCheck}; \neg \text{okCreditCheck} \rightarrow \odot) \}
 \end{aligned}$$

Notice the termination scope includes the *WarehousePackaging* process so that a failed credit check results in a termination instruction being sent to that process. This will cause *WarehousePackaging* to terminate eventually, possible before all the items in the order have been packed.

The *WarehousePackaging* process consists of a compensation pair in parallel with the *PackOrder* process:

$$\textit{WarehousePackaging} = (\mathbf{BookCourier} \div \mathbf{CancelCourier}) \parallel \textit{PackOrder}$$

The compensation pair books the courier, with the compensation action being to cancel the courier booking. **CancelCourier** results in a second message being sent to the courier rather than reversing the send of the message which booked the courier. The *PackOrder* process packs each of the items in the order in parallel. Each **PackItem** activity is reversed by a corresponding **UnpackItem**:

$$\textit{PackOrder} = \underline{\textit{let}} \ O = \textit{order} \ \underline{\textit{in}} \ \parallel_{i \in O} . (\mathbf{PackItem}(i) \div \mathbf{UnpackItem}(i))$$

In the case that a credit check fails, the *FulfillOrder* process terminates with the courier possibly having been booked and possibly some of the items having being packed. The reversal instruction will then be invoked and will result in the appropriate compensation activity being invoked for those activities that did take place.

Chapter 3

Extending StAC with Multiple Compensation

In this section we present some extensions to the StAC language. The most important of these extensions is that a process can have several simultaneous compensation tasks associated with it. A process decides which task to attach the compensation activities to, and each individual compensation task can be reversed or accepted. This contrasts with the language presented in Chapter 2, where scoping of compensation is hierarchical and each scope has a single implicit compensation task. To distinguish different compensation tasks, the operators that deal with compensation, *i.e.*, compensation pair, acceptance and reversal, are indexed by the compensation task index to which they apply. The syntax of StAC_i is presented in table 3.1.

The motivation for extending the StAC language was that StAC_i had a clear semantics for compensation and made it easier to describe parallel compensation. It is easier to define the operational semantics of multiple compensation tasks, than a hierarchy of compensation scopes, because with multiple compensation it is possible to refer directly to a compensation task by its index, while with nested compensations this is not possible. Later, when applying StAC to some case studies it emerged that some features that were difficult to model in StAC could be easily modelled in StAC_i using multiple compensation tasks. This suggests that multiple compensation is a useful concept.

Utilising the facility of multiple interleaved compensation tasks, we introduce

Process	::=	A	(activity label)
		0	(null)
		$b \rightarrow P$	(condition)
		$rec(N)$	(recursion)
		$P; Q$	(sequence)
		$P \parallel Q$	(parallel)
		$\parallel_{x \in X}. P_x$	(generalised parallel)
		$P \sqcap Q$	(choice)
		$\sqcap_{x \in X}. P_x$	(generalised choice)
		$\underline{let} \ X = e \ \underline{in} \ P_X$	(let)
		\odot	(early termination)
		$\{P\}$	(termination scoping)
		$P \div_i Q$	(indexed compensation pair)
		\boxtimes_i	(indexed reverse)
		\boxdot_i	(indexed accept)
		$J \triangleright i$	(merge)

 Table 3.1: StAC_i Syntax

the mechanisms of multiple compensation: selective compensation and alternative compensation. With selective compensation, the reversal selects some activities to be compensated, while preserving the compensations for other activities. With alternative compensation, several alternative compensation tasks may be attached to an activity and the reversal selects one of these alternatives for invocation. We illustrate selective compensation through a travel agency example, and multiple compensation through a meeting scheduling example.

In the next section we describe informally the StAC_i operators that deal with multiple compensation. In the following sections we present the travel agency and the arrange meeting examples.

3.1 Extended Compensation Operators

Most of the StAC_i operators are retained from StAC without any alterations. The new operators are operators that deal with compensation (written with bold font in table 3.1), and reflect the extensions to the StAC language. In the extended language, process $P \div_i Q$ has P as its primary process and, when P completes, compensation Q is remembered on compensation task i , where i is an index.

Note that compensation indices are constants and not expressions that can be evaluated. The instruction to accept compensation task i is given by \boxplus_i , while the instruction to reverse compensation task i is given by \boxminus_i . To help illustrate indexed compensation, consider the process:

$$(A \div_1 A'); (B \div_2 B'); \boxminus_1; (C \div_2 C'); \boxminus_2.$$

This process will start by invoking A , followed by B and then the reversal causes compensation A' to be invoked. Compensation B' will not be invoked at this stage as it is on compensation task 2 and only compensation task 1 is invoked by the first reversal operator. After the first reversal, activity C is performed. Reversal is then invoked on compensation task 2 which causes C' followed by B' to be executed.

Informally it can be said that the compensation information of a process is maintained by a compensation function that for each compensation task index, returns the associated compensation process. When the primary task of a compensation pair concludes its execution, the compensation task is composed in sequence with the original compensation process for that task.

An important operator in StAC_i is the merge operator. The expression $J \triangleright i$, where J is a set of indices, merges all compensation tasks belonging to J into the compensation task i . When merging compensation tasks, those tasks are merged in parallel. In the process

$$(A \div_1 A'); (B \div_2 B'); \{1, 2\} \triangleright 3$$

the merge operator will compose in parallel the compensation task 1 (A') with compensation task 2 (B'), and add parallel process $A' \parallel B'$ to compensation task 3. Since the compensation task 3 is empty, the resulting compensation task 3 is just $A' \parallel B'$.

Consider the following process that uses three individual compensation tasks:

$$(A \div_1 A'); (B \div_2 B'); (C \div_3 C'); \{1, 2\} \triangleright 3.$$

Initially it executes A , B and C and then merges compensation tasks 1 and 2 into compensation task 3. Joining compensation tasks 1 and 2 results in the parallel process $A' \parallel B'$, that will be put in front of the compensation task 3, giving $(A' \parallel B'); C'$ as the resulting compensation for task 3.

The compensation scoping brackets $[\dots]$ do not apply to the indexed compensation operators. The scoping brackets can be described by indexed compensation tasks. For example, the StAC process

$$(A \div A'); [(B \div B'); \boxtimes; (C \div C')]$$

can be represented in StAC_i as

$$(A \div_1 A'); (B \div_2 B'); \boxtimes_2; (C \div_2 C'); \{2\} \triangleright 1$$

where the braces are represented as a “new” compensation task. When the reversal instruction is invoked on compensation task 2 it will only execute B' . Compensation process A' that in StAC was outside the braces, in StAC_i is in a different compensation task, and does not get invoked. The merge is used to preserve any compensations not reversed within the scoping brackets. In this example, at the end of the scoping brackets, the compensation C' has to be preserved. This is done by merging compensation task 2 into task 1. In Section 4.6 we present a complete translation of StAC to StAC_i terms.

3.2 Selective Compensation: Travel Agency Example

The travel agency [LR00] is a company that offers on-line trip reservation services to its clients. A client can compose an itinerary with several flights, car rentals, and hotel reservations. It is necessary to verify the client credit card's details before making the reservations. If the credit card is accepted, the client is asked to decide whether s/he wants to reserve his/her itinerary or to quit the reservation. Once the client's order has been confirmed, the reservations for the flights, car rentals, and hotels are made. Since these reservations are independent they are made in parallel to speed up the overall process. If all the

reservations in the client's itinerary are successful, the final itinerary is sent to client, and this concludes the trip reservation process. Otherwise, if any of the reservations failed, the client is contacted and given the choice of selecting an alternative itinerary to substitute the failed reservation or aborting the reservation.

The travel agency handles trip reservations for several clients concurrently. A client can request a trip reservation from the travel agency on-line site:

$$TravelAgency = \parallel_{c \in CLIENT} . \mathbf{Request}(c); TripReservation(c)$$

A trip is arranged by getting an itinerary, followed by verifying the client's credit card, and depending on whether the card is accepted or rejected the reservation is continued or abandoned:

$$\begin{aligned} TripReservation(c) = & \quad GetItinerary(c); \\ & \quad \mathbf{VerifyCreditCard}(c); \\ & \quad \mathbf{accepted}(c) \rightarrow ContinueReservation(c) \\ & \quad \square \\ & \quad \neg\mathbf{accepted}(c) \rightarrow QuitReservation(c) \end{aligned}$$

Getting an itinerary involves continually iterating over offering the client the choice of selecting from a flight, a car or a hotel until **EndSelection** is invoked. The *SelectFlight* process selects a single flight using the **SelFlight** activity. The car and hotel selection is defined similarly.

$$\begin{aligned} GetItinerary(c) = & \quad (SelectFlight(c) \square SelectCar(c) \square SelectHotel(c)) \\ & \quad \star \mathbf{EndSelection}(c) \\ SelectFlight(c) = & \quad \parallel_{f \in FLIGHT} . \mathbf{SelFlight}(c, f) \\ SelectCar(c) = & \quad \parallel_{a \in CAR} . \mathbf{SelCar}(c, a) \\ SelectHotel(c) = & \quad \parallel_{h \in HOTEL} . \mathbf{SelHotel}(c, h) \end{aligned}$$

In process *ContinueReservation* the client is asked to decide whether to reserve the itinerary or to abandon the reservation. If the client decides to reserve the

itinerary, the reservations are made:

$$\begin{aligned} \text{ContinueReservation}(c) = & \text{ConfirmOrder}(c); \text{MakeReservation}(c) \\ & \parallel \\ & \text{CancelOrder}(c); \text{QuitReservation}(c) \end{aligned}$$

In *MakeReservation* the flight, car and hotel reservations are made concurrently. If any of the reservations failed, the client is then contacted, otherwise the process ends:

$$\begin{aligned} \text{MakeReservation}(c) = & \\ & (\text{FlightReservations}(c) \parallel \text{CarReservations}(c) \parallel \text{HotelReservations}(c)); \\ & \neg \text{okReservations}(c) \rightarrow \text{ContactClient}(c) \\ & \parallel \\ & \text{okReservations}(c) \rightarrow \text{EndTrip}(c) \end{aligned}$$

The *FlightReservations* process reserves a single flight using the **ReserveFlight** activity. The travel agency uses two compensation tasks: compensation task *S*, representing compensation for reservations that have been booked successfully, and compensation task *F*, representing compensation for reservations that have failed. The choice between which task to add the compensation to is determined by the outcome of the **ReserveFlight** activity.

Since we use two compensation tasks, instead of having a compensation pair we have a compensation triple, with a primary process *P* and two compensations *Q*₁ and *Q*₂. We model this triple with a construction of the form:

$$P; (c \rightarrow (\text{null} \div_1 Q_1)) \parallel (\neg c \rightarrow (\text{null} \div_2 Q_2))$$

If *P* makes *c* true, this is equivalent to $P \div_1 Q_1$ with *Q*₁ being added to compensation task 1. If *P* makes *c* false, this is equivalent to $P \div_2 Q_2$ with *Q*₂ being added to compensation task 2. With this construction it is possible to organize the compensation information into several compensations tasks, where each one of those tasks can later be reversed or accepted independently.

All the flights reservations are made concurrently. The flight reservation and

its associated compensations is defined as follows:

$$\begin{aligned}
 \text{FlightReservations}(c) &= \underline{\text{let}} \ R = \text{flights}(c) \ \underline{\text{in}} \\
 &\quad \parallel f \in R . \text{FlightReservation}(c, f) \\
 \text{FlightReservation}(c, f) &= \\
 &\quad \mathbf{ReserveFlight}(c, f); \\
 &\quad \mathbf{flightIsReserved}(c, f) \rightarrow (\text{null} \div_{S(c)} (\mathbf{CancelFlight}(c, f) \parallel \mathbf{ClearFlight}(c, f))) \\
 &\quad \parallel \\
 &\quad \neg \mathbf{flightIsReserved}(c, f) \rightarrow (\text{null} \div_{F(c)} \mathbf{ClearFlight}(c, f))
 \end{aligned}$$

The **ClearFlight** activity removes flight f from the client's itinerary, while the **CancelFlight** activity cancels the reservation of flight f with the airline. If the activity **ReserveFlight** is successful, then to compensate it one has to cancel the reservation with the airline and also remove that flight from the client's itinerary. Otherwise, if the flight reservation fails its only necessary to remove the flight from the client's itinerary in order to compensate, it is not necessary to cancel the flight reservation. The car and hotel reservations are defined similarly and are omitted here.

The *ContactClient* process is called if some reservations failed. In this process the client is offered the choice between continuing or quitting:

$$\begin{aligned}
 \text{ContactClient}(c) &= \mathbf{Continue}(c); \boxtimes_{F(c)}; \text{GetItinerary}(c); \text{MakeReservation}(c) \\
 &\quad \parallel \\
 &\quad \mathbf{Quit}(c); \text{QuitReservation}(c)
 \end{aligned}$$

In the case that the client decides to continue, reverse is invoked on compensation task $F(c)$, the failed reservations. This has the effect of removing all failed reservations from the client's itinerary. Compensation task S is preserved as the successful reservations may need to be compensated at a later stage. The client continues by adding more items to the itinerary, which are then reserved. In the case that the client decides to quit, reversal is invoked on both compensation threads. This has the effect of removing all reservations from the client's itinerary and cancelling all successful reservations.

$$\text{QuitReservation}(c) = (\boxtimes_{S(c)} \parallel \boxtimes_{F(c)}); \mathbf{RemoveClient}(c)$$

Finally, a successful trip reservation is ended by accepting both compensation tasks:

$$EndTrip(c) = \Box_{S(c)} \parallel \Box_{F(c)}$$

In [LR00] the authors describe the same travel agency example using a method (spheres of compensation) that does not support multiple compensation, but the resulting model has a different behavior. It is not possible to model the cancellation of part of the itinerary, while maintaining the compensation information for successful bookings with a single compensation task. This happens because the aborting instruction (a instruction similar to the reversal) causes the cancellation of the entire itinerary, so the client will have to choose the complete itinerary all over again.

In general, by selective compensation, we mean that some compensation tasks can be reversed back selectively, while the remaining compensations are maintained. We have modelled the selection criteria in the travel agency by using two compensation tasks and deciding immediately when the primary process is completed to which of these tasks to add the compensation. We then invoke the compensations selectively by picking the appropriate compensation task.

An important feature of selective compensation is that those compensations that are not selected for reversing are preserved. This feature makes it difficult to model selective compensation in StAC (of Chapter 2) that does not support interleaved compensation tasks.

3.3 Alternative Compensation: Arrange Meeting Example

In this example, the goal is to select a date for a meeting for which everyone in the team is available. Initially a set of possible dates is proposed based on the availability of the meeting room. Every member of the team suggests possible dates from the initially proposed set of dates. If an agreement is reached between team members, the meeting is scheduled, otherwise it will be cancelled.

The top level process is defined as a sequence of three processes. First, a set

of possible dates on which the room is available is selected. Next, the team chooses possible dates for the meeting. Last, a date is selected for the meeting and then the meeting is scheduled.

$$\textit{ArrangeMeeting} = \textit{CheckRoom}; \textit{CheckTeam}; \textit{Decide}$$

In this example compensation is used in a novel way. Instead of the usual use of compensation when there is a failure or change of plan, here compensation is used to perform a positive task. The arrange meeting application uses two compensation tasks: *CF* and *CL*. Compensation task *CF* represents activities that need to be confirmed, like the booking of the room or the date for the meeting, while compensation task *CL* represents activities that need to be cancelled.

Process *CheckRoom* has a compensation pair within another compensation pair. In practice, it means that the date selection has two compensation actions: compensation **ConfirmRoom** in task *CF* and compensation **CancelRoom** in task *CL*.

$$\textit{CheckRoom} = (\textit{SelectAvailableDates} \div_{CF} \mathbf{ConfirmRoom}) \div_{CL} \mathbf{CancelRoom}$$

The **SelectAvailableDates** activity chooses a set of dates where the meeting room is available and temporarily books the room for those dates. The compensation activity **ConfirmRoom** will confirm the booking of a single date for the room and remove all the remaining dates. The compensation activity **CancelRoom** will remove all the dates temporarily booked.

In process *CheckTeam* each member of the team concurrently suggests several dates for the meeting:

$$\textit{CheckTeam} = \parallel_{t \in \textit{TEAM}} . (\mathbf{SuggestDates}(t) \div_{CF} \mathbf{ConfirmDate}(t)) \div_{CL} \mathbf{CancelDates}(t)$$

In the **SuggestDates** activity, the team member chooses from the possible dates his/her available dates for the meeting, and those dates will be inserted in the member's diary. The compensation activity **ConfirmDate** confirms the final date for the meeting and removes the remaining dates from the diary. The compensation **CancelDates** cancels all dates for the meeting in the diary.

The process *Decide* verifies if there is a date where all team members are available. If there is an agreement on the date of the meeting, the booking of the meeting is confirmed, otherwise the meeting is cancelled:

$$\begin{aligned} \textit{Decide} = & \neg\textbf{emptyDates} \rightarrow \textbf{SelectDate}; \boxtimes_{CF} \\ & \square \\ & \textbf{emptyDates} \rightarrow \boxtimes_{CL}; \boxtimes_{CF} \end{aligned}$$

When an agreement is not reached (**emptyDates** is true), the meeting has to be cancelled. This is achieved by reversing the compensation task *CL* and accepting compensation task *CF*. The reversal of compensation task *CL* will execute the cancellations, removing the temporary bookings of the meeting room, and clearing the suggested dates from the team member's diary. When the members of the team reach an agreement about the date for meeting (**emptyDates** is false), reversal is invoked on compensation task *CF*. The reversal on *CF* will execute the confirmations, confirming the booking of the room and the meeting date on each member diary. Notice that no action is done on compensation task *CL*, therefore the cancellations stored in *CL* are retained. Keeping the cancellations might be useful later on, if the meeting has to be cancelled.

Alternative compensation allows the reversal of some processes while retaining the others. This could not be achieved using a single compensation task because the reversal would cause the execution of all stored processes. In the arrange meeting example we used multiple compensation tasks to achieve a clear separation between the confirmation and cancellation compensation tasks. This separation allows to reverse the confirmations while retaining the cancellations, and use those cancellations further on.

The distinctive feature in alternative compensation is that activities can have several alternative compensation activities remembered for them simultaneously. Later a decision is made about which of the compensations attached to an activity should be invoked, or even more than one compensation could be invoked.

In this example, the compensation mechanism is used to perform a positive task and not just a compensation task. All confirmations are performed by invoking the reversal instruction on the compensation task *CF*. In this case, reversal is not

invoked with the intention of correcting some failure, but to perform a positive task.

Chapter 4

Semantics

This chapter starts by justifying the use of an operational approach to formalise the semantics of StAC_i . The operational semantics will be presented in two phases: the first phase (Section 4.2) shows the operational rules for StAC without considering early termination operators, and focusing on the interpretation of the compensation; the second phase (Section 4.3) shows the rules for early termination. The complete StAC_i operational semantics is obtained by merging the rules presented in those two phases. Section 4.4 discusses possible evolutions of some StAC_i examples, where each example concentrates on a complex combination of StAC_i operators, as for example, early termination and compensation. Section 4.5 presents an animator for the StAC_i language, which encodes the operational rules into Prolog predicates. In Section 4.6 we define a translation from StAC to StAC_i , so the interpretation of a StAC process is given in terms of StAC_i by a translation function. Section 4.7 formally justifies the integration of a StAC process with a B Machine containing the description its state and activities. In Section 4.8 we discuss the use of other semantics models to formalise StAC_i , and compare those models with the operational approach we have use.

4.1 Introduction

There are several different techniques that could be used to formalised the semantics of StAC_i . The most important are the operational, denotational, axiomatic and algebraic semantics. These different semantics complement each other, showing a different perspective of a language. But depending on the purpose of the semantics and the language features, one technique can be more appropriate than

another.

We decided to describe StAC_i using an operational approach. We have several arguments in favour of an operational approach. First, an operational approach is a more natural approach as our informal description of StAC_i is quite operational. Second, StAC should be able to provide precise answers to business processes scenarios, and considering that the operational semantics gives an abstract implementation for the language, those answers (besides being precise) detail the steps taken to reach that answer. Third, the integration of StAC_i with B can be easily justified through an operational approach using the results presented in [But00]. Last, it is easy to build an animator from the operational rules, and the animator can help validating specifications.

At the moment, our approach to the refinement of StAC_i specifications uses the B refinement relation. To verify if a StAC_i specification R is a refinement of StAC_i specification S we perform two steps. First, for each system specification we combine the two parts of the specification in a standard B machine. Those two parts are: the StAC_i processes that describes the execution order of the operations and compensation information; the B machine that describes the state of the system and its activities. Second, we verify if R is a refinement of S , by using the B notion of system refinement. Since both machines are standard B machines we can use a B tool to generate the appropriate proof obligations. This is outlined in Chapter 5.

4.2 Operational Semantics for Compensation

This section presents the operational semantics for the StAC_i operators excluding early termination. By excluding early termination in this first part of the operational semantics we are avoiding the complexity introduced by its operators and focusing on the formalisation of compensation. Although we are not considering termination, the results obtained through this section are still valid for the complete StAC_i language. In practice we are just dividing the presentation of the semantics in two separated parts that can be combined to form the overall StAC_i semantics.

Plotkin [Plo81] describes how to use transition systems to define an operational semantics; here a system is defined in terms of transitions rules between configurations. For the operational semantics of StAC_i configuration is a tuple:

$$(P, C, \sigma) \in Process \times (I \rightarrow Process) \times \Sigma$$

In the above tuple, C is a function that for each task i returns the compensation process $C(i)$. Σ represents the state of the B machine and it is necessary since an activity may change the state variables. The labelled transition

$$(P, C, \sigma) \xrightarrow{A} (P', C, \sigma')$$

denotes that the execution of a basic activity A may cause a configuration transition from (P, C, σ) to (P', C, σ') . Notice that the execution of an activity does not alter the compensation function, only the operators compensation pair, merge, accept, and reverse may alter it.

4.2.1 Normalisation

This section presents a set of normalisation rules to simplify process expressions that describe the same object. Essentially, the normalisation is introduced to deal with terminated processes¹. Executing process $null; P$ is the same as executing just process P . If the first process in the sequence has already terminated, then the second process can be executed immediately.

The normalisation function has two arguments, the process to be normalised and the current state, and it will return a normalised process. The current state σ is necessary to normalise conditional processes. This section describes the normalisation for processes that do not include early termination, the rest of the normalisation function will be presented in the section dedicated to termination.

The first set of normalisation rules are related to the composition of process $null$ with the sequential, choice and parallel operators. For example, rule N2

¹CSP uses the event \checkmark to deal with termination: a process has terminated when it produces \checkmark .

says that process $null \parallel P$ can be simplified to the normalisation of process P .

$$\begin{aligned}
\text{N1 } norm(null; P, \sigma) &= norm(P, \sigma) \\
\text{N2 } norm(null \parallel P, \sigma) &= norm(P, \sigma) \\
\text{N3 } norm(P \parallel null, \sigma) &= norm(P, \sigma) \\
\text{N4 } norm(null \sqcap P, \sigma) &= norm(P, \sigma) \\
\text{N5 } norm(P \sqcap null, \sigma) &= norm(P, \sigma)
\end{aligned}$$

An implication of N4 and N5 is the following process equality:

$$(P \sqcap null); Q = P; Q$$

which differs from the CSP interpretation of the *skip* operator [Ros98]:

$$(P \sqcap skip); Q = Q \sqcap (P \sqcap skip); Q$$

that can nondeterministically offer Q and ignore P . StAC *null* operator has a similar interpretation to the CCS inactive agent $\mathbf{0}$, they both represent not performing any action, while CSP uses *skip* to represent successful termination.

The following two rules are related to the conditional operator. Rule N6 says that, if the boolean function b is evaluated to *true* in the current state σ , then the conditional process can be simplified to process $norm(P, \sigma)$. Otherwise, if the boolean function b is evaluated to *false*, the conditional process can be simplified to *null*.

$$\begin{aligned}
\text{N6 } norm(b \rightarrow P, \sigma) &= norm(P, \sigma), \text{ if } b(\sigma) = \textit{true} \\
\text{N7 } norm(b \rightarrow P, \sigma) &= null, \text{ if } b(\sigma) = \textit{false}
\end{aligned}$$

Rule N8 and N9 normalise generalised operators. Both rules state that if X is an empty set, the generalised process can be normalised to *null*. The last two rules normalise generalised processes that have *null* as its internal process.

$$\begin{aligned}
\text{N8 } norm(\parallel_{x \in \emptyset} . P(x), \sigma) &= null \\
\text{N9 } norm(\sqcap_{x \in \emptyset} . P(x), \sigma) &= null \\
\text{N10 } norm(\parallel_{x \in X} . null, \sigma) &= null \\
\text{N11 } norm(\sqcap_{x \in X} . null, \sigma) &= null
\end{aligned}$$

The next set of rules show processes that are already in a normalised form.

$$\begin{aligned}
\text{N12 } \text{norm}(A, \sigma) &= A \\
\text{N13 } \text{norm}(\text{null}, \sigma) &= \text{null} \\
\text{N14 } \text{norm}(\boxtimes_i, \sigma) &= \boxtimes_i \\
\text{N15 } \text{norm}(\boxdot_i, \sigma) &= \boxdot_i \\
\text{N16 } \text{norm}(J \triangleright i, \sigma) &= J \triangleright i \\
\text{N17 } \text{norm}(\text{rec}(N), \sigma) &= \text{rec}(N)
\end{aligned}$$

The following set of rules show how to normalise composite constructs, where the previous rules could not be applied, *e.g.*, for N19 we assume $P \neq \text{null}$ and $Q \neq \text{null}$, while N20 requires $P \neq \text{null}$ and $X \neq \emptyset$. The normalisation rules are recursively defined on the components of the operator.

$$\begin{aligned}
\text{N18 } \text{norm}(P; Q, \sigma) &= \text{norm}(P, \sigma); \text{norm}(Q, \sigma) \\
\text{N19 } \text{norm}(P \parallel Q, \sigma) &= \text{norm}(P, \sigma) \parallel \text{norm}(Q, \sigma) \\
\text{N20 } \text{norm}(\parallel_{x \in X} P(x), \sigma) &= \parallel_{x \in X} \text{norm}(P(x), \sigma) \\
\text{N21 } \text{norm}(P \sqcap Q, \sigma) &= \text{norm}(P, \sigma) \sqcap \text{norm}(Q, \sigma) \\
\text{N22 } \text{norm}(\sqcap_{x \in X} P(x), \sigma) &= \sqcap_{x \in X} \text{norm}(P(x), \sigma) \\
\text{N23 } \text{norm}(P \div_i Q, \sigma) &= \text{norm}(P, \sigma) \div_i Q
\end{aligned}$$

Rule N24 states that in the normalisation of a let expression all occurrences of X are replaced by the evaluation of e in the current state σ .

$$\text{N24 } \text{norm}(\text{let } X = e \text{ in } P_X, \sigma) = \text{norm}(P_{e(\sigma)}, \sigma)$$

The rules presented in this section are not sufficient to normalise a process. When applying the normalisation function norm to the example below it is obvious that the resulting process is not yet normalised.

$$\text{norm}((\text{false} \rightarrow A); B, \sigma) = \text{null}; B$$

To overcome this problem we defined function *normalisation* that recursively applies the function norm until a process is normalised.

$$\text{normalisation}(P, \sigma) = \begin{cases} P & \text{if } \text{normalised}(P, \sigma) \\ \text{normalisation}(\text{norm}(P, \sigma), \sigma) & \text{otherwise} \end{cases}$$

The boolean function *normalised* verifies if a process is in its normal form or not. The structure of this function is similar to the *norm* function and its not presented here (function *normalised* is presented in Appendix B).

4.2.2 Operational Rules

In the beginning of Section 4.2 we said that the labelled transition $(P, C, \sigma) \xrightarrow{A} (P', C, \sigma')$ represents a configuration transition caused by activity A . But in fact, StAC operational rules will instead be transitions from normalised processes, *i.e.*,

$$(normalisation(P, \sigma), C, \sigma) \xrightarrow{A} (P', C, \sigma')$$

Until now we only used activities as transition labels, but all $StAC_i$ basic processes may be used. As we will see next, most of the $StAC_i$ operational rules use the transition label B , that denotes a basic process. The set \mathcal{B} of all basic processes, such that $B \in \mathcal{B}$, is defined as:

$$\mathcal{B} = \mathcal{A} \cup \{\div_i, \boxtimes_i, \boxdot_i, J \triangleright i, \odot\}$$

where \mathcal{A} represents the set of all activity labels and the remaining set represents the labels of basic processes. In the compensation operators, i an index, and J is a set of indices. The elements of \mathcal{B} are called basic processes because they can not be further decomposed. Although this section does not deal with termination, we have included early termination in \mathcal{B} because \odot is a basic process. Also, excluding \odot from \mathcal{B} would reduce the applicability of the rules presented in this section.

Similarly to CSP we can define a trace of the behaviour of a $StAC_i$ process. In $StAC_i$ the trace of a process describes the sequence of basic processes that occurred. We will use the notation

$$(P, C, \sigma) \xRightarrow{t} (Q, C', \sigma')$$

to denote that process P evolves into process Q by executing the finite sequence t of basic processes from set \mathcal{B} .

The construction of $StAC_i$ traces has the following properties:

1. The empty trace $\langle \rangle$ states that no process has occurred and is represented as:

$$(P, C, \sigma) \xRightarrow{\langle \rangle} (P, C, \sigma)$$

2. If process P evolves into process Q by executing the transition B , followed by the execution of trace t , then there must exist an intermediate process P' that satisfies the right side of the equality:

$$(P, C, \sigma) \xRightarrow{\langle B \rangle^t} (P'', C'', \sigma'') = \exists (P', C', \sigma'). (normalisation(P, \sigma), C, \sigma) \xrightarrow{B} (P', C', \sigma') \wedge (P', C', \sigma') \xRightarrow{t} (P'', C'', \sigma'')$$

where $t \in \mathcal{B}^*$.

Properties 1 and 2 describe the way operational rules will be used: first, the process to be executed is normalised; second, apply the operational rules to the normalised process; third, repeat the first and second steps.

We do not distinguish internal and external choice in the semantics – there is just one kind of choice. Nevertheless, we assume that the choice between enabled transitions with different labels, *e.g.*, $(A; P) \parallel (B; Q)$, is made externally by the environment. Conversely the choice between enabled transitions with the same label, *e.g.*, $(A; P) \parallel (A; Q)$, is made internally by the system.

Next, we give a set of operational rules for StAC_i programs without taking into account early termination.

Activity

We assume that an activity is a relation from states to states, and write $\sigma \xrightarrow{A} \sigma'$ when σ is related to σ' by \xrightarrow{A} . The execution of an activity imposes a change in the state, leaving the compensation function unchanged.

$$\text{R1} \quad \frac{\sigma \xrightarrow{A} \sigma'}{(A, C, \sigma) \xrightarrow{A} (null, C, \sigma')}$$

Condition

In the conditional process $b \rightarrow P$ the execution of P is guarded by a boolean function b . As we have seen, the conditional process is either normalised to P or

null, depending on the result of applying b to the current state.

Recursion

The recursive call of a process N (where $N = P$ is an equation) will execute the process obtained after normalising P :

$$\text{R2} \quad \frac{N = P \wedge (\text{normalisation}(P, \sigma), C, \sigma) \xrightarrow{B} (P', C', \sigma')}{(\text{rec}(N), C, \sigma) \xrightarrow{B} (P', C', \sigma')}$$

This rule does not avoid badly defined recursions as $P = P$ or $P = Q \star A$ where A is an initial activity of Q .

Sequence

This rule states that the sequence $P; Q$ imposes an order in the execution of processes P and Q : the activities within process P are executed first.

$$\text{R3} \quad \frac{(P, C, \sigma) \xrightarrow{B} (P', C', \sigma')}{(P; Q, C, \sigma) \xrightarrow{B} (P'; Q, C', \sigma')}$$

Parallel

The following two rules state that parallel processes can be executed in an arbitrary order.

$$\text{R4} \quad \frac{(P, C, \sigma) \xrightarrow{B} (P', C', \sigma')}{(P \parallel Q, C, \sigma) \xrightarrow{B} (P' \parallel Q, C', \sigma')}$$

$$\text{R5} \quad \frac{(P, C, \sigma) \xrightarrow{B} (P', C', \sigma')}{(Q \parallel P, C, \sigma) \xrightarrow{B} (Q \parallel P', C', \sigma')}$$

Note that the parallel process $P \parallel Q$ terminates (i.e., reduces to *null*) when both P and Q terminate.

