An Algebra to Represent Task Flow Models

Carlos Alberto Fernández-y-Fernández[1], Anthony J. H. Simons[2]

[1]Instituto de Electrónica y Computación - Universidad Tecnológica de la Mixteca.
Km. 2.5 carretera Huajuapan – Acatlima. Huajuapan de León, Oaxaca, México

[2]Department of Computer Science - The University of Sheffield
Sheffield, U.K.

{C.Fernandez, A.Simons}@dcs.shef.ac.uk

Abstract. *This paper presents the abstract syntax representation for the Task Flow model in the Discovery Method. The abstract task algebra is based on simple and compound tasks structured using operators such as sequence, selection, and parallel composition. Recursion and encapsulation are also considered. The axioms of the algebra are presented as well as a set of examples showing a combination of basic elements in the expressions.*

1 Introduction

There has been a steady take up in the use of formal calculi for software construction over the last 25 years (Bogdanov, Bowen et al. 2003), but mainly in academia. Although there are some accounts of their use in industry (basically in critical systems), the majority of software houses in the "real world" have preferred to use visual modelling as a kind of "semi-formal" representation of software.

A method is considered formal if it has well-defined mathematical basis. Formal methods provide a syntactic domain (i.e. the notation or set of symbols for use in the method), a semantic domain (like its universe of objects), and a set of precise rules defining how an object can satisfy a specification (Wing 1990). In addition, a specification is a set of sentences built using the notation of the syntactic domain and it represents a subset of the semantic domain.

Spivey says that formal methods are based on mathematical notations and *"they describe what the system must do without saying how it is to be done"* (Spivey 1989), which applies to the non-constructive approach only. Mathematical notations commonly have three characteristics:

- conciseness - they represent complex facts of a system in a brief space;
- precision - they can specify exactly everything that is intended;
- unambiguity - they do not admit multiple or conflicting interpretations.

Essentially, a formal method can be applied to support the development of software and hardware. This paper shows the results of applying a particular process algebra, called Task Algebra, to characterise the Task Flow models in the Discovery Method. The advantage is that this will allow software engineers to use diagram-based design methods that have a secure formal underpinning. The next section presents a general vision of Process Algebra. In section 3 the Task Flow notation used in the

Discovery Method is presented. Section 4 depicts the proposed algebra. Finally in section 5, conclusions are drawn and future research directions are indicated.

2 Process Algebra

The term process algebra or process calculus is used to define an axiomatic approach for processes. There is not a unique definition for processes although Baeten (Baeten 2004) says a process refers to the behaviour of a system. Process Algebras have been used to model concurrent systems (Cheng 1994). Common concepts in the different process algebras are *process* (sometimes called agent) and *action* (Glabbeek 1997). A process can be seen as any concurrent system with a behaviour based in discrete actions. An action is considered something that happens instantaneously and it is atomic. An action is expressed in conjunction with other actions, using particular operations defined by the algebras.

Some of the principal process algebras comprise ACP, CCS, CSP, and more recently Pi-Calculus. The term process algebra was coined by Bergstra and Klop in the paper (Bergstra and Klop 1982) where the Algebra of Communicating Processes (ACP) was presented. The Calculus of Communicating Systems (CCS) was proposed by R. Milner (Milner 1980). The contrasting calculus of Communicating Sequential Processes (CSP) was proposed by Hoare (Hoare 1985). An extension and revision to CCS, the Pi-calculus was later proposed by Milner (Milner 1999).

2.1 ACP

The Algebra of Communicating Processes is an algebra proposed in 1982 when Bergstra and Klop wanted to research a question about unguarded recursive equations (Baeten 2004). The algebra is defined using a combination of instantaneous atomic actions and algebraic operators in order to generate a variety of processes. These operators are used to represent union, concatenation, and concurrency:

- Concatenation, also known as composition or sequencing, uses the symbol · and represents the order of the actions. Where, for instance, $a \cdot b \cdot c$ indicates that action a happens before action b and action b happens before action c.
- Union is used to specify a choice between actions, using the symbol + to represent the union. For example, $a+b$ represents that action a or action b can occur but not both of them.
- Concurrency is represented with the interleaving || and left-merge operator $\mathbin{|\!\lfloor}$, where $p \parallel q$ allows all possible interleavings of actions in the processes p and q, whereas $p \mathbin{|\!\lfloor} q$ always prefers the first action of p before the first action of q and otherwise behaves like ||.

These operators satisfy the following axioms (for all $a \in Action,\ and\ x,y,z \in Process$):

$$x+y=y+x$$

$$x+(y+z)=(x+y)+z$$

$$x+x = x$$

$$(x \cdot y) \cdot z = x \cdot (y \cdot z)$$

$$(x+y) \cdot z = x \cdot z + y \cdot z$$

$$x \parallel y = (x \mathbin{\lfloor\!\lfloor} y) + (y \mathbin{\lfloor\!\lfloor} x)$$

$$(a.x) \mathbin{\lfloor\!\lfloor} y = a.(x \parallel y)$$

$$(x+y) \parallel z = (x \parallel z) + (y \parallel z)$$

$$a \mathbin{\lfloor\!\lfloor} y = a \cdot y$$

As was mentioned, these axioms just expressed the concatenation, union and concurrency (via the left-merge operator). These axioms represent the Basic Process Algebra, which was later extended to include communication an presented by Bergstra in (Bergstra and Klop 1984).

