SPECIFICATION AND VERIFICATION OF COMMUNICATION CONSTRAINTS FOR INTEROPERABLE TRANSACTIONS

A HH NGU*
School of Computer Science & Engineering, University of New South Wales,
P.O.Box 1, Kensington
NSW 2033, Australia

and

R MEERSMAN & H WEIGAND
InfoLab, Tilburg University, P.O.Box 90153
5000 LE Tilburg, The Netherlands

Recieved November 15, 1993
Revised March 20, 1994

ABSTRACT

The specification of communication behavior is fundamental in developing interoperable transactions. In particular, the temporal ordering of messages exchanged between different communicating agents must be declaratively specified and verified in order to guarantee consistency of data in the various component systems. This paper shows that by expressing communication constraints in propositional temporal logic, the tableau method can be applied to construct a dependency graph. If the specification is correct, this method guarantees that all possible execution paths satisfying the specification will be generated. The declarative specification and verification of communication constraints in interoperable transactions is demonstrated using the classic business trip. It is argued that the specification formalism provides an improvement over the Flexible Transaction Model.

Keywords: Interoperable Transactions, Communication Constraints, Temporal Propositional Logic, Dependency Graph, Specification, Verification

1. Introduction

Many organizations today are experiencing explosive growth in the number of separate autonomous information systems and the overall volume of electronically stored data. This gives increasing urgency to the problem of developing convenient and effective mechanisms to provide uniform access to a multitude of heterogeneous systems which may hold overlapping and/or inconsistent data. The technology which provides concepts, methodologies, techniques and tools for the efficient and

*This work was supported by three months ITK fellowship at Tilburg University, The Netherlands.

transparent access across these diverse systems is called *interoperability*. A common definition of a Cooperative Information System (CIS) is a system which is interoperable. In our interpretation, a CIS consists of a finite set of autonomous, communicating *agents*, and a (static) description of their *interoperation behavior* as a finite set of *obligations*, called *communication constraints*, expressed as formulas in a suitable logic.

Agents by themself are (only) capable of emitting, receiving and storing *messages*, and of performing *actions* (in this paper we emulate actions by emitting additional messages). Agents therefore in particular may have internal *states*.

Autonomy of the agents is achieved by making them aware of each other *only* through the presence of the obligations (on which they therefore have to "agree") which define goals, i.e. functional constraints on the results of their interoperation. In this first approach we assume these constraints to be statically specified, i.e. the type of interoperation behavior is not dependent on the actual state of the agents. We also concentrate on constraints (obligations) that describe primarily the synchronization aspects of communication rather than the conceptual content (semantics) of the messages.

Behavior of the CIS as a whole is described by means of *interoperable transactions* that are consistent with the communication constraints in a precise manner, described below, based on their possible execution sequences. Each interoperable transaction is defined as a logical unit of work involving different autonomous agents. A subtransaction is a logical unit of work of the interoperable transaction. Each subtransaction consists of one or more logical unit of operations. An operation is a logical unit of work (atomic) at a particular autonomous agent. Since it is possible for an interoperable transaction to contain another interoperable transaction, the three levels of distinction could be generalized into one concept, *transaction*. The difference between the three levels is more in the granularity of the logical unit of work and the number of autonomous agents being involved.

An important aspect of an interoperable transaction is the specification of the coordination between operations in different autonomous component agents. Coordination can be modeled as a communication process. Each agent participating in the process has certain obligations to fulfill depending on the role it is playing. The obligations of agents here are mainly the messages each agent agrees to provide in the context of cooperation within a particular interoperable transaction. The set of agents exchange messages in the order specified by the communication constraints. These include sequences of actions that must occur together, conflicting actions, and triggers which cause other actions to be taken when a request begins or ends. Each process exchanges messages (operations) in accordance with mutually agreed obligations or constraints. Through these messages, true cooperation within an interoperable transaction is achieved without violating the autonomy of the agents. It is for that reason that we emphasize the communication, and focus on the synchronization of messages rather than operations.

Various formalisms have been proposed for the specification of communication between objects. In [LM92], the concept of `activity` is used to specify the communication behavior and interactions in the framework of object oriented databases. The notion of `contract`, which emphasizes the mutual obligations of participants in the communication process, has also been proposed [HHG90, Wei92]. Under this formalism, violation of the obligations which lead to either compensations or sanctions is also specified. In the SYNTHESIS system [SK92], the `script` facility is used to describe the flow of messages among a collection of activities and generate an execution model. In the later versions of TAXIS [Nix84], scripts are used to specify the communication protocol between objects and the user. In the `event` model by King and Mcleod [KM85], a finite state diagram is used to model communication paths. Recently, the flexible transaction model[ELLR90] has been proposed for the specification of interoperable database systems. Special purpose languages such as IPL (Interbase Parallel Language [BCC+92]) have been developed to facilitate the specification of subtransaction dependencies so as to guarantee the correct operation semantics of interoperable transaction execution (for example, an operation must be executed completely or not at all).

There is no shortage of formalisms for the specification of communication between objects. We believe that the existing formalisms such as `activity`, `contract`, `flexible transaction model` or variations of them are adequate for modeling the communication behavior between objects in an interoperable transaction. However, declarative specification and verification of the communication behavior are unexplored in the proposed formalisms. The objective of this paper is to investigate the use of Propositional Temporal Logic (PTL) [MP81] for the specification and verification of communication constraints in the design of interoperable transactions.