Generalised Parallel

The rule for the parallel operator is generalised over a set X . An instance of process P_x is chosen and executed, the result is composed in parallel with the

remaining instances of P_x .

$$\text{R6} \quad \frac{(P_{x_1}, C, \sigma) \xrightarrow{B} (P'_{x_1}, C', \sigma') \wedge x_1 \in X}{(\parallel_{x \in X} P_x, C, \sigma) \xrightarrow{B} ((\parallel_{x \in (X - \{x_1\})} P_x) \parallel P'_{x_1}, C', \sigma')}$$

Choice

The next two rules state that in $P \parallel Q$ only one of the processes P or Q is executed.

$$\text{R7} \quad \frac{(P, C, \sigma) \xrightarrow{B} (P', C', \sigma')}{(P \parallel Q, C, \sigma) \xrightarrow{B} (P', C', \sigma')}$$

$$\text{R8} \quad \frac{(P, C, \sigma) \xrightarrow{B} (P', C', \sigma')}{(Q \parallel P, C, \sigma) \xrightarrow{B} (P', C', \sigma')}$$

Generalised Choice

This operator extends choice over a set X . Only one instance of process P_x is chosen to be executed.

$$\text{R9} \quad \frac{(P_{x_1}, C, \sigma) \xrightarrow{B} (P'_{x_1}, C', \sigma') \wedge x_1 \in X}{(\parallel_{x \in X} P_x, C, \sigma) \xrightarrow{B} (P'_{x_1}, C', \sigma')}$$

Compensation Pair

In the compensation pair where P is the primary process and Q is the compensation task, an evolution in process P does not alter process Q .

$$\text{R10} \quad \frac{(P, C, \sigma) \xrightarrow{B} (P', C', \sigma')}{(P \div_i Q, C, \sigma) \xrightarrow{B} (P' \div_i Q, C', \sigma')}$$

The rule below adds the compensation process Q to the compensation function C , which only happens after process P has finished. The justification for defining R11 as an observable transition, instead of being defined by normalisation, is that R11 changes the compensation function C . Because the normalisation only changes the process being normalised, it does not change σ , R11 had to be defined

as a transition.

$$\text{R11} \quad \frac{}{(null \dot{\div}_i Q, C, \sigma) \xrightarrow{\dot{\div}_i} (null, C[i := (Q; C(i))], \sigma)}$$

$C[i := Q; C(i)]$ denotes that compensation task i is set to Q in sequence with the previous compensation for task i . In this manner, the compensation process is built in the reverse order of the execution of the primary processes.

Reverse

In the next rule, the operator \boxtimes_i causes the compensation task i to be executed, and also resets that compensation task to *null*.

$$\text{R12} \quad \frac{}{(\boxtimes_i, C, \sigma) \xrightarrow{\boxtimes_i} (C(i), C[i := null], \sigma)}$$

Note that compensation tasks do not store any state with them: if the state changes between the compensation being stored and executed, the current state is used.

Accept

The operator \boxdot_i clears the compensation task i to *null*.

$$\text{R13} \quad \frac{}{(\boxdot_i, C, \sigma) \xrightarrow{\boxdot_i} (null, C[i := null], \sigma)}$$

Merge

The operator $J \triangleright i$ merges all compensation tasks of set J in parallel on to the front of compensation task i .

$$\text{R14} \quad \frac{}{(J \triangleright i, C, \sigma) \xrightarrow{J \triangleright i} (null, C[i := (\parallel_{j \in J} C(j)); C(i), J := null], \sigma)}$$

In the above rule the expression $J := null$ denotes attributing to all tasks of set J the process *null*, i.e., $\{j := null \mid j \in J\}$. Set J must be disjoint from i .

Discussion on the Operational Rules

At this point we will discuss some consequences of the rules for the compensation pair operator. The operational rules R10 and R11 state that the compensation process Q will only be added to the compensation function C after its primary process P has finished. But rule R11 does not state when the update of the compensation function will happen, allowing the occurrence of any concurrent processes before that update. To illustrate what we have been discussing, we will present a possible evolution for process:

$$(A \dot{\div}_i A') \parallel (B \dot{\div}_i B')$$

Instead of using the complete configuration, in the transitions below we are only considering changes in the process and in the compensation function for task i .

$$\begin{aligned} ((A \dot{\div}_i A') \parallel (B \dot{\div}_i B'), C(i) = \text{null}) &\xrightarrow{A} ((\text{null} \dot{\div}_i A') \parallel (B \dot{\div}_i B'), C(i) = \text{null}) \\ &\xrightarrow{B} ((\text{null} \dot{\div}_i A') \parallel (\text{null} \dot{\div}_i B'), C(i) = \text{null}) \\ &\xrightarrow{\dot{\div}_i} (\text{null} \dot{\div}_i A', C(i) = B') \\ &\xrightarrow{\dot{\div}_i} (\text{null} \dot{\div}_i A', C(i) = A'; B') \end{aligned}$$

The first transition is caused by the occurrence of activity A , which leaves the compensation task i unchanged. In the second transition activity B occurs. Even though the compensation process A' has not yet been added to the compensation task i , B is allowed to occur because $\text{null} \dot{\div}_i A'$ is composed in parallel with $B \dot{\div}_i B'$. The third transition adds activity B' to the compensation task i , and in the last transition activity A' is pushed into the top of the compensation task i . The above process evolution shows that for a parallel process the compensation is remembered in an arbitrary order, which is what is expected in a parallel process. For example, in process

$$(P \dot{\div}_i P') \parallel (Q \dot{\div}_i Q')$$

the execution of P and Q basic activities is interleaved, no order is imposed on their execution. Similarly, compensation activities P' and Q' can be remembered in any order. Nevertheless, the compensations will be remembered in some order. A way to avoid this is to use an independent compensation task for each

concurrent compensation pair, *i.e.*:

$$((P \dot{\div}_j P') \parallel (Q \dot{\div}_k Q')) ; \{j, k\} \triangleright i$$

The merge will preserve the compensations of P and Q in parallel, avoiding imposing an order in the execution of compensations of concurrent processes.

4.3 Operational Semantics for Termination

To define the operational rules for termination, some extensions have to be made to StAC_i . These extensions guarantee the termination will only affect the intended processes.

The early termination process forces all processes within its “nearest” termination block (or scope) to terminate. For example, in process (4.1) that has two nested termination blocks, we want the early termination to only affect its closest termination block. The early termination of (4.1) must not have any affect in the outermost termination block. We are assuming that all processes have an implicit outermost termination block, so if an early termination does not have a surrounding termination scope, it will affect the overall process. For clarity, we will represent in the examples the implicit outermost termination block.

$$\{ A \parallel \{ B ; \odot ; C \} \} \quad (4.1)$$

The main difficulty in defining an operational interpretation for termination operators is to ensure that an early termination will only effect the appropriate termination block. Process (4.2) has two early termination instructions, the scope of the first one is the outermost termination block, while the scope for the second one is the innermost block. In order to guarantee that the second early termination will not interfere with outermost scope, we have to distinguish the two early termination instructions and also define an explicit connection between each early termination and its nearest termination block.

$$\{ (A ; \odot ; B) \parallel \{ C ; \odot ; D \} \} \quad (4.2)$$

In order to deal with the problems discussed we have made some extensions to

StAC_i . The connection between an early termination instruction and a compensation scope will be done by labelling them with identical indices. For example, \odot_k will affect termination block $\{\dots\}_k$ because that have the same index k . Until now we have not considered the fact that after an early termination the processes within the termination block may continue to execute for several steps. We will represent this delay as a counter that is decreased every time an activity occurs, where its initial value is nondeterministically selected. To represent the delay counter we have extended the labelling of termination blocks to a tuple $\langle v, k \rangle$. The first component can either be \top the top element or a natural number, *i.e.*, $v : \top \mid \mathbb{NAT}$. If v is \top , it states that an early termination has not occurred; if not, an early termination has happened and several concurrent activities (or other basic processes) may be executed. The second component is an index from a set of termination indices K . The early termination operator \odot keeps the initial labelling.

When a termination operator \odot_k occurs within process $\{P\}_{\langle \top, k \rangle}$, the tuple $\langle \top, k \rangle$ will be updated to $\langle n, k \rangle$, where n is a natural number nondeterministically selected. The number n indicates how many remaining concurrent activities can occur inside the brackets $\{\dots\}$. In the following we present valid tuple values for a process $\{P\}$:

$\{P\}_{\langle \top, k \rangle}$ – Process P can continue its execution, no termination instruction has occurred.

$\{P\}_{\langle n, k \rangle}$ – A termination instruction was invoked, but n (for $n > 0$) activities of process P can still be executed. This feature emulates the delay of sending a termination message to a concurrent process.

$\{P\}_{\langle 0, k \rangle}$ – A termination instruction has previously been executed, and process P must terminate immediately.

We will use again process (4.2) to illustrate the labelling of termination operators. In the resulting process, the two termination blocks have different labels. The first early termination has index 1 since its closest termination block is labelled $\langle \top, 1 \rangle$. The second early termination has the same index as its closest termination block.

$$\{ (A; \odot_1; B) \parallel \{ C; \odot_2; D \}_{\langle \top, 2 \rangle} \}_{\langle \top, 1 \rangle}$$

We have defined a new function \mathbb{L} that does the labelling of termination operators. This function is similar to the translation function \mathbb{T} presented in Section 4.6. Function \mathbb{L} only effects the termination block and the early termination instructions. For each termination block a “fresh” index (for example k) is selected and the label $\langle \mathbb{T}, k \rangle$ is attached to the termination block. Each early termination instruction is indexed with the index of the closest termination block. The labelling function \mathbb{L} is described in detail in Appendix B.

We will discuss now some consequences of the interaction of early termination with compensation pairs. Although the operational rules for termination operators have not yet been defined, we will discuss a possible evolution for process (4.3) following the informal interpretation given to those operators.

$$\{ (A \div_i A') \parallel \odot_1 \}_{\langle \mathbb{T}, 1 \rangle} \quad (4.3)$$

After the occurrence of activity A and process \odot_1 , process (4.3) can evolve into (4.4). Taking into account that the interpretation given to a termination block labelled $\langle 0, 1 \rangle$ is to terminate immediately, process $null \div_i A'$ will not be executed. Consequently compensation activity A' will not be remembered, even though its primary action has been executed.

$$\{ (null \div_i A') \}_{\langle 0, 1 \rangle} \quad (4.4)$$

The execution of process (4.3) we have discussed does not have the interpretation we intended for compensation pairs, because the concept of compensation is built on the assurance that if the primary process has occurred its compensation will be remembered. To solve this problem we have created a new operator called *protected block*, which guarantees that when a termination block reaches the label $\langle 0, k \rangle$ any already started protected blocks will be allowed to continue their execution. We refer to a protected block that has already started as *ongoing*. This new operator could be used to protect the execution of any process, but we are only going to use it in two StAC constructs. The first construct is the compensation pair:

$$|P \div_i Q|$$

We are considering that a compensation pair always has an implicit protection

block, so $P \div_i Q$ is a notation simplification for $|P \div_i Q|$. The second construct is the compensation triple, used in Section 3.2 to specify travel agency example:

$$|P; (c \rightarrow (null \div_1 Q_1)) \parallel (\neg c \rightarrow (null \div_2 Q_2))|$$

that ensures that one of the compensations Q_1 or Q_2 will be remembered.

Informally, we can say that if a process has a protection block it ensures that once the process has started its execution it will be allowed to finish, even after the occurrence of an exit. In order to know if a protected block started its execution or not, a boolean value will be attached to each block. The boolean value will be initially *false*, and when the protected process executes a basic activity the boolean value will be changed to *true*.

4.3.1 Normalisation

This section completes the definition of function *norm* by describing the normalisation rules for termination. Rule N25 says that when the process inside a termination block is the *null* process, the termination block has finished and it can be simplified to *null*.

$$\text{N25 } \text{norm}(\{null\}_{\langle v, k \rangle}, \sigma) = null$$

Rule N26 says the invocation of a termination instruction within a sequential process causes the sequential process to terminate immediately. The next rule (N27) shows that early termination is already in a normalised form.

$$\text{N26 } \text{norm}(\odot_k; P, \sigma) = \odot_k$$

$$\text{N27 } \text{norm}(\odot_k, \sigma) = \odot_k$$

The next rule shows how to normalise a termination block, assuming that $P \neq null$ and $v \neq 0$. In this case, the *norm* will normalise the process within the termination block.

$$\text{N28 } \text{norm}(\{P\}_{\langle v, k \rangle}, \sigma) = \{\text{norm}(P, \sigma)\}_{\langle v, k \rangle}$$

Rule N29 states that a termination block indexed by the tuple $\langle 0, k \rangle$ may still continue its execution in order to finish all protected blocks already started. Nevertheless, all processes outside an ongoing protected block will be immediately terminated. The function that terminates those processes is called *terminate* and it will be defined next.

$$\text{N29 } \text{norm}(\{P\}_{\langle 0, k \rangle}, \sigma) = \{\text{norm}(\text{terminate}(P), \sigma)\}_{\langle 0, k \rangle}$$

The last two rules show how to normalise protected blocks.

$$\text{N30 } \text{norm}(|\text{null}|_v, \sigma) = \text{null}$$

$$\text{N31 } \text{norm}(|P|_v, \sigma) = |\text{norm}(P)|_v$$

Function *terminate*

Function *terminate* is only invoked within a termination block that is trying to finish its execution, but before that happens all ongoing protected blocks must be allowed to continue their normal execution. This function finishes all processes that no longer can continue, and keeps the protected blocks that have already started.

The first three definitions show processes that may continue running, as they may contain a protected block. In the sequential process $P; Q$, the second process Q can be eliminated because its execution only starts after P has finished.

$$\begin{aligned} \text{terminate}(P; Q) &= \text{terminate}(P) \\ \text{terminate}(P \parallel Q) &= \text{terminate}(P) \parallel \text{terminate}(Q) \\ \text{terminate}(\{P\}_{\langle n, k \rangle}) &= \{\text{terminate}(P)\}_{\langle n, k \rangle} \end{aligned}$$

The next rule shows that a protected block that has not started its execution (its label is *false*) is terminated immediately. The following rule states that a protected block that has started its execution (its label is *true*) can continue until it has finished.

$$\begin{aligned} \text{terminate}(|P|_{\text{false}}) &= \text{null} \\ \text{terminate}(|P|_{\text{true}}) &= |P|_{\text{true}} \end{aligned}$$

The last rule states that a process, for which the previous rules could not be applied, has to finish immediately:

$$\text{terminate}(P) = \text{null}$$

4.3.2 Operational Rules

This section concludes the presentation of StAC_i operational rules started in Section 4.2.2, by defining the operational rules for the termination.

Protected Block

Rule R15 states that the occurrence of a basic process within a protected process P will place the label *true* on the protection block. It is not necessary to distinguish whether the value v is initially *true* or *false*, in both cases the final label will be *true*.

$$\text{R15} \quad \frac{(P, C, \sigma) \xrightarrow{B} (P', C', \sigma')}{(|P|_v, C, \sigma) \xrightarrow{B} (|P'|_{\text{true}}, C', \sigma') \quad \wedge \quad v \in \text{BOOL}}$$

Termination Scoping

When an early termination occurs, the compensation function and the state remain unchanged. In conjunction with rule R17, this rule may cause a termination block to terminate.

$$\text{R16} \quad \frac{}{(\odot_k, C, \sigma) \xrightarrow{\odot_k} (\text{null}, C, \sigma)}$$

The execution of the early termination instruction \odot_k causes the termination block with the same index k to change its label from $\langle \top, k \rangle$ to $\langle n, k \rangle$, where n is a natural number chosen nondeterministically. The new label denotes that n activities within P' can still be executed.

$$\text{R17} \quad \frac{(P, C, \sigma) \xrightarrow{\odot_k} (P', C, \sigma) \quad \wedge \quad n \in \text{NAT}}{(\{P\}_{\langle \top, k \rangle}, C, \sigma) \xrightarrow{\odot_k} (\{P'\}_{\langle n, k \rangle}, C, \sigma)}$$

When an early termination has not yet occurred, a termination block evolves by executing basic processes of the process within the braces. In this case the

execution of basic process does not cause any changes to the termination block label, as long it is not an early termination with same index of the termination block.

$$\text{R18} \quad \frac{(P, C, \sigma) \xrightarrow{B} (P', C', \sigma') \wedge B \neq \odot_k}{(\{P\}_{\langle \top, k \rangle}, C, \sigma) \xrightarrow{B} (\{P'\}_{\langle \top, k \rangle}, C', \sigma')}$$

Here an early termination has occurred and $n + 1$ represents the number of basic processes that may still occur inside the termination block. This rule states that in P' the number of basic processes that may be executed will be decreased to n , if a basic process occurs within the termination block.

$$\text{R19} \quad \frac{(P, C, \sigma) \xrightarrow{B} (P', C', \sigma') \wedge n \in \text{NAT}}{(\{P\}_{\langle n+1, k \rangle}, C, \sigma) \xrightarrow{B} (\{P'\}_{\langle n, k \rangle}, C', \sigma')}$$

When a termination block reaches the label $\langle 0, k \rangle$, the value 0 in the label states that P must terminate. However, all termination blocks that have started their execution must be allowed to finish, which is guaranteed by rule R20. This rule only allows the execution of basic processes within ongoing protected blocks, which is assured by the normalisation rules N29, N30, and N31. The processes not allowed to run would have been cleared, because $\{P\}_{\langle 0, k \rangle}$ is a normalised process.

$$\text{R20} \quad \frac{(P, C, \sigma) \xrightarrow{B} (P', C', \sigma')}{(\{P\}_{\langle 0, k \rangle}, C, \sigma) \xrightarrow{B} (\{P'\}_{\langle 0, k \rangle}, C', \sigma')}$$

4.4 Examples

Next we will present some StAC_i examples, where each will focus on some complex combination of StAC_i operators, such as early termination combined either with parallel or compensation pair operators. For each process a possible sequence (or sequences) of process evolutions will be shown. The process evolutions are obtained by applying the operational rules of previous sections. The transition:

$$P \xrightarrow{A} P'$$

shows that the occurrence of activity A will cause the normalised process P to evolve to process P' , omitting the state and compensation function. We do not show in the above transition that this process evolution may be the result of applying a sequence of operational rules. We will use the notation $P \rightsquigarrow Q$ to describe the normalisation of P into Q .

Termination and Parallelism

This example illustrates the use of termination blocks combined with the parallel operator. The following process:

$$\{ A \parallel (B; \odot; C) \} \parallel D \quad (4.5)$$

has a termination block in parallel with activity D , and inside the termination block are two parallel processes.

The left expression in transition (4.6) is the result of applying the labelling function \mathbb{L} (that labels termination blocks and early termination instructions) to process (4.5). The transition (4.6) describes the process evolution when activity B is executed. The occurrence of \odot_1 (4.7) changes the label of the termination block from $\langle T, 1 \rangle$ to $\langle 1, 1 \rangle$ and terminates immediately the remaining activity in sequence with \odot_1 . The first number in the label is chosen nondeterministically and shows, in this case, that only one concurrent activity may occur inside the termination block. Expression (4.8) describes the process evolution caused by the execution of activity D . Because D is outside the termination block it does not change the termination block label. In the last expression (4.9) activity A occurs and it causes the termination block to end. As shown in (4.9), after the execution of A , the termination block label becomes $\langle 0, 1 \rangle$. There are several possible process evolutions different from the one presented here: for example, in (4.6) we could have chosen to execute A or D instead of B , or we could have

chosen 0 instead of 1 as the first component of the termination block label.

$$\{ A \parallel (B; \odot_1; C) \}_{\langle T, 1 \rangle} \parallel D \xrightarrow{B} \{ A \parallel (null; \odot_1; C) \}_{\langle T, 1 \rangle} \parallel D \quad (4.6)$$

$$\rightsquigarrow \{ A \parallel (\odot_1; C) \}_{\langle T, 1 \rangle} \parallel D$$

$$\xrightarrow{\odot_1} \{ A \parallel (null; C) \}_{\langle 1, 1 \rangle} \parallel D \quad (4.7)$$

$$\rightsquigarrow \{ A \}_{\langle 1, 1 \rangle} \parallel D$$

$$\xrightarrow{D} \{ A \}_{\langle 1, 1 \rangle} \parallel null \quad (4.8)$$

$$\rightsquigarrow \{ A \}_{\langle 1, 1 \rangle}$$

$$\xrightarrow{A} \{ null \}_{\langle 0, 1 \rangle} \quad (4.9)$$

$$\rightsquigarrow null$$

Nested Termination Blocks

To illustrate the use of nested termination blocks we will use the following process:

$$\{ \{ \odot \parallel A \} \parallel \odot \parallel (B; C) \} \quad (4.10)$$

This example has an outermost compensation block that includes three parallel processes. One of those processes is also a compensation block.

The first process in (4.11) shows that the early termination instructions have the same index as the closest termination block. The transition (4.11) describes the process evolution when \odot_2 occurs. The exit (or early termination) operator only affects the surrounding termination block with the same index, so the label of the termination block $\{ \dots \}_{\langle T, 2 \rangle}$ will change to $\langle 2, 2 \rangle$, where the first number in the label is a number chosen nondeterministically. Notice that \odot_2 does not affect $\{ \dots \}_{\langle T, 1 \rangle}$, because they have different indices. Next, activity B is executed followed by the early termination instruction with index 1. This alters the label of the outermost termination block to $\langle 1, 1 \rangle$, meaning that only one concurrent activity may occur inside it. In the last transition, activity C occurs and causes

the outermost termination block to end.

$$\{\{\odot_2 \parallel A\}_{\langle T, 2 \rangle} \parallel \odot_1 \parallel (B; C)\}_{\langle T, 1 \rangle} \xrightarrow{\odot_2} \{\{null \parallel A\}_{\langle 2, 2 \rangle} \parallel \odot_1 \parallel (B; C)\}_{\langle T, 1 \rangle} \quad (4.11)$$

$$\begin{aligned} &\rightsquigarrow \{\{A\}_{\langle 2, 2 \rangle} \parallel \odot_1 \parallel (B; C)\}_{\langle T, 1 \rangle} \\ &\xrightarrow{B} \{\{A\}_{\langle 2, 2 \rangle} \parallel \odot_1 \parallel (null; C)\}_{\langle T, 1 \rangle} \end{aligned} \quad (4.12)$$

$$\begin{aligned} &\rightsquigarrow \{\{A\}_{\langle 2, 2 \rangle} \parallel \odot_1 \parallel C\}_{\langle T, 1 \rangle} \\ &\xrightarrow{\odot_1} \{\{A\}_{\langle 2, 2 \rangle} \parallel null \parallel C\}_{\langle 1, 1 \rangle} \end{aligned} \quad (4.13)$$

$$\begin{aligned} &\rightsquigarrow \{\{A\}_{\langle 2, 2 \rangle} \parallel C\}_{\langle 1, 1 \rangle} \\ &\xrightarrow{C} \{\{A\}_{\langle 2, 2 \rangle} \parallel null\}_{\langle 0, 1 \rangle} \\ &\rightsquigarrow null \end{aligned} \quad (4.14)$$

Alternatively, after (4.13), activity A could be executed causing the following process evolution:

$$\begin{aligned} &\xrightarrow{A} \{\{null\}_{\langle 1, 2 \rangle} \parallel C\}_{\langle 0, 1 \rangle} \\ &\rightsquigarrow null \end{aligned} \quad (4.14')$$

The execution of A causes the labels of both termination blocks to be decreased by one, making the outermost block to terminate.

A different evolution for process (4.10) is described next. First \odot_1 is executed (4.15). This makes the label of the outermost termination block to evolve from $\langle T, 1 \rangle$ to $\langle 2, 1 \rangle$, indicating that two concurrent instructions may still occur inside it – again this number is chosen nondeterministically. The occurrence of A decreases the number of allowed instructions to one. When \odot_2 occurs, it alters the label of the outermost termination block to $\langle 0, 1 \rangle$, causing the termination block to end.

$$\{\{A \parallel \odot_2\}_{\langle T, 2 \rangle} \parallel \odot_1 \parallel (B; C)\}_{\langle T, 1 \rangle} \xrightarrow{\odot_1} \{\{A \parallel \odot_2\}_{\langle T, 2 \rangle} \parallel null \parallel (B; C)\}_{\langle 2, 1 \rangle} \quad (4.15)$$

$$\begin{aligned} &\rightsquigarrow \{\{A \parallel \odot_2\}_{\langle T, 2 \rangle} \parallel (B; C)\}_{\langle 2, 1 \rangle} \\ &\xrightarrow{A} \{\{null \parallel \odot_2\}_{\langle T, 2 \rangle} \parallel (B; C)\}_{\langle 1, 1 \rangle} \end{aligned} \quad (4.16)$$

$$\begin{aligned} &\rightsquigarrow \{\{\odot_2\}_{\langle T, 2 \rangle} \parallel (B; C)\}_{\langle 1, 1 \rangle} \\ &\xrightarrow{\odot_2} \{\{null\}_{\langle 3, 2 \rangle} \parallel (B; C)\}_{\langle 0, 1 \rangle} \\ &\rightsquigarrow null \end{aligned} \quad (4.17)$$

Early Termination within Compensation

In process (4.18) the compensation for activity A is an early termination instruction. This compensation will be invoked later, and we will see next the consequences of having an early termination within a compensation process.

$$\{ \{ A \div_i \odot \}; (B \parallel \boxtimes_i) \} \quad (4.18)$$

After the execution of A (4.19), instruction \odot_2 will be inserted in compensation task i ending the termination block with label $\langle T, 2 \rangle$. Next, the instruction reverse is executed (4.20) calling the early termination that is in compensation task i . The execution of instruction \odot_2 does not interfere with the remaining termination block, since they have different indices. Last, activity B occurs ending the termination block.

$$\{ \{ A \div_i \odot_2 \}_{\langle T, 2 \rangle}; (B \parallel \boxtimes_i) \}_{\langle T, 1 \rangle} \xrightarrow{A} \{ \{ null \}_{\langle T, 2 \rangle}; B \parallel \boxtimes_i \}_{\langle T, 1 \rangle} \quad (4.19)$$

$$\rightsquigarrow \{ (B \parallel \boxtimes_i) \}_{\langle T, 1 \rangle}$$

$$\xrightarrow{\boxtimes_i} \{ B \parallel \odot_2 \}_{\langle T, 1 \rangle} \quad (4.20)$$

$$\xrightarrow{\odot_2} \{ B \parallel null \}_{\langle T, 1 \rangle} \quad (4.21)$$

$$\rightsquigarrow \{ B \}_{\langle T, 1 \rangle}$$

$$\xrightarrow{B} \{ null \}_{\langle T, 1 \rangle} \quad (4.22)$$

$$\rightsquigarrow null$$

If the compensation pair $A \div_i \odot_2$ was not surrounded by the termination block labelled $\langle T, 2 \rangle$, the early termination would interfere with the outermost termination block (as they would have the same index).

4.5 Executable Semantics

We have implemented an animator for StAC_i processes [LAB⁺01] based on the CSP(PL) animator described in [Leu01]. The StAC_i animator (Figure 4.5) was developed in SICStus Prolog 3.8 and encodes the operational rules and the normalisation functions into Prolog predicates (see Appendix C for the complete prolog encoding). At the moment it supports step-by-step animation and back-

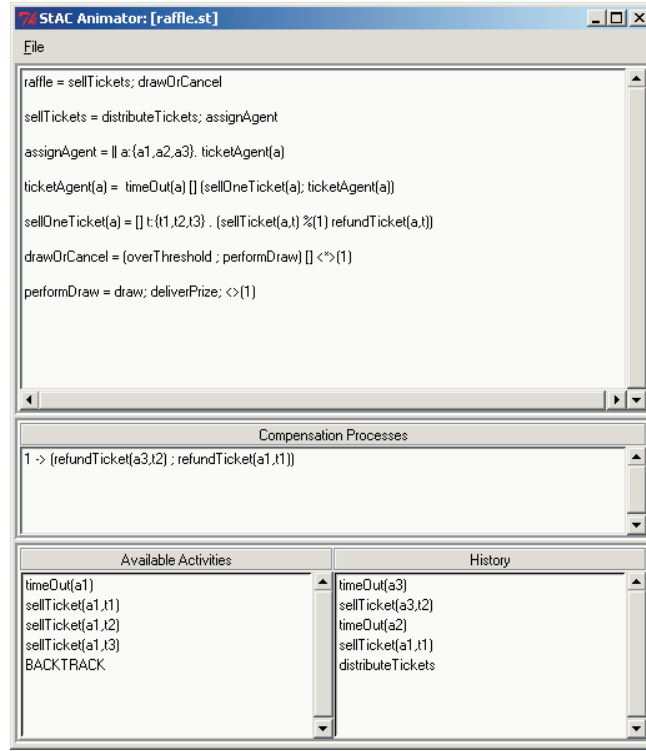


Figure 4.1: The StAC animator

tracking of StAC_i processes.

Although the StAC_i configuration is a tuple with three elements (P, C, σ) , in prolog we will not deal with the B part of the specification. Therefore, the configuration in prolog will just have two first elements (P, C) , which will be represented as the predicate $\text{conf}(P, C)$. The configuration transitions will be encoded as the predicate trans , where $\text{trans}(\text{conf}(P, C), B, \text{conf}(P1, C1))$ says that the occurrence of the basic process B will cause the configuration $\text{conf}(P, C)$ to evolve into configuration $\text{conf}(P1, C1)$. Having defined trans , we will illustrate how to encode two of the operational rules in prolog. Rule R3 states that if the occurrence of B causes the configuration $\text{conf}(P, C)$ to evolve to $\text{conf}(P1, C1)$, then the configuration $\text{conf}(\text{seq}(P, Q), C)$ will evolve to $\text{conf}(\text{seq}(P1, Q), C1)$, where $\text{seq}(P, Q)$ represents process $P; Q$.

$$\begin{aligned}
 \text{(R3)} \quad & \text{trans}(\text{conf}(\text{seq}(P, Q), C), B, \text{conf}(\text{seq}(P1, Q), C1)) :- \\
 & \text{trans}(\text{conf}(P, C), B, \text{conf}(P1, C1)).
 \end{aligned}$$

Rule R17 describes the occurrence of an early termination, represented in prolog as the predicate `exit(I)`, within a termination block. Executing process `exit(I)` will cause the configuration `conf(block(P,true,I),C)` to evolve into `conf(block(P1,N,I),C)`, if `P1` is the process obtained after the occurrence `exit(I)` in `P` and `N` is an arbitrary natural number. To encode rule R17 in prolog we have eliminated the non-deterministic choice of a natural number, and replace it with a randomly selected natural number between 0 and 3. We have chosen number 3 as the upper limit for `N` so that only a small number of basic processes should be allowed to occur after an early termination. Assigning a big number to `N` reduces (or even eliminates) the impact of an early termination within a process, because it can give the process enough time to terminate all its activities.

$$(R17) \quad \text{trans}(\text{conf}(\text{block}(P,\text{true},I),C), \text{exit}(I), \text{conf}(\text{block}(P1,N,I),C)) :- \\ \text{trans}(\text{conf}(P,C), \text{exit}(I), \text{conf}(P1,C)), \text{random}(0,3,N).$$

The animator has helped through the definition and validation of the operational semantics, allowing variations between different operational semantics for StAC_i to be examined. But more important, it helps validating individual specifications, *e.g.*, the examples in Section 4.4 were developed using the animator.

4.6 Translation from StAC to StAC_i

As mentioned before, in StAC_i a process can have several simultaneous compensation tasks, which extends the StAC concept of compensation scope with a single implicit compensation task. So, instead of defining a semantics for StAC language, we have defined a translation of StAC processes into StAC_i processes. The interpretation of a StAC process is given in terms of StAC_i by the translation function.

The translation function \mathbb{T} converts a StAC process into a StAC_i process:

$$\mathbb{T} : L(\text{StAC}) \times I \rightarrow L(\text{StAC}_i)$$

where $L(\text{StAC})$ and $L(\text{StAC}_i)$ represent StAC and StAC_i languages and I is a infinite set of indices. The parameter I is necessary in order to define \mathbb{T} recur-

sively. To translate a process P we have to select an index i from I , and $\mathbb{T}(P, i)$ will construct a StAC_i process.

