Modelling Processes Using a Stepwise Refinement Technique

Jin Sa and B.C. Warboys,

Department of Computer Science,
University of Manchester,
Manchester M13 9PL,
England.

Published in the proceedings of the Third Workshop on Software Process Technology EWSPT'94, February, 1994, Villard de Lans (Grenoble), France, also in LNCS 772.

Modelling Processes Using a Stepwise Refinement Technique

Jin Sa and Brian C. Warboys

Department of Computer Science,
University of Manchester,
Manchester M13 9PL, England.
emails: {jsa@cs.man.ac.uk, brian@cs.man.ac.uk}

Abstract. In many existing process modelling approaches, the emphasis is on the *representations* of process models, but not so much on the *development* of them. In this paper we present a method, called *OBM*, for developing such representations using a stepwise refinement technique. An example is used to show how a model may be gradually refined.

1 Background and Introduction

In EWSPT'92 [10], we reported a small case study [24] which we undertook to investigate the application of an existing concurrent formal specification method, Base Model (BM) [23], to process modelling. The case study showed that although BM could be used as a specification method for some applications, particularly those targeted by the process modelling language PML [5], there was still a wide scope for improvement so that BM could be a more suitable technique for process modelling.

Our subsequent work mainly concentrated on specifying[1] the ISPW-6 software process problem example (ISPW-6 example for short) [15]. As a result of this work, BM was specialised with some constructs specific to modelling software process. The details of this work have been described in [21].

Since the ISPW-6 example was clearly described, the emphasis of the modelling exercise as reported in [21] was mainly on the *representation* of a model, but not so much on the *development* of a model. This emphasis is in fact also true with many existing process modelling techniques. However, in reality, processes are often unclear to the process participants and to the process modeller. This is especially true in organisations, where the same process may be viewed from different angles or at different levels of abstraction by different people or at different times. This makes it very difficult for the process modeller to model all the details of a process. In this paper, we present an approach, called Organisational Base Model (OBM), which provides a means to gradually identify the details of a process in an organisation and develop the detailed representation of the organisational process by a stepwise refinement technique.

[1] The words "specify", "define" and "model" are often used interchangeably in this paper.

OBM is based on BM. The specification constructs are defined using a temporal logic as it is a widely accepted formal tool for the specification and verification of concurrent systems [17].

2 Brief Overview of OBM

In OBM, a system is considered to be composed of components which may be executed concurrently. Each component provides a number of operations and contains some variables which can only be accessed by the operations. An operation may call other operations in other components. An *active* operation of a component is invoked by the component itself, and a *passive* operation is invoked by another operation in another component (or by an agent).

The model of a system is developed at different levels. Each level specifies the system at a certain level of abstraction. The specification at each level contains a collection of component specifications. Each component specification defines the operations and the pattern to which its operations, when executed, conform. Such patterns are called *operation patterns*.

In developing a model, *abstract objects* are used to define components which may be refined in the subsequent levels. *Agent objects* and *step objects* are used to define components which will no longer be refined. *Boxed objects* are used to compose a number of components. The process for developing a model using OBM is as follows:

1. Define the top level as an abstract object or a collection of abstract objects.
2. At each level, e.g. level i, for each abstract object A in level i, a level i+1 refinement may be carried out to produce level i+1 specification for component A. The level i+1 specification contains a number of components. Each component is defined as either an abstract object, an agent object or a step object.
3. Check the consistency of each level i+1 refinement
4. Repeat 2 to 3.

The rest of this paper describes two OBM constructs and illustrates the above development process with an example.

3 Abstract Object

Representation

An abstract object may have two types of operation: *compound operations* and *simple operations*. A compound operation may be refined into several operations which may themselves be compound operations. In the rest of this paper the term operation denotes simple operation and the term *coperation* denotes both compound operation and operation. An abstract object provides coperations and may require coperations provided by other components. The definition of an abstract object is divided into two parts: the interface part and the body part.

The **interface part** consists of:

- a list of provided coperations
- a list of required coperations
- an operation pattern defined in terms of the provided coperations.

The provided coperations are further divided into three categories: supply, receive and action.

supply: supply artifacts to other components, e.g. send scheduled tasks. Supply operations are usually active operations.
receive: receive artifacts from other components, e.g. receive progress reports. Receive operations are usually passive operations.
action: perform activities, e.g. schedule and monitor. Action operations may be either active or passive.

Supply coperations are matched with receive coperations, for example, to send an artifact d from component P to component Q, P needs to have a supply coperation $dtoQ$ and Q needs to have a receive coperation $dfrP$. The artifact d is passed from P to Q after $dtoQ$ has completed the call to $dfrP$. Each supply coperation may be matched with a number of receive coperations.

Operation patterns are defined using *ordering expressions*. Section 6.6 describes a subset of ordering expressions. The full definition can be found in [23, 22]. Operation patterns are defined in terms of coperation names.

The **body part** consists of

- a *call template* for each required coperation
- a definition for each provided coperation, which contains
  - the type of this coperation, i.e. active or passive
  - the order of the operation calls made by this coperation
  - the pre- and the post-conditions of the coperation

Each call template is in the form

!*name*	
prop	*func*
obj	*another*
apar	*v*

where *name* is the name of the coperation being called; by our convention, the name of the call template is always the name of the called coperation prefixed with "!"; "prop" defines the type of the call, e.g. a functional call sends a call request and receives a result, a procedural call sends a call request and does not expect any result; "obj" states the name of the component providing the required coperation and "apar" specifies the actual parameter.