Traditionally, PTL has been used mainly in specifying and verifying communication protocols in computer networks and concurrent systems. A "model graph" or dependency graph is constructed by tableau methods [Wol82], whose paths correspond to possible execution sequences if the original specification is correct. The execution path provides the information as to the execution order of operations within an interoperable transaction. For example, it determines when an operation is to be executed serially or concurrently. This tableau method has been shown to be sound and complete by Wolper, i.e. if the specifications are consistent, then the method will find all sequences satisfying the specifications. Otherwise, if there is a contradiction in the specifications, the method fails to generate any sequence.

The use of PTL for specifying communication constraints in CIS transaction results in a declarative specification which is clear and simple. A great variety of properties can be expressed without explicit reference to the time parameter. The graph generation algorithm that we have adopted for PTL is based on a well developed theory and is polynomial in time. Other alternatives such as using the first order temporal logic would result in a much more complex graph generation algorithm which is undecidable.

In section 3, we show how the communication process of a classic interoperable transaction in the business domain can be declaratively specified. Section 4 describes how a dependency graph can be generated for the business trip and used to verify the communication constraints. Our main contribution is the application of the dependency graph construction algorithm for verifying the execution sequences of transactions (operations or subtransactions) in the different autonomous systems at the design stage, thus allowing the designer to be informed about the capability (for example, the execution sequence which leads to deadlock can be prevented) of the interoperable transaction *before* it is being implemented. Our second contribution is the specification of interoperable transaction using PTL.

A verified interoperable transaction can be reused through binding with different agents having the similar coordination protocol, and participation in the composition of other interoperable transactions. The details of communication constraints reusability will be discussed in our future paper.

2. Concepts for communication specification

We are not going to propose yet another formalism for communication specification. Instead we will concentrate on the core concepts required for specifying a communication process in an interoperable system.

In any communication process it is necessary to identify the set of participating agents, the possible message types provided by each agent to support its role as a communicating agent and its flow of information (for example, the indication of sending and receiving sources of a message). The correctness of any communication protocol requires a certain set of constraints to be true. These constraints represent the synchronization of the messages. It is a non-trivial task to write correct communication constraints between many agents. Thus a declarative specification of constraints should be used. Propositional Temporal Logic (PTL) is chosen for specifying the constraints among the participating agents.

The temporal operators `SOMETIME`, `NEXT`, `ALWAYS` and `UNTIL` provide the necessary semantics to cover the various kinds of dependencies between events/messages in a communication process. For example, `SOMETIME` conveys the obligation to honor the events in future states. The `NEXT` operator is similar to the concept of `trigger` used in active databases; it guarantees an event to occur after the current one. The PTL formula `~B UNTIL A` (`~` is the symbol for `NOT`) expresses the constraint that the event B cannot happen until event A has happened. The unary operator `ALWAYS` is used to express cyclic sequences. For example, `ALWAYS C` means that the PTL formula C will always has the value `TRUE`. This is because the condition C is always being regenerated. In [Ngu90], it is shown that the composition of the various operators provides all the basic control constructs required in transaction modelling in a single system. For example two mutually exclusive events `A` and `B` can be specified as:

```
ALWAYS (A => NEXT (~(A OR B) UNTIL A_trigger))
ALWAYS (B => NEXT (~(A OR B) UNTIL B_trigger))
```

To specify that an event `A` can happen only once, we need `ALWAYS (A => NEXT (ALWAYS (~A)))` . To specify a repetitive sequence, we need the initial constraint and the cyclic constraint:

```
(~A) UNTIL B
ALWAYS (A => (NEXT ((~A) UNTIL B)))
```

Most importantly, PTL specifications can be verified formally by using the well proven tableau method [Wol82].

In the following sections, we illustrate the specification and verification of communication constraints through modelling an interoperable transaction.

3. Specification of an Interoperable Transaction

The architecture that we assume for building a cooperative information system is shown in Figure 1. We call it a Communication Management Systems (CMS) instead of the more popular terms such as federated systems or heterogeneous systems. This re-emphasizes the importance of modelling communication aspects in an interoperable system.

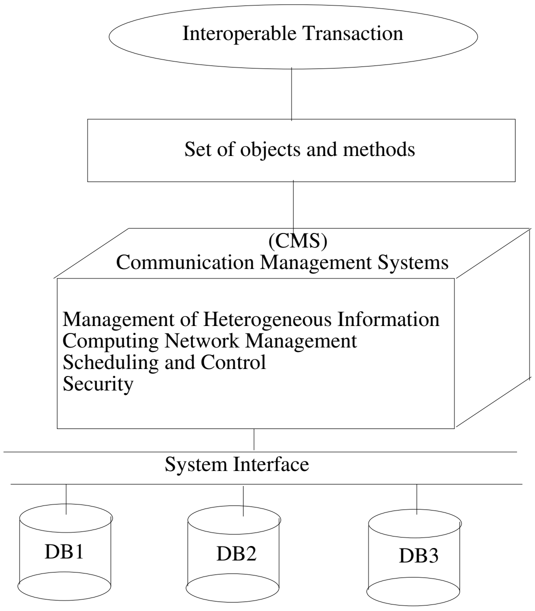

Figure 1: Architecture of CMS

We can think of the conceptual schema for an interoperable application to be a set of objects, where each object is accessible by means of a well-defined interface, or set of methods to component agents (systems). The problems of integration and translation of the underlying heterogeneous data is assumed to be managed by the communication management systems. This will not be discussed in the context of this paper. Consider as an application the communication between agents involved in planning a business trip; this is an interoperable transaction which involves agents from any airline company, any credit card company and any hotel. The agents can come from different databases/systems such as an airline database or a hotel chain's database.

