Formal Verification of Workflow Schemas

C. Karamanolis[1], D. Giannakopoulou[1], J. Magee[1], S. M. Wheater[2]

[1] *Dept. of Computing, Imperial College of Science, Technology and Medicine*
{ctk,dg1,jnm}@doc.ic.ac.uk

[2] *Dept. of Computing Science, University of Newcastle upon Tyne*
Stuart.Wheater@ncl.ac.uk

Abstract

Practical experience indicates that the definition of real-world workflow applications is a complex and error-prone process. Existing workflow management systems provide the means, in the best case, for very primitive syntactic verification, which is not enough to guarantee the overall correctness and robustness of workflow applications. The report introduces a method for formal verification of system behaviour which, in C3DS, is defined as workflow schemas. Workflows are modelled by means of an automata-based method, which facilitates exhaustive compositional reachability analysis. The workflow behaviour is checked against safety properties, which can be either generic (applicable to all workflow schemas) or domain specific (applicable to a given schema). The analysis is performed in an automated way using the LTSA toolkit, which is accessible by designers who are not experts in formal methods.

1 Introduction

Workflow Management Systems provide automated support for defining and controlling various activities (tasks) associated with business processes [1, 2]. A *Workflow Schema* is used to represent the structure of an application in terms of tasks as well as temporal and data dependencies between tasks. A *Workflow Application* (or just *Workflow*) is executed by instantiating the corresponding workflow schema [3]. In the context of the C3DS project, Workflow Schemas have been chosen as the means for specifying the *behaviour view* of the system, i.e. the dependencies (data and temporal) between the constituent services of the system.

The aim of providing automated support for composing and executing complex systems, as those required for implementing business processes, is to reduce costs and flow times, to improve the robustness of the process and to increase productivity and quality of service [4, 5]. However, specifying a real-world workflow schema is a complex manual process, which is prone to errors. Incorrectly specified workflow schemas result in erroneous workflow applications, which, in turn, may cause dramatic problems in the organisation where they are deployed. Therefore, it is crucial to be able to verify the correctness of a workflow schema before it becomes operational.

Many commercial workflow management systems provide the means for some basic syntactic verification, while a workflow schema is designed. They check, for example, for the existence of inputs and outputs in task specifications. However, more thorough and rigorous analysis is required to ensure that the schema is correct [6, 7]. For instance, we need to be able to check that the workflow eventually terminates, that there are no potential deadlocks, or that a certain path of execution is possible. This report proposes a novel method for formal verification of workflow schemas, by means of *Labelled Transition Systems* (LTS). Analysis is done by automated tools provided as part of the LTSA toolkit.

LTSs and in particular the TRACTA approach followed here have been used extensively to model and analyse concurrent systems. TRACTA and LTSA are already being used in C3DS for modelling and analysis of the *structural view* of systems [8]; that is, the hierarchical structure of the system components that provide the resources for the execution of complex services as workflow applications. Our experience (backed by the feedback we have been having from the Industrial Advisory Board of C3DS) indicates that the method is accessible and usable by practising engineers, who are not experts of model checking techniques. We have therefore decided to adapt these techniques for modelling and analysis of the *behaviour view* of the system. For the discussion in this report, we use the C3DS notation for workflow schema definitions [9, 10]. However, the method is generic and can be used in combination with other approaches to workflow specification.

Due to the size and complexity of most real-world workflow schemas, any viable analysis method should follow an incremental (compositional) approach, which should be applied at each step of the design procedure [11]. TRACTA addresses exactly this problem by enforcing a close integration of modelling and analysis with system design. In particular, *Compositional Reachability Analysis* (CRA) is used for modelling and analysis of system components, as they are composed from other sub-components. CRA improves the computational complexity of analysis (together with minimisation techniques) and favours reusability of specifications.

The remainder of this report is organised as follows. Section 2 provides an overview of the semantics of the C3DS workflow definition notation, outlines the requirements for verification in this context and introduces the TRACTA approach. Sections 3 and 4 form the core of the report. Section 3 proposes a complete and formal modelling, in terms of LTSs, of all workflow schema elements. Section 4 discusses the classes of safety properties that can be verified using the TRACTA techniques and the LTSA toolkit. In both sections, the theoretical concepts are illustrated by means of a case study: the hierarchical composition of a workflow for business trip reservations. The report is concluded (section 5) with a critical discussion of the proposed method and directions of future work.

2 Background

2.1 Defining workflow schemas

A workflow schema must be expressive enough to be able to represent the structure of a business process. The schema represents a workflow application as a collection of tasks (services) and their dependencies. A task is an application-specific unit of activity. There can be two types of dependencies between tasks: 1) *notification* dependencies indicating temporal (causal) relations; 2) *dataflow* dependencies indicating that a task requires some input (data) from another task. In the following, we present the principles for workflow schema definitions [9, 10].

A task can start in one of several initial states and can terminate in one of several output states. Thus, a task is modelled as having a set of *input sets* and a set of *output sets*. Each such set consists of a (possibly empty) set of data objects. In Figure 1, task t_3 is represented as having three input sets I_1, I_2, and I_3, and two output sets O_1 and O_2.

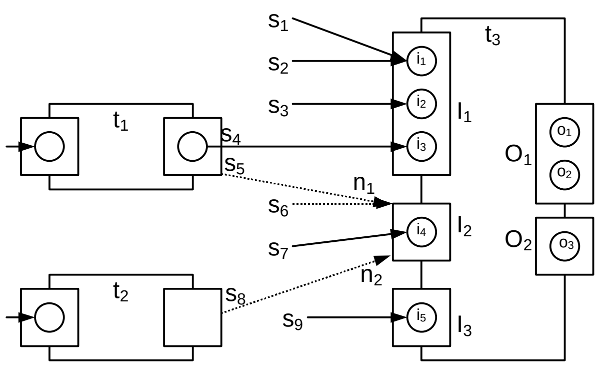


Figure 1. A workflow schema defining inter-task dependencies.