The order of the operation calls is defined using ordering expressions.

The pre- and the post-conditions of a compound operation are defined in terms of the parameter, the result and the values received from the calls made to the required coperations.

Semantics

The execution of an abstract object is determined by its operation pattern. The following six types of events may be used to define the behaviour of an abstract object.

- *start*: start the execution of an active coperation.
- *end*: finish the execution of an active coperation.
- *accept*: accept a call request and start the execution of a passive coperation.
- *return*: return the result of a passive coperation to the caller and finish its execution.
- *send*: send a request to a component to call one of its coperations.
- *receive*: receive the result returned from the called coperation.

Call Templates: Each functional call template specifies a set of sequences. Each sequence contains a *send* event, followed by a number of idle steps, followed by a *receive* event. Each procedural call template specifies a set of sequences. Each sequence contains a *send* event and followed by a number of idle steps.

The temporal semantics of call templates are described in sub-section 6.5.

Order of Calls: The ordering expression defining the order of the operation calls made by a coperation specifies a set of sequences. Each sequence is concatenated from the sequences specified by the call templates of this coperation.

The temporal semantics of ordering expression are defined in sub-section 6.6.

Coperations: Each coperation specifies a set of sequences. Each sequence of an active coperation contains a *start* event, followed by a sequence determined by the order of the operation calls, followed by a *finish* event. The pre-condition of the coperation must be true in the step that the *start* event is taking place, and the post-condition must be true in the step immediately after the *finish* event. The sequences of passive coperations are similar to those of active ones with the exception that the *start* and *finish* events are replaced by the *accept* and *return* events respectively.

The temporal semantics of coperations are defined in sub-section 6.4.

Operation Pattern: An operation pattern specifies a set of sequences. Each sequence is a combination of the sequences specified by its coperations. The ordering of the combination must conform to the operation pattern.

The temporal semantics of operation patterns are defined in sub-section 6.6.

Abstract Object: An abstract object specifies a set of sequences. Each sequence is specified by its operation pattern.

The temporal semantics of abstract objects are given in section 6.2.

4 Refinement

The level i+1 refinement of a level i abstract object consists of the following:

Component Refinement

Replacing the level i abstract object by a number of level i+1 components.

Each component may be either an abstract object, an agent object or a step object.

Coperation Refinement

Replacing the sub-patterns of the level i operation pattern by operation patterns defined in terms of level i+1 coperations. A single coperation can also be regarded as a sub-operation pattern.

The consistency checking of refinements is described in section 7.

5 Boxed Object

Given a level i+1 refinement of a level i abstract object, in order to justify the refinement, it is necessary to show that the composition of the sub-components is consistent with the abstract object. A boxed object is used to compose the sub-components.

Internal Coperations: Given a level i+1 refinement of a level i component, a level i+1 coperation is *internal* if it is not used in the **Coperation Refinement** part.

Interleavings: In OBM, parallelism is defined by interleaving the events of different components. As mentioned in section 3, each specified sequence of a component contains idle steps as well as events. Each idle step can be viewed as a slot for another component to perform an event.

Synchronisations: The executions of two components are synchronised if one is performing a *send* (or *receive*) event and the other is performing the corresponding *accept* (or *return*) event.

Representation

A boxed object contains a number of components and a list of internal coperations.

Semantics

A boxed object specifies a set of sequences. Each sequence is merged from the sequences of its sub-components according to the following two constraints:

- there is at most one sub-component performing a non-synchronising event at any time;
- if a sub-component is performing a synchronising event, the corresponding sub-component must be performing the matching event at the same time.

In the merged sequence, internal events are replaced with idle steps. All synchronising events are internal, they become idle steps in the merged sequence. Therefore each step of the merged sequence contains at most one (non-internal) event.

The temporal semantics of boxed objects are defined in section 6.3.

6 Temporal Semantics

Long before temporal logic was used to reason about programs, there was substantial work on temporal logic in philosophy for reasoning about time, e.g. [19, 20]. The application of temporal logic in computer science was first introduced in [6]. A logic framework for reasoning about properties of concurrent programs was later developed by Pnueli [18]. Although that framework provided very powerful specification and verification tools for global programs, it did not provide much support for decomposition and composition of components. In [4], a compositional temporal logic proof system for specification and verification of concurrent program was introduced. This approach offered support for *development* of concurrent systems. In this paper, we will use the temporal logic described in [4] to define the semantics of OBM.

6.1 Temporal Operators

The following temporal operators are used in the semantic definitions in this paper. Their intuitive meanings are described below:

fin: end of the sequence.
$\bigcirc P$: P is true at the next moment.
$\Box P$: P is true always.
$P\ \mathcal{C}\ Q$: P is true, and then Q is true.
$P\ \mathcal{C}^*\ Q$: P is true for a number of times, and then Q is true.
$P\ \mathcal{U}\ Q$: P is true until Q is true.