The modelling of this interoperable transaction is done in an incremental manner. The business trip is a long running transaction (may take days to complete) which is better modeled by an aggregation of smaller transactions such as `flight_reservation`, `hotel_reservation` and `creditc_payment` with communication constraints specified among these three smaller subtransactions[1]. If no con-

[1] A subtransaction in the parent interoperable transaction can be an interoperable transaction

straints are specified, the subtransactions can attempt to communicate whenever they like and for as long as they like. In the following examples, keywords are in boldface.

Transaction Business_trip
aggregation_of :
  flight_reservation
  hotel_reservation
  creditc_payment
constraints:
/* no order is specified between hotel reservation and flight reservation */
/* ~ stands for NOT */
~creditc_payment UNTIL flight_reservation
~creditc_payment UNTIL hotel_reservation
goal = (flight_reservation & hotel_reservation & creditc_payment)
End Transaction

No constraint is specified between `flight_reservation` and `hotel_reservation`. Thus the two transactions can happen in any order or concurrently. The `goal` or the success of the business trip depends on having all the three subtransactions running to a successful state. Note that this specific interoperable transaction is an aggregation of three subtransactions. Below is the detailed specification of each constituent transaction:

Transaction hotel_reservation
agents:
  u:user (all the potential customers in the network)
  h:hotel(all the hotels in the network)
user can send:
/* the room is an attribute of the hotel agent */
 {**messages:**
  request(reserve(room)) to the hotel
  request(cancel(reserve(room))) to the hotel
 }
hotel can send :
 {**messages :**
  reserve(room) to self
  cancel(reserve(room)) to self
  confirm(reserve(room)) to user
  assert(no_available(room)) to user
 }
constraints:
 /* each message is represented as a propositional event */
 ~reserve(room) UNTIL request(reserve(room))
 ~cancel(reserve(room)) UNTIL request(cancel(reserve(room)))
 ~request(cancel(reserve(room))) UNTIL reserve(room)
 ~confirm(reserve(room)) UNTIL reserve(room)
 ~assert(no_available(room)) UNTIL reserve(room)
 ALWAYS(reserve(room) => NEXT(SOMETIME(confirm(reserve(room))
  XOR assert(no_available(room)) XOR request(cancel(reserve(room))))))
Goal = {confirm(reserve(room))}
Exit = {assert(no_available(room)), cancel(reserve(room))}
End Transaction

Transaction flight_reservation

by itself

agents:
t:travel_agent (all the travel agents in the network)
a:airline(all airline companies)
travel_agent can send:
{**messages:**
request(reserve(ticket))) to airline
request(cancel(reserve(ticket))) to airline
request(get(flight_schedule)) to airline
check(flight_schedule) to self
}
airline can send:
{**messages:**
reserve(ticket) to self
cancel(reserve(ticket)) to self
confirm(reserve(ticket)) to travel_agent
assert(no_available(ticket)) to travel_agent
get(flight_schedule) to self
}
constraints:
~request(get(flight_schedule)) UNTIL request(reserve(ticket))
~get(flight_schedule) UNTIL request(get(flight_schedule))
~check(flight_schedule) UNTIL get(flight_schedule)
ALWAYS(check(flight_schedule) => NEXT(~check(flight_schedule)
UNTIL get(flight_schedule)))
~reserve(ticket) UNTIL check(flight_schedule)
~cancel(reserve(ticket)) UNTIL request(cancel(reserve(ticket)))
~confirm(ticket) UNTIL reserve(ticket)
~assert(no_available(ticket)) UNTIL reserve(ticket)
~request(cancel(reserve(ticket))) UNTIL reserve(ticket)
ALWAYS(reserve(ticket) =>
NEXT(SOMETIME(confirm(reserve(ticket))
XOR assert(no_available(ticket)))))
ALWAYS(confirm(ticket) => NEXT(TRUE OR request(cancel(reserve(ticket)))))
ALWAYS(reserve(ticket) => SOMETIME(request(cancel(reserve(ticket)))))
Goal = {confirm(reserve(ticket))}
Exit = {assert(no_available(ticket)), cancel(reserve(ticket)) }
End Transaction

Transaction creditc_payment
agents:
c:card (MASTER card and VISA card)
u:user (all MASTER card or VISA card holders in the network)
user can send:
{**message:**
request(pay_with(creditcard)) to the card
request(cancel(pay_with(creditcard))) to the card
request(increase(credit)) to the card
}
card can send:
{**messages:**
validate(creditcard) to self
pay_with(creditcard) to self
cancel(pay_with(creditcard)) to self
assert(not_enough(credit)) to user
assert(invalid(creditcard)) to user
assert(refuse_to(increase(credit)))
}
constraints:
~pay_with(creditcard) UNTIL validate(creditcard)

```
~validate(creditcard) UNTIL request(pay_with(creditcard))
~request(cancel(pay_with(creditcard))) UNTIL assert(not_enough(credit))
validate(creditcard) =>
    NEXT(SOMETIME(pay_with(creditcard)
    XOR assert(invalid(creditcard)) XOR
    assert(not_enough(credit))))
~request(increase(credit)) UNTIL assert(not_enough(credit))
~request(cancel(pay_with(creditcard))) UNTIL
    assert(not_enough(credit))
~increase(credit) UNTIL request(increase(credit))
~assert(refuse_to(increase(credit))) UNTIL request(increase(credit))
```

Goal = {pay_with(creditcard)}
Exit= {cancel(pay_with(creditcard)) or
assert(invalid(creditcard)) or assert(refuse_to(increase(credit))}
End Transaction

Agents are bound to the objects which need to communicate. For each agent to be communicating, we require the definition of its obligation in terms of messages(actions) that it can send. For example, the common actions are requesting, asking, ordering, demanding, canceling, interrogating, asserting, claiming, warning, ensuring etc. For each communicative transaction, the `Exit` condition specifies the action which stops the transaction in a failure state, while the `Goal` specifies the action which makes the transaction run to a successful state.