The execution of a task is triggered by the availability of an input set; only the first available input set will trigger the task. For an input set to be available, *all* its dataflow and notification dependencies must be satisfied. For example, in Figure 1, input set I_1 of task t_3 requires three dependencies to be satisfied: objects i_1, i_2 and i_3 must become available (dataflow dependencies). On the other hand, input set I_2 requires three dependencies to be satisfied: object i_1 must become available and two notifications, n_1 and n_2, must be signalled (notifications are modelled as data-less input objects). A given input can be obtained from more than one source (e.g., two for object i_1 in set I_1 of task t_3). If multiple input sources become available simultaneously, then one source is selected deterministically by the execution environment.

The notification dependencies are represented by dotted lines, for example, s_6 is a notification source for notification dependency n_1. A notification dependency may have more than one alternative sources too. For example, n_1 has two alternative sources, s_5 and s_6. A task terminates producing output objects belonging to exactly one of a number of output sets (e.g. O_1 or O_2 for task t_3).

To allow workflow applications to be designed in a hierarchical way, tasks can be *composite*: they are realised as a collection of instances of other, inter-dependent tasks. Therefore a task can be either primitive (implemented by some application service) or composite (consists of other primitive of composite tasks). Figure 2 illustrates an example of a composite task called *Reservation*. The task provides the schema definition for a trip reservation workflow.

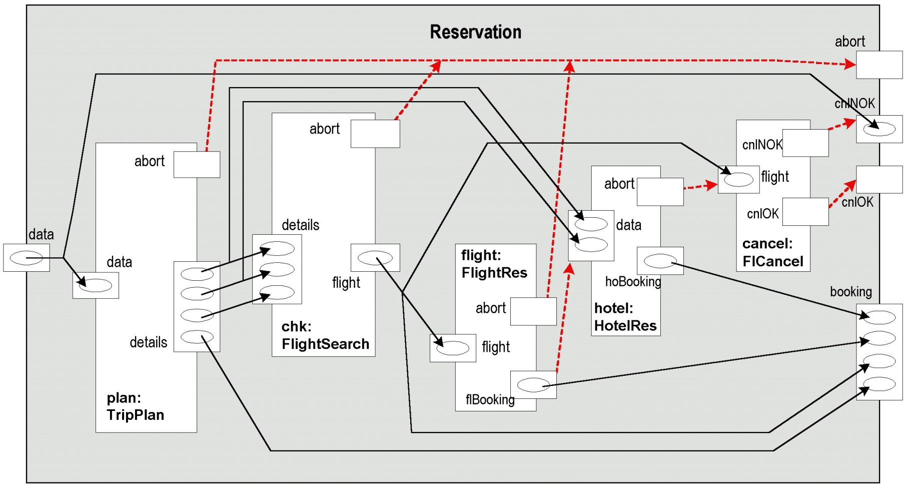


Figure 2. A composite workflow task.

Composite tasks facilitate reusability of workflow schema definitions. Figure 3 shows a composite task *BusinessTrip*, which reuses the definition of the *Reservation* task to define a workflow schema for a business trip reservation process. The latter two examples (tasks *Reservation* and *BusinessTrip*) will be used as case-studies for the rest of the report, in order to illustrate the modelling and analysis concepts discussed.

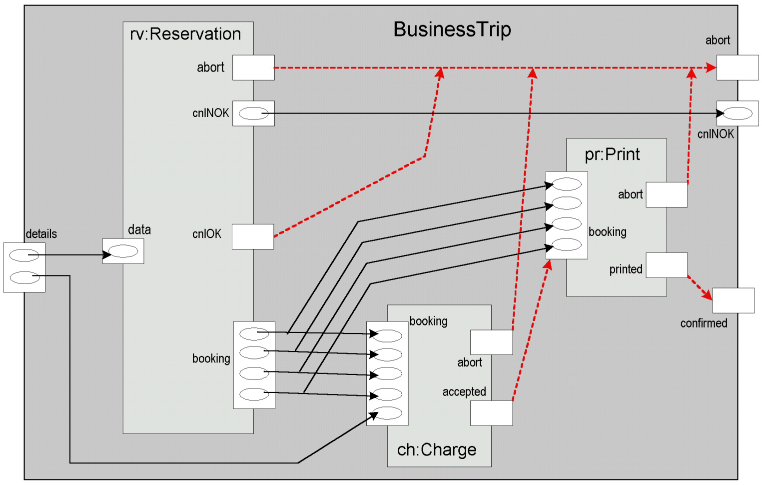


Figure 3. A composite workflow task containing a composite sub-task.

2.2 Requirements for a workflow verification method

Our experience with building large workflow systems indicates that it is important for the designer to be able to apply a rigorous verification method on the workflow schema and argue formally about the correctness of the resulting workflow applications. In this context, we have identified a number of requirements to be satisfied by any such verification method:

1. Have a solid mathematical foundation and allow for rigorous and formal analysis of both safety and liveness properties.
2. Perform exhaustive analysis at design-time (of the workflow schema) as well as interactive simulation of the workflow model.
3. Employ algorithms that are computationally efficient in order to be applicable to real-world systems. These algorithms should be supported by automated tools.
4. Follow a compositional approach in order to enable incremental analysis while the system is designed and to support re-use of specifications in multiple contexts.
5. Generate meaningful diagnostic information, in the form of execution traces, to indicate potential errors to the designer.
6. Use a comprehensible graphical representation for humans and also an equivalent well-defined and space-efficient formal notation for usage with the tools.
7. Be understandable and accessible by users who have no special expertise in the area of modelling and formal methods.

2.3 The TRACTA approach to behaviour modelling and analysis

The TRACTA approach has been extensively used for modelling and analysing concurrent and distributed systems [12-14]. It is based on the use of *Labelled Transition Systems* (LTS) for modelling the behaviour of system components and for expressing system properties.

In order to integrate analysis with other activities of software development, TRACTA uses a compositional approach to modelling, by following the phases of hierarchical system design. Behaviour is attached to the software architecture by specifying a labelled transition system for each primitive component in the hierarchy (primitive is a system component which cannot be expanded to sub-components, at least for the sake of analysis). Following the terminology of traditional process algebras, the LTS of a primitive component is equivalent to a finite-state interacting process. An LTS contains all the reachable *states* and executable transitions (triggered by *actions*) of a process. The behaviour of composite system components is defined as the composition of the LTSs of their constituent components.