6.2 Temporal Semantics of Abstract Objects

The semantics of an abstract object is determined by the semantics of its operation pattern with the constraint that at most one event may take place at any time. This constraint is to ensure that parallel activities are interleaved. Let

- Abs denote the name of an abstract object;
- X denote the context in which the semantics of Abs is defined; a context of a component contains all the information required to define its semantics, for instance, the component name, names of all the provided coperation, names of all the required coperations and names of all the variables;
- $op.pattern$ denote the operation pattern of Abs;
- $atmostone(X)$ denote the formula stating that at most one event may take place;
- $[\![Abs]\!]_X$ denote the semantics of Abs in context X;

The semantics of Abs is

$$[\![Abs]\!]_X \triangleq [\![op.pattern]\!]_X \ \wedge\ \Box atmostone(X)$$

where $op.pattern$ is defined using an ordering expression. The semantics of a subset of the ordering expression is defined in section 6.6. Due to lack of space, $atmostone(X)$ is not defined in this paper. Its definition can be found in [22].

6.3 Temporal Semantics of Boxed Objects

Let

- $Bobj$ denote the name of a boxed object containing two sub-components $C1$ and $C2$;
- W, $W1$ and $W2$ denote the contexts of $Bobj$, $C1$ and $C2$ respectively;
- $Iops$ denote the internal coperations of $Bobj$;
- $interleave(W, W1, W2)$ denote the temporal formula stating that events from $W1$ and $W2$ are interleaved;
- $synchronise(W1, W2)$ denote the temporal formula stating that there exists an event in $W1$ which is synchronised with an event in $W2$;
- $hide(\{C1, C2\}, Iops)$ denote the temporal formula stating that all the events made by coperations in $Iops$ are hidden.

The semantics of $Bobj$ is

$$\begin{array}{l}
[\![Bobj]\!]_W \triangleq \\
([\![C1]\!]_{W1}\ \mathcal{C}\ (idle(W1)\ \mathcal{U}\ fin)) \wedge ([\![C2]\!]_{W2}\ \mathcal{C}\ (idle(W2)\ \mathcal{U}\ fin)) \wedge \\
\Box(interleave(W, W1, W2)\ \vee\ synchronise(W1, W2))\ \wedge \\
\Box hide(\{C1, C2\}, Iops)
\end{array}$$

In the above formula, $C1$ and $C2$ are extended with idle steps so that their sequences have the same length. The definitions of *idle*, *interleave*, *synchronise* and *hide* are given in [22].

6.4 Temporal Semantics of Coperations

The semantics of the different types of coperations are slightly different. To save space, only the semantics of passive compound operations are defined here. Let

- cop denote a passive compound operation;
- Abs denote the name of the abstract object providing cop;
- X denote the context in which the semantics of Abs is defined;
- $callorder$ denote the ordering expression defining the order of the calls made by cop;
- $precond$ and $postcond$ denote the pre- and the post-conditions of cop;

The semantics is

$$\begin{array}{l}
[\![cop]\!]_X \triangleq \\
accepting(X, cop, precond)\ \mathcal{C}\ [\![callorder]\!]_X\ \mathcal{C}\ returning(X, cop, postcond)
\end{array}$$

cop is defined as an *accept* event, followed by a sequence of operation calls, followed by a *return* event.

$$\begin{array}{l}
accepting(X, cop, precond) \triangleq \\
idle(X)\ \mathcal{U} \\
(\pi_{abs}\ \wedge\ acc\text{-}abs\text{-}cop\ \wedge \\
reset.history(X)\ \wedge \\
precond(fpar\text{-}abs\text{-}cop)\ \wedge\ fin)
\end{array}$$

Due to lack of space we shall not explain all the details of the formula. Briefly speaking, the *accept* event is defined as a number of idle steps following by a step in which some flags and variables are set to indicate the event and the pre-condition must be true. Note how the pre-condition is embedded in the temporal formula in the definitions of the *accept* event. The *return* event is defined similarly.

6.5 Temporal Semantics of Call Templates

Due to lack of space, only the semantics of functional call templates are defined here. Let

- $op1$ denote the coperation being called by cop;
- $O1$ denote the name of the component providing $op1$;

Given the call template,

$!op1$	
prop	$func$
obj	$O1$
apar	v

The semantics is

$$
\begin{array}{l}
[\![!op1]\!]_X \triangleq \\
sending(X, O1, op1, v)\ \mathcal{C}\ receiving(X, O1, op1) \\
\\
sending(X, O1, op1, v) \triangleq \\
(idle(X)\ \wedge\ \exists n \cdot occno\text{-}O1\text{-}op1 = n)\ \mathcal{U} \\
(\pi_{abs}\ \wedge\ send\text{-}O1\text{-}op1\ \wedge \\
apar\text{-}O1\text{-}op1(n) = v)\ \wedge\ fin)
\end{array}
$$

Briefly speaking, the *send* event may be preceded by a number of idle steps. In the *send* event, the parameter of this call is put in the variable $apar\text{-}O1\text{-}op1$. When the executions of the calling component and the called component are synchronised on the *send* event and the corresponding *accept* event of $op1$, the value of $apar\text{-}O1\text{-}op1$ is passed on to the formal parameter of $op1$. This property is defined by the predicate *synchronise* in [22]. The definition of the *receive* event is similar.