2.2 CCS

Even though the Calculus of Communicating Systems was presented by Milner in 1973, it was not until 1980 that he published the book (Milner 1980) that is now considered the definitive reference on CSS. In CCS a process is represented by a number of states representing the possible lines of action that can be realised. The states of the process are presented as dots (usually open dots), while the actions represent the transitions from a state to other (Glabbeek 1997). The rules and axioms in CCS are provided as laws.

In CCS, 0 (nil) represents the most basic process, offering a deadlock behaviour. CCS also provides an action prefixing operator, where an action a can be prefixed to a process P to denote sequential composition of a and P. An action can be seen as an input or output communication on a port.

The choice operator proposed by Milner in CCS is +. It is commutative, associative and idempotent. Additionally, the CCS operator | represents parallel composition, where, for instance, the expression P/Q depicts two processes running in parallel. Communication between two processes happens when there is an action a in one process and a complementary action $\bar{a}$ in the other one.

2.3 CSP

CSP was proposed by Hoare in (Hoare 1978), initially without a formally defined semantics. Later a semantic model was proposed based on trace theory (Hoare 1981). A new model was proposed and CSP changed its name to Theoretical CSP (TCSP) (Brookes, Hoare et al. 1984), which later was called again CSP.

The trivial element in CSP is the event, which is defined as instantaneous and indivisible. Events are notated in lowercase, for instance x, y, z. are events in CSP. Processes are notated in uppercase. There are also primitive processes such as *STOP* and *SKIP* to represent basic predefined behaviours.

CSP builds processes from actions using a prefix operator $\rightarrow$, such that $x \rightarrow P$ denotes a process formed by prefixing the process P with the action x. CSP has two choice operators, for external and internal choice. The external choice operator $\square$ is defined, such that $(x \rightarrow P) \square (y \rightarrow Q)$ denotes a choice between two processes,

according to whether the environment supplies the event x or y, after which P or Q execute, respectively. The internal choice operator $\sqcap$ makes a nondeterministic choice and may refuse events from the environment. A response is only mandatory if all prefixes are available. Concurrency is represented by the interleaving operator |||, such that P ||| Q denotes a nondeterministic choice between all possible interleavings of the actions of P and Q. The synchronising operator $||_A$ forces its operands to synchronise, such that $P \; ||_A \; Q$ forces synchronised communication between P and Q on all the events in A.

3 The Task Flow Models

The Discovery Method is an object-oriented methodology proposed formally in 1998 by Simons (Simons 1998; Simons 1998); it is considered by the author to be a method focused mostly on the technical process (Simons 2000). From version 1, Discovery has been using a simple and semantically clearer notation based on UML, but changing some models where this is considered appropriate. In addition, it is consistent with the process model of OPEN (Henderson-Sellers, Firesmith et al. 1999), and has been tested in a number of industrial projects by MSc students at the University of Sheffield. The simple and unambiguous Discovery notation makes it an appropriate option to work with.

The Discovery Method is organized into four phases; Business Modelling, Object Modelling, System Modelling, and Software Modelling (Simons 2007).The Business Modelling phase is task-oriented. A task is defined in the Discovery Method as something that "has the specific sense of an activity carried out by stakeholders that has a business purpose" (Simons To be published). This task-based exploration will lead eventually towards the two kinds of Task Diagrams: The Task Structure and Task Flow Diagrams.

The workflow is represented in the Discovery Method using the Task Flow Diagram. It depicts the order in which the tasks are realised in the business, expressing also the logical dependency between tasks. While the notation used in the Discovery Method is largely based on the Activity Diagram of UML, it maintains consistently the labelled ellipse notations for tasks. Figure 1 shows the notation for the Task Flow Diagram.

Tasks are connected by an arrow indicating the direction of the flow. Choice is represented by a diamond and exception, a special case of a choice, is represented using a half-diamond symbol. The full diamond is used to split the flow in two or more choices, whereas the half-diamond symbol represents the choice between continuing the normal flow or the exceptional flow. Because the conditions on the choices are mutually exclusive, the half-diamond decision only needs to express one of the conditions, the one raising the exception. The s*tart* and *end* symbols are the standard symbols used in flowcharts and state diagrams. There is also a particular kind of end symbol identified as *fail*. *Fail* is notated as a small circle crossed by a diagonal line and represents exit with failure from the task described by the diagram. By contrast, the traditional *end* symbol represents an exit with success from the same task.