TRACTA exhaustively explores the reachable states of an LTS, a technique known as reachability analysis. The main disadvantage of this technique is state explosion. That is, the exponential relation between the system state-space and the number of its constituent components. TRACTA takes advantage of the hierarchical structure of the system in order to address this problem. As the system behaviour is composed in a bottom-up manner, internal details (actions) of a subsystem's behaviour are hidden and the subsystem is minimised, at intermediate stages of the analysis. In general, only a subset of the actions in a subsystem's LTS are of interest to external systems (processes) that have to interact with it.

Explicit representation of LTSs becomes impractical for systems with more that a few states. For this reason, TRACTA uses a simple process algebra notation called FSP (stands for *Finite State Process*) to specify the behaviour of components in a system [15]. FSP is not a different way of modelling a system. It is a specification language with well-defined semantics in terms of LTSs, which provides a concise way for describing LTSs. Each FSP expression can be mapped onto a finite LTS and vice versa.

TRACTA is supported by the *LTSA software tool*, which provides for automatic composition, analysis, minimisation, animation and graphical display of system models expressed in FSP.

Primitive system components

Primitive system components are defined as finite-state processes in FSP using action prefix "`->`", choice "`|`" and recursion. If `x` is an action and `P` a process, then `(x->P)` describes a process that initially engages in the action `x` and then behaves exactly as described in `P`. If `x` and `y` are actions, then `(x->P|y->Q)` describes a process which initially engages in either of the actions `x` or `y`, and the subsequent behaviour is described by `P` or `Q`, respectively. The definition of a primitive component may use an *auxiliary* process (used as a means for modular FSP specifications).

FSP uses an *interface* operator '`@`', which specifies (using prefix matching) the set of action labels which are visible at the interface of the component and thus may be shared (synchronisation points – used for interaction) with other components. All other actions are "hidden" and will appear as silent "τ" (tau) actions during analysis, if they do not disappear during minimisation of the component. When it is more concise to describe what actions are hidden rather than which actions remain observable, the *hiding* operator "\" may be used instead.

Composite system components

Composite-component processes are defined in terms of other, non-auxiliary, processes. Their identifiers are prefixed with "`||`". The process of a composite component does not define additional behaviour; it is simply obtained as the parallel composition of instances of the processes it is made of. Process instances are denoted as "*instance-name:type-name*". The LTS of the instance is identical to that of the type, with action labels prefixed with the instance name. The instance name is not necessary if there is just one instance of a process in a given context. Composition expressions use parallel composition (`||`) together with operators such as re-labelling (`/`), action hiding (\) or interface (`@`). Communication is modelled by means of synchronisation of shared actions (the remaining actions are interleaved). Actions that correspond to interaction interfaces are re-labelled to a common name in order to be synchronised when behaviours are composed. Re-label specifications are of the form "*new-label/old-label*".

More details of the TRACTA approach and the FSP specification language will become clear during the discussion of wokflow modelling and analysis, in the following sections.

3 Workflow modelling

The model of each workflow schema consists of two parts:

- A generic part, which is concerned with modelling elements that are common to every schema, such as input/output interfaces and dataflow/notification dependencies between tasks.
- An application-specific part, which is concerned with the model of actual tasks in the schema and their inter-dependencies.

The models are presented in the form of FSP specifications and, when appropriate, as LTS diagrams produced by the LTSA tool.

3.1 Task interfaces

A task interacts with its environment through its *interface sets*. Interface sets consist of zero or more data *objects* (representing dataflow dependencies) and inbound and outbound *notifications* (representing notification dependencies). Interface sets model the common denominator of the behaviour of input and output sets of tasks.

- *An interface set is "available", if all its dataflow and notification dependencies are satisfied. When an interface set is available, then all of its constituent objects and outbound notifications are also available.*

An interface object can perform `input` and `output` actions, reflecting the fact that the object receives and outputs data, respectively. To model the fact that an interface becomes available when *all* its constituent objects are available (a logical AND operation), we use an action `available`, on which all objects in a set need to synchronise. An object can only perform `available` after performing action `input`. Therefore, the behaviour of an object with identification `ID` (to uniquely identify it in the set) is modelled as follows:

```
Object (ID=1) = (input[ID] -> available -> output[ID] -> STOP).
```

Action `available` is also used to make sure that all inbound notifications are received before an interface set becomes available and also, that outbound notifications are provided only after the interface set becomes available:

```
InNotification (ID=1) = (inNotify[ID] -> available -> STOP).
OutNotification = (available -> outNotify -> STOP).
```

An interface set is, then, modelled as the parallel composition of a set of objects and inbound and outbound notifications. If an interface set does not contain any objects and has no notification dependencies, it is unconditionally available, as modelled by process `Default`.