The first set of rules show the processes that remain unchanged by function \mathbb{T} . Basic activities, null , invocation of a recursive process, and \odot have the same representation for StAC and StAC_i .

$$\begin{aligned}\mathbb{T}(A, i) &= A \\ \mathbb{T}(\text{null}, i) &= \text{null} \\ \mathbb{T}(\text{rec}(N), i) &= \text{rec}(N) \\ \mathbb{T}(\odot, i) &= \odot\end{aligned}$$

The next two rules describe the translation of the acceptance and reversal instructions. The translation of acceptance within index i , results in an acceptance instruction for compensation task i . The translation of the reversal instruction is defined similarly.

$$\begin{aligned}\mathbb{T}(\boxtimes, i) &= \boxtimes_i \\ \mathbb{T}(\boxdot, i) &= \boxdot_i\end{aligned}$$

The following rules show how to translate composite constructs. The translation rules are recursively defined on the constituents of the constructor. For example, the translation of a sequential process $P; Q$ with an index i is the sequential composition of the translation of process P and Q with the same index i .

$$\begin{aligned}\mathbb{T}(b \rightarrow P, i) &= b \rightarrow \mathbb{T}(P, i) \\ \mathbb{T}(P; Q, i) &= \mathbb{T}(P, i); \mathbb{T}(Q, i) \\ \mathbb{T}(P \parallel Q, i) &= \mathbb{T}(P, i) \parallel \mathbb{T}(Q, i) \\ \mathbb{T}(\bigsqcup_{x \in X} P_x, i) &= \bigsqcup_{x \in X} \mathbb{T}(P_x, i) \\ \mathbb{T}(\text{let } X = e \text{ in } P_X, i) &= \text{let } X = e \text{ in } \mathbb{T}(P_X, i) \\ \mathbb{T}(\{P\}_{\langle \tau, k \rangle}, i) &= \{\mathbb{T}(P, i)\}_{\langle \tau, k \rangle} \\ \mathbb{T}(P \div Q, i) &= \mathbb{T}(P, i) \div_i \mathbb{T}(Q, i) \\ \mathbb{T}(|P|_v, i) &= |\mathbb{T}(P, i)|_v\end{aligned}$$

The following set of rules are more complex. The main difficulty comes from parallel processes and their compensation information. Since we do not know the order of execution of $P \parallel Q$, it implies that we also do not know in which order their compensation should be executed. The solution is to create a new

compensation task for each parallel process, so their compensation processes will also be a parallel process: the parallel composition of the new compensation tasks. We are assuming that new compensation tasks will be empty initially. Process P and process Q are translated using the compensation tasks j and k . The resulting processes will be composed in parallel. Last, the new compensation tasks j and k are merged into the initial task i , which means that the compensations of the parallel processes are retained (unless they have been explicitly committed). Notice that compensation tasks are merged in parallel, so the outcome of the merge is process $C(j) \parallel C(k)$, that will be pushed on top of $C(i)$. The second rule is a generalisation of the first rule over a set of parallel processes.

$$\begin{aligned}\mathbb{T}(P \parallel Q, i) &= (\mathbb{T}(P, j) \parallel \mathbb{T}(Q, k)); \{j, k\} \triangleright i \\ \mathbb{T}(\parallel_{x \in X} P_x, i) &= (\parallel_{x \in X} \mathbb{T}(P_x, j_x)); J \triangleright i\end{aligned}$$

where j and k are new distinct indices, and $J = \{j_x \mid x \in X\}$ is a set of new indices such that $x \neq x' \Rightarrow j_x \neq j_{x'}$. The final merge in the second rule means that the compensations of the parallel processes are retained, they are merged in parallel in front of compensation task i .

In the last rule we translate the compensation scoping $[P]$. The scoping brackets are translated to a new compensation task j , then process P is translated using index j . Last, the compensation task j is merged into the initial index i , so all the compensation information that was not reversed or accepted can be preserved by adding it to compensation task i .

$$\mathbb{T}([P], i) = \mathbb{T}(P, j); \{j\} \triangleright i$$

To clarify the translation rules described in this section, we will exemplify the translation of the StAC process $(A \div A' \parallel B \div B'); C$ into a StAC_i process with the same behaviour.

$$\begin{aligned}\mathbb{T}((A \div A' \parallel B \div B'); C, i) &= \mathbb{T}(A \div A' \parallel B \div B', i); \mathbb{T}(C, i) \\ &= (\mathbb{T}(A \div A', j) \parallel \mathbb{T}(B \div B', k)); \{j, k\} \triangleright i; C \\ &= (A \div_j A' \parallel B \div_k B'); \{j, k\} \triangleright i; C\end{aligned}$$

The first rule applies the function \mathbb{T} to both sequential processes $A \div A' \parallel B \div B'$ and C . To translate the parallel process it is necessary to create a new index for each parallel process. After the translation of both parallel processes using the new indices j and k , those indices are merged into the initial index i .

4.7 Integration of StAC_i and B

The operational semantics rules presented in Sections 4.2 and 4.3 allow us to consider a process as an LTS as in [Plo81]. Furthermore, [But00] shows how a B machine can be viewed as an LTS. For those reasons, the semantics of the integration of StAC_i and B will be based on the operational semantics.

A B machine can be viewed as an LTS, where the state space is represented by the cartesian product of the types of state of the machine variables; labels are represented by the operations names and the transitions are represented by the operations.

The semantics of B operations is given in terms of weakest preconditions. For a statement S and postcondition Q , $[S]Q$ represents the weakest precondition under which S is guaranteed to terminate in a state satisfying Q .

In order to define when a transition is allowed by a B operation, we use the notion of conjugate weakest precondition defined as follows:

$$\langle S \rangle Q \triangleq \neg[S]\neg Q.$$

$\langle S \rangle Q$ represents the weakest precondition under which it is possible for S to establish Q (as opposed to the guarantee offered by $[S]Q$). Rules for $[S]$ and $\langle S \rangle$ for a subset of B constructors are shown in Figure 4.1.

Suppose the B machine represents activity A with an operation of the form

$$A \triangleq S,$$

where A is the operation identifier and S is a B AMN statement on the machine state σ , then the transition

$$\sigma \xrightarrow{A} \sigma'$$

$[x := E] Q$	– substitute E for x in Q
$[\mathbf{CHOICE} \ S \ \mathbf{OR} \ T \ \mathbf{END}] Q$	$\hat{=} [S] Q \wedge [T] Q$
$[\mathbf{ANY} \ x \ \mathbf{WHERE} \ P \ \mathbf{THEN} \ S \ \mathbf{END}] Q$	$\hat{=} \forall x \bullet (P \Rightarrow [S] Q)$
$[\mathbf{SELECT} \ P \ \mathbf{THEN} \ S \ \mathbf{END}] Q$	$\hat{=} P \Rightarrow [S] Q$
$\langle x := E \rangle Q$	$\hat{=} [x := E] Q$
$\langle \mathbf{CHOICE} \ S \ \mathbf{OR} \ T \ \mathbf{END} \rangle Q$	$\hat{=} \langle S \rangle Q \vee \langle T \rangle Q$
$\langle \mathbf{ANY} \ x \ \mathbf{WHERE} \ P \ \mathbf{THEN} \ S \ \mathbf{END} \rangle Q$	$\hat{=} \exists x \bullet (P \wedge \langle S \rangle Q)$
$\langle \mathbf{SELECT} \ P \ \mathbf{THEN} \ S \ \mathbf{END} \rangle Q$	$\hat{=} P \wedge \langle S \rangle Q$

Figure 4.2: $[S]$ and $\langle S \rangle$ rules

is possible provided

$$[v := \sigma] (\langle S \rangle (v = \sigma')) \quad (4.23)$$

Here v represents the variables of the state machine. For example, when S is the following B statement,

$$S = \mathbf{SELECT} \ x = 0 \ \mathbf{THEN} \ x := 1 \ \mathbf{END}$$

the transition $\sigma \xrightarrow{A} \sigma'$ is possible if it satisfies the (4.23) condition:

$$\begin{aligned}
& [x := \sigma] (\langle S \rangle (x = \sigma')) \\
&= [x := \sigma] (\langle \mathbf{SELECT} \ x = 0 \ \mathbf{THEN} \ x := 1 \ \mathbf{END} \rangle (x = \sigma')) \\
&= [x := \sigma] (x = 0 \wedge \langle x := 1 \rangle (x = \sigma')) \\
&= [x := \sigma] (x = 0 \wedge 1 = \sigma') \\
&= \sigma = 0 \wedge 1 = \sigma'
\end{aligned}$$

So $\sigma \xrightarrow{A} \sigma'$ may happen provided that $\sigma = 0 \wedge \sigma' = 1$.

A parameterised B operation of the form

$$A(x) \hat{=} S$$

represents a set of activity definitions with labels of the form $A.i$, and the ope-

ration corresponding to activity $A.i$ is given by the statement $x := i; S$. We have not described output parameters since an activity in StAC_i does not have outputs. In StAC instead of having outputs parameters, an output is assign to a variable. An illustration of this approach is presented in the e-bookstore example (see Appendix A.2 for a complete description), where operation **ProcessCard** sets the variable **accepted** to either TRUE or FALSE instead of returning that same value. Later the variable **accepted** is inspected by process Pay to determine whether to execute the compensation process or not.

As in Butler [But92], we have considered the use of output parameters in StAC , but it raises problems when the output is chosen nondeterministically. In that case, the resulting LTS will have one transition for each output value, causing external choice on the LTS. For example, from the operation A^2 (4.24) with output parameter y , where y is chosen nondeterministically,

$$y \leftarrow A \hat{=} y : \in \{0, 1\} \quad (4.24)$$

represents a set of unparameterised activities $A.i$, where i belongs to the set of possible output values:

$$\begin{aligned} &\mathbf{init} \quad skip \\ &A.i \hat{=} \mathbf{SELECT} \ i \in \{0, 1\} \ \mathbf{THEN} \ skip \ \mathbf{END} \end{aligned} \quad (4.25)$$

As we discussed before, operation $A.i$ introduces external choice. An alternative definition for $A.i$ that would solve the limitations of (4.25) is presented in (4.26). In this approach, the value of i is set in advance by a local variable. The local variable x is assigned a value from the set of possible output variables, which introduces internal choice. When $A.i$ is invoked, the value of i is already defined by x .

$$\begin{aligned} &\mathbf{var} \quad x \\ &\mathbf{init} \quad x : \in \{0, 1\} \\ &A.i \hat{=} \mathbf{SELECT} \ i = x \ \mathbf{THEN} \ x : \in \{0, 1\} \ \mathbf{END} \end{aligned} \quad (4.26)$$

We decided to use the method of assigning outputs to variables for two main reasons. First, it avoids the external choice caused by approach (4.25), and the

²The operation definition $y \leftarrow A \hat{=} \dots$ denotes that y is an output parameter of A .

complexity of approach (4.26). Second, it is a convenient way of representing process outcomes, where that outcome may be consulted by several processes.

4.8 Discussion

In this section we discuss alternative approaches to the formalisation of StAC semantics. We consider algebraic and denotational semantics, and the use of bisimulation to determine process equivalence.

We could have built an algebraic semantics for StAC_i , and defined a set of axioms for process equality. However, the nature of the operators supported by StAC_i , like compensation and particularly early termination, eliminates any “relevant” algebraic laws. For example, law (4.27) states that the behaviour of two compensation pairs composed in parallel should be the same as the behaviour of a single compensation pair, where: its primary process is the sequential composition of the primary processes P and Q ; and its compensation is the sequential composition of compensations Q' and P' (compensations are composed in the reverse order of the primary processes).

$$(P \div_1 P'); (Q \div_1 Q') = (P; Q) \div_1 (Q'; P') \quad (4.27)$$

With our operational approach, it can be proved that the law (4.27) is true if all processes in the equation (P , Q , P' , and Q') are basic activities. In general equation (4.27) is false, as P and Q may have nested compensation pairs.

Although we have not defined a denotational semantics for StAC_i , we could have used the method described by Roscoe [Ros98] where a denotational semantics for CSP is deduced from its operational semantics. More specifically, Roscoe shows how to extract a process’s traces, failures, and divergences from its operational execution. It seems feasible to extend Roscoe’s method to deal with StAC_i specific operators, as compensation and termination operators. Still this does not help to prove algebraic laws of operations. To define a denotational semantics, *e.g.* trace semantics, it would be difficult, because we cannot defined the behaviour of, *e.g.* \boxtimes , without knowing the context in which it appears, as this determines which compensations should be executed. We would have to include

compensation tasks in the semantic model, which is not an abstract approach.

Milner [Mil89] defines the semantics of CCS through an operational approach. He proceeds to define an equivalence relation between processes, which he calls bisimulation. The (weak) bisimulation relation determines if two processes are observation equivalent, *i.e.*, the environment with which they interact can not distinguish them. This equivalence relation can be used to verify if a process is an implementation of another process. We could have followed Milner's approach and define a bisimulation relation between StAC_i processes. The difficulty of using bisimulation lies in the method used to prove bisimilarity between processes: one has to find a binary relation that relate those processes and satisfies the bisimulation conditions. We use instead a state machine representation of the specification, which allows the application of model checking or refinement to verify if one process is an implementation of another process. It could be argued that the determination of a bisimulation relation is similar to the determination of a refinement invariant. The advantage of refinement, when using the B method, is the tool support.

Chapter 5

Refinement

In this chapter we explore the refinement of StAC specifications. With refinement the development of a system can start with a very abstract view of the system and gradually add details into the abstract model. The abstract specification will be connected to the “more concrete” specification (that has more details) by a refinement relation that guarantees the concrete specification can replace the abstract specification without the user noticing any change. With this approach a system development is a sequence of specifications, that starts with a very abstract specification and may end with an implementation. Furthermore, because all the steps in the sequence are proved correct, one can be certain that the implementation has the same behaviour as the initial abstract specification.

Notice that the refinement strategy presented in this chapter is not intended as a general refinement method for StAC specifications, our purpose is to study the applicability of refinement in the context of compensation by applying it to a case study.

The next section presents some possible approaches to StAC refinement. Section 5.2 studies a possible strategy to system refinement. In Section 5.3 we apply the strategy described in Section 5.2 to the refinement of the e-bookstore example.

5.1 Refinement Approaches

Taking in account that StAC specifications have two parts, a set of StAC processes and a B machine, the following alternative approaches could be applied to the

refinement of StAC:

- Develop a set of structural refinement rules for StAC.
- Refine the B machine that describes the StAC activities while maintaining the StAC processes unaltered.
- Combine the two parts of the StAC specification in a standard B machine and apply the B notion of system refinement.

From all the possible approaches to StAC refinement, the structural refinement is the method who has the most advantages. To prove a StAC refinement using a set of structural rules, would involve the verification of some well defined properties. For example, a structural rule could state that a parallel process $Q \parallel R$ is a refinement of process P , if it is possible to establish property C . Unfortunately, it would be very difficult to devise those rules for StAC. The main difficulty lies in the fact that StAC has to implicitly maintain multiple compensation tasks, and that the contents of those compensation tasks is only determined at execution time. For that reason we have decided to choose a different approach to the refinement of StAC processes.

The second method for the refinement of StAC specifications is too limited as it does not allow the refinement of StAC processes, only the activities are further refined. Also, because StAC processes may access the B machine variables, this may impose restrictions on the refinement of those variables. In this approach the B machine that contains the system state and activities (which is a standard B machine) can be refined by using the B refinement notion.

In the last method the two parts of a StAC specification are combined into a standard B machine. With this approach the behavioral information defined in the StAC processes will be embedded in the resulting B machine. The fact that StAC has specific operators to deal with compensation makes the “translation” of StAC into a standard B machine a complex task. Compensation tasks and operators like accept, reverse and compensation pair have to be explicitly represented in B. Given that the resulting B machine is standard B, it enables the use of the B refinement relation to verify refinement between StAC specifications. Although the embedding of StAC into B is complex, we have decided to follow

this method as it allows StAC processes and activities to be refined simultaneously.

The remainder of this chapter explores a possible approach to the embedding to StAC into standard B, where some restrictions were imposed on the type of processes supported. Later this strategy is applied to a case study.

5.2 StAC Refinement

The strategy we have defined for refinement of StAC specifications is described in Figure 5.1, and it is based on the csp2b [But00] approach. The first step (*a*) extracts a State Transition Diagram (STD) from a set of StAC process, where the resulting STD describes the order of execution of the activities. The construction of a state transition system (either in a textual or diagrammatic form) is necessary to determine the execution order of the operations. We decided to use a diagrammatic form instead of a textual one because we believe that the diagrammatic form is easier to understand. Besides, in our experience, having a STD helped the determination of the invariant as it is possible to visualise the evolution of both the abstract and refined system.

In the second step (*b*) the information of the STDs is explicitly included in the original B specifications, where the resulting M_B and N_B specifications are standard B machines. With this approach, to prove that N is a refinement of M , it is necessary to build both M_B and N_B machines and prove within the B method that N_B is a refinement of M_B . Because the resulting B machines are standard B we have used *Atelier-B* to generate the proof obligations and its prover to assist in proving those obligations.

It would be difficult to capture the StAC operational semantics into B. The difficulty lays in the fact that the compensation function stores arbitrary complex processes, and that those processes are constructed at “run-time”. To deal with the storage of arbitrary compensation processes, one would have to represent StAC processes as abstract data types, and because B is not well suited for the representation of abstract data types this would be a difficult task. Even if one could represent arbitrary compensation processes in B, the resulting B ma-

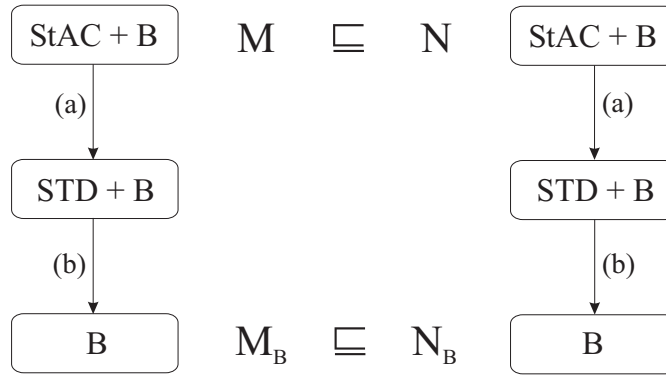


Figure 5.1: StAC refinement

chine would be a complex machine making it difficult to use any of the B tools as we have used to prove the refinement of the e-bookstore case study. For these reasons we imposed some restrictions to the types of processes supported by the embedding of StAC into B defined in Figure 5.1:

- Compensation pairs must be formed by basic activities, *i.e.*, in process $A \div_i D$ both A and D must be basic activities. We imposed this limitation because compensation processes are built at run time, and to support arbitrary complex compensation processes we would have to devise a way to store in B such processes, which would be a difficult task. As we intend to use *Atelier-B* to prove refinement, it is important that the extended machine is not a complex machine, which would make the refinement proofs difficult.
- Parallel processes are supported only at the outermost level. As is mentioned in Section 7.2.3, this limitation could be overcome (partially) by using the method proposed in [SZ02] for translating statecharts into B, which supports the translation of arbitrary parallel processes at any level. Nevertheless, we still would have to restrict the use of generalised parallel processes to the outermost level, as the set used for indexing a generalised process may be an infinite set.
- Early termination is not considered, because at the time the StAC refinement was studied the language did not include termination operators.

5.2.1 Embedding StAC into B

This section describes in detail how to embed the StAC behavioural and compensation related information into a standard B machine. But, before getting into the details, we will use a simple example to illustrate how the StAC embedding into B works in practice.

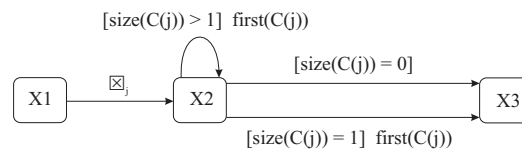
Counter Example

In here we will use a counter example to illustrate how the extended B machine will be built. Our counter starts by increasing a counter. Next the system verifies if the counter has reached its maximum value, in this case the counter has to be restored to an acceptable value, otherwise the current value of the counter is accepted.

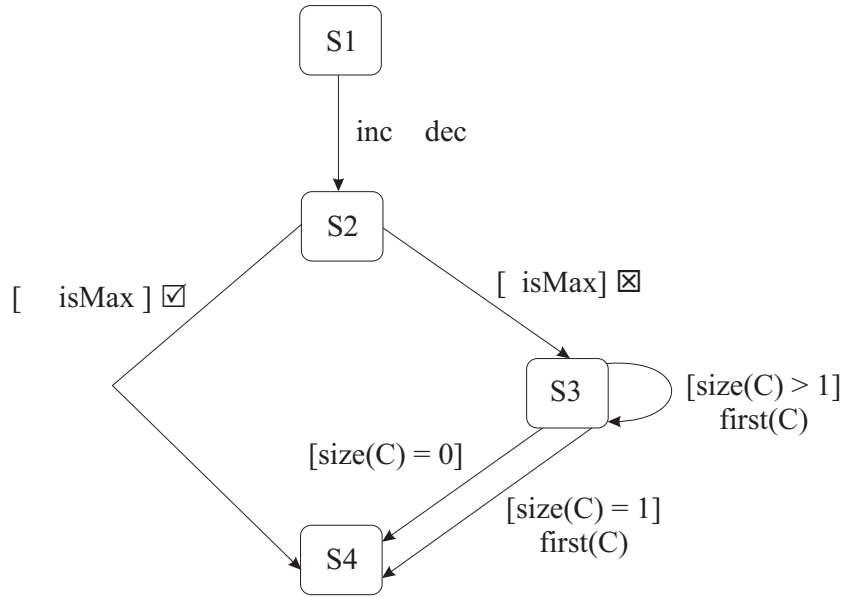
The first process in *Counter* is a compensation pair, the primary task *inc* increases the counter by one, while the compensation task *dec* decreases the counter by one. The second process is a choice guarded by the boolean function *isMax*. If *isMax* evaluates to true, then the reversal is invoked. Otherwise, the acceptance is called.

$$Counter = inc \div dec; (isMax \rightarrow \boxtimes \parallel \neg isMax \rightarrow \boxdot)$$

Figure 5.2 presents the STD for the *Counter* process, which was obtained after applying the rules that will be described later in Section 5.2.1.1. The transition label $inc \div dec$ states that operation *inc* will be executed, and simultaneously *dec* is stored as the compensation for *inc*. The guarded transition from state *S2* to *S4* shows that the accept operator may be executed if *isMax* is false. The reversal has a more complex representation, because it has to invoke sequentially the activities in the compensation. In the general form, \boxtimes_j is represented by the following STD:



where $C \in INDEX \rightarrow seq(ACTIVITY)$ is the compensation function that for each task index returns a sequence of compensation activities. The occurrence of

Figure 5.2: STD for *Counter*

\boxtimes_j causes the state to evolve from $X1$ to $X2$. In state $X2$ only one of the three transitions will be enabled, depending on the number of compensation activities on task j . If compensation task j is empty, the STD will evolve to $X3$ without performing any action. If the compensation task j has a single compensation activity, that activity will be executed and the STD will evolve to $X3$. Otherwise, the compensation activities will be executed sequentially until a single activity remains.

Because *Counter* has a single compensation task, in the STD of Figure 5.2 the expression $C(i)$ necessary to deal with multiple compensation tasks was replaced by C . The STD could be further simplified because we know that the size of C in state $s3$ (after the occurrence of the reversal) will be one, but we decided to use the general rule for reversal.

Machine *Counter_B* shows both the original B machine (called *Counter*) and the extensions necessary to deal with compensation. The original machine has a constant *max*, a variable *ctr*, and operations *inc* and *dec* that will increase and decrease the *ctr* variable. The extended machine has two new sets: *STATE* contains the STD states; while *ACTIVITY* has the names of the B operations that

correspond to compensation activities in the *Counter* StAC process¹. Besides the new states, there are also two new variables, *state* and *C*, that contain respectively, the machine current state and a sequence of compensation activities currently stored. In the initialisation, *state* is *s1* and the compensation function is empty.

MACHINE *Counter_B*

CONSTANTS *max*

PROPERTIES *max* $\in \mathbb{N}$

SETS

STATE = { *s1*, *s2*, *s3*, *s4* };
ACTIVITY = { *dec* }

VARIABLES

ctr, *state*, *C*

DEFINITIONS

isMax == *ctr* = *max*

INVARIANT

ctr $\in \mathbb{N} \wedge$

state $\in STATE \wedge$
C $\in seq(ACTIVITY)$

INITIALISATION

ctr := 1 .. (*max* - 1) \wedge

state := *s1* \wedge
C := []

In the initial B machine *Counter*, operation **inc**² had a single statement where the value of *ctr* is increased by one. Because *inc* is the primary task of *inc* \div *dec*, the extended operation has a SELECT statement that will store the compensation *dec* at the front of the compensation whenever *inc* is executed. Besides that, the

¹Because there is a single compensation activity in *Counter*, the set *ACTIVITY* could be removed. We decide to keep this set to illustrate how a more complex machine could be constructed.

²The expression $e \rightarrow s$ represents prepending element *e* to sequence *s*.

state is updated to $s2$.

inc \triangleq

SELECT $state = s1$ THEN $C := dec \rightarrow C$ $state := s2$ END

||
BEGIN $ctr := ctr + 1$ **END**

Assuming that an activity A is described as the B operation $A \triangleq Q$, we have defined that the extended B operation would have the form:

$$\mathbf{SELECT } G \mathbf{ THEN } P \mathbf{ END } \parallel Q \quad (5.1)$$

in order to maintain a clear separation between the original operation definition and the StAC extensions. Alternatively, we could have defined the extended operation in the following way:

$$\mathbf{SELECT } G \mathbf{ THEN } P \parallel Q \mathbf{ END } \quad (5.2)$$

Because we are assuming the Q is non-aborting, *i.e.*, it is either a **SELECT** or an **ANY** statement, the statements 5.1 and 5.2 are equivalent. Both 5.1 and 5.2 are enabled, if G and the guard of P are also enabled.

Operation dec is not invoked directly, it will be called after the reversal is invoked. This operation will be enabled if the current state is $s3$, the size of sequence C is greater than one, and the compensation operation at the top of sequence C is dec . If all conditions are met, the first element of compensation is removed. Furthermore the state variable is updated when the compensation is empty.

dec \triangleq

SELECT $state = s3 \wedge size(C) \geq 1 \wedge first(C) = dec$ THEN $C := tail(C)$ IF $size(C) = 1$ THEN $state := s4$ THEN END

||
BEGIN $ctr := ctr - 1$ **END**

Reverse is new operation added to the initial *Counter* machine. This operation only updates the current state of the machine.

Reverse $\hat{=}$
SELECT $state = s2 \wedge isMax$ **THEN**
 $state := s3$
END

Again, **Accept** is a new operation to be added to the original machine. This operation updates the current state of the machine and clears the compensation information by setting *C* to the empty sequence.

Accept $\hat{=}$
SELECT $state = s2 \wedge \neg isMax$ **THEN**
 $C := [] \parallel$
 $state := s4$
END

After extending the initial machine *Counter* as we have shown, the resulting machine (*Counter_B*) will contain, in addition to the system state and activities, the behavioural and compensation information described in process *Counter*.

5.2.1.1 From StAC to STDs

Here we will describe how to construct an STD from a StAC specification, by giving each StAC operator a correspondent representation in STDs. From a set of StAC processes, one can construct a STD that defines the order of execution of the system activities and compensation operators. Starting with basic activities, they cause an evolution in the state:



The *null* operator does not cause any transition in the state of the system:



The sequential process *P; Q* is represented by a STD with two state transitions:



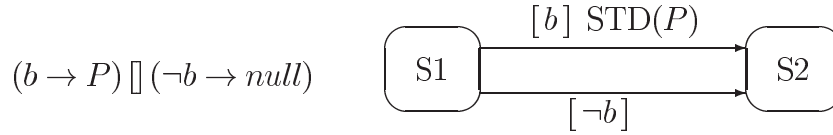
The above diagram raises two problems. The first problem arises when processes P or Q are not basic processes. In this case the transition labels P and Q do not represent transitions, they represent a nested STD. Therefore, from now on, we will use the transition label $\text{STD}(P)$ to describe that a transition contains a nested STD – the STD for process P that has to be unfolded. To unfold a STD one has to replace each transition labelled $\text{STD}(P)$ by a graph describing the STD for process P . The second problem arises from the sequential composition of STDs and how to connect the final state of the STD for P to the initial state of the STD for process Q . A way to solve this problem is to consider that the final state of P is the initial state of Q . This approach overcomes the need of a special transition label to represent the conclusion of the first process in the sequence, but still guarantees that process P has to finish before process Q starts. At the end of this section a small example with sequencing will be presented illustrating how to build a STD from a set of StAC processes. The “correct” STD for process $P;Q$ is presented next, where the final state of process P ($S2$) is used as initial state of process Q :



The conditional expression $b \rightarrow P$ is described as a guarded transition. If the condition b holds, the occurrence of process P will cause a transition in the state.



If the conditional process $b \rightarrow P$ does not have an alternative process, *i.e.*, is not a part of a process with a structure $(b \rightarrow P) \parallel (\neg b \rightarrow Q)$, it is necessary to extend $b \rightarrow P$ to process $(b \rightarrow P) \parallel (\neg b \rightarrow \text{null})$. The justification for adding an alternative branch to each conditional process arises from a difference in the interpretation of conditional processes with a false guard by StAC and STDs: in StAC a false guard is the same as *null* (the process terminates immediately), while in STDs the transition will not occur with a false guard (the process deadlocks), causing the STD to halt. To solve this problem the STD will have an extra guarded transition:



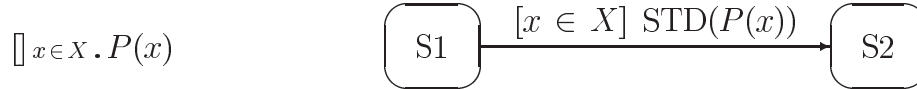
A recursive process is represented by a loop transition, as mentioned before the transition labelled P has to be extended to include the STD for process P :



The choice between two processes is represented by alternative P and Q transitions between the initial and final state.

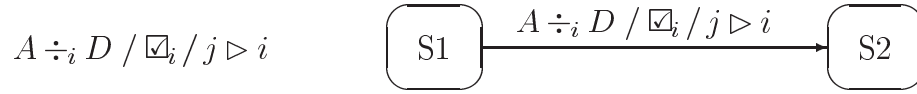


The generalised choice was introduced in the StAC language to avoid dealing with process parameters. Instead StAC has alternative processes for each value of the variable, i.e., $\parallel_{x \in X} P(x)$ describes choosing one of the processes $P(x_1), \dots, P(x_n)$. The generalised choice is represented as a guarded transition:

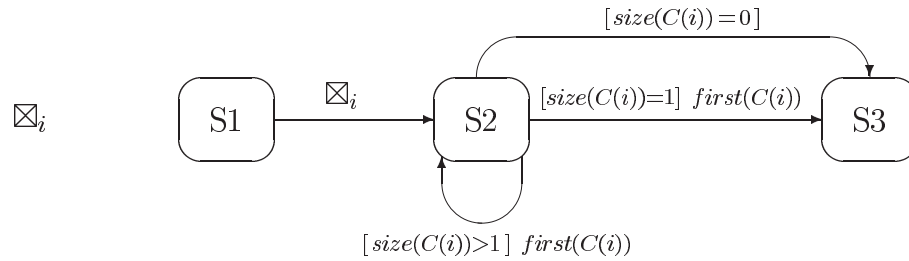


We have not defined a STD for the let statement *let* $X = e$ *in* P_X , instead every occurrence of X will be replaced by the state expression e . The let statement was created so that generalised parallel processes would always have a fixed set of indices, but as we are only considering generalised processes at the outermost level, this implies that their indices will always be a fixed set and that let statements will not be used.

We restricted the compensation pair operator so that both primary and compensation tasks are basic activities. The resulting STD for the process $A \div_i D$ has a single transition with the special label $A \div_i D$. The STD for the accept and merge operators is similar to the compensation pair STD, they only differ on the transition label. In page 85 we present a small example that illustrates how the information contained in the STD will be transposed into B.



To represent the reverse operator in STDs we are assuming that the compensation information is maintained as a function from compensation task indices to sequences of operations, *i.e.*, $C \in INDEX \rightarrow seq(ACTIVITIES)$. After the occurrence of \boxtimes_i there are three alternative transitions. The choice of one of them depends on the size of compensation task i : when it is empty, the STD evolves to state $S3$ (the final state of the STD for the reversal); if it has a single activity, it will execute the last activity and evolve to the final state; if it has more than one activity, it will execute the activity at the front of the sequence and continue in state $S2$.



Because items on the compensation stack are always basic activities and not complex processes, those transitions are not expanded further.

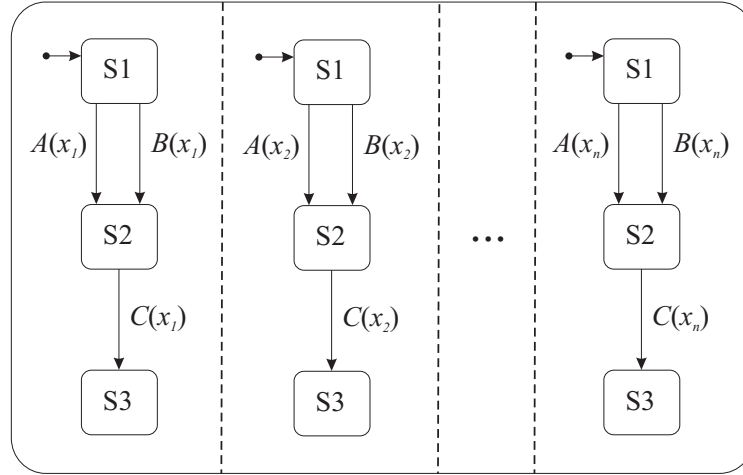
One of the restrictions of the embedding of a StAC process into B is that parallel processes are only supported at the outermost level. The parallel process $P \parallel Q$ is represented as a state with two concurrent substates, the STDs for P and Q . With this approach the activities of both P and Q can be interleaved. We will follow a similar notation and interpretation to the statecharts.