The constraints specify the message exchanges between the agents and the overall execution sequence of operations. Although the agent is bound to variables which range over the object's type, the constraint assumes a particular instantiation of the agent. This is justified by the fact that our intention is a clear specification and verification, not code generation. We take a functional approach to specifying the messages. For example, the message `request(reserve(room))` has the generic action `request` applying to another action `reserve(room)`. This gives the provision for stating meta constraints such as "$\forall\ \alpha$ `~service(`α`) UNTIL request(service(`α`))`". The clause `service(`α`)` can be replaced by `reserve(room)`, `cancel(reserve(room))`, `confirm(reserve(room))`, `request(reserve(room))`, `assert (no_available(room))`, `or request(cancel(reserve(room)))`. It is obvious that meta constraints like this will reduce the effort in specifying trivial and repetitive constraints. The message `cancel(reserve(room))` has the meaning of undoing all the effects of the reservation. Its semantics are the same as `compensating` transactions. We introduce the operator `XOR` for the convenience of specifying the mutually exclusive actions.

In the `flight_reservation` example, we state that whenever the airline object receives a request for a reservation, after checking for the appropriate schedule, sometime later it will send an answer message back to the travel agent which is either to confirm the reservation, or that there are no available seats on the required flight. The imposition of only these two kinds of answers is enforced by the constraint:

```
ALWAYS(reserve(ticket) =>
NEXT(SOMETIME(confirm(reserve(ticket)) XOR assert(no_available(ticket )))))
```

After the ticket is reserved, the user may want to cancel the reservation. Thus it is possible for the cancellation to occur after confirmation of the reservation or after the ticket has been reserved. This is specified by:

ALWAYS(confirm(ticket) => NEXT(TRUE OR request(cancel(reserve(ticket)))))
ALWAYS(reserve(ticket) => SOMETIME(request(cancel(reserve(ticket)))))

Specifying constraints in this way is not new; for example, enforcement methods of integrity constraints in databases are abundant in the literature [LS87, Kun84]. However, verification of such constraints using dependency graphs has not been exploited.

If a dependency graph cannot be constructed from these constraints then the specifications must be modified and possibly new actions or messages must be added. The whole process is iterative in nature. The dependency graph guarantees the correctness of the specifications.

4. Generation of Dependency Graphs

Constraint specifications are verified by constructing a dependency graph for each transaction. To do that we apply the tableau decomposition rule [Wol81] to all the constraints in the transaction. The objective of the decomposition rule is to separate the *current* from the *next* states. The PTL tableau decomposition rules are based on the following identities:

~X UNTIL Y = Y OR (~X AND NEXT (~X UNTIL Y))
ALWAYS X = X AND NEXT(ALWAYS (X))
SOMETIME X = X OR NEXT(SOMETIME (X))
NEXT X = TRUE AND NEXT (X)

In [KT90], a set of tableau decomposition functions is defined on the basis of these identities and the Single Event Assumption (SEA) in this method.

Definition: The *Tableau Decomposition Function* TA is a function from a set of PTL formulae to a set of PTL formulae, such that the formulae for the current state and the next state are separated. It is defined as follows (F_1 and F_2 are PTL formulae and P is the set of propositional atoms)):

TA(p) = p, for $p \in P$
TA(~p) = ~p, for $p \in P$
TA(F_1 AND F_2) = TA(F_1) AND TA(F_2)
TA(F_1 OR F_2) = TA(F_1) OR TA(F_2)
TA(NEXT F_1) = NEXT F_1
TA(ALWAYS (F_1)) = TA(F_1) AND NEXT (ALWAYS (F_1))
TA(SOMETIME (F_1)) = TA(F_1) OR NEXT (SOMETIME (F_1))
TA(F_1 UNTIL F_2) = TA(F_2) OR (TA(F_1) AND NEXT (F_1 UNTIL F_2))
TA(~(F_1 AND F_2)) = TA(~ F_1) OR TA(~F_2)
TA(~(F_1 OR F_2)) = TA(~ F_1) AND TA(~F_2)
TA(~NEXT (F_1)) = NEXT (~ F_1)
TA(~ALWAYS (F_1)) = TA(~ F_1) OR NEXT (SOMETIME (~ F_1))
TA(~SOMETIME (F_1)) = TA(~ F_1) AND NEXT (ALWAYS (~ F_1))
TA(~(F_1 UNTIL F_2)) = TA(~F_2) AND (TA(~ F_1)
OR NEXT (~(F_1 UNTIL F_2)))

The TA function is used as the basis for the graph construction algorithm. Some of the applications of the TA function result in formulae which are of the disjunctive form `A OR B`; these represent some form of choice. Typically A is a propositional formula and B is a temporal formula prefixed with the operator `NEXT`. These formulae are of special interest. The condition they define may be *fulfilled* in the current state (propositional part becoming *true*) or the condition can be delayed until later.

4.1. Construction of Graph

The process of constructing the graph can be described as a process of iteratively separating a formula into *current* and the *next* parts, where further iteration involves processing the *next* part. This is the same as evaluating the TA function during the construction process. The set of PTL formulae is interpreted as the conjunction of all the member formulae.

Definition: A dependency graph is a directed graph G = $< V, E, \gamma >$, where V is a set of vertices: each vertex v $\in$ V represents a global state and is identified by a PTL formula defining the condition to be satisfied by all sequences starting from v; E is a set of events(messages) used to label edges of the graph; γ is a set of triples $< e, u, v >$ defining an edge labeled e which leads from u to v. The set of events E is synonymous with the set of propositions P if the propositions in P represent occurrences of events.