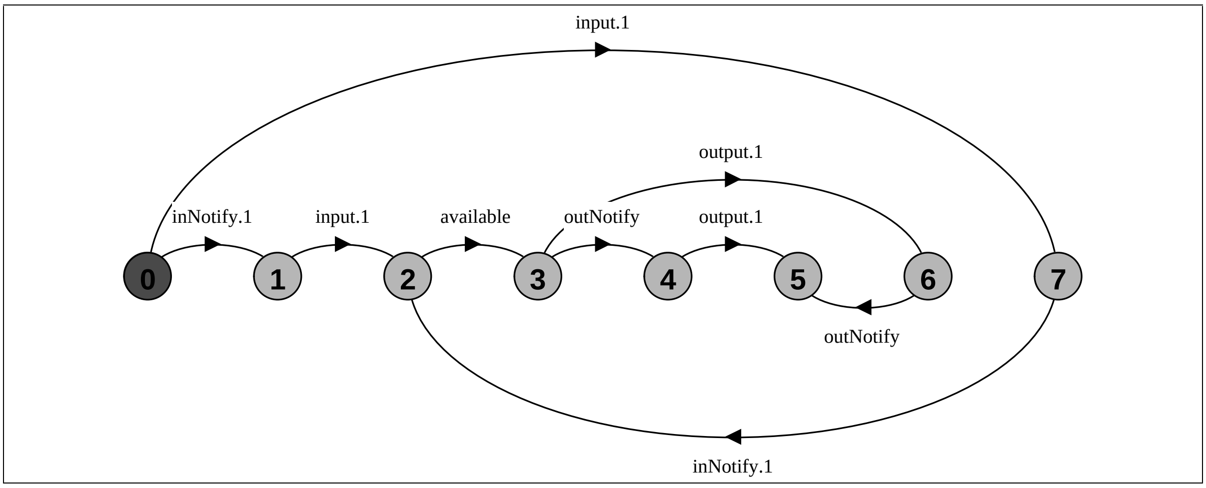


Figure 4: LTS of an interface set, with one object, one in- and one out- notification.

```
|| Iface (Objs=1, INotfs=1, ONotfs=1) =
      if (ONotfs >= 2) then
             Iface_Problem
      else (   if (Objs > 0) then
                    (forall [i:1..Objs] Object(i))
             || if (INotfs > 0) then
                    (forall [i:1..INotfs] InNotification(i))
             || if (ONotfs > 0) then
                    OutNotification
             || if (Objs == 0 && INotfs == 0 && ONotfs == 0) then
                    Default
      ).

Default = (available -> STOP).
Iface_Problem = (erroneous -> ERROR).
```

An interface produces at most one outbound notification (which can be bound to more than one task). Thus, no identifier is required for these type of notifications. A process `Iface_Problem` is introduced to model a transition to an error state, if an interface instance is specified with more than one outbound notification. Figure 4 illustrates the LTS of an interface with one object, one inbound and one outbound notification.

3.2 Primitive tasks

The main entities of a primitive task that need to be modelled are its interfaces, qualified as *input* and *output sets*. They are modelled as interfaces that have zero notifications. The reason is that, in the general case, a task should be modelled in a reusable way: the designer has no knowledge of the context in which the task may be instantiated.

```
minimal
|| AbsInputSet (Objs=1) =  (Iface(Objs, 0, 0))
                                  / {ready/available}
                                  @ {ready, input}.

minimal
|| AbsOutputSet (Objs=1) =        (Iface(Objs, 0, 0))
                                  / {enable/available}
                                  @ {enable, output}.
```

Action `available` of the `Iface` process is renamed to `ready` or `enable` to differentiate between input and output sets, respectively. Moreover, information that is concerned with the outputs of input sets and the inputs of output sets is encapsulated within the model of primitive tasks. The only actions kept explicitly visible are the ones prefixed with labels `input` and `ready` for input sets, and `output` and `enabled` for output sets.

The prefix `minimal` is added to the processes to make sure that, during the generation of the model, our tools will not only hide the actions that are not made visible, but will also minimise the corresponding LTSs. Minimisation results in a more compact but behaviourally equivalent model.

A primitive task's behaviour is dictated by two rules:

- *The execution of a task starts as soon as one of its input sets is available.*
- *When the execution of a task completes, exactly one of its output sets is available.*

The two rules also capture the causal dependency between a task's input and output sets. This behaviour pattern is common to all primitive tasks and is modelled by the process `AbsTaskImpl.` This process also models the fact that, even if more than one input set is available, just one is selected by the internal task behaviour and exactly one output is produced.

```
AbsTaskImpl (InSets=1, OutSets=1) = ( ready[i:1..InSets] -> Execute ),
Execute = ( enable[o:1..OutSets] -> STOP ).
```

A specific primitive task is then defined as the parallel composition of instances of its input and output sets with an instance of the above default implementation process. For example, the primitive task `TripPlan` of Figure 2 is modelled as shown below. The renaming reflects the bindings of the task's interfaces to `AbsTaskImpl`.

```
|| TripPlan = (     AbsTaskImpl(1, 2)
                    ||     data:AbsInputSet(1)
                    ||     abort:AbsOutputSet(0)
                    ||     details:AbsOutputSet(4)
                 )
                 /{  data.ready/ready[1],
                     abort.enable/enable[1],
                     details.enable/enable[2] }.
```

3.3 Composite tasks

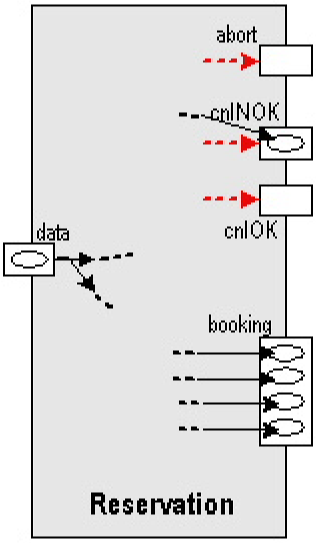
Composite tasks are constructed out of a number of constituent task (sub-task) instances. Sub-tasks are either primitive or composite tasks. In the context defined by a composite task, the data objects of its input and output set(s) ("external" interfaces) are bound to objects of input and output sets, respectively, of sub-tasks. Similarly, there may be notification dependencies between external and internal interfaces.

However, incoming notification dependencies to the composite's own input sets and outgoing notification dependencies from the composite's output sets are not known in this context. The aim is to achieve reusability of the composite task's model, by making it context independent. This principle is captured in the specifications of the external input and output sets of composite tasks: an `InputSet` is an interface set with no input notifications and an `OutputSet` is an interface set with no output notifications.

```
|| InputSet (Objs=1, ONotfs=1) =
       if (Objs ==0 && ONotfs==0) then Iface_Problem
                           else (Iface(Objs, 0, ONotfs))
       / {ready/available}.

|| OutputSet (Objs=1, INotfs=1) =
       if (Objs ==0 && INotfs==0) then Iface_Problem
                           else (Iface(Objs, INotfs, 0))
       / {enable/available}.
```

The conditional specification in the above model states that: 1) an external input set must have at least one data object or one outgoing notification; 2) an external output set must have at least one data object or at least one incoming notification. Process `Iface_Problem` is again used to model a transition to an error state, if either of the above conditions is not satisfied. Composite task `Reservation` (of Figure 2) has one external input and four external output sets. Figure 5 illustrates a general diagram of the task interfaces and the corresponding model.




```
|| Reservation = (
   /* Selectors of interface sets ...*/
      InSelector(1) || OutSelector(4)
   /* Interfaces ... */
      /* Input sets */
       || data:InputSet(1, 0)
      /* Output sets */
       || abort:OutputSet(0, 1)
        || cnlNOK:OutputSet(1, 1)
        || cnlOK:OutputSet(0, 1)
        || booking:OutputSet(4, 0)
     ...
   ) /{ /* mappings to selectors... */
        data.ready/ready[1],
        abort.enable/enable[1],
        cnlNOK.enable/enable[2],
        cnlOK.enable/enable[3],
        booking.enable/enable[4], ... }.

InSelector (InSets=1) = (ready[i:1..InSets] -> STOP).
OutSelector (OutSets=1)= (enable[i:1..OutSets] -> STOP).
```

Figure 5. "External" input and output sets of a composite component.