The generalised parallel process is represented, similarly to the binary parallel process, as a composition of several identical STDs being executed concurrently. For example, the process

$$\parallel x \in \{x_1, x_2, \dots, x_n\} . (A(x) \parallel B(x)); C(x)$$

will be represented by n identical concurrent STDs:



5.2.1.2 Building the integral B machine

This section describes how to embed in the system's original B machine the behavioural and compensation related information, described in the associated STD. To achieve that we will add state variables to deal with the STD states and the compensation tasks. Furthermore, the operators accept, reverse and merge will be added to the machine as new operations (only if they are used in the StAC specification) and the original operations will be altered to explicitly handle the STD states and compensation.

State Variables

The boxed elements in Figure 5.3 are the additional sets and variables that need to be included into the original B machine. *STATE* represents the set of states of the STD build from the StAC processes, *INDEX* is the set of all compensation tasks indices, and *ACTIVITY* represents the set of names of the B operations that are used as compensation tasks. The variable *state* describes in which STD state the machine is. In the initialisation clause we are assuming that there is a single initial state (s_1) in the STD, otherwise the variable would be initialised to any of the initial states.

The compensation function C associates to each task a sequence of activities, which implies that the compensation cannot be an arbitrary StAC process. Another restriction is that compensation will be represented as a stack of activities,

thus invoking the reversal operator will execute those activities in the reverse order they were placed in the compensation. A consequence of representing compensation tasks as stacks is that the resulting B machine will be more deterministic than the original StAC specification: the resulting machine imposes an order in the execution of the compensation activities where that order may not have existed in the original StAC specification. Alternatively, the compensation tasks could be represented as a set of activities, entailing a parallel composition of the compensation activities. Using this alternative representation causes the resulting B machine to be less deterministic than its associated StAC specification. This is not viable as any ordering imposed on the compensation activities will be lost.

MACHINE M_B

SETS

S

$STATE = \{s_1, s_2, \dots, s_n\}$
 $INDEX = \{i_1, i_2, \dots, i_k\}$
 $ACTIVITY = \{D_1, D_2, \dots, D_m\}$

VARIABLES

$V, \boxed{state, C}$

INVARIANT

$I \wedge$

$state \in STATE \wedge$
 $C \in INDEX \rightarrow seq(ACTIVITY)$

INITIALISATION

$init \parallel$

$state := s_1 \parallel$
 $C := \lambda index. (index \in INDEX \mid \emptyset)$

Figure 5.3: State of the integral B machine

If a system is specified as a generalised parallel process $\parallel_{x \in X} P(x)$, then it will be represented by n (where n is the number of elements of X) identical STDs evolving concurrently. The variable *state* of Figure 5.3 has to be extended to a function that associates a state to each STD:

$$state \in X \rightarrow STATE$$

Compensation activity arguments Compensation activities with arguments are supported as long as they have the same number and type of arguments. Assuming that the compensation activities have two arguments of type ARG_1 and ARG_2 the compensation function has the following representation in B:

$$C \in INDEX \rightarrow seq(ACTIVITY \times ARG_1 \times ARG_2)$$

To overcome this limitation on the arguments of compensation activities, one would have to extend B to include disjoint union (or coproducts). Then it would be possible to have a set ARG which could be defined as the coproduct of all types of compensation activities arguments:

$$ARG = ARG_1 + ARG_2 + \dots + ARG_n$$

Here ARG is used as a polymorphic argument that can be cast to a specific argument type. This would allow compensation activities to have different number and type of arguments. The next clause defines a general compensation function that does not impose restrictions on the arguments of the compensations:

$$C \in INDEX \rightarrow seq(ACTIVITY \times seq(ARG))$$

Given that the arguments of a compensation activity are defined as sequence of type ARG , they can be any size and type.

Operations

Each operation in the initial B machine has to be extended to include the information of its associated STD. In all the diagrams of this section we are considering that activity A is described as the B operation $A \hat{=} Q$ where Q is an AMN statement, and for simplicity we are not considering operation arguments. The first diagram shows that activity A may occur when the system is in state $s1$ and that the execution of A causes the state to evolve from $s1$ to $s2$. The operation A will be extended with a SELECT statement, that ensures A will be not enabled if the condition $state = s1$ does not hold. One has to take into account that Q can also be a guarded statement, in this case A will be enabled when both $state = s1$ and the guard of Q holds³.

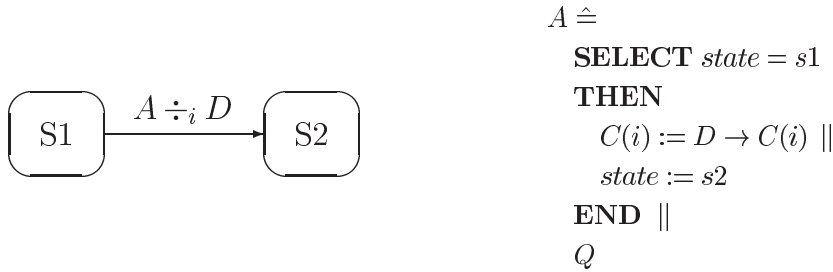
³This is valid provided Q is non-aborting [But00], which is an assumption that we make.



The following STD has a transition A guarded by b , therefore operation A will be extended with a SELECT statement with the additional condition b :



When a STD has a transaction labelled with a compensation pair $A \div_i D$, it describes the execution of A , and the inclusion of D in the compensation task i . Operation A will be guarded by the state variable, and when the machine is in state $s1$, statement Q will be executed (assuming that Q is enabled) and simultaneously the label D will be pushed on top of the compensation task i .



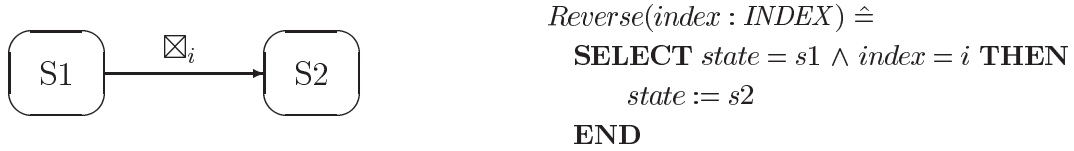
Compensation activity D may be invoked in a later stage if the reversal for task i is invoked. Assuming that the reversal for task i is invoked in state s_j (see STD for reversal in page 92) the compensation operations will be invoked in state s_{j+1} . The SELECT statement added to operation D will be enabled when the system is on state s_{j+1} and D is the activity on the top of the compensation task. The execution of operation D removes the label D from the top of compensation task i and executes its original statement R . If D was the last activity in the compensation task, the execution of the compensation task has finished and the

system will evolve to a new state.

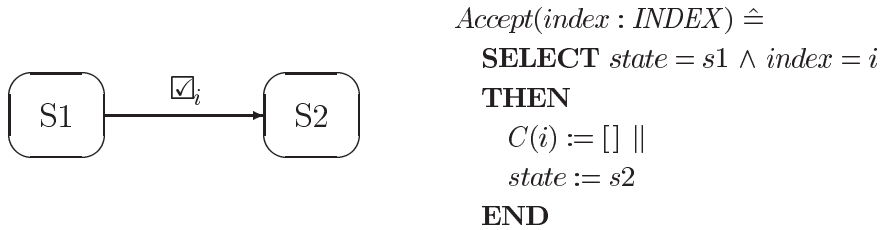
$$\begin{aligned}
 D \triangleq & \\
 & \mathbf{SELECT} \text{ state} = s_{j+1} \wedge \text{first}(C(i)) = D \wedge \text{size}(C(i)) \geq 1 \\
 & \mathbf{THEN} \\
 & \quad C(i) := \text{tail}(C(i)) \parallel \\
 & \quad \mathbf{IF} \text{ size}(C(i)) = 1 \mathbf{THEN} \text{ state} := s_{j+2} \mathbf{END} \\
 & \mathbf{END} \parallel \\
 & R
 \end{aligned}$$

It may be the case that the STD has several transitions labelled \boxtimes_i , consequently operation D must have alternative **SELECT** statement for each invocation of \boxtimes_i . In the last diagram of this section (page 98) we describe how to represent in B the fact that the same operation is used as a label of several transitions.

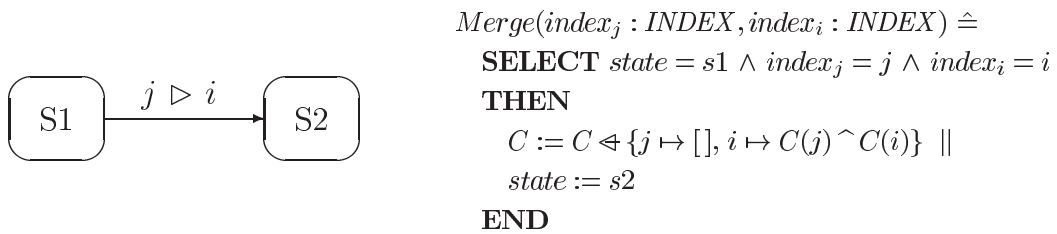
When the overall STD of a system has a transition labelled \boxtimes_i , a new operation *Reverse* has to added to the machine. The *Reverse* will be enabled when the system is on state $s1$ and the parameter index is i ($i \in \text{INDEX}$).



The *Accept* operation will clear the compensation task i by assigning to it the empty sequence:



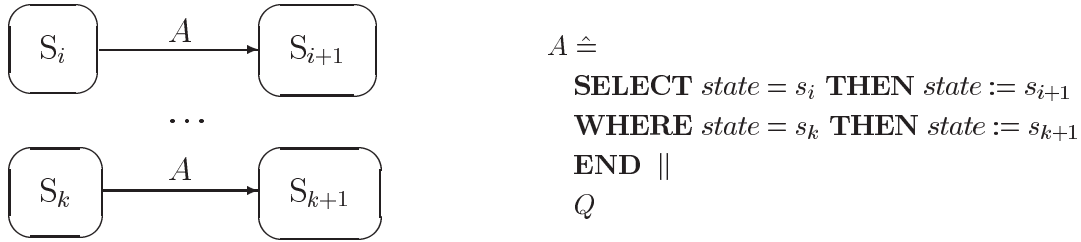
The *Merge*⁴ operation places the compensation task j on front of the compensation task i , and at the same time clear task j :



⁴The expression $s \hat{\ } t$ represents the concatenation of sequences s and t .

We are limiting the merge operator to a single task j instead of a set J of tasks. Representing the general merge operator is problematic because the tasks of set J have to be composed in parallel on top of compensation task i , and we only allow sequential composition of compensation activities. To overcome this restriction one could represent parallel composition of the set of tasks J as the nondeterministic interleaving of $C(j)$ for every $j \in J$. The problem with this approach is that it does not follow the StAC interpretation of parallel composition. Take for example process $A \parallel B$ (A and B are basic activities), in StAC this process is the same as $(A; B) \sqcap (B; A)$, while the nondeterministic interleaving of A and B is equivalent to $(A; B) \sqcap (B; A)$, where \sqcap is the CSP internal choice operator.

When the same operation label is used in several transitions of the STD, the operation will be extended with a SELECT statement that has an alternative branch for each one of those transitions. In the diagram bellow A is used in two transitions, as a result operation A will have an additional SELECT statement with two branches that will enable the operation in state s_i and s_k :



Similarly if the labels \boxtimes_i , \boxdot_i , $J \triangleright i$ appear in several transitions, they will have an alternative SELECT branch for each of those transitions. Furthermore, when \boxtimes_i appears in several transitions, besides affecting its own definition it also affects every compensation activity of compensation task i . Each compensation activity of task i will have an alternative branch for every \boxtimes_i transition.

5.2.2 Refinement in B

The B notion of system refinement applies the standard technique of data refinement to the state of the abstract system. In data refinement an abstraction invariant is used to connect the abstract variables to the concrete variables, and each operation of the abstract system is refined by the correspondent concrete operation.

ration. The following definitions of system refinement were taken from [But02a].

If S is a statement over the abstract variables a , T is a statement over the concrete variables c , and AI is an abstraction invariant, we will write

$$S \sqsubseteq_{AI} T$$

to denote that S is refined by T under the abstraction invariant AI .

Definition 1 (Data Refinement) $S \sqsubseteq_{AI} T$ if the following condition holds:

$$AI \Rightarrow [T] \langle S \rangle AI$$

Which states that if AI is true, then T is guaranteed to terminate in a state where S either fails to terminate or terminates in state satisfying AI .

Definition 2 (B Refinement) A system M is refined by a system N under abstraction invariant AI , if N has an operation $N.a$ corresponding to each operation $M.a$, such that:

$$M.a \sqsubseteq_{AI} N.a$$

Definition 3 (Refinement - Internal operations) Each internal operation $N.i$ of the concrete system N must be such that

$$skip \sqsubseteq_{AI} N.i$$

Although the standard B notion of system refinement does not support internal operations, this can be overcome by adding *skip* operations to the abstract system for each internal operation introduced in the refined system. This allows the use of a B tool to verify refinements with internal operations. To achieve that, both B tools (*Atelier-B* and *B-Toolkit*) start by generating the proof obligations necessary to validate the refinement, and then provide a theorem prover to assist the user in proving the obligations.

5.3 Case Study

The e-bookstore example (see Section 2.3.2) will be used to study the applicability of the refinement strategy of embedding StAC processes into the original B ma-

chine. We will start by defining a more abstract specification of the e-bookstore that provides a simplified functionality of the system without using compensation. This specification captures the basic properties that must be preserved by the system. Some of these properties are: a client cannot exceed his/her predefined budget; books are transferred from the shelf to the basket; transactions can be accepted or rejected; if rejected, books are returned to the shelf. From now on, the abstract e-bookstore will be called *Bookstore0* while the concrete e-bookstore specification will be renamed *Bookstore1*. Ultimately we want to prove that

$$Bookstore0 \sqsubseteq Bookstore1$$

by using the B notion of system refinement.

Abstract model

The abstract e-bookstore is defined as an infinite set of parallel *Client* processes:

$$Bookstore0 = \parallel_{c \in CLIENT} . Client0(c)$$

Process *Client0* is a sequential process, which starts with activity **Arrive** that initialises the client information. The next activity is **Checkout**, which represents a client choosing simultaneously all the books s/he wants to buy. Activity **Checkout** is followed by a choice between paying for the books or abandoning the bookstore without buying any books. The **Pay** activity verifies whether the card of the client is accepted and if the card is rejected the books in the basket will be returned. In the **Quit** activity the client's basket and its content will be returned to the shelves. The last process **Exit** represents the packaging of the all books in the client's basket.

$$Client0(c) = \mathbf{Arrive}(c); \mathbf{Checkout}(c); (\mathbf{Pay}(c) \sqcap \mathbf{Quit}(c)); \mathbf{Exit}(c)$$

The state of the *Bookstore0* machine has two sets: *CLIENT* that represents all clients that can be on-line simultaneously; and *BOOK* that represents all books available in the bookstore. Variables *basket*, *budget*, and *accepted* are partial functions that return for each client, respectively, the selected books, the allowed spending money, and the card status. These functions have the same domain, which represents the set of clients accessing on-line the bookstore. The variable

shelf returns for each book its availability, and *price* contains the price of each book. The first clause in the invariant states that if a client is on-line, then s/he must have a basket, a budget, and a credit card status. The second clause states that every on-line client must keep his/her basket within the predefined budget.

MACHINE *Bookstore0*

SETS *CLIENT*, *BOOK*

VARIABLES *basket*, *budget*, *accepted*, *shelf*, *price*

DEFINITIONS

$overBudget(c) == \sum b. (b \in basket(c) \mid price(b)) > budget(c);$

$inBudget(s, c) == \sum b. (b \in s \mid price(b)) \leq budget(c);$

$inStock(s) == \{b \mid b \in s \wedge shelf(b) \geq 1\} = s$

INVARIANT

$basket \in CLIENT \rightarrow \mathcal{F}(BOOK) \wedge$

$budget \in CLIENT \rightarrow \mathbb{N}_1 \wedge$

$accepted \in CLIENT \rightarrow \mathbf{BOOL} \wedge$

$shelf \in BOOK \rightarrow \mathbb{N} \wedge$

$price \in BOOK \rightarrow \mathbb{N}_1 \wedge$

$dom(basket) = dom(budget) = dom(accepted) \wedge$

$\forall c \in CLIENT. c \in dom(basket) \Rightarrow \neg overBudget(c)$

Next, we will describe in detail most of the *Bookstore0* operations. If client *c* is not already on-line, **Arrive** will initialise the new client's information.

Arrive(*c* : *CLIENT*) $\hat{=}$

SELECT $c \notin dom(basket)$ **THEN**

ANY *a* **WHERE** $a \in \mathbb{N}_1$ **THEN**

$basket := basket \cup \{c \mapsto \emptyset\} \parallel$

$budget := budget \cup \{c \mapsto a\} \parallel$

$accepted := accepted \cup \{c \mapsto \mathbf{FALSE}\}$

END

END

Checkout⁵ is enabled for clients that are already on-line, and it chooses non-deterministically a set of books that are within the client's budget and in stock, and puts them in the basket. This operation gives a very simplified view of choosing books in a bookstore, usually a client would want to choose the books

⁵The expression $r_1 \triangleleft r_2$ represents overriding of r_1 by r_2 .

him/herself.

```

Checkout( $c : CLIENT$ )  $\hat{=}$ 
  SELECT  $c \in dom(basket)$  THEN
    ANY  $books$  WHERE  $books \subseteq BOOK \wedge inStock(books) \wedge inBudget(books, c)$ 
    THEN
       $basket(c) := books \parallel$ 
       $shelf := shelf \triangleleft \lambda(book).(book \in books \mid shelf(book) - 1)$ 
    END
  END

```

Operation **Pay** describes the payment of the books at a very abstract level. **Pay** performs two actions, verifying the client's card and returning the books in the basket to the shelves, if the card is rejected.

```

Pay( $c : CLIENT$ )  $\hat{=}$ 
  SELECT  $c \in dom(basket)$  THEN
    CHOICE
       $accepted(c) := TRUE$ 
    OR
       $accepted(c) := FALSE \parallel$ 
       $basket(c) := \emptyset \parallel$ 
       $shelf := shelf \triangleleft \lambda(book).(book \in basket(c) \mid shelf(book) + 1)$ 
    END
  END

```

Quit represents the client leaving the bookstore without buying any books, so it just returns the books in the client's basket to the shelf. The last operation **Exit** does not alter any state variable it just assigns the basket to an output variable.

Concrete Model

The specification of the e-bookstore presented here differs slightly from the one on Section 2.3.2. First of all, the specification used in this section is written in $StAC_i$ using explicitly several compensation tasks, while the specification presented in Section 2.3.2 was written in $StAC$. Second, the e-bookstore specification has been updated several times, and the version used in this chapter was the one we have

used during the refinement work.

$$\begin{aligned}
Bookstore1 &= \parallel_{c \in clients} . Client1(c) \\
Client1(c) &= \mathbf{Arrive}(c); ChooseBooks(c); \\
&\quad (\mathbf{Quit}(c); \boxtimes_c \\
&\quad \parallel \\
&\quad \mathbf{Pay}(c); (\neg \mathbf{accepted1}(c) \rightarrow \boxtimes_c)); \\
&\quad \mathbf{Exit}(c) \\
ChooseBooks(c) &= \mathbf{Checkout}(c) \parallel (ChooseBook(c); ChooseBooks(c)) \\
ChooseBook(c) &= \parallel_{b \in BOOK} . (\mathbf{AddBook}(c, b) \div_{c1} \mathbf{ReturnBook}(c, b)); \\
&\quad (\mathbf{overBudget}(c) \rightarrow \boxtimes_{c1}); c1 \triangleright c
\end{aligned}$$

The main difference between the abstract and concrete specifications is that each abstract operation corresponds to a sequence of concrete operations:

- Process *ChooseBooks* is a recursive process that selects individually each book, and for each book added to a basket, the budget is verified and if exceeded that book is returned to the shelves.
- The concrete process **Quit** returns the books in the basket one at the time by invoking the reversal instruction, as opposed to the abstract process **Quit** which returns immediately all books in basket.
- The abstract activity **Pay** is replaced by a sequential process that starts by invoking the concrete **Pay**, which will choose to assign the value TRUE or FALSE to the variable *accepted1*. If the card is rejected, the reversal instruction will be invoked causing the books in the basket to be returned in the reverse order they were selected.

The state of *Bookstore1* is similar to the abstract state (each abstract variable v will be replaced by a concrete variable $v1$), so we will only describe the concrete activities that are not identical to their abstract representations. In operation **AddBook** the SELECT construct enables the operation if c is a on-line client, and book b is not already in the basket of the client. If all conditions are met, book b is added to the basket of client c . The operation **ReturnBook** has similar

enabling conditions, but instead it removes a book from the client's basket.

```

AddBook( $c : CLIENT, b : BOOK$ )  $\hat{=}$ 
  SELECT  $c \in dom(basket1) \wedge b \notin basket1(c) \wedge shelf1(b) > 0$ 
  THEN
     $basket1(c) := basket1(c) \cup \{b\} \parallel$ 
     $shelf1(b) := shelf1(b) - 1$ 
  END

```

Checkout is used in process *ChooseBooks* to exit its recursive definition, so it does not need to perform any explicit action. Operation **Quit** is similar to operation **Checkout**. **Quit** is used to determine which action the client wants to perform, quit the bookstore or pay the books.

```

Checkout( $c : CLIENT$ )  $\hat{=}$  SELECT  $c \in dom(basket1)$  THEN skip END

```

Both **accepted1** and **overBudget** are used as guards of conditional processes, so they are specified in B as boolean expressions: **accepted1** is a boolean state variable; **overBudget** is a B definition (see machine *Bookstore0* on page 101).

Operation **Pay** is described as a choice between attributing the value TRUE or FALSE to the variable *accepted1* depending on the card being accepted or rejected. This is a simple abstraction of the real processing which may involve getting authorisation from a credit card company.

```

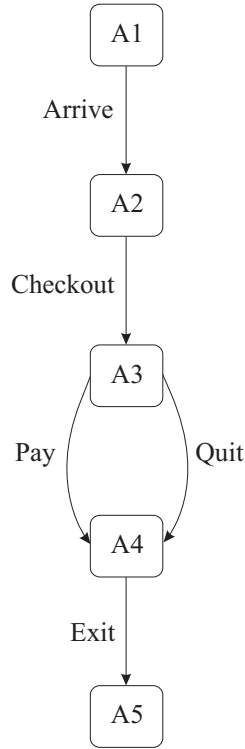
Pay( $c : CLIENT$ )  $\hat{=}$ 
  SELECT  $c \in dom(basket1)$  THEN
    CHOICE  $accepted1(c) := \text{TRUE}$  OR  $accepted1(c) := \text{FALSE}$  END
  END

```

The next sections describe the steps needed to be done in order to prove that *Bookstore1* refines *Bookstore0*.

5.3.1 Dealing with Single Clients

Both abstract and concrete e-bookstore are generalised parallel processes, executing concurrently all clients accessing on-line the bookstore. Therefore, to simplify the determination of the gluing invariant, we decided to deal first with a single client and later extend the invariant for any number of concurrent clients.

Figure 5.4: STD for *Client0*

5.3.1.1 Constructing the *Client0_B* machine

To prove that *Client1* refines *Client0* we will follow the steps described in Figure 5.1. The first step is to extract a STD from the processes for *Client0* using the rules of Section 5.2.1.1. The resulting STD for the *Client0* process (see Figure 5.4) describes the execution order of its activities.

Next, we extend the *Client0* machine to include the behavioural information of its associated STD. The extended machine *Client0_B* has two additional components, a set *STATE* that contains the states of the STD and a variable *state*

that will keep track of the machine current state:

```

MACHINE Client0B

SETS
  BOOK;
  STATE = { a1, a2, a3, a4, a5 }

VARIABLES basket, budget, accepted, shelf, price, state

INVARIANT
  basket ⊆ BOOK ∧
  budget ∈ ℕ ∧
  accepted ∈ BOOL ∧
  shelf ∈ BOOK → ℕ ∧
  price ∈ BOOK → ℕ1 ∧
  state ∈ STATE ∧
  Σ(book).(book ∈ basket | price(book)) ≤ budget

```

The last clause in the invariant guarantees the initial requirement of the client buying within the budget: the cost of all the books in the basket must not exceed the predefined budget.

Because process *Client0* does not deal with compensation, we only have to extend each operation with a SELECT statement that ensures the operation will be executed in the order defined by the STS of Fig. 5.4. In **Checkout** the SELECT statement enables the operation when the system is on state *a2*. The remaining operations are extended in a similar way.

Checkout $\hat{=}$

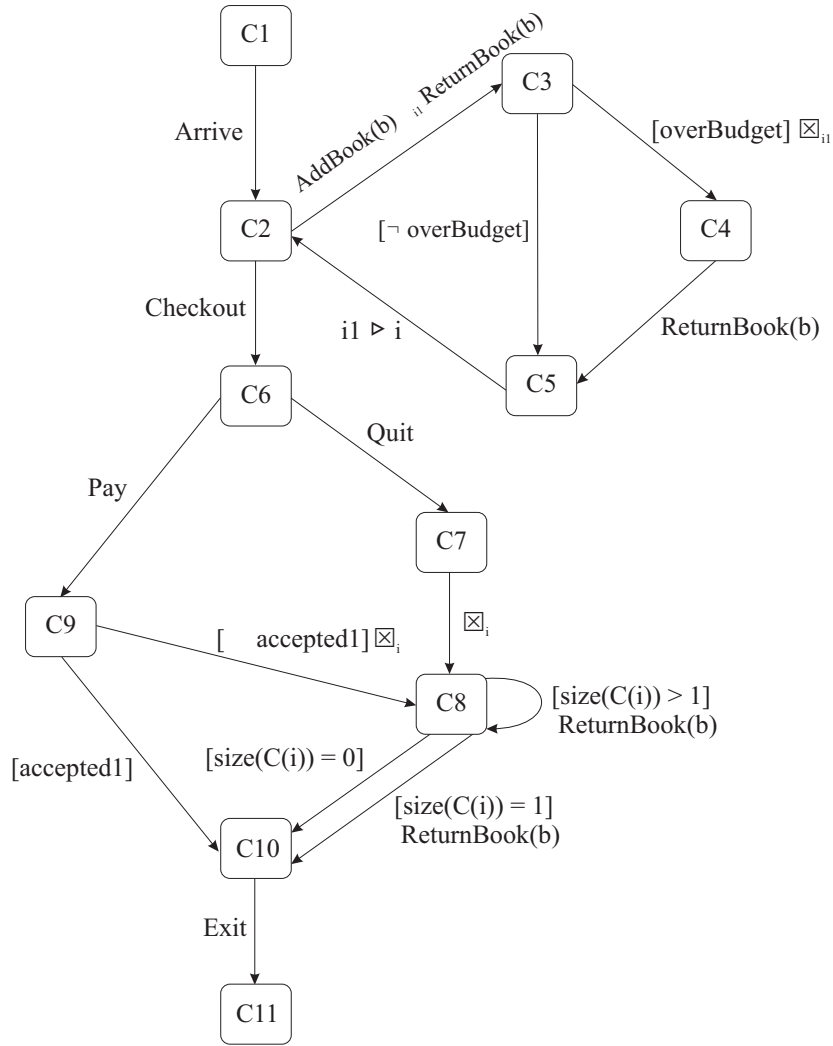
```

SELECT state = a2 THEN state := a3 END ||
ANY books WHERE books ⊆ BOOK ∧ inStock(books) ∧ inBudget(books)
THEN
  basket := books ||
  shelf := shelf ⋈ λ(book).(book ∈ books | shelf(book) − 1)
END

```

5.3.1.2 Constructing the *Client1_B* machine

Figure 5.5 shows the STD extracted from the *Client1* StAC specification. In state *c3*, after adding a book, the system verifies whether the budget was exceed

Figure 5.5: STD for *Client1*

or not, in the former case the reverse will be invoked on compensation task $i1$ (we have changed the compensation task identifier $c1$ and c to $i1$ and i to avoid confusion with the STD concrete states). Notice that the transition after the reversal, from state $c4$ to $c5$, does not follow the rules we presented on Section 5.2.1.1 on how to construct a STD. In addition to those rules we have used specific information about this example to simplify its STD. First, the transitions that occur after the reversal (starting from $c4$ and $c8$) invoke directly the operation **ReturnBook** instead of the general expression $C(j)$, this simplification can be done because **ReturnBook** is the only operation name in both compensation tasks. Second, in state $c4$ we know that the sequence that represents task $i1$ has

exactly one element and that this element is **ReturnBook**, therefore state $c4$ has a single transition that causes the last book added to the basket to be returned to the shelf.

Now that we have the STD for the *Client1* process, the next step is to merge it into the *Client1* machine. Two new sets and variables were added to the state of *Client1* machine. The set *STATE1* has the states used in the STD and the set *INDEX* has the compensation task indices used in the e-bookstore processes. Given that **ReturnBook** is the only compensation operation in the system we only need to “store” in *C* the value of the argument of **ReturnBook**. When the primary activity of the compensation pair **AddBook**(b) \div_{i1} **ReturnBook**(b) occurs, it is necessary to keep the value of b stored so that if a reversal occurs **ReturnBook** will be invoked with the correct argument.

REFINEMENT *Client1_B*

REFINES *Client0_B*

SETS

$STATE1 = \{c1, c2, \dots, c10, c11\};$
 $INDEX = \{i, i1\}$

VARIABLES *basket1, budget1, accepted1, shelf1, price1, state1, C*

INVARIANT

$basket1 \subseteq BOOK \wedge$
 $budget1 \in \mathbb{N} \wedge$
 $accepted1 \in \mathbf{BOOL} \wedge$
 $shelf1 \in BOOK \rightarrow \mathbb{N} \wedge$
 $price1 \in BOOK \rightarrow \mathbb{N}_1 \wedge$
 $state1 \in STATE1 \wedge$
 $C \in INDEX \rightarrow seq(BOOK)$

The concrete operation **Checkout** is used to exit the recursive process of adding single books to the basket, and it becomes enabled in state $c2$ and its execution causes the state to evolve to $c6$.

Checkout $\hat{=}$

SELECT *state* = $c2$ **THEN** *state* := $c6$ **END**

The operation **Quit** is similar to **Checkout**, as it does not perform any action besides changing the state.

The extensions to operation **AddBook** are more complex, because **AddBook** is the primary task of $\mathbf{AddBook}(b) \div_{i1} \mathbf{ReturnBook}(b)$. Therefore, the parameter b has to be added to compensation task $i1$ whenever **AddBook** is executed.

$\mathbf{AddBook}(b : BOOK) \triangleq$

SELECT $state = c2$ **THEN**
 $state := c3$ ||
 $C(i1) := b \rightarrow C(i1)$
END

||

SELECT $b \notin basket1 \wedge shelf1(b) > 0$ **THEN**
 $basket1 := basket1 \cup \{b\}$ ||
 $shelf1(b) := shelf1(b) - 1$
END

Operation **ReturnBook** is a compensation action, so it will be invoked after the occurrence of the reversal in states $c4$ and $c8$. In state $c4$ the operation is called after the reversal of task $i1$, and it will be enabled if the parameter b is equal to the book on top of $C(i1)$ ⁶. In state $c8$ the operation **ReturnBook** is successively invoked until the compensation task i is empty.

$\mathbf{ReturnBook}(b : BOOK) \triangleq$

SELECT $state1 = c4 \wedge size(C(i1)) = 1 \wedge first(C(i1)) = b$ **THEN**
 $C(i1) := tail(C(i1))$ ||
 $state1 := c5$
WHERE $state1 = c8 \wedge size(C(i)) \geq 1 \wedge first(C(i)) = b$ **THEN**
 $C(i) := tail(C(i))$ ||
IF $size(C(i)) = 1$ **THEN** $state1 := c10$ **END**
END

||

SELECT $b \in basket1$ **THEN**
 $basket1 := basket1 - \{b\}$ ||
 $shelf1(b) := shelf1(b) + 1$
END

The **Reverse**, **Merge** and **Null** are new operations to be added to $Client1_B$ machine. The **Reverse** may be invoked in three different states, $c3$, $c7$, and $c9$.

⁶ $C(i1)$ does not need to be a sequence as it contains at most one element.