The initial node in the graph is associated with the formulae comprising the PTL specifications. The formulae associated with this node describes the condition which must be satisfied by all sequences of events originating at this node. The graph construction process uses the decomposition rules (which is the same as using the TA function), to decide which events can happen in the current state, and what will be the new conditions to be satisfied given some events occurring in the current state.

The construction algorithm iteratively processes each node in the graph. ¿From the single event assumption, we know that at most one event can occur at any one point in time. Therefore in order to find out which event can occur in the current state, identified by some formula F, we first look up the decomposition rule/table to obtain the result of the application of TA to F. Then we choose each alternative event in turn, and perform Boolean simplification. Choosing a particular event means that the proposition representing this event is set to `TRUE` and all other propositions (not within the scope of any temporal operator) are set to `FALSE`. The result of Boolean simplification can be one of the following:

- `FALSE` - the chosen event cannot occur in the current state;
- `TRUE` - the resulting node represents a successful termination state;
- F', a PTL formula - occurrence of the chosen event leads to another state, identified with F'.

4.2. *Example*

The algorithm described above has been implemented using Allegro Common Lisp in [Kie91]. We illustrate the algorithm using a simple example. Consider the following PTL specifications which involve the transactions A,B,C,D and G. The temporal relationships between them are expressed by the following PTL formulae:

1: ~A **UNTIL** G
2: ~B **UNTIL** G
3: ~C **UNTIL** A
4: ~D **UNTIL** B
5: **ALWAYS** (A => **NEXT** (~(A **OR** B) **UNTIL** G))
6: **ALWAYS** (B => **NEXT** (~(A **OR** B) **UNTIL** G))

The transaction A and B are mutually exclusive, i.e if A has happened then it is not possible for B to occur on the path which contains A and vice versa. To guarantee a true mutual exclusion, it is necessary to specify that if A or B has happened, then A or B cannot happen again until the transaction which triggers them happens. Here G is the triggering transaction. A specification which involves **ALWAYS** implies a non-terminating path (cyclic). A non-terminating path requires us to specify for each event/transaction when it can be repeated. These constraints can be added systematically. For the above example, the following constraints are added:

7: **ALWAYS** (C => (**NEXT** (~C **UNTIL** A)))
8: **ALWAYS** (D => (**NEXT** (~D **UNTIL** B)))
9: **ALWAYS** (G => (**NEXT** (~(G) **UNTIL** (C **OR** D))))

Now we will illustrate how the dependency graph for the above 9 formulae is generated. In the initial state, any of the five transactions (A,B,C,D,G) is allowed to happen. The Single Event Assumption implies that choosing a particular transaction is equivalent to a logical proposition representing the transaction becoming **TRUE**. All the other propositions outside the scope of any temporal operator cannot happen at this point in time and are assigned the value **FALSE**.

If G happened initially, formulae 1, 2, and 9 are selected and simplified. Formula 9 which involves **ALWAYS** operator simplifies to a new constraint **~G UNTIL** (C OR D) and regenerates the original constraints by the identity rule for **ALWAYS**. The vertex 2 in Figure 2 is labeled with the set of formulae which must be satisfied by all sequences of transactions given that G has happened. Any of the four remaining transactions can happen at the next point in time. However, the only way to satisfy the set of formulae at vertex 2 is to choose either transaction B or A to happen. This will generate the set of formulae in vertex 3 and vertex 4 respectively which act as the specification for the next state. To continue the path from vertex 4 requires transaction D to happen next. Two vertices are distinct iff the two formulae are syntactically distinct. The graph construction process tests for syntactic equivalence of two vertices. For a non-terminating (cyclic) path, the graph construction process terminates successfully when further processing will lead to something that is already in the graph. In this example this happens when vertex 1 and 5 have the same set of formulae associated with them. The final dependency graph for this example with A and B not occurring on the same path is shown in Figure 2.

