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

Andrew D H Farrell Marek J Sergot
Department of Computing
Imperial College London
London SW7 2AZ. United Kingdom.
{andrew.farrell, m.sergot}@imperial.ac.uk

Claudio Bartolini
Hewlett Packard Labs
1501 Page Mill Road
Palo Alto, CA 94304-1126. USA
claudio.bartolini@hp.com

April 27, 2006

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. We underwrite the meta-model by giving it a formal semantic characterisation using a language that we define called $\mathcal{L}$CCS, an abstract machine language which has a mapping to a prioritised form of standard CCS. We present the Liesbet meta-model and its semantic characterisation in $\mathcal{L}$CCS, explain how we have facilitated the verification of 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.
Keywords: 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) (http://www.wfmc.org) defines a *business process* to be [52] *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* [52].

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 [30], 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 [22, 17] 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 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)* [33], 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* [33]. 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* [53].

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 [4]. *Service Composition* is a principal aspect of the Web Services framework, where *composite (web) services* may be created by inter-connecting

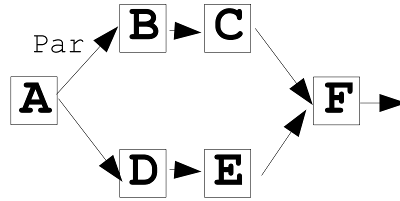


Figure 1: An example workflow model.

deployed web services from potentially many different service providers. WS-BPEL [34] 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 [33, 51]. 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 [34]. An example of a WSC language is WS-CDL (Web Services Choreography Description Language) [18].

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., [22, 43]. We follow the one presented in [43]. 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 [1]. 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 [43], or, in the case of service compositions, such data would be that sent and received in messages that are exchanged between services [1].

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 [38] and CCS/π-calculus [31, 32, 40], 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. $\texttt{new s a.(b.c.s}|\texttt{d.e.s)}|\overline{\texttt{s}}.\overline{\texttt{s}}.\texttt{f}$ 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 [31, 32, 40].

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 [38] or CCS/π-calculus [31, 32, 40].

A computational view workflow model may be directly executable by a workflow engine; that is, the engine may directly understand and execute Petri-nets [38] or CCS/π-calculus [31, 32, 40]. 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 [44], 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 [13] 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 [6]. [43] 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 a CCS-inspired abstract machine language (which represents a computational view ontology) that we define, called $\mathcal{L}$CCS. Importantly, $\mathcal{L}$CCS has a mapping to a prioritised form of standard CCS [31, 32], which we present in [13]. 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 (action sequencing, interleaved concurrent action execution, process templating through replication, and so on). We call an $\mathcal{L}$CCS semantically-characterised Liesbet workflow model, an *$\mathcal{L}$CCS Liesbet model*.

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 [46, 47, 45, 25]. 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 (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 $\mathcal{L}$CCS), in order to fix precisely the meaning of Liesbet constructs. With such a characterisation to hand, we are then 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 $\mathcal{L}$CCS and present an $\mathcal{L}$CCS based formalisation of the Liesbet constructs introduced in Section 2. In Section 5 we discuss verification of Liesbet models. In Section 6 we conclude with a discussion and overview of related work. Appendix A presents some further Liesbet constructs and their $\mathcal{L}$CCS characterisations, and Appendix B a BNF grammar for Liesbet.

2 Liesbet Meta-model and its $\mathcal{L}$CCS Characterisation

We have defined in this work a workflow meta-model (called Liesbet) corresponding to the *information view* of a workflow model, and a CCS-inspired [31, 32] characterisation (called $\mathcal{L}$CCS) 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 the appendix. The constructs that we introduce here are: Activity (`Act`), Synchronisation (`Sync`, `Cond` and `FreeChoice`), Sequence (`Seq` and `SeqCancel`), Parallel (`Par` and `PriPar`), Exclusive Choice (`DefaultChoice` and `Choice`), Multiple Choice (`MultiChoice`) and Cancel Activity (`CancelActivity`).

For each Liesbet construct, we present what we call an 'Easy Syntax'. For the purposes of implementing a model checker for verification of 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 Appendix B.

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. As a $\mathcal{L}$CCS-characterisation is a closed system [13], and as we do not model the environment in our $\mathcal{L}$CCS-characterisation of Liesbet, we make the simplifying assumption that basic activities always complete successfully.

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 (only) 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(S,Seq(B,C))` is the top-level workflow activity type.

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 – `st(Ready)`, `st(Running)`, `st(Cancelled)` or `st(Completed)`. We also consider an activity instance to be *finished*, if it is in a `st(Cancelled)` or `st(Completed)` state.

```
st(Ready) -execute-> st(Running)
st(Ready) -cancel-> st(Cancelled)

st(Running) -complete-> st(Completed)
st(Running) -cancel-> st(Cancelled)
```

- An activity instance begins life in the `st(Ready)` state. At some point, the parent of the activity instance will initiate execution of the instance. The instance will be moved into the `st(Running)` state, by virtue of the `execute` action.
- When the work of the instance is done, it is moved to the `st(Completed)` state, by means of the `complete` action.
- From the `st(Ready)` and `st(Running)` states, the instance may be moved into the `st(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, in a single, atomic reduction.

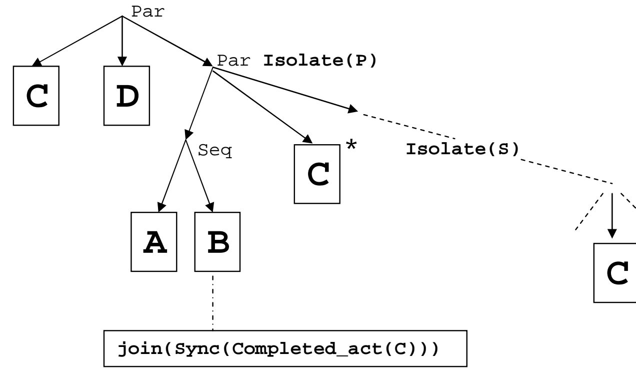
  Cancellation of an activity may happen because of the execution of a `CancelActivity` instance (Section 2.8), because of a failed join condition (Section 2.5), or because of dead-path elimination. Dead-path elimination [28] is performed in workflow model enactment when it is identified that an activity 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 `st(Cancelled)` state.

Isolated Scopes

As explained in Sections 2.2 and 4.3.2 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.

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(...)
```



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 $*$.

Figure 2: Isolated Scopes in Operation

This has the effect of creating a *visibility horizon* on the workflow state, $\mathcal{S}$, for activity instances that exist within an instance of the isolated scopes A and B.

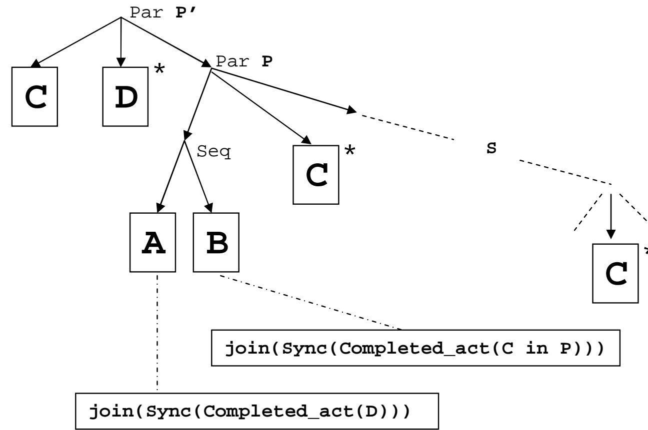
When an instance A exists within the scope of another activity instance which is isolated, the instance A can only query the state of activity instances that are descendants of the isolated scope instance that is *the most immediate ancestor* of A, and this isolated scope instance itself. Moreover, if 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 A. 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).

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 [13]. 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.

2.2 `Sync`, `Cond` and `FreeChoice` 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.



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 all instances of type D marked $*$. (For this particular workflow model there is only one such instance of D.) 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.

Figure 3: Reference Types in Operation

Easy Syntax

```
Sync(StopQuery, GoQuery)
Sync(GoQuery)
Cond(GoQuery)
FreeChoice
```

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 any 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 `st(Ready)`, `st(Running)`, or `st(Cancelled)` states, as well as *finished* instances (those in `st(Completed)` or `st(Cancelled)` states). To use such queries, the keyword `Completed` is replaced appropriately in the previous query descriptions.

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:

- Evaluate a query against the current variable state in the data perspective (see Section 1) which we do not model. We may write `EvalExpr(`*EXPR*`)`, where *EXPR* is an arbitrary boolean expression over current variable state.

- 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 `st(Cancelled)`. If `GoQuery` is satisfied, and `StopQuery` is not satisfied beforehand, then the synchronisation activity instance goes to `st(Completed)`. While neither query is satisfied, the instance remains in the `st(Running)` state.

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

An instance of the activity type `Cond(GoQuery)` moves from the `st(Running)` state to `st(Completed)` if `GoQuery` is satisfied and to `st(Cancelled)` if `¬GoQuery` is satisfied. `Cond(GoQuery)` is thus equivalent in behaviour to `Sync(¬GoQuery,GoQuery)`.

`FreeChoice` is an activity type that non-deterministically goes from `st(Running)` to `st(Completed)` or `st(Cancelled)`.

Finally, note that a `Sync` type can be used to effect the YAWL workflow pattern *Milestone* [46, 25], 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(C), Finished_act(B) | ¬Finished_act(C))`. Here, if `C` has already finished, the `Sync` cancels. Or, if `B` has finished, but `C` has not finished, the `Sync` completes.

2.3 `Seq` and `SeqCancel` – Sequence

The Liesbet constructs `Seq/SeqCancel` are a direct facilitation of the YAWL workflow pattern *Sequence* [46, 25], which is where *an activity in a workflow model is enabled after the completion of another activity.*

We also support an unordered sequence construct, which in YAWL [46] is called the *interleaved parallel routing* construct, but do not present details here. The interested reader should consult [13].

Easy Syntax

```
Seq(Act1, ..., Actn)
SeqCancel(Act1, ..., Actn)
```

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 (`st(Completed)` or `st(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 `st(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* [46, 25], 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(Act1, ..., Actn)
```

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 (`st(Completed)` or `st(Cancelled)`), the parallel instance goes to `st(Completed)`. Note therefore that cancelling child instances does not cancel the `Par` activity.

We also support a *Priority Parallel* (`PriPar`) construct. Its formalisation in $\mathcal{L}$CCS is not presented in detail in this article but is presented in full in [13]. `PriPar` allows arbitrarily complex structured activities to be specified as running in parallel, but the execution of individual basic activities within these structured activities occurs according to a priority ordering. For example, suppose we define the customised activity type `PriPar(Seq(A,B), Seq(C,D))`. Here, execution of basic activities `C` or `D` may only be *initiated* if the execution of `A` or `B` is not possible *and* if instances of these activities are not already running. In sum, (partial) execution of the second `Seq(C,D)` may occur if and only if `Seq(A,B)` is blocked (e.g., because of a blocked join condition, Section 2.5).

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 `st(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 (`Sync` or `Cond`). 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 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 $\mathcal{L}$CCS.

- `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* [46, 25], 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(Guard1, ContAct1; ... ; Guardn, ContActn; ContAct_d)
Choice(Guard1, ContAct1; ... ; Guardn, ContActn)
```

Informal Operational Semantics

Each $\texttt{Guard}_i$ is a *guard* activity type and $\texttt{ContAct}_i$ 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 the basic act type that trivially completes (see Appendix A), can be used to effect a non-deterministic choice .

The first guard instance that goes to `st(Completed)` initiates its corresponding continuation instance. All other continuation instances go to `st(Cancelled)`. In the case of `DefaultChoice`, if all of the $\texttt{Guard}_i$ activities go to `st(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 `st(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 *Multi-Choice* [46, 25], 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(Guard1, ContAct1; ... ; Guardn, ContActn)
```

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.

We also support a variant, `MultiChoiceMin`, which imposes a minimum number of branches that must be executed. If this does not occur, the instance is cancelled. Details of this variant are not presented here. The interested reader should consult [13].

2.8 `CancelActivity` – Cancel Activity

The Liesbet construct `CancelActivity` is a direct facilitation of the YAWL workflow pattern *Cancel Activity* [46, 25], 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., `st(Running)`) and all possible future running (i.e., `st(Ready)`) instances of the named activity type, `CancelAct`, within its visibility horizon. Optionally, `CancelActivity` may specify a reference type (see Section 2.1) to constrain the visibility horizon.

2.9 Support for YAWL Workflow Patterns

In Table 1, we present an overview of how the Liesbet meta-model supports the YAWL workflow patterns [46]. Note the difference in our support for Discriminator (Pattern #9) compared with that presented in [46].