In the case of composite tasks, we have, again, to model the fact that exactly one input set is selected even if more than one is available and exactly one output set is enabled when the task terminates. The later is modelled by processes `InSelector` and `OutSelector`, instances of which are used together with the necessary action renaming to guarantee single set selection, as shown in Figure 5.

When instances of tasks are interconnected within the context of a composite task, two additional generic processes, `ContextOutNotfs` and `ContextInNotfs` provide the "glue" required for specifying notification dependencies. Their models are based on the abstract input and output sets, since they introduce conditions for an input to become ready or provide outputs, in the form of notifications, as soon as an output set is enabled. Recall that we can have only a single outbound notification per output set.

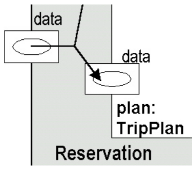
```
|| ContextOutNotfs =    AbsOutputSet( 1 )
                                   / { outNotify/output[1] }.

|| ContextInNotfs (INotfs=1) =   AbsInputSet( INotfs )
                                   / { inNotify/input }.
```

All tasks, whether primitive or composite, may need to be augmented with the above behaviour, when they are put in a context. For example, the model of task `Reservation` consists of the parallel composition of five constituent task instances, together with the processes modelling their notifications glue.

```
|| Reservation = (
   ...
   /* Constituent tasks ... */
   || plan:TripPlan       || plan.abort:ContextOutNotfs
   || chk:FlightSearch    || chk.abort:ContextOutNotfs
   || flight:FlightRes    || flight.abort:ContextOutNotfs
                          || flight.flBooking:ContextOutNotfs
   || hotel:HotelRes      || hotel.data:ContextInNotfs(1)
                          || hotel.abort:ContextOutNotfs
   || cancel:FlCancel     || cancel.flight:ContextInNotfs(1)
                          || cancel.cnlNOK:ContextOutNotfs
                          || cancel.cnlOK:ContextOutNotfs
   ) / { ... }
```

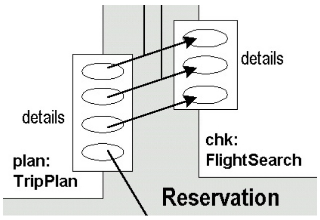
The implementation of composite tasks is modelled by the appropriate bindings between interface sets. In the case of the `Reservation` task, the dataflow dependency between the `data` object of `Reservation`'s (external) input set and the object of the input set of task `plan:TripPlan` is modelled by the renaming of Figure 6.




```
|| Reservation = ( ... )
   / {..., data.output[1]/plan.data.input[1], ... }
```

Figure 6. Dataflow dependencies between external and sub-task input objects.

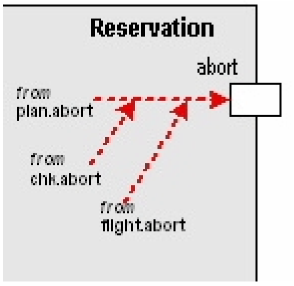
The dataflow dependency between three of the objects of the `data` output set of `plan:TripPlan` and the `data` input set of `chk:FlightSearch` are modelled by a similar renaming of all the corresponding data objects, as shown in Figure 7. A single data object may be the source of more than one dependency, as it is the case with the first two objects of the `details` output set of `plan:TripPlan`.




```
|| Reservation = ( ... )
   / { ...,
      plan.details.output[1]/chk.details.input[1],
      plan.details.output[2]/chk.details.input[2],
      plan.details.output[3]/chk.details.input[3]
    , ... }
```

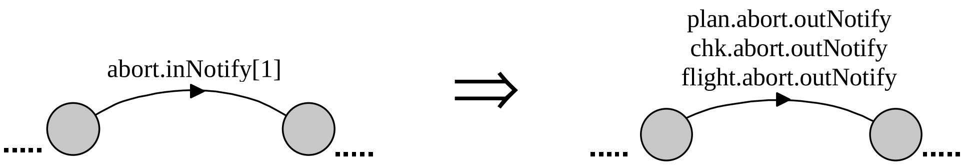
Figure 7. Dataflow input dependencies between data objects of sub-tasks.

A given set or object may have more than one alternative input sources. Availability of *any* of the input sources (logical OR) is enough to enable the set or object, accordingly. Alternative dependency sources are modelled by means of *relational relabelling*. In our example, the `abort` output set of `Reservation` can be enabled by a number of alternative sources: `plan.abort`, `chk.abort` and `flight.abort`. The relational relabelling of Figure 8(a) states that a transition labelled `abort.inNotify[1]` in the LTS of the "external" output set `abort` is, now, performed when *any* of the other three transitions occurs. The corresponding transformation of the LTS is shown in Figure 8(b).




```
|| Reservation = ( … )
   / { ... ,
          { plan.abort.outNotify,
            chk.abort.outNotify,
            flight.abort.outNotify
          } /abort.inNotify[1]
      ,... }
```

(a) Modelling alternative input sources for notification dependencies.



(b) LTS transformation due to relational relabelling.

Figure 8. Relational relabelling used to model alternative input sources.

4 Workflow analysis

This section describes how to customise generic LTS analysis techniques for the domain of workflow systems.

4.1 Interactive simulation

A practical first step in checking a process is to simulate its behaviour. Simulation is performed as a user-controlled animation of the process. For composite processes, the LTS of their behaviour is not composed first. The method does not, as a result, suffer from state explosion. The LTSs of the components of the process are used to determine the current state of the process, as well as which actions are enabled at that state. The enabled actions are the "ticked" actions in the "animator window". When the user selects one of these actions, the process transits to the corresponding next state. The LTSA tool highlights the transitions on the LTS diagrams of the component processes and presents the corresponding system trace.