6.6 Temporal Semantics of Ordering Expression

In this section, only a subset of the ordering expression is defined. The full definition can be found in [22].

$$[\![\circ]\!]_C \triangleq fin$$

$\circ$ indicates the end.

$$[\![P1; P2]\!]_C \triangleq [\![P1]\!]_C\ \mathcal{C}\ [\![P2]\!]_C$$

$P; Q$ means that P is followed by Q.

$$[\![P1 \mid\mid P2]\!]_C \triangleq$$
$$([\![P1]\!]_{C1}\, \mathcal{C}\, (idle(C1)\, \mathcal{U}\, fin)) \wedge ([\![P2]\!]_{C2}\, \mathcal{C}\, (idle(C2)\, \mathcal{U}\, fin))$$

$P1 \mid\mid P2$ means that $P1$ is in parallel with $P2$.

$$[\![P^+; Q]\!]_C \triangleq [\![P]\!]_C\, \mathcal{C}^*\, [\![Q]\!]_C$$

$P^+; Q$ means that there is one or more occurrence of P before Q happens.

$$[\![[P]; Q]\!]_C \triangleq ([\![P]\!]_C\, \mathcal{C}\, [\![Q]\!]_C) \oplus [\![Q]\!]_C$$

$[P]; Q$ means that P may happen before Q. $\oplus$ denotes the exclusive "or" operator.

7 Consistency of Refinement

Given a level i abstract object A and its level i+1 refinement, the consistency of the level i+1 refinement is validated by the following stages:

1. Identify the internal coperations.
   Let *Iops* denote the set of internal coperations.
2. Define the boxed object of the level i+1 refinement.

   Boxed Object B
   contains
   $r1, r2, ..., rn$
   hidden coperations
   Iops
3. Replace the operation pattern of A with level i+1 coperations.
   Let A' denote the same specification as A except that the operation pattern is replaced according to the operation patterns as defined in the **Coperation Refinement** part.
4. Hide calls made to internal coperations.
   After replacing the coperations of A with level i+1 coperations, A' may then contain calls made to the internal coperations. Let $hidecall(Iops, \{r1, r2, ..., rn\})$ denote the formula stating that the internal calls are hidden.
5. Extend the variables of A' so that A' and B contain the same variables.
   Let *vars* denote the set of variables in $r1, r2, ..., rn$, but not in A'; $anyvalue(vars)$ denote the formula stating that variables in *vars* can have any value.

Let Ca and Cb denote the contexts of A' and B respectively; we define that the level i+1 refinement is consistent with A if

$$[\![B]\!]_{Cb} \Rightarrow ([\![A']\!]_{Ca} \wedge \Box hidecalls(Iops, \{r1, r2, ..., rn\}) \wedge \Box anyvalue(vars))$$

Due to lack of space, *hidecalls* and *anyvalue* are not defined here. The above formula states that all the sequences specified by the left hand side are specified by the right hand side, i.e. the refinement does not violate the abstract specification.

8 Example

As mentioned before, the ISPW-6 example describes a clearly defined software process, although it may not be necessary to develop a model for this example using a stepwise refinement technique, due to the fact that it is a well known example within the software process community, in this paper, we will still use this example to illustrate how to gradually develop a model using our approach.

8.1 Brief Description of the ISPW-6 Example

A full description of the example is given in [15]. The core problem focuses on the designing, coding, unit testing, and management of a rather localized change to a software system. This is prompted by a change in requirements, and can be thought of as occurring either late in the development phase or during the support phase of the life-cycle. In order to confine the problem, it is assumed that the proposed requirements change has already been analyzed and approved by the configuration control board and only a single code unit is to be affected. The problem begins with the project manager scheduling the change and assigning the work to appropriate staff. The example problem ends when the new version of the code has successfully passed the new unit tests. During the software process, the following steps may be taken by various personnel: Schedule and Assign Tasks, Modify Design, Review Design, Modify Code, Modify Test Plans, Modify Unit Test Package, Test Unit and Monitor Progress.

8.2 Top Level

At the top level, one may view the ISPW-6 example as one component which responses to requirements issued by the configuration control board (CCB); it then repeatedly carries out some activities to produce the required changes until either that the changes are completed or a cancellation is received from CCB.

In OBM, this component can be specified as an abstract object as follows:

<u>Abstract Object</u> $ISPW6\text{-}top$ <u>The Interface Part:</u>

provides

$supply$: $none$
$receive$: $reqfrCCB, canfrCCB$
$action$: $produce$

requires

ObjName : **OpNames**
$none$: $none$

operation pattern

$$reqfrCCB; produce^{+}; [canfrCCB]; \circ$$

In the above specification, *ISPW6-top* does not provide any operation to supply artifact. In reality, the produced software should be supplied to some clients. Since this is not mentioned in [15], it is not modelled here. *ISPW6-top* provides two coperations for receiving information from CCB: *reqfrCCB* receives the requirements; *canfrCCB* receives cancellation. It also provides one action operation, *produce*, which produces the changes required.

The operation pattern specifies that first of all *ISPW6-top* is allowed to receive the requirements from CCB, and then it will repeatedly produce the required changes until either that the process is finished or a cancellation is received from CCB, and the process is then finished.

The body part of *ISPW6-top* is trivial and hence not defined here due to lack of space.