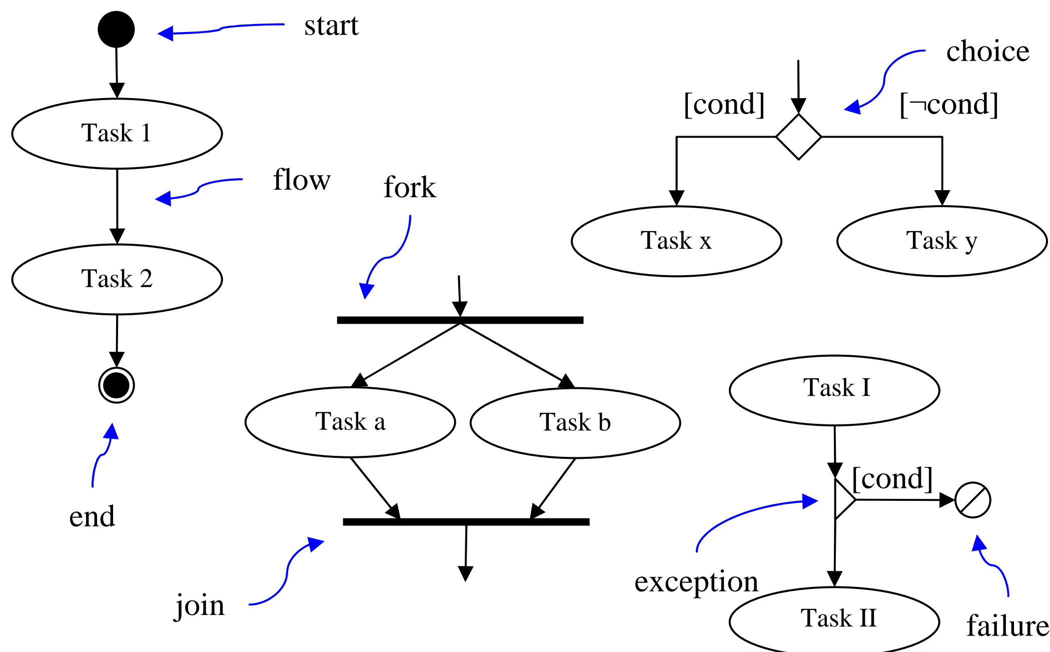

Figure 1 Elements of the Task Flow Diagram

Finally, the Task Flow diagram in the Discovery method allows the representation of parallel tasks. This representation in the diagram is necessary because business processes, just like other kind of processes, are sometimes independent from other processes and, consequently, could be performed concurrently. The Task Flow diagram employs the *fork* and *join* symbols to delimit two or more concurrent flows. The *fork* and *join* symbols are common to many different notations for flow and state diagrams, such as those surveyed by (Beeck 1994). A *fork* is a transition with one source task and many target tasks. A *join* is a transition from many source tasks to one target task. After a fork, the concurrent flows are understood to execute simultaneously. Tasks in each subflow are executed sequentially, but tasks in different subflows may execute in a nondeterministic order. A join indicates a syncrhonisation point, where the concurrent subflows must all terminate before proceeding to the next task. Forks and joins have to be balanced: for each *fork* a corresponding *join* symbol closing the parallel tasks section should exist.

4 The Task Algebra for Task Flow Models

Even though Task Flow models could be represented using one of the process algebras described above, a particular algebra was defined with the aim of having a clearer translation between the graphical model and the algebra. One of the main difficulties with applying an existing process algebra was the notion that processes consist of atomic steps, which can be interleaved. This is not the case in the Task Algebra, where even simple tasks have a non-atomic duration and are therefore treated as intervals, rather than atomic events.

A simple task in the Discovery Method (Simons 1998; Simons 1998; Simons 2002) is the smallest unit of work with a business goal. A simple task is the minimal representation of a task in the model. A compound task can be formed by either simple

or compound tasks in combination with operators defining the structure of the Task Flow Model.

In addition to simple tasks and compound tasks, the abstract syntax also requires the definition of three instantaneous events. These may form part of a compound task in the abstract syntax.

4.1 The Abstract Syntax

The basic elements of the abstract syntax are the simple task, which is defined using a unique name to distinguish from others; ε representing the empty activity; and the success σ and failure ϕ symbols, representing a finished activity.

Simple and compound tasks are combined using the operators that construct the structures allowed in the Task Flow Model. The basic syntax structures for the Task Flow Model are sequential composition, selection, parallel composition, repetition, and encapsulation:

- **Sequential composition** defines the chronological order of execution for a task or a group of tasks from the left to the right and ';' is used as the operator.
- **Selection** is represented with the symbol '+' and it means that there is a choice between the operands.
- **Parallel composition** defines the simultaneous execution of the elements in the expression. It is represented by the symbol '||'.
- **Repetition** allows the reiteration of an expression in the form of an until-loop and while-loop structure. It is represented using the μx fixpoint.
- Finally, **encapsulation** is used to group a set of tasks and structures. This constructs a compound task and is represented using curly brackets *'{' '}'*.

The abstract syntax has the following definition in Backus Naur form:

Activity ::= ε	-- empty activity
| σ	-- *succeed*
| ϕ	-- *fail*
| Task	-- a single task
| Activity ; Activity	-- a sequence of activity
| Activity + Activity	-- a selection of activity
| Activity || Activity	-- parallel activity
| μx.(Activity ; ε + x)	-- until-loop activity
| μx.(ε + Activity ; x)	-- while-loop activity
Task::= Simple	-- a simple task
| { Activity }	-- encapsulated activity

A task can be either a simple or a compound task. Compound tasks are defined between brackets '{' and '}', and this is also called encapsulation because it introduces a different context for the execution of the structure inside it. Curly brackets are used in the syntax context to represent diagrams and sub-diagrams but also have implications for the semantics that will be explained later. Also, parentheses can be used to help

comprehension or to change the associativity of the expressions. Expressions associate to the right by default.

