Group Decision and Negotiation (2007) 16: 213–254
DOI 10.1007/s10726-006-9064-4

Formalising Workflow: A CCS-inspired Characterisation of the YAWL Workflow Patterns

ANDREW D. H. FARRELL AND MAREK J. SERGOT
Department of Computing, Imperial College, London, SW7 2AZ, UK
(E-mails: andrew.farrell@imperial.ac.uk; m.sergot@imperial.ac.uk)

CLAUDIO BARTOLINI
Hewlett-Packard Laboratories, 1501 Page Mill Road, Palo Alto, CA 94304-1126, USA
(E-mail: claudio.bartolini@hp.com)

Abstract

We present work concerning the *formal specification of business processes*. It is of substantial benefit to be able to pin down the meaning of business processes precisely. This is an end in itself, but we are also concerned to do so in order that we might prove properties about the business processes that are being specified. It is a notable characteristic of most languages for representing business processes that they lack a robust semantics, and a notable characteristic of most commercial Business Process Management products that they have no support for verification of business process models. We define a high-level meta-model, called `Liesbet`, for representing business processes. The ontological commitments for `Liesbet` are sourced from the YAWL workflow patterns, which have been defined from studies into the behavioural nature of business processes. A formal characterisation of `Liesbet` is provided using Milner's Calculus of Communicating Systems (CCS). In this article, we omit some of the technical details of this characterisation and instead present the essential features by means of an abstract machine language, called LCCS. We also explain how we have facilitated the verification of certain properties of business processes specified in `Liesbet`, and discuss how `Liesbet` supports the YAWL workflow patterns. We include a simple three-part example of using `Liesbet`.

Key words: business process, workflow, meta-model, formal semantics, CCS, verification

1. Introduction

This article presents work concerning the *formal specification of business processes*. It is of substantial benefit to be able pin down the meaning of business processes precisely. This is an end in itself, but we are also concerned to do so in order that we might prove properties about the business processes that are being specified. We start by presenting some background to the modelling and specification of business processes.

The operation of companies and organisations is characterised by a number of business processes that need to be carried out in a way that is strategically aligned with the objectives of the business. The Workflow Management Coalition (WfMC) defines a *business process* to be (Workflow Management Coalition 1999) *a set of one or more linked procedures or activities which collectively realise a business objective or policy goal, normally within the context of an organisational structure defining functional roles and relationships* (Workflow Management Coalition 1999).

Business Process Management (BPM) is a term that has been used to refer to *aligning business processes with an organisation's strategic goals, designing and implementing process architectures, establishing process measurement systems that align with organisational goals, and educating and organising business managers so that they will manage processes effectively* (http://www.bptrends.com). In Marin (2002), BPM is described *as process technology enhanced with process management capabilities, implemented in a way that is appealing to business users*. Although BPM tends to be a term that is differently applied, the consensus behind its use seems to be the notion of a managed automation of business processes, where the management generally is meant to align the enactment of a process to the objectives of the (business) enterprise.

Workflow technologies (Georgakopoulos et al. 1995; Jablonski and Bussler 1996) have become a key enabling technology for the implementation of BPM. Notably, one of the principal areas in which Information Technology (IT) has been deployed to help automate the enactment of business processes has been in the *co-ordination of activity enactment*. (IT has also been used to provide application-led support to the enactment of individual activities, empowering workers to complete activities in a more timely and efficient manner.)

Workflow technologies handle the co-ordination of activities in a business process by initiating their execution through assigning agents to them at appropriate times. The term *workflow* is used in an abstract sense in that it refers to the automation of a (specific) business process, without any reference to *how* the process is automated. In contrast, the term *workflow model* refers specifically to the machine representation of a business process.

An example graphical representation of a workflow is presented in Figure 1. The language used to express a workflow model is commonly referred to as a workflow language. In the context of formalising such languages, the term *workflow meta-model*, or *workflow ontology*, is commonly used to refer to the collection of constructs used to represent a workflow model. We use the terms workflow meta-model and workflow ontology interchangeably in this report. Finally, the term *workflow*

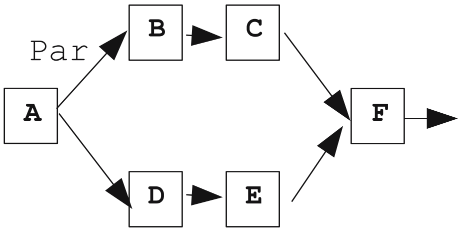

Figure 1. An example workflow model.

management system (WfMS) (a.k.a. process engine) is used to refer to the engine responsible for executing workflow models.

Within enterprises, there is a seemingly inexorable drive to improve agility and competitiveness. One proposed means of improving the efficacy of enterprise operation is the *Service-Oriented Architecture (SOA)* (Newcomer and Lomow 2005), where IT applications are repackaged as services with a standard interface, thus promoting re-use of enterprise components. *Web services* are rapidly emerging as a key facilitator of the SOA. They are proposed as *the cornerstone for architecting and implementing business processes and collaborations within and across organisational boundaries*. W3C defines a web service as *a software application identified by a URI, whose interfaces and bindings are capable of being defined, described and discovered as XML artefacts. A Web service supports direct interactions with other software agents using XML-based messages exchanged via Internet-based protocols* (WWW Consortium 2002).

Web services are used to encapsulate business functionalities. They can be invoked by applications or other web services using standardised XML-based Internet protocols, such as HTTP, SOAP, WSDL and UDDI (Curbera et al. 2002). *Service Composition* is a principal aspect of the Web Services framework, where *composite (web) services* may be created by inter-connecting deployed web services from potentially many different service providers. WS-BPEL (OASIS 2005) is a standardised language for (web) service composition. A composition is the equivalent of a workflow model in the context of SOA. Just as for a workflow model, a composition is concerned with the co-ordination of activities and the data that passes between them, except that the work carried out for an activity is typically realised by a web service rather than some other kind of IT application, or other resource. As compositions and workflows share many similarities, they are typically discussed together when talking about business process modelling.

An important distinction should be made between *Web Services Orchestration (WSO)* and *Web Services Composition (WSC)*. WSO is concerned with defining composite web services from web services that may belong to the same enterprise, or some other. WSC is concerned with defining collaborations between web services (Newcomer and Lomow 2005; Weerawarana et al. 2005). WSOs are typically viewed as under-writing WSCs, or facilitating the driving of WSC-style interactions across enterprise boundaries. That is, the WSO is the private, end-point, or local, perspective of the operation of a business process, which will need to support the public, global view (WSC) of the collaboration between the business process and others. In this work, we are solely concerned with Web Services Orchestration. An example of a WSO language is WS-BPEL (OASIS 2005). An example of a WSC language is WS-CDL (Web Services Choreography Description Language) (WS-CDL 2004).

It is often convenient to divide the description of a workflow model into several different perspectives. There have been several suggested taxonomies for workflow perspectives, e.g., (Jablonski and Bussler 1996; van der Aalst 2004). We follow the one presented in van der Aalst (2004). Here, Van der Aalst describes a number of different perspectives, but we shall concentrate on just two – the *control* and *data* perspectives. The control perspective is arguably the most important in the definition of a workflow model. It is concerned with the definition of the (partial) ordering by which activities should be executed (by a WfMS). Figure 1 is an example of a workflow model defined at the control perspective.

The data perspective is concerned with the management of data during the enactment of the workflow model. We can define two types of data: *control* and *application* (or *production*) data. Control data is used to evaluate branching conditions, or, more generally, is used by the WfMS to determine how execution should proceed (Alonso et al. 2004). It is usually declared, or allocated, within a workflow model, and its scope of existence is the workflow model. It is simply meant to control the enactment of the model. Application data, on the other hand, is data that primarily exists outside of the model, but is imported, and used, by the model. For example, in the case of workflow models, such data may be documents, forms and tables (van der Aalst 2004), or, in the case of service compositions, such data would be that sent and received in messages that are exchanged between services (Alonso et al. 2004).

A principal aim of the Process Modelling Group (PMG) (http://www.petripi.org) is to try to understand the behavioural nature of business processes, in order that ontologies may be developed for them, and so that the utility of formal tools or languages, such as Petri-nets (Reisig and Rozenberg 1998) and CCS/π-calculus (Milner 1989, 1999; Sangiorgi and Walker 2001), for providing a robust semantics for such ontologies, may be determined. The manifesto of the PMG says that it has been formed *to encourage the study of business processes and to experiment with them; it aims to ease their understanding by humans, to implement them on machines, and to develop their underlying science*.

In considering this aim, an important distinction that should be made is between who, or what, will use such business process modelling ontologies, and for what purpose. By understanding this distinction (as well as understanding the very behavioural nature of business processes themselves), we may discern what ontological commitments are appropriate for describing business processes. We distinguish between (at least) the following classes of *user* of such ontologies, where for each class, the pertaining ontology defines a *view* of business processes or workflows.

- *Presentation view*: Business managers, executives, customers.
- *Authoring view*: Business analysts and process authors – i.e. those responsible for capturing/authoring workflows. This view would have an associated ontology whose constructs would be considered to be intuitive to a process author. The ontology would most likely be graphical in nature. For instance, Figure 1 might constitute a workflow model defined using such an ontology.
- *Information view*: Serialisation (or file) format and reference point for the computational view (see below), in that it fixes the sufficient and (as much as possible) necessary representational requirements of the modelling approach. Note that in some modelling approaches, it may be appropriate to divide this view into two, along these two themes. However, we have not needed to make such a

distinction in our modelling approach. Note that the information view will typically be closely aligned to the authoring view (for ease of mapping between the two views) and will, as a consequence, make similar ontological commitments to that of the authoring view, albeit they will likely be represented by distinct ontologies.

- *Computational view*: Process engine, or the process engine implementer. `new s a.(b.c.s-|d.e.s-)|s.s.f` might be a computational view of a particular workflow model, such as the one illustrated in Figure 1, where the ontology used would be CCS/π-calculus-like (Milner 1989, 1999; Sangiorgi and Walker 2001).

Primarily, the computational view will define an ontology to provide a semantic characterisation of the ontology defined at the information view. That is, the computational view fixes the precise meaning of workflow models, by providing a semantic characterisation of information view models. The definition of the computational view will be facilitated by the use of some formal tool, such as Petri-nets (Reisig and Rozenberg 1998) or CCS/π-calculus.

A computational view workflow model may be directly executable by a workflow engine; that is, the engine may directly understand and execute Petri-nets or CCS/π-calculus. In this case, a translator will map models serialised using the information view format to the computational view. Or, as the computational view fixes the meaning of models, an engineer may implement a process engine capable of understanding models written at the information view, and ensure their enactment according to computational view semantics. In either case, it is imperative that the computational view provides an intuitive and tidy characterisation of the information view ontology.

The existence of the computational view is important for precision and robustness in the definition of workflow models, and for verifying properties of workflow models, such as workflow soundness (see below). It is a notable characteristic of most workflow languages that they lack a robust semantics (van der Aalst 2003), which would be provided by the computational view, and a notable characteristic of most commercial workflow products that they have no support for verification of workflow models.

Workflow soundness (van der Aalst 2004) is a property of the control perspective of workflow models. It is a highly desirable property that corresponds to the absence of basic errors in a workflow model. Errors can quickly creep into workflow models as they are being defined. Such errors may lead to undesirable execution of some or all instances of a workflow model (Dong and Shensheng 2003). van der Aalst (2004) says that "the errors may lead to angry customers, back-log, damage claims, and loss of goodwill". It is important, therefore, that soundness of workflow models is verified prior to model deployment.

The authoring, information and computational views of a workflow model may be represented using the same ontology or using distinct ontologies. An example of the former is the use of Petri-nets for workflow modelling where the same formal tool is used for all views. In the case where there are distinct ontologies for different workflow views, it is unlikely that the information or authoring views will be defined formally, i.e., using some mathematical formalism. Rather, they will usually be abstracting syntaxes, or ontologies, for the computational view.

In this work, we are concerned with capturing the computational view of workflows as an end in itself, as well as for facilitating the verification of workflow properties. For these purposes it is also appropriate to define an information view ontology, to serve as an abstract syntax which can, on the one hand, act as a serialisation syntax, and on the other hand, act as a reference point for the computational view ontology to target. Its primary purpose, however, is to fix concisely what we are concerned with representing. As a result, it may closely resemble an authoring view ontology – which we do not define in this article.

We define the `Liesbet` meta-model, or ontology, for the definition of (the control perspective of) workflow models at the information view. We underwrite `Liesbet` by providing a formal semantic characterisation using Milner's Calculus of Communicating Systems (CCS) (Milner 1989, 1999). We defer details of this characterisation to an associated technical report (Farrell 2006), and choose instead to provide a somewhat informal presentation of the semantic characterisation of `Liesbet` in this article. To this end, we introduce a CCS-based abstract machine language, called *LCCS*, which provides a number of CCS agent (or, process) definitions for the specification of workflow models. We have chosen CCS as the basis for our computational view ontology because of the natural, and intuitive, way in which it represents process dynamics (such as action sequencing and interleaved concurrent action execution), and because of the wealth of mathematical theory that underwrites it.