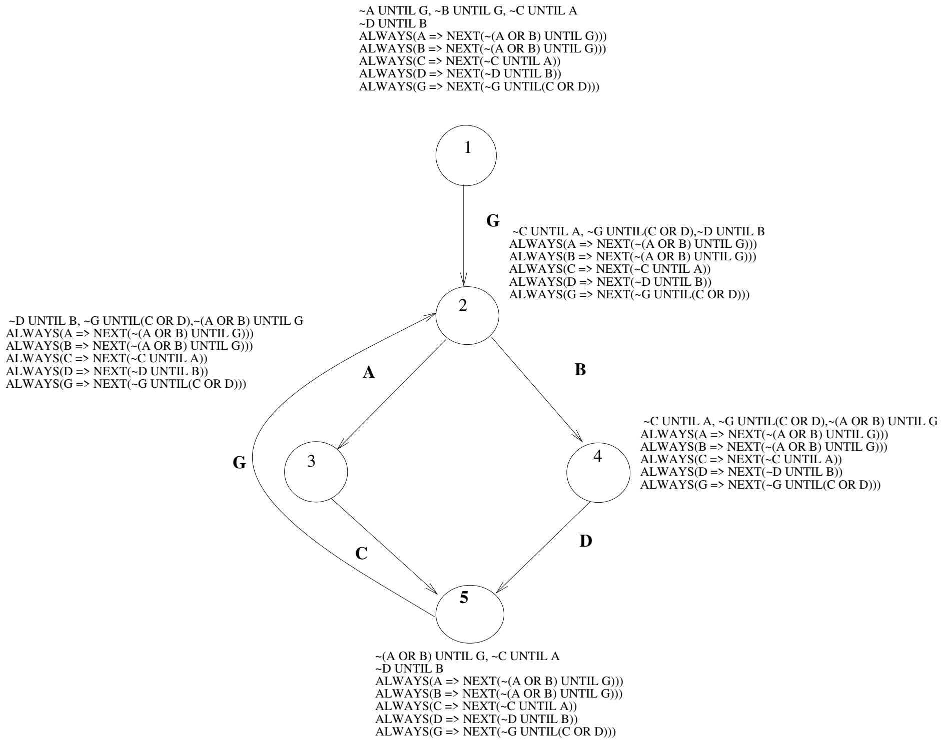

Figure 2: Cyclic Dependency Graph

The communication constraints specified in each of the transactions in Section 3. is first parsed for syntactic mistakes and then pre-processed to add the dummy constraints if non-terminating path is required and replaced the formulae having an `XOR` operator with the actual mutually exclusive formulae. Then, the above dependency graph algorithm is applied. If the specification is correct, the generation process will terminate successfully. Figure 3 shows the graph for the transaction `hotel_reservation` which can be executed in three possible ways {a, b, c, goal }, {a, b, c, exit}, {a, b, c, d, exit}. However, only one of the paths will lead to a successful transaction. For simplicity, only the formulae labeling the initial node are shown below. Notice that the formula 6, 7, and 8 are conditions for mutual exclusion (this replaces the formula `ALWAYS` in the original specification Section 3.). The formulae 9, 10, 11, 12, 13, 14 are the dummy constraints which need to be added to generate a non-terminating path.

a:

1: ~reserve(room) UNTIL request(reserve(room))
2: ~cancel(reserve(room)) UNTIL request(cancel(reserve(room)))
3: ~request(cancel(reserve(room))) UNTIL reserve(room)
4: ~confirm(reserve(room)) UNTIL reserve(room)
5: ~assert(no_available(room)) UNTIL reserve(room)

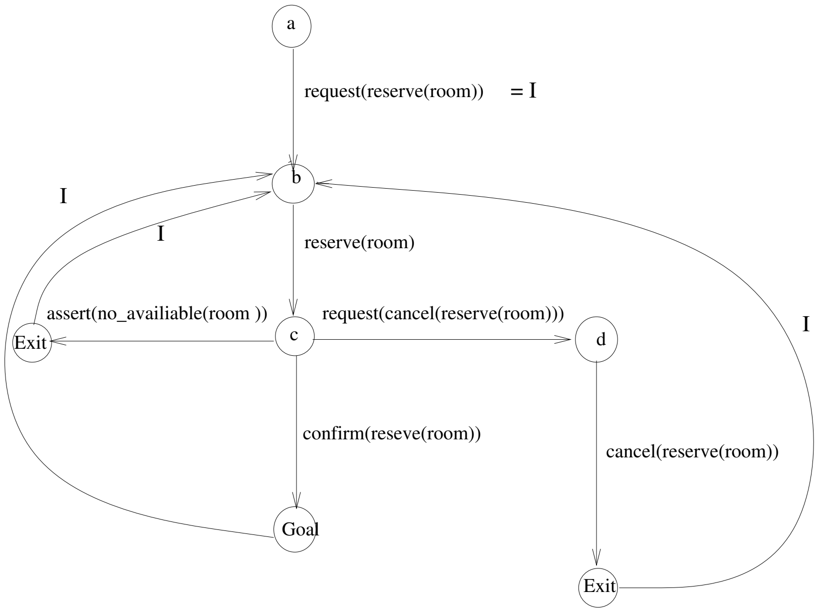

Figure 3: Dependency Graph for Hotel_Reservation

6: ALWAYS(confirm(reserve(room)) => NEXT(
~(confirm(reserve(room)) OR assert(no_available(room)) OR
request(cancel(reserve(room))))UNTIL request(reserve(room)))
7: ALWAYS(assert(no_available(room)) => NEXT(
~(confirm(reserve(room)) OR assert(no_available(room)) OR
request(cancel(reserve(room))))UNTIL request(reserve(room))))
8: ALWAYS(request(cancel(reserve(room))) => NEXT(
~(confirm(reserve(room)) OR assert(no_available(room)) OR
request(cancel(reserve(room)))) UNTIL request(reserve(room))))
9: ALWAYS(reserve(room) =>
NEXT(~reserve(room) UNTIL request(reserve(room)))
10: ALWAYS(cancel(reserve(room)) =>
NEXT(~cancel(reserve(room)) UNTIL request(cancel(reserve(room))))
11: ALWAYS(request(cancel(reserve(room))) =>
NEXT(~request(cancel(reserve(room))) UNTIL reserve(room))
12: ALWAYS(confirm(reserve(room)) =>
NEXT(~confirm(reserve(room)) UNTIL reserve(room))
13: ALWAYS(assert(no_available(room)) =>
NEXT(~assert(no_available(room)) UNTIL reserve(room))
14: ALWAYS(request(reserve(room)) =>
NEXT(~(request(reserve(room)))
UNTIL(confirm(reserve(room)) OR assert(no_available(room))
OR request(cancel(reserve(room)))))

The dependency graphs for other transactions are shown in Appendix A. Notice that the events `confirm(ticket)` and `request(cancel(reserve(ticket)))` are not mutually exclusive. It is possible to have `request(cancel(reserve(ticket)))` happen after `confirm(ticket)`,but not the other way around.