Workflow Pattern	**Satisfied How?**
1 Sequence	`Seq`
2 Parallel Split	`Par`
3 Synchronisation, A.k.a. AND-JOIN	Yes$^{(i)}$
4 Exclusive Choice	`DefaultChoice, Choice`
5 Simple Merge, A.k.a. XOR-JOIN	Yes$^{(i)}$
6 Multiple Choice	`MultiChoice`
7 Synchronising Merge, A.k.a. OR-JOIN	Yes$^{(i)}$
8 Multimerge	`Multimerge` (see Appendix A)
9 Discriminator	`Disc`$^{(ii)}$ (see Appendix A)
10 Arbitrary Cycles$^{(iii)}$	Yes$^{(iv)}$
11 Implicit Termination$^{(v)}$	Yes
12 Multiple Instances (MIs) Without Synchronisation	`MultiNoSync` (see Appendix A)
13 MIs Without A Priori Design Time Knowledge	`MultiLimit` (see Appendix A)
14 MIs Instances With A Priori Run Time Knowledge	`MultiKnown` (see Appendix A)
15 MIs Without A Priori Run Time Knowledge	`Multi` (see Appendix A)
16 Deferred Choice	`DeferredChoice` (see Appendix A)
17 Interleaved Parallel Routing (Unordered Sequence)	`UnorderedSeq`$^{(vi)}$
18 Milestone	`Sync`
19 Cancel Activity	`CancelActivity`
20 Cancel Case	`Exit` (see Appendix A)

Table 1: Satisfaction of YAWL Workflow Patterns [46, 25]

Comments

(i) Supported in multiple ways:
- Implicit Synchronisation when activity completes
- Arbitrary Synchroniser can run in parallel

(ii) Our version of the Discriminator pattern (see Appendix A) is more general than that presented in [46]. In [46], Discriminator is supported by means of a threshold on the number of completed instances of a multiple instance activity. While we also support this sort of Discriminator, our main support for the Discriminator pattern is through the `Disc` activity type, which can be used to synchronise on arbitrary activity instance executions, not just those of a continuation activity type belonging to a single multiple-instance activity instance.

(iii) An arbitrary cycle can be thought of as being a repeating piece of workflow logic that can not be mapped to a multiple-instance activity. This is an equivalent definition to that given in [25].

(iv) We naturally support arbitrary cycles, as any use of activity types within a Liesbet model definition may create a cycle; but we assume that, where possible, the desired behaviour will be represented using multiple-instance activity types. We do not support the verification of models (for soundness and other properties) containing arbitrary cycles, as discussed in [13]. It is clear that supporting the verification of such models would increase the complexity of verification quite considerably, and is not something we currently need to support for the purposes of our work.

(v) From [25]: 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).

(vi) Liesbet, and its $\mathcal{L}$CCS characterisation, operates on the basis of implicit termination

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 2.9 we summarise how Liesbet supports these. In [13], we present many further examples.

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 `st(Completed)`) nor both fail (go to `st(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 Formalisation of Liesbet via $\mathcal{L}$CCS Abstract Machine Language

We now present the CCS-inspired language (or ontology in the parlance of Section 1), $\mathcal{L}$CCS, that we shall use to give a semantic characterisation to Liesbet. It should be emphasised that $\mathcal{L}$CCS represents a sugared syntax – an abstract machine language – that has a mapping to standard CCS [31, 32] augmented with a notion of process priority. It is essentially standard CCS with a number of additional process artefacts.

A key aspect of $\mathcal{L}$CCS is that we assume that, at any stage, a workflow model $\mathcal{W}$ is characterised by a pair $\prec\mathcal{S}, \mathcal{P}\succ$, where $\mathcal{S}$ is a *state chalkboard* and $\mathcal{P}$ is a CCS-like workflow process. $\mathcal{L}$CCS is thus a hybrid language where the use of a state chalkboard facilitates arbitrarily complex queries on workflow state.

We present a brief overview of (standard) CCS and the additional process artefacts that $\mathcal{L}$CCS admits, and then proceed to describe how we have used $\mathcal{L}$CCS to give a formal characterisation of the Liesbet constructs introduced in Section 2.

4.1 Overview of CCS

We begin with a brief summary of the main features of CCS. For readers unfamiliar with CCS, [40, 31, 32] are excellent starting points.

Firstly, we assume the availability of an infinite set of action names $\mathcal{N}$, ranged over by $a, b, \ldots$, and a corresponding set of co-names $\bar{\mathcal{N}} = \{\bar{a} | a \in \mathcal{N}\}$, where $\mathcal{N}$ and $\bar{\mathcal{N}}$ are disjoint. An action name and its co-name are complementary, and together constitute the two halves of a CCS interaction. Interactions are the means by which we derive reductions for CCS processes, where reductions are the means by which CCS processes evolve[1].

A CCS process term, P, is defined by the following grammar (adapted from [32]).

$$\mathcal{P} \texttt{ ::= } \mathcal{M} \mid \mathcal{P}|\mathcal{P}' \mid \texttt{new}\ \breve{z}\ \mathcal{P} \mid !\mathcal{P}$$

$$\mathcal{M} \texttt{ ::= } \texttt{true} \mid x(\breve{z}).\mathcal{P} \mid \bar{x}<\breve{y}>.\mathcal{P} \mid \mathcal{M} \texttt{ + } \mathcal{M}'$$

- `|` is *parallel*, and denotes the running of two processes $\mathcal{P}$ and $\mathcal{P}'$ concurrently (in an interleaved fashion).
- `!` is *replication*, and allows multiple copies of the process $\mathcal{P}$ to be introduced in parallel with $!\mathcal{P}$. It may be thought of as a *process factory*.
- `new` is *restriction*, and restricts the scope of the action names, $\breve{z}$, that it specifies to be $\mathcal{P}$.
- `+` is *summation*, and specifies a choice of behaviours: the process $\mathcal{M} + \mathcal{M}'$ may continue (exclusively) as $\mathcal{M}$ or $\mathcal{M}'$.
- **true** is the inactive process. (In standard CCS, this is written `0`.)
- $a(\breve{z}).\mathcal{P}$ is an *input prefixed process*, where $\mathcal{P}$ may only occur after the prefixing *input action* $a(\breve{z})$ has been consumed. The input action is associated with a co-named output action in forming an interaction. In an interaction, a set of values are passed on $\breve{z}$, which are substituted for occurrences of these parameters in $\mathcal{P}$. Note that only values may be passed, and not names of channels (unlike π-calculus [32, 40]).
- $\bar{a}<\breve{y}>.\mathcal{P}$ is an *output prefixed process*, where $\mathcal{P}$ may only occur after the prefixing *output action* $\bar{a}<\breve{y}>$ has been consumed. The output action is associated with a corresponding input action in forming an interaction. In an interaction, a set of values are passed on $\breve{y}$.

[1] CCS also has a transition-based semantics, which we do not employ in this work.

$$\texttt{INTER} \quad \frac{\mathcal{P} = \bar{x}<\breve{y}>.\mathcal{P}' + \mathcal{M} \qquad \mathcal{Q} = x(\breve{z}).\mathcal{Q}' + \mathcal{N}}{\prec\mathcal{S}, \mathcal{P}|\mathcal{Q}\succ \rightarrow \prec\mathcal{S}, \mathcal{P}'|\mathcal{Q}'\{^{\breve{y}}/_{\breve{z}}\}\succ}$$

$$\texttt{PAR} \quad \frac{\prec\mathcal{S}, \mathcal{P}\succ \rightarrow \prec\mathcal{S}, \mathcal{P}'\succ}{\prec\mathcal{S}, \mathcal{P}|\mathcal{Q}\succ \rightarrow \prec\mathcal{S}, \mathcal{P}'|\mathcal{Q}\succ}$$

$$\texttt{RES} \quad \frac{\prec\mathcal{S}, \mathcal{P}\succ \rightarrow \prec\mathcal{S}, \mathcal{P}'\succ}{\prec\mathcal{S}, \texttt{new}\ \breve{z}\ \mathcal{P}\succ \rightarrow \prec\mathcal{S}, \texttt{new}\ \breve{z}\ \mathcal{P}'\succ}$$

$$\texttt{STRUCT} \quad \frac{\prec\mathcal{S}, \mathcal{P}\succ \rightarrow \prec\mathcal{S}, \mathcal{P}'\succ \qquad \mathcal{P} \equiv \mathcal{Q} \qquad \mathcal{P}' \equiv \mathcal{Q}'}{\prec\mathcal{S}, \mathcal{Q}\succ \rightarrow \prec\mathcal{S}, \mathcal{Q}'\succ}$$

Figure 4: CCS Reduction Rules – slightly simplified from [32]

We make use of the reduction-based semantics for CCS [32]. In this semantics, reduction is the metaphor by which processes evolve, and at its core is the reduction relation, $\rightarrow$. A CCS process $\mathcal{P}$ may always be represented as a parallel of summations (omitting scope restrictions), i.e., in the form:

$$\mathcal{P}_{0,0} \ + \ \ldots \ + \ \mathcal{P}_{0,j_0} \mid \ldots \mid \mathcal{P}_{i,0} \ + \ \ldots \ + \ \mathcal{P}_{i,j_i}$$

a It is on such a representation that we derive reductions.

In order to derive a reduction for $\mathcal{P}$, we derive a reduction for two of its constituent (parallelised) process terms based on the interaction of an input and output action. Then we proceed to build a derivation for a reduction of $\mathcal{P}$ as a whole based on this interaction.

Formally, the CCS reduction relation, $\rightarrow$, is defined as the smallest relation closed under the reduction rules presented in Figure 4 (adapted from [32]).

`INTER` is the means by which we derive a reduction based on the interaction of an input and output action. The process $\mathcal{P}$ (resp. $\mathcal{Q}$) is a sum whose left operand is a process consisting of a prefixing action followed by a process $\mathcal{P}'$ (resp. $\mathcal{Q}'$), and whose right operand is the process $\mathcal{M}$ (resp. $\mathcal{N}$). The prefixing action in the left operand for $\mathcal{P}$ (resp. $\mathcal{Q}$) is an output action (resp. input action), which passes data $\breve{y}$ on x (resp. which binds the value names $\breve{z}$ in the rest of $\mathcal{Q}'$ to the data values $\breve{y}$, passed on x). In the interaction, the two actions – the input and the output – are consumed, and the other operands of the sums are rendered void. The process $\mathcal{P}|\mathcal{Q}$ will evolve to $\mathcal{P}'|\mathcal{Q}'\{^{\breve{y}}/_{\breve{z}}\}$.

From [40], any reduction may be built up by firstly applying rule `INTER`, followed by an application of `PAR` (which allows us to augment the two interacting processes with an arbitrary number of other processes operating in parallel), followed by zero or more applications of `RES` (which allows us to restrict the scope of action names), followed lastly by an application of `STRUCT`.

The rule `STRUCT` allows us to re-write processes in order that we may prime them for the derivation of reductions. This is necessary, as `INTER` requires that two process terms are contiguous for them to interact (and ordered with the outputting process term occurring first). `STRUCT` employs a *structural congruence* relation, which is defined to be closed under the axioms for structural congruence presented in Table 2 and the laws of equational reasoning presented in Table 3 [40].

The first three equations in Table 3 ensure that $\equiv$ is an equivalence relation. CONG ensures that structural congruence is a congruence. From [40], an equivalence relation $\mathcal{R}$ is a congruence if $(\mathcal{P}, \mathcal{Q}) \in \mathcal{R}$ implies $(\mathcal{C}[\mathcal{P}], \mathcal{C}[\mathcal{Q}]) \in \mathcal{R}$ for every *context* $\mathcal{C}$. A context is obtained

[SC-ALPHA]	$\mathcal{P} \equiv \alpha(\mathcal{P})$ where α renames the restricted names in $\mathcal{P}$
[SC-SUM-ASSOC]	$\mathcal{P}_1 + (\mathcal{P}_2 + \mathcal{P}_3) \equiv (\mathcal{P}_1 + \mathcal{P}_2) + \mathcal{P}_3$
[SC-SUM-COMM]	$\mathcal{P}_1 + \mathcal{P}_2 \equiv \mathcal{P}_2 + \mathcal{P}_1$
[SC-SUM-INACT]	$\mathcal{P} + \mathbf{true} \equiv \mathcal{P}$
[SC-COMP-ASSOC]	$\mathcal{P}_1 \vert (\mathcal{P}_2 \vert \mathcal{P}_3) \equiv (\mathcal{P}_1 \vert \mathcal{P}_2) \vert \mathcal{P}_3$
[SC-COMP-COMM]	$\mathcal{P}_1 \vert \mathcal{P}_2 \equiv (\mathcal{P}_2 \vert \mathcal{P}_1)$
[SC-COMP-INACT]	$\mathcal{P} \vert \mathbf{true} \equiv \mathcal{P}$
[SC-RES-INACT]	$\mathtt{new}\ \widetilde{z}\ \mathbf{true} \equiv \mathbf{true}$
[SC-RES-COMP]	$\mathtt{new}\ \widetilde{z}\ (\mathcal{P} \vert \mathcal{P}') \equiv \mathcal{P} \vert\ \mathtt{new}\ \widetilde{z}\ \mathcal{P}'$ $\{\widetilde{z}\} \cap \mathrm{fn}(\mathcal{P}) == \emptyset$ where $\mathrm{fn}(\mathcal{P})$ yields the unrestricted names of $\mathcal{P}$
[SC-REP]	$!\mathcal{P} \equiv \mathcal{P} \vert !\mathcal{P}$

Table 2: Structural Congruence Rules for $\mathcal{L}$CCS

REFL	$\mathcal{P} \equiv \mathcal{P}$
SYMM	$\mathcal{P} \equiv \mathcal{Q}$ implies $\mathcal{Q} \equiv \mathcal{P}$
TRANS	$\mathcal{P} \equiv \mathcal{Q}$ and $\mathcal{Q} \equiv \mathcal{R}$ implies $\mathcal{Q} \equiv \mathcal{R}$
CONG	$\mathcal{P} \equiv \mathcal{Q}$ implies $\mathcal{C}[\mathcal{P}] \equiv \mathcal{C}[\mathcal{Q}]$

Table 3: Rules for Equational Reasoning

from a $\mathcal{L}$CCS process $\mathcal{P}$ when the hole [.] replaces a non-degenerative occurrence of **true** in $\mathcal{P}$. **true** occurs degeneratively if it is the left or right term of a sum $\mathcal{M} + \mathcal{M}'$.