The ontological commitments that any approach to business process modelling makes should be sourced from an understanding of the behavioural nature of business processes. Members of the PMG community have previously set about characterising the behavioural nature of business processes, in the form of the YAWL (*Y*et *A*nother *W*orkflow *L*anguage) workflow patterns (Kiepuszewski 2003; van der Aalst et al. 2004; van der Aalst and ter Hofstede 2002, 2005). We use these patterns as the basis for the definition of our information view ontology (the `Liesbet` meta-model).

In summary, our main aims are as follows. We are concerned with defining an ontology for workflow workflow (`Liesbet`). It is intended to be used in other work that we are undertaking (as outlined in Section 6), in particular to support the modelling of abstract workflows for use in planning the fulfilment of business processes. We consider the YAWL workflow patterns to be a good starting point from which to derive the sufficient and necessary ontological commitments for `Liesbet`. Further, we seek to provide a formal characterisation of `Liesbet` using Milner's CCS to fix precisely the way in which workflow models evolve – a presentation, which we give in Farrell (2006) using standard CCS, and which we (somewhat informally) give in this article using a conceived abstract machine language, called

LCCS. With such a characterisation to hand, we are able to define a verification procedure for `Liesbet` workflow models, enabling us to verify model soundness and other model properties.

The structure of this article is as follows. In Section 2 we present an introduction to the `Liesbet` meta-model and some of its (core) constructs, and discuss how we have supported the YAWL workflow patterns through `Liesbet`. Section 3 provides a simple three-part example that uses the meta-model. In Section 4 we introduce LCCS and present an LCCS-based characterisation of the `Liesbet` constructs introduced in Section 2. In Section 5 we discuss verification of `Liesbet` models, and also discuss notions of `Liesbet` model equivalence. In Section 6 we conclude with a discussion and overview of related work. In the Appendix, we present a BNF grammar for `Liesbet`.

2. Liesbet Meta-Model

We have defined in this work a workflow meta-model (called `Liesbet`) corresponding to the *information view* of a workflow model, and a CCS-based (Milner 1989, 1999) characterisation (called LCCS) of `Liesbet`, corresponding to the *computational view*.

In this section we give an introduction to some `Liesbet` basics, and then proceed to introduce the most commonly used constructs of the `Liesbet` meta-model. For convenience, we describe just a handful of constructs in this section, leaving the remainder to an accompanying technical report (Farrell 2006). The constructs that we introduce here are: Activity (`Act`), Synchronisation (`Sync`), Sequence (`Seq` and `SeqCancel`), Parallel (`Par`), Exclusive Choice (`DefaultChoice` and `Choice`), Multiple Choice (`MultiChoice`), Cancel Activity (`CancelActivity`), Free Choice (`Free-Choice`) and Empty.

For each `Liesbet` construct, we present what we call an 'Easy Syntax'. For the purposes of implementing a verification approach for `Liesbet` workflow models, we have also defined an XML serialisation (or file format) syntax for each construct. This is not presented in this article, for brevity. A BNF grammar for `Liesbet` is presented in the Appendix.

2.1. Liesbet basics

We start by introducing some terminology. A *customised activity type* is a customisation of a `Liesbet` meta-model construct when used in the specification of a `Liesbet` workflow model. In contrast, the term *generic activity type* is used synonymously with meta-model construct. For example, in the `Liesbet` model `Seq (A, B)`, the `Seq` is a 'sequence' *generic* activity type which is *customised* to mean a sequence that contains two activity types, `A` and `B`.

A *basic activity type*, defined using the `Liesbet` meta-model construct `Act`, corresponds to a self-contained piece of work, where *conceptually* we would defer to the environment to inform us when the work of the activity type has completed.

In contrast, *structured activity types*, defined using any other `Liesbet` construct, exist for the purpose of marshalling instances of basic activity types (i.e. `Act` types), where the enactment of instances of these other constructs (e.g., `Par` and `Seq`) is handled wholly within the realms of the workflow engine.

During enactment of a workflow model, activity types will be *instantiated* to create *activity instances*. It is through activity instances that work is realised in the enactment of a workflow model. If an activity type is instantiated twice in the enactment of a model, the work associated with that type will be carried out twice.

Basic activity types defined in 'Easy syntax' may either be simply defined *in situ*, or in a separate definition which is then referred to when instantiating the activity type elsewhere. For basic activities, defining them *in situ* is done simply by referring to them, e.g. `A`, or `A (join(...), ...)`. Defining them separately would be done thus: `A=Act`, or `A=Act (join(...),...)`. Here, `A` is the customised type name and `Act` is the generic type for basic activity types. `join(...)` is one of the optional attributes that may be attached to an activity type to express synchronisation conditions (see Section 2.5 below).

For structured activity types defined *in situ*, an explicit name for the activity type is *not* given. An example might be `Par (A,B)`, where `Par` is the (structured) generic type name, and `Par (A,B)`, the customised type *definition*. Structured activity types can also be defined separately and assigned a name, e.g. `P = Par (A,B)`. Here, `P` is the customised type *name*.

Activity types that are defined separately and not *in situ* are called *defined types*. Consider the following simple `Liesbet` model as an example.

```
Par (S1,Seq (B,C))
S1 = Seq (A,B)
```

Here, `A`, `B`, and `C` are *in situ* definitions of basic activity types; we can tell this as they are not defined types. The second argument of the `Par` is a structured activity type defined *in situ*. In contrast, the first argument, `S1`, is a defined type.

The definition of a workflow model will include just one defined type that is unnamed. This is taken to be the top-level activity of the workflow model. A workflow model is a hierarchical structure with this activity at its root. In the example, `Par(S1,Seq (B,C))` is the top-level workflow activity type.

2.1.1. Finite state machine for activity instances

The following Finite State Machine (FSM) is defined for the operation of an activity instance. An activity instance may be in one of four states – `Ready`, `Running`, `Cancelled` or `Completed`. We also consider an activity instance to be *finished*, if it is in a `Cancelled` or `Completed` state.

```
Ready – execute → Running
Ready – cancel → Cancelled
```

```
Running – complete → Completed
Running – cancel → Cancelled
```

- An activity instance begins life in the `Ready` state. At some point, the parent of the activity instance will initiate execution of the instance. The instance will be moved into the `Running` state, by virtue of the `execute` action.
- When the work of the instance is done, it is moved to the `Completed` state, by means of the `complete` action.
- From the `Ready` and `Running` states, the instance may be moved into the `Cancelled` state, by means of the `cancel` action. This will have the effect of not only immediately cancelling the activity instance itself, but also all of its descendants.

Cancellation of an activity may happen because of the execution of a `CancelActivity` instance (Section 2.8), because of a failed join condition (Section 2.5), or because of *dead-path elimination* (DPE) (Leymann and Roller 1999). DPE is performed in workflow model enactment when it is identified that an activity instance will never be executed. This happens, for instance, when executing a `Choice` activity instance. Those continuation activity instances within the `Choice` instance that correspond to unselected branches are moved to the `Cancelled` state.

2.1.2. Isolated scopes

As explained in Section 2.2, below, instances (of certain activity types) may query the state of other activity instances. However, since the enactment of a workflow model may create multiple instances of the same activity type, there is potential ambiguity about which specific instance is referred to in the query. In the example shown in Figure 2, the join condition on activity `B` queries the state of activity `C` of which there are three separate instances. `Liesbet` provides several methods for disambiguating such references, of which the *isolated scope* declaration is the most fundamental.

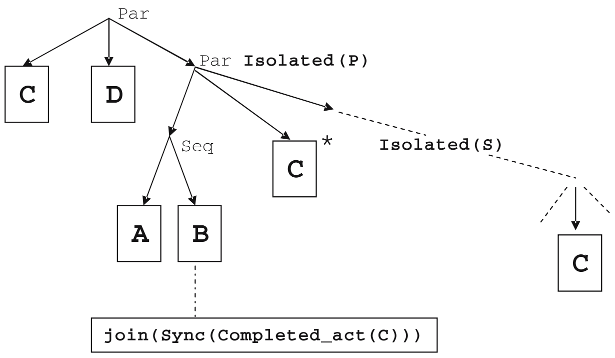

Figure 2. Isolated scopes in operation. The join condition on activity type `B` will have a visibility horizon that is restricted to the descendants of the isolated scope `P`, but not including the isolated scope `S` and its descendants. The only candidate instance of activity type `C` for the query in the join condition of `B` is thus the instance of `C` marked *.

Any activity may be marked as an *isolated scope*. In Easy Syntax this is achieved by encapsulating the definition of an activity type in the container `Isolated`. In the example below, both activity types `A` and `B` are isolated scopes but `C` is not. The scope of an activity type is not isolated, by default.

```
Par(Isolated(A), B)
A = ...
B = Isolated(...)
C = Seq(...)
```

This has the effect of creating a *visibility horizon* on the workflow state for activity instances that exist within an instance of the isolated scopes `A` and `B`.

When an instance *i* exists within the scope of another activity instance which is isolated, the instance *i* can only query the state of activity instances that are descendants of the isolated scope instance that is *the most immediate ancestor* of *i*, and this isolated scope instance itself. Moreover, if any of these descendant instances *more immediately* fall within the scope of a different isolated scope instance, then these particular instances will not be visible to the querying instance *i*. The visibility horizon for a querying instance is thus the sub-tree extending from its (immediate) ancestor isolated scope instance, from which are pruned any sub-trees extending from further isolated scope instances (as is demonstrated in the figure). We call the pruning aspect, *co-related instance pruning*.

There is another way of creating a visibility horizon for an activity instance, and that is by using so-called reference activity types in queries. These queries take (what is called) a ‘plain’ or ‘distinct’ reference type. We will not describe the use of distinct reference types in this article – instead, we refer the reader to Farrell (2006). The use of a *reference type* is similar to that of an isolated scope, in that it is used to limit the visibility horizon of querying instances, except that, in contrast to the use of isolated scopes, we may specify within individual *queries* what the visibility horizon for the query should be. That is, it is the individual *query* that determines the visibility horizon, within the visibility constraints of any isolated scopes that might exist. Note that multiple queries may be made by a single querying instance, all with different visibility horizons. As a result, we can set a much finer granularity for the visibility of certain queries, rather than setting a universal visibility horizon for a whole tree of querying instances. Figure 3 shows an example of using queries with reference types. A workflow model may use a mixture of isolated scopes and reference queries.

Note that in order to seek the most immediate ancestor of a querying instance having a particular reference type, we traverse the instance tree from the querying instance towards the root instance. If in doing so, we first encounter an activity instance that is marked as an isolated scope, then this instance is taken to be the reference type instance for the purpose of establishing the visibility horizon.

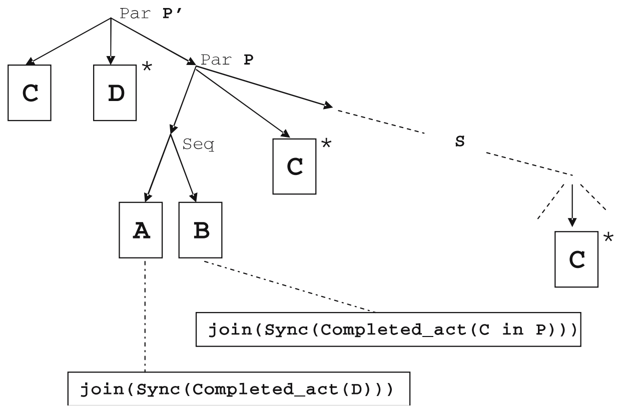
Par P'
C
D
*
Par P
Seq
C
*
S
A
B
C
*
join(Sync(Completed_act(C in P)))
join(Sync(Completed_act(D)))

Figure 3. Since P is not an isolated scope in this example, the visibility horizon for the join condition on activity type A extends beyond P, and includes the instance of type D marked *. For the join condition on activity type B, the visibility horizon is determined by means of a reference type, specified as P. Since S is not an isolated scope, the instances of C marked * will be in the visibility horizon of this instance of B.

2.2. Sync – synchronisation activity types

The synchronisation activity types of Liesbet represent *synchronisation points* in the workflow model. The most general of these constructs is Sync(StopQuery, GoQuery) in which StopQuery and GoQuery are queries on the current workflow state. There is a race between which of these queries is satisfied first, which ultimately determines whether the synchronisation activity itself completes successfully or not.

Easy Syntax
Sync(StopQuery, GoQuery)
Sync(GoQuery)

A StopQuery or GoQuery query is a *blocking* query on current workflow state that must be satisfied. That is, a query blocks until it is satisfied. A query is a boolean compound (using | for conjunction and + for disjunction) of the following (where ATN stands for Activity Type Name).