5. Related Work

The Flexible Transaction Model as described in [ELLR90] has been used for specifying the control and data flow in telecommunication service order provisioning [ANRS92]. Although the model allows specification of work flow control in the form of intra-transaction dependencies[2] and success/failure conditions, the verification of such dependencies is not supported. Therefore, a consistent and correct specification is expected to be the input to the underlying rule-based scheduler based on first order (predicate) temporal logic. The flexible transaction specifications are processed into parameterized guarded commands by the scheduler and are finally translated into the Distributed Operation Language (DOL) for execution at different autonomous systems.

The execution order of subtransactions in a flexible transaction model is restricted to a set of partial order dependencies. This is too simple for specifying all the control structures required. For example, it is not possible to specify cyclic sequences such as A, B, A, B, ... Dependencies such as *A will always follow B, A will never follow B* and *B will never follow A* are not catered for. The temporal operators such as `ALWAYS` and `SOMETIME` provide the means to specify both partial order and cyclic dependencies in our approach. Thus we are not restricted to a directed graph structure. The dependency specification language, the synchronization language and the final execution codes are in three different languages in the flexible transaction model. It is not clear why the dependency specification language and the synchronization language should be different. It is possible to argue that using a formal language like a propositional temporal logic programming language for the specification creates unnecessary complexity for users, but using two different languages to achieve the same goal without verification might result in the wrong execution code. In the Flexible Transaction Model, the predicate temporal logic[Nes93] is used as the synchronization language, which is far more than was needed and is too complex for automatic verification.

We believe that the correctness of any specification for controlling the data flow in a cooperating system is of vital importance, as any apparently minor, local problem can have disastrous effects which are distributed among the communicating transactions. The writing of such specification requires great attention to detail and is prone to errors. The use of propositional temporal logic not only has the advantages of being more declarative as compared to the flexible transaction model, but also lends itself to a well-proven verification method. Furthermore by using the full propositional temporal logic system as specification language, we avoid imposing any practical restrictions on the types of dependencies that can be expressed.

The verified interoperable transaction can be reused in other contexts; for example, by instantiation with other objects. We can create a generic interoperable

[2] The classical transaction model deals only with inter-transaction concurrency

transaction for reservation of airline tickets and hotel rooms just by changing the participating agents. The functional style of specification with meta-constraints make it easy to reuse the specification. There is no evidence that the flexible transaction specification can be reused in other contexts.

We are also exploring a more flexible and better way of specifying the compensating transaction. In particular, we are looking at how recovery and failure handling can be specified using PTL without compromising the graph generation process. In the flexible transaction model, only a subtransaction that can be compensated is able to participate in the recovery process.

6. Conclusions

We define a CIS system as a set of autonomous agents (systems) which cooperate with each other through a communication process. The interoperable transaction is the formalism we adopt for specifying this communication process. Temporal communication constraints form an essential part of the specification of an interoperable transaction. It is not a trivial task to specify the temporal constraints correctly. This is a research paper which applies the propositional temporal logic for the specification of communication constraints in a CIS system. This has resulted in a modeling mechanism in CIS system which is clear and simple and yet has a polynomial algorithm to prove the correctness (in a particular sense) of the specification. We have presented the dependency graph construction algorithm and shown how it can be applied to our example interoperable transaction.

The information in the dependency graph is valuable for the verification of the execution sequence of operations in an interoperable transaction at the design stage. Constraints that lead to deadlock can be detected in the graph and must be resolved. The graph allows the designer to study the transaction logic before it is implemented. The possible execution sequence could therefore also serve as valuable information for the scheduling of operations or subtransactions execution in different systems. Although the present paper concentrates on the verification of the execution aspects of the interoperable transaction, we are also exploring the extraction of synchronization code from the dependency graph and which would enable the generation of the distributed program code for interoperable transactions.

The current implementation of the system is restricted to accepting the PTL constraints of each interoperable transaction separately and verifying it by generating a dependency graph. If the system fails to generate the final dependency graph, the partially generated graph is manually inspected to rectify the problem. We have also began the experiment of generating the distributed program code to enforce the communication constraints.

The writing of the propositional temporal constraints seems to be complex. However, it is definitely less complex than writing a distributed program code or writing first order temporal constraints. Some common patterns of constraints could be picked up easily after some practices. Graphical representations, supported by CASE tools, can be useful.

Acknowledgments

The first author would like to thank Dr. D.H. Kieronska for the sharing of her knowledge of PTL. She would also like to thank ITK for the opportunity and its support to visit Tilburg University to carry out this research work.

References

[ANRS92] Mansoor Ansari, Linda Ness, Marek Rusinkiewicz, and Amit Sheth. Using flexible transactions to support multi-system telecommunication applications. In *Proceedings of the 18th VLDB Conference*, Vancouver, Canada, August 1992.

[BCC+92] O. Bukhres, Jiansan Chen, Jindong Chen, A. Elmargarmid, Y. Leu, and G. Zhu. The interbase parallel language. In *Proceedings of the 2nd International Workshop on Research Issues on Data Engineering: Transaction and Query Processing*, 1992.

[ELLR90] A. Elmagarmid, Y. Leu, W. Litwin, and M. Rusinkiewicz. A multidatabase transaction model for interbase. In *Proceedings of the 16th VLDB Conference*, August 1990.

[HHG90] R. Helm, I.M. Holland, and D. Gangopadhyay. Contracts: Specifying behavioural compositions in object-oriented systems. In *Proceedings of ACM ECOOP/OOPSLA*, pages 169–180, October 1990.

[Kie91] D.H. Kieronska. A system for the synthesis of concurrent programs: an algorithmic approach to state graph constructions and transformations. Technical report, PhD thesis, Department of Computer Science, University of Western Australia, 1991.