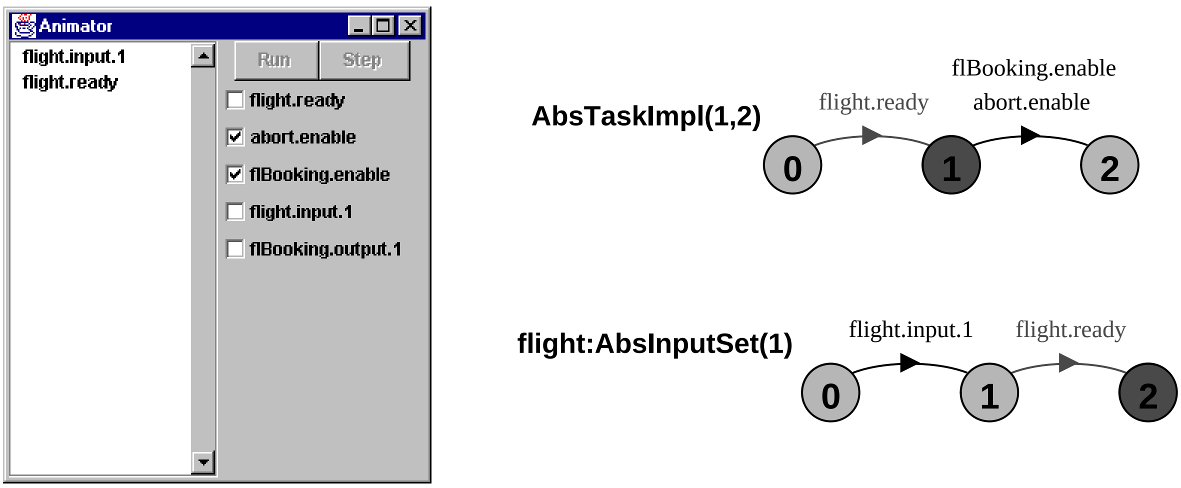


Figure 9: Interactive simulation of task type FlightRes.

Figure 9 illustrates the interactive simulation of an instance of task `FlightRes` (see Figure 2). We can see that after the input to the task has been provided, its input set `flight` becomes ready. Action `flight.ready` is performed synchronously by processes `flight:AbsInputSet(1)` and `AbsTaskImpl(1,2)`. Since this is a primitive task, the outputs become available, as soon as an input set is ready. In this case, when actions `abort.enable` and `flBooking.enable` become available, users may select which output to enable, according to the scenario they wish to check.

Interactive simulation provides an intuitive way for the system designers to experiment with different execution scenarios. However, in the general case, interactive simulation cannot establish the correctness of a real system, since designers cannot simulate all its possible execution scenarios. For that reason, techniques are required for rigorously checking the models of workflow systems.

4.2 Properties

The model-checking techniques associated with TRACTA can be used to check a workflow system exhaustively, against both generic and domain-specific properties. When a property is violated, our tools provide a *counterexample*, an execution trace that violates the property.

Generic properties: deadlock

The LTSA identifies deadlock states in the LTS of a process, as states with no outgoing transitions. Reachability of such states is checked by default for every process in the system. This is because LTSA has been mainly aimed at reactive models that exhibit non-terminating behaviours. A typical way of dealing with terminating executions is to add a looping transition to each valid terminating state of a system. For workflow tasks that are expected to terminate, we provide a generic process called `ValidTaskTermination`, which models the fact that a valid terminating state of a task is one where some output of the task has been enabled:

```
ValidTaskTermination = (out_enabled -> TERM),
TERM = (term_ok -> TERM).
```

When composed with a task that we wish to check for deadlock, this process will add looping transitions to the valid terminating states of the task. In this way, only real deadlock states will have no outgoing transitions in the resulting LTS.

```
|| Complete_Reservation =
    ( Reservation || reservation:ValidTaskTermination)
    / { {abort.enable, cnlNOK.enable, cnlOK.enable, booking.enable} /
                                            reservation.out_enabled}.
```

In this example, an instance of `ValidTaskTermination` is composed with an instance of `Reservation`. Relational relabelling is applied so that the `ValidTaskTermination` process transits to its terminating state whenever *any one* of the outputs of the `Reservation` task is enabled. In this way, valid terminating states of process `Complete_Reservation` will have looping transitions labelled with action `reservation.term_ok`. Indeed, the LTSA tool does not detect any deadlocks in process `Complete_Reservation`:

```
States Composed: 120 Transitions: 254 in 0ms
No deadlocks/errors
```

According to the workflow models of this report, the fact that a task has no deadlocks implies that it eventually terminates. In this context, this is the main liveness property of interest. Specific liveness-checking techniques are required [14], when the behaviour model of the resources used for the execution of each primitive task is also introduced in the system model. The analysis of workflow schemas, in the presence of resource models, is an ongoing research issue as discussed in section 5.

Generic safety properties

In TRACTA, safety property violations are identified by the reachability of a special "error state", represented as state -1 in LTSs. The error state has special semantics [13]. Firstly, it never has any outgoing transitions, since there is no meaning in exploring a system after a safety violation has occurred. Moreover, in the context of parallel composition, local errors are propagated globally. That is, if any component of a global state is an error state, then this global state is also an error state. Safety properties are specified as FSP primitive processes, whose definition is prefixed with the keyword "property".

A fundamental requirement, to be satisfied by all composite tasks, is:

- *The output produced by a task causally depends on the input that triggers the task execution.*

It is expressed by means of a safety property:

```
property Task_InOut_Relation = ( input_ready -> output_enable -> STOP ).
```

This property has an alphabet of two actions: `{input_ready, output_enable}`. It asserts that action `output_enable` can occur only after `input_ready`, after which none of these actions is allowed to occur again. In the corresponding LTS, any trace from the property's alphabet that does not satisfy the property leads to the error state. Property process `Task_InOut_Relation` is composed with process `Complete_Reservation` in order to check for potential violations of the property in the non-blocking version of the reservation task. Figure 10 illustrates the LTS for property `Task_InOut_Relation`, after relational relabelling is applied. It specifies, that if *any* one of the input sets (just `data` in our example) is enabled, then (and only then) *any* one of the output sets may be enabled by the corresponding task.

```
|| Check_InOut_Reservation =
(Complete_Reservation || Task_InOut_Relation)
/ {   data.ready / input_ready,
     { abort.enable, cnlNOK.enable, cnlOK.enable, booking.enable}/ output_enable
  }.
```