8.3 Level 1 Refinement

ISPW6-top can be refined as follows:

Component Refinement

ISPW6-top is replaced by two level 1 components: *MTeam* and *TTeam*. *MTeam* is responsible for management issues and *TTeam* is responsible for technical issues. *MTeam* and *TTeam* are defined as follows:

Abstract Object *MTeam* The Interface Part:

provides

$$\begin{array}{ll} supply & : tasktoTTeam_1 \\ receive & : reqfrCCB_1, canfrCCB_1, repfrTTeam_1 \\ action & : schedule_1, monitor_1 \end{array}$$

requires

$$\begin{array}{ll} \mathbf{ObjName} & : \mathbf{OpNames} \\ TTeam & : taskfrMTeam_1 \end{array}$$

operation pattern

$$\begin{array}{l} reqfrCCB_1; schedule_1; tasktoTTeam_1; \\ ((schedule_1; tasktoTTeam_1) \oplus monitor_1 \oplus repfrTTeam_1)^+; \\ [canfrCCB_1]; \circ \end{array}$$

where the subscripts are used to indicate the level numbers. Subscripts may be omitted if the level numbers are obvious. The operations are used for the following purposes:

$tasktoTTeam_1$**:** supply the scheduled tasks to the technical team.
$reqfrCCB_1$**:** receive the requirements from CCB.
$canfrCCB_1$,**:** receive the cancellation from CCB.
$repfrTTeam_1$**:** receive the progress report from the technical team.
$schedule_1$**:** schedule and assign tasks.
$monitor_1$**:** monitor the progress.

The operation pattern of *MTeam* states that after receiving the requirements from CCB, *MTeam* is allowed to do some scheduling and send the scheduled task to *TTeam*. After that *MTeam* is allowed to repeatedly perform its main process until either that it is finished or a cancellation is received from CCB. The main process of *MTeam* is either performing more scheduling or monitoring the progress or receiving report from *TTeam*.

Abstract Object *MTeam* The Body Part:

call templates

$!taskfrMTeam_1$	
prop	$func$
obj	$TTeam$
apar	t

coperation definitions

$tasktoTTeam_1$ **is** active

$tasktoTTeam_1$**'s call order**

$!taskfrMTeam_1; \circ$

$tasktoTTeam_1$**'s pre- and post-conditions**

trivial

———————————-

In the above, the body part of *MTeam* contains one call template. The name of the call template is !*taskfrMTeam*. Its type is a function call. The called object is *TTeam*. The parameter of the call is t.

Due to lack of space, only one coperation, *tasktoTTeam*, is defined. It is an active coperation; the order of the call is simply !*taskfrMTeam* because this is the only operation call; the pre- and the post-conditions are trivial.

Abstract Object *TTeam* The Interface Part:

provides

$supply : reptoMTeam_1$
$receive : taskfrMTeam_1$
$action : workonchange_1$

requires

$MTeam : repfrTTeam_1$

operation pattern

$$taskfrMTeam_1; workonchange_1; (workonchange_1 \oplus reptoMTeam_1)^+; \circ$$

In the above definition, the coperations are used for the following purposes:

$reptoMTeam_1$**:** supply the progress report to *MTeam*.
$taskfrMTeam_1$**:** receive the scheduled task from *MTeam*.
$workonchange_1$**:** work on the required changes.

The operation pattern of *TTeam* states that after receiving an assigned task from *MTeam*, *TTeam* is allowed to do some work on the required changes, and subsequently, it is allowed to either perform more work or report its progress to *MTeam* repeatedly.

The body part is trivial, and not included here due to lack of space.

Coperation Refinement

The following sub-operation patterns of *ISPW6-top* are replaced with the level 1 coperations as follows:

$$reqfrCCB \triangleq reqfrCCB_1$$

$$canfrCCB \triangleq canfrCCB_1$$

$$\begin{array}{l} produce^+;[canfrCCB];\circ \triangleq \\ schedule_1; \\ (((schedule_1 \oplus monitor_1)^+;[canfrCCB_1];\circ) \mid\mid (workonchange_1^+;\circ)) \end{array}$$

8.4 Consistency of the Level 1 Refinement

1. Let *intops* denote the set of internal operations, *intops* contains

$$\{tasktoTTeam_1, taskfrMTeam_1, reptoMTeam_1, repfrTTeam_1\}$$

2. Let *ISPW6-level1* denote the boxed object containing *MTeam* and *TTeam*,

Boxed Object *ISPW6-level1*
contains
MTeam, *TTeam*
hidden coperations
intops

3. Let *ISPW6-top'* denote the same specification as *ISPW6-top* except that the sub-operation patterns are replaced according to the level 1 **Coperation Refinement** part. The resulting operation pattern of *ISPW6-top'* is

$$\begin{array}{l} reqfrCCB_1; schedule_1; \\ (((schedule_1 \oplus monitor_1)^+;[canfrCCB_1];\circ) \mid\mid (workonchange_1^+;\circ)) \end{array}$$

4. Let $hidecalls(intops, \{MTeam, TTeam\})$ denote the formula for hiding the internal calls.
5. Let $anyvalue(vars)$ denote the formula for extending the variables of *ISPW6-top'*.

Let $C1$ and $C0$ denote the contexts of *ISPW6-level1* and *ISPW6-top'* respectively, according to section 7, the aim is to prove

$$\begin{array}{l} [\![ISPW6\text{-}level1]\!]_{C1} \Rightarrow \\ ([\![ISPW6\text{-}top']\!]_{C0} \ \wedge \\ \Box hidecalls(intops, \{MTeam, TTeam\}) \ \wedge \\ \Box anyvalue(vars)) \end{array}$$