- Completed_act(ATN) – This query is satisfied if and only if an instance of the activity type ATN, within the visibility horizon of the querying instance, has completed.
- Completed_act(ATN in Ref_ATN) – This query is the same as Completed_act(ATN) except that it specifies a *plain reference type*, Ref_ATN, in order to create a visibility horizon for the query.
- Completed_all(ATN) – This query is satisfied if and only if all extant instances of the activity type ATN, within the visibility horizon of the querying instance, have completed.

- `Completed_all(ATN in Ref_ATN)` – This query is a combination of `Completed_all(ATN)` and `Completed_act(ATN in Ref_ATN)`.

Queries can also be made to ascertain the existence of activity instances in the `Cancelled` state, as well as *finished* instances (i.e., those in `Completed` or `Cancelled` states), and those *not* in a `Ready` state (i.e., those in `Completed`, `Cancelled` or `Running` states). To use such queries, the keyword `Completed` is replaced with the keywords: `Cancelled`, `Finished`, and `NotReady`, respectively.

In the following example, the query is satisfied if either an instance of activity type `A` or `B` has completed, and an instance of activity type `C` has completed.

```
((Completed_act(A) + Completed_act(B)) | Completed_act(C))
```

We may also write `True` for the query that is trivially satisfied, and `False` for the query that can never be satisfied.

Informal Operational Semantics

When an instance of the activity type `Sync(StopQuery, GoQuery)` is running, and `StopQuery` is satisfied before `GoQuery`, then the synchronisation activity instance goes to `Cancelled`. If `GoQuery` is satisfied, and `StopQuery` is not satisfied beforehand, then the synchronisation activity instance goes to `Completed`. While neither query is satisfied, the instance remains in the `Running` state.

An instance of the activity type `Sync(GoQuery)` will remain in the `Running` state until the `GoQuery` query is satisfied, whereupon it will move to `Completed`. It is thus equivalent in behaviour to `Sync(False, GoQuery)`.

Finally, note that a `Sync` type can be used to effect the YAWL workflow pattern `Milestone` (Kiepuszewski 2003; van der Aalst et al. 2004), which is *where the enabling of an activity depends on the (instance of the) workflow model being in a specified state, i.e., the activity is only enabled if a certain milestone has been reached which did not expire yet. Consider three activities named A, B, and C. Activity A is only enabled if activity B has been executed and C has not been executed yet, i.e., A is not enabled before the execution of B and A is not enabled after the execution of C.*

This synchronisation behaviour can be captured in `Liesbet` as `Sync(Finished_act(B), NotReady_act(C)))`. Here, if `C` has started executing then the `Sync` will complete successfully (go to `Completed`). However, if `B` has finished but we are yet to execute `C` then the `Sync` will go to `Cancelled`. Thus, the milestone has been reached but is yet to expire, if the `Sync` gets cancelled; but, the milestone has expired if the `Sync` gets completed. If neither occur, then `B` has not yet finished executing and `C` has not started, thus, the milestone is yet to be reached. If we author a `DefaultChoice` construct that executes different behaviours depending on whether the `Sync` gets completed or cancelled, then we may effect the `Milestone` behaviour we desire – `DefaultChoice(Sync(Finished_ act(B), NotReady_act(C)), MilestoneStop, MilestoneGo)`. Here, we execute `MilestoneGo` if the `Sync` gets cancelled – the milestone is satisfied. But, we execute `MilestoneStop` if the `Sync` completes, as the milestone has expired.

2.3. Seq and SeqCancel – sequence

The `Liesbet` constructs `Seq/SeqCancel` are a direct facilitation of the YAWL workflow pattern *Sequence* (Kiepuszewski 2003; van der Aalst et al. 2004), which is where *an activity in a workflow process is enabled after the completion of another activity in the same process.*

We also support an unordered sequence construct, which in YAWL is called the *interleaved parallel routing* construct, but do not present details here. The interested reader should consult Farrell (2006).

Easy Syntax

`Seq(Act`$_1$, ..., `Act`$_n$)
`SeqCancel(Act`$_1$, ..., `Act`$_n$)

Informal Operational Semantics

When a sequence (`Seq/SeqCancel`) instance is running, it executes each constituent activity in the order specified, waiting for each to get to a finished `Completed` or `Cancelled` state. For `Seq`, if a constituent activity is cancelled, then the sequence continues as normal; for `SeqCancel`, the sequence is cancelled. When the last constituent activity finishes, `Seq` goes to `Completed`, and `SeqCancel` goes to `Completed` if the last constituent activity completed successfully and to `Cancelled` otherwise.

2.4. Par – Parallel

The `Liesbet` construct `Par` is a direct facilitation of the YAWL workflow pattern *Parallel* (Kiepuszewski 2003; van der Aalst et al. 2004), which is *a point in the workflow model where a single thread of control splits into multiple threads of control which can be executed in parallel, thus allowing activities to be executed simultaneously or in any order.*

Easy Syntax

`Par(Act`$_1$, ..., `Act`$_n$)

Informal Operational Semantics

When the parallel instance is running, it starts the execution of each child instance in parallel. Once all have reached a finished state (`Completed` or `Cancelled`), the parallel instance goes to `Completed`. Note therefore that cancelling child instances does not cancel the `Par` activity.

2.5. Activity join and transition conditions

An activity definition in `Liesbet` may optionally specify a *join condition* and/or a *transition condition* for the activity type.

A join condition is used to specify conditions under which execution of an activity can occur. When execution of an activity instance is initiated, the join condition, if specified, is evaluated. If the join condition is satisfied, then the instance is executed (moves to `Running`); if the condition is not satisfied, the activity instance is cancelled.

A join condition can be any activity type, although it would rarely be anything but a synchronisation activity type (i.e., a `Sync`). Activity types that are used as join conditions may not themselves specify join (nor transition) conditions.

A transition condition for an activity `A` is used to specify synchronisation conditions that must be evaluated after `A` has finished executing. Whereas a join condition can be any activity type, a transition condition must be a `Par` activity type, which will encapsulate the synchronisation activity types.

Easy Syntax

Join and transition conditions, when specified, sit to the right of an activity type definition. They are given in a separate set of parentheses, and are enclosed in the containers `join` and `trans`. There are thus three possible forms (besides an activity definition without join and transition conditions).

```
A(join(AJoin))
AJoin = ...

A(trans(ATrans))
ATrans = Par(...)

A(join(AJoin),trans(ATrans))
AJoin = ...
ATrans = Par(...)
```

Informal Operational Semantics

An activity type with a join condition should be considered as being equivalent to a `SeqCancel` activity type containing (in order) the join condition activity type and the actual activity type. This realises the desired behaviour, namely: that if the join condition does not complete successfully, the activity instance that it is attached to is not executed. If a transition condition is specified, then the join condition (if any) and the actual activity type are run first, followed by the transition condition `Par`. Even if the join condition or the instance of the actual activity type get cancelled, the transition condition will still be evaluated.

In summary, the following mappings should be applied, at the level of the meta-model (that is, at the information view). Note that as there exist mappings for join and transition conditions at the level of the meta-model, they do not demand specific treatment within a semantic characterisation, such as that provided by LCCS.

- `A(join(AJoin),trans(ATrans))` maps to `Seq(SeqCancel(AJoin, A), ATrans)`
- `A(join(AJoin))` maps to `SeqCancel(AJoin, A)`
- `A(trans(ATrans))` maps to `Seq(A, ATrans)`

where `ATrans` is always of the form `Par(...)`

The root activity of a Liesbet workflow model is not permitted to have join, nor transition, conditions.

2.6. DefaultChoice, Choice – Exclusive Choice with and without default

The Liesbet constructs DefaultChoice/Choice are a direct facilitation of the YAWL workflow pattern *Exclusive Choice* (Kiepuszewski 2003; van der Aalst et al. 2004), which is *a point in the workflow model where, based on a decision or workflow control data, one of several branches is chosen.*

Easy Syntax
DefaultChoice(Guard$_1$, ContAct$_1$; ...; Guard$_n$, ContAct$_n$; ContAct$_d$)
Choice(Guard$_1$, ContAct$_1$; ...; Guard$_n$, ContAct$_n$)

Informal Operational Semantics
Each Guard$_i$ is a *guard* activity type and ContAct$_i$ is a *continuation* activity type. A guard will usually be a synchronisation activity type (Section 2.2), although it could actually be any activity type. For example, Empty, which is an activity type that trivially completes (see Section 2.9), can be used to effect a non-deterministic choice.

The first guard instance that goes to Completed initiates its corresponding continuation instance. All other continuation instances go to Cancelled. In the case of DefaultChoice, if all of the Guard$_i$ activities go to Cancelled, then an instance of the default continuation activity type, ContAct$_d$, is executed. In the case of Choice, which has no default activity type, the Choice will itself go to Cancelled. The DefaultChoice/Choice instance completes once the executed continuation instance has finished.

2.7. MultiChoice – Multiple Choice

The MultiChoice construct is a direct facilitation of the YAWL workflow pattern *Multiple Choice* (Kiepuszewski 2003; van der Aalst et al. 2004), which is *a point in the workflow model where, based on a decision or workflow control data, a number of branches are chosen.*

Easy Syntax
MultiChoice(Guard$_1$, ContAct$_1$; ...; Guard$_n$, ContAct$_n$)

Informal Operational Semantics
MultiChoice is similar to Choice, except that there is no race between guard instances to complete first. For MultiChoice, those guard instances that complete successfully have their corresponding continuation instances executed. Those that go to cancelled have their corresponding instances cancelled.

2.8. CancelActivity

The `Liesbet` construct `CancelActivity` is a direct facilitation of the YAWL workflow pattern *Cancel Activity* (Kiepuszewski 2003; van der Aalst et al. 2004), which is where *an activity is cancelled.*

Easy Syntax

```
CancelActivity(CancelAct)
CancelActivity(CancelAct in RefAct)
```

Informal Operational Semantics
A `CancelActivity` instance will cancel all running (i.e., `Running`) and all possible future running (i.e., `Ready`) instances of the named activity type, `CancelAct`, within its visibility horizon. Optionally, `CancelActivity` may specify a reference type (see Section 2.1.2) to constrain the visibility horizon.

2.9. Empty

Do nothing but trivially complete! Useful, for example, for an empty default branch in a `Default Choice` activity.

Easy Syntax

```
Empty
```

2.10. FreeChoice

Non-deterministically complete or cancel.

Easy Syntax

```
FreeChoice
```

2.11. Support for YAWL workflow patterns

In Table 1, we present an overview of how the `Liesbet` meta-model supports the YAWL workflow patterns (van der Aalst et al. 2004).

3. Three-Part Liesbet example

In this section, we illustrate the use of the `Liesbet` meta-model by means of a three-part *Travel Agency* example. We have sought to keep this example simple, as its main purpose is to give a general impression of how processes may be modelled with `Liesbet`. We do not seek to cover, in this section, the full range of constructs supported by `Liesbet`. The main source of requirements for `Liesbet` is the YAWL workflow patterns – in Section 2.11, we summarise how `Liesbet` supports these. In Farrell (2006), we present further examples.

Table 1. Satisfaction of the YAWL Workflow Patterns (Kiepuszewski 2003; van der Aalst and ter Hofstede 2005).

Workflow Pattern	Satisfied How?
1. Sequence	`Seq`
2. Parallel Split	`Par`
3. Synchronisation, A.k.a. AND-JOIN	Yes (See Note 1)
4. Exclusive Choice	`DefaultChoice`, `Choice`
5. Simple Merge, A.k.a. XOR-JOIN	Yes (See Note 1)
6. Multiple Choice	`MultiChoice`
7. Synchronising Merge, A.k.a. OR-JOIN	Yes (See Note 1)
8. Multimerge	`Multimerge` (see Farrell (2006))
9. Discriminator	`Discriminator` (see Farrell (2006))
10. Arbitrary Cycles	Yes (See Note 2)
11. Implicit Termination	Yes (See Note 3)
12. Multiple Instances (MIs) Without Synchronisation	`MultiLimit`, `Multi` or `Par` (see Farrell (2006))
13. MIs With A Priori Design Time Knowledge	`Par` (see Farrell (2006))
14. MIs With A Priori Run Time Knowledge	`MultiLimit` or `Multi` (see Farrell (2006))
15. MIs Without A Priori Run Time Knowledge	`MultiLimit` or `Multi` (see Farrell (2006))
16. Deferred Choice	`DeferredChoice` (see Farrell (2006))
17. Interleaved Parallel Routing (Unordered Sequence)	`UnorderedSequence` (see Farrell (2006))
18. Milestone	`Sync`
19. Cancel Activity	`CancelActivity`
20. Cancel Case	`Exit` (see Farrell (2006))

Notes
1. Supported in multiple ways:
   a. Implicit Synchronisation when activity completes.
   b. Arbitrary Synchroniser can run in parallel.
2. An arbitrary cycle is a cycle with multiple sources and/or sinks (as described in Kiepuszewski (2003)). `Liesbet` supports the definition of arbitrary cycles, as any use of activity types within a `Liesbet` model definition may create a cycle. However, a `Liesbet` workflow fragment with arbitrary cycles may be converted to one consisting solely of multiple-instance activity instances. The algorithm for this conversion is trivial but is omitted here for brevity. The use of arbitrary cycles, however, should be carefully marshalled by an authoring tool, as their intuitive meaning is often unclear. It is assumed that an authoring tool will perform a conversion of arbitrary cycles to multiple-instance activity types so that its output will have such cycles removed. Thus, in considering issues such as verification, we can assume an absence of arbitrary cycles.
3. Implicit termination (from Kiepuszewski (2003)): *A given subprocess should be terminated when there is nothing else to be done. In other words, there are no active activities in the subprocess and no other activity can be made active (and at the same time the subprocess is not in deadlock).*
`Liesbet` operates on the basis of implicit termination.