In each of one this states the reversal will cause the state to evolve to a new state.

```

Reverse(index : INDEX)  $\triangleq$ 
  SELECT state1 = c3  $\wedge$  index = i1  $\wedge$  overBudget(basket1) THEN state1 := c4
  WHEN state1 = c7  $\wedge$  index = i THEN state1 := c8
  WHEN state1 = c9  $\wedge$  index = i  $\wedge$   $\neg$ accepted1 THEN state1 := c8
END

```

The **Merge** is enabled on state *c5* if applied with the expected task indices. **Merge** will put compensation task *i1* on top of task *i* and clear task *i1*.

```

Merge(index1 : INDEX, index2 : INDEX)  $\triangleq$ 
  SELECT index1 = i1  $\wedge$  index2 = i  $\wedge$  state1 = c5
  THEN
    C := {index2  $\mapsto$  C(index1)  $\cap$  C(index2), index1  $\mapsto$  []} ||
    state1 := c2
  END

```

The **Null** operator is used when the STD has unlabelled guarded transitions. The STD for *Client1* has two empty transitions, one from state *c3* to *c5* and another from *c9* to *c10*. In state *c3* the empty transition occurs when the book added to basket by **AddBook** keeps the basket within the budget, no action has to be done and the client may continue choosing books. In state *c9* the client has decided to pay for the books and his/her card was accepted, again no further action needs to be done and the state evolves to *c10*.

```

Null  $\triangleq$ 
  SELECT state1 = c3  $\wedge$   $\neg$ overBudget(basket1) THEN state1 := c5
  WHEN state1 = c9  $\wedge$  accepted1 THEN state1 := c10
END

```

5.3.1.3 Devising an Abstraction Invariant

We need to devise an invariant *I* that relates the variables of the abstract system to those of the refined system:

Abstract	<i>basket</i>	<i>budget</i>	<i>accepted</i>	<i>shelf</i>	<i>price</i>	<i>state</i>	
Concrete	<i>basket1</i>	<i>budget1</i>	<i>accepted1</i>	<i>shelf1</i>	<i>price1</i>	<i>state1</i>	<i>C</i>

Atelier-B was used to generate the proof obligations and to help construct most of the invariant clauses in a incremental way. We will explain next how we used

Atelier-B to help us construct the final invariant.

When applying the *Atelier-B* automatic prover to a refinement (or machine) there are two possible outcomes, all proofs are proved (the specification is proven correct) or there are some proof obligations left unproved. When the automatic prover fails to prove an obligation, the user has to examine each failed proof obligation and determine the reason for that failure:

1. The proof obligation is too complex to be done automatically.
2. The proof obligation is impossible to prove with the present invariant clauses.
3. The proof obligation is false, so the refinement claim is invalid.

In the first case, the user has to assist the automatic prover in its demonstration, by using a set of interactive commands provided by *Atelier-B*. In the second case, the invariant is too weak as its clauses are not sufficient to prove all proof obligations. This can be solved by strengthen the invariant with new clauses. With some specifications, the clauses to be added can be extracted almost directly from unproved obligations. This is what we called earlier “*Atelier-B* helping to construct the invariant” which is done by strengthening the invariant with the failed proof obligations. In the last case, either the specification or the refinement (or both) have to change.

We are going to follow the incremental approach of building the invariant presented on [But02b]. Initially we just added the clause C_1 to the invariant, stating that the concrete variables *price1*, *budget1* and *accepted1* are equal to the correspondent abstract variables⁷.

$$C_1 \quad price1 = price \wedge budget1 = budget \wedge accepted1 = accepted$$

After using the automatic prover on *Client1_B* with the clause C_1 several proof obligations where left unproved. Figure 5.6 shows *Atelier-B* interactive prover applied to one of those unproved obligations, where the user is asked to help the

⁷This is generated automatically by *Atelier-B* if abstract and concrete variables have the same name.

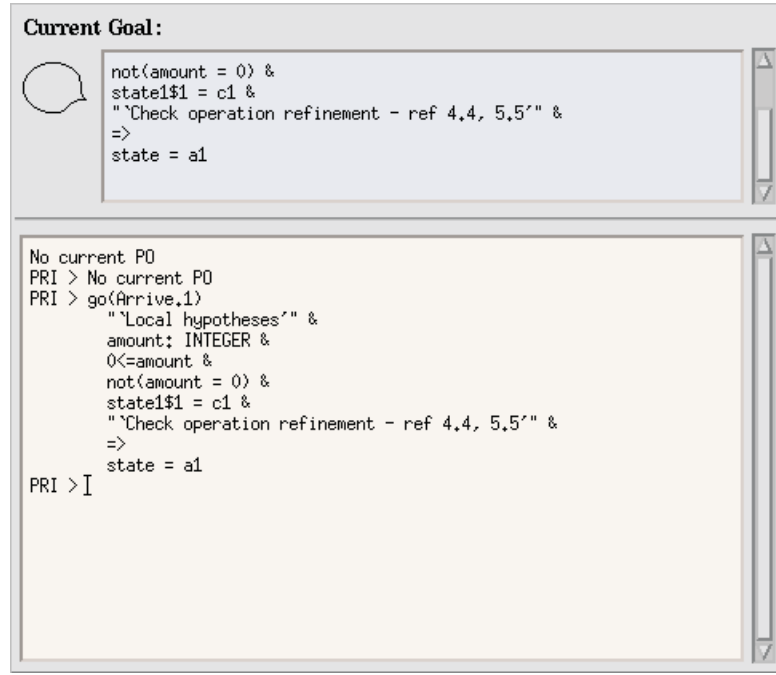


Figure 5.6: AtelierB interactive prover

automatic prover discharging the proof obligation **Arrive1**. This proof obligation corresponds directly to $C_{2.1}$. The other unproved obligations corresponded to the remaining clauses of C_2 . The clauses C_2 relate the abstract variable *state* to the concrete variable *state1*. Those clauses could be deduced directly from analysing both the abstract and concrete STD. For example, in any of the states $\{c2, c3, c4, c5\}$ the transition *Arrive* has occurred and the only external operation that may occur next is **Checkout**, which are the incoming and outgoing transitions of *a2*.

$$\begin{array}{ll}
 C_{2.1} & state1 = c1 \Rightarrow state = a1 \\
 C_{2.2} & state1 \in \{c2, c3, c4, c5\} \Rightarrow state = a2 \\
 C_{2.3} & state1 = c6 \Rightarrow state = a3 \\
 C_{2.4} & state1 \in \{c7, c8, c9, c10\} \Rightarrow state = a4 \\
 C_{2.5} & state1 = c11 \Rightarrow state = a5
 \end{array}$$

The invariant C_3 describes in which states both abstract and concrete baskets have the same books. Clause $C_{3.1}$ was not constructed directly by *Atelier-B*, but it was introduced in the process of interactively proving the proof obligation for

Checkout. $C_{3,1}$ shows that after **Checkout** both systems baskets are identical, because in the concrete system the client has finished choosing individually each book of *basket1* and in the abstract system a set of books was placed in *basket*. Also in the final state both baskets must have the same books. Clause $C_{3,2}$ was constructed by *Atelier-B* and it shows that if the client's card was accepted, both baskets must have the same books.

$$C_{3,1} \quad state1 \in \{c6, c10\} \Rightarrow basket1 = basket$$

$$C_{3,2} \quad state1 = c9 \wedge accepted1 = \text{TRUE} \Rightarrow basket1 = basket$$

Clause $C_{4,1}$ was added in order to prove clause $C_{3,1}$ and it says that the abstract basket will be empty after the occurrence of **Quit**, while the concrete basket still has all the books chosen by the client. The clause $C_{4,2}$ was constructed directly by *Atelier-B*, and it states that after **Pay** and if the client's card was not accepted the abstract basket will be empty, because operation **Pay** removes the books from the basket at the same times the card is rejected. The concrete system operation **Pay** just verifies the card maintaining all the books in the basket, as they will be removed by invoking the reversal.

$$C_{4,1} \quad state1 \in \{c7, c8\} \Rightarrow basket = \emptyset$$

$$C_{4,2} \quad state1 = c9 \wedge accepted1 = \text{FALSE} \Rightarrow basket = \emptyset$$

The clauses C_5 relates the compensation tasks to the concrete basket. Clauses $C_{5,1}$ and $C_{5,2}$ say that each book on the basket must be in one of the compensation tasks but not in both. The last clause is necessary to prove the two previous clauses.

$$C_{5,1} \quad ran(C(i1)) \cup ran(C(i)) = basket1$$

$$C_{5,2} \quad ran(C(i1)) \cap ran(C(i)) = \emptyset$$

$$C_{5,3} \quad state1 \in \{c2, c6, c7, c8, c9\} \Rightarrow C(i1) = []$$

Although in the abstract system the value of the books in the basket is always within the budget, in the concrete system after the operation **AddBook** the basket may exceed the budget, and as a result the last book added to basket must be returned. The fact that there are states in the concrete machine where the budget is exceeded does not breach the abstract invariant, because this only happens with internal operations that are not visible to the abstract machine. Clause $C_{6,1}$ asserts that before adding a new book to the basket (state *c2*) and

after returning the last book added if the budget was exceeded (state $c5$), the basket is within the budget. Clauses $C_{6.2}$ and $C_{6.3}$ were directly constructed by *Atelier-B* after we added clause $C_{6.1}$. These two clauses show that if the budget was exceeded after the occurrence of **AddBook**, this was caused by adding the last book to the basket.

$$\begin{array}{ll}
C_{6.1} & state1 \in \{c2, c5\} \Rightarrow inBudget(basket1) \\
C_{6.2} & state1 = c3 \wedge overBudget(basket1) \Rightarrow \\
& inBudget(basket1 - \{first(C(i1))\}) \\
C_{6.3} & state1 = c4 \Rightarrow inBudget(basket1 - \{first(C(i1))\})
\end{array}$$

The last clause of the invariant (C_7) says that each book in the abstract and concrete system are either in basket or in the shelf, although the abstract and concrete values may not agree. For example, a book might be in the shelf in the abstract system and in the basket in the concrete system.

$$\begin{array}{ll}
C_7 & \forall book. book \in BOOK \Rightarrow shelf(book) + inBasket(book, basket) = \\
& shelf1(book) + inBasket(book, basket1)
\end{array}$$

Proving the Refinement

We have proved using *Atelier-B* that the clause $C_1 \wedge C_2 \wedge \dots \wedge C_7$ is a gluing invariant for the refinement of $Client0_B$ by $Client1_B$. The total number of proof obligations adds up to 202, of those 171 were automatically proved by the prover of *Atelier-B*. From the remaining 31 proofs, 13 of those were fairly easy to prove by interaction with the prover. For the remaining 18 unproved obligations it was necessary to define a user rule file to assist the automatic prover. The user rules are necessary when the prover rule database does not have rules to deal with a specific type of proof. Most of our rules were concerned with sequences or lambda expressions, for which the rule database had a very limited set of rules. Even with the user rules 4 proofs were difficult and time consuming.

Proving the refinement for a single client was very useful, because it allowed us to develop a gluing invariant in an incremental way. The *Atelier-B* prover constructed most of the invariant clauses and by attempting to prove proof obligations for weak gluing invariants we have constructed some invariant clauses needed to prove other proof obligations.

5.3.2 Dealing with Multiple Clients

In this section we show how to generalise the gluing invariant for a single client to deal with any number of concurrent clients.

5.3.2.1 Alterations on both B Machines

In the machine and refinement the variables associated to the client were extended to partial functions, where the domain of those functions describe the clients currently on-line.

<p>MACHINE <i>Bookstore0_B</i></p> <p>SETS</p> <p><i>CLIENT</i>;</p> <p><i>BOOK</i>;</p> <p><i>STATE</i> = { <i>a1</i>, <i>a2</i>, <i>a3</i>, <i>a4</i>, <i>a5</i> }</p> <p>...</p> <p>INVARIANT</p> <p><i>basket</i> ∈ <i>CLIENT</i> → $\mathcal{F}(\text{BOOK})$ ∧</p> <p><i>budget</i> ∈ <i>CLIENT</i> → \mathbb{N} ∧</p> <p><i>accepted</i> ∈ <i>CLIENT</i> → BOOL ∧</p> <p><i>state</i> ∈ <i>CLIENT</i> → <i>STATE</i> ∧</p> <p><i>shelf</i> ∈ <i>BOOK</i> → \mathbb{N} ∧</p> <p><i>price</i> ∈ <i>BOOK</i> → \mathbb{N}_1</p>	<p>REFINEMENT <i>Bookstore1_B</i></p> <p>REFINES <i>Bookstore0_B</i></p> <p>SETS</p> <p><i>STATE1</i> = { <i>c1</i>, <i>c2</i>, ..., <i>c10</i>, <i>c11</i> };</p> <p><i>INDEX</i> = { <i>i</i>, <i>i1</i> }</p> <p>...</p> <p>INVARIANT</p> <p><i>basket1</i> ∈ <i>CLIENT</i> → $\mathcal{F}(\text{BOOK})$ ∧</p> <p><i>budget1</i> ∈ <i>CLIENT</i> → \mathbb{N} ∧</p> <p><i>accepted1</i> ∈ <i>CLIENT</i> → BOOL ∧</p> <p><i>state1</i> ∈ <i>CLIENT</i> → <i>STATE1</i> ∧</p> <p><i>C</i> ∈ <i>CLIENT</i> → (<i>INDEX</i> → <i>seq</i>(<i>BOOK</i>)) ∧</p> <p><i>shelf1</i> ∈ <i>BOOK</i> → \mathbb{N} ∧</p> <p><i>price1</i> ∈ <i>BOOK</i> → \mathbb{N}_1</p>
---	--

Each operation of the abstract and concrete system will have an extra parameter, the client that is invoking the operation. As an example, we present the abstract operation **Quit** of the *Bookstore0_B*:

```

Quit(c : CLIENT) ≐
  SELECT state(c) = a3 THEN state(c) := a4 END ||
  SELECT c ∈ dom(basket)
  THEN
    basket(c) := ∅ ||
    shelf := shelf ⋈ λ(book).(book ∈ basket(c) | shelf(book) + 1)
  END

```

All the remaining abstract and concrete operations have to be extended in a similar way to include the parameter *c*.

5.3.2.2 Alterations on the Abstraction Invariant

The gluing invariant I_B for the bookstore refinement will be similar to the client invariant I_C . Although the invariant I_B needs an extra clause asserting that the domains of the variables related to the bookstore clients are the same. This clause implies that if a client has a basket, s/he must also have a budget, a compensation function, *etc.*. The last conjunction of the clause B_0 states that the set of clients on-line in the abstract and concrete system must be the same.

$$B_0 \quad \begin{aligned} & \text{dom}(\text{basket1}) = \text{dom}(\text{budget1}) \quad \wedge \quad \text{dom}(\text{basket1}) = \text{dom}(\text{accepted1}) \wedge \\ & \text{dom}(\text{basket1}) = \text{dom}(\text{state1}) \quad \wedge \quad \text{dom}(\text{basket1}) = \text{dom}(\text{thread1}) \wedge \\ & \text{dom}(\text{basket1}) = \text{dom}(\text{basket}) \end{aligned}$$

As we said I_B is similar to I_C and, with the exception of B_0 , all the clauses of I_B were obtained by generalising each I_C clause for a set of clients. The gluing invariant I_B is defined as the following conjunction:

$$I_B = B_0 \wedge C_1 \wedge B_2 \wedge \dots \wedge B_7$$

The clause C_1 stays unaltered, because generalising it over a set of clients does not alter the original clause. We will describe in more detail the clauses B_4 and B_7 .

Clause $B_{4.1}$ universally quantifies clause $C_{4.1}$ over the set of on-line clients. A client is on-line if it is defined for the partial function *state1*. It is not necessary to verify the other client functions, because B_0 says they all have the same domain. All the clauses from B_2 to B_6 were generalised similarly.

$$B_{4.1} \quad \begin{aligned} & \forall \text{client}. \text{client} \in \text{dom}(\text{state1}) \wedge \\ & \text{state1}(\text{client}) \in \{c7, c8\} \Rightarrow \text{basket}(\text{client}) = \emptyset \end{aligned}$$

Clause B_7 states the same property of clause C_7 , that a book can either be in the shelf or in the basket, although in the former clause one has to consider that a book might be in basket of several clients.

$$B_7 \quad \forall \text{book}. \text{book} \in \text{BOOK} \Rightarrow \text{shelf}(\text{book}) + \text{booksSold}(\text{book}, \text{basket}) = \text{shelf1}(\text{book}) + \text{booksSold}(\text{book}, \text{basket1})$$

where *booksSold* is the following B definition:

$$booksSold(b, t) == card(\{client \mid client \in dom(t) \wedge b \in t(client)\}).$$

Proving the Refinement

The fact that almost every clause of the invariant has a universal quantification increased considerably the complexity of the proof obligations. The total number of proofs amounts to 250 and only 93 where automatically proved by the *Atelier-B* prover. From the remaining 157 proofs, 100 of those where time consuming but not difficult because we replicated the strategies used in the refinement of the single client system. Of the other 57 proofs, 45 where fairly difficult, and the last 12 proof obligations where extensive and difficult to prove. Nevertheless, all were proved.

5.4 Discussion

Our initial experience of applying the StAC refinement approach to the bookstore example was that it is a difficult process. That view changed when we tried to prove the refinement of a single client, as most of the invariant clauses where constructed by the *Atelier-B* prover and it was possible to check informally if the proof obligations where provable or not. We can conclude that the difficulty level of applying the StAC refinement strategy depends significantly on the system under study. The difficulty level was reasonable for the single client system, but too high for the bookstore system. We believe that the complexity in proving the invariant for the bookstore refinement was the result of the universal quantification on the clauses of the invariant. Those clauses where necessary because the bookstore was defined as a generalised parallel process. Considering that for the case study presented in this chapter the invariant for the refinement of a single process was easily extended to a set of parallel processes, possibly this tactic could be used for other systems defined as generalised parallel processes.

An alternative to refinement would be to develop a model checker for StAC processes. A model checker would allow the verification of several types of properties, as for example invariants – which are properties that must be preserved by the specification at all times, and assertions – which are properties that are

expected to be preserved at the specific point where they were written. In [AB02] the authors explore the use of two existing system, the model checker SPIN and the verification framework STeP, to verify StAC specifications.

Chapter 6

Extending UML for Modelling StAC Specifications

This chapter describes how to extend UML so it can be used as a modelling tool for StAC. The processes of a StAC specification can be modelled with an extended version of activity diagrams, and the data part can be modelled with class diagrams. We will model some of the examples presented in Chapter 2 and 3 with the extended version of UML. In the last section we discuss some possible alterations to the extensions we have proposed for UML activity diagrams.

6.1 Introduction

UML has become the universal modelling language used across a wide range of domains. One of the reasons that contributed to UML's success is that it allows users to adapt the UML language to domain specific models. UML has three extension mechanisms: stereotypes, constraints and tagged values. We will use stereotypes to extend the activity diagrams with a notation for compensation and early termination. This extension of the activity diagrams will be called from now on *compensation activity diagrams*.

A UML class diagram describes the static structure of a system, containing classes and associations. A class describes a set of objects with a common structure in a system, and it is represented by a rectangle with three sections (see classes A and B in Figure 6.1): the class name, a list of attributes, and a list of operations. The associations describe the static relationships that exist between classes, they are



Figure 6.1: Class structure and associations between classes

represented as paths or lines connecting two classes. In Figure 6.1 we show an association connecting classes A and B: the arrow indicates the direction in which to read the association; the label *role A* indicates the behaviour or role expected of class A within the association; the label *0..n* is called role multiplicity, and it says that each instance of class A is related to a set of instances of class B. The multiplicity indicates the lower and upper bounds for the number of instances that can participate in an association, common values are 0..1, \star , 1..n, and 0..n. For example, 0..1 describes either none or one instance, and 0..n describes a set of instances, the lower bound 0 implies that that set may be empty.

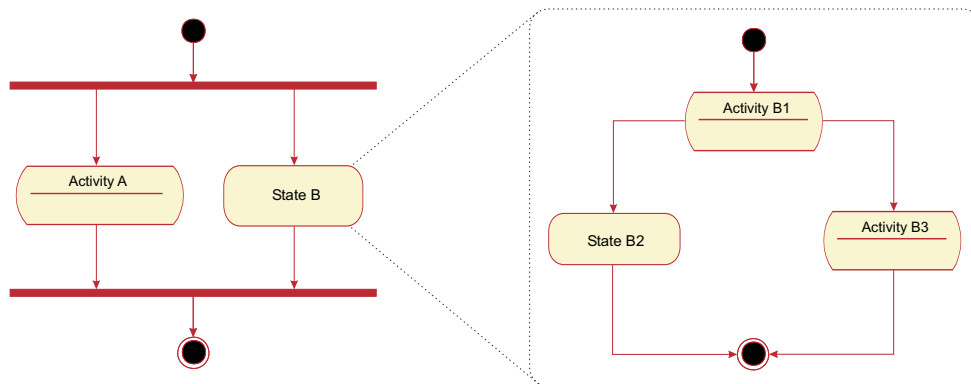


Figure 6.2: State hierarchy in activity diagrams

Activity diagrams were already briefly described in Section 1.2.2.2, but state hierarchy was not explained then. By state hierarchy we mean nested activity diagrams, where a state may have an activity diagram within itself. In Figure 6.2 the state B contains a nested activity diagram. The subdiagram becomes active when state B is active, and conversely state B finishes when its subdiagram reaches the final state. Although not represented in Figure 6.2, and not used in any of StAC examples modelled with activity diagrams, one can also have several transitions going directly into, or out of, a subdiagram. When modelling StAC

processes with activity diagrams we will consider that states contain a nested activity diagram and that activities are basic states that cannot be further decomposed.

The U2B [SB01] is a tool that translates *Rational Rose* UML Class Diagrams into B machines. In the latest version of U2B [SB01] a separate machine is created for each class, containing a set of all instances of that class and a variable with the subset of its current instances. Definitions of sets and constants can be described in the class documentation box, since any text in it will be copied to the machine associated to the class. Associations and attributes are converted to variables whose type is a function from the current instances of the associated class or attribute. The behaviour of the class operations is described in a textual format as annotations in the text boxes provided by *Rational Rose* for the pre-conditions and semantics of the operations. One can say that the U2B tool reorganises and processes the information described in a class diagram and converts that information into a B machine.

6.2 Representing StAC in UML

UML can be used as a modelling language for StAC specifications, where the data is described using UML class diagrams and the behaviour is described using extended activity diagrams. Both class and activity diagrams will be annotated with B following the approach described in [SB01]. Although a tool was not developed for dealing with the proposed extensions, we will show through this chapter that an implementation of a translator similar to the U2B tool is feasible. The StAC translator would generate a B specification plus a set of StAC processes from the UML class and activity diagrams.

6.2.1 Representing Data

The data for a StAC specification will be described as a class diagram, composed of classes and associations. To model the StAC specification data, one has to define the system classes, their attributes and the associations between those classes. It is not necessary to connect the StAC activities to particular classes (as their operations), as StAC specifications have a global state and all activities

may access it without restrictions.

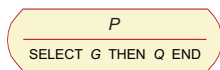
6.2.2 Representing Behaviour

To represent StAC processes, activity diagrams have to be extended with graphical representations for the compensation operators and early termination. The generalised parallel and choice operators can be represented as a normal activity diagram state with annotations describing their multiplicity. The compensation activity diagrams used through this chapter were initially devised by Muan Yong Ng (of the DSSE group at the University of Southampton) based on the BPBeans graphical notation. We have contributed with some alterations to Ng's notation, and added B statements to the diagram activities.

The remaining StAC operators can be represented with the standard activity diagram components. A sequential process can be represented by a sequence of transitions between states, and a conditional process is described as a guarded transition. A parallel process is represented by an activity diagram that starts with a fork with several concurrent paths and ends with a join, ensuring that a parallel process only terminates when all of its concurrent processes have terminated. The choice operator is represented by a state with several outgoing transitions that describe alternative paths.

Activity

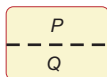
A StAC activity is represented by an activity in the context of activity diagrams.



The diagram activities are decorated with AMN statements, so they correspond almost directly to B operations. In the above activity *P* is described by a SELECT statement, but other AMN statements could be used instead.



Compensation Pair

The compensation pair is represented by a state divided by a dashed line. This




state has two substates, the substate above the dashed line is the primary process and the substate beneath the dashed line is the compensation process.


Acceptance and Reversal

Following the BPBeans notation, the acceptance and reversal operators are   stereotypes states represented, respectively, by a box with a tick and a box with cross. When modelling an example in StAC_i (with multiple compensation tasks) both operators need to be labelled with a task index.

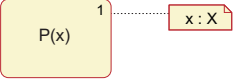
Early Termination

The early termination operator is defined as stereotype state represented by a  not explicitly represented, instead we assume that the scoping of an early termination is its surrounding state. If an early termination is defined in state P , then its scope will be state P and all of P substates.

Generalised Parallel

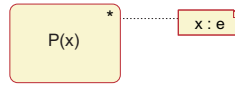
The generalised parallel operator is defined as a state with a multiplicity marker  in the right-upper corner. This feature is known as dynamic concurrency in UML and it indicates that a state will have multiple parallel instantiations. Similarly to the StAC interpretation of the parallel operator, a dynamic concurrent state finishes when all its components have finished. To model generalised parallel processes, the concurrent state has to be extended with a note that defines the set of indices used to distinguish each of its parallel instances.

Generalised Choice

The generalised choice operator is modelled in a similar manner to the generalised parallel operator, and although the marker 1 is not supported by the activity diagrams notation it can be defined as a stereotype state. 

Let

The let statement will not have a graphical representation, instead we consider that a diagram for a generalised process that uses a state expression:



is a notation simplification for the StAC expression $\underline{\text{let}} X = e \underline{\text{in}} \parallel x \in X . P_X$.

Merge

The merge operator is supported implicitly by the compensation activity diagrams. When using either a parallel operator or a compensation scoping in StAC, there is an implicit merge at the end of those processes. There is no necessity to provide an explicit notation for the merge operator, as this operator was only defined for translating StAC processes into StAC_i processes.

6.3 Examples

This section shows the UML description of several examples, where each of the examples illustrates different features of StAC and StAC_i languages. The first example focuses on multiple compensation, while the last two examples focus on implicit compensation tasks and early termination.

6.3.1 Arrange Meeting

The UML description presented here extends the arrange meeting example of Section 2.3.2 to consider not just a single team, but several teams trying to schedule different meetings. Figure 6.3 shows the class diagram for the arrange meeting data. The class diagram has four classes, *PERSON*, *TEAM*, *ROOM*, and *DATE* and six associations between those classes. The association *members* relates each team to a set of persons that belong to the team, the association multiplicity 1..n indicates that each team has to have at least a member. The class *TEAM* has two other associations, *selectedDate* that shows the final date (if it exists) for a meeting, and *availableDates* that contains a subset of the dates for which the room is available. The class *PERSON* has two associations with the class *DATE*, *diary* that contains all the previous bookings for each

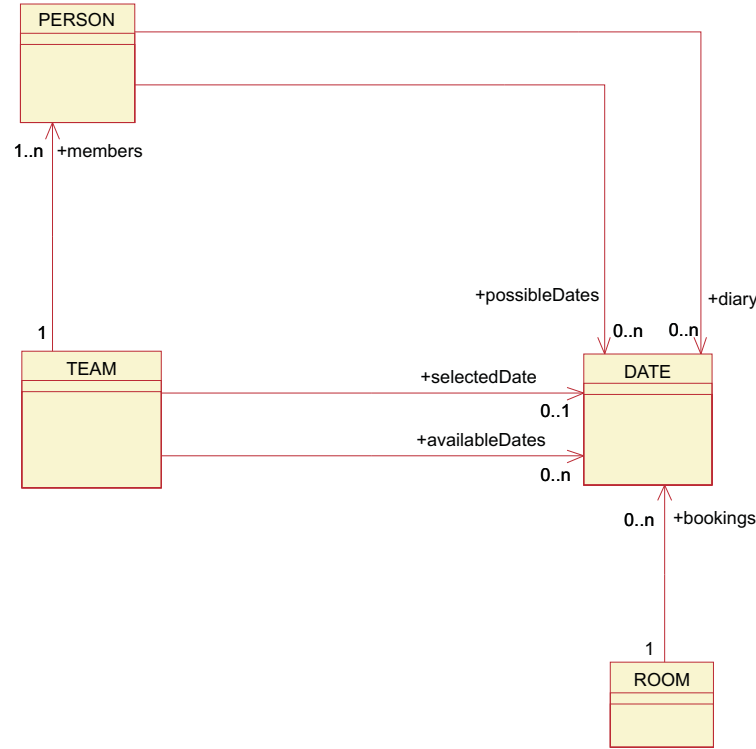


Figure 6.3: Class diagram for the arrange meeting example

team member, and *possibleDates* that represents a set of dates where the team member is available for the meeting. More specifically, the set *possibleDates* is extracted from the set of *availableDates* excluding the dates already booked in the member's diary. The association *bookings* describes the bookings (final and intermediate) for the meeting room. Next, we present the B machine that could be extracted from the class diagram of Figure 6.3 following a similar approach to the U2B tool:

MACHINE *ArrangeMeeting*

SETS *PERSON; TEAM; ROOM; DATE* (1)

VARIABLES *members, selectedDate, availableDates, diary, possibleDates, bookings* (2)

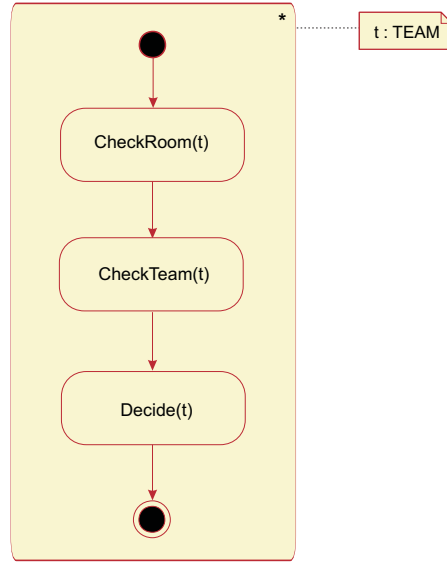
INVARIANT

members \in *TEAM* $\rightarrow \mathcal{P}1(\text{PERSON}) \wedge$ (3)

selectedDate \in *TEAM* $\rightarrow \text{DATE} \wedge$ (4)

availableDates \in *PERSON* $\rightarrow \mathcal{P}(\text{DATE}) \wedge$ (5)

diary \in *PERSON* $\rightarrow \mathcal{P}(\text{DATE}) \wedge$ (6)


 Figure 6.4: Activity diagram for process *ArrangeMeeting*

$$possibleDates \in PERSON \rightarrow \mathcal{P}(DATE) \wedge \quad (7)$$

$$bookings \subseteq DATE \wedge \quad (7)$$

$$inter(ran(members)) = \emptyset \wedge union(ran(members)) = PERSON \quad (8)$$

INITIALISATION

$$members : (members \in TEAM \rightarrow \mathcal{P}1(DATE) \wedge inter(ran(members)) = \emptyset \\ \wedge union(ran(members)) = PERSON) \parallel \quad (9)$$

$$selectedDate : (selectedDate \in TEAM \rightarrow DATE) \parallel \quad (10)$$

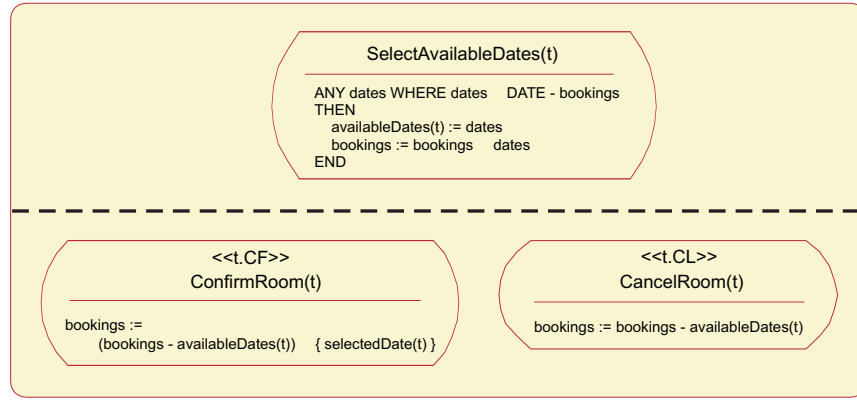
$$availableDates : (availableDates \in TEAM \rightarrow \mathcal{P}(DATE)) \parallel \quad (11)$$

$$diary : (diary \in PERSON \rightarrow \mathcal{P}(DATE)) \parallel \quad (12)$$

$$possibleDates : (possibleDates \in PERSON \rightarrow \mathcal{P}(DATE)) \parallel \quad (13)$$

$$bookings : \in \mathcal{P}(DATE) \quad (14)$$

Each class corresponds to a set in the B machine. For each association a B variable is created, where its type is extracted from the multiplicity information attached to the association. The variable *diary* (6) is defined as a total function from *PERSON* to a subset of *DATE*, which was retrieved from the multiplicity $0..n$ of the association *diary*. The variable *members* needs two clauses in the invariant, clause (3) and (8). The additional clause (8) arises from the extra multiplicity 1 at the beginning of the association arrow, that states that each person can only belong to a single team. In the initialisation the variables are assigned arbitrary functions, with the exception of *bookings* that is assigned a subset of *DATE*.