Note that the inactive process **true** is suffixed to every summation operand in a CCS process, although it is often omitted in examples for convenience. The enactment of a process is considered to have halted successfully if the residual process is structurally congruent to the inactive process **true**. A process is considered to be deadlocked if no progression (through reduction) can be made but the process is not structurally congruent to **true**.

The heart of process evolution in CCS is the interaction of an input prefixed process term, residing within a summation (possibly degenerative), with another such output prefixed process term.

We define an abstract machine language that might appear to add more to the reduction-based semantics of processes than just this simple interaction-based metaphor. However, all of the additional process artefacts that we admit may be mapped to standard CCS (with priority – more information is given in [13]). This is an important point, as staying (largely) within the realms of standard CCS enables us to make use of the wealth of mathematical results that exists for it, such as notions of bisimulation equivalence.

4.2 $\mathcal{L}$CCS Abstract Machine Language

The $\mathcal{L}$CCS abstract machine language has a semantics grounded in standard CCS augmented with a notion of process priority (see [13] for more details). It admits the following additional language artefacts as *syntactic sugar* to standard CCS, that have proved helpful in characterising the YAWL workflow patterns.

- A sequence operator *;* which allows us to place arbitrary processes in sequence. The only sequences that are admitted in standard CCS are those consisting of an input or output action followed by an arbitrary process.
- The process **false**. This represents the deadlocked process. It is different from **true**, in that occurrences of **true** may be removed from a process by applications of structural congruence, whereas **false** represents a process that cannot evolve (i.e. reduce) in any way, nor be removed. **false** has infinitely many mappings to an equivalent process in

standard CCS. An example is `new a a`. This process cannot be reduced, as the scope of `a` has been restricted and there is no process with a complementary (output) action operating in parallel within the scope of the restriction.

- $(\mathcal{P}$ `||t>` $\mathcal{Q})$ – a transaction scope. Informally, $\mathcal{P}$ may evolve as long as no reduction is derivable for $\mathcal{Q}$. As soon as a reduction is derivable for $\mathcal{Q}$ to $\mathcal{Q}'$, the transaction scope evolves to $\mathcal{Q}'$.
- $(\mathcal{P}_0$ `||>` $\ldots$ `||>` $\mathcal{P}_i$ `||>` $\ldots$ `||>` $\mathcal{P}_n)$ – a parallel priority scope. Informally, the evolution of a process $\mathcal{P}_j$ within a parallel priority scope may occur while no reduction can be derived for any higher-priority processes $\mathcal{P}_i$, $i<j$.

We also conceive $\mathcal{L}$CCS as a hybrid language $\prec\mathcal{S}, \mathcal{P}\succ$, where $\mathcal{P}$ is a CCS-like workflow process and $\mathcal{S}$ is a state chalkboard – essentially, a collection of *fluent-value* pairs. A fluent is a property of the state chalkboard which may vary in value over the course of enacting a workflow model. A class of $\mathcal{L}$CCS actions called *-actions are used to update and query the state of $\mathcal{S}$, from within $\mathcal{P}$. The workflow process $\mathcal{P}$ should be seen primarily as *orchestrating* changes to the workflow state $\mathcal{S}$. There are number of advantages of using a state chalkboard as a metaphor in characterising the YAWL workflow patterns (see Section 6). It should be emphasised, however, that in the mapping of $\mathcal{L}$CCS to standard CCS, *-actions are mapped to regular CCS actions which interact with a realisation of $\mathcal{S}$ as a standard CCS process. In characterising the workflow patterns, it is useful to use a *-action metaphor with associated additional reduction rules (presented in Section 4.3) as the characterisation in standard CCS is far from trivial.

As for the workflow state chalkboard $\mathcal{S}$, it suffices to consider it to be a binary relation

$$\{_{j\in 1,\ldots,n}\ f_j = v_j\}$$

recording fluent-value pairs. We write $f_j = v_j$ to indicate that the fluent f_j has value v_j. Note that f_j will be a possibly parameterised string literal. The type of values, v_j, that may be stored in $\mathcal{S}$ are

- Integer and boolean data – integer values stored in $\mathcal{S}$ are encapsulated within `i(...)`, e.g. `i(-1)`, `i(0)`, `i(1)`, and so on; boolean values are `b(true)` and `b(false)`.
- State data – values that are names of states, specifically, `st(Ready)`, `st(Running)`, `st(Cancelled)`, and `st(Completed)`.
- List data – values within $\mathcal{S}$ may be *ordered* lists of integer, boolean, state, or name data (but not list data).
- $\mathcal{L}$CCS names – any value that is not a list, `st(...)`, `i(...)`, or `b(...)`.

List delimiters are [and] and the empty list is denoted [], $\in$ tests list membership, $i : L$ extracts the ith member from L – where a list is indexed from 0 – and $\iota : L$ extracts the last member from L. Further, + is a concatenation operator for lists such that $[a_0, \ldots, a_m]$ + $[b_0, \ldots, b_n]$ yields $[a_0, \ldots, a_m, b_0, \ldots, b_n]$. $|L|$ yields the number of elements in L. We may also use quantifiers $\forall$ and $\exists$ in writing tests (for rule premises) on lists; for example $\forall un \in L.State(un) == st(Completed)$.

Typically, we will write process terms of the form: $a(0 : L)? \ldots ?a(\iota : L)$ (where ? is some $\mathcal{L}$CCS process operator, such as + or |) to mean an ordered ?-composition of all members of L in the order from 0 to ι. Similarly, we may write $a(\iota : L)? \ldots ?a(0 : L)$ to mean a ?-composition ordered from ι to 0. So $a(0 : L); \ldots ; a(\iota : L)$, where L is $[\alpha, \beta, \gamma]$, would expand to $a(\alpha); a(\beta); a(\gamma)$.

Updates on $\mathcal{S}$ are specified thus within reduction rules:

$$\mathcal{S} \oplus \{f = v\} \triangleq \begin{cases} \mathcal{S} \cup \{f = v\} & \text{if } f \neq f_j \text{ for } j \in 1 \ldots n, \\ \{_{j\in 1,\ldots,m-1,m+1,\ldots,n} f_j = v_j\} \cup \{f = v\} & \text{if } f = f_m \end{cases}$$

Extracting values from state $\mathcal{S}$ is specified thus: $v := [\mathcal{S} \mapsto f]$. Here, v is assigned to the current value of f in $\mathcal{S}$. If f does not exist within $\mathcal{S}$ then v is assigned to [].

Fluent	Written by ...	Read by ...	Description
Act_type(un)	*add_activity	_GetCandidates/4, see [13]	The name of the customised activity type of activity instance un
State(un)	*add_activity, *execute *complete, *cancel	*execute, *complete *cancel, *running	The state of activity instance un; one of st(Ready), st(Running) st(Completed), or st(Cancelled)
Children(un)	*add_activity	*add_activity, *complete *cancel	The (list of) children of activity instance un
Parent(un)	*add_activity	_GetCandidates/4	The parent instance of activity instance un
Root	*add_activity	*exit	The name of the root activity instance of the workflow model
Scope(un)	*add_scope_activity	_GetCandidates/4	Records whether an activity is an isolated scope, Section 2.1
Integer(un, I)	*int	*iszero, *isnotzero, ...	Records Integer data
Boolean(un, B)	*bool	*istrue, *isfalse	Records Boolean data

Table 4: Fluents stored in $\mathcal{S}$ in the course of $\mathcal{L}$CCS workflow model enactment, and actions that read and write them.

We have also experimented with representing $\mathcal{S}$ as a *Description Logic* knowledge base, so that we may structure knowledge hierarchically (e.g., as action taxonomies) and query such hierarchical state in the processing of the workflow model. For example, when we ask whether an activity of a particular type has completed, the query may be answered by all sub types of the action type. The version using Description Logic will be described elsewhere.

4.3 $\mathcal{L}$CCS Characterisation of Liesbet Workflow Models

4.3.1 Specifying $\mathcal{L}$CCS Workflow Models

We now show how individual Liesbet models may be mapped to a *$\mathcal{L}$CCS Workflow Model Specification*, which is, essentially, a $\prec \mathcal{S}, \mathcal{P} \succ$ pair in which $\mathcal{P}$ – the workflow process – makes use of a library of $\mathcal{L}$CCS process specifications for generic activity types. Table 4 summarizes the fluents that may be stored in the workflow state $\mathcal{S}$ as a $\mathcal{L}$CCS workflow model specification is enacted.

In the following section, we present informal reduction rules for *-actions which should be fairly intuitive and easy to understand. In the mapping to standard CCS, *-actions are converted to process patterns which use regular CCS actions.

Consider a very simple workflow model consisting of a parallel activity type which contains two sequence types, each of which themselves contain two basic activity types. The last activity type of one of the sequences, and the first activity type of the other, are the same type. In Liesbet Easy Syntax, the example model would be: `Par((Seq(A,B),Seq(B,C))`. The mapping into $\mathcal{L}$CCS for this workflow model is as follows. (In [13, 11], we give an enactment narrative for this example workflow specification.)

```
<[],
new un
  (
   new _p, _s, _z
    (
     new s1,s2,a,b,c
       RepAddStructAct(s1,[a,b],_s) |
       RepAddStructAct(s2,[b,c],_s) |
```

```
      RepAddBasicAct(a,_z) |
      RepAddBasicAct(b,_z) |
      RepAddBasicAct(c,_z)
   ||>
      new p AddRootAct(p,un,[s1,s2],_p) |

      RepStructAct(Par)(_p) |
      RepStructAct(Seq)(_s)
   ||>
      RepBasicAct(_z)
   )
 <t|| *finished(un)
 )
>
```

Note that all $\mathcal{L}$CCS workflow process specifications, $\mathcal{P}$, must take this form, namely, `((RepAddStructs | RepAddBasicActs ||> AddRootAct | RepStructActs ||> RepBasicAct) <t|| *finished(un))`. We make use of a parallel priority scope, and, a little later, we will explain the rationale behind structuring a top-level workflow specification in this way. Firstly, however, we will explain the process abbreviations `AddRootAct`, `RepAddStructAct`, `RepAddBasicAct`, `RepStructAct` and `RepBasicAct`. These are defined as follows.

```
AddRootAct(at, un, [at1, ..., atn], gt)
=
   new un1, ..., unn
   at1-<un,un1>; ...; atn-<un,unn>;
   *add_activity(at,[],un);
   *execute(un);
   gt-<un,[un1,...,unn]>

RepAddStructAct(at, [at1, ..., atn], gt)
=
   !at(pt,un);
   new un1, ..., unn
   at1-<un,un1>; ...; atn-<un,unn>;
   *add_activity(at,pt,un);
   *running(un);
   gt-<un,[un1,...,unn]>

RepAddBasicAct(at, gt)
=
   !at(pt,un);
   *add_activity(at,pt,un);
   *running(un);
   gt-<un>

RepStructAct(Activity)(gt)
=
   !gt(un,kids);
   (
      *running(un);Activity(un, kids);*complete(un)
   <t||
      *cancelled(un)
   )

RepBasicAct(gt)
```

```
=
  !gt(un); (*complete(un) <t|| *cancelled(un))
```

Elaborating, with regard to the $\mathcal{L}$CCS workflow model specification, the initial workflow state $\mathcal{S}$ is empty. The lines containing `RepAddStructAct`, `RepAddBasicAct` and `AddRootAct`, along with the line allocating names `s1,s2,a,b` and `c`, are all specific to a particular Liesbet model. The remainder of the example $\mathcal{L}$CCS workflow model specification is not model-specific. The lines containing `RepStructAct(Activity)(gt)` are specific to *generic* activity types; that is, there is one such process term for each generic activity type, here, `Seq` and `Par`, used in the workflow model. Finally, the line containing `RepBasicAct(gt)` is ever-present (unless we have a model consisting of just one basic activity, as remarked later) and there is only ever one of them – its purpose being to realise generic logic pertaining to basic activity types.

Referring to the specification of `RepStructAct` presented above, the `Activity` parameter is (the name of) one of the (library of $\mathcal{L}$CCS) process specifications for generic activity types, defined in the following sections, such as `Seq` (Section 4.3.3) or `Par` (Section 4.3.4). The parameter `gt` refers to a *guard channel* along which we can signal the exposure of process logic for effecting instances of a sequence, a parallel, etc. `RepAddStructAct` uses a replication (a factory for $\mathcal{L}$CCS process terms) meaning that an unlimited number of instances of the generic activity type may be created. The data received on `gt` are the unique name, `un`, of the sequence, parallel, etc., instance being created, and a list of child instances, `kids`.

Once we have signalled along `gt`, the exposed process logic proceeds as follows. Firstly, we wait for the parent activity instance to move this activity instance to the `st(Running)` state, as reflected by `*running(un)`. The reduction rule for `*running` simply checks that the instance is in the `st(Running)` state.

QUERY-RUNNING

$$\begin{array}{l} \mathcal{W} = \prec\mathcal{S}, \mu.\mathcal{P} + \mathcal{M}\succ \\ \mu = \texttt{*running(un)} \\ [\mathcal{S} \mapsto \texttt{State(un)}] == \texttt{st(Running)} \\ \hline \mathcal{W} \rightarrow \prec\mathcal{S}, \mathcal{P}\succ \end{array}$$

There are similar query rules for `st(Ready), st(Cancelled)` and `st(Completed)`, obtained by replacing `st(Running)` in the above reduction rule by one of these other states. There is also a rule for `*finished(un)` which simply checks whether the given activity instance is in either a `st(Cancelled)` or `st(Completed)` state.