3.1. Travel agency

Adapted from PMG (http://www.petripi.org):

> Consider a fragment of the process of booking trips involving six steps: *Register, (Booking of) Flight, (Booking of) Hotel, (Booking of) Car, Pay* and *Cancel.* The process starts with activity *Register* and ends with *Pay* or *Cancel.* The activities *Flight*, *Hotel* and *Car* may succeed or fail.

Presented in the following sub-sections are a number of variants of the Travel Agency scenario.

3.1.1. Travel Agency 1

Adapted from PMG (http://www.petripi.org)

> Every trip involves a flight, hotel and car and these are booked in parallel, having registered with the travel agent. If all three succeed, the payment follows. Otherwise activity cancel is executed. Cancel is delayed until all three bookings succeed/fail and does not withdraw work.

We now present a solution, using the Liesbet meta-model.

```
PaySync = Sync(Completed_act(Flight) | Completed_act(Hotel) |
                                       Completed_act(Car))
CancelSync = Sync(Cancelled_act(Flight) + Cancelled_act(Hotel)
                                        + Cancelled_act(Car))
PayCancelChoice = Choice(PaySync,Pay; CancelSync,Cancel)
Book = Par(Flight,Hotel,Car)

Seq(Register,Book,PayCancelChoice)
```

Here, we execute basic activity Register and structured activities Book and PayCancelChoice in sequence. Book consists of the basic activities of booking a Flight, a Hotel and a Car, which are carried out in parallel. Once Book has finished, PayCancelChoice is executed. It uses two Sync activities, which by definition will not both succeed (go to Completed) nor both fail (go to Cancelled). If PaySync succeeds then all booking attempts must have completed successfully and we execute the basic activity Pay. Otherwise CancelSync will succeed and the basic activity Cancel will be executed. The purpose of Cancel might be to carry out some house-keeping, such as updating the agency's database records.

3.1.2. Travel Agency II

Adapted from PMG (http://www.petripi.org)

> Every trip involves a flight, hotel and car and these are booked in parallel, having registered with the travel agent. If all three succeed, the payment follows. Otherwise activity cancel is executed. Activity cancel should be executed the moment the first activity fails and, at the same time, all outstanding booking activities should be withdrawn.

We now present a solution, using the Liesbet meta-model.

```
PaySync = Sync(Completed_act(Flight) | Completed_act(Hotel) |
                                       Completed_act(Car))
CancelSync = Sync(Cancelled_act(Flight) + Cancelled_act(Hotel)
                                        + Cancelled_act(Car))
Withdraw = Par(CancelActivity(Book), Cancel)
PayCancelChoice = Choice(PaySync,Pay; CancelSync,Withdraw)

Book = Par(Flight,Hotel,Car)

Par(Seq(Register,Book),PayCancelChoice)
```

This is a variant of the first travel agency solution. It is the same except that the choice of whether to pay or cancel is made in parallel with the Book activity, meaning that Book may be cancelled once any of the booking attempts fail. That is, if at any time CancelSync succeeds the structured activity Withdraw will be executed. This has the effect of executing the basic activity Cancel and, in parallel, cancelling the booking activity Book by means of the Liesbet construct CancelActivity.

3.1.3. Travel Agency III

Adapted from PMG (http://www.petripi.org)

> Every trip may involve a flight, hotel and/or car and these are booked in parallel, having registered with the travel agent. A trip should involve at least a flight, hotel or car but may be any combination of the three bookings, e.g., a flight and car but not a hotel. If all bookings succeed, the payment follows. Otherwise activity cancel is executed. Activity cancel should be executed the moment the first activity fails and, at the same time, all outstanding booking activities should be withdrawn.

We now present a solution, using the Liesbet meta-model. We do not define the synchronisation activities BookFlightSync, BookHotelSync, and BookCarSync here. Their definitions are straightforward and are omitted for brevity.

```
PaySync = Sync((Completed_act(Flight) +
                 Cancelled_act(BookFlightSync)) |
                (Completed_act(Hotel) +
                 Cancelled_act(BookHotelSync))  |
                (Completed_act(Car) +
                 Cancelled_act(BookCarSync)))

CancelSync = Sync((Cancelled_act(Flight) |
                    Completed_act(BookFlightSync)) +
                   (Cancelled_act(Hotel) |
                    Completed_act(BookHotelSync))  +
                   (Cancelled_act(Car) |
                    Completed_act(BookCarSync)))
```

```
Withdraw = Par(CancelActivity(Book), Cancel)
PayCancelChoice = Choice(PaySync,Pay; CancelSync,Withdraw)

Book = MultiChoice(BookFlightSync,Flight;
                   BookHotelSync,Hotel;
                   BookCarSync,Car)

Par(Seq(Register,Book),PayCancelChoice)
```

This is a variant of the second travel agency solution. It is largely the same except that we make a `MultiChoice` for the booking activity, `Book`, meaning that not all booking activities, `Flight`, `Hotel`, `Car`, have to be executed. As such, `PaySync` and `CancelSync` are adjusted accordingly, to account for booking activities not being executed. The activity `Pay` will eventually be executed, unless one of the booking activities we do execute fails – then, `Withdraw` is executed instead.

4. CCS-based Characterisation of Liesbet

We now present the CCS-based (Milner 1989, 1999) language (or ontology in the parlance of Section 1), *LCCS*, that we shall use to give a semantic characterisation to `Liesbet`. It should be emphasised that LCCS has been conceived as an abstract machine language whose purpose is to give a somewhat informal account of the semantics of `Liesbet`, in order to promote ease of understanding. We provide a precise characterisation of the semantics of `Liesbet` using standard CCS in an accompanying technical report (Farrell 2006).

In the following, we present a brief overview of (standard) CCS and the abstract machine language, LCCS, and then proceed to describe how we have used LCCS to give a characterisation of the `Liesbet` constructs introduced in Section 2.

4.1. An overview of CCS

We present a brief overview of some of the main features of CCS. For readers unfamiliar with CCS, (Bruns 1997; Milner 1989, 1990, 1999) are excellent starting points.

First, we assume the availability of an infinite set of action names **N**, ranged over by *a, b*, ..., and a corresponding set of co-names (or, co-actions) $\mathbf{N}^{-} = \{a^{-} | a \in \mathbf{N}\}$, where **N** and $\mathbf{N}^{-}$ are disjoint and in bijection via $^{-}$, and where $a^{=} = a$. The set $\mathbf{L} = \mathbf{N} \cup \mathbf{N}^{-}$ is the set of *labels*, ranged over by l and l^{-}, and τ is a distinguished *silent action*, such that $\tau \notin \mathbf{L}$. The set $\mathbf{Act} = \mathbf{L} \cup \{\tau\}$ is the set of actions that may be performed by a CCS *agent*. We assign α, β, γ ... to range over **Act**.

The set **E** of CCS *agents* is defined inductively. It is the smallest set which includes the following expressions, where E and E_i are already in **E** (from Milner (1989)):

- agent constants
- $\alpha.E$ – *prefix* $(\alpha \in \mathbf{Act})$

- $\sum_{i \in I} E_i$ – *summation*, where the indexing set I may be empty, in which case we write **0** to indicate the deadlocked agent
- $E_1|E_2$ – *composition*
- $E\backslash L$ – *restriction* $(L \subseteq \mathbf{L})$
- $E[f]$ – *relabelling*. The relabelling function $f : \mathbf{Act} \rightarrow \mathbf{Act}$ relabels action names, where $f(l^{-}) = f^{-}(l)$ and $f(\tau) = \tau$

An *agent constant* is an agent whose meaning is given by a defining equation. In the definition $A =_{\mathrm{def}} E$, A is an agent constant, and E an agent. The definitional mechanism is the means by which recursive behaviour may be defined.

Note that, for the *Concurrency Workbench of the New Century* (`CWB-NC`) (http://www.cs.sunysb.edu/~cwb), which is a tool that we use for verification (see Section 5), ‘a denotes a^{-}, an output on a, and `nil` denotes **0**. Also, the keyword `proc` is used in `CWB-NC` for the definition of agent constants.

In CCS, a system is characterised by a number of agents which may perform transitions. Note that we often use the term τ-*synchronisation* for τ-transitions, in order to emphasise the notion of *synchronisation*. (We prefix the word *synchronisation* with τ in order to avoid ambiguity with the use, in this article, of the word *synchronisation* in the context of `Liesbet`’s *synchronisation activity types*.) The transitions that a system may make define a *labelled transition system* $(\mathbf{E}, \mathbf{Act}, \{\rightarrow^{\alpha}\})$, where $\rightarrow^{\alpha} \subseteq \mathbf{E} \times \mathbf{E}$ is a *transition relation* for each $\alpha \in \mathbf{Act}$ (Milner 1989). The semantics for the set of agents, **E**, is given by the definition of each transition relation $\rightarrow^{\alpha}$ over **E**. The following set of transition rules enable us to build the transition relations over each agent in **E**, using ACT to begin with.

$$\overset{\texttt{ACT}}{\frac{}{\alpha.E \overset{\alpha}{\rightarrow} E}} \qquad \overset{\texttt{SUM}}{\frac{E_j \overset{\alpha}{\rightarrow} E'_j}{\sum_{i \in I} E_i \overset{\alpha}{\rightarrow} E'_j}} j \in I$$

$$\overset{\texttt{COM}_1}{\frac{E \overset{\alpha}{\rightarrow} E'}{E|F \overset{\alpha}{\rightarrow} E'|F}} \qquad \overset{\texttt{COM}_2}{\frac{F \overset{\alpha}{\rightarrow} F'}{E|F \overset{\alpha}{\rightarrow} E|F'}} \qquad \overset{\texttt{COM}_3}{\frac{E \overset{l}{\rightarrow} E' F \overset{\bar{l}}{\rightarrow} F'}{E|F \overset{\tau}{\rightarrow} E'|F'}}$$

$$\overset{\texttt{RES}}{\frac{E \overset{\alpha}{\rightarrow} E'}{E\backslash L \overset{\alpha}{\rightarrow} E'\backslash L}} (\alpha, \bar{\alpha} \notin L) \qquad \overset{\texttt{REL}}{\frac{E \overset{\alpha}{\rightarrow} E'}{E[f] \overset{f(\alpha)}{\rightarrow} E'[f]}} \qquad \overset{\texttt{CON}}{\frac{E \overset{\alpha}{\rightarrow} E'}{A \overset{\alpha}{\rightarrow} E'}} (A \overset{def}{=} E)$$

- ACT allows us to infer transitions for prefixed agents. That is, the agent $\alpha.E$ may make a transition labelled with action α to the agent E
- Given an agent E_j which makes an α-labelled transition to agent E'_j, we may, by SUM, infer an α-labelled transition for a summation agent $\sum_{i \in I} E_i$, where $j \in I$, such that it too transitions to E'_j

- Given an agent E which makes an α-labelled transition to agent E', we may, by COM_1, infer an α-labelled transition for a composed agent $E|F$ such that it transitions to $E'|F$. Similarly, COM_2 allows us to infer an α-labelled transition for the right-hand agent in a composition
- Given two agents E and F that make complementary α-labelled transitions to E' and F, respectively, we may, by COM_3, infer a τ-synchronisation for the composed agent $E|F$ to $E'|F'$
- Given an agent E which makes an α-labelled transition to agent E', we may, by RES, infer an α-labelled transition for the L-restricted agent $E\backslash L$ so long as α or its co-action, is not in L. The restriction $\backslash L$ has the effect of restricting the *scope* of an action in E, when named in L, to be E
- Given an agent E which makes an α-labelled transition to agent E', we may, by REL, infer an $f(\alpha)$-relabelled transition from $E[f]$ (which is the result of relabelling names comprising agent E by f) to $E'[f]$ (which is the result of a similar f-relabelling of E').
- Given an agent E which makes an α-labelled transition to agent E', we may, by CON, infer an α-labelled transition for A to E' just in case A is an agent constant whose definition is E

4.2. LCCS abstract machine language

The LCCS abstract machine language is conceived to augment CCS with *a library of agent constants*, namely: state-tracking agents for activities, scheduling agents for structured and basic activity instances, and agents pertaining to the generic activity types of structured activity types (i.e. `Seq`, `Par`, `Choice`, etc). All of these have an elaboration in the CCS characterisation of `Liesbet`, presented in Farrell (2006).