 Figure 6.5: Activity diagram for process *CheckRoom*

The behaviour of the arrange meeting example is described in StAC as the process

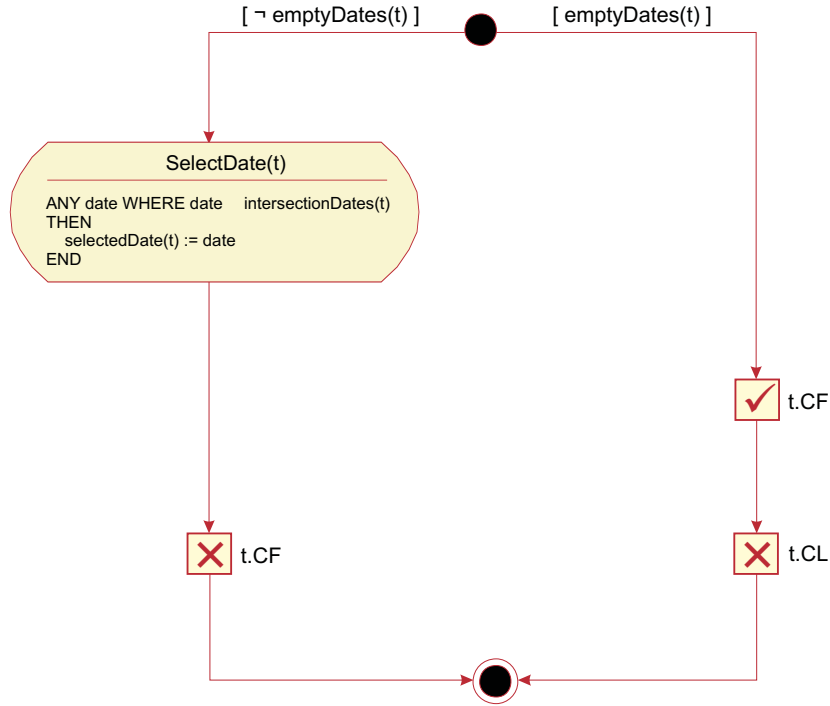
$$ArrangeMeeting = \parallel t \in TEAM . CheckRoom(t); CheckTeam(t); Decide(t).$$

This process is represented in activity diagrams as a concurrent state over the set of teams. Each team will execute the sequential diagram inside the concurrent state that starts with the state *CheckRoom*, followed by *CheckTeam* and finishes with the state *Decide*. Each of these states includes a nested activity diagram, only activities are not decomposable.

The process *CheckRoom* is a compensation pair with two compensation actions, that is represented as a nested compensation pair:

$$CheckRoom(t) = (\mathbf{SelectAvailableDates}(t) \div_{CF(t)} \mathbf{ConfirmRoom}(t)) \div_{CL(t)} \mathbf{CancelRoom}(t)$$

In compensation activity diagrams a compensation pair is represented by a compensation state, where the primary process is above the dashed line and the compensation processes are placed beneath the dashed line. Figure 6.5 shows that *CheckRoom* has as its primary action the activity *SelectAvailableDates*, that is described by an ANY statement that chooses nondeterministically a set of dates where the room is not booked. The diagram shows that there are two compensation actions: *ConfirmRoom* which is the compensation action for task *CF* (the compensation tasks are represented in the diagram as a stereotype); and


 Figure 6.6: Activity diagram for process *Decide*

CancelRoom that is the compensation action for task *CL*. From this diagram it is possible to extract almost directly a B operation from each annotated activity. In the activities of the *CheckRoom* diagram, and in all activities of the examples, for simplicity we have omitted a condition specifying that t is in set *TEAM*. The typing of variable t could be extracted from the diagram of process *ArrangeMeeting* (Figure 6.4), where t was introduced. For example, from activity *CancelRoom* we can extract the following B operation:

```

CancelRoom( $t : TEAM$ ) =
    BEGIN
         $bookings := bookings - availableDates(t)$ 
    END
    
```

The diagram for the *CheckTeam* is not going to be presented here (see Appendix D for a complete description of the arrange meeting example) as it is similar to the *CheckRoom* diagram. The process *Decide* is described as a choice that depends

on the outcome of the boolean function *emptyDates*:

$$\begin{aligned} \text{Decide}(t) = & \neg\mathbf{emptyDates}(t) \rightarrow \mathbf{SelectDate}(t); \boxtimes_{CF(t)} \\ & \parallel \\ & \mathbf{emptyDates}(t) \rightarrow \boxtimes_{CL(t)}; \boxtimes_{CF(t)}. \end{aligned}$$

Figure 6.6 shows process *Decide* modelled as a compensation activity diagram. The StAC choice is describe in the diagram as two outgoing transitions from the initial state, both transitions are guarded by the outcome of *emptyDates*. If its outcome is false (the team members have agreed on a set of dates for the meeting), the activity *SelectDate* will nondeterministically choose a date from the previously agreed dates. After selecting the date the reversal is invoked on the *CF* task for team *t*. If the outcome of *emptyDates* is true, the next state is the acceptance of task *CF*, followed by the reversal of task *CL*.

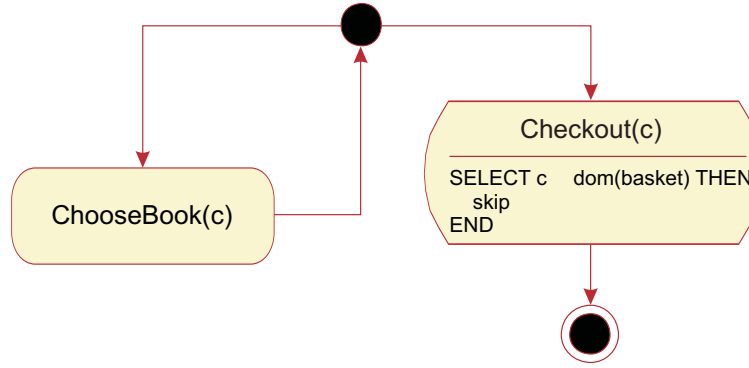
From the UML description of the arrange meeting example it is possible to extract both the B machine containing the system state plus the activities, and the StAC processes that describe the system behaviour. The B machine state can be generated from the class diagram, while the operations can be extracted from the activities annotated with AMN statements. Given that all basic components of activity diagrams correspond to some StAC operator, the generation of a StAC process from an activity diagram seems feasible and fairly simple.

6.3.2 E-Bookstore

This section presents the activity diagrams for two processes of the bookstore specification, *ChooseBooks* that uses recursion and *ChooseBook* that uses compensation scoping and implicit compensation tasks. The remaining processes in the bookstore use features either already presented or that will be described in the next section.

ChooseBooks is described in StAC as a recursive process:

$$\text{ChooseBooks}(c) = \text{ChooseBook}(c) \star \mathbf{Checkout}(c).$$

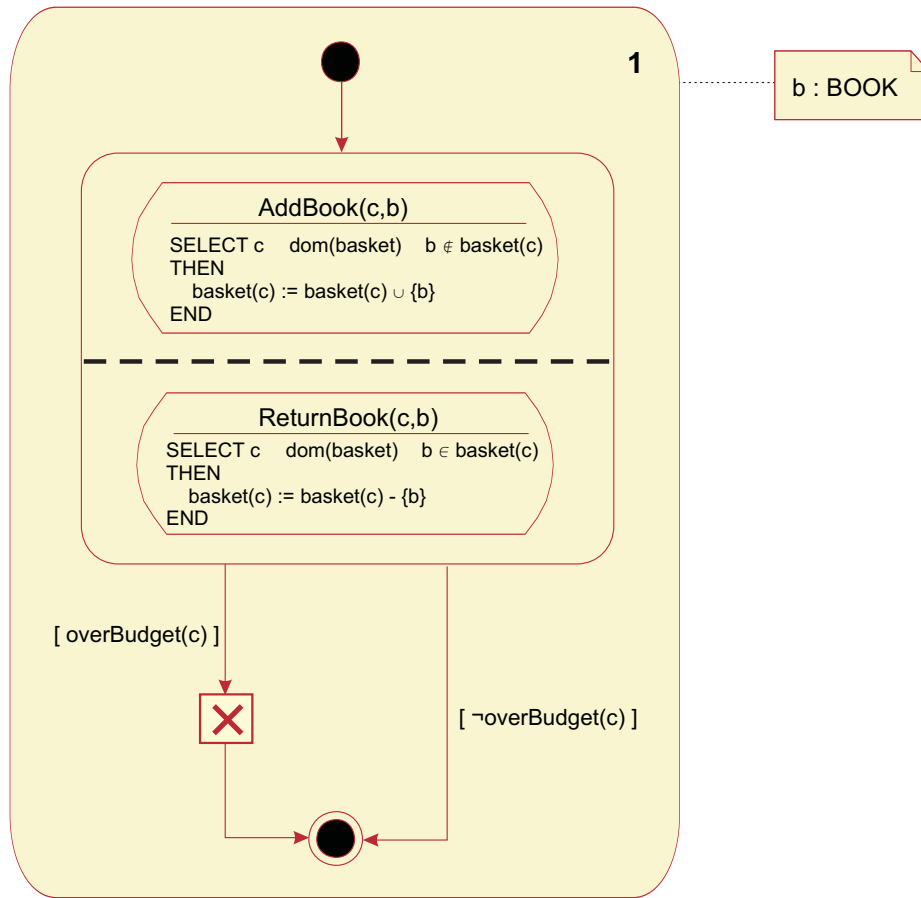

 Figure 6.7: Activity diagram for process *ChooseBooks*

The above process is described by the cyclic diagram presented in Figure 6.7. From the initial state there are two alternative outgoing transitions, leading either to *ChooseBook* or *Checkout*. Selecting the transition that leads to *ChooseBook*, will cause the diagram to return to the initial state. Otherwise, the activity *Checkout* will terminate the *ChooseBooks* state.

The StAC process *ChooseBook* is complex process that uses generalised choice and compensation scoping:

$$\begin{aligned}
 \textit{ChooseBook}(c) \quad = \quad & \boxed{\boxed{b \in \textit{BOOK}} . [(\textit{AddBook}(c, b) \div \textit{ReturnBook}(c, b)); \\
 & \textit{overBudget}(c) \rightarrow \boxtimes]}
 \end{aligned}$$

In Figure 6.8 the outermost state of the diagram has the multiplicity marker 1 that represents a generalised choice over the set *BOOK*. Within this state there exists an implicit compensation scope, so the compensation pair and the reversal within that state will just affect the new compensation task. The innermost state is a sequential process that starts with a compensation pair, and depending on the outcome of *overBudget* it can either invoke the reversal or do nothing. The compensation pair primary action is the activity *AddBook*, and its compensation action is the activity *ReturnBook*. Notice that neither the compensation activity nor the reversal have references to compensation task identifiers. Since this example is specified in StAC the compensation tasks are implicit. As expected an example modelled using compensation activity diagrams can be described using either implicit or explicit compensation tasks, but not both.

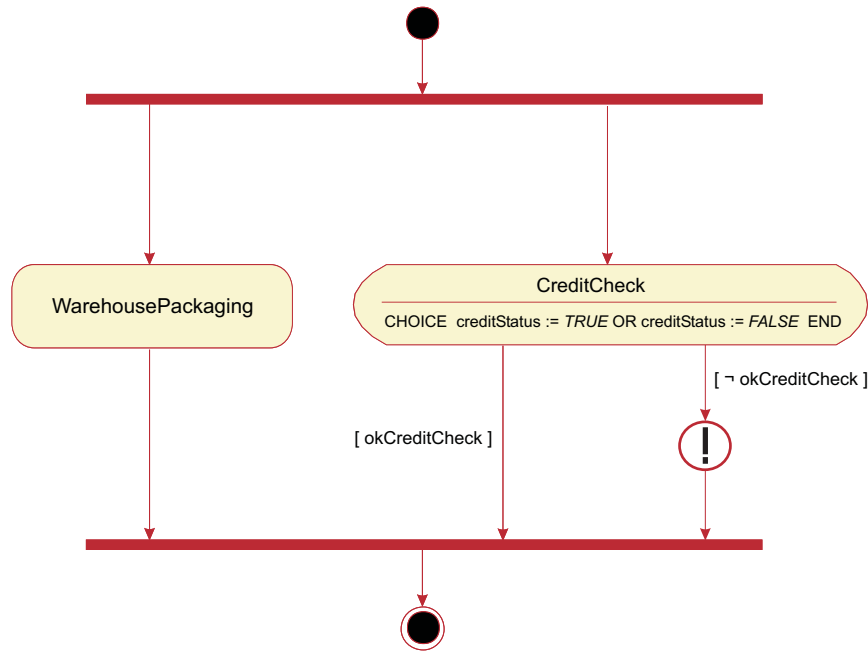

 Figure 6.8: Activity diagram for process *ChooseBook*

6.3.3 Order Fulfillment

This section presents the activity diagram for the process *FulfillOrder* of the order fulfillment example. This process focuses on parallel and early termination:

$$\begin{aligned}
 \textit{FulfillOrder} = \{ & \textit{WarehousePackaging} \parallel \\
 & (\textit{CreditCheck}; \neg \textit{okCreditCheck} \rightarrow \odot) \}
 \end{aligned}$$

The parallel process is represented in the diagram of Figure 6.9 as a fork with two parallel threads: the thread on the right leads to the *WarehousePackaging* state and then finishes; the thread on the left leads to the *CreditCheck* activity and depending on its outcome the variable *okCreditCheck* will be either set to true or false. If the variable *okCreditCheck* is set to false, the early termination will be


 Figure 6.9: Activity diagram for process *FulfillOrder*

invoked causing all the activities within the current state to terminate (with an indeterminate delay). In this diagram the early termination will only affect the *WarehousePackaging* state. We decided not to represent termination scopes in the compensation activity diagrams, instead the scope will be implicitly defined as the state surrounding the early termination.

6.4 Discussion

This chapter proposes an approach to extending UML activity diagrams for modelling StAC specifications. This approach has a strong emphasis on the B notation, as the activities in the diagrams are annotated with AMN statements. The UML/StAC models can provide a “common ground” between formal and non-formal methods users. Nevertheless, some might claim that the emphasis on the B notation is too strong, making it difficult for non-formal methods users to fully understand a system modelled with compensation activity diagrams. Also, within the UML community [SW01] it is argued that UML extensions should conform with the UML standard, avoiding creating yet another UML dialect. Following this view we have started studying the development of compensation activity di-

agrams where the activities annotations are Object Constraint Language (OCL) [WK99] expressions. At the moment we have only specified the arrange meeting example with OCL annotations, but this work is still on its early stages.

Chapter 7

Discussion

7.1 Conclusions

The starting point for this thesis came from a collaboration with IBM concerning the ABC business process modelling tool. Our main contribution was to give a clear meaning to compensation, especially the one of parallel processes. To formalise the concept of compensation a process language called StAC [BF00] was defined. The StAC language was based on the ABC notation and it formalises the ABC concepts. Similarly to ABC, StAC allows parallel and sequential composition, and provides compensation and termination constructs. StAC extends the ABC compensation constructs, because its compensations can be an arbitrary complex process. Conversely, ABC imposes some restrictions on the type of processes allowed in a compensation. An example of these restrictions is that, in ABC, compensations cannot have nested compensations.

Chapter 3 presents the StAC_i language that was originally developed to describe effectively the semantics of StAC. The extended language StAC_i allows a process to have several simultaneous compensation tasks that can be independently reversed or accepted. We have called the simultaneous compensation tasks of StAC_i multiple compensation. The fact that compensation tasks can be identified by their indices means that StAC_i has a clear semantics for compensation. Also, it makes it easier to describe parallel compensation, because a different compensation task can be used for each parallel process. Later, when applying StAC to a few case studies it emerged that the concept of nested compensation was too limited to specify a relevant range of examples. For instance, in the

travel agency example when some trip reservations fail, we only want to reverse the compensations for the reservations that failed, the remaining compensations (related to the successful reservations) should be maintained. Although the travel agency could be modelled in StAC, the specification would be complex and difficult to understand. It became apparent that the StAC_i language could overcome this limitation by using two independent compensation tasks: one for successful reservations and another for failed reservations. We called this compensation mechanism selective compensation. Alternative compensation is another compensation mechanism introduced within multiple compensation, whereby several alternative compensations may be attached to a primary task.

In Chapter 4 we present an operational semantics for StAC_i . The fact that StAC_i (and StAC) is a very specialised language with unusual operators had clear consequences in the semantics. Although, the formalisation of the compensation was fairly straightforward, it required an additional element in the configuration tuple (besides the current process and its state). This additional element is the compensation function that relates each task index to the associated compensation process. When later we introduced termination in the language, the complexity of the formalisation increased considerably. To deal with nested termination we labelled both early termination and termination scoping. These labels are used to ensure that an early termination will only affect the termination block with an identical index. The main hazard of having both termination and compensation is that the occurrence of a termination within a process could prevent a process from saving its compensation. To avoid this problem, we introduced the protected block construct that ensures that a process, within such a block, will be allowed to continue its normal execution after the occurrence of an early termination. We have implemented an animator for StAC_i by encoding the operational rules in Prolog. The animator allows the user to explore the behaviour of a StAC_i specification. The semantics of the StAC language was defined through a translation from StAC to StAC_i terms, by indexing explicitly the hierarchical compensations of StAC. We have used the results presented in [But00] to formally justify the integration of StAC with B by using an operational approach.

In Chapter 5 we explored a strategy for the refinement of StAC_i specifications. Several restrictions were imposed to the type of processes supported, for exam-

ple, termination and complex compensation tasks were not considered. We have devised a way to combine a system of StAC_i processes and its associated B machine into a standard B machine. The result of this combination is a machine that deals explicitly with compensation, and contains variables that represent the implicit states and the compensation function of the StAC_i processes. The fact that the resulting machine is a standard B machine allows the use of the B notion of system refinement to validate refinement between StAC_i specifications. To verify the applicability of this strategy we used the e-bookstore example and defined two specifications for this example: the abstract specification presents a very simplified view of the bookstore and it does not use compensation; the concrete specification replaces each abstract operation by a sequence of concrete operations including compensation operators. We used *Atelier B* to help construct the gluing invariant for the bookstore refinement, and its automatic prover to help prove the proof obligations.

The last chapter describes an approach to using UML to model StAC specifications. The motivation for this work was to make StAC accessible to non-formal methods users. In UML, the specification state can be represented with class diagrams and the specification behaviour with activity diagrams. To represent the compensation and termination operators, the activity diagrams were extended with graphical notations for those operators. Also, the diagram activities (that are the graphical representation for StAC activities) were annotated with AMN statements. Although a tool was not developed, it would be possible to automatically generate from the UML model the correspondent StAC processes and B machine.

7.2 Related Work

In this section we compare our work with related work comprising three different areas: first, with languages that support compensation; second, with other combinations of the B method with process algebras or other temporal ordering of B operations; and last, with other approaches for representing formal languages in UML.

7.2.1 Compensation

The concept of compensation was introduced in transaction processing as a way of recovering from failure. Several models within transaction processing and business processing support some kind of compensation. Yet, most approaches to compensation are quite limited, requiring a compensation to be atomic (it cannot be further decomposed), and imposing the invocation of compensations to be based on system failure. Only ConTracts (see Section 1.2.4) have a structured approach to compensation in some ways similar to StAC. Recently, IBM, Microsoft, and BEA have been developing a business process language called Business Process Execution Language (BPEL) [CGK⁺]. Although some features are not yet fully developed, the first public draft of BPEL was already released. Document [CGK⁺] describes the language current features, and discusses necessary extensions to the language. In BPEL a process can be described either as an executable process or as an abstract process. While the former is an implementation, the latter describes the processes visible behaviour. There can be several executable processes for an abstract process. The language supports similar operators to StAC, such as compensation, concurrency, and sequencing. An advantage of BPEL is that it supports both synchronous and asynchronous communication, while in StAC communication is done by shared global variables.

Next, we summarise the StAC extensions the concept of compensation [CVG⁺02], and at the same time emphasise the distinctions between StAC and other languages that support compensation. We will focus the comparison on the ConTracts and BPEL, as they are the models with most similarities with StAC.

Nested compensations In StAC, a compensation can be any StAC process, and this allows for the definition of nested compensations. In both BPEL and ConTracts, compensation can be a complex process, but nested compensations are not permitted.

Compensation invocation In StAC the invocation of compensation is done by the system, instead of being based on the occurrence of a system failure. Although in ConTracts the invocation of compensation can be done by the system, it has to be made explicitly within a conditional instruction (if the outcome of a step is false, then the compensation is executed), because ConTracts do not

have equivalent instructions to the StAC acceptance and reversal. In BPEL, the invocation of compensation is done by the system, and the language has similar instructions to the acceptance and reversal of StAC.

Multiple compensation The most distinctive feature in StAC is multiple compensation, which allows a process to have several independent compensation tasks. Neither ConTracts nor BPEL covers multiple compensation. Although more restricted than multiple compensation, BPEL intends to support nested scopes of compensation. These scopes will have identifiers so they can be explicitly invoked, yet a reversal cannot invoke a compensation outside its scope.

Formalisation The StAC language, to the best of our knowledge, is the only language that formalises compensation. In Korth *et al.* [KLS90], a compensation is formalised in terms of the properties it has to guarantee. Yet, their approach does not provide a language as StAC does, instead focusing on properties of compensation. BPEL is layered on top of XML (its processes and data are specified in the BPEL dialect of XML), and at the moment BPEL does not have a formal semantics.

7.2.2 Combining B with Process Algebras

There are several methods for combining B with process algebras. We discuss here two of those methods (both combine CSP and B) that are closest to our approach of combining StAC and B. One of the methods, developed by Butler [But00] was already presented in Section 1.2.3.1. We have followed Butler's approach of proving the integration of CSP and B to justify the integration of StAC and B. A different method for combining B and CSP was defined by Treharne & Schneider [TS00, ST02]. The authors propose a method of using CSP to control the behaviour of a B machine, where the consistency of a combined specification is based on the CSP failures-divergences semantics. The advantage of Treharne & Schneider approach is that the two parts of a specification can be verified and refined separately, while in our approach the StAC processes are explicitly embedded in the original B machine. But, StAC does not have a failures-divergences semantics.

Papatsaras & Stoddart [PS02], describe a railway case study that combines event

B and state machines. The authors present two specifications for the railway system: one has a global model, and another has several communicating components. Each component will have a machine that describes its event, and possibly several state machines that describe potential event evolutions. The information contained in those state machines will be explicitly included in the machine. The main result of this work is that it allows a component to be described with a single machine and to be used in different scenarios. Although there are some similarities between Papatsaras & Stoddart's work and our embedding of a STD into a B machine, these works have different aims. While Papatsaras & Stoddart deal with distribution and scalability, we are essentially concerned with compensation.

7.2.3 Representing Formal languages in UML

There are several approaches involving formal languages and UML, most of these focus either on formalising a UML notation or producing a formal specification from a UML model. Our work is more related to the latter, as our aim is to use UML as a graphical representation for StAC specifications, and to generate StAC specifications from those graphical models. Other work with a similar perspective is that of [SB01], [SZ02], and [NB02]. In [SB01], Snook & Butler describe the U2B tool that we already discussed.

Sekerinski & Zurob [SZ02] present a translation from the UML statecharts to B AMN. This translation supports hierarchy, concurrency, and communication. There are not many common points between our UML work and the Sekerinski & Zurob work, nevertheless we could use their translation method to improve our embedding of STDs into a B machine. Sekerinski & Zurob's approach supports arbitrary parallel processes at any level, while we only support the translation of parallel processes as the outermost process.

In [NB02], Ng & Butler present a method for graphically describing CSP in UML: state diagrams are used to model the dynamic behaviour of a system; and class diagrams are used to model the static relationships between the CSP processes and refinement assertions. Also, the method is supported by a tool that generates CSP, which can be output to FDR. Ng & Butler's work has the advantage supporting the representation of refinement.

Last, we compare our UML work with the *precise UML* (pUML) approach [EK99]. The goal of pUML is to develop a precise semantics for UML, and Evans & Clark in [EK99] discuss the use of denotational semantics as a way to formalise some core elements of UML. Our work has different aims from pUML, we use UML as a graphical modelling language for StAC, we do not intend to formalise UML.

7.3 Future Work

In its present state the StAC animator has a restricted functionality, as it only animates the behavioural part of the StAC specification. We would like to evolve the animator in two ways. First, to support the evaluation of the state and of the B AMN expressions that describe the activities. Second, to include a model checker for StAC processes that could verify invariants and assertions.

Continuing the collaboration with IBM, we intend to adapt the compensation activity diagrams to support the use of OCL expressions (instead of B expressions) as the activity annotations. We intend to develop a tool to automatically generate a StAC specification from the UML model.

Also in collaboration with IBM, we will study and formalise the BPEL language through StAC. Our language is appropriate for the formalisation of BPEL because both languages have similar constructs (*e.g.*, compensation, termination, and concurrency), and StAC can abstract the implementation details included in BPEL.

Appendix A

Examples

A.1 Raffle

StAC

$Raffle = SellTickets; DrawOrCancel$
 $SellTickets = \mathbf{DistributeTickets}; SellTicketsAgent$
 $SellTicketsAgent = \parallel_{a \in AGENT} . SellOneTicket(a) \star \mathbf{timeOut}(a)$
 $SellOneTicket(a) = \underline{let} \ U = unsold(a) \ \underline{in}$
 $\quad \parallel_{t \in U} . (\mathbf{SellTicket}(a, t) \div \mathbf{RefundTicket}(a, t))$
 $DrawOrCancel = \mathbf{overThreshold} \rightarrow PerformDraw$
 $\quad \parallel$
 $\quad \neg \mathbf{overThreshold} \rightarrow \boxtimes$
 $PerformDraw = \mathbf{Draw}; \mathbf{DeliverPrize}; \boxtimes$

B Machine

MACHINE *Raffle*

SETS

AGENT;

TICKET

CONSTANTS

threshold

PROPERTIES

threshold $\in \mathbb{N}$

VARIABLES

sold, unsold, winner

DEFINITIONS

overThreshold == $\text{card}(\bigcup(\text{agent}).(\text{agent} \in \text{AGENT} \mid \text{sold}(\text{agent}))) \geq \text{threshold}$

INVARIANT

$\text{sold} \in \text{AGENT} \rightarrow \mathcal{P}(\text{TICKET}) \wedge$
 $\text{unsold} \in \text{AGENT} \rightarrow \mathcal{P}(\text{TICKET}) \wedge$
 $\text{winner} \subseteq \text{TICKET}$
 $\text{card}(\text{winner}) \leq 1$

INITIALISATION

$\text{sold} := \lambda \text{agent} . (\text{agent} \in \text{AGENT} \mid \emptyset) \parallel$
 $\text{unsold} := \lambda \text{agent} . (\text{agent} \in \text{AGENT} \mid \emptyset) \parallel$
 $\text{winner} := \emptyset$

OPERATIONS

DistributeTickets $\hat{=}$

ANY $f \in \text{AGENT} \rightarrow \mathcal{P}(\text{TICKET}) \wedge$
 $\forall a_1, a_2 \in \text{AGENT} . a_1 \neq a_2 \Rightarrow f(a_1) \cap f(a_2) = \emptyset \wedge$
 $\text{union}(\text{ran}(f)) = \text{TICKET}$

THEN

$\text{unsold} := f$

END;

timeOut($a : \text{AGENT}$) $\hat{=}$ *skip*;

SellTicket($a : \text{AGENT}, t : \text{TICKET}$) $\hat{=}$

BEGIN

$\text{unsold}(a) := \text{unsold}(a) - \{t\} \parallel$

$\text{sold}(a) := \text{sold}(a) \cup \{t\}$

END

RefundTicket($a : \text{AGENT}, t : \text{TICKET}$) $\hat{=}$

BEGIN

$\text{unsold}(a) := \text{unsold}(a) \cup \{t\} \parallel$

$\text{sold}(a) := \text{sold}(a) - \{t\}$

END

Draw $\hat{=}$

ANY a, t **WHERE** $a \in AGENT \wedge t \in TICKET \wedge t \in sold(a)$ **THEN**

$winner := \{t\}$

END

DeliverPrize $\hat{=}$ *skip*

END

A.2 E-Bookstore

StAC

$$\begin{aligned}
 Bookstore &= \parallel_{c \in CLIENT} . Client(c) \\
 Client(c) &= \mathbf{Arrive}(c); \\
 &\quad ChooseBooks(c); \\
 &\quad (\mathbf{Quit}(c); \boxtimes \\
 &\quad \parallel \\
 &\quad Pay(c); \boxtimes); \\
 &\quad \mathbf{Exit}(c) \\
 ChooseBooks(c) &= ChooseBook(c) \star \mathbf{Checkout}(c) \\
 ChooseBook(c) &= \parallel_{b \in BOOK} . [(\mathbf{AddBook}(c, b) \div \mathbf{ReturnBook}(c, b)); \\
 &\quad \mathbf{overBudget}(c) \rightarrow \boxtimes] \\
 Pay(c) &= \mathbf{ProcessCard}(c); \neg \mathbf{accepted}(c) \rightarrow \boxtimes
 \end{aligned}$$

B Machine

MACHINE *Bookstore*

SETS

CLIENT;
BOOK

VARIABLES

basket, budget, accepted, price

DEFINITIONS

$overBudget(c) == \sum(b) . (b \in basket(c) \mid price(b)) > budget(c)$

INVARIANT

$basket \in CLIENT \rightarrow \mathcal{P}(BOOK) \wedge$
 $budget \in CLIENT \rightarrow \mathbb{N}_1 \wedge$
 $accepted \in CLIENT \rightarrow \mathbf{BOOL} \wedge$
 $price \in BOOK \rightarrow \mathbb{N}_1 \wedge$
 $dom(basket) = dom(budget) \wedge dom(basket) = dom(accepted)$

INITIALISATION

$basket := \emptyset \parallel$
 $budget := \emptyset \parallel$
 $accepted := \emptyset \parallel$
 $price : (price \in BOOK \rightarrow \mathbb{N}_1)$

OPERATIONS

Arrive($c : CLIENT$) \triangleq
SELECT $c \notin dom(basket)$ **THEN**
 ANY a **WHERE** $a \in \mathbb{N}_1$ **THEN**
 $basket := basket \cup \{c \mapsto \emptyset\} \parallel$
 $budget := budget \cup \{c \mapsto a\} \parallel$
 $accepted := accepted \cup \{c \mapsto \text{FALSE}\}$
 END
END;

AddBook($c : CLIENT, b : BOOK$) \triangleq
SELECT $c \in dom(basket) \wedge b \notin basket(c)$ **THEN**
 $basket(c) := basket(c) \cup \{b\}$
END;

ReturnBook($c : CLIENT, b : BOOK$) \triangleq
SELECT $c \in dom(basket) \wedge b \in basket(c)$ **THEN**
 $basket(c) := basket(c) - \{b\}$
END;

ProcessCard($c : CLIENT$) \triangleq
SELECT $c \in dom(basket)$ **THEN**
 CHOICE
 $accepted(c) := \text{TRUE}$
 OR
 $accepted(c) := \text{FALSE}$
 END
END;

Checkout($c : CLIENT$) $\hat{=}$
SELECT $c \in dom(basket)$ **THEN** *skip* **END**;

Quit($c : CLIENT$) $\hat{=}$
SELECT $c \in dom(basket)$ **THEN** *skip* **END**;

Exit¹($c : CLIENT$) $\hat{=}$
SELECT $c \in dom(basket)$ **THEN**
 $basket := \{c\} \triangleleft basket$ ||
 $budget := \{c\} \triangleleft budget$ ||
 $accepted := \{c\} \triangleleft accepted$
END

END

¹The expression $s \triangleleft r$ represents anti-restriction of r by s , also known as domain subtraction.