Following that, we proceed with the passed `Activity` process specification, passing the unique identifiers for the instance, `un`, and the instance's children as the list `kids`. After the `Activity` logic has been consumed, we perform some housekeeping in moving the (structured) activity instance into a `st(Completed)` state, by means of the `*complete(un)` *-action whose reduction rule is presented next. It has the effect of moving an activity instance in a `st(Running)` state to `st(Completed)`. It will do so if all of its immediate children have themselves finished (have completed or have been cancelled).

ACTIVITY-COMPLETE

$$\begin{array}{l} \mathcal{W} = \prec\mathcal{S}, \mu.\mathcal{P} + \mathcal{M}\succ \\ \mu = \texttt{*complete(un)} \\ [\mathcal{S} \mapsto \texttt{State(un)}] == \texttt{st(Running)} \\ \mathit{Children} := [\mathcal{S} \mapsto \texttt{Children(un)}] \\ \forall un_i \in \mathit{Children}.[\mathcal{S} \mapsto \texttt{State(un}_i\texttt{)}] == \texttt{st(Completed)} \vee \ldots == \texttt{st(Cancelled)} \\ \mathcal{S}' := \mathcal{S} \oplus \{\texttt{State(un)} = \texttt{st(Completed)}\} \\ \hline \mathcal{W} \rightarrow \prec\mathcal{S}', \mathcal{P}\succ \end{array}$$

Finally, we use a transaction scope, `( ... <t|| *cancelled(un))`. This has the effect of immediately consuming all of the process logic for the activity instance, represented by

ellipses (...), if has been determined that the instance has been cancelled.

The purpose of `RepAddStructAct` is to initiate the exposure of process logic pertaining to an instance of the customised activity type, `at`, and to create instances of `at`'s child types `[at1, ..., atn]`. It uses a replication, meaning that it can repeat these tasks, which are initiated by an interaction on the guard channel `at`, an unlimited number of times. Passed along `at` is the unique name, `un`, that has been assigned for the instance to be created, and the unique name, `pt`, of its parent instance. Whenever an interaction on `at` occurs, we proceed to initiate the creation of the child (customised) activity types of `un`, signalling on the appropriate `RepAddStructAct` and `RepAddBasicAct` guard channels (here, `s1,s2,a,b,c`) to do that. Along these channels, we pass the unique name of the parent instance, `un`, and a freshly allocated unique name, e.g., `un1`, for the new child instance. We then add the activity instance, `un`, to the workflow state, $\mathcal{S}$, by using `*add_activity`, passing the customised type name, `at`, along with the unique name of the instance and that of its parent. The reduction rule for `*add_activity` is as follows.

ACTIVITY-ADD

$$\frac{\begin{array}{l}\mathcal{W} = \prec\mathcal{S}, \mu.\mathcal{P} + \mathcal{M}\succ \\ \mu = \texttt{*add_activity(atn,pt,un)} \\ \mathcal{S}'' := \mathcal{S} \oplus \{\texttt{Act_type(un) = atn}\} \oplus \{\texttt{State(un) = st(Ready)}\} \\ \texttt{if (pt} \neq [\,]\texttt{) then } \{ \\ \quad\quad \textit{parents_children} := [\mathcal{S} \mapsto \texttt{Children(pt)}] + [\texttt{un}] \\ \quad\quad \mathcal{S}' := \mathcal{S}'' \oplus \{\texttt{Parent(un) = pt}\} \oplus \{\texttt{Children(pt)} = \textit{parents_children}\} \\ \}\texttt{ else } \{ \\ \quad\quad \mathcal{S}' := \mathcal{S}'' \oplus \{\texttt{Root = un}\} \\ \}\end{array}}{\mathcal{W} \rightarrow \prec\mathcal{S}', \mathcal{P}\succ}$$

Essentially, this does two things. It adds the unique name, `un`, of the activity instance to the stored children of its parent instance, `pt`, as long as the instance is not the root instance of the workflow model. Alternatively, if the instance is the root instance (indicated by `pt` being the empty set), the fluent `Root` is assigned in the workflow state chalkboard to `un`. It also adds to $\mathcal{S}$ the activity type name, `atn`, of `un`, its parent (if it exists), `pt`, and sets the state of `un` to `st(Ready)`.

Then, we wait for the activity instance to be set to `st(Running)` (by its parent) before we expose process logic to effect its behaviour. Once running, we signal on the guard channel `gt` pertaining to the generic type of the customised activity type, `at`, that process logic effecting an instance of the generic type should be exposed, passing parameters `un` – the name of the instance – and `[un1, ..., unn]` – the children of the instance.

The purpose of `AddRootAct` is to handle the creation of an instance of the root activity type. There should be just one of these present in an $\mathcal{L}$CCS workflow model specification. Similarly to `RepAddStructAct`, we firstly create instances of the child (customised) activity types of the root type, signalling on the appropriate `RepAddStructAct` and `RepAddBasicAct` guard channels to do that. Then, we add the instance of the root type that we are creating to $\mathcal{S}$, setting it to the `Running` state by means of `*execute`. Then, similarly to `RepAddStructAct`, we expose process logic to effect the root type by signalling on `gt`, a guard channel which pertains to the generic type of the customised activity type.

The reduction rule for `*execute` is as follows.

ACTIVITY-EXECUTE-READY

$$\frac{\begin{array}{l}\mathcal{W} = \prec\mathcal{S}, \mu.\mathcal{P} + \mathcal{M}\succ \\ \mu = \texttt{*execute(un)} \\ [\mathcal{S} \mapsto \texttt{State(un)}] == \texttt{st(Ready)} \\ \mathcal{S}' := \mathcal{S} \oplus \{\texttt{State(un) = st(Running)}\}\end{array}}{\mathcal{W} \rightarrow \prec\mathcal{S}', \mathcal{P}\succ}$$

The premises of the reduction rule say that the current value of `State(un)` within $\mathcal{S}$ – which reflects the current state of the activity instance `un` – must be `Ready`. The effect of the rule will be to move to the instance into the `Running` state.

The purpose of `RepAddBasicAct` is the same as that of `RepAddStructAdd`, except for basic activity types. Similarly to `RepAddStructAct`, it adds an instance of the customised activity type to the workflow state $\mathcal{S}$. However, as a basic activity type has no children, we simply proceed to signal on the guard channel (which is common to all basic activity types) to expose a copy of the generic logic for all basic activity types. This logic, realised by `RepBasicAct`, simply completes the activity instance (once its parent instance has set it `st(Running)`), according to the `ACTIVITY-COMPLETE` reduction rule, making use of a transaction scope which serves trivially to consume the `*complete` *-action in the event that the instance is cancelled.

Note that the process logic responsible for adding instances of activity types to $\mathcal{S}$ and signalling the *exposure* of process logic effecting generic activity types is put at a higher level of priority than this process logic itself. The reason for this is simple – adding (structured or basic) instances to $\mathcal{S}$ needs to take priority over progressing the workflow process for existing (structured) instances, in order to maintain model consistency. Moreover, the completion of basic activities is put at a further lower level of priority to effect the notion that the model is advanced as far as it can be prior to processing completion events for basic activities. This is an appropriate semantics, as discussed in [13].

We use a transaction scope to specify that, once the root instance has reached a finished state (either `st(Completed)` or `st(Cancelled)`), the remainder of the workflow specification, which will consist solely of replications [13], should be instantaneously garbage-collected.

Finally, if the Liesbet model consists of a single basic activity type, it should simply be specified thus.

```
<[],new un, at *add_activity(at,[],un); *execute(un); *complete(un) >
```

Isolated Scopes

We use `*add_activity` to add instances of activity types which are *not isolated* to the workflow state $\mathcal{S}$. In contrast, we use `*add_scope_activity` to add instances of activity types which are *isolated*. The reduction rule for `*add_scope_activity` is now presented.

ACTIVITY-ADD-SCOPE

$$\frac{\begin{array}{l}\mathcal{W} = \prec\mathcal{S}, \mu.\mathcal{P} + \mathcal{M}\succ \\ \mu = \texttt{*add_scope_activity(atn,pt,un)} \\ \mathcal{S}'' := \mathcal{S} \oplus \{\texttt{Scope(un) = b(true)}\} \\ \mu' := \texttt{*add_activity(atn,pt,un)} \\ \prec\mathcal{S}'', \mu'\succ \rightarrow \prec\mathcal{S}', \mathbf{true}\succ\end{array}}{\mathcal{W} \rightarrow \prec\mathcal{S}', \mathcal{P}\succ}$$

4.3.2 Sync and Cond

The $\mathcal{L}$CCS mapping for the `Sync(StopQuery, GoQuery)` Liesbet activity type is the following process specification, which is part of the library of process specification for the $\mathcal{L}$CCS semantic characterisation of Liesbet. $\mathcal{L}$CCS mappings of Liesbet constructs, presented in the remainder of this article, are also part of the library of process specifications.

```
Sync(StopQuery, GoQuery)(un)
=
   *{GoQuery} + *{¬GoQuery | StopQuery};*cancel(un)
```

The `Sync` process abbreviation is supplied with the `StopQuery` and `GoQuery` queries in the first set of parentheses. These are the same as those specified in the Liesbet Easy Syntax, save for a * being added to queries to turn them into *-actions. So, `Completed_act(at)` would become `*Completed_act(at)`. Note that $\mathcal{L}$CCS-actions are not case-sensitive; the fact that we would usually write this as `*completed_act(at)` makes no difference. Similarly, `True` is mapped to `true`, as a (case-insensitive) identity mapping, and `False` is mapped to `false`; `true` being the degenerative process that is trivially consumed, and `false` being the inconsumable process. The operators + and | also have an identity mapping. `Sync` is also passed, in the second set of parentheses, the unique name of the synchronisation instance, `un`.

In the $\mathcal{L}$CCS specification for `Sync`, there is a prioritised race that takes place between the `GoQuery` and `StopQuery` queries being satisfied, i.e., being consumed. If `GoQuery` is satisfiable at some workflow state then it will win irrespective of the satisfiability of `StopQuery` at the workflow state. The only way for `StopQuery` to win at a particular workflow state is for `GoQuery` to be unsatisfiable at the state.

The reduction rule for $\neg\mathcal{Q}$, where the primary premise of the rule is that $\mathcal{Q}$ is not (atomically) consumable from the current workflow state $\mathcal{S}$, is as follows.

QUERY-NOT

$$\frac{\begin{array}{l}\mathcal{W} = \prec\mathcal{S}, \mu.\mathcal{P} + \mathcal{M}\succ \\ \mu = \neg\mathcal{Q} \\ \neg(\prec\mathcal{S}, \mathcal{Q}\succ \twoheadrightarrow \prec\mathcal{S}', \mathbf{true}\succ)\end{array}}{\mathcal{W} \rightarrow \prec\mathcal{S}, \mathcal{P}\succ}$$

$\prec\mathcal{S}, \mathcal{Q}\succ \twoheadrightarrow \prec\mathcal{S}', \mathcal{Q}'\succ$ in the premises of a rule denotes that $\prec\mathcal{S}', \mathcal{Q}'\succ$ is derivable from $\prec\mathcal{S}, \mathcal{Q}\succ$ by a sequence of one or more $\rightarrow$ reductions.

Note also the use of the $*\{\ldots\}$ containers for `GoQuery` and `StopQuery` in the process specification for `Sync`. These ensure that the queries will be satisfied atomically, according to the following reduction rule.

ACTIVITY-MU-ATOMIC

$$\frac{\begin{array}{l}\mathcal{W} = \prec\mathcal{S}, \mu.\mathcal{P} + \mathcal{M}\succ \\ \mu = *\{\mathcal{Q}\} \\ \prec\mathcal{S}, \mu\succ \twoheadrightarrow \prec\mathcal{S}', \mathbf{true}\succ\end{array}}{\mathcal{W} \rightarrow \prec\mathcal{S}', \mathcal{P}\succ}$$

In Sync, if the GoQuery query is satisfied first, we complete the activity instance – see Section 4.3.1. If the StopQuery query is satisfied first, we cancel the Sync activity instance. The reduction rule for *cancel is presented next. Its effect depends on whether the state of the given instance is st(Completed) or not. If it is st(Completed), the rule says that the *cancel action should be trivially consumed. If it is not st(Completed)

ACTIVITY-CANCEL

$\mathcal{W} = \prec\mathcal{S}, \mu.\mathcal{P} + \mathcal{M}\succ$
$\mu = \texttt{*cancel(un)}$
$State := [\mathcal{S} \mapsto \texttt{State(un)}]$
$\texttt{if}\ (State \neq \texttt{st(Completed)})\ \texttt{then}\ \{$
$\quad\mathcal{S}'' := \mathcal{S} \oplus \{\texttt{State(un) = st(Cancelled)}\}$
$\quad Children := [\mathcal{S}'' \mapsto \texttt{Children(un)}]$
$\quad\mu' := \texttt{*cancel}(0 : Children)\texttt{;...;*cancel}(\iota : Children)$
$\quad\prec\mathcal{S}'', \mu'\succ \twoheadrightarrow \prec\mathcal{S}', \mathbf{true}\succ$
$\}\ \texttt{else}\ \{$
$\quad\mathcal{S}' := \mathcal{S}$
$\}$

$\mathcal{W} \rightarrow \prec\mathcal{S}', \mathcal{P}\succ$

The variant, Sync(GoQuery) is specified as follows, and reflects the fact that there is no StopQuery query to evaluate.

```
Sync(GoQuery)(un)
=
   *{GoQuery}
```

The specification for Cond(GoQuery) is as follows, and reflects the fact that the StopQuery query is, in fact, the negation of the GoQuery query.

```
Cond(GoQuery)(un)
=
   *{GoQuery} + ¬*{GoQuery};*cancel(un)
```

The specification for FreeChoice is as follows, and reflects the fact that the activity instance non-deterministically completes or cancels itself.

```
FreeChoice(un)
=
   *complete(un) + *cancel(un)
```

Note that it does not matter that the RepStructAct container also uses a *complete *-action. The use of *complete in RepStructAct means that most of the process specifications for the generic activity types presented here can be given without their needing to specify *complete themselves. However, sometimes, as here, it is appropriate for the process specification to use a *complete *-action also.