LCCS defines a countably infinite number of $\texttt{Tracker}^n$ agents, which serve to maintain the current state of activity instances (that is, `Running`, `Ready`, `Completed`, or `Cancelled`). n is the number of structured activity instances that the pertaining instance has as children. $\texttt{Tracker}^n$ agents define a number of channels, which can be used to query the current state of the agent, and to progress the state of the agent, specifically:

- `exec` (resp., `comp`, `canc`) – used to instruct the instance to move to the `Running` (resp. `Completed`, `Cancelled`) state
- `runn` (resp. `cotd`, `cald`, `find`, `nread`) – used to query whether the instance is a `Running` (resp. `Completed`, `Cancelled`, *Finished* or not `Ready`) state
- `yes`, `no` – used to answer state queries.

In defining a semantics for `Liesbet`, it is appropriate to consider that the progression of structured activity instances takes priority over the completion, or cancellation, of basic activity instances. That is to say, on executing the root instance of a workflow model, we advance the model internally as far as we can go. Then, we offer the *basic activity instances* that are now in a Running state to the environment. The environment may (in time) signal the completion, or cancellation, of *one* of these instances. Then, again, we progress the model internally as far we can go, and proceed, once more, to offer the current set of running basic instances to the environment to select one to complete, or cancel. This procedure continues until all instances have reached a finished state.

In view of this intended behaviour, we must progress structured activities ahead of progressing basic activities in the model. As a result, we need some way of specifying the *priority* of structured instances over basic instances. As there is no notion of priority in CCS, we code an explicit scheduler into the semantics. In an alternative characterisation using Cleaveland's Prioritised CCS (PCCS) (Cleaveland et al. 1996), which we present in Farrell (2006), we make use of the facility to specify priorities and, thus, avoid the need for an explicit scheduler.

LCCS defines (a countably infinite number of) scheduler agents, $\texttt{Scheduler}^s$, where s is the number of structured activity instances that exist within a model. It has channels lock, idle, prog and reset.

- lock – may be used by an agent pertaining to the generic activity type of a structured instance (e.g. Seq2 – see Section 4.3) to grab execution
- idle (resp. prog) – an agent that has grabbed execution using lock may signal that it has no work to do (resp. has done some work) using idle (resp. prog)
- reset – an agent that has signalled that it has idled (using idle) may be reset, in order that it may grab execution again, using reset

Whenever the scheduler agent receives s synchronisations on idle, without any structured instance indicating progression, this indicates that all of the structured instances are currently idling. In this case, one of the basic instances may either go to Completed, or Cancelled. To realise this logic, LCCS defines a $\texttt{Basics}^b$ agent, where b is the number of basic activity instances that exist within a model. Alternatively, if a structured instance reports progression, then all of the structured instances are reset so that they may attempt to grab the execution lock again.

In the LCCS-characterisation, any structured instance may attempt to grab the lock at any time, unless it has done so already and has not been reset. An alternative, and (arguably) equally valid semantics, is to impose some order by which structured instances may claim the execution lock. This would lead to a reduced state space, which would be desirable for more efficient verification.

LCCS also defines the agents Comp and Idle. The agent pertaining the generic activity type of a structured instance will eventually evolve to Comp, in order that the $\texttt{Tracker}^n$ agent pertaining to the structured instance may be told to move into a Completed state. Note that such completion will not occur until all of the children of the pertaining instance have themselves moved to a *finished* state. The agent will then evolve into a recurring `Idle` agent, whose purpose is to grab the execution lock, but report idle. This occurs in order that all structured instances report their status prior to allowing a basic instance to complete or cancel.

Finally, LCCS defines a number of other agents, which pertain to `Liesbet`'s generic activity types for structured activities, namely `Seq`, `Par`, `Choice` etc. We shall present the (standard) CCS definitions of these agents, in the following section, in order to give a flavour of the CCS semantic characterisation.

4.3. LCCS-characterisation of Liesbet workflow models

We provide `Liesbet` models with an operational semantics by giving their translation to LCCS. The translation process is documented in full in Farrell (2006). For the purposes of this article, it is sufficient to present the translation informally. It is also worth noting that we have implemented a translator for `Liesbet` models, which outputs their characterisations in CCS, so that properties of the models may be verified using `CWB-NC`.

For the translation process, we start at the root of the given `Liesbet` workflow model, and work our way down the workflow tree in a depth-first fashion. The translation process assumes that there will be no cycles in the workflow model. As documented in Section 2.11, it is a reasonable stipulation to make that `Liesbet` models are free from cycles. The workflow engine that we have implemented implements a check for such cycles.

For every structured activity instance, a `Tracker`^{n} agent will be inserted into the LCCS workflow specification, to run in parallel (|), at the top level. Also, there will be inserted an agent pertaining to the generic activity type of the activity instance, such as `Seq2` for a sequence with two children. Both of these agents have their channels suitably relabelled, in order to connect them together, and to connect them to the scheduler agent (see Section 4.2). For every basic activity instance, there is a suitably relabelled `Tracker0` agent, which is set to run, in parallel, at the top level.

We now proceed to present the definitions of the agents for the generic activity types. There will be a definition of `Seq`^{n}, in the translation of a workflow model, for each distinct number of child types of a `Seq` `Liesbet` activity type. In the following example, one of the `Seq`s has two children – as such, it is called `Seq2`. Note that the child types are executed in decreasing order. Thus, the first item in a sequence should be labelled *n*, for a sequence with *n* child instances; the next *n*–1, and so on. This convention simplifies the definition of the agents, which can be seen from studying `Seq3`. For this activity type, we are able to make use of the definition of `Seq2f`. A definition for a 4-child sequence would make use of the definition of `Seq3f`, and so on.

```
proc Seq2 =
   lock.'cald.(yes.'idle.reset.Idle +
               no.'runn.(yes.'exec2.'prog.Seq2f +
                         no.'idle.reset.Seq2))

proc Seq2f =
   lock.'find2.(yes2.'exec1.'prog.Comp +
                no2.'idle.reset.Seq2f)

proc Seq3 =
   lock.'cald.(yes.'idle.reset.Idle +
               no.'runn.(yes.'exec3.'prog.Seq3f +
                         no.'idle.reset.Seq3))

proc Seq3f =
   lock.'find3.(yes3.'exec2.'prog.Seq2f +
                no3.'idle.reset.Seq3f)

proc SeqCancel2 =
   lock.'cald.(yes.'idle.reset.Idle +
               no.'runn.(yes.'exec2.'prog.SeqCancel2f +
                         no.'idle.reset.SeqCancel2))

proc SeqCancel2f =
   lock.'cotd2.(yes2.'exec1.'prog.Comp +
                no2.'cald2.(yes2.'canc.yes.'prog.Idle +
                              no2.'idle.reset.SeqCancel2f))
```

The definition for a Seq^n agent executes its first child instance, (labelled *n*), and then effects its corresponding 'finishing' agent, namely, Seq^{n}f to effect the remaining logic. That being, to wait for the first instance to finish, and, then, to execute the next one. After that, the logic for Seq^{n-1}f is exposed. And so on, until we reach Seq2f, whereon, we wait for the penultimate instance to finish, and then execute the last. Following that, we expose Comp, to complete the instance (see Section 4.2). SeqCancel^n differs from Seq^n, in that whenever a child instance is cancelled, the SeqCancel^n instance is cancelled.

Similarly, there will be a version of Par^n in the translation of the workflow model for each distinct number of child types of a Liesbet Par. The definition of Par2 for parallel types with two children is shown. It executes all children together (in the same execution window).

```
proc Par2 =
   lock.'cald.(yes.'idle.reset.Idle +
               no.'runn.(yes.'exec1.'exec2.'prog.Comp +
                         no.'idle.reset.Par2))
```

There will be distinct versions of `DefaultChoice`^{n} and `DefaultChoice`^{n} `f` for each distinct number of continuation child types (not including the default) of a `Liesbet DefaultChoice`.

In `DefaultChoice2f`, which is exposed once we ascertain that the choice activity type has been put into a running state (by its parent) and have set the *guard instances* of the choice type running, we check to see whether any of the guard instances have completed. If so, we expose `DefaultChoice2fComp`, which serves to execute a continuation instance pertaining to one of the completed guard instances. It also cancels the remaining continuation instances (including the default instance). If, on the other hand, none of the guard instances have completed, but, commensurately, none of them are running either, then all of them must have been cancelled. In this case, we execute the default continuation instance. If none of these possibilities obtain, we expose another copy of `DefaultChoice2f`.

```
proc DefaultChoice2 =
   lock.'cald.(yes.'idle.reset.Idle +
             no.'runn.(yes.'exec1g.'exec2g.'prog.DefaultChoice2f +
                       no.'idle.reset.DefaultChoice2))

proc DefaultChoice2f =
   lock.'cotd1g.(yes1g.DefaultChoice2fComp +
                 no1g.'cotd2g.(yes2g.DefaultChoice2fComp
                                      +
                   no2g.'runn1g.(yes1g.'idle.reset.DefaultChoice2f
                                       +
                   no1g.'runn2g.(yes2g.'idle.reset.DefaultChoice2f
                                       +
            no2g.'canc1c.yes1c.'canc2c.yes2c.'execd.'prog.Comp))))

proc DefaultChoice2fComp =
   ( 'cotd1g.(yes1g.('win.'exec1c.'tidy.nil +
                    'lose.'canc1c.yes1c.'tidy.nil) +
             no1g.'lose.'canc1c.yes1c.'canc1g.yes1g.'tidy.nil)
   | 'cotd2g.(yes2g.('win.'exec2c.'tidy.nil +
                    'lose.'canc2c.yes2c.'tidy.nil) +
             no2g.'lose.'canc2c.yes2c.'canc2g.yes2g.'tidy.nil)
   | win.tidy.lose.tidy.'cancd.yesd.'prog.Comp)\{win,lose,tidy}
```

For `Choice`^{n}, which has no default continuation instance, we do much the same. However, in the case that all guard instances get cancelled, we cancel the choice instance, as shown.

```
proc Choice2 =
   lock.'cald.(yes.'idle.reset.Idle +
            no.'runn.(yes.'exec1g.'exec2g.'prog.Choice2f +
                     no.'idle.reset.Choice2))

proc Choice2f =
   lock.'cotd1g.(yes1g.Choice2fComp +
                no1g.'cotd2g.(yes2g.Choice2fComp
                               +
                  no2g.'runn1g.(yes1g.'idle.reset.Choice2f
                                 +
                  no1g.'runn2g.(yes2g.'idle.reset.Choice2f
                                 +
       no2g.'canc1c.yes1c.'canc2c.yes2c.'canc.yes.'prog.Comp))))

proc Choice2fComp =
   ( 'cotd1g.(yes1g.('win.'exec1c.'tidy.nil +
                     'lose.'canc1c.yes1c.'tidy.nil) +
              no1g.'lose.'canc1c.yes1c.'canc1g.yes1g.'tidy.nil)
   | 'cotd2g.(yes2g.('win.'exec2c.'tidy.nil +
                     'lose.'canc2c.yes2c.'tidy.nil) +
              no2g.'lose.'canc2c.yes2c.'canc2g.yes2g.'tidy.nil)
   | win.tidy.lose.tidy.'prog.Comp)\{win,lose,tidy}
```

There will be distinct versions of MultiChoice^n and MultiChoice^{n}f for each distinct number of continuation child types of a Liesbet MultiChoice. For MultiChoice^n, once it is running, and we have executed its guard instances, we proceed to MultiChoice^n f, whereon, we check for continuation instances that are still in the ready state. For those that are, we check their guard instances and act appropriately – for those which have now completed successfully, we execute their corresponding continuation instances, for those which have been cancelled, we cancel their corresponding continuation instances, and for those which are still running, we do nothing. If all guard instances have finished, we expose Comp.

```
proc MultiChoice2 =
    lock.'cald.(yes.'idle.Idle +
             no.'runn.(yes.'execg1.'execg2.'lock.MultiChoice2f +
                      no.'idle.MultiChoice2))

proc MultiChoice2f =
    lock.('nreadc1.(yesc1.'done.nil +
                 noc1.'cotdg1.(yesg1.'execc1.'done.nil
                                +
                 nog1.'caldg1.(yesg1.'cancc1.yesc1.'done.nil +
                                nog1.'done.nil))
        |
      |
       'nreadc2.(yesc2.'done.nil +
                noc2.'cotdg2.(yesg2.'execc2.'done.nil
                                  +
                nog2.'caldg2.(yesg2.'cancc2.yesc2.'done.nil +
                                 nog2.'done.nil))
      |
        done.done.'findg1.(yesg1.'findg2.(yesg2.Comp +
                                           nog2.MultiChoice2f)
                             +
                    nog1.MultiChoice2f))\{done}
```

Finally, the definitions of `FreeChoice` and `Empty` are presented. In the first case, we may either complete or cancel the instance – a non-deterministic choice. In the second case, we trivially complete the instance.

```
proc FreeChoice =
  lock.'cald.(yes.'idle.reset.Idle +
                no.'runn.(yes.('comp.yes.'prog.Idle +
                                'canc.yes.'prog.Idle) +
                          no.'idle.reset.FreeChoice)
proc Empty =
        lock.'cald.(yes.'idle.reset.Idle +
                      no.'runn.(yes.'comp.yes.'prog.Idle +
                                no.'idle.reset.Empty))
```

We defer a presentation of the semantic characterisation that we have provided for `Sync` and `CancelActivity` activity types to Farrell (2006), as it is rather involved.