A.3 Order Fulfillment

StAC

$$\begin{aligned}
 ACME &= \text{AcceptOrder} \div \text{RestockOrder}; \\
 &\quad \text{FulfillOrder}; \\
 &\quad \text{okFulfillOrder} \rightarrow \checkmark \\
 &\quad \square \\
 &\quad \neg \text{okFulfillOrder} \rightarrow \boxtimes \\
 \text{FulfillOrder} &= \{ \text{WarehousePackaging} \\
 &\quad \parallel \\
 &\quad \text{CreditCheck}; (\neg \text{okCreditCheck} \rightarrow \odot) \} \\
 \text{WarehousePackaging} &= (\text{BookCourier} \div \text{CancelCourier}) \parallel \text{PackOrder} \\
 \text{PackOrder} &= \underline{\text{let}} \ O = \text{order} \ \underline{\text{in}} \ \parallel_{i \in O} . \text{PackItem}(i) \div \text{UnpackItem}(i)
 \end{aligned}$$

B Machine

MACHINE *ACME*

SETS

ITEM;
ORDER

VARIABLES

stock, order, courierBooking, creditStatus, packaging

DEFINITIONS

$\text{inStock}(i) == \{ \text{item} \mid \text{item} \in i \wedge \text{stock}(i) > 0 \} = i;$
 $\text{okFulfillOrder} == \text{creditStatus};$
 $\text{okCreditCheck} == \text{creditStatus}$

INVARIANT

$\text{stock} \in \text{ITEM} \rightarrow \mathbb{N} \wedge$
 $\text{order} \in \mathcal{P}(\text{ITEM}) \wedge$
 $\text{courierBooking} \in \mathbf{BOOL} \wedge$
 $\text{creditStatus} \in \mathbf{BOOL} \wedge$
 $\text{packaging} \in \text{order} \rightarrow \mathbf{BOOL}$

INITIALISATION

$stock : (stock \in ITEM \rightarrow \mathbb{N}) \wedge$
 $order := \emptyset \wedge$
 $courierBooking := \text{FALSE} \wedge$
 $creditStatus := \text{FALSE} \wedge$
 $packaging := \emptyset$

OPERATIONS

AcceptOrder $\hat{=}$

ANY $items$ **WHERE** $items \subseteq ITEM \wedge inStock(items)$ **THEN**
 $stock := stock \triangleleft \lambda(item). (item \in items \mid stock(item) - 1) \parallel$
 $order := items \parallel$
 $packaging := \lambda(item). (item \in items \mid \text{FALSE})$
END;

RestockOrder $\hat{=}$

BEGIN
 $stock := stock \triangleleft \lambda(item). (item \in items \mid stock(item) + 1)$
END;

CreditCheck $\hat{=}$

BEGIN
CHOICE
 $creditStatus := \text{TRUE}$
OR
 $creditStatus := \text{FALSE}$
END
END;

BookCourier $\hat{=}$

BEGIN
 $courierBooking := \text{TRUE}$
END;

```
CancelCourier  $\hat{=}$   
  BEGIN  
    courierBooking := FALSE  
  END;  
  
PackItem(i : ITEM)  $\hat{=}$   
  BEGIN  
    packaging(i) := TRUE  
  END;  
  
UnpackItem(i : ITEM)  $\hat{=}$   
  BEGIN  
    packaging(i) := FALSE  
  END  
  
END
```

A.4 Travel Agency

StAC

$$\begin{aligned}
\textit{TravelAgency} &= \parallel c \in \textit{CLIENT} . \mathbf{Request}(c); \textit{TripReservation}(c) \\
\textit{TripReservation}(c) &= \textit{GetItinerary}(c); \\
&\quad \mathbf{VerifyCreditCard}(c); \\
&\quad \mathbf{accepted}(c) \rightarrow \textit{ContinueReservation}(c) \\
&\quad \parallel \\
&\quad \neg \mathbf{accepted}(c) \rightarrow \textit{QuitReservation}(c) \\
\textit{GetItinerary}(c) &= (\textit{SelectFlight}(c) \parallel \textit{SelectCar}(c) \parallel \textit{SelectHotel}(c)) \star \mathbf{EndSelection}(c) \\
\textit{SelectFlight}(c) &= \parallel f \in \textit{FLIGHT} . \mathbf{SelfFlight}(c, f) \\
\textit{SelectCar}(c) &= \parallel a \in \textit{CAR} . \mathbf{SelCar}(c, a) \\
\textit{SelectHotel}(c) &= \parallel h \in \textit{HOTEL} . \mathbf{SelHotel}(c, h) \\
\textit{ContinueReservation}(c) &= \mathbf{ConfirmOrder}(c); \textit{MakeReservation}(c) \\
&\quad \parallel \\
&\quad \mathbf{CancelOrder}(c); \textit{QuitReservation}(c) \\
\textit{MakeReservation}(c) &= \\
&\quad (\textit{FlightReservations}(c) \parallel \textit{CarReservations}(c) \parallel \textit{HotelReservations}(c)); \\
&\quad \neg \mathbf{okReservations}(c) \rightarrow \textit{ContactClient}(c) \\
&\quad \parallel \\
&\quad \mathbf{okReservations}(c) \rightarrow \textit{EndTrip}(c) \\
\textit{FlightReservations}(c) &= \underline{\textit{let}} R = \textit{flights}(c) \underline{\textit{in}} \parallel f \in R . \textit{FlightReservation}(c, f) \\
\textit{FlightReservation}(c, f) &= \\
&\quad \mathbf{ReserveFlight}(c, f); \\
&\quad \mathbf{flightIsReserved}(c, f) \rightarrow (\textit{skip} \div_{S(c)} (\mathbf{CancelFlight}(c, f) \parallel \mathbf{ClearFlight}(c, f))) \\
&\quad \parallel \\
&\quad \neg \mathbf{flightIsReserved}(c, f) \rightarrow (\textit{skip} \div_{F(c)} \mathbf{ClearFlight}(c, f)) \\
\textit{CarReservations}(c) &= \underline{\textit{let}} C = \textit{cars}(c) \underline{\textit{in}} \parallel a \in C . \textit{CarReservation}(c, a) \\
\textit{CarReservation}(c, a) &= \\
&\quad \mathbf{ReserveCar}(c, a); \\
&\quad \mathbf{carIsReserved}(c, a) \rightarrow (\textit{skip} \div_{S(c)} (\mathbf{CancelCar}(c, a) \parallel \mathbf{ClearCar}(c, a))) \\
&\quad \parallel \\
&\quad \neg \mathbf{carIsReserved}(c, a) \rightarrow (\textit{skip} \div_{F(c)} \mathbf{ClearCar}(c, a)) \\
\textit{HotelReservations}(c) &= \underline{\textit{let}} H = \textit{hotels}(c) \underline{\textit{in}} \parallel h \in H . \textit{HotelReservation}(c, h)
\end{aligned}$$

$$\begin{aligned}
HotelReservation(c, h) &= \\
&\quad \mathbf{ReserveHotel}(c, h); \\
&\quad \mathbf{hotelIsReserved}(c, h) \rightarrow (skip \div_{S(c)} (\mathbf{CancelHotel}(c, h) \parallel \mathbf{ClearHotel}(c, h))) \\
&\quad \parallel \\
&\quad \neg \mathbf{hotelIsReserved}(c, h) \rightarrow (skip \div_{F(c)} \mathbf{ClearHotel}(c, h)) \\
ContactClient(c) &= \mathbf{Continue}(c); \boxtimes_{F(c)}; GetItinerary(c); MakeReservation(c) \\
&\quad \parallel \\
&\quad \mathbf{Quit}(c); QuitReservation(c) \\
QuitReservation(c) &= (\boxtimes_{S(c)} \parallel \boxtimes_{F(c)}); \mathbf{RemoveClient}(c) \\
EndTrip(c) &= \boxtimes_{S(c)} \parallel \boxtimes_{F(c)}
\end{aligned}$$

B Machine

MACHINE *TravelAgency*

SETS

$CLIENT$;
 $FLIGHT$;
 CAR ;
 $HOTEL$;
 $CARD = \{visa, mastercard, switch, none\}$

VARIABLES

$clients, flights, cars, hotels, creditCard, accepted,$
 $flightReservations, carReservations, hotelReservations$

DEFINITIONS

$okReservations(c) ==$
 $flights(c) \subseteq \{f \mid f \in FLIGHT \wedge c \in flightReservations(f)\} \wedge$
 $cars(c) \subseteq \{a \mid a \in CAR \wedge c \in carReservations(a)\} \wedge$
 $hotels(c) \subseteq \{h \mid h \in HOTEL \wedge c \in hotelReservations(h)\};$
 $flightIsReserved(c, f) == c \in flightReservations(f);$
 $carIsReserved(c, a) == c \in carReservations(a);$
 $hotelIsReserved(c, h) == c \in hotelReservations(h);$

INVARIANT

$$\begin{aligned}
& clients \subseteq CLIENT \wedge \\
& flights \in clients \rightarrow \mathcal{P}(FLIGHT) \wedge \\
& cars \in clients \rightarrow \mathcal{P}(CAR) \wedge \\
& hotels \in clients \rightarrow \mathcal{P}(HOTEL) \wedge \\
& creditCard \in clients \rightarrow CARD \wedge \\
& accepted \in clients \rightarrow \mathbf{BOOL} \wedge \\
& flightReservations \in FLIGHT \rightarrow \mathcal{P}(CLIENT) \wedge \\
& carReservations \in CAR \rightarrow \mathcal{P}(CLIENT) \wedge \\
& hotelReservations \in HOTEL \rightarrow \mathcal{P}(CLIENT) \wedge
\end{aligned}$$
INITIALISATION

$$\begin{aligned}
& clients := \emptyset \parallel \\
& flights := \emptyset \parallel \\
& cars := \emptyset \parallel \\
& hotels := \emptyset \parallel \\
& creditCard := \emptyset \parallel \\
& accepted := \emptyset \\
& flightReservations := \lambda f. (f \in FLIGHT \mid \emptyset) \parallel \\
& carReservations := \lambda a. (a \in CAR \mid \emptyset) \parallel \\
& hotelReservations := \lambda h. (h \in HOTEL \mid \emptyset)
\end{aligned}$$
OPERATIONS

$$\begin{aligned}
& \mathbf{Request}(c : CLIENT) \triangleq \\
& \quad \mathbf{SELECT} \ c \notin clients \ \mathbf{THEN} \\
& \quad \quad clients := clients \cup \{c\} \parallel \\
& \quad \quad flights := flights \cup \{c \mapsto \emptyset\} \parallel \\
& \quad \quad cars := cars \cup \{c \mapsto \emptyset\} \parallel \\
& \quad \quad hotels := hotels \cup \{c \mapsto \emptyset\} \parallel \\
& \quad \quad creditCard := clients \cup \{c \mapsto none\} \parallel \\
& \quad \quad accepted := clients \cup \{c \mapsto \mathbf{FALSE}\} \\
& \quad \mathbf{END};
\end{aligned}$$

```

VerifyCreditCard( $c : CLIENT$ )  $\hat{=}$ 
  SELECT  $c \in clients$  THEN
    ANY  $cc$  WHERE  $cc \in CARD - \{none\}$  THEN
       $creditCard(c) := cc \parallel$ 
       $accepted(c) := bool(cc \neq switch)$ 
    END
  END;

SelFlight( $c : CLIENT, f : FLIGHT$ )  $\hat{=}$ 
  SELECT  $c \in clients \wedge f \notin flights(c)$  THEN
     $flights(c) := flights(c) \cup \{f\}$ 
  END;

SelCar( $c : CLIENT, a : CAR$ )  $\hat{=}$ 
  SELECT  $c \in clients \wedge a \notin cars(c)$  THEN
     $cars(c) := cars(c) \cup \{a\}$ 
  END;

SelHotel( $c : CLIENT, h : HOTEL$ )  $\hat{=}$ 
  SELECT  $c \in clients \wedge h \notin hotels(c)$  THEN
     $hotels(c) := hotels(c) \cup \{h\}$ 
  END;

ReserveFlight( $c : CLIENT, f : FLIGHT$ )  $\hat{=}$ 
  SELECT  $c \in clients \wedge f \in flights(c) \wedge$ 
     $c \notin flightReservations(f)$  THEN
    CHOICE
       $flightReservations(f) := flightReservations(f) \cup \{c\}$ 
    OR
       $flightReservations(f) := flightReservations(f)$ 
    END
  END;

```

ReserveCar($c : CLIENT, a : CAR$) $\hat{=}$
SELECT $c \in clients \wedge a \in cars(c) \wedge$
 $c \notin carReservations(a)$ **THEN**
CHOICE
 $carReservations(a) := flightReservations(a) \cup \{c\}$
OR
 $carReservations(a) := flightReservations(a)$
END
END;

ReserveHotel($c : CLIENT, h : HOTEL$) $\hat{=}$
SELECT $c \in clients \wedge h \in hotels(c) \wedge$
 $c \notin hotelReservations(h)$ **THEN**
CHOICE
 $hotelReservations(h) := hotelReservations(h) \cup \{c\}$
OR
 $hotelReservations(h) := hotelReservations(h)$
END
END;

ClearFlight($c : CLIENT, f : FLIGHT$) $\hat{=}$
SELECT $c \in clients \wedge f \in flights(c)$ **THEN**
 $flights(c) := flights(c) - \{f\}$
END;

ClearCar($c : CLIENT, a : CAR$) $\hat{=}$
SELECT $c \in clients \wedge a \in cars(c)$ **THEN**
 $cars(c) := cars(c) - \{a\}$
END;

ClearHotel($c : CLIENT, h : HOTEL$) $\hat{=}$
SELECT $c \in clients \wedge h \in hotels(c)$ **THEN**
 $hotels(c) := hotels(c) - \{h\}$
END;

CancelFlight($c : CLIENT, f : FLIGHT$) $\hat{=}$
SELECT $c \in flightReservations(f)$ **THEN**
 $flightReservations(f) := flightReservations(f) - \{c\}$
END;

CancelCar($c : CLIENT, a : CAR$) $\hat{=}$
SELECT $c \in carReservations(a)$ **THEN**
 $carReservations(a) := carReservations(a) - \{c\}$
END;

CancelHotel($c : CLIENT, h : HOTEL$) $\hat{=}$
SELECT $c \in hotelReservations(h)$ **THEN**
 $hotelReservations(h) := hotelReservations(h) - \{c\}$
END;

ConfirmOrder($c : CLIENT$) $\hat{=}$
SELECT $c \in client$ **THEN skip END;**

CancelOrder($c : CLIENT$) $\hat{=}$
SELECT $c \in client$ **THEN skip END;**

RemoveClient($c : CLIENT$) $\hat{=}$
SELECT $c \in client$ **THEN**
 $clients := clients - \{c\}$ ||
 $flights := \{c\} \triangleleft flights$ ||
 $cars := \{c\} \triangleleft cars$ ||
 $hotels := \{c\} \triangleleft hotels$ ||
 $creditCard := \{c\} \triangleleft clients$ ||
 $accepted := \{c\} \triangleleft clients$
END;

EndSelection($c : CLIENT$) $\hat{=}$
SELECT $c \in clients$ **THEN skip END;**

Quit($c : CLIENT$) $\hat{=}$
SELECT $c \in clients$ **THEN skip END;**

Continue($c : CLIENT$) $\hat{=}$
SELECT $c \in clients$ **THEN skip END**

END

A.5 Arrange Meeting

StAC

$ArrangeMeeting = CheckRoom; CheckTeam; Decide$

$CheckRoom = (\text{SelectAvailableDates} \div_{CF} \text{ConfirmRoom})$
 $\div_{CL} \text{CancelRoom}$

$CheckTeam = \parallel_{t \in TEAM} . (\text{SuggestDates}(t) \div_{CF} \text{ConfirmDate}(t))$
 $\div_{CL} \text{CancelDates}(t)$

$Decide = \neg \text{emptyDates} \rightarrow \text{SelectDate}; \boxtimes_{CF}$
 \parallel
 $\text{emptyDates} \rightarrow \boxtimes_{CL}; \boxtimes_{CF}$

B Machine

MACHINE *Meeting*

SETS

$TEAM;$
 $DATE$

VARIABLES

$diary, availableDates, selectedDate, bookings$

DEFINITIONS

$interDates == \bigcap (member). (member \in TEAM \mid diary(member))$

INVARIANT

$diary \in TEAM \rightarrow \mathcal{F}(DATE) \wedge$
 $availableDates \subseteq DATE \wedge$
 $selectedDate \in DATE \wedge$
 $bookings \in \mathcal{F}(DATE)$

INITIALISATION

$diary := \lambda team. (team \in TEAM \mid \emptyset) \parallel$
 $availableDates := \emptyset \parallel$
 $selectedDate : (selectedDate \in DATE) \parallel$
 $bookings : (bookings \in \mathcal{F}(DATE))$

OPERATIONS

SelectAvailableDates $\hat{=}$

ANY *dates* **WHERE** $dates \subseteq DATE - bookings$ **THEN**
 $availableDates := dates \parallel$
 $bookings := bookings \cup dates$
END;

BookRoom $\hat{=}$

BEGIN
 $bookings := (bookings - availableDates) \cup \{selectedDate\}$
END;

CancelRoom $\hat{=}$

BEGIN
 $bookings := bookings - availableDates$
END;

SuggestDates($m : TEAM$) $\hat{=}$

ANY *dates* **WHERE** $dates \subseteq availableDates$ **THEN**
 $diary(m) := dates$
END;

ConfirmDate($m : TEAM$) $\hat{=}$

BEGIN
 $diary(m) := \{selectedDate\}$
END;

CancelDates($m : TEAM$) $\hat{=}$

BEGIN
 $diary(m) := \emptyset$
END;

```
SelectDate  $\hat{=}$   
  SELECT interDates  $\neq \emptyset$  THEN  
    ANY date WHERE date  $\in$  DATE  $\wedge$  date  $\in$  interDates THEN  
      selectedDate := date  
    END  
  END  
END
```

Appendix B

Semantics - Auxiliary Functions

B.1 Labelling Function

Function \mathbb{L} labels the compensation block and the early termination instruction.

$\mathbb{L}(A, k)$	$= A$
$\mathbb{L}(null, k)$	$= null$
$\mathbb{L}(b \rightarrow P, k)$	$= b \rightarrow \mathbb{L}(P, k)$
$\mathbb{L}(rec(N), k)$	$= rec(N)$
$\mathbb{L}(P; Q, k)$	$= \mathbb{L}(P, k); \mathbb{L}(Q, k)$
$\mathbb{L}(P \parallel Q, k)$	$= \mathbb{L}(P, k) \parallel \mathbb{L}(Q, k)$
$\mathbb{L}(\parallel x \in X . P_x, k)$	$= \parallel x \in X . \mathbb{L}(P_x, k)$
$\mathbb{L}(P \sqcap Q, k)$	$= \mathbb{L}(P, k) \sqcap \mathbb{L}(Q, k)$
$\mathbb{L}(\sqcap x \in X . P_x, k)$	$= \sqcap x \in X . \mathbb{L}(P_x, k)$
$\mathbb{L}(\underline{let} X = e \underline{in} P_X, k)$	$= \underline{let} X = e \underline{in} \mathbb{L}(P_X, k)$
$\mathbb{L}(\odot, k)$	$= \odot_k$
$\mathbb{L}(\{P\}, k)$	$= \{ \mathbb{L}(P, l) \}_{\langle \top, l \rangle}$ where l is a “fresh” index
$\mathbb{L}(P \div_i Q, k)$	$= \mathbb{L}(P, k) \div_i \mathbb{L}(Q, k)$
$\mathbb{L}(\boxtimes_i, k)$	$= \boxtimes_i$
$\mathbb{L}(\boxdot_i, k)$	$= \boxdot_i$
$\mathbb{L}(J \triangleright i, k)$	$= J \triangleright i$
$\mathbb{L}(P _v, k)$	$= \mathbb{L}(P, k) _v$

B.2 Normalisation Functions

B.2.1 *norm*

N1	$norm(null; P, \sigma)$	$= norm(P, \sigma)$
N2	$norm(null \parallel P, \sigma)$	$= norm(P, \sigma)$
N3	$norm(P \parallel null, \sigma)$	$= norm(P, \sigma)$
N4	$norm(null \sqcap P, \sigma)$	$= norm(P, \sigma)$
N5	$norm(P \sqcap null, \sigma)$	$= norm(P, \sigma)$
N6	$norm(b \rightarrow P, \sigma)$	$= norm(P, \sigma), \text{ if } b(\sigma) = true$
N7	$norm(b \rightarrow P, \sigma)$	$= null, \text{ if } b(\sigma) = false$
N8	$norm(\parallel x \in \emptyset . P_x, \sigma)$	$= null$
N9	$norm(\sqcap x \in \emptyset . P_x, \sigma)$	$= null$
N10	$norm(\parallel x \in X . null, \sigma)$	$= null$
N11	$norm(\sqcap x \in X . null, \sigma)$	$= null$
N12	$norm(A, \sigma)$	$= A$
N13	$norm(null, \sigma)$	$= null$
N14	$norm(\boxtimes_i, \sigma)$	$= \boxtimes_i$
N15	$norm(\boxdot_i, \sigma)$	$= \boxdot_i$
N16	$norm(J \triangleright i, \sigma)$	$= J \triangleright i$
N17	$norm(rec(N), \sigma)$	$= rec(N)$
N18	$norm(P; Q, \sigma)$	$= norm(P, \sigma); norm(Q, \sigma), \text{ if } P \neq null \wedge P \neq \odot_k$
N19	$norm(P \parallel Q, \sigma)$	$= norm(P, \sigma) \parallel norm(Q, \sigma), \text{ if } P \neq null \wedge Q \neq null$
N20	$norm(\parallel x \in X . P_x, \sigma)$	$= \parallel x \in X . norm(P_x, \sigma), \text{ if } X \neq \emptyset \wedge P_x \neq null$
N21	$norm(P \sqcap Q, \sigma)$	$= norm(P, \sigma) \sqcap norm(Q, \sigma), \text{ if } P \neq null \wedge Q \neq null$
N22	$norm(\sqcap x \in X . P_x, \sigma)$	$= \sqcap x \in X . norm(P_x, \sigma), \text{ if } X \neq \emptyset \wedge P_x \neq null$
N23	$norm(P \div_i Q, \sigma)$	$= norm(P, \sigma) \div_i Q, \text{ if } P \neq null$
N24	$norm(\underline{let} X = e \underline{in} P_X, \sigma)$	$= norm(P_{e(\sigma)}, \sigma), \text{ if } P_X \neq null$
N25	$norm(\{null\}_{\langle v, k \rangle}, \sigma)$	$= null$
N26	$norm(\odot_k; P, \sigma)$	$= \odot_k$
N27	$norm(\odot_k, \sigma)$	$= \odot_k$
N28	$norm(\{P\}_{\langle v, k \rangle}, \sigma)$	$= \{norm(P, \sigma)\}_{\langle v, k \rangle}, \text{ if } P \neq null \wedge v \neq 0$
N29	$norm(\{P\}_{\langle 0, k \rangle}, \sigma)$	$= \{norm(terminate(P), \sigma)\}_{\langle 0, k \rangle}, \text{ if } P \neq null$
N30	$norm(null _v, \sigma)$	$= null$
N31	$norm(P _v, \sigma)$	$= norm(P) _v, \text{ if } P \neq null$

B.2.2 *normalised*

$normalised(A)$	$= true$
$normalised(null)$	$= true$
$normalised(b \rightarrow P)$	$= false$
$normalised(rec(N))$	$= true$
$normalised(\boxtimes_i)$	$= true$
$normalised(\boxdot_i)$	$= true$
$normalised(J \triangleright i)$	$= true$
$normalised(P; Q)$	$= normalised(P) \wedge normalised(Q) \wedge$ $P \neq null \wedge P \neq \odot_k$
$normalised(P \parallel Q)$	$= normalised(P) \wedge normalised(Q) \wedge$ $P \neq null \wedge Q \neq null$
$normalised(\parallel_{x \in X} . P_x, \sigma)$	$= normalised(P_x) \wedge X \neq \emptyset \wedge P_x \neq null$
$normalised(P \parallel\!\!\! \parallel Q)$	$= normalised(P) \wedge normalised(Q) \wedge$ $P \neq null \wedge Q \neq null$
$normalised(\parallel_{x \in X} . P_x)$	$= normalised(P_x) \wedge X \neq \emptyset \wedge P_x \neq null$
$normalised(P \div_i Q)$	$= normalised(P)$
$normalised(\underline{let} X = e \text{ in } P_X)$	$= false$
$normalised(\odot_k)$	$= true$
$normalised(\{P\}_{\langle v, k \rangle})$	$= normalised(P) \wedge P \neq null \wedge v \neq 0$
$normalised(\{P\}_{\langle 0, k \rangle})$	$= normalised(P) \wedge terminate(P) = P \wedge$ $P \neq null$
$normalised(P _v)$	$= normalised(P) \wedge P \neq null$

B.2.3 *terminate*

$terminate(P; Q)$	$= terminate(P)$
$terminate(P \parallel Q)$	$= terminate(P) \parallel terminate(Q)$
$terminate(\{P\}_{\langle n, k \rangle})$	$= \{terminate(P)\}_{\langle n, k \rangle}$
$terminate(P _{false})$	$= null$
$terminate(P _{true})$	$= P _{true}$
$terminate(P)$	$= null, \quad \text{otherwise}$

Appendix C

StAC Animator

C.1 Operational Rules

```
/* R1 */ trans(conf(act(A),C), act(A), conf(null,C)).

/* R2 */ trans(conf(rec([N|ListV]),C), B, conf(P1,C1)) :-
    equation([N|ListV],P),
    normalisation(P,Pi),
    trans(conf(Pi,C), B, conf(P1,C1)).

/* R2 */ trans(conf(rec([N|ListV]),C), B, conf(P1,C1)) :-
    equation([N|List],P),
    List \== ListV,
    normalisation(P,Pi),
    instantiate(Pi,List,ListV,Pj),
    trans(conf(Pj,C), B, conf(P1,C1)).

/* R3 */ trans(conf(seq(P,Q),C), B, conf(seq(P1,Q),C1)) :-
    trans(conf(P,C), B, conf(P1,C1)).

/* R4 */ trans(conf(par(P,Q),C), B, conf(par(P1,Q),C1)) :-
    trans(conf(P,C), B, conf(P1,C1)).

/* R5 */ trans(conf(par(Q,P),C), B, conf(par(Q,P1),C1)) :-
    trans(conf(P,C), B, conf(P1,C1)).
```



```

/* R6 */ trans(conf(gpar(V,X,P),C), B, conf(par(gpar(V,X1,P),Pr),C1)) :-
    member(E,X), select(E,X,X1),
    instantiate(P,[V],[E],Pi),
    trans(conf(Pi,C), B, conf(Pr,C1)).

/* R7 */ trans(conf(choice(P,_),C), B, conf(P1,C1)) :-
    trans(conf(P,C), B, conf(P1,C1)).

/* R8 */ trans(conf(choice(_,P),C), B, conf(P1,C1)) :-
    trans(conf(P,C), B, conf(P1,C1)).

/* R9 */ trans(conf(gchoice(V,X,P),C), B, conf(Pr,C1)) :-
    member(E,X),
    instantiate(P,[V],[E],Pi),
    trans(conf(Pi,C), B, conf(Pr,C1)).

/* R10 */ trans(conf(pair(P,Q,I),C), B, conf(pair(P1,Q,I),C1)) :-
    trans(conf(P,C), B, conf(P1,C1)),
    P1 \== null.

/* R11 */ trans(conf(pair(null,Q,I,_),C), pairT(I), conf(null,R)) :-
    push_comp(C,I,Q,R).

/* R12*/ trans(conf(compensate(J),C), compensate(J), conf(P1,C1)) :-
    comp_seq(C,J,P1),
    clear_comp_l(C,J,C1).

/* R13*/ trans(conf(commit(J),C), commit(J), conf(null,C1)) :-
    clear_comp_l(C,J,C1).

/* R14 */ trans(conf(merge(J,I),C), merge(J,I), conf(null,C2)) :-
    merge_par(C,J,P1), clear_comp_l(C,J,C1),
    push_comp(C1,I,P1,C2).

/* R15 */ trans(conf(protected(P,_),C), B, conf(protected(P1,true),C1)) :-
    trans(conf(P,C), B, conf(P1,C1)).

```

```

/* R16 */ trans(conf(exit(I),C), exit(I), conf(null,C)).

/* R17 */ trans(conf(block(P,true,I),C), exit(I), conf(block(P1,N,I),C)) :-
    trans(conf(P,C), exit(I), conf(P1,C)),
    random(0,3,N).

/* R18 */ trans(conf(block(P,true,I),C), B, conf(block(P1,true,I),C1)) :-
    trans(conf(P,C), B, conf(P1,C1)),
    B \== exit(I).

/* R19 */ trans(conf(block(P,N,I),C), B, conf(block(P1,M,I),C1)) :-
    trans(conf(P,C), B, conf(P1,C1)),
    N \== true, M is N-1, M @>= 0.

/* R20 */ trans(conf(block(P,0,I),C), B, conf(block(P1,0,I),C1)) :-
    trans(conf(P,C), B, conf(P1,C1)).

```

C.2 Normalisation Functions

C.2.1 normalisation

```

normalisation(P,P) :- normalised(P).
normalisation(P,Pr) :- \+(normalised(P)),
    norm(P,Pi),
    normalisation(Pi,Pr).

```

C.2.2 norm

```

norm(seq(null,P),Pr) :- norm(P,Pr).
norm(par(null,P),Pr) :- norm(P,Pr).
norm(par(P,null),Pr) :- norm(P,Pr).
norm(choice(null,P),Pr) :- norm(P,Pr).
norm(choice(P,null),Pr) :- norm(P,Pr).
norm(gpar(_,_,null),null).
norm(gchoice(_,_,null),null).
norm(cond(true,P),Pr) :- norm(P,Pr).
norm(cond(false,_),null).

```

```

norm(gpar(_, [], _), null).
norm(gchoice(_, [], _), null).
norm(act(A), act(A)).
norm(null, null).
norm(compensate(J), compensate(J)).
norm(commit(J), commit(J)).
norm(merge(J, I), merge(J, I)).
norm(call(N), call(N)).
norm(seq(P, Q), seq(Pr, Qr)) :-
    P \== null, P \= exit(_), norm(P, Pr), norm(Q, Qr).
norm(par(P, Q), par(Pr, Qr)) :-
    P \== null, Q \== null, norm(P, Pr), norm(Q, Qr).
norm(gpar(V, X, P), gpar(V, X, Pr)) :-
    X \== [], P \== null, norm(P, Pr).
norm(choice(P, Q), choice(Pr, Qr)) :-
    P \== null, Q \== null, norm(P, Pr), norm(Q, Qr).
norm(gchoice(V, X, P), gchoice(V, X, Pr)) :-
    X \== [], P \== null, norm(P, Pr).
norm(pair(P, Q, J), pair(Pr, Q, J)) :- norm(P, Pr).
norm(block(null, _, _), null).
norm(seq(exit(I), _), exit(I)).
norm(exit(I), exit(I)).
norm(block(P, N, I), block(Pr, N, I)) :-
    P \== null, N @>= 1, norm(P, Pr).
norm(block(P, 0, I), block(Pr, 0, I)) :-
    P \== null, terminate(P, Pi), norm(Pi, Pr).
norm(protected(null, _), null).
norm(protected(P, V), protected(Pr, V)) :- norm(P, Pr).

```

C.2.3 normalised

```

normalised(seq(null, _)) :- fail.
normalised(par(null, _)) :- fail.
normalised(par(_, null)) :- fail.
normalised(choice(null, _)) :- fail.
normalised(choice(_, null)) :- fail.
normalised(gpar(_, _, null)) :- fail.
normalised(gchoice(_, _, null)) :- fail.

```

```

normalised(cond(true,_)) :- fail.
normalised(cond(false,_)) :- fail.
normalised(gpar(_,[],_)) :- fail.
normalised(gchoice(_,[],_)) :- fail.
normalised(act(_)) :- true.
normalised(null) :- true.
normalised(compensate(_)) :- true.
normalised(commit(_)) :- true.
normalised(merge(_,_)) :- true.
normalised(call(_)) :- true.
normalised(seq(P,Q)) :-
    P \== null, P \= exit(_),
    normalised(P), normalised(Q).
normalised(par(P,Q)) :-
    P \== null, Q \== null,
    normalised(P), normalised(Q).
normalised(gpar(_,X,P)) :- X \== [], P \== null, normalised(P).
normalised(choice(P,Q)) :-
    P \== null, Q \== null,
    normalised(P), normalised(Q).
normalised(gchoice(_,X,P)) :- X \== [], P \== null, normalised(P).
normalised(pair(P,_,_,_)) :- normalised(P).
normalised(block(null,_,_)) :- fail.
normalised(seq(exit(_),_)) :- fail.
normalised(exit(_)) :- true.
normalised(block(P,N,_)) :- P \== null, N @>= 1, normalised(P).
normalised(protected(null,_)) :- fail.

normalised(protected(P,_)) :- P \== null, normalised(P).