For the purpose of enacting an $\mathcal{L}$CCS Liesbet model, EvalExpr(*EXPR*) queries are mapped to the trivially satisfiable process **true**.

We refer the reader to [13] for the definitions of the reduction rules for *-actions which facilitate the Liesbet queries Completed_act and Completed_all queries, presented in Section 2.2. There exist similar rules for st(Ready), st(Running), and st(Cancelled) states, and rules for Finished_act and Finished_all, which are queries made on instances being in st(Completed) or st(Cancelled) states.

4.3.3 Seq and SeqCancel

The $\mathcal{L}$CCS mapping for the Seq/SeqCancel Liesbet activity types is as follows.

```
Seq(un, [un1, ..., unn])
=
    *execute(un1);*finished(un1);...;*execute(unn)

SeqCancel(un, [un1, ..., unn])
=
   (
    *execute(un1);*completed(un1);...;*execute(unn);
   <t||
    (*cancelled(un1) + ... + *cancelled(unn));*cancel(un)
   )
```

Seq executes each child instance in turn, starting one instance after its predecessor has finished. In contrast, SeqCancel insists that all child instances must complete successfully for the sequence instance to complete. If any of them get cancelled, the sequence instance is cancelled.

4.3.4 Par

The $\mathcal{L}$CCS mapping for the Par Liesbet activity type is as follows.

```
Par(un, [un1, ..., unn])
=
       *{*execute(un1) | ... | *execute(unn)};
```

Par (atomically) executes all child instances in parallel.

4.3.5 DefaultChoice and Choice

The $\mathcal{L}$CCS mapping for the DefaultChoice Liesbet activity type is as follows.

```
DefaultChoice(un, [ung1, unc1, ..., ungn, uncn, und])
=
   *{*execute(ung1) | ... | *execute(ungn)};
   (
      Select(*completed(ung1))(unc1, ung2, unc2, ..., ungn, uncn)
      +...+
      Select(*completed(ungn))(uncn, ung1, unc1, ..., ungn-1, uncn-1)
      +
      Select(*cancelled(ung1) | ... | *cancelled(ungn))(und, unc1, ..., uncn)
   )

Select(Trigger)(unci, u1, ..., uj) =
   Trigger;(*execute(unci)|*cancel(u1)|...|*cancel(uj))
```

DefaultChoice uses Select to select between which instance of the continuation activity types named $\texttt{unc}_1$, ... $\texttt{unc}_n$, $\texttt{unc}_d$ to execute. Select takes two sets of parameters. The first is a process definition, Trigger, which is always a single *-action. Whichever Trigger in the specified Selects is the first to be consumed will determine which Select process instance (in the sum of Selects) continues to be executed. The first parameter in the second set of parentheses is the continuation activity instance $\texttt{unc}_i$ to execute, and the remaining parameters are continuation activity instances that are to be cancelled, as reflected in the definition for Select.

We define a Select process instance for each $\texttt{ung}_i$, and one for the default continuation activity instance. The Trigger for all Selects but the last is that the corresponding $\texttt{ung}_i$ has completed. The trigger for the last Select is that all $\texttt{ung}_i$ instances have been cancelled.

The $\mathcal{L}$CCS mapping for the Choice Liesbet activity type is as follows.

```
Choice(un, [ung1, unc1, ..., ungn, uncn])
=
   *{*execute(ung1) | ... | *execute(ungn)};
   (
      Select(*completed(ung1))(unc1, ung2, unc2, ..., ungn, uncn)
      +...+
      Select(*completed(ungn))(uncn, ung1, unc1, ..., ungn-1, uncn-1)
      +
      (*cancelled(ung1) | ... | *cancelled(ungn));*cancel(un)
   )
```

This is identical to DefaultChoice, except that as there is no default branch, we cancel the whole Choice in the event that all of the guard activity instances go to st(Cancelled).

4.3.6 MultiChoice

The $\mathcal{L}$CCS mapping for MultiChoice is as follows.

```
MultiChoice(un, [ung1, unc1, ..., ungn, uncn])
=
   SelectCancel(ung1, unc1) | ... | SelectCancel(ungn, uncn)

SelectCancel(ung,unc)
=
   *execute(ung);(*completed(ung);*execute(unc) + *cancelled(ung);*cancel(unc))
```

MultiChoice performs a number (one for each guard activity instance) of SelectCancel process instances in parallel. SelectCancel simply executes the passed guard activity instance ung, and waits for it to complete successfully, in which case the continuation activity instance unc is executed, or for it to fail (i.e., get cancelled), in which case the continuation activity instance is cancelled.

4.3.7 CancelActivity

The $\mathcal{L}$CCS mapping for the CancelActivity Liesbet activity type is as follows. There are two different versions, one which takes a reference type (ref_atn) – see Section 2.1 – and one which does not.

```
CancelActivity(atn)(un)
=
   *cancel_all(atn, un)

CancelActivity(atn, ref_atn)(un)
=
   *cancel_all_ref(atn, un, ref_atn)
```

The reduction rule for *cancel_all uses _GetCandidates/4, defined in [13], to retrieve all instances of activity type atn within the visibility horizon of the CancelActivity instance. It effects *cancel on these instances, which in turn will cause st(Ready) and st(Running) instances to go to st(Cancelled).

ACTIVITY-CANCEL-ALL

$$\frac{\begin{array}{l}\mathcal{W} = \prec\mathcal{S}, \mu.\mathcal{P} + \mathcal{M}\succ \\ \mu = \texttt{*cancel_all(atn,src_un)} \\ \mathit{Cands} := \texttt{_GetCandidates(atn, src_un, [], [])} \\ \mu' = \texttt{*cancel(}0 : \mathit{Cands}\texttt{);...;*cancel(}\iota : \mathit{Cands}\texttt{)} \\ \prec\mathcal{S}, \mu'\succ \twoheadrightarrow \prec\mathcal{S}', \mathbf{true}\succ\end{array}}{\mathcal{W} \rightarrow \prec\mathcal{S}', \mathcal{P}\succ}$$

The reduction rule for `*cancel_all_ref` is the same as the rule for `*cancel_all`, except that the visibility horizon is scoped on a plain reference type, `ref_atn`.

ACTIVITY-CANCEL-ALL-REF

$$\frac{\begin{array}{l}\mathcal{W} = \prec\mathcal{S}, \mu.\mathcal{P} + \mathcal{M}\succ \\ \mu = \texttt{*cancel_all_ref(atn,src_un,ref_atn)} \\ \mathit{Cands} := \texttt{_GetCandidates(atn, src_un, ref_atn, [])} \\ \mu' = \texttt{*cancel(}0 : \mathit{Cands}\texttt{);...;*cancel(}\iota : \mathit{Cands}\texttt{)} \\ \prec\mathcal{S}, \mu'\succ \twoheadrightarrow \prec\mathcal{S}', \mathbf{true}\succ\end{array}}{\mathcal{W} \rightarrow \prec\mathcal{S}', \mathcal{P}\succ}$$

Also, as we note in [13], there is an additional reduction rule for the `*execute` *-action. As activity instances can be cancelled before they are executed, we do not wish `*execute` *-actions to be a source of deadlock, should an instance be cancelled, and then an attempt to execute it be made (by its parent instance). As a result, we specify that `*execute` *-actions are trivially consumed on instances that have been cancelled.

5 Verification of Liesbet Models

In this section we introduce our support for static verification of certain properties of workflow models, and in particular the key property of *model soundness*. We describe how we verify this property, as well as how we provide support for checking temporal logic specifications against $\mathcal{L}$CCS characterised Liesbet models.

5.1 Liesbet Model Soundness

In the verification of $\mathcal{L}$CCS-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 [43, 49]. 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.

Deadlock of a Liesbet model refers to a situation where, along some enactment path for a workflow model $\prec\mathcal{S},\mathcal{P}\succ$, we reach a state where we cannot advance $\mathcal{P}$ according to the current $\mathcal{S}$, but the current $\mathcal{P}$ is not (structurally congruent to) the $\mathcal{L}$CCS **true** process (see description for *proper completion*, just below). Every thread of execution (in the enactment of a model) is blocked at a synchronisation point (enforced by a `Sync` or a `Cond` activity); that is, each thread is blocked waiting for some change in the (`Ready`, `Running`, `Completed`, `Cancelled`) state of particular activity instances. (There is 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 [13], this other kind of deadlock does not arise.)

Livelock in a $\mathcal{L}$CCS Liesbet model would be where there are infinite executions of $\mathcal{L}$CCS 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 $\mathcal{L}$CCS-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)`. Although we allow such definitions, we strongly recommend that cyclic behaviour is limited to that which can be expressed using multiple-activity instances (so called structured cycles). For this example, we would represent it instead using a `MultiSeq` activity type (see Appendix A). For structured cycles, model livelock is not possible. So, under the assumption that arbitrary cycles are only used with extreme care, we do not consider model livelock to be a significant issue in Liesbet. As documented in [13], the possibility of livelock at the level of the $\mathcal{L}$CCS characterisation can be discounted according to the $\mathcal{L}$CCS specifications for generic activity types and the template used for $\mathcal{L}$CCS.

We define proper completion for an $\mathcal{L}$CCS Liesbet model $\prec\mathcal{S},\mathcal{P}\succ$ to be achieved when the root workflow instance and all of its descendant instances have reached `Finished` (i.e., `Cancelled` or `Completed`) states. As we enclose an $\mathcal{L}$CCS-characterised Liesbet model in a transaction scope whose trigger is `*finished(un)`, where `un` is the instance of the root activity type of the model, it is sufficient to say that proper completion is achieved when the workflow process $\mathcal{P}$ evolves to (be structurally congruent to) the empty process **true**, as we discuss further in [13]. This will occur if, and only if, we have an absence of locking in a

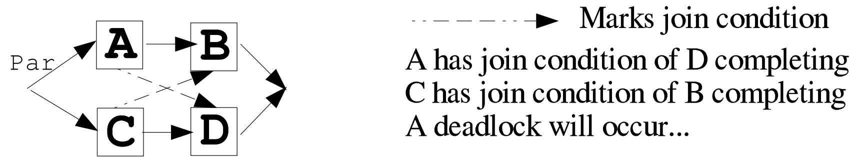


Figure 5: Example 2 – Graphical Representation of Liesbet Model

$\mathcal{L}$CCS Liesbet model. As a result, verification of $\mathcal{L}$CCS Liesbet model soundness reduces to verifying an absence of model deadlock and an absence of dead activity instances [13].

5.2 Constraining/Adjusting the $\mathcal{L}$CCS-Characterisation for Verification Purposes

Model soundness of an $\mathcal{L}$CCS-characterised Liesbet model is decidable, with the following caveat. (With the introduction of *multiple-instance activity types* to Liesbet there is another caveat, as discussed in Appendix A).

The caveat is that we do not admit Liesbet models which have arbitrary cycles. An arbitrary cycle can be thought of as being a repeating piece of workflow logic that cannot be mapped to a multiple-instance activity – essentially because it can have multiple sources (or entrances) and sinks (or exits). Liesbet models may specify arbitrary cycles by simply naming activity types as children that are also ancestor types, but we stipulate that models to be verified may not specify them. We consider this to be a reasonable approach.

The verification process for model soundness may be conceptualised as a number of separate verification runs. 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.

5.3 Approach using SPIN

Briefly, we have a implemented a tool that makes use of the SPIN model-checker [21] to verify model soundness. It is also capable of checking the validity of LTL [9] formulas against a $\mathcal{L}$CCS Liesbet workflow model. SPIN was a natural choice – it is easy to set up, and we wanted to be able to code the $\mathcal{L}$CCS semantic characterisation of Liesbet in the imperative programming language C, mainly for ease of implementation. We are in the process of implementing our own model checking wrapper for Liesbet models so that we can test formulas written in a variety of temporal logics, including CTL [9] and EAGLE [27]. We are also considering the implementation of a simple engine for $\mathcal{L}$CCS so that we can 'execute' $\mathcal{L}$CCS models directly. More information concerning all of these efforts is presented in [13].

As a simple example, consider the workflow presented graphically in Figure 5 and in XML in Figure 6. Here, we have a clear example of *model deadlock*. We see from Figure 5 that to be able to start executing activity `A`, we must have completed activity `D`. In order to have started activity `D`, we must have previously executed activity `C`. Activity `C` requires that activity `B` has completed, but in order to have started activity `B` we must have completed activity `A`.