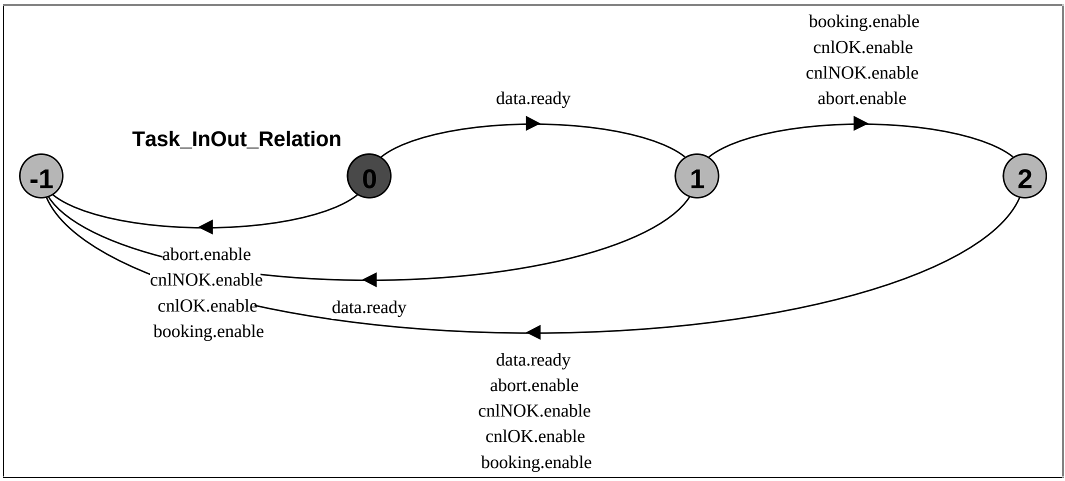


Figure 10. LTS of property `Task_InOut_Relation`.

Another typical requirement for any workflow schema is:

- *For each task, there must exist at least one execution of the workflow where this task is triggered.*

To check this for some task T, we introduce a property `PathsToSubtask` to the model, which states that no input set of T ever becomes ready. If our analysis tools return a counterexample, it means that indeed, there exists some execution where T is triggered, as desired. If the LTSA detects no violations, it means that T never plays any role in the context of the specific workflow.

```
property PathsToSubtask = STOP + {reachable}.
```

Here, action `reachable` (explicitly added to the alphabet of the property) expresses the fact that a task is triggered. In the case of a task, `reachable` is relationally relabelled to the set of `ready` actions corresponding to the task's input sets. For example, we proceed as follows to check that task `plan` is triggered in at least one execution of `Complete_Reservation`:

```
|| ExistPathsToPlan = ( Complete_Reservation || PathsToSubtask)
                        / { plan.data.ready/reachable }.
```

The LTSA tool returns the following result:

```
Trace to property violation in PathsToSubtask:
       data.input.1
       data.ready
       data.output.1
       plan.data.ready
```

The counterexample gives the prefix of an execution of `Complete_Reservation` where task `Plan` is triggered.

Domain-specific safety properties

In addition to checking generic properties of workflows, our techniques can be used for properties that refer to the particular workflow under analysis. Some examples are briefly described in this section.

- `Correct_Booking` asserts that booking is enabled only if both a flight and a hotel have been booked, and they have been booked in this order.

```
property Correct_Booking =  (     flight.flBooking.output[1] ->
                                  hotel.hoBooking.output[1] ->
                                  booking.enable -> STOP ).
```

- `Correct_Abort` asserts that, if any of the tasks `plan`, `chk`, or `flight` aborts, then the only possible outcome is an abort.

```
property Correct_Abort = No_Abort_Seen,
No_Abort_Seen =
(  {cnlNOK.enable, cnlOK.enable, booking.enable} -> No_Abort_Seen
 | {plan.abort.enable, chk.abort.enable, flight.abort.enable} -> Abort_Seen ),
Abort_Seen = (abort.enable -> STOP).
```

- `Adds_To_Abort` additionally checks, that no task is triggered (i.e. no input set becomes ready) subsequently to any abort action.

```
property Adds_To_Abort = No_Abort_Seen,
No_Abort_Seen =
({plan.data.ready, chk.details.ready, flight.flight.ready,
                                        hotel.data.ready} -> No_Abort_Seen
|{plan.abort.enable, chk.abort.enable, flight.abort.enable} -> STOP).
```

In TRACTA, any number of properties may be checked *simultaneously* on a system. All the properties of interest can be composed with the process to be analysed; reachability of the error state is then checked. We can even compose properties amongst themselves, before they are applied to a process, as the following example illustrates.

```
|| Strict_Abort_Check = (Correct_Abort || Adds_To_Abort).
```

```
|| Check_All = (  Complete_Reservation
                || Correct_Booking
                || Strict_Abort_Check).
```

4.3 Modularity and Abstraction

After checking thoroughly that a task satisfies its requirements, the behaviour of the task may be abstracted before re-using it in some other context. The only actions that need to be visible by the context of a task are actions related to its interfaces. Specifically, the interface of an abstracted task consists of the `input` actions of its input sets and the `output` actions of its output sets. Additionally, the `ready` actions of input sets and `enabled` actions of output sets must also be exposed, in order to be able to add notifications to and from the task when it is introduced in a context. The LTS of the task is then minimised. For example, the `Complete_Reservation` task is abstracted as follows:

```
minimal
|| AbstractReservation = ( Complete_Reservation )
          @ {data.input, data.ready,
             abort.enable, abort.output,
             cnlNOK.enable, cnlNOK.output,
             cnlOK.enable, cnlOK.output,
             booking.enable, booking.output
            }.
```

Minimisation reduces the size of the LTS of the reservation task from 120 down to 26 states.

5 Discussion and conclusions

The report has proposed a method for modelling and verifying workflow schemas (defining the behavioural view of complex systems and services in C3DS), in lines with the TRACTA approach. TRACTA satisfies the fundamental requirements that have been set in section 2.2. It is a mature method that has been extensively used for model checking of complex concurrent and distributed systems. It uses a solid automata-based theory to allow exhaustive analysis on the static model of a system, at design time.