4.4. A complete example

We now present a complete example: `Par(Seq(A,B), Seq(B,C))`. It is a parallel activity consisting of two sequences, which respectively consist of instances of basic activity types `A` and `B`, and `B` and `C`. Note that we have annotated the output from the translator (that we have implemented for `Liesbet` models), which already labels instances with a number, by adding information concerning the customised activity type of instances. For brevity, we have also replaced the relabelling of names for agent constants by ellipses; that is we simply write [...].

```
*************************************
* LCCS Verification Run *************
* # 0
* Generated from: file:owl/LiesbetTest.xmi
* On: Tue Mar 9 14:39:25 BST 2006
*************************************

**
* Definitions of Tracker, Scheduler and some other agents have
* been cut from here. Refer to Farrell (2006) for full example.
**

proc Seq2 =
   lock.'cald.(yes.'idle.reset.Idle +
               no.'runn.(yes.'exec2.'prog.Seq2f +
                         no.'idle.reset.Seq2))

proc Seq2f =
   lock.'find2.(yes2.'exec1.'prog.Comp + no2.'idle.reset.Seq2f)

proc Par2 =
   lock.'cald.(yes.'idle.reset.Idle +
               no.'runn.(yes.'exec1.'exec2.'prog.Comp +
                         no.'idle.reset.Par2))

proc Workflow0 =
   (
***Instance:0 **Par(Seq(A,B), Seq(B,C))
   Tracker2[…] | Par2[…] |

***Instance:1 **Seq(A,B)
   Tracker2[…] | Seq2[…]|

***Instance:2 **A
   Tracker0[…] |

***Instance:3 **B
   Tracker0[…] |

***Instance:4 **Seq(B,C)
   Tracker2[…] | Seq2[…] |

***Instance:5 **B
   Tracker0[…] |

***Instance:6 **C
   Tracker0[…] |

   Basics4[…] | Scheduler3

   )\{…}
```

5. Verification of Liesbet Models

In this section we introduce our support for static verification of the key property of *model soundness*. As documented in Farrell (2006), we also support the verification of LCCS characterised `Liesbet` models against arbitrary temporal logic specifications.

5.1. Liesbet model soundness

In the verification of LCCS-characterised `Liesbet` models, we are fundamentally concerned with the notion of model soundness, which is a property of the *control perspective*. Van der Aalst and colleagues have defined this property (Verbeek et al. 2001; van der Aalst 2004). We now present a definition of soundness based on theirs but adapted for our needs. A workflow model is sound (at the control perspective) if (and only if) it satisfies the following conditions.

- *Option to complete* – It should always be possible to complete a workflow instance
- *Proper completion* – It should not be possible that the workflow model signals completion of an instance while there is still work in progress for that instance
- *No dead activities* – For every activity instance that may be created in the enactment of a workflow model, there must exist at least one enactment path where that instance is run. This property ensures that every activity instance plays a meaningful role in the workflow model.

The first property, option to complete, stipulates that the workflow model should not be subject to locking along any of its enactment paths. Specifically, we consider two types of locking, namely, deadlock and livelock.

A `Liesbet` model is considered to be in a state of *deadlock* whenever we reach a state where some activity instances are yet to finish while not being able to progress the model any further. *Model deadlock* may be introduced into a model, through an inappropriate use of synchronisation primitives (see Farrell (2006)). Using these, we may block the execution of an instance until some model state is realised. However, if this state is never realisable, the blocking will be a source of deadlock. Such deadlock is *model specific*. There is also one other potential kind of deadlock. It is conceivable that in our semantic characterisation we have introduced sources of deadlock, for instance in the process specifications for the generic activity types. However, as explained in Farrell (2006), this other kind of deadlock does not arise.

Livelock in a LCCS `Liesbet` model would be where there are infinite executions of LCCS reductions within the model without any progress being made towards (proper) completion of the model. This is not considered to be a significant issue for LCCS-characterised `Liesbet` models. *Model livelock* pertains to the situation where we define an infinite cycle within a `Liesbet` model. A simple example of this is `X = Seq(A,X)`. As described in Section 2.11, we stipulate that cyclic behaviour be limited to that which can be expressed using multiple-activity instances (so-called structured cycles). For this example, we would use a `MultiSeq` activity type (see Farrell (2006)). For structured cycles, model livelock is not possible, under the assumption that structured cycles will eventually be exited. As documented in Farrell (2006), the possibility of livelock at the level of the LCCS characterisation can be discounted.

We define proper completion for an LCCS `Liesbet` model to be achieved when the root workflow instance and all of its descendant instances have reached finished (i.e., `Cancelled` or `Completed`) states. As, according to the LCCS semantics for `Liesbet`, the root instance may not complete until all of its child instances have entered a finished state, it is sufficient to say that, for proper completion, the root instance should be in a finished state. This will occur if, and only if, we have an absence of locking in a LCCS `Liesbet` model. As a result, verification of LCCS `Liesbet` model soundness reduces to verifying an absence of model deadlock and an absence of dead activity instances (Farrell 2006).

It is worth noting that the general procedure for verification, including that for model soundness, splits a `Liesbet` model into a number of verification runs based on isolated scopes. This is because isolated scopes can not be permeated by synchronisation primitives, and so can be verified in isolation when it comes to model soundness. A full argument for this is presented in Farrell (2006). Thus, we start with the root instance of the workflow model for the first verification run, seeking to check every path through the workflow model for deadlock and dead activity instances. Whenever an instance of an isolated scope would be created in the verification run, that instance is skipped. The scope type for that instance will be verified in a separate verification run, where it can be treated as its own workflow, with itself as the root type. The results of the verification process are the sum of the separate verification runs.

5.2. Example verification of LCCS-characterised model using concurrency workbench

We present a simple verification example, where we check for an absence of deadlock in the `Liesbet` model presented in Section 4.4: `Par(Seq(A,B), Seq(B,C))`. That is, we wish to check that along all enactment paths, the root instance (and thus the workflow model as a whole) eventually reaches a *finished* state. This is a key property to check in verifying the soundness of workflow models, as we have described in Section 5.1.

In the definition of the scheduling agent, `Scheduler`^{n} (which we describe briefly in Section 4.2), we perform an output on (unrestricted) channel `rfind`, once the root instance has reached a finished state. In order to check for an absence of deadlock, we need to verify that, on all enactment paths, an output on `rfind` eventually occurs. The test that we write is the simple modal-mu formula:

```
prop cotd =
min X = <->tt /\ [-'rfind]X
```

The output from running this test under CWB-NC for the example model is presented.

```
>cwb-nc.bat ccs
cwb-nc.bat ccs
Currently supported languages are : ccs, pccs, sccs, tccs, csp,
lotos

The Concurrency Workbench of the New Century
(Version 1.2 --- June, 2000)

cwb-nc> load test.ccs
Execution time (user,system,gc,real):(0.047,0.000,0.000,0.047)
cwb-nc> load test.mu
Execution time (user,system,gc,real):(0.000,0.000,0.000,0.000)
cwb-nc> chk Workflow0 cotd
Invoking alternation-free model checker.
Building automaton...
.........1000...
States: 1343
Transitions: 1529
Done building automaton.
TRUE, the agent satisfies the formula.
Execution time (user,system,gc,real):(16.499,0.000,3.343,16.499)
cwb-nc>
```

5.3. Observational equivalence of LCCS-characterised Liesbet models

The question of *when two workflows are equivalent* is an important issue in the study of workflow. As reported in Hidders et al. (2005), this is a non-trivial question. The crux is how to treat *internal actions* – those actions which progress the model but are not concerned with the fulfilment of (basic) activity instances. One approach to formalising workflow equivalence is that taken by Kiepuszewski (2003). There, workflows are considered to be equivalent if there exists a weak bisimulation (as defined by Milner for CCS) between them (where activity completions are considered to be the only observable actions), with the additional requirement that all enactment paths within the workflows must lead to proper completion. Notwithstanding the issues highlighted in Hidders et al. (2005), which we do not seek to resolve, for simplicity we have adopted the approach taken in Kiepuszewski (2003). (We could also include the possibility of cancelling basic activity instances as part of the set of observable actions, but omit to do so for simplicity).

The definition of *Observational Equivalence* (Milner, 1989) requires some additional notation. The transition $E \Rightarrow^{\alpha} E'$, for $\alpha \in \mathbf{Act}$, means that E may transition to E' through an α-labelled transition prefixed and postfixed by a number (including zero) of τ-transitions. That is, $E \Rightarrow^{\alpha} E'$ if $E(\rightarrow^{\tau})^{p}(\rightarrow^{\alpha})(\rightarrow^{\tau})^{q}E'$, where $p, q \geq 0$. Also, for $t \in \mathbf{Act}^{*}$, $t^{\wedge} \in \mathbf{L}^{*}$ is the sequence gained by deleting all occurrences of τ from t.

Observational Equivalence, $\approx$, is the largest symmetric relation such that whenever $E \approx F$, if $E \rightarrow^{\alpha} E'$ then $F \Rightarrow^{\alpha^{\wedge}} \approx E'$. Elaborating, two CCS agents are observationally equivalent if and only if, abstracting away from τ-transitions, whenever either agent can make an α-labelled transition the other agent can similarly perform an α-labelled transition and the resulting agents are themselves observationally equivalent.

In order to be able to define an appropriate notion of equivalence between `Liesbet` models, we need to make visible transitions pertaining to the completion of basic activities. We define two LCCS-characterised `Liesbet` models to be *model equivalent* just when they are observationally equivalent according to the offering, to the environment, of transitions pertaining to the completion of basic activities, and completion (or cancellation) of the root instance. Further details are deferred to Farrell (2006).

The concept of model equivalence is demonstrated in the following examples.

`Liesbet` **Model Equivalence, Example 1 v Strong Equivalence**

Observational equivalence is a *weaker* notion than *strong equivalence*. For strong equivalence, we do not abstract away from τ-actions. An example that highlights this distinction is the following simple one.

Let `Liesbet` Model `Workflow0` be defined as: `A`, and `Liesbet` Model `Workflow1` be defined as `Seq(A)`. These two models are *model equivalent*, as they both effect just `A`. However, they would not be equivalent if we were to define model equivalence on the basis of strong equivalence. This is because, for model `Workflow1`, there is more internal activity in encapsulating `A` within a `Seq` activity type.

If we check their observational equivalence under `CWB-NC`, we can see that they are observationally equivalent.

```
cwb-nc> load test.ccs
Execution time (user,system,gc,real):(0.031,0.000,0.000,0.031)
cwb-nc> load test.mu
Execution time (user,system,gc,real):(0.000,0.000,0.000,0.000)
cwb-nc> eq -S obseq Workflow0 Workflow1
Building automaton...
States: 59
Transitions: 57
Done building automaton.
Transforming automaton...
Done transforming automaton.
TRUE
Execution time (user,system,gc,real):(0.125,0.000,0.000,0.125)
```

But, if we check for strong equivalence, we can see that they are not found to be equivalent. Strong equivalence is too strong a notion for workflow model equivalence.

```
cwb-nc> eq -S bisim Workflow0 Workflow1
Building automaton...
States: 59
Transitions: 57
Done building automaton.
FALSE...
Workflow1a satisfies:
   <t><t><t><t><t><t><t><t><t><t>[t]ff
Workflow1b does not.
Execution time (user,system,gc,real):(0.078,0.000,0.000,0.078)
cwb-nc>
```

Liesbet **Model Equivalence, Example 2 v Trace Equivalence**

Observational equivalence is a *stronger* notion than *trace equivalence*. For trace equivalence, we are concerned solely with comparing the possible *sequences* of basic activity completion of workflow models. For observational equivalence, however, we seek to compare the choices of basic activities to complete at corresponding stages of evolution of workflow models.