The abstract syntax represents in a simple way every basic structure used for the Task Flow Diagram. For instance, supposing there are three tasks *a, b and c*; a sequence composition of these elements can be specified as follows:

a; b; c

Which means the execution of *a*, then *b*, and then *c*. The selection operator '+' should be used for representing the choice among tasks:

a + b + c

The concurrent execution of these three tasks may be represented using the parallel composition operator '||':

a || b || c

Meaning that *a*, *b* and *c* are executed simultaneously and may terminate in any order. Finally, the repetition operator works either as an until-loop or a while-loop. The difference between each repetition is, as can be supposed, that the until-loop structure guarantees at least one execution of the activity in the repetition:

$\mu x.(a ; \varepsilon + x)$

Repetition is modelled using recursion. In the example above, μx binds x to the expression $(a;\varepsilon+x)$, where a occurs at least once and, if under the choice of x, the expression is expanded (i.e. the expression is repeated recursively, x being the fixed-point of bound by μ.) The next example shows a while-loop:

$\mu x.(\varepsilon + a ; x)$

As in the until-loop, μx binds x to the expression $(\varepsilon+a; x)$, but the choice is put in front of the expression to be repeated.

4.2 Task Model Constructions

Just as the graphical structures of the Task Flow Model can be composed, basic definitions in the abstract syntax may form complex expressions. The abstract syntax definition can be considered like a Universal Algebra which, to accomplish an accurate representation of the diagram syntax, has to be limited by axioms. The abstract syntax definition and its axioms form an Ideal or Quotient Algebra.

4.2.1 Simple task

As it was explained before, a simple task is the minimal representation of a task in the abstract syntax with significance for the expressions; whilst an Activity is formed using a combination of operators (sequence, selection parallel composition, and repetition), simple tasks, empty activity, end with success and end with fail. Empty and finished activities are vacuous activities. Empty is represented with ε, success with σ and fail using ϕ. The fact that simple tasks cannot be vacuous activities is formalised in the next axioms:

(sp.1) $\forall a \in Simple \bullet a \neq \varepsilon \wedge a \neq \phi \wedge a \neq \sigma$

(sp.2) $\forall a \in Simple \bullet \forall y, z \in Activity \bullet a \neq (y; z) \wedge a \neq (y + z)$
$\wedge\, a \neq (y \,||\, z) \wedge a \neq \mu x.(y; \varepsilon + x) \wedge a \neq \mu x.(\varepsilon + y; x)$

Simple tasks are different from *succeed*, *fail* and empty activities because simple tasks represent processes with interval duration different from zero. *Succeed*, *fail* and empty activities are considered instantaneous events.

4.2.2 Empty activity

As was said before, the symbol ε is used to represent the empty activity. It is needed because the selection is a binary operator and ε is used to characterise the empty branch, where in combination with the selection operator it is used as the choice between doing something or nothing.

As a result of the existence of this element, a set of axioms must be defined to interpret the meaning of the empty activity when it is a part of other kinds of expression. These rules are specified within each operator description.

4.2.3 Finished activity

The finished activity is necessary to represent situations when an activity should terminate before the normal end. The abstract syntax representation allows two kind of finished activity: *succeed* and *fail*. *Succeed* is useful to represent an early exit from within an expression, returning the control to the higher scope. On the other hand, *fail* is used to represent the termination of all tasks, and the failure is propagated to the higher levels.

σ and ϕ are considered instantaneous events. Similar to the empty activity, the finished activities have an effect in many operator constructions, and they will be defined later.

4.2.4 Sequential composition

Sequential composition is defined as the consecutive execution of activities, from the left to the right. Tasks are separated by ';'. For example:

a; b; c ; d ⇔ a; (b; (c ; d))

The intuitive meaning is that first *a* will be executed, then *b*, and so on until the task *d*. Parentheses can be used to group elements but the meaning is not altered whatsoever. An associative axiom is defined to support this notion. Axioms for distribution, empty sequence and finished activity are also defined. Commutativity and idempotence properties are not considered for sequences:

(s.1) $\forall a,b,c \in Activity \bullet a;(b;c) \Leftrightarrow (a;b);c$ -- associative sequence

(s.2) $\forall a,b,c \in Activity \bullet (a+b);c \Leftrightarrow (a;c)+(b;c)$ -- right distributivity of sequence over selection

(s.3) $\forall a \in Activity \bullet a;\varepsilon \Leftrightarrow \varepsilon;a \Leftrightarrow a$ -- empty sequence

(s.4) $\forall a \in Activity \bullet \phi;a \Leftrightarrow \phi$ -- *fail* on sequence

(s.5) $\forall a \in Activity \bullet \sigma;a \Leftrightarrow \sigma$ -- *succeed* on sequence

Rule (s.2) defines that a right sequence is distributed over a left selection. Left distribution of sequence over selection is not allowed because, as in ACP (Baeten and Weijland 1990; Baeten 2004), left sequence distribution changes the point where the choice is made. It follows that:

$\forall a, b, c \in Activity \bullet a; (b + c) \neq (a; b) + (a; c)$

Because in the expression *a;(b+c)*, initially *a* is executed and then the choice between *b* and *c* is made; while in the expression *(a;b)+(a;c)* the choice is first and afterwards *a* is executed. The difference in the branching position can be easily appreciated in Figure 2.