Let Cm and Ct denote the contexts of $MTeam$ and $TTeam$ respectively; according to section 6.3, the semantics of the boxed object $ISPW6\text{-}level1$ can be expanded as follows:

$$
\begin{array}{l}
\llbracket ISPW6\text{-}level1 \rrbracket_{C1} \triangleq \\
(\llbracket MTeam \rrbracket_{Cm}\ \mathcal{C}\ (idle(Cm)\ \mathcal{U}\ fin))\ \wedge \\
(\llbracket TTeam \rrbracket_{Ct}\ \mathcal{C}\ (idle(Ct)\ \mathcal{U}\ fin))\ \wedge \\
\Box(interleave(C1, Cm, Ct)\ \vee\ synchronise(Cm, Ct))\ \wedge \\
\Box hide(\{MTeam,\ TTeam\}, intops)
\end{array}
$$

Expand the semantic definitions of the abstract objects $MTeam$ and $TTeam$ according to the definition in section 6.2, the above formula becomes:

$$
\begin{array}{l}
\llbracket ISPW6\text{-}level1 \rrbracket_{C1} \triangleq \\
((\llbracket MTeam.pattern \rrbracket_{Cm} \wedge \Box atmostone(Cm))\ \mathcal{C}\ (idle(Cm)\ \mathcal{U}\ fin))\ \wedge \\
((\llbracket TTeam.pattern \rrbracket_{Ct} \wedge \Box atmostone(Ct))\ \mathcal{C}\ (idle(Ct)\ \mathcal{U}\ fin))\ \wedge \\
\Box(interleave(C1, Cm, Ct)\ \vee\ synchronise(Cm, Ct))\ \wedge \\
\Box hide(\{MTeam,\ TTeam\}, intops)
\end{array}
$$

We then further expand the semantics of the operation patterns according to the definitions in section 6.6 to give:

$$
\begin{array}{l}
\llbracket ISPW6\text{-}level1 \rrbracket_{C1} \triangleq \\
(((\llbracket reqfrCCB_1 \rrbracket_{Cm}\ \mathcal{C}\ \llbracket schedule_1 \rrbracket_{Cm}\ \mathcal{C}\ \llbracket tasktoTTeam_1 \rrbracket_{Cm}\ \mathcal{C} \\
((\llbracket schedule_1 \rrbracket_{Cm}\ \mathcal{C}\ \llbracket tasktoTTeam_1 \rrbracket_{Cm}\ \oplus\ \llbracket monitor_1 \rrbracket_{Cm}\ \oplus \\
\llbracket repfrTTeam_1 \rrbracket)\ \mathcal{C}^* \\
(fin\ \oplus\ \llbracket canfrCCB_1 \rrbracket_{Cm}\ \mathcal{C}\ fin)))\ \wedge \\
\Box atmostone(Cm))\ \mathcal{C}\ (idle(Cm)\ \mathcal{U}\ fin)) \\
\wedge \\
(((\llbracket taskfrMTeam_1 \rrbracket_{Ct}\ \mathcal{C}\ \llbracket workonchange_1 \rrbracket_{Ct}\ \mathcal{C} \\
(\llbracket workonchange_1 \rrbracket_{Ct}\ \oplus\ \llbracket reptoMTeam_1 \rrbracket_{Ct})\ \mathcal{C}^*\ fin)\ \wedge \\
\Box atmostone(Ct))\ \mathcal{C}\ (idle(Ct)\ \mathcal{U}\ fin)) \\
\wedge \\
\Box(interleave(C1, Cm, Ct)\ \vee\ synchronise(Cm, Ct))\ \wedge \\
\Box hide(\{MTeam,\ TTeam\}, intops)
\end{array}
$$

Each sequence specified by the above formula is merged from the following two:

MTeam: $reqfrCCB_1$ followed by $schedule_1$ followed by $tasktoTTeam$ followed by an iteration of either $schedule_1$ followed by $tasktoTTeam$ or $monitor_1$ or $repfrTTeam$ until either $canfrCCB_1$ or the end;

TTeam: $taskfrMTeam$ followed by $workonchange_1$ followed by an iteration of either $workonchange_1$ or $reptoMTeam$;

with the constraints that

1. the events between the two sets are either interleaved or synchronised,
2. all events of the internal coperations are hidden.

Since $taskto TTeam_1$ and $taskfrMTeam_1$ are synchronised coperations, it means that TTeam cannot start its execution until MTeam has done $reqfrCCB_1$ and the first $schedule_1$, see the illustration below:

MTeam: $reqfrCCB_1$, $schedule_1$ $taskto TTeam_1$, ...
TTeam: $taskfrMTeam_1$, $workonchange_1$,...

In addition, according to the second constraint, events of $taskto TTeam_1$, $taskfrMTeam_1$, $repfrTTeam_1$ and $reptoMTeam_1$ are hidden because they are internal coperations. Therefore each sequence specified by the above formula is as follows:

ISPW6-level1: $reqfrCCB_1$ followed by $schedule_1$ followed by a sequence merged from the following two:

1. an iteration of either $schedule_1$ or $monitor_1$ until either $canfrCCB_1$ or the end;
2. $workonchange_1$ followed by an iteration of $workonchange_1$.