[KM85] Roger King and Dennis Mcleod. A database design methodology and tool for information systems. *ACM Transactions on Office Information Systems*, 3(1):2–21, January 1985.

[KT90] D.H. Kieronska and C.P. Tsang. An improved tableau decomposition algorithm for propositional temporal logic and its application to planning. In C.P. Tsang, editor, *Australian AI Conference*, pages 167–177, Perth, Western Australia, 1990.

[Kun84] C.H. Kung. A temporal framework for database specification and verification. In *Proceedings of the 10th International Conference on Very Large Databases*, pages 91–99, 1984.

[LM92] L. Liu and R. Meersman. Activity model: A declarative approach for capturing communication behaviour in object-oriented databases. In *Proceedings of the 18th International Conference on Very Large Databases*, pages 481–493, August 1992.

[LS87] U.W. Lipeck and G. Saake. Monitoring dynamic integrity constraints based on temporal logic. *Information Systems*, 12:255–269, 1987.

[MP81] Z. Manna and A. Pnueli. Verification of concurrent programs:the temporal framework. In R.S. Boyer and J. S. Moore, editors, *The Correctness Problem in Computer Science*, pages 215–273, London, 1981. Academic Press.

[Nes93] Linda Ness. L.0: A truely concurrent executable temporal logic language for protocols. *IEEE Transactions on software Engineering*, 19(4):410–422, April 1993.

[Ngu90] A.H.H. Ngu. Specification and verification of temporal relationships in transaction modelling. *Information Systems*, 15(2):257–267, 1990.

[Nix84] Brian Nixon. Taxis 84 selected papers. Technical Report CSRG-160, Department of Computer Science, University of Toronto, Toronto, Canada M5S 1A4, 1984.

[SK92] Amit Sheth and Leonid Kalinichenko. Information modeling in multidatabase systems: Beyond data modeling. In *Proceedings of the First Intl. Conf. on Information and Knowledge Management*, November 1992.

[Wei92] H. Weigand. Towards a design methods for interoperable databases. In D.K. Hsiao, E.J. Neuhold, and R. Sacks-Davis, editors, *IFIP DS-5 Semantics of Interoperable Database Systems*, Lorne, Victoria, Australia, November 1992.

[Wol81] P. Wolper. Specification and synthesis of communicating processes using an extended temporal logic. In *Proceedings of the 9th Annual ACM Symposium on principles of Programming Languages*, pages 20–33, 1981.

[Wol82] P. Wolper. Synthesis of communicating processes from temporal logic specifications. Technical Report STAN-CS-82-925, Stanford University, 1982.

Appendix A

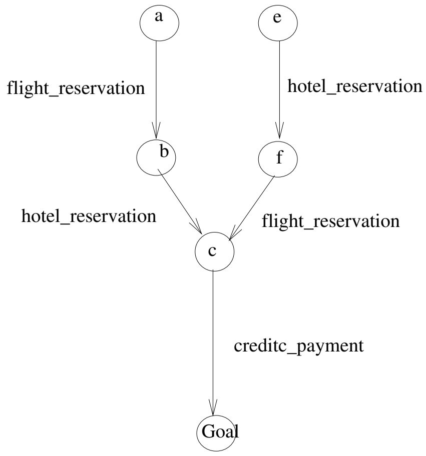

Figure 4: Dependency Graph for Business_Trip

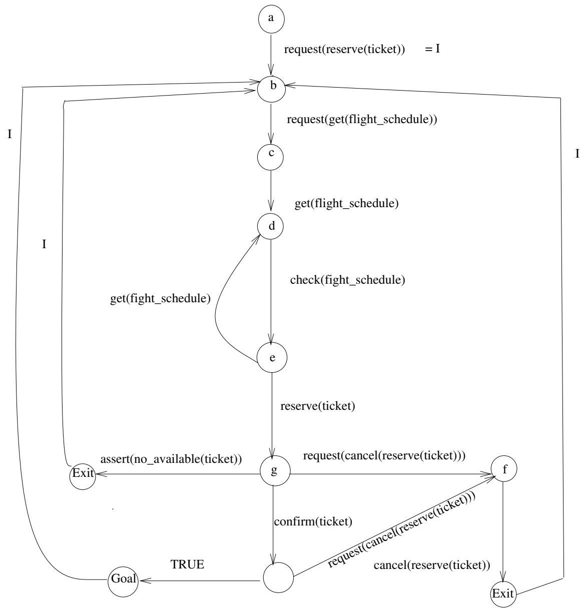

Figure 5: Dependency Graph for Flight_Reservation

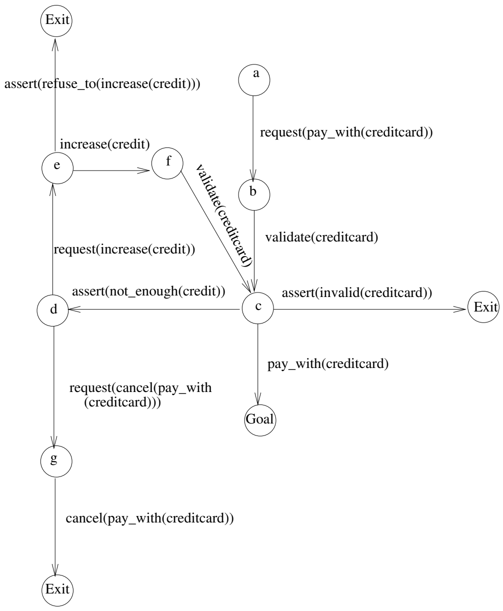

Figure 6: Dependency Graph for Creditc_Payment