Figure 2 State transition diagram for expressions a;(b+c) and (a;b)+(a;c)

Empty and finished activities may coexist in an expression, in which case the rules (s.3), (s.4) and (s.5) are confluent and may interact, for instance:

a) a1; ϕ; ε; a2

$\Rightarrow$ a1; ϕ -- by applying (s.4), or (s.3) and (s.4)

b) a1; ε; σ; a2

$\Rightarrow$ a1; σ -- by applying (s.3) and (s.5)

Or:

c) ϕ; ε $\Leftrightarrow$ ε; ϕ $\Leftrightarrow$ ϕ -- by applying (s.3) and (s.4)

d) σ; ε $\Leftrightarrow$ ε; σ $\Leftrightarrow$ σ -- by applying (s.3) and (s.5)

4.2.5 Selection

The selection of activities is performed with the '+' operator. It represents the choice among a group of activities, for instance:

a + b + c + d $\Leftrightarrow$ a + (b + (c + d))

Intuitively each branch is evaluated from the left to the right. Guards are implicit and are not represented in the syntax. The guards are supposed to be mutually exclusive and exhaustive. When a guard is satisfied the left activity is executed and the right branch is discarded, otherwise the left activity is discarded and the next guard is verified. Logically, the last guard does not need to be checked and the order in which the branches are considered is irrelevant.

The axioms defined for the selection operator are:

(sel.1) $\forall a, b, c \in Activity \bullet (a+b)+c \Leftrightarrow a+(b+c) \Leftrightarrow a+b+c$

-- associative selection

(sel.2) $\forall a, b \in Activity \bullet a+b \Leftrightarrow b+a$ -- commutative selection

(sel.3) $\forall a \in Activity \bullet a+a \Leftrightarrow a$ -- idempotent selection

In the case of the empty activity, it is also possible to reduce the expression if both sides have the empty activity by the idempotent rule (sel.3). But, if just one of the elements (right or left) is ε, then the selection has no reductions. $\forall a \in Activity$, the following expressions are irreducible:

$a + \varepsilon$ -- irreducible selection of empty activity or activity

The same applies to the finished activities, where the selection between any of the finished activities or a general *Activity* has no reduction:

$\phi + a$ -- irreducible selection of *fail* or activity

$\sigma + a$ -- irreducible selection of *succeed* or activity

$\varepsilon + \phi$ -- irreducible selection of empty activity or *fail*

$\varepsilon + \sigma$ -- irreducible selection of empty activity or *succeed*

As described above, selection interacts with sequences and the right distributivity axiom may be applied. Its interaction with parallel composition is shown below.

4.2.6 Parallel composition

Parallel composition is defined as the simultaneous execution of all its tasks and it is represented with the operator '||'. An example is the expression:

$a \,||\, b \,||\, c \,||\, d \Leftrightarrow a \,||\, (\, b \,||\, (\, c \,||\, d\,)\,)$

Intuitively it expresses that the elements *a*, *b*, *c*, and *d* are initiated at the same time and executed simultaneously. The end of any of them is non-deterministic. Like the last operators, a set of axioms are defined:

(p.1) $\forall a, b, c \in Activity \bullet (a \,||\, b) \,||\, c \Leftrightarrow a \,||\, (b \,||\, c)$ -- associative parallel composition

(p.2) $\forall a, b \in Activity \bullet a \,||\, b \Leftrightarrow b \,||\, a$ -- commutative composition

(p.3) $\forall a, b, c \in Activity \bullet (a+b) \,||\, c \Leftrightarrow (a \,||\, c) + (b \,||\, c)$

-- right distributivity of concurrency over selection

(p.4) $\forall a \in Activity \bullet a \,||\, \varepsilon \Leftrightarrow a$ -- instant synchronisation

(p.5) $\forall a \in Activity \bullet a \,||\, \phi \Leftrightarrow \phi$ **if** $a \neq \sigma$ -- instant failure

(p.6) $\forall a \in Activity \bullet a \,||\, \sigma \Leftrightarrow \sigma$ -- instant success

The associative and commutative axioms (p.1, p.2) reflect the nondeterministic order of concurrent activity. Also, it is possible to do right and left distribution of

concurrent composition over selection, but only the one axiom is necessary. Right distribution over selection is defined in (p.3) and left distribution is derived by applying (p.1) and (p.3):

$\forall a,b,c \in Activity \bullet a \parallel (b+c) \Leftrightarrow (a \parallel b) + (a \parallel c)$ -- left distribution of concurrency over selection, by axiom (p.1) and (p.3)