When we input the Liesbet XML code to our interface to SPIN, we get the results shown in Figure 7, indicating that an invalid end-state error is detected, as required.

```
<?xml version="1.0"?>
<liesbet  xmlns="http://www.doc.ic.ac.uk/adf02/schema/liesbet">
  <par name="P1">
    <seq name="S1">
      <act name="A">
        <join name="J1"><false/><completed_act ref="D"/></join>
      </act>
      <act name="B"/>
    </seq>
    <seq name="S2">
      <act name="C">
        <join name="J2"><false/><completed_act ref="B"/></join>
      </act>
      <act name="D"/>
    </seq>
  </par>
</liesbet>
```

Figure 6: Example 2 – Liesbet XML Code

```
pan: invalid end state (at depth 17)
pan: wrote test3.xml.prm.trail
(Spin Version 4.2.5 -- 2 April 2005)
Warning: Search not completed
        + Compression

Full statespace search for:
        never claim             - (none specified)
        assertion violations    +
        cycle checks            - (disabled by -DSAFETY)
        invalid end states      +

State-vector 100 byte, depth reached 18, errors: 1
       5 states, stored
       0 states, matched
       5 transitions (= stored+matched)
      14 atomic steps
hash conflicts: 0 (resolved)
...truncated
```

Figure 7: Example 2 – SPIN Output

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 [38] and π-calculus/CCS [31, 32, 40], for modelling and verifying the behaviour of processes.

The YAWL Workflow Patterns are a significant contribution made by members of the PMG community [46, 47, 45, 25]. 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 [46, 47, 45, 25, 42, 41, 5, 6, 37]. [46] 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 with an abstract machine language, $\mathcal{L}$CCS, which is inspired by CCS. Importantly, as we show in [13], $\mathcal{L}$CCS has a mapping to standard CCS (with priority), which enables us to make use of the wealth of mathematical results that exist for CCS. A $\mathcal{L}$CCS-characterised Liesbet model is a pair $\prec\mathcal{S},\mathcal{P}\succ$, where $\mathcal{S}$ is a state chalkboard which is updated and queried by means of distinguished *-actions within the workflow process $\mathcal{P}$. We consider that both Liesbet (at the information view) and $\mathcal{L}$CCS (at the computational view) give succinct and intuitive accounts of the YAWL workflow patterns.

In [5] and [37] (resp. [42]) 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.) The $\mathcal{L}$CCS semantic characterisation is rather different from these approaches, however, and has some key advantages, arising largely from its use of a state chalkboard.

The use of a separate state chalkboard which is updated as actions within a process specification are consumed has been found to be a useful representational device in many contexts. An example is the field of (cognitive) robotics, where a common representational device for robot programs is a control program, like a CCS or π process, along with a state chalkboard for recording the effects of executing actions of the control program, where these effects are specified by effect axioms. A robot will be able to *sense* whether the environment has the state it expects, and adjust its representation of the environment accordingly.

An example of a robot programming language is Golog [39]. $\mathcal{L}$CCS is similar to Golog, in that they both employ a state chalkboard (in our case, $\mathcal{S}$), which is updated and queried as a result of actions (in our case, *-actions) that are specified within the control program (in our case, $\mathcal{P}$). That is, $\mathcal{P}$ orchestrates changes to $\mathcal{S}$. The main benefits of using a CCS-like language to represent $\mathcal{P}$ are that CCS provides a natural way of expressing concurrency, as well as process templating through replication. We have also found CCS-like communication to be useful for effecting inter-process synchronisation at a local level (i.e., within generic activity type specifications), as well as using it to provide a way of creating new activity instances and exposing the associated process logic for these instances.

One advantage of the use of a state chalkboard is exemplified by the need of many of the YAWL patterns to query (if only implicitly) the state of activity instances within the workflow model. In our approach, it is trivial to record and answer queries regarding instance

state using the state chalkboard. There is no need, therefore, to route CCS/π channels to appropriate parts of the process specification to effect such state querying, which often proves cumbersome to specify.

An example is the use of cancellation activity instances (pattern #19). We simply update the `State` fluents for all pertaining instances to `st(Cancelled)`, and use *transaction scopes* (...`<t||`...) to garbage-collect redundant $\mathcal{L}$CCS process logic pertaining to activity instances that have been cancelled. This is the extent of the modelling effort to effect cancellation, which is considerably tidier than having to encode cancellation channels as operands of sums at every step of an instance's evolution, and to signal along the channels of the pertaining instances to initiate the cancellation. Moreover, a model author can, through the use of *isolated scopes* and *reference query types*, gain a fine level of control over what activity instances are cancelled.

A further useful by-product of explicitly recording activity instance states in the state chalkboard is that we may support powerful inter-thread synchronisation patterns, which has been a requirement in our work, such as the synchronisation patterns highlighted in [24] and [3], as well as trivially supporting the `Milestone` pattern (#18), as shown in Section 2.2. Again, isolated scopes and reference query types provide a model author with a fine level of control over which extant activity instances are pertinent to a synchronisation instance.

[42] defines a high-level syntax for CCS which abstracts from the cumbersome details of using channels for inter-thread synchronisation and for instance cancellation. Moreover, such a syntax could, in some guise, be used for the purposes of authoring workflow models (an authoring view), or, in other guises, be used for the purpose of exchanging workflow model specifications (a serialisation syntax). It could also be used to give a concise presentation of the expressivity, or ontological commitments, of the workflow modelling approach taken (an information view).

However, it is just as important when fixing the meaning of a workflow model (the computational view) to ensure that the formal tool employed will enable an eloquent characterisation. It is clear from [42], as recognised by the author, that succinctly describing the YAWL workflow patterns purely in CCS is difficult, if not impossible. This is compounded when we consider other needs, such as those for advanced inter-thread synchronisation, as described. Through the use of a state chalkboard and transaction scopes, we have arrived at a semantic characterisation which we believe to be intuitive, tidy, and, relatively to [42], easy to understand. As stated in the introduction to this article, it is important that any computational view has these characteristics.

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 [46] 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, 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 explicitly replicate the process definition to achieve something similar.

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. So, 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.

Our $\mathcal{L}$CCS characterisation of the YAWL workflow patterns is *primarily* process-oriented.

Moreover, Liesbet has been defined, e.g., through the use of isolated scopes and reference query types, in such a way that it lends itself naturally to a process-oriented semantic characterisation. It may also be seen to have data-driven aspects, through the use of a state chalkboard.

It is worth noting that we have extended the Liesbet meta-model to be able to represent the control perspective of WS-BPEL [34] for the purpose of verification of soundness of WS-BPEL compositions. Approaches to the formal specification of WS-BPEL [34] compositions, typically for the purpose of verifying certain properties of compositions, include [8, 7, 2, 50, 29, 23, 15, 26, 16, 14, 10, 35]. For more information regarding our characterisation of WS-BPEL, the interested reader is invited to consult [12].

The question of *when two workflows are equivalent* is an important issue in the study of workflow. As reported in [20], this is a non-trivial question. The crux is how to treat so-called 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 [25]. 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 [20], which we do not seek to resolve, for simplicity we have adopted the approach taken in [25], as described in [13].

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 an abstract workflow model, specified in an extended version of Liesbet that supports WS-BPEL-like fault and compensation handling, 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 [13], 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 [34], and formal models for this language could easily be used in our work instead, as could many other ontologies, such as PSL [19].

Regarding the verification of Liesbet model properties, we have implemented an approach which uses the SPIN model checker, as described in Section 5. We are also in the process of implementing our own model checking wrapper which will be customised for our needs. As well as being able to verify *workflow model soundness*, the wrapper will allow us to verify model properties specified in temporal logics such as LTL [9], CTL [9] and the soft constraint language EAGLE [27].

In [13] we provide a more detailed treatment of the $\mathcal{L}$CCS characterisation of Liesbet, further examples of Liesbet models, and a more comprehensive discussion of verification of $\mathcal{L}$CCS Liesbet models using the SPIN model checker and our custom-built model checking wrapper. We also give an $\mathcal{L}$CCS narrative for the example workflow specification `Par(Seq(A,B),Seq(B,C))` presented in Section 4.3.1, showing how the workflow model evolves during enactment. This is also available at [11].

Acknowledgement

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

References

[1] Gustavo Alonso, Fabio Casati, Harumi Kuno, and Vijay Machiraju. *Web Services, ISBN: 3540440089*. Springer, 2004.

[2] Jesus Arias Fisteus anmd Luis Sanchez Fernandez and Carlos Delgado Kloos. Formal Verification of BPEL4WS Business Collaborations. In K. Bauknecht, M. Bichler, and B. Proll, editors, *EC-Web 2004*, volume 3182 of *Lecture Notes in Computer Science*, pages 76–85. Springer, 2004.

[3] Khalid Belhajjame, Christine Collet, and Genoveva Vargas-Solar. A Flexible Workflow Model for Process-Oriented Applications. In M. Tamer Özsu, Hans-Jörg Schek, Katsumi Tanaka, Yanchun Zhang, and Yahiko Kambayashi, editors, *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, 2001.

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

[5] Yang Dong and Zhang Shensheng. Modeling Workflow Patterns with Pi-calculus. Technical report, Shanghai Jiao Tong University, 2004.

[6] Yang Dong and Zhang Shensheng. Approach for Workflow Modeling Using π-calculus. *Journal of Zhejiang University SCIENCE*, 4(6):643–650, Nov/Dec 2003.

[7] Ziyang Duan, Arthur Bernstein, Philip Lewis, and Shiyong Lu. 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.*, pages 232–241, November 2004.

[8] Ziyang Duan, Arthur Bernstein, Philip Lewis, and Shiyong Lu. Semantics Based Verification and Synthesis of BPEL4WS Abstract Processes. In *Proceedings of the IEEE Conference on Web Services (ICWS'04)*, pages 734–737, 2004.

[9] E. Allen Emerson. Temporal and modal logic. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, pages 995–1072. 1990.

[10] Roozbeh Farahbod, Uwe Glasser, and Mona Vajihollahi. Absract 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, 2005.

[11] Andrew D H Farrell. Enactment Narrative of Liesbet Model: `Par(Seq(A,B),Seq(B,C))`. *Available at* $http://www.doc.ic.ac.uk/\sim adf02/narrative$.

[12] Andrew D H Farrell. Liesbet Support for Verification of WS-BPEL Compositions. *Available at* $http://www.doc.ic.ac.uk/\sim adf02/bpel$.

[13] Andrew D H Farrell. Formalising Workflow: A CCS-inspired Characterisation of the YAWL Workflow Patterns. Technical report, HP Labs Technical Report, 2006, To Appear.

[14] Andrea Ferrara. Web services: a process algebra approach. In Marco Aiello, Mikio Aoyama, Francisco Curbera, and Mike P. Papazoglou, editors, *ICSOC*, pages 242–251. ACM, 2004.

[15] H. Foster, S. Uchitel, J. Magee, and J. Kramer. Model-based verification of web service composition. In *Proceedings of the 18th IEEE International Conference on Automated Software Engineering Conference (ASE 2003)*, 2003.

[16] Xiang Fu, Tevfik Bultan, and Jianwen Su. Analysis of interacting bpel web services. In Stuart I. Feldman, Mike Uretsky, Marc Najork, and Craig E. Wills, editors, *WWW*, pages 621–630. ACM, 2004.

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

[18] 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*.

[19] Michael Gruninger. The Process Specification Language (PSL): Theory and Applications. *AI Magazine*, 24(3):63–74, September 2003.

[20] Jan Hidders, Marlon Dumas, Wil M.P. van der Aalst, Arthur H.M. ter Hofstede, and Jan Verelst. When Are Two Workflows the Same? In Mike Atkinson and Frank Denhe, editors, *Proceedings Computing: The Australasian Theory Symposium, Newcastle, NSW, Australia*, pages 3–11, 2005.

[21] Gerard J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual, ISBN: 0-321-22862-6*. Addison-Wesley, 2004.

[22] S. Jablonski and C. Bussler. *Workflow Management - Modeling Concepts, Architecture and Implementation, ISBN: 1850322228* . International Thomson Computer Press, September 1996.

[23] Raman Kazhamiakin and Marco Pistore. A parametric communication model for the verification of bpel4ws compositions. In Mario Bravetti, Leïla Kloul, and Gianluigi Zavattaro, editors, *EPEW/WS-FM*, volume 3670 of *Lecture Notes in Computer Science*, pages 318–332. Springer, 2005.

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

[25] B. Kiepuszewski. *Expressiveness and Suitability of Languages for Control Flow Modelling in Workflows*. PhD thesis, Queensland University of Technology, Brisbane, Australia, 2003.

[26] Mariya Koshkina and Franck van Breugel. Verification of Business Processes for Web Services, CS-2003-11. Technical report, Department of Computer Science, York University, Toronto, 2003.

[27] U. Dal Lago, M. Pistore, and P. Traverso. Planning with a language for Extended Goals, Technical Report 0205-01. Technical report, Istituto Trentino di Cultura, May 2002.

[28] F. Leymann and D. Roller. *Production Workflow: Concepts and Techniques*. Prentice-Hall PTR, Upper Saddle River, New Jersey, USA, 1999.

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

[30] Mike Marin. Business Process Technology: From EAI and Workflow to BPM. In Layna Fischer, editor, *The Workflow Handbook 2002, ISBN:0-9703509-2-9*. 2002.

[31] Robin Milner. *Communication and Concurrency, ISBN: 0-13-115007-3*. Prentice Hall, 1989.

[32] Robin Milner. *Communicating and Mobile Systems: The π-Calculus, ISBN:0-521-64320-1*. Cambridge University Press, 1999.

[33] Eric Newcomer and Greg Lomow. *Understanding SOA with Web Services, ISBN: 0-321-18086-0*. Addison-Wesley, 2005.

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

[35] Chun Ouyang, Wil M.P. van der Aalst, Stephen Breutel, Marlon Dumas, Arthur H.M. ter Hofstede, and Eric Verbeek. Formal Semantics and Analysis of Control Flow in WS-BPEL, BPM Report BPM-05-15 (Revised Version). Technical report, BPMcenter.org, June 2005.