An example that highlights this distinction is the following simple one. Let model `Workflow0` be defined as `Seq(A, Choice(Empty, B, Empty, C))`, and let `Workflow1` be defined as `Choice(Empty, Seq(A, B), Empty, Seq(A, C))`. For `Workflow0`, we do not make a commitment on the choice between `B` and `C` until after we have performed `A`. For `Workflow1`, in contrast, we make the choice before we execute `A`. These models do not maintain the same choices of activities to complete at corresponding points in their evolution. That is, after `A` has been completed, both `B` and `C` are available in `Workflow0`, whereas either `B` or `C` is available in `Workflow1`. However, the two models are *trace equivalent*, as they both manifest the sequences of activity completion: `A,B` and `A,C`.

If we check their observational equivalence under `CWB-NC`, we can see that they are not observationally equivalent. As reported, `Workflow0` is capable of completing either `B` or `C` after completing `A`, but `Workflow1` is not capable of this.

```
cwb-nc> eq -S obseq Workflow0 Workflow1
Building automaton...
.........1000.........2000.........3000.........4000.........5000
.........6000.........7000.........8000.........9000.........10000
.........11000.....
States: 11523
Transitions: 14647
Done building automaton.
Transforming automaton...
Done transforming automaton.
FALSE...
Workflow0 satisfies:
        <<'eyes_a>>(<<'rfind>>tt /\ <<'eyes_b>>tt /\
<<'eyes_c>>tt)
Workflow1 does not.
Execution time
(user,system,gc,real):(4529.139,941.599,3332.864,5475.923)

cwb-nc>
```

But, if we check for trace equivalence, we can see that they are found to be equivalent. Trace equivalence is too weak a notion for workflow model equivalence.

```
Execution time (user,system,gc,real):(0.024,0.004,0.004,0.179)
cwb-nc> eq -S trace Workflow0 Workflow1
Building automaton...
.........1000.........2000.........3000.........4000.........5000
.........6000.........7000.........8000.........9000.........10000
.........11000.....
States: 11523
Transitions: 14647
Done building automaton.
Transforming automaton...
Done transforming automaton.
TRUE
Execution time (user,system,gc,real):(42.283,1.676,12.633,44.286)
```

6. Related Work, Discussion and Future Work

In this work we have considered the formal representation of workflow for the purposes of performing (static) verification of certain properties, such as workflow soundness. We have thereby sought to make a contribution to the work of the *Process Modelling Group* (PMG) (http://www.petripi.org), which is concerned with understanding the behavioural nature of processes, such as business processes (which are their primary concern), and seeking to understand the utility of formal tools, such as Petri-nets (Reisig and Rozenberg 1998) and CCS/π-calculus (Milner 1989, 1999; Sangiorgi and Walker 2001), for modelling and verifying the behaviour of processes.

The YAWL Workflow Patterns are a significant contribution made by members of the PMG community (van der Aalst and ter Hofstede 2002, 2005; Kiepuszewski 2003; van der Aalst 2004). They are the result of extensive studies of many workflow languages and tools, and many representative workflow scenarios, in order to arrive at a definitive set of representational requirements, or 'patterns', for workflow. Our contribution to the PMG effort is primarily to provide a formal characterisation (i.e., at the computational view, Section 1) of the YAWL workflow patterns, and as a consequence to fix the meaning of workflow models that make use of such patterns.

Approaches to the modelling of YAWL workflow patterns include (van der Aalst and ter Hofstede 2002, 2005; Dong and Shensheng 2003, 2004; Kiepuszewski 2003; Puhlmann and Weske 2005; Stefansen 2005a, b; van der Aalst et al. 2004). van der Aalst et al. (2004) presents a *graphical* (authoring view) meta-model for the definition of workflow models, which is formally underwritten with a Petri-net inspired transition-system based semantics (the computational view).

We have presented a meta-model, called `Liesbet`, at the *information view*. This view is primarily concerned with describing concisely the sufficient and (as much as possible) necessary representational requirements for the workflow modelling approach. `Liesbet` is formally underwritten (at the computational view) by a semantic characterisation in Milner's Calculus of Communicating Systems (Milner 1989, 1990, 1999), and in a prioritised form of CCS thanks to (Cleaveland and Hennessy 1990, 1996), as presented in Farrell (2006). In this article, we have used a conceived abstract machine language, LCCS, to provide a more informal account of the semantics of `Liesbet`.

In Dong and Shensheng (2003) and Puhlmann and Weske (2005) (resp. Stefansen (2005a)) are presented π-calculus-based (resp. CCS-based) characterisations of the YAWL patterns. (Actually, these works could all be classified as being CCS-like, in that none of them make use of the channel-passing aspect of the π-calculus and thus could be viewed as using variants of CCS rather than π-calculus.)

A key difference in our work lies in the capability for arbitrary querying of workflow state facilitated by the use of state tracking agents – the $\texttt{Tracker}^n$ agents. These other approaches only support very primitive querying against workflow state, in order to facilitate the `Milestone` YAWL workflow pattern. Moreover, in our approach, in synchronising the performance of activity instances, or in cancelling activity instances, a model author can, through the use of *isolated scopes* and *reference query types*, gain a fine level of control over how activity instances are synchronised, or what instances are cancelled. The support of arbitrary synchronisation patterns, such as those highlighted in Keller et al. (2003) and Belhajjame et al. (2001), has been a requirement in our work.

These other works do not appear to provide any tool support for the purposes of verification either. We have implemented a tool which translates `Liesbet` models to Milner's CCS, or Cleaveland's PCCS, so that properties of `Liesbet` models may be verified using the *Concurrency Workbench of the New Century* (`CWB-NC`), (http://www.cs.sunysb.edu/~cwb).

We have sought to make verification under `CWB-NC` practicable by ensuring that the CCS/PCCS characterisations are as efficient as possible in their semantic characterisation of `Liesbet`. Unfortunately, both characterisations do still lead to rather inflated state spaces. It is notable that the simple example, presented in Section 5.2, had a state space, when constructed by `CWB-NC`, of **1463** states. When compared with the output from `CWB-NC` for the PCCS model of **53** states, it is quite clear to see that the use of an explicit scheduler in the CCS semantics does lead to an explosion in the size of the state space, even for such a simple model. It is notable that the minimal state space (as argued in Farrell (2006)) for this particular model is **15** states. As described in Farrell (2006), we have provided another semantic account of `Liesbet` which minimises the possible state space of a `Liesbet` model. It is this semantic account, presented in the Situation Calculus (Reiter 2001), which provides the formal basis of our workflow engine for `Liesbet`.

It is important to note a distinction between *data-driven* and *process-oriented* computational models for workflows and compositions. YAWL's semantic characterisation presented in van der Aalst et al. (2004) is data-driven, but this does not mean that the YAWL workflow patterns are necessarily best characterised using a data-driven approach.

A process-oriented workflow model, such as one based on CCS or π-calculus, will principally operate in terms of the consumption (that is, execution) of process actions. An action is scheduled for consumption whenever it reaches the head of the process specification, and as actions are consumed they are removed from the head. In a data-driven approach, such as Petri-nets generally, an activity is scheduled for execution whenever there exist token(s) in the input conditions of an activity. If tokens are fed back to these input conditions, then the activity might be executed many times. In the process-oriented approach, we would need to replicate the process definition explicitly to achieve something similar. Our LCCS characterisation of the YAWL workflow patterns is *primarily* process-oriented, although the use of $\texttt{Tracker}^n$ agents, to record instance state, also gives it a data-driven flavour, where these agents could be viewed as a *state chalkboard*.

Certain artefacts that are easily represented using a process-oriented approach may not be so easily represented using a data-driven approach, and vice versa. Indeed, criteria that we have alluded to in the Introduction to this article include the eloquence, or succinctness of the representation, as well as its intuitiveness. Perhaps, we should also add to that the requirement to be able to represent the state spaces of workflow models *minimally* as our Situation Calculus approach, documented in Farrell (2006), does. It is certainly the case, as we argue further in Farrell (2006), that the CCS and PCCS representations fall rather short on these criteria; while some of their failings are compensated for (in part) by the use of an abstract machine language, such as LCCS, which serves to abstract (by means of *defined agent constants*) from some of the detail of the technical underpinnings of the semantics, while still providing a computational view of `Liesbet` models.

There is a need to understand the nature of the business processes that we would like to represent in order to understand which is more appropriate, in which circumstances. This is a stated aim of PMG. The YAWL workflow patterns originated from researchers who are members of this group; now the group is actively looking to evolve their understanding of the behavioural nature of business processes in order to further ground studies into the use of formal tools for their representation.

It is worth noting that we have extended the `Liesbet` meta-model to be able to represent the control perspective of WS-BPEL (OASIS 2005) for the purpose of verification of soundness of WS-BPEL compositions. Approaches to the formal specification of WS-BPEL (OASIS 2005) compositions, typically for the purpose of verifying certain properties of compositions, include (Foster et al. 2003; Koshkina and van Breugel 2003; Duan et al. 2004a, b; Ferrara 2004; Fisteus et al. 2004; Fu et al. 2004; Viroli 2004; Ouyang et al. 2005; Farahbod et al. 2005; Kazhamiakin and Pistore 2005; Lucchi and Mazzara 2006). More information regarding our characterisation of WS-BPEL will be provided in a future report.

The work presented in this report is part of a larger effort looking at the planning of fulfilment strategies for (primarily, enterprise) workflows. Essentially, we use `Liesbet` models, as abstract templates, to drive the planning procedure for the realisation of a business process, where the planned strategy must satisfy a collection of constraints. We also need to verify properties of these abstract workflows, such as soundness Farrell (2006), before they can be used to guide the planning process. Although we allow an arbitrary representation for the abstract workflow models, we have sought to propose an ontology for them, and for this purpose the YAWL workflow patterns were a natural choice. An obvious alternative would be WS-BPEL (OASIS 2005), and formal models for this language could easily be used in our work instead, as could many other ontologies, such as PSL (Gruninger 2003).

In Farrell (2006), we present full details of the work described in this article.

Appendix – Liesbet (Easy Syntax) Grammar

The following is a presentation of the grammar of the Easy Syntax for `Liesbet` in BNF (Backus-Naur Form). Refer to Farrell (2006) for full details regarding `Liesbet` constructs.

```
<Liesbet_Model> ::= <Activity_Type> <Activity_Type_Defs>
<Activity_Type_Defs> ::= <Activity_Type_Def> |
                         <Activity_Type_Def> <Activity_Type_Defs>
<Activity_Type_Def> ::= <Activity_Type_Name> = <Activity_Type>
<Activity_Type> ::= <Activity>(<ActConds>)
<Activity_Type_Name> ::= α|β|γ|…
<Activity> ::= <Activity_Type_Name> | Act | <StructAct>
<ActConds> ::= join(<GuardAct>) | trans(<GuardAct>) |
               join(<GuardAct>),trans(<GuardAct>)
<StructAct> ::= <CancelActs> | <Choices> | <Merges> | <Empty> | <Exit> |
                FreeChoice | <MultiActs> | <ParSeq> | <SyncActs>
<CancelActs> ::= CancelActivity(<Activity_Type_Name>) |
CancelActivity(<Activity_Type_Name> in <Activity_Type_Name>)
<Choices> ::= DefaultChoice(<GuardContActs>; <ContAct>) |
              Choice(<GuardContActs>) | DeferredChoice(<ContActs>) |
              MultiChoice(<GuardContActs>) | MultiChoiceMin(<T>, <GuardContActs>)
<Merges> ::= Disc(<GuardActs>; <ContAct>) | Multimerge(<GuardActs>; <ContActs>)
<MultiActs> ::= <MultiLimitActs> | <MultiNoLimitActs>
<MultiLimitActs> ::= MultiLimit(<T>, <Go>, <ExecAct>) |
                     MultiSeqLimit(<T>, <Go>, <ExecAct>)
<MultiNoLimitActs> ::= Multi(<Go>, <ExecAct>) | MultiSeq(<Go>, <ExecAct>)
<ParSeq> ::= Par(<ExecActs>) |
             Seq(<ExecActs>) | SeqCancel(<ExecActs>) | UnorderedSeq(<ExecActs>)
<SyncActs> ::= Sync(<GoQuery>) | Sync(<StopQuery>, <GoQuery>)

<GuardContActs> ::= <GuardAct> <ContAct> | <GuardAct> <ContAct>; <GuardContActs>
<GuardActs> ::= <GuardAct> | <GuardAct>, <GuardActs>
<ContActs> ::= <ContAct> | <ContAct>, <ContActs>
<ExecActs> ::= <ExecAct> | <ExecAct>, <ExecActs>

<GuardAct> ::= <Activity_Type_Name> | <Activity_Type>
<ContAct> ::= <Activity_Type_Name> | <Activity_Type>
<DefAct> ::= <Activity_Type_Name> | <Activity_Type>
<ExecAct> ::= <Activity_Type_Name> | <Activity_Type>
<Go> ::= <Activity_Type_Name> | <Activity_Type>

<T> ::= 1 | 2 | ...

<GoQuery> ::= <Query>
<StopQuery> ::= <Query>
<Query> ::= <Query>|...|<Query> | <Query>+...+<Query> | QueryOnAct | True | False
<QueryOnAct> ::= <QueryOnCompletedAct> | <QueryOnCancelledAct> |
                 <QueryOnNotReadyAct> | <QueryOnFinishedAct>

<QueryOnCompletedAct> ::= Completed_Act(<Activity_Type_Name>) |
                     Completed_Act(<Activity_Type_Name> in <Activity_Type_Name>) |
                     Completed_Act(<Activity_Type_Name> in <Activity_Type_Name>
                                                  dist in <Activity_Type_Name>) |
                          Completed_All(<Activity_Type_Name>) |
                     Completed_All(<Activity_Type_Name> in <Activity_Type_Name>)

<QueryOnCancelledAct> ::= Cancelled_Act(<Activity_Type_Name>) |
                     Cancelled_Act(<Activity_Type_Name> in <Activity_Type_Name>) |
                     Cancelled_Act(<Activity_Type_Name> in <Activity_Type_Name>
                                                  dist in <Activity_Type_Name>) |
                          Cancelled_All(<Activity_Type_Name>) |
                     Cancelled_All(<Activity_Type_Name> in <Activity_Type_Name>)

<QueryOnNotReadyAct> ::= NotReady_Act(<Activity_Type_Name>) |
                     NotReady_Act(<Activity_Type_Name> in <Activity_Type_Name>) |
                     NotReady_Act(<Activity_Type_Name> in <Activity_Type_Name>
                                                 dist in <Activity_Type_Name>) |
                     NotReady_All(<Activity_Type_Name>) |
                     NotReady_All(<Activity_Type_Name> in <Activity_Type_Name>)

<QueryOnFinishedAct> ::= Finished_Act(<Activity_Type_Name>) |
                     Finished_Act(<Activity_Type_Name> in <Activity_Type_Name>) |
                     Finished_Act(<Activity_Type_Name> in <Activity_Type_Name>
                                                 dist in <Activity_Type_Name>) |
                     Finished_All(<Activity_Type_Name>) |
                     Finished_All(<Activity_Type_Name> in <Activity_Type_Name>)
```