The use of instant events such as ε, σ and ϕ may occur too in combination with parallel composition. Axioms (p.4), (p.5) and (p.6) define instant synchronisation, *fail* and *succeed* respectively. Whilst (p.4) performs the elimination of ε whether it is on the right or the left of the parallel operator, (p.5) and (p.6) establish that any activity in parallel composition with *fail* or *succeed* is equivalent to just itself. Although the parallelism is resolved as the simultaneous execution of simple activities (i.e. concurrency between a single task and an *Activity* means that the single task could occur at any time among all the simple actions of such Activity), *Succeed* and *fail* are considered as instantaneous events and they have priority over the elements of the *Activity*. In addition, *succeed* has a major priority than *fail*, therefore in the case of a parallel composition between these two elements *succeed* will prevail (p.6).

Logically, this set of rules is confluent, which can be easily proved. The specific case of ϕ //ε can be resolved using any of the rules defined for each symbol to work with parallel composition. For example, the next expressions are equivalents:

$\phi \parallel \varepsilon \Leftrightarrow \varepsilon \parallel \phi \Leftrightarrow \phi$ -- from (p.2), (p.4) and (p.5)

The result is obtained by applying either the rule (p.4) or rule (p.5). The instantaneous events are also confluent with the rest of the parallel axioms:

$\forall a,b \in Activity \bullet (a \parallel b) \parallel \varepsilon \Leftrightarrow a \parallel (\varepsilon \parallel b) \Leftrightarrow a \parallel b \parallel \varepsilon \Leftrightarrow a \parallel b$ -- by (p.1) and (p.4)

$\forall a \in Activity \bullet a \parallel \varepsilon \Leftrightarrow \varepsilon \parallel a \Leftrightarrow a$ -- by (p.2) and (p.4)

$\forall a \in Activity \bullet a \parallel \phi \Leftrightarrow \phi \parallel a \Leftrightarrow \phi$ **if** $a \neq \sigma$ -- by (p.2) and (p.5)

$\forall a \in Activity \bullet a \parallel \sigma \Leftrightarrow \sigma \parallel a \Leftrightarrow \sigma$ -- by (p.2) and (p.6)

$\forall a,b \in Activity \bullet (a \parallel b) \parallel \phi \Leftrightarrow a \parallel (\phi \parallel b) \Leftrightarrow a \parallel b \parallel \phi \Leftrightarrow \phi$

if $a \neq \sigma \wedge b \neq \sigma$ -- by (p.1) and (p.5)

$\forall a,b \in Activity \bullet (a \parallel b) \parallel \sigma \Leftrightarrow a \parallel (\sigma \parallel b) \Leftrightarrow a \parallel b \parallel \sigma \Leftrightarrow \sigma$ -- by (p.1) and (p.6)

$\forall a,b \in Activity \bullet (a+b) \parallel \varepsilon \Leftrightarrow (a \parallel \varepsilon) + (b \parallel \varepsilon) \Leftrightarrow a+b$ -- by (p.3) and (p.4)

$\forall a,b \in Activity \bullet (a+b) \parallel \phi \Leftrightarrow (a \parallel \phi) + (b \parallel \phi) \Leftrightarrow \phi$

if $a \neq \sigma \wedge b \neq \sigma$ -- by (p.3) and (p.5)

4.2.7 Repetition

Repetition of tasks is defined as an until- and while-loop. The structures in the abstract syntax are constructed using recursion. The until-loop is formed by an *Activity* followed by an option of continuing or repeating *x*:

$\mu x.(a; \varepsilon + x)$

Intuitively can be seen that the *Activity* is repeated as long as ε is not chosen. When ε is chosen (i.e. the end state of the recursion function is reached) the recursion terminates, which means that the next activity outside of the until-loop may be executed. The choice of the fixed-point x results in expanding unrolling the expression.

The until-loop has only one axiom specifying the unrolling of the recursions on the loop:

(r.1) $\forall a \in Activity \bullet \mu x.(a;\varepsilon + x) \Leftrightarrow a;\varepsilon + \mu x.(a;\varepsilon + x)$

-- unrolling one cycle of until-loop repetition

This rule can be applied as many times as necessary resulting possibly in an infinite repetition of the activity and the option to continue or repeat:

$\mu x.(a;\varepsilon + x) \Rightarrow a;\varepsilon + \mu x.(a;\varepsilon + x) \Rightarrow a;\varepsilon + (a;\varepsilon + \mu x.(a;\varepsilon + x)) \Rightarrow ...$ -- by (r.1)

Additionally, there are three special cases where the expression may be reduced, those ones when any of the instantaneous events is involved. In one case an until-loop containing just the empty element ε can be reduced just to ε:

$\mu x.(\varepsilon;\varepsilon + x) \Rightarrow \varepsilon;\varepsilon + \mu x.(\varepsilon;\varepsilon + x) \Rightarrow \varepsilon;\varepsilon + (\varepsilon;\varepsilon + \mu x.(\varepsilon;\varepsilon + x)) \Rightarrow ... \Rightarrow \varepsilon$

-- by (r.1) and (s.3)

The reduction of empty sequences can be made by the axiom (s.3). The recursion keeps going infinitely or finishes when the ε in the selection is chosen.