Now consider the right hand side of the proof, if we expand $[\![ISPW6\text{-}top']\!]$ according to the semantic definition of the abstract objects, we get the following formula,

$$
\begin{array}{l}
[\![reqfrCCB_1; schedule_1;\\
(((schedule_1 \ \oplus \ monitor_1)^+; [canfrCCB_1]; \circ) \ ||\\
(workonchange_1^+; \circ))]\!]_{C0} \ \wedge\\
\Box atmostone(C0) \ \wedge\\
\Box hidecalls(intops, \{MTeam, TTeam\}) \ \wedge\\
\Box anyvalue(vars))
\end{array}
$$

Again expanding the semantics of the operation pattern, we get,

$$
\begin{array}{l}
[\![reqfrCCB_1]\!]_{C0} \ \mathcal{C} \ [\![schedule_1]\!]_{C0} \ \mathcal{C}\\
(((([\![schedule_1]\!]_{C0} \ \oplus \ [\![monitor_1]\!]_{C0}) \ \mathcal{C}^*\\
(fin \ \oplus \ [\![canfrCCB_1]\!]_{C0} \ \mathcal{C} \ fin)) \ \mathcal{C} \ (idle(C0) \ \mathcal{U} \ fin))\\
\wedge\\
([\![workonchange_1]\!]_{Ct} \ \mathcal{C}^* \ fin \ \mathcal{C} \ (idle(Ct) \ \mathcal{U} \ fin))) \ \wedge\\
\Box atmostone(C0) \ \wedge\\
\Box hidecalls(intops, \{MTeam, TTeam\}) \ \wedge\\
\Box anyvalue(vars))
\end{array}
$$

Each sequence specified by the above formula contains a $reqfrCCB_1$ followed by a $schedule_1$ followed by a sequence merged from the following two:

1. an iteration of either $schedule_1$ or $monitor_1$ until either $canfrCCB_1$ or the end;
2. an iteration of $workonchange_1$.

It is not difficult to see that the behaviour specified by $ISPW6\text{-}level1$ satisfies the behaviour specified by $ISPW6\text{-}top'$.

9 Discussion, Related Work and Future Work

9.1 Information Flow

In many existing process modelling approaches, activities are modelled as operations. Inputs and outputs of activities are treated as parameters and results of the operations. However we found that problems exist with this type of modelling if the results of an operation are not needed by the caller of this operation, but by some others, and similarly if the inputs of an operation are not supplied by the caller of this activity. In our approach, there is a clear distinction between operations regarding information flow and operations regarding activities. Information flow is explicitly modelled with the supply and receive operations. We believe this gives better control to where the information is going and when the information may be transferred.

9.2 Multiple Levels of Abstractions

The need for modelling processes in multiple levels of abstraction has been clearly recognised. Dowson pointed out in [9] that modelling formalism should accommodate a wide range of model granularity, and allow the refinement of initially large-grain models to address increasing details. The example above has shown that OBM has achieved this requirement.

In OBM, multiple levels of abstraction can be reflected in two ways. One is that a model is refined with increasing details as illustrated by the example above. The other is that different parts of the model may be represented at different levels of abstraction. For example, one may decide to refine $TTeam$, but not $MTeam$, in the next level of refinement.

Several existing approaches also provide means to represent a model in multiple levels of abstraction. Some of these are based on hybrid formalisms that use different notations for large-grain and small-grain aspects of process. An example of this type of approach is Process Weaver [11]. Another type of approach is incremental definitions. Examples of this type of approach include: tasks in Epos [8], activities in Slang [3], blackboard in Oikos [2]. However there is no support for formal reasoning between different levels of abstraction. Since Slang is based on Petri-net, it should be possible to perform formal reasoning, however, it is not clear to the authors whether such a facility is provided by [3]. Work on consistency proof for refinement in Oikos is in development [2, 7].

In [16], a method based on data flow diagram (DFD) for modelling process interface is defined. Although the method has a formal basis for process decomposition, it provides very limited expressive power.

9.3 Alternative Refinement

In OBM, it is possible to have different refinements for the same abstract object. For instance, in the above example, one possible refinement for $TTeam$ is to decompose it into two components: Design Team and Implementation Team, where Design Team is responsible for modify design, review design etc. and the Implementation Team is responsible for modify code and testing. Alternatively it can be decompose into three components: Design, Coding and Testing. The consistency of both refinements can be checked against $TTeam$.

This facility allows us to refine sub-processes according to the different needs. For instance, for different projects, in which there will be different resources and different requirements, it is possible to refine a part of a process to suit the specific need without affecting the behaviour of the overall process. This is particularly useful in process modification.

9.4 Verification and Validation

In [12], the concepts of verification of a software process model and validation of a software process model are defined. Verification means to prove the correctness of a model. Validation means to test if a model behaves as expected using simulation. If a model can be verified to have a certain property, it means that all processes of that model have that property. In contrast, validation only tests individual process, and in general, it is not possible to test all processes.

Ideally, one would like to verify a model. However, verifications can be complicated, and sometimes theoretically impossible. Although validation does not test all the possibilities, it does give valuable feedbacks so that confidence in the model can be built up.

We are aiming to support both verification and validation facilities so that they can be used in a complementary way.

9.5 BM Stepper

Currently we are working with the Southampton University to develop a simulation tool, called BM stepper, for stepping through the operation patterns. The BM stepper is implemented using the object-oriented proto-typing language *Enact* [13]. Our experience has shown that the BM stepper has proved to be useful with identifying errors and gaining confidences [21].

At the moment, the BM stepper only covers a sub-set of the operators used in OBM operation patterns. More work will be done to extend the BM stepper to include all the operators.

9.6 Tool Support

OBM was first used in a case study reported in [25]. The details of the proofs for the example described above and the one in [25] are very similar. In fact, for most part of the proof, it is only necessary to substitute the names. This similarity suggests that tools may be built to assist the proofs.