Acknowledgements

The first author is supported by an EPSRC bursary and a CASE award from HP Laboratories, Bristol, UK.

References

Alonso, G., F. Casati, H. Kuno, and V. Machiraju. (2004). *Web Services*, ISBN: 3540440089. Springer.

Belhajjame, K., C. Collet, and G. Vargas-Solar. (2001). "A Flexible Workflow Model for Process-Oriented Applications", in M. Tamer Ozsu, H.-J. Schek, K. Tanaka, Y. Zhang, and Y. Kambayashi, (eds.), *Proceedings of the 2nd International Conference on Web Information Systems Engineering (WISE_01), Organized by WISE Society and Kyoto University, Kyoto, Japan, 3–6 December 2001, Volume 1 (Main Program)*. IEEE Computer Society.

Bruns, G. (1997). *Distributed Systems Analysis with CCS*, ISBN: 0-13-398389-7. Prentice-Hall.

Cleaveland, R. and M. Hennessy. (1990). "Priorities in Process Algebras"*Information and Computation* 87(1/2), 58–77.

Cleaveland, R., V. Natarajan, S. Sims, and G. Luttgen. (1996). "Modeling and Verifying Distributed Systems Using Priorities. A Case Study"*Software – Concepts and Tools* 17(2), 50–62.

Curbera, F., M. Duftler, R. Khalaf, W. Nagy, N. Mukhi, and S. Weerawarana. (2002). "Unravelling the Web Services Web: An introduction to SOAPWSDL and UDDI", *IEEE Internet Computing* 6(2), 86–93.

Dong, Y. and Z. Shensheng. (2003). "Approach for Workflow Modeling Using π-Calculus"*Journal of Zhejiang University Science* 4(6), 643–650.

Dong, Y. and Z. Shensheng. (2004). Modeling Workflow Patterns with π-calculus. Technical report, Shanghai Jiao Tong University.

Duan, Z., A. Bernstein, P. Lewis, and S. Lu. (2004a). "A Model for Abstract Process Specification, Verification and Composition", in *Proceedings of the Second International Conference on Service Oriented Computing (ICSOC_04)*. New York City, NY, USA, November 2004, pp. 232–241.

Duan, Z., A. Bernstein, P. Lewis, and S. Lu. (2004b). "Semantics Based Verification and Synthesis of BPEL4WS Abstract Processes", in *Proceedings of the IEEE Conference on Web Services (ICWS_04)*, pp. 734–737.

Farahbod, R., U. Glasser, and M. Vajihollahi. (2005). Abstract Operational Semantics of the Business Process Execution Language for Web Services, SFU-CMPT-TR-2005-04. Technical report, School of Computing Science, Simon Fraser University.

Farrell, A. D. H. (2006). Formalising Workflow: A CCS-inspired Characterisation of the YAWL Workflow Patterns. Technical report, HP Labs Technical Report (to appear).

Ferrara, A. (2004). "Web Services: A Process Algebra Approach", in M. Aiello, M. Aoyama, F. Curbera, and M. P. Papazoglou, (eds.), *ICSOC, ACM*, pp. 242–251.

Fisteus, J. A., L. S. Fernandez, and C. D. Kloos. (2004). "Formal Verification of BPEL4WS Business Collaborations", in K. Bauknecht, M. Bichler, and B. Proll, (eds.), *EC-Web 2004, Volume 3182 of Lecture Notes in Computer Science*. Springer, pp. 76–85.

Foster, H., S. Uchitel, J. Magee, and J. Kramer. (2003). "Model-Based Verification of Web Service Composition", in *Proceedings of the 18th IEEE International Conference on Automated Software Engineering Conference (ASE 2003)*.

Fu, X., T. Bultan, and J. Su. (2004). "Analysis of Interacting BPEL Web Services", in S. I. Feldman, M. Uretsky, M. Najork, and C. E. Wills, (eds.), *WWW, ACM*, pp. 621–630.

Georgakopoulos, D., M. Hornick, and A. Sheth. (1995). "An Overview of Workflow Management: From Process Modelling to Workflow Automation Infrastructure"*Distributed and Parallel Databases* 3(2), 119–153.

Gruninger, M.. (2003). "The Process Specification Language (PSL): Theory and Applications"*AI Magazine* 24(3), 63–74.

Hidders, J., M. Dumas, W. M. P. van der Aalst, A. H. M. ter Hofstede, and J. Verelst. (2005). "When are Two Workflows the Same?", in M. Atkinson, and F. Denhe, (eds.), *Proceedings Computing: The Australasian Theory Symposium*. Newcastle, NSW, Australia, pp. 3–11.

Jablonski, S. and C. Bussler. (1996). *Workflow Management – Modeling Concepts, Architecture and Implementation*, ISBN: 1850322228. International Thomson Computer Press.

Kazhamiakin, R. and M. Pistore. (2005). "A Parametric Communication Model for the Verification of bpel4ws Compositions", in M. Bravetti, L. Kloul, and G. Zavattaro, (eds.), *EPEW/WS-FM, Volume 3670 of Lecture Notes in Computer Science*. Springer, pp. 318–332.

Keller, A., J. L. Hellerstein, J. L. Wolf, and V. Krishnan. (2003). The CHAMPS System: Change Management with Planning and Scheduling. IBM Research Report, RC22882 (W0308-089), August 25, 2003.

Kiepuszewski, B. (2003). Expressiveness and Suitability of Languages for Control Flow Modelling in Workflows. PhD thesis, Queensland University of Technology, Brisbane, Australia.

Koshkina, M. and F. van Breugel. (2003). Verification of Business Processes for Web Services, CS-2003-11. Technical report, Department of Computer Science, York University, Toronto.

Leymann F. and D. Roller. (1999). *Production Workflow: Concepts and Techniques*. Prentice-Hall.

Lucchi, R. and M. Mazzara. (2006). "A Foundational Mechanism for WS-BPEL Recovery Framework", *Journal of Logic and Algebraic Programming (JLAP)* (to appear).

Marin, M. (2002). "Business Process Technology: From EAI and Workflow to BPM", in L. Fischer (ed.), *The Workflow Handbook 2002*, ISBN:0-9703509-2-9.

Milner, R. (1989). *Communication and Concurrency*, ISBN: 0-13-115007-3. Prentice Hall.

Milner, R (1990). "Operational and Algebraic Semantics of Concurrent Processes", in *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics (B)*, pp. 1201–1242.

Milner, R. (1999). *Communicating and Mobile Systems: The π-Calculus*, ISBN:0-521-64320-1. Cambridge University Press.

Newcomer, E. and G. Lomow. (2005). *Understanding SOA with Web Services*, ISBN: 0-321-18086-0. Addison-Wesley.

OASIS. (2005). Web Services Business Process Execution Language Version 2.0 Working Draft 1st September 2005; at: http://www.oasis-open.org/apps/org/workgroup/wsbpel.

Ouyang, C., W. M. P. van der Aalst, S. Breutel, M. Dumas, A. H. M. ter Hofstede, and E. Verbeek. (2005). Formal Semantics and Analysis of Control Flow in WS-BPEL, BPM Report BPM-05-15 (Revised Version). Technical report, BPMcenter.org, June 2005.

Puhlmann, F. and M. Weske. (2005). "Using the π-calculus for Formalizing Workflow Patterns", in W. M. P. van der Aalst et al., (eds.), *BPM 2005, Volume 3649 of Lecture Notes in Computer Science*. Springer.

Reisig, W. and G. Rozenberg. (1998). *Lectures on Petri Nets I: Basic Models*, ISBN:3-540-65307-4. Springer.

Reiter, R. (2001). *Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems*, ISBN: 0-262-18218-1. The MIT Press.

Sangiorgi, D. and D. Walker. (2001) The π-calculus. *A Theory of Mobile Processes*, ISBN:0-521-78177-9. Cambridge University Press.

Stefansen, C. (2005a). "A SMAll Workflow Language based on CCS, TR-06-05", in *Proceedings of 17th Conference on Advanced Information Systems Engineering, CAiSE05*, (to appear).

Stefansen, C. (2005b). A SMAll Workflow Language based on CCS, TR-06-05. Technical report, Harvard University, Division of Engineering and Applied Sciences, Cambridge, MA.

van der Aalst, W. M. P. (2003). "Don't Go with the Flow: Web Services Composition Exposed", in *Trends and Controversies. Web Services: Been there, Done that? IEEE Intelligent Systems*, Jan–Feb 2003, pp. 72–76.

van der Aalst, W. M. P. (2004). Business Process Management Demystified: A Tutorial on Models, Systems and Standards for Workflow Management, BPM Center Report BPM-04-03. Technical report, BPMcenter.org.

van der Aalst, W. M. P., L. Aldred, M. Dumas, and A. H. M. ter Hofstede. (2004). "Design and Implementation of the YAWL System", in *Proceedings of The 16th International Conference on Advanced Information Systems Engineering (CAiSE 04), Riga, Latvia*. Springer Verlag, June 2004.

van der Aalst, W. M. P. and A. H. M. ter Hofstede. (2002). "Workflow Patterns: On the Expressive Power of (Petri-net-based) Workflow Languages", in K. Jensen (ed.), *Proceedings of the Fourth Workshop on the Practical Use of Coloured Petri Nets and CPN Tools (CPN 2002), Volume 560 of DAIMI, Aarhus, Denmark, August 2002*, pp. 1–20.

van der Aalst, W. M. P. and A. H. M. ter Hofstede. (2005). "YAWL: Yet Another Workflow Language"*Information Systems* 30(4), 245–275.

Verbeek, H. M. W., T. Basten, and W. M. P. van der Aalst. (2001). "Diagnosing Workflow Processes Using Woflan"*The Computer Journal* 44(4), 246–279.

Viroli, M. (2004). "Towards a Formal Foundation to Orchestration Languages", in M. Bravetti and G. Zavattaro (eds.), *Proceedings of 1st International Workshop on Web Services and Formal Methods (WS-FM 2004), Volume105 of ENTCS*. Elsevier.

Weerawarana, S., F. Curbera, F. Leymann, T. Storey, and D. F. Ferguson. (2005). *Web Services Platform Architecture*, ISBN: 0-13-148874-0. Prentice Hall.

Workflow Management Coalition. (1999). Workflow Management Coalition Terminology & Glossary. Document Number: WFMC-TC-1011. Document Status: Issue 3.0. February 1999.

WS-CDL W3C Working Group. Web Services Choreography Description Language Version 1.0 W3C Working Draft 17 December 2004. Available at: http://www.w3.org/TR/ws-cdl-10.

WWW Consortium. (2002). Web Services Architecture Requirements; at: http://www.w3c.org/TR/wsa-reqs. October 2002.