[36] Benjamin C. Pierce and David N. Turner. Pict: A programming language based on the pi-calculus. In G. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT Press, 2000.

[37] Frank Puhlmann and Mathias Weske. Using the π-calculus for Formalizing Workflow Patterns. In W.M.P. van der Aalst et al, editor, *BPM 2005*, volume 3649 of *Lecture Notes in Computer Science*. Springer, 2005.

[38] Wolfgang Reisig and Grzegorz Rozenberg. *Lectures on Petri Nets I: Basic Models, ISBN:3-540-65307-4*. Springer, 1998.

[39] Raymond Reiter. *Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems, ISBN: 0-262-18218-1*. The MIT Press, 2001.

[40] Davide Sangiorgi and David Walker. *The π-calculus. A Theory of Mobile Processes, ISBN:0-521-78177-9*. Cambridge University Press, 2001.

[41] Christian Stefansen. A SMAll Workflow Language based on CCS, TR-06-05. In *Proceedings of 17th Conference on Advanced Information Systems Engineering, CAiSE05, to appear*, 2005.

[42] Christian Stefansen. A SMAll Workflow Language based on CCS, TR-06-05. Technical report, Harvard University, Division of Engineering and Applied Sciences, Cambridge, MA 02138, March 2005.

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

[44] W.M.P. van der Aalst. Don't go with the flow: Web Services Composition Exposed. In *Trends and Controversies. Web Services: Been there, Done that? IEEE Intelligent Systems*, pages 72–76, Jan–Feb 2003.

[45] W.M.P. van der Aalst, L. Aldred, M. Dumas, and A.H.M. ter Hofstede. 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.

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

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

[48] Vasco T. Vasconcelos. TyCO Gently. DI/FCUL TR 01–4, DIFCUL, July 2001.

[49] H. M. W. Verbeek, T. Basten, and W. M. P. van der Aalst. Diagnosing workflow processes using Woflan. *The Computer Journal*, 44(4):246–279, 2001.

[50] M. Viroli. Towards a Formal Foundation to Orchestration Languages. In M. Bravetti and G. Zavattaro, editors, *Proceedings of 1st International Workshop on Web Services and Formal Methods (WS-FM 2004), volume105 of ENTCS*. Elsevier, 2004.

[51] Sanjiva Weerawarana, Francisco Curbera, Frank Leymann, Tony Storey, and Donald F. Ferguson. *Web Services Platform Architecture, ISBN: 0-13-148874-0*. Prentice Hall, 2005.

[52] Workflow Management Coalition. Workflow Management Coalition Terminology & Glossary. Document Number: WFMC-TC-1011. Document Status: Issue 3.0. February 1999.

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

A Some other Liesbet Constructs, and their $\mathcal{L}$CCS Characterisations

A.1 `DeferredChoice`

The `DeferredChoice` Liesbet construct is a direct facilitation of the YAWL workflow pattern *Deferred Choice* [46, 25], which, similarly to YAWL's Exclusive Choice construct, is *a point in the workflow model where one of several branches is chosen.* In contrast, the choice is actually made by the environment, as opposed to being made explicitly (e.g. based on data or a decision). That is, *several alternatives are offered to the environment; and, once the environment activates one of the branches, the alternative branches are withdrawn.*

Easy Syntax

```
DeferredChoice(ContAct1, ..., ContActn)
```

Informal Operational Semantics

The conceptual meaning of `DeferredChoice` is that an exclusive choice (2.6), or, more generally, a multiple choice (2.7), is made *by the environment* between executing instances of continuation activity types $\texttt{ContAct}_1$, ..., $\texttt{ContAct}_n$. The `DeferredChoice` instance goes to `st(Completed)` when all instances of the chosen continuation types have finished.

However, because we do not model the environment in our $\mathcal{L}$CCS characterisation of Liesbet, and an $\mathcal{L}$CCS-characterisation is a closed system [13], we need to make a simplification to our characterisation of `DeferredChoice`. We simply make a non-deterministic multiple choice between the continuation activity instances. This can be modelled at the information view, or Liesbet meta-model level, as follows.

```
DeferredChoice(ContAct1, ..., ContActn)
=
    MultiChoice(FreeChoice,ContAct1; ...;FreeChoice,ContActn)
```

Note that the guards (`FreeChoice`) are all identical. They make an unbiased, non-deterministic choice between completing a pertaining guard instance successfully or cancelling it.

A.2 `Multimerge`

The `Multimerge` Liesbet construct is a direct facilitation of the YAWL workflow pattern *Multimerge* [46, 25], which is *a point in a workflow model where two or more branches reconverge without synchronisation. If more than one branch gets activated, possibly concurrently, the activity following the merge is started for every activation of every incoming branch.*

Note that the Liesbet `Multimerge` construct subsumes the behaviour of the multimerge described above. The original multimerge workflow pattern, from [46, 25], dictates that the same continuation activity instance be executed for each path that merges. We can accommodate this, but more flexibly we allow that different continuation activities can be chosen.

Easy Syntax

```
Multimerge(Guard1, ..., Guardn; ContAct1, ..., ContActm)
```

Informal Operational Semantics

For `Multimerge`, when running, any of the $\texttt{Guard}_i$ going to `st(Completed)` will cause an instantiation and execution of one of the continuation activities – the first guard to be completed initiates $\texttt{ContAct}_1$, the second to complete initiates $\texttt{ContAct}_2$, and so on. Note,

however, that the number of continuation activities m may be less than (or equal to) the number of $\texttt{Guard}_i$ activities n. When we get to the $m+1$th guard instance to complete, a fresh instance of $\texttt{ContAct}_m$ executed. In other words, $\texttt{Cond}_k = cdCond_m$ for $m < k \leq n$. Any $\texttt{Guard}_i$ instances going to cancelled do not result in the execution of a continuation activity. Note, to effect that all $\texttt{Guard}_i$ instances do result in the execution of a continuation instance, we simply use conditions for each $\texttt{Guard}_i$ that are bound to complete successfully. So, if we are interested in merging workflow paths – the archetypal use of a multi-merge – we simply specify that the last activity instance on each of the paths must have finished.

We also support a variant construct, `MultimergeMax`, which merges a maximum number of branches. Details of this variant are not presented here. The interested reader should consult [13].

In $\mathcal{L}$CCS

In the $\mathcal{L}$CCS mapping, `Multimerge` is passed n guard activity instances, and n continuation activity instances, where the mth continuation activity type has been replicated for continuation types $m+1$ up to, and including, n.

```
MultiMerge(un, [ung1, ..., ungn, unc1, ..., uncn])
=
   new epeel, cpeel
   (
      (
         (SelectMultiMerge(epeel, ung1) | ... | SelectMultiMerge(epeel, ungn));!cpeel
        |
          MergeContAct(epeel, cpeel, unc1); ...; MergeContAct(epeel, cpeel, uncn)
      )
     <t||
      *complete(un)
   )

MergeContAct(epeel, cpeel, unc)
=
   epeel-; *execute(unc);
   +
   cpeel-; *cancel(unc)

SelectMultiMerge(epeel, ung)
=
   *execute(ung);
   (
      *completed(ung); epeel
      +
      *cancelled(ung)
   )
```

`Multimerge` works with two internal channels which control the execution (`epeel`) and cancellation (`cpeel`) of continuation activity instances. Firstly, we use instances of process (one for each guard activity instance) `SelectMultiMerge` to initiate the execution of guard activity instances. When one completes successfully, we signal on `epeel` to start executing a continuation activity instance. Which one is executed is determined by the sequence of `MergeContAct` process instances, each of which blocks on `epeel` and `cpeel`, waiting for an interaction with one of these channels.

Once all the guard instances have finished, we signal using `cpeel` that any remaining continuation activity instances may be cancelled. This is realised through a combination of the replication, `!cpeel`, and the `cpeel` operand of the sum in `MergeContAct`. Then, once

all continuation instances have finished, the `*complete(un)` reduction will be able to be consumed, and as we use the `<t||` operator, the remnants of the `MultiMerge` process will be garbage-collected.

A.3 `Disc` – Discriminator m from n

The `Disc` Liesbet construct is a direct facilitation of the YAWL workflow pattern *Discriminator* [46, 25], which is *a point in a workflow model that waits for one of the incoming branches to complete before activating the subsequent activity. From that moment on it waits for all remaining branches to complete and 'ignores' them.*

We generalise the discriminator pattern by providing an "`M` from `N`" discriminator, where `M` paths must complete before activating the subsequent activity.

Easy Syntax

```
Disc(M, Guard1, ..., Guardn; ContAct)
```

Informal Operational Semantics

When the discriminator is running, it waits until `M` of the named $\texttt{Guard}_i$ instances have gone to `st(Completed)` and then executes an instance of `ContAct`. The discriminator instance goes to completed when the continuation activity and guard instances have finished. If all guard activity instances finish and not enough have completed successfully, then the discriminator goes to `st(Cancelled)`.

In $\mathcal{L}$CCS

The $\mathcal{L}$CCS mapping for the `Disc` Liesbet activity type is as follows.

```
Disc(M)(un, [ung1, ..., ungn, unc])
=
    *int(M,"count",un);
    new dpeel
      (DiscReduceCount(ung1) | ... | DiscReduceCount(ungn)); dpeel-
      | *islessequal(0,"count",un)); *execute(unc); dpeel +
        dpeel; ( *islessequal(0,"count",un)); *execute(unc); +
                 *isgreater(0,"count",un)); *cancel(unc);
               )

DiscReduceCount(ung)
=
  *execute(ung);(*completed(ung);*dec("count",un) + *cancelled(ung))
```

`Disc` takes in the `M` count, reflecting the number of guard activity instances that must complete successfully before the continuation instance, `unc`, can be executed. In order to provide its characterisation, we use a group of *-actions, defined fully in [13], that $\mathcal{L}$CCS provides to facilitate the storage of integer data, within $\mathcal{S}$. Briefly,

- `*int(`V`,`I`,un)` – assigns in $\mathcal{S}$, the value `i(`V`)` to integer fluent I, belonging to activity instance `un`.
- `*dec(`I`,un)` (resp. `*inc(`I`,un)`) – decrements (resp. increments) (by 1) the current value of integer fluent I, belonging to activity instance `un`, in $\mathcal{S}$.
- `*iszero(`I`,un)` (resp. `*isnotzero(`I`,un)`) – is consumed only if the value of integer fluent I, belonging to activity instance `un`, in $\mathcal{S}$ is (resp. is not) i(0).

- `*isgreater(`V`,`I`,un)` (resp. `*isgreaterequal(`V`,`I`,un)`, `*isless(`V`,`I`,un)`, `*islessequal(`V`,`I`,un)`, `*isequal(`V`,`I`,un)`, `*isnotequal(`V`,`I`,un)`) – is consumed only if the value of integer fluent I, belonging to activity instance `un`, in $\mathcal{S}$, is such that $I >$`i(`V`)` (resp. $I >=$`i(`V`)`, $I <$`i(`V`)`, $I <=$`i(`V`)`, $I ==$`i(`V`)`, $I! =$`i(`V`)`).

Each `DiscReduceCount` executes its associated guard instance; and, "count" is decremented by 1 if the guard completes successfully. Once "count" reaches zero, the continuation instance is executed. If all of the guard instances complete and we have not yet set the continuation instance running, we use the local channel `dpeel` to trigger the right-hand operand of the sum which then exposes a further sum which will either execute or cancel the continuation instance depending on whether the threshold count has been reached. Further details explaining why the *Discriminator* pattern is formalised in this way are given in [13].

A.4 Liesbet: Exit

The `Exit` Liesbet construct is a direct facilitation of the YAWL workflow pattern *Cancel Case* [46, 25], which is where *a whole workflow instance is cancelled.*

Easy Syntax

```
Exit
```

Informal Operational Semantics

An `Exit` activity instance immediately cancels the whole workflow model instance.

In $\mathcal{L}$CCS

The $\mathcal{L}$CCS mapping for the `Exit` Liesbet activity type is as follows.

```
Exit(un)
=
   *exit;
```

The (following) reduction rule for `*exit` specifies that we identify the root instance of the workflow model, and effect `*cancel` on it.

ACTIVITY-EXIT

$$\frac{\begin{array}{l}\mathcal{W} = \prec\mathcal{S}, \mu.\mathcal{P} + \mathcal{M}\succ \\ \mu = \texttt{*exit} \\ root := [\mathcal{S} \mapsto \texttt{Root}] \\ \mu' := \texttt{*cancel}(root) \\ \prec\mathcal{S}, \mu'\succ \rightarrow \prec\mathcal{S}', \mathbf{true}\succ\end{array}}{\mathcal{W} \rightarrow \prec\mathcal{S}', \mathcal{P}\succ}$$

A.5 Liesbet: Empty Action

Do nothing! Useful, for example, for an empty default branch in a `DefaultChoice` activity.

Easy Syntax

```
Empty
```

Informal Operational Semantics

An instance of `Empty` trivially completes.

In $\mathcal{L}$CCS

For `ActivityEmpty`, we simply complete, so the $\mathcal{L}$CCS mapping is the empty process, **true**.

```
Empty(un)
=
   true
```

A.6 Liesbet: `Multi*` – Multiple-Instance Activities

Multiple-instance activities enable the creation of multiple instances of the same `ExecAct` activity. These activities fall into two broad categories. Those that limit the number of instances that may be created according to a maximum number, and those that do not.