The TRACTA approach is fully automated within the LTSA toolkit. The algorithms employed for process composition, action hiding and minimisation are computationally efficient and scale well for real-world workflow schemas. In addition, LTSA provides a graphical representation of LTSs and an animation facility for simulating the execution of the model. Diagnostic information is presented in the form of counterexamples: traces of execution that lead to violation of a desired property. All these facilitate the use of the method by designers that are not experts in formal methods. In fact, with an automated production of the model from the workflow schema definition (which is currently under development), the workflow designers will not have to write any FSP code apart from expressing the task properties they wish to analyse.

The feature of TRACTA that makes it particularly suitable for behaviour analysis of complex workflow schemas, as those required for application domains that are of interest to C3DS, is *compositionality*. TRACTA traditionally follows a compositional approach to modelling and analysis, in order to address the state explosion problem which is inherent to all exhaustive reachability analysis techniques. We have exploited this feature, by making the models of tasks context independent and re-usable. Therefore, designers can check the model of their system in an incremental manner, while the system is designed. Design errors can be spotted early in the design and right in the components (tasks) where they occur.

The lack of compositionality is the main weakness of the *Woflan* system, according to its designers [11]. Woflan is a verification tool that uses a special type of Petri-nets to model and analyse the behaviour of workflow processes. Errors in the model are reported in the form of "behaviour error messages", similar to our "counter-example traces". The main advantage of the system is the theoretical robustness of the Petri-nets and the clear representation of workflow state by token-based nets. However, the system lacks a means for visual representation of the model. In addition, Woflan can only handle systems with just up to 10^5 states. In comparison, LTSA can typically handle LTSs with more than 10^6 states, which may correspond to a system that is several orders of magnitude larger, before minimisation.

There are a number of directions we are planning to follow in order to extend the work presented in this report. The proposed modelling method has been illustrated by means of a specific workflow notation, the one used in C3DS. However the method is generic and can be adapted for other approaches to workflow scheme specification. To justify this claim, we are planning mappings for other (proprietary) notations used by commercial workflow management systems. In addition, the proposed method has to be extended with a generic model of recursive tasks (tasks that can trigger new instances of their own type), a common design pattern, especially in business processes.

The work presented in this report focuses on the modelling and analysis of workflow schemas, irrespectively of the environment in which schemas are instantiated and executed. Such models can be enriched with the behaviour of system resources used for the enactment of workflow instances. Analysis of the extended models can then ensure that workflow specifications are consistent with the constraints set by the execution environment. We are currently investigating what are the required abstractions for modelling system resources in this setting.

Acknowledgements

This work has been supported in part by ESPRIT LTR Project C3DS (Project No. 24962). We gratefully acknowledge our colleagues Jeff Kramer, Santosh Shrivastava and Frederic Ranno for helpful discussions as well as the members of the C3DS Advisory Board for their comments and feedback.

References

[1] Koulopoulos, T.M., *The Workflow Imperative*, New York: Van Nostrand Reinhold 1995.

[2] Georgakopoulos, D., Hornick, M., and Sheth, A., *An overview of workflow management: from process modelling to workflow automation infrastructure.* International Journal on Distributed and Parallel Databases. Vol. **3**(2), April 1995: pp. 119-153.

[3] Wheater, S.M., Shrivastava, S.K., and Ranno, F. "A CORBA Compliant Transactional Workflow System for Internet Applications", in Proc. of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing. 1998, Lake District, UK. Also: C3DS 1st year deliverable, task C2.1, part III.

[4] Workflow-Management-Coalition, *Workflow Handbook*, ed. P. Lawrence, New York: John Wiley and Sons 1997.

[5] Schal, T., *Workflow Management for Process Organisations*. Lecture Notes in Computer Science. Vol. 1096, Berlin: Springer Verlag 1996.

[6] Sheth, A.P., van de Aalst, W.M.P., and Arpinar, I.B., *Processes Driving the Networked Economy.* IEEE Concurrency. Vol. **7**(3), July - September 1999.

[7] van der Aalst, W.M.P., *The Application of Petri-NEts to Workflow Management.* The Journal of Circuits, Systems and Computers. Vol. **8**(1) 1998: pp. 21-66.

[8] Magee, J., Kramer, J., and Giannakopoulou, D. "Behaviour Analysis of Software Architectures", in Proc. of the 1st Working IFIP Conference on Software Architecture (WICSA1). 22-24 February 1999 San Antonio, TX, USA. Also: C3DS 1st year deliverable, task B1.1, part II.

[9] Ranno, F., Shrivastava, S.K., and Wheater, S.M. "A Language for Specifying the Composition of Reliable Distributed Applications", in Proc. of the 18th International Conference on Distributed Computing Systems (ICDCS-98). 1998, Amsterdam, The Netherlands. Also: C3DS 1st year deliverable, task C2.1, part II.

[10] Ranno, F., *A language and toolkit for the specification, execution and monitoring of dependable distributed applications*. PhD thesis. Department of Computing Science, University of Newcastle upon Tyne. June 1999, Newcastle upon Tyne.

[11] Verbeek, H.M.W., Basten, T., and van der Aalst, W.M.P., *Diagnosing Workflow Processes using Woflan*, 1999, Eidhoven University of Technology: Eidhoven.

[12] Magee, J., Kramer, J., and Giannakopoulou, D. "Analysing the Behaviour of Distributed Software Architectures: a Case Study", in Proc. of the 5th IEEE Workshop on Future Trends of Distributed Computing Systems. October 1997 Tunis, Tunisia, pp. 240-245

[13] Giannakopoulou, D., Kramer, J., and Cheung, S.C., *Analysing the Behaviour of Distributed Systems using Tracta.* Journal of Automated Software Engineering, special issue on Automated Analysis of Software. Vol. **6**(1), January 1999: pp. 7-35.

[14] Giannakopoulou, D., Magee, J., and Kramer, J. "Checking Progress with Action Priority: Is it Fair?", in Proc. of the 7th European Software Engineering Conference held jointly with the 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'99). September 1999 Toulouse, France. Springer, Lecture Notes in Computer Science 1687. M.L. O. Nierstrasz, Ed

[15] Magee, J. and Kramer, J., *Concurrency: State Models & Java Programs*. Worldwide Series in Computer Science: John Wiley & Sons 1999.