9.7 Executable Specifications

The BM stepper allows us to simulate simple operation patterns. Investigation on supporting executable OBM with *Enact* is underway. This involves defining the semantics of the OBM constructs in *Enact*. A case study, called "Integrated Store Solution" (ISS400) [1], is being undertaken to illustrate both the development process of OBM and the use of the executable OBM.

9.8 Data Refinement

In this paper, we have described component refinement and coperation refinement. What remains to be studied is data refinement. One aspect of our future work is to investigate how to reason about the consistency between the refinement and its abstract object if data refinement is included. A possible solution is to use the data reification technique as described in [14].

The work on OBM is ongoing. There is still a wide scope for improvement. However, we believe the concepts and features offered by OBM can play an important part in the area of process modelling.

Acknowledgements

The authors would like to thank Professor P. Henderson for the detailed discussions and suggestions on the ideas for the executable OBM, Mark Greenwood for his comments on the earlier drafts and all members of IPG for the useful discussions through out this work.

References

1. ISS400 Technical Introduction. Technical Report 53423/001, Internationl Computer Limited, June 1993.
2. V. Ambriola and Carlo Montangero. Oikos at the Age of Three. In *Proceedings EWSPT'92*, volume 635 of *Lecture Notes in Computer Science*. Springer Verlag, 1992.
3. S. Bandinelli, A. Fuggetta, and S. Grigolli. Sprocess Modelling In-the-Large with SLANG. In *Proceedings of the Second International Conference on the Software Process*, Berlin, 1993.
4. H. Barringer, R. Kuiper, and A. Pnueli. Now you may compose temporal logic specifications. In *Proceedings of the 16th A.C.M. Symposium on Theory of Computing*, 1984.
5. R.F. Bruynoghe, J.M. Parker, and J.S. Rowles. PSS: A system for Process Enactment. In *Proceedings of the first International Conference on the Software Process, Manufacturing Complex Systems.* IEEE Computer Society Press, 1991.

6. R.M. Burstall. Program Proving as Hand Simulation with a Little Induction. In *Information Processing 74*, Pages 308-312, North-Holland Pub. Co., 1974.
7. X.J. Chen and C. Montangero. Compositional Refinement in Multiple Blackboard Systems. In *ESOP'92 European Symposium on Programming*, February.
8. R. Conradi et al. Design, Use and Implementation of SPELL, a Language for Software Process Modeling and Evolution. In *Proceedings EWSPT'92*, volume 635 of *Lecture Notes in Computer Science*. Springer Verlag, 1992.
9. Mark Dowson. Software Process Themes and Issues. In *Proceedings of the Second International Conference on the Software Process*. IEEE Computer Society Press, 1993.
10. J. C. Derniame (Editor). *Lecture Notes in Computer Science 635: Software Process Technology*. Springer-Verlag, 1992.
11. C. Fernström. PROCESS WEAVER: Adding Process Support to UNIX. In *Proceedings of the Second International Conference on the Software Process*, Berlin, 1993.
12. V. Gruhn and A. Saalmann. Software Process Validation Based on FUNSOFT Net. In *Proceedings EWSPT'92*, volume 635 of *Lecture Notes in Computer Science*. Springer Verlag, 1992.
13. P. Henderson. *Object-Oriented Specification and Design with C++*. International Series in Software Engineering. McGraw-Hill, 1993.
14. C.B. Jones. *Systematic Software Development Using VDM*. Computer Science. Prentice-Hall International, 1986.
15. M. Kellner, P. Feiler, A. Finkelstein, T. Katayama, L. Osterweil, M. Penedo, and H.D.Rombach. ISPW-6 Software Process Example. In *Proceedings of the First International Conference on Software Process*, Washington, DC, 1991. IEEE Computer Society Press.
16. C. Kung. Process Interface Modelling and Consistency Checking. *Journal of System and Software*, 15:185-191, 1991.
17. L. Lamport. What Good is Temporal Logic? In R.E.A. Mason, editor, *Information Processing 83*, pages 657-668, IFIP, 1983.
18. A. Pnueli. The Temporal Semantics of Concurrent Programs. *Theoretical Computer Science*, 13:45–60, 1981.
19. A.N. Prior. Time and Modality. Oxford University Press, 1957.
20. N. Rescher and A. Urquhart. Temporal Logic. Springer-Verlag Wien New York 1971.
21. I. Robertson J. Sa R.A. Snowdon R.F. Bruynooghe, R.M. Greenwood and B.C. Warboys. *Towards a Total Process Modelling System: A Case Study Using ISPW-6*. A Chapter in "Advances in Software Process Technology", to be published by Research Studies Press, J. Wiley, In Preparation, 1993.
22. J. Sa. OBM: A Tutorial. Project Report, March 1993.
23. J. Sa and B.C. Warboys. Specifying Concurrent Object-based Systems using Combined Specification Notations. Technical Report UMCS-91-9-2, Department of Computer Science, University of Manchester, July 1991.
24. J. Sa and B.C. Warboys. Integrating a Formal Specification Method with PML: A Case Study. In *Proceedings EWSPT'92*, volume 635 of *Lecture Notes in Computer Science*, pages 106–123. Springer Verlag, 1992.
25. J. Sa, B.C. Warboys and J.A. Keane. OBM: A Specification Method for Modelling Organisational Process. *Proceeding of Workshop on Constraint Processing at CSAM'93*, St. Petersburg, July 1993.