On the other hand, if the activity in the until-loop contains just ϕ or σ, the expression may be reduced and the recursion is eliminated:

$\mu x.(\phi;\varepsilon + x) \Rightarrow \phi;\varepsilon + \mu x.(\phi;\varepsilon + x) \Rightarrow ... \Rightarrow \phi$ -- by (r.1) and (s.4)

$\mu x.(\sigma;\varepsilon + x) \Rightarrow \sigma;\varepsilon + \mu x.(\sigma;\varepsilon + x) \Rightarrow ... \Rightarrow \sigma$ -- by (r.1) and (s.5)

As the examples above show, it is possible to reduce the until-loop using the axioms (s.4) or (s.5) already defined.

Alternatively, the while-loop is formed by the option of doing an *Activity* followed by repeating x, or the option of finishing the execution of the loop:

$\mu x.(\varepsilon + a ; x)$

As the until-loop, the while-loop has only one axiom specifying the unrolling of the recursions on the loop:

(r.2) $\forall a \in Activity \bullet \mu x.(\varepsilon + a;x) \Leftrightarrow \varepsilon + a;\mu x.(\varepsilon + a;x)$

-- unrolling one cycle of while-loop repetition

Applying this rule as many times as necessary results in an infinite repetition of the option to finish the loop or doing the activity and repeat:

$\mu x.(\varepsilon + a;x) \Rightarrow \varepsilon + a;\mu x.(\varepsilon + a;x) \Rightarrow \varepsilon + a;(\varepsilon + a;\mu x.(\varepsilon + a;x)) \Rightarrow ...$ -- by (r.2)

Additionally, there are three special cases where the expression may be reduced, those ones when any of the instantaneous events is involved. The while-loop containing just the empty element ε can be reduced just to ε:

$\mu x.(\varepsilon + \varepsilon; x) \Rightarrow \varepsilon + \varepsilon; \mu x.(\varepsilon + \varepsilon; x) \Rightarrow \varepsilon + \varepsilon; (\varepsilon + \varepsilon; \mu x.(\varepsilon + \varepsilon; x)) \Rightarrow ... \Rightarrow \varepsilon$

-- by (r.2) and (s.3)

The reduction of empty sequences can be made by the axiom (s.3). The recursion keeps going infinitely or finishes when the ε in the selection is chosen. Finally, in the cases where the activity in the while-loop contains only the symbol ϕ or σ, the expression may be reduced and the recursion is eliminated:

$\mu x.(\varepsilon + \phi; x) \Rightarrow \varepsilon + \phi; \mu x.(\varepsilon + \phi; x) \Rightarrow ... \Rightarrow \varepsilon + \phi$ -- by (r.2) and (s.4)

$\mu x.(\varepsilon + \sigma; x) \Rightarrow \varepsilon + \sigma; \mu x.(\varepsilon + \sigma; x) \Rightarrow ... \Rightarrow \varepsilon + \sigma$ -- by (r.2) and (s.5)

4.2.8 Encapsulation

The encapsulation of tasks is used to isolate an *Activity* from the rest of the expression giving it a scope and a name. It is built by using curly brackets "{ }" around the *Activity*. Consequently, *{act}* represents the encapsulation of the *Activity act*. But, the real importance of encapsulation is denoting the scope of a compound task to limit the effect of σ and ϕ, which represent early exit. A more detailed example could be:

{{a1; {a2+a3}; a4}; a5}

Supposing *a1, a2, ..., a5* are simple tasks, in that case the expression also could be expressed as a set of compound tasks:

let X = {a2+a3}

let Y = {a1; X; a4}

{Y;a5}

Using encapsulation is a way of abstracting the representation of a complex task flow and treating it as a single task (i.e. a subtask, part of another larger task), in the same way that a complex diagram can be divided into different sub-diagrams to facilitate comprehension.

As mentioned above, when a *succeed* event occurs in an expression, this corresponds to an early exit from the scope of the enclosing task. The normal flow of control resumes at the task boundary. A different result is obtained when a *fail* event occurs in the expression. In this case, the *fail* event is promoted to the higher level, beyond the immediate task boundary. All the usual axioms apply to activity that is encapsulated within a task. Some additional axioms describe the specific effects of σ at the task boundary:

(e.1) $\{\sigma\} \Leftrightarrow \varepsilon \Leftrightarrow \{\varepsilon\}$ -- vacuous subtask

(e.2) $\forall a \in Activity \bullet \{a; \sigma\} \Leftrightarrow \{a\}$ -- coincident exit

(e.3) $\forall a \in Activity \bullet \{a + \sigma\} \Leftrightarrow \{a\} + \varepsilon$ -- vacuous selection

(e.4) $\{\phi\} \Leftrightarrow \phi$ -- promotion of fail

(e.5) $\forall a \in Activity \bullet \{a; \phi\} \Leftrightarrow \{a\}; \phi$ -- promotion of fail in sequence

(e.6) $\forall a \in Activity \bullet \{a + \phi\} \Leftrightarrow \{a\} + \phi$ -- promotion of fail in selection

The vacuous subtask axiom (e.1) denotes that *succeed* alone within curly brackets is equivalent to the empty activity because *succeed* has no influence outside of its scope. Similarly, if *succeed* is next to the left bracket, it has no effect and may be removed even forming part of a sequence (e.2). The axiom (e.3) promotes the selection outside of the encapsulation area changing *succeed* for ε. Basically it establishes that a selection between an activity and *succeed* is equivalent to the choice of that activity within brackets and nothing (ε). If *fail* is alone within the curly brackets, it is promoted to the higher level by the axiom (e.4). The axiom (e.5) denotes the promotion of *fail* when this is next to the left bracket in a sequence. Finally, the axiom (e.6) promotes the selection and *fail* outside the curly brackets.