Easy Syntax

```
MultiLimit(T, Go, ExecAct)
MultiKnown(Go, ExecAct)
Multi(Go, ExecAct)
MultiNoSync(Go, ExecAct, ContAct)

MultiSeqLimit(T, Go, ExecAct)
MultiSeq(Go, ExecAct)
```

Informal Operational Semantics

When a multiple-instance activity is running, multiple instances of `ExecAct` may be created according to instances of the synchronisation activity type, `Go`, successfully completing. Notably an instance of `Go` may only be satisfied in a distinct way from any satisfaction of its previous instances. To this end, a `Go` type will make use (not necessarily exclusively) of `*_dist` sub-queries in its `GoQuery` part – see 2.2. This ensures that the same satisfaction of `Go` can not be used to create multiple instances of `ExecAct`.

Every time an instance of `Go` is satisfied, i.e., moves to `st(Completed)`, an instance of `ExecAct` is created and executed. When this happens, a further instance of `Go` is created and set running, as long as the instance limit, if present, for the multiple-instance activity has not been reached. If a `Go` instance fails (i.e., goes to `st(Cancelled)`), the multiple-instance activity will not allow the creation of any more instances of `ExecAct`, and will move to the `st(Completed)` state once all its offspring `ExecAct` instances have finished. If the multiple-instance activity is cancelled, then all instances of `ExecAct` are cancelled.

Note that the semantic attribution of behaviour to a non-limited multiple-instance activity (namely, `Multi`, `MultiSeq`, and `MultiNoSync`) is rather complex when we allow the `ExecAct` activity type to be a *non-isolated scope*. In the case where `ExecAct` is non-isolated, querying instances (i.e., `Sync` and `Cond` instances, and also, in effect, `CancelActivity` instances) will have a visibility horizon that extends outside the `Multi*` instance, and, moreover, co-related activity instances outside the `Multi` will have access to activity instances within instances of `ExecAct`. As an unlimited number of instances of `ExecAct` may be created, a particular Liesbet model may be undecidable. Thus, to ensure decidability, we stipulate that `ExecAct` types, for non-limited multiple-instance activity types, are always declared as being isolated scopes. If the model author would prefer to use non-isolated `ExecAct` scopes then they should use limited multiple-instance activity types, i.e., `MultiLimit*` and `MultiSeqLimit*`. To the same end, we also stipulate that unlimited multiple-instance activity types themselves run in an isolated scope. As a consequence, `Go` (in non-limited types) may only contain queries on the data perspective, and not the control perspective, of the workflow model. If the model author would like to write queries on the control perspective then, again, a limited type should be used.

We now describe the informal semantics for each of the Liesbet types introduced above.

`MultiLimit(T, Go, ExecAct)`

The `MultiLimit` Liesbet construct is a direct facilitation of the YAWL workflow pattern *Multiple Instances With A Priori Design Time Knowledge* [46, 25], which is where *an activity is enabled multiple times. The number of instances of the given activity is known at design time. Once all instances are completed some other activity needs to be started.*

For `MultiLimit`, the threshold `T` determines that only `T` instances of `ExecAct` may be created.

`MultiKnown(Go, ExecAct)`
`Multi(Go, ExecAct)`

The `MultiKnown` (resp. `Multi`) Liesbet construct is a direct facilitation of the YAWL workflow pattern *Multiple Instances With (resp. Without) A Priori Run Time Knowledge* [46, 25], which is where *an activity is enabled multiple times. The number of instances of the given activity is not known during design time, but is known at some stage during runtime, before the instances of that activity have to be created (resp. nor is it known at any stage during runtime, before the instances of that activity have to be created). Once all instances are completed some other activity needs to be started.*

`MultiKnown` is similar to `MultiLimit` except that its threshold is only known to it at run-time, whereas `Multi` operates according to an absence of a maximum threshold, and will only stop instantiating `ExecAct` when an instance of `Go` goes to `st(Cancelled)`.

`MultiNoSync(Go, ExecAct, ContAct)`

The `MultiNoSync` Liesbet construct is a direct facilitation of the YAWL workflow pattern *Multiple Instances Without Synchronisation* [46, 25], which is where *multiple instances of an activity can be created, i.e., there is a facility to spawn off new threads of control. Each of these threads of control is independent of other threads. Moreover, there is no need to synchronise these threads.*

Its informal operational semantics are similar to that of `Multi`, except that when the `Go` condition finally goes to `st(Cancelled)` to indicate that no more instances of `ExecAct` should be created, the continuation activity `ContAct` is instantiated and executed, irrespective of whether the `ExecAct` instances have finished. That is, there is no synchronisation of the instances of `ExecAct`, unlike (implicitly) other multiple-instance activity types, prior to the continuation activity being instantiated and run.

`MultiSeqLimit(T, Go, ExecAct)`

As `MultiLimit`, but instances of `ExecAct` are created sequentially, in that one instance must go to finished before another is created.

`MultiSeq(Go, ExecAct)`

As `Multi`, but instances of `ExecAct` are created sequentially.

Note that we also support three further variants, whose characterisations in $\mathcal{L}$CCS are not presented in this article. These are briefly described here.

`MultiLimitCompl(T, Go, ExecAct)`
`MultiLimitComplCancelRem(T, Go, ExecAct)`

`MultiSeqLimitCompl(T, Go, ExecAct)`

The limit, `T`, specified for `MultiLimitCompl` determines that no more `ExecAct` instances may be created once `T` instances of `ExecAct` have *completed successfully*. The multiple-instance activity does not itself complete, however, until all extant instances of `ExecAct` have finished. `MultiLimitComplCancelRem` is the same as `MultiLimitCompl` except that when the instance threshold is reached, all outstanding instances of `ExecAct` are cancelled. This enables the multiple-instance activity to complete straightaway. `MultiSeqLimit` is the same as `MultiLimitCompl`, except that instances of `ExecAct` are created sequentially.

In $\mathcal{L}$CCS

MultiLimit

In the following $\mathcal{L}$CCS characterisation, T is the threshold on completion, go is the customised activity type name of Go, and execact is the type name of ExecAct.

```
MultiLimit(T, go, execact)(un)
=
   *int(T,"thresh",un);
   *bool(false,"stop",un);
   new gpeel
   (
       (gpeel- | GoML(go, execact, gpeel, un))
      <t|| (*iszero("thresh",un) + *istrue("stop",un))
   )

GoML(go, execact, gpeel, un) =
    !gpeel; new ungs
          go-<ungs,un>; *execute(ungs);
          (
             *completed(ungs); ExecActML(execact, gpeel, un)
             +
             *cancelled(ungs);*bool(true,"stop",un);
          )

ExecActML(execact, gpeel, un) =
   new une execact-<une,un>; *execute(une); *dec("thresh",un); gpeel-
```

We start by initiating a completion count ("thresh") to the instance limit for the multiple instance activity. We also initiate, to b(false), a boolean value ("stop"), which will be set to b(true) if any go instance fails to complete successfully. The consequence of this should be that no more instances of execact are created. The following boolean-data related *-actions exist.

- *bool(V,B,un) – assigns in $\mathcal{S}$, the value b(V) to boolean fluent B, belonging to activity instance un.
- *istrue(B,un) (resp. *isfalse(B,un)) – is consumed only if the value of boolean fluent B, belonging to activity instance un, in $\mathcal{S}$, is b(true) (resp. b(true)).

Having set up the two fluents in $\mathcal{S}$, we run the GoML process, responsible for creating instances of the go type and subsequently executing instances of execact, in the body of a transaction scope. The trigger for the transaction scope is the "thresh" count going to zero, or the "stop" flag going to b(true) on account of an instance of go going to st(Cancelled). As soon as either happens we garbage-collect the remnants of MultiLimit process.

GoML initiates the creation of a go instance, and subsequently executes it, every time an interaction is performed on gpeel. Once the go instance has completed, the embedded process ExecActML initiates the creation of an instance of execact and proceeds to execute it. Then, it decrements the "thresh" count (by 1), and signals on gpeel that another instance of go should be created. If a go instance is cancelled, then the "stop" flag is set to b(true) meaning that no more instances of execact may be created, and the whole instance logic is garbage-collected.

MultiKnown

MultiKnown is the same as MultiLimit except that instead of the threshold being passed in as a design-time coded parameter of the process specification, we assume that the threshold

value will have been set in $\mathcal{S}$ at run-time, prior to the execution of the `MultiKnown` activity instance. Its specification, therefore, will be identical to `MultiLimit` except for the setting of the "thresh" fluent in $\mathcal{S}$ (i.e., `*int(T,"thresh",un)`), which will be absent.

Multi

`Multi` is also the same as `MultiLimit` except that it does not have any limiting threshold on the number of `execact` instances.

```
Multi(go, execact)(un)
=
   *bool(false,"stop",un);
   new gpeel
   (
       (gpeel- | GoML(go, execact, gpeel, un))
      <t|| (*istrue("stop",un))
   )

GoML(go, execact, gpeel, un) =
    !gpeel; new ungs
          go-<ungs,un>; *execute(ungs);
          (
             *completed(ungs); ExecActML(execact, gpeel, un)
             +
             *cancelled(ungs);*bool(true,"stop",un);
          )

ExecActML(execact, gpeel, un) =
   new une execact-<une,un>; *execute(une); gpeel-
```

MultiNoSync

`MultiNoSync` is essentially the same as `Multi`, except that when the "stop" flag goes to true, we execute the continuation activity instance, `unc`. The specification for `MultiNoSync` is identical to that of `Multi`, save for the first and last lines.

```
MultiNoSync(T, go, execact)(un, [unc])
=
   *bool(false,"stop",un);
   new gpeel
   (
       (gpeel- | GoML(go, execact, gpeel, un))
      <t|| *istrue("stop",un); *execute(unc)
   )
```

`MultiSeqLimit` and `MultiSeq`

For these activity types, executions of `execact` instances are sequential. That apart, they are identical to their non-sequential counterparts, `MultiLimit` and `Multi`. In all cases, this means that the `gpeel-` in `ExecAct*` is placed after it has been ascertained that the current `execact` instance has finished.

MultiSeqLimit

```
ExecActML(execact, gpeel un) =
  new une execact-<une,un>;
         *execute(une); *dec("thresh",un); *finished(une); gpeel-
```

MultiSeq

```
ExecActMNL(execact, gpeel, un) =
  new une execact-<une,un>;
         *execute(une); *finished(une); gpeel-
```

B Liesbet (Easy Syntax) Grammar

The following is a presentation of the grammar of the Easy Syntax for Liesbet in BNF (Backus-Naur Form). Note that the presented grammar is not as restrictive as the Liesbet serialisation syntax presented in [13]. `Activity_Type_Name` ranges over a finite set of customised type names, disjoint from the finite set of generic activity type names, `Activity`.

```
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 ::= Act | StructAct
ActConds ::= join(GuardAct) | trans(GuardAct) | join(GuardAct),trans(GuardAct)
StructAct ::= CancelActs | Choices | Merges | Empty | Exit | MultiActs | ParSeq | SyncActs
CancelActs ::= CancelActivity(CancelAct) | CancelActivity(CancelAct in RefAct)
Choices ::= DefaultChoice(GuardContActs, DefAct) | Choice(GuardContActs) |
            DeferredChoice(ContActs) | MultiChoice(GuardContActs)
Merges ::= Disc(GuardActs, ContAct) | Multimerge(GuardActs, ContActs)
MultiActs ::= MultiLimitActs | MultiNoLimitActs
MultiLimitActs ::= MultiLimit(T, Go, ExecAct) | MultiSeqLimit(T, Go, ExecAct)
MultiNoLimitActs ::= MultiKnown(Go, ExecAct) | Multi(Go, ExecAct) |
                     MultiNoSync(Go, ExecAct, ContAct) | MultiSeq(Go, ExecAct)
ParSeq ::= Par(ExecActs) | Seq(ExecActs) | SeqCancel(ExecActs) | UnorderedSeq(ExecActs)
SyncActs ::= Sync(GoQuery) | Sync(StopQuery, GoQuery) | Cond(GoQuery) | FreeChoice

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

CancelAct ::= Activity_Type_Name
QueryAct ::= Activity_Type_Name
RefAct ::= Activity_Type_Name

T ::= 1 | 2 | ...

GoQuery ::= Query
StopQuery ::= Query
Query ::= Query|Query | Query+Query | !Query | EvalExpr(String) | True | False | QueryOnAct
QueryOnAct ::= QueryOnCompletedAct | QueryOnCancelledAct | QueryOnRunningAct |
               QueryOnReadyAct | QueryOnFinishedAct

QueryOnCompletedAct ::= Completed_Act(Query_Act) | Completed_Act(Query_Act in Ref_Act) |
                        Completed_All(Query_Act) | Completed_All(Query_Act in Ref_Act)

QueryOnCancelledAct ::= Cancelled_Act(Query_Act) | Cancelled_Act(Query_Act in Ref_Act) |
                        Cancelled_All(Query_Act) | Cancelled_All(Query_Act in Ref_Act)

QueryOnRunningAct ::= Running_Act(Query_Act) | Running_Act(Query_Act in Ref_Act) |
                      Running_All(Query_Act) | Running_All(Query_Act in Ref_Act)

QueryOnReadyAct ::= Ready_Act(Query_Act) | Ready_Act(Query_Act in Ref_Act) |
                    Ready_All(Query_Act) | Ready_All(Query_Act in Ref_Act)

QueryOnFinishedAct ::= Finished_Act(Query_Act) | Finished_Act(Query_Act in Ref_Act) |
                       Finished_All(Query_Act) | Finished_All(Query_Act in Ref_Act)
```