```

C.2.4 terminate

```

terminate(seq(P,_),Pr) :- terminate(P,Pr).
terminate(par(P,Q),par(Pr,Qr)) :- terminate(P,Pr), terminate(Q,Qr).
terminate(block(P,N,I),block(Pr,N,I)) :- terminate(P,Pr).
terminate(protected(_,false),null).
terminate(protected(P,true),protected(P,true)).
terminate(_,null).

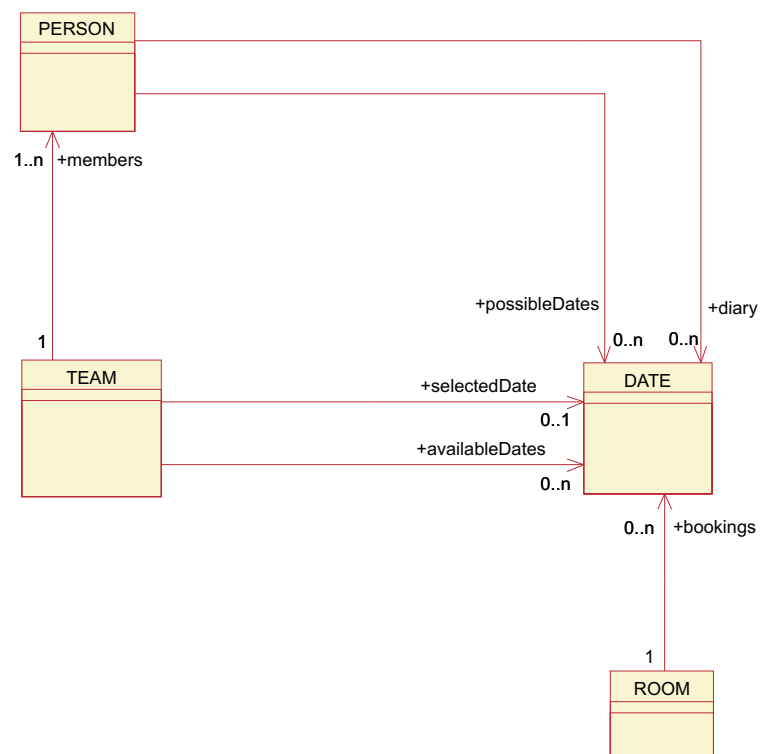
```

Appendix D

Examples modelled in UML

D.1 Arrange Meeting

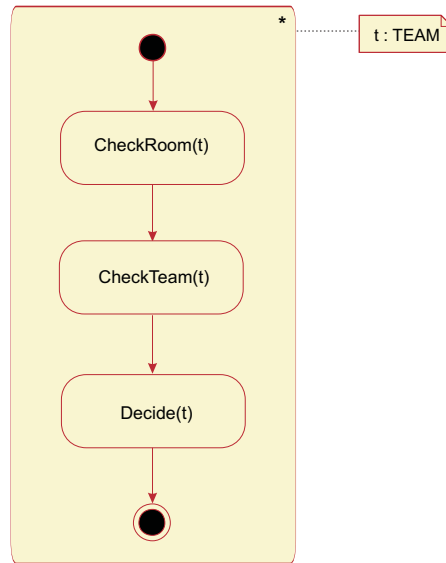
D.1.1 Class Diagram



D.1.2 Compensation Activity Diagrams

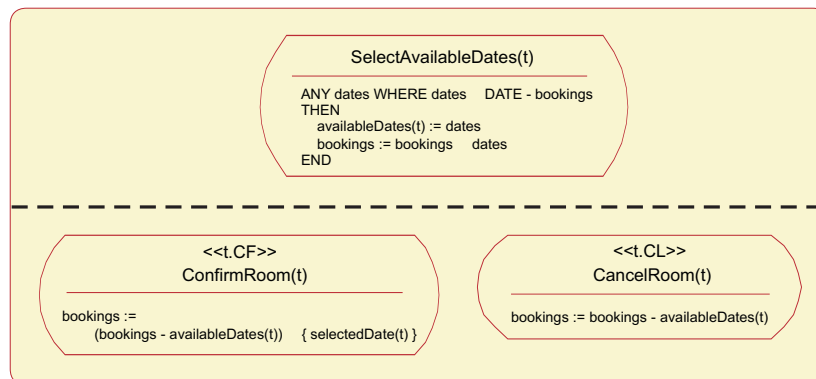
ArrangeMeeting Process

$ArrangeMeeting = \parallel t \in TEAM . CheckRoom(t); CheckTeam(t); Decide(t)$

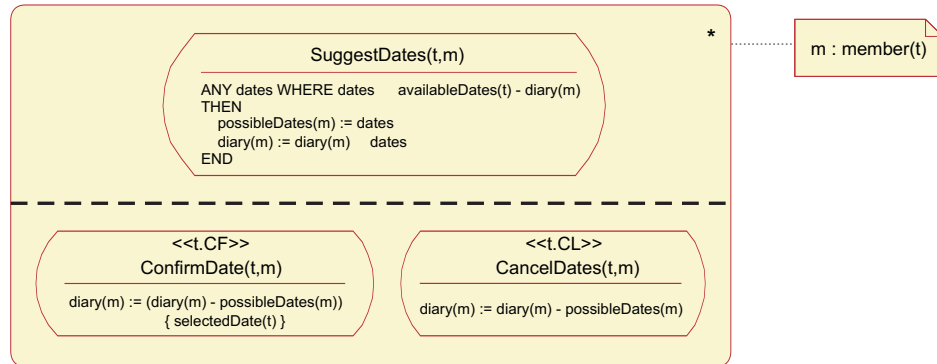


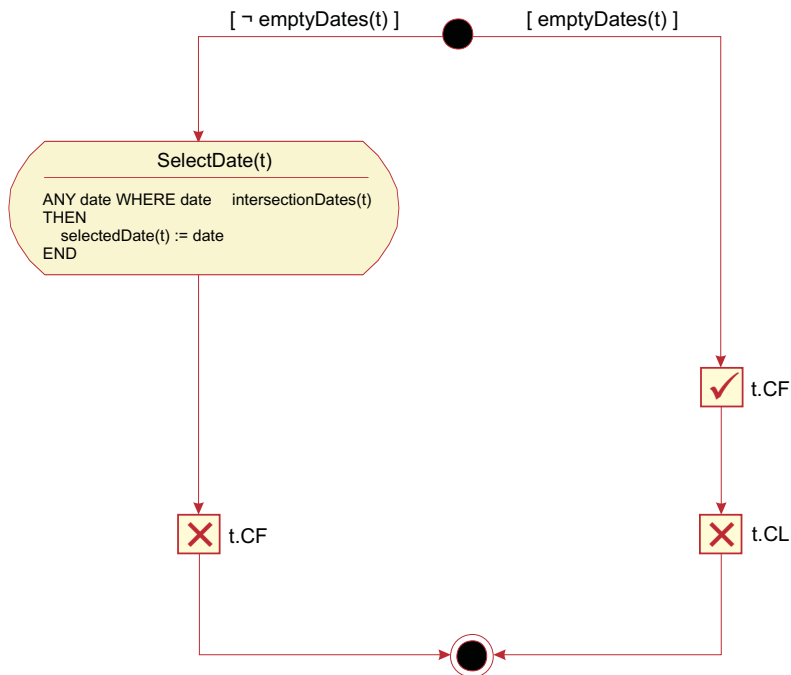
CheckRoom Process

$CheckRoom(t) = (\text{SelectAvailableDates}(t) \div_{CF(t)} \text{ConfirmRoom}(t)) \div_{CL(t)} \text{CancelRoom}(t)$



CheckTeam Process

$$\begin{aligned}
\text{CheckTeam}(t) = & \text{let } M = \text{member}(t) \text{ in} \\
& \parallel m \in M . (\text{SuggestDates}(t, m) \div_{CF(t)} \text{ConfirmDate}(t, m)) \\
& \div_{CL(t)} \text{CancelDates}(t, m)
\end{aligned}$$
**Decide Process**

$$\begin{aligned}
\text{Decide}(t) = & \neg \text{emptyDates}(t) \rightarrow \text{SelectDate}(t); \boxtimes_{CF(t)} \\
& \parallel \\
& \text{emptyDates}(t) \rightarrow \boxtimes_{CL(t)}; \boxdot_{CF(t)}
\end{aligned}$$


D.2 E-Bookstore

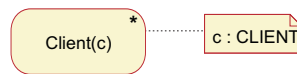
D.2.1 Class Diagram



D.2.2 Compensation Activity Diagrams

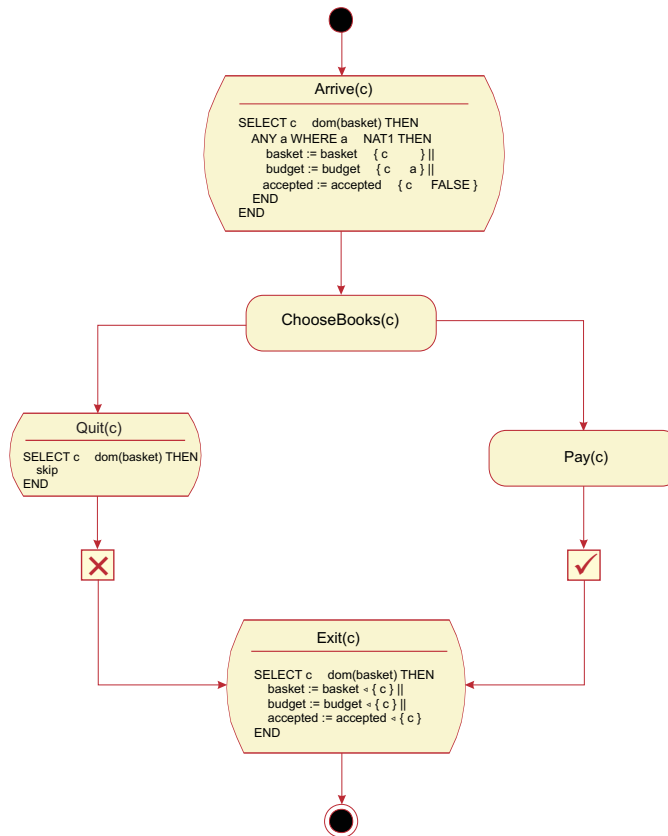
Bookstore Process

$Bookstore = \parallel_{c \in CLIENT} . Client(c)$



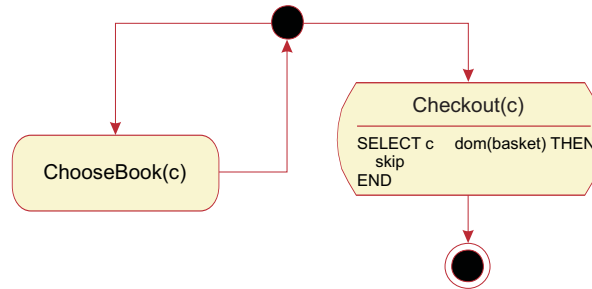
Client Process

$Client(c) = \mathbf{Arrive}(c); ChooseBooks(c); ((\mathbf{Quit}(c); \boxtimes) \parallel (\mathbf{Pay}(c); \boxplus)); \mathbf{Exit}(c)$

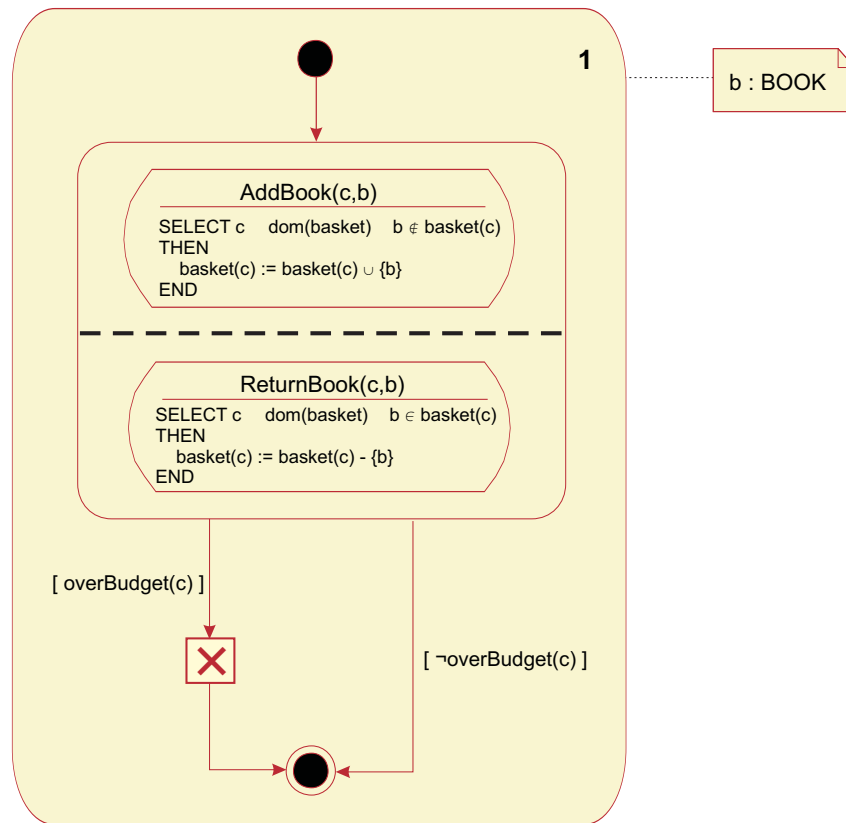


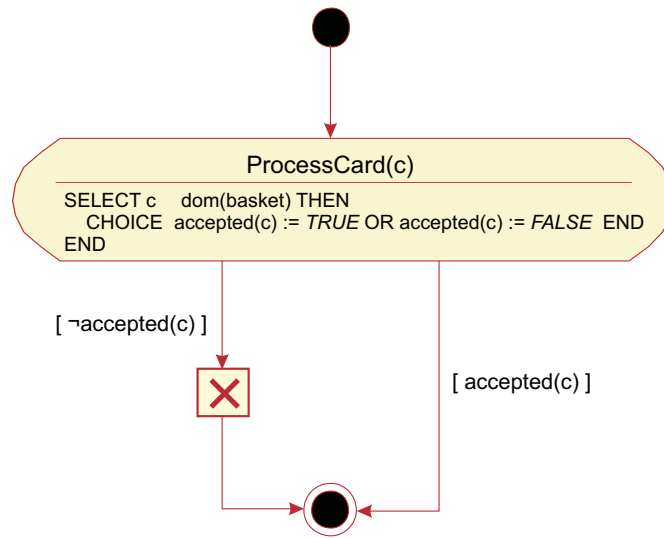
***ChooseBooks* Process**

$$ChooseBooks(c) = ChooseBook(c) \star Checkout(c)$$

***ChooseBook* Process**

$$ChooseBook(c) = \coprod_{b \in BOOK} . [(AddBook(c, b) \div ReturnBook(c, b)); \text{overBudget}(c) \rightarrow \boxtimes]$$



***Pay* Process**
$$Pay(c) = \mathbf{ProcessCard}(c); \neg\mathbf{accepted}(c) \rightarrow \boxtimes$$


D.3 Order Fulfillment

D.3.1 Class Diagram



D.3.2 Compensation Activity Diagrams

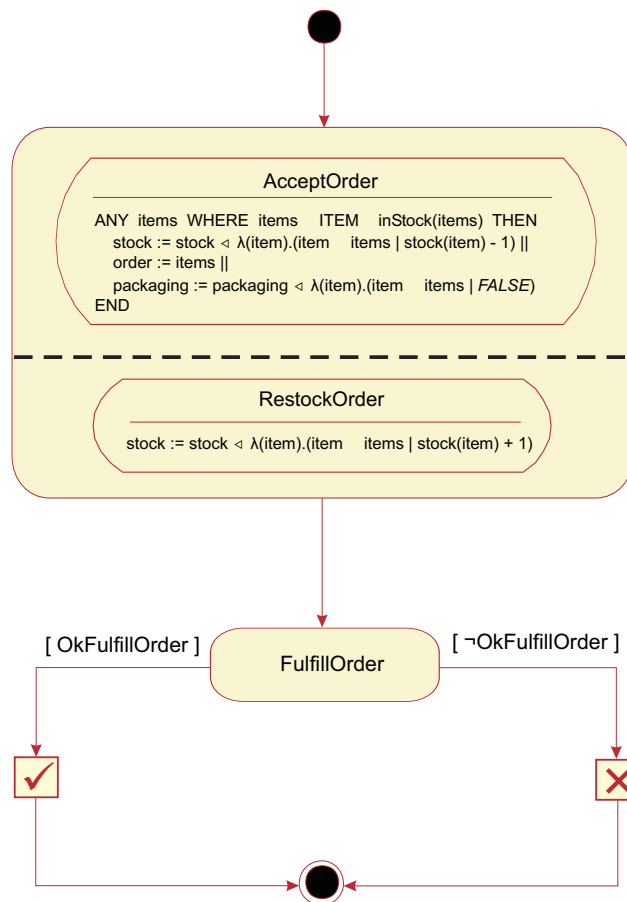
ACME Process

ACME = **AcceptOrder** ÷ **RestockOrder**; *FulfillOrder*;

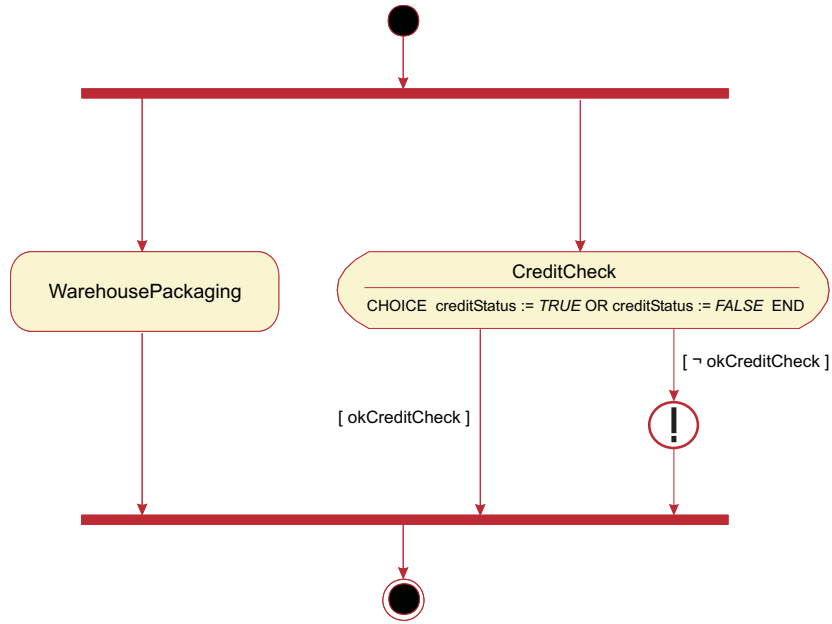
okFulfillOrder → ✓

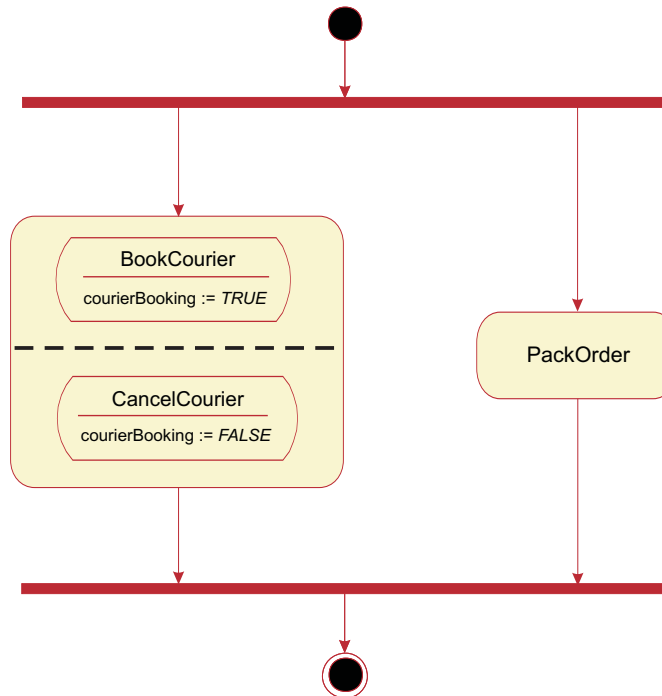
□

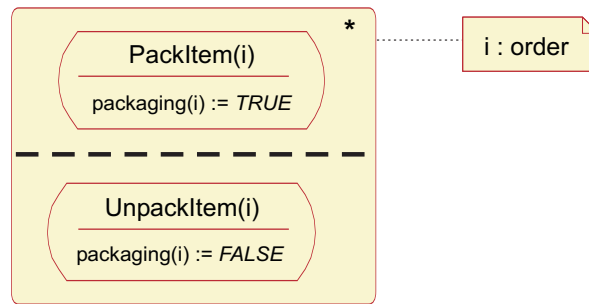
¬**okFulfillOrder** → ✗



***FulfillOrder* Process**

$$FulfillOrder = \{ WarehousePackaging \parallel (CreditCheck; \neg okCreditCheck \rightarrow \odot) \}$$
***WarehousePackaging* Process**

$$WarehousePackaging = (BookCourier \div CancelCourier) \parallel PackOrder$$


PackOrder Process
$$PackOrder = \underline{let} \ O = order \ \underline{in} \ \parallel \ i \in O . \mathbf{PackItem}(i) \div \mathbf{UnpackItem}(i)$$


Bibliography

- [AB02] J. Augusto and M. Butler. Some observations about using SPIN and STeP to verify StAC specifications. Technical report, Department of Electronics and Computer Science, University of Southampton, October 2002.
- [Abr96] J.R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.
- [AN02] J. Arlow and I. Neustad. *UML and the Unified Process*. Addison-Wesley, 2002.
- [Ate98] Steria – Technologies de L’Information. *Atelier-B User Manual*, 1998.
- [Bac93] R. Back. Refinement of parallel and reactive programs. *Lecture Notes for the Summer School on Program Design Calculi*, 836:73–92, 1993.
- [BF00] M. Butler and C. Ferreira. A process compensation language. In *Integrated Formal Methods(IFM’2000)*, volume 1945 of *LNCS*, pages 61 – 76. Springer-Verlag, 2000.
- [Bto96] B-Core (UK) Ltd. *B-Toolkit User’s Manual*, 1996.
- [But92] M. Butler. *A CSP Approach to Action Systems*. DPhil thesis, University of Oxford, 1992.
- [But97] M. Butler. An approach to the design of distributed systems with B AMN. In D. Till J. Bowen, M. Hinchey, editor, *10th International Conference of Z Users (ZUM’97)*, volume *LNCS 1212*, pages 223–241. Springer-Verlag, 1997.

- [But00] M. Butler. csp2B: A practical approach to combining CSP and B. *Formal Aspects of Computing*, 12:182–198, 2000.
- [But02a] M. Butler. On the use of data refinement in the development of secure communications systems. *Formal Aspects of Computing*, To appear, 2002.
- [But02b] M. Butler. A system-based approach to the formal development of embedded controllers for a railway. *Design Automation for Embedded Systems*, 6(4):355–366, 2002.
- [BV94] R. Back and J. Von Wright. Trace refinement of action systems. *LNCS*, 836:367–384, 1994.
- [CGK⁺] F. Curbera, Y. Golland, J. Klein, F. Leymann, D. Roller, S. Thatte, and S. Weerawarana. Business process execution language for web services. <http://www-106.ibm.com/developerworks/library/ws-bpel/>.
- [Cri84] Flaviu Cristian. Correct and robust programs. *IEEE Transactions on Software Engineering*, 10(2), March 1984.
- [CVG01a] M. Chessell, D. Vines, and C. Griffin. An introduction to compensation with business process beans. Technical report, Transaction Processing Design and New Technology Development Group, IBM UK Laboratories, August 2001.
- [CVG⁺01b] M. Chessell, D. Vines, C. Griffin, V. Green, and K. Warr. Business process beans: System design and architecture document. Technical report, Transaction Processing Design and New Technology Development Group, IBM UK Laboratories, January 2001.
- [CVG⁺02] M. Chessell, D. Vines, C. Griffin, M. Butler, C. Ferreira, and P. Henderson. Extending the concept of transaction compensation. *IBM Systems Journal*, 41(4):743–758, 2002.
- [Dav78] C. Davies, Jr. Data processing spheres of control. *IBM Systems Journal*, 17(2):179–198, 1978.

- [EGLT76] K. Eswaran, J. Gray, R. Lorie, and I. Traiger. The notions of consistency and predicate locks in a database system. *Communications of the ACM*, 19(11), 1976.
- [EJL⁺99] H. Eertink, W. Janssen, P.O. Luttighuis, W. Teeuw, and C. Vissers. A business process design language. In *FM'99 – Formal Methods*, LNCS 1708, pages 76–95. Springer-Verlag, 1999.
- [EK99] A. Evans and S. Kent. Core meta-modelling semantics of UML: The pUML approach. In R. France and B. Rumpe, editors, *The Unified Modeling Language (UML'99)*, volume 1723 of *LNCS*, pages 140–155. Springer-Verlag, 1999.
- [Elm92] A. Elmagarmid, editor. *Database Transaction Models for Advanced Applications*. Morgan Kaufmann Publishers, 1992.
- [FDR97] Formal Systems (Europe) Ltd. *Failures-Divergences Refinement: FDR2 User Manual*, 1997.
- [FS00] M. Fowler and K. Scott. *UML Distilled - A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley, second edition, 2000.
- [GHS02] M. Gogolla and B. Henderson-Sellers. Analysis of UML stereotypes within the UML metamodel. In S. Cook J.-M. Jezequel, H. Hussmann, editor, *5th Conference of Unified Modeling Language (UML'2002)*. Springer-Verlag, 2002.
- [GMS87] H. Garcia-Molina and K. Salem. Sagas. In *Proceedings of ACM SIGMOD*, pages 249–259, 1987.
- [GR93] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, 1993.
- [Gra81] J. Gray. The transaction concept: Virtues and limitations. In *Proceedings of 7th VLBD*, pages 144–154, 1981.
- [Har87] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.

- [Har88] D. Harel. On visual formalisms. *Communications of the ACM*, 31(5):514–530, 1988.
- [HC01] M. Hammer and J. Champy. *Reengineering the Corporation*. Nicholas Brealey Publishing Ltd, 2001.
- [HJ98] C.A.R. Hoare and H. Jifeng. *Unifying Theories of Programming*. Prentice Hall, 1998.
- [HN96] D. Harel and A. Naamad. The STATEMATE semantics of state-charts. *ACM Transactions on Software Engineering and Methodology*, 5(5):293–333, 1996.
- [Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.
- [HR83] T. Haerder and A. Reuter. Principles of transaction-oriented database recovery. *ACM Computing Survey*, 15(4), 1983.
- [JJV97] W. Janssen, H. Jonkers, and J. Verhoosel. What makes business processes special? An evaluation framework for modelling languages and tools in business process redesign. In *Proceedings 2nd CAiSE/IFIP 8.1*, 1997.
- [JK97] S. Jajodia and L. Kerschberg, editors. *Advanced Transaction Models and Architectures*. Kluwer Academic Publishers, 1997.
- [JMMS98] W. Janssen, R. Mateescu, S. Mauw, and J. Springintveld. Verifying business processes with SPIN. In *Proceedings 4th International SPIN Workshop*, 1998.
- [KLS90] H. Korth, E. Levy, and A. Silberschatz. A formal approach to recovery by compensating transactions. In *16th VLDB Conference*, Brisbane, Australia, 1990.
- [KM95] S. King and C. Morgan. Exits in the refinement calculus. *Formal Aspects of Computing*, 7(1):54–76, 1995.
- [LAB⁺01] M. Leuschel, L. Adhianto, M. Butler, C. Ferreira, and L. Mikhailov. Animation and model checking of CSP and B. In *VCL'2001 - Workshop on Verification and Computational Logic*, 2001.

- [Lan96] K. Lano. *The B Language and Method: A Guide to Practical Formal Development*. Formal Approaches to Computing and Information Technology. Springer-Verlag, 1996.
- [Leu01] M. Leuschel. Design and implementation of the high-level specification language CSP(PL) in Prolog. In *Proceedings of the PADL'01*, 2001.
- [LR00] F. Leymann and D. Roller. *Production Workflow: Concepts and Techniques*. Prentice Hall PTR, 2000.
- [LS02] H. Ledang and J. Souquieres. Contributions for modelling UML state-charts in B. In K. Sere M. Butler, L. Petre, editor, *Integrated Formal Methods 2002*, volume LNCS 2335. Springer-Verlag, 2002.
- [MH99] R. Monson-Haefel. *Enterprise Java Beans*. O'Reilly, 1999.
- [Mil89] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.
- [Mos82] J. Moss. Nested transactions and reliable distributed computing. In *IEEE Symposium on Reliability in Distributed Software and Database Systems*. IEEE CS Press, 1982.
- [MRS⁺00] T. Mikalsen, I. Rouvellou, S. Sutton Jr., Stefan Tai, M. Chessell, C. Griffin, and D. Vines. Transactional business process servers: Definition and requirements. In *OOPSLA'2000 - Business Object Component Workshop*, 2000.
- [MS99] E. Meyer and J. Souquieres. A systematic approach to transform OMT diagrams to a B specification. In *Formal Methods (FM'99)*, LNCS 1708, pages 875–895. Springer-Verlag, 1999.
- [NB02] M. Ng and M. Butler. Tool support for visualising CSP in UML. In *4th International Conference on Formal Engineering Methods (ICFEM 2002)*, 2002.
- [Oul95] M. Ould. *Business Processes Modelling and Analysis for Re-engineering and Improvement*. Wiley, 1995.

- [PHWA98] K. Phalp, P. Henderson, R. Walters, and G. Abeysinghe. Rolenact: Role-based enactable models of business processes. *Information and Software Technology*, 40, 1998.
- [Plo81] G. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University, Computer Science Department, September 1981.
- [PS02] A. Papatsaras and B. Stoddart. Global and communicating state machine models in event driven B: A simple railway case study. In D. Bert et al., editor, *Formal Specification and Development in Z and B (ZB 2002)*, pages 458–476. Springer-Verlag, 2002.
- [RJB99] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modelling Language Reference Manual*. Addison-Wesley, 1999.
- [Ros98] A. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, 1998.
- [RSS97] A. Reuter, K. Schneider, and F. Schwenkreis. ConTracts revisited. In S. Jajodia and L. Kerschberg, editors, *Advanced Transaction Models and Architectures*. Kluwer Academic Publishers, 1997.
- [SB01] C. Snook and M. Butler. Using UML class diagrams for constructing B specifications. Technical report, DSSE Group, Department of Electronics and Computer Science, University of Southampton, 2001.
- [Sch00] S. Schneider. *Concurrent and Real-time Systems: The CSP Approach*. Wiley, 2000.
- [Sch01] S. Schneider. *The B-Method: An Introduction*. Palgrave, 2001.
- [Sek98] E. Sekerinski. Graphical design of reactive systems. In D. Bert, editor, *B'98: The 2nd International B Conference*, volume LNCS 1393, pages 182–197. Springer-Verlag, 1998.
- [ST02] S. Schneider and H. Treharne. Communicating B Machines. In D. Bert et al., editor, *Formal Specification and Development in Z and B (ZB 2002)*, pages 416–435. Springer-Verlag, 2002.

- [SW01] A. Schleicher and B. Westfechtel. Beyond stereotyping: Metamodeling approaches for the UML. In R. Sprague Jr., editor, *34th Annual Hawaii International Conference System Sciences (HICSS'93)*. IEEE Computer Society, 2001.
- [SZ02] E. Sekerinski and R. Zurob. Translating statecharts to B. In K. Sere M. Butler, L. Petre, editor, *Integrated Formal Methods 2002*, volume LNCS 2335. Springer-Verlag, 2002.
- [Thu97] V. Thurner. A formally founded description technique for business processes. Technical report, Department of Computer Science, Technical University of Munich, December 1997.
- [TS00] H. Treharne and S. Schneider. How to drive a B Machine. In J. Bowen et al., editor, *Formal Specification and Development in Z and B (ZB 2000)*. Springer-Verlag, 2000.
- [WK99] J. Warmer and A. Kleppe. *The Object Constraint Language: Precise Modelling with UML*. Addison-Wesley, 1999.
- [WR92] H. Wachter and A. Reuter. The ConTract model. In A. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*. Morgan Kaufmann Publishers, 1992.
- [WS92] G. Weikum and H. Schek. Concepts and applications of multilevel transactions and open nested transactions. In A. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*. Morgan Kaufmann Publishers, 1992.
- [ZJ93] P. Zave and M. Jackson. Conjunction as composition. *ACM Transactions on Software Engineering and Methodology*, 2(4), 1993.
- [ZJ96] P. Zave and M. Jackson. Where do operations come from? An approach to multiparadigm specification technique. *IEEE Transactions on Software Engineering*, 17(2), 1996.