Additional axioms are not required for parallel composition and repetition, since the transformations can be derived from the existing ones:

$\forall a \in Activity \bullet \{a \,||\, \sigma\} \Leftrightarrow \varepsilon$ -- by (p.5) and (e.1)

$\{\mu x.(\sigma; \varepsilon + x)\} \Leftrightarrow \varepsilon$ -- by (r.1), (s.5) and (e.1)

5 Conclusions

The present paper depicted the abstract syntax representation for the Task Flow model in the Discovery Method. The abstract task algebra is based on simple and compound tasks structured using operators such as sequence, selection, and parallel composition. Recursion and encapsulation are also considered. The axioms of the algebra were presented as well as a set of examples showing a combination of basic elements in expressions denoting simple, and more complex, Task Flow diagrams.

Current work in progress involves the definition of the denotational semantics for the task algebra, giving the semantics in terms of traces. Additionally, model-checking techniques will be developed to validate Task Models represented in the algebra.

6 References

Baeten, J. C. M. (2004). A brief history of process algebra, Technische Universiteit Eindhoven.

Baeten, J. C. M. and W. P. Weijland (1990). Process algebra. Cambridge; New York, Cambridge University Press.

Beeck, M. v. d. (1994). "A Comparison of Statecharts Variants." Lecture Notes In Computer Science **863**: 128-148.

Bergstra, J. A. and J. W. Klop (1982). Fixed point semantics in process algebras. Amsterdam, Mathematical Centre.

Bergstra, J. A. and J. W. Klop (1984). The Algebra of Recursively Defined Processes and the Algebra of Regular Processes. 11th ICALP, Springer Verlag.

Bogdanov, K., J. P. Bowen, et al. (2003, December 2003). "Working together: Formal Methods and Testing." Retrieved June 2004, 2004, from http://www.fortest.org.uk/documents/landscape3.pdf.

Brookes, S. D., C. A. R. Hoare, et al. (1984). "A Theory of Communicating Sequential Processes." J. ACM **31**(3): 560-599.

Cheng, M. H. M. (1994). "Calculus of Communicating Systems: a synopsis." from citeseer.ist.psu.edu/cheng94calculu.html.

Glabbeek, R. J. v. (1997). "Notes on the methodology of CCS and CSP." Theoretical Computer Science **177**(2): 329-349.

Henderson-Sellers, B., D. G. Firesmith, et al. (1999). "Instanting the process metamodel." Journal of Object-Oriented Programming (ROAD) **12**(3): 51-57.

Hoare, C. A. R. (1978). "Communicating sequential processes." Communications of the ACM **21**(8): 666-677.

Hoare, C. A. R. (1981). A Model for Communicating Sequential Processes. U. o. O. C. L. Programming Research Group. Oxford, University of Oxford.

Hoare, C. A. R. (1985). Communicating sequential processes. Englewood Cliffs, N.J., Prentice/Hall International.

Milner, R. (1980). A calculus of communicating systems. Berlin; New York, Springer-Verlag.

Milner, R. (1999). Communicating and mobile systems: the pi-calculus. Cambridge, UK, Cambridge University Press.

Simons, A. J. H. (1998). Object Discovery - A process for developing applications. Workshop 6, British Computer Society SIG OOPS Conference on Object Technology (OT '98), Oxford, BCS.

Simons, A. J. H. (1998). Object Discovery - A process for developing medium-sized applications. Tutorial 14, 12th European Conference on Object-Oriented Programming (ECOOP '98), Brussels, AITO/ACM.

Simons, A. J. H. (2000). "The Discovery EBook." Retrieved June 2004, 2004, from http://www.dcs.shef.ac.uk/~ajhs/discovery/ebook/.

Simons, A. J. H. (2002). "Discovery Method. Systems Analysis and Design for Object-Oriented Applications." Retrieved June 2004, 2004, from http://www.dcs.shef.ac.uk/~ajhs/campus_only/com3410.html.

Simons, A. J. H. (2007). Discovery and Invention (in preparation - personal communication).

Simons, A. J. H. (To be published). Discovery Method. (Discovery and Invention). Sheffield, University of Sheffield.

Spivey, J. M. (1989). "An Introduction to Z and Formal Specifications." Software Engineering Journal IEEE/BCS **4**(1): 40-50.

Wing, J. M. (1990). "A Specifier's Introduction to Formal Methods." IEEE Computer **23**(9